PosAx-O: Exploring Operator-level Approximations for Posit Arithmetic in Embedded AI/ML

Amritha Immaneni, Salim Ullah, Suresh Nambi, Siva Satyendra Sahoo, Akash Kumar
Chair of Processor Design, Center for Advancing Electronics Dresden (cfaed),
Technische Universität Dresden, Dresden, Germany

***Abstract*—The quest for low-cost embedded AI/ML applications has motivated innovations across multiple abstractions of the computation stack. Novel approaches for arithmetic operations have primarily involved quantization, precision-scaling, approximations, and modified data representation. In this context, Posit has emerged as an alternative to the IEEE-754 standard as it offers multiple benefits, primarily due to its dynamic range and tapered precision. However, the implementation of Posit arithmetic operations tends to result in high resource utilization and power dissipation. Consequently, recent works have delved into the idea of exploiting the error resilience of machine learning algorithms by using low-precision Posit arithmetic. However, limiting the exploration to precision-scaling limits the scope for application-specific optimizations for embedded AI/ML applications. To this end, we explore operator-level optimizations and approximations for low-precision Posit numbers. Specifically, we identify and eliminate redundant operations in state-of-the-art Posit arithmetic operator designs and provide a modular framework for exploring approximations in various stages of the computation. We also present a novel framework for behaviorally testing the corresponding Posit approximate designs in Artificial Neural Networks. The proposed optimizations and approximations exhibit considerable resource improvements with a small error in many cases. For instance, a Posit-based multiplier with 1-bit reduced precision shows a 33% improvement in power and utilization, with only a 0.2% degradation in overall accuracy.**

***Index Terms*—Approximate Computing, Arithmetic Operator Design, Circuit Synthesis, Posit Arithmetic**

I. Introduction

As the world moves toward more prevalent automated systems, Machine Learning (ML) models are becoming ubiquitous and indispensable in a wide range of applications. However, the convenience and productivity offered by automation come at the cost of a large carbon footprint. Factors affecting the energy consumption of ML models include training time, computing infrastructure, and the type of energy used [1]. Moving toward greener systems, machine learning inference has recently shifted from servers to edge devices, opening up new advantages and challenges. To this end, the high computational complexity, memory footprint, and storage requirements of ML models are some of the main challenges in deploying them on embedded systems at the edge. Many recent works have proposed various optimization techniques, such as ML model reduction and optimization [2], low precision quantization techniques [3], and utilization of approximate arithmetic circuits [4] to enable ML inference at the edge. In this work, we have explored various optimization techniques for the recently proposed Posit number scheme to enable resource-efficient and highly accurate computations for embedded AI/ML applications.

TABLE I: Comparing range of number representations

n	es	useed	Min(scale)	Max(scale)
Posit8	4	65536	1.26E-29	7.92E+28
Posit16	3	256	1.93E-34	5.2E+33
Posit32	2	16	7.52E-37	8.3E+34
FP32	-	-	1.0E-45	3E+38
FP16	-	-	6E-08	7E+04

Posits have been shown to offer multiple benefits over the IEEE 754-2008 Floating-Point (FP) standards as they exhibit better dynamic range, better resolution with tapered accuracy, and have eliminated the ambiguity of positive and negative infinity and zero [5]. As seen in TABLE I, Posit(8,4)[1] and Posit(16,3)[2] have a comparable dynamic range to IEEE 754-2008 single-precision Floating-Point (FP32), motivating the usage of 8-bit or 16-bit Posit schemes resulting in significant memory savings. For example, Fig. 1, adapted from [6], shows the comparison of the effect of using different quantization methods (number representation schemes) across multiple performance aspects – error in the quantization of weights, Critical Path Delay (CPD) of Multiply and Accumulate (MAC) unit, and storage requirements of the weights of the Conv2_1 layer of pre-trained VGG16 network [7]. As seen in the figure, Posit number representation schemes offer a considerable reduction in the memory requirements with almost negligible loss in accuracy compared to FP32.

However, the memory and accuracy advantages of the Posit scheme incur energy and delay costs in implementing the Posit arithmetic operators. The number decoding logic for the Posit scheme is more complex than that of FP due to the variable regime length, resulting in an undesirable amount of delay, energy and total utilized resources. Additionally, as shown in Fig. 1, the CPD of the Posit MAC is much higher compared to Fixed-Point (FxP) operators of similar bit-width. Consequently, the implementation cost of Posit-based hardware could supersede the accuracy advantages they offer over more straightforward quantization schemes such as 8-bit Integer (INT8) and FxP. Therefore, it is imperative that the

This work is supported by the Deutsche Forschungsgemeinschaft (DFG) under the X-ReAp project (Project number 380524764).

[1]8-bit Posit numbers
[2]16-bit Posit numbers. Posit number scheme is summarized in Section II.

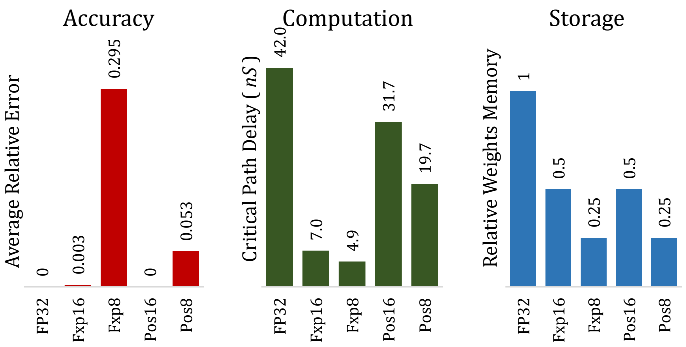

Fig. 1: Accuracy and performance comparison of various schemes for numbers representation for the Conv2_1 layer of pre-trained VGG16 [7]: Average absolute relative error concerning FP32-based parameters, critical path delay, normalized memory footprints [6].

hardware architectures for Posit arithmetic are optimized and made to be more energy- and resource-efficient.

Most of the related research in efficient architectures for Posit arithmetic operators has focused on circuit-level modifications and further parameterizing the number representation to modulate the implementation for application-specific requirements. While such an approach allows some degree of application-specific optimizations, it excludes leveraging a large body of work related to application-specific approximations in arithmetic operators. Approximations can significantly improve the power performance and area (PPA) metric of the design by introducing errors in the computation, which an application's inherent error resilience can tolerate. However, searching for the appropriate level of approximation in the arithmetic operator requires efficient Design Space Exploration (DSE). To this end, we propose a novel framework for exploring approximations in Posit-based arithmetic operators. The related contributions are listed below.

Contributions:

1) We present a modular model of Posit-based arithmetic operators that enables further exploration of optimizations and approximations in the operators' implementations. Specifically, we model the adder, multiplier, and Fused Multiply and Accumulate (FMA) operators into generic operations to allow the modular characterization of each operation to identify areas of improvement.

2) With the proposed model, we present circuit-level optimization to the *accurate* operators' implementations. Specifically, we identify and eliminate redundant operations in the design and report up to 17% and 40% reduction in power and resource utilization, respectively, compared to the state-of-the-art implementation.

3) We explore the effect of standard approximation techniques for integer arithmetic on the corresponding Posit operator implementations. Specifically, we report the results of integrating different types of approximations—generic, architecture- and quantization-specific methods in the Posit operators.

4) We present a novel PyTorch-based framework for estimating the impact of *approximate Posit operators* in Artificial Neural Networks (ANN)-based applications. The proposed framework for exploring Posit operator-level approximations for ANN applications will be available as an open-source tool at *https://Blinded_for_peer_review*.

The rest of the paper is organized as follows. Section II provides a brief overview of the Posit number system and discusses some related works for Posit arithmetic operator implementations. The proposed modeling of the Posit operators and the corresponding optimizations and exploration of approximations using the model are described in Section III and Section IV, respectively. The results from the experimentation with the proposed framework are discussed in Section V. Finally, we conclude the paper in Section VI with a summary of the proposed methods and a discussion of the scope of related future work.

II. Background and Related Works

A. Posit Number System

The structure of a Posit-based number with bit-width n and exponent width es is shown in Fig. 2. The sign bit can be 0 or 1, depending on whether the number is positive or negative, respectively. If the sign bit is 0, the Posit can be decoded as it is; else, the number has to be $2's$ complemented and then decoded. The scale factor of the Posit number is made up of two components: the *regime* and the *exponent*. The regime is a variable-length bit-sequence following the sign bit, consisting of contiguous 0s or 1s, terminated by the opposite bit. The *run-length* of the regime is equivalent to the number of 0s or 1s until the terminating bit. Depending on whether the run-length, m, consists of 0s or 1s, the regime value can be found as shown in (1) [5].

$$k = \begin{cases} -m, & r\ bits = 0 \\ m-1, & r\ bits = 1 \end{cases} \tag{1}$$

We can then decode the Posit value as:

$$\begin{gathered} sign \ \times \ useed^{k} \times 2^{exp} \ \times \ mantissa \\ useed = 2^{2^{es}} \ ; \ mantissa = 1.f_1 f_2 ... \end{gathered} \tag{2}$$

The *useed* is an important tuning parameter, and the *mantissa* consists of fraction bits with the hidden leading 1. It is important to note that, unlike in FP, Posit does not include an exponent bias. Moreover, the Posit number representation does not need to have any fraction bits at all, as the regime bits can occupy anywhere between 2 to $n-1$ bits. The variable-length regime and exponent bits give rise to what is known as *tapered precision*, where numbers closer to 1 are more accurately represented. Finally, there is no special representation for infinity, as Posit numbers do not overflow. The representation

$$\overbrace{\underbrace{s}_{1\ Bit}}^{Sign}\ \overbrace{\underbrace{r\,r\cdots r\,\bar{r}}_{Run\ Length}}^{Regime\ bits}\ \overbrace{\underbrace{e_1\,e_2\cdots e_{ES}}_{Max.\ ES-Bit}}^{Exponent\ bits,\ if\ any}\ \overbrace{\underbrace{f_1\,f_2\,\cdots\cdots.}_{Remaining-Bit,\ if\ any}}^{Fraction\ bits,\ if\ any}$$

Fig. 2: Posit number representation

for *zero* in Posits is simply all zero bits, and the other special representation of 1 followed by $n-1$ 0s represents the value $\pm\infty$, sometimes called *projective infinity* or *complex infinity* or *the point at infinity*. The $\pm\infty$ represents the reciprocal of *zero* and not necessarily any overflow in operations. These simple representations allow for a less ambiguous representation and simplify certain aspects of the decoding logic.

B. Related Works

Similar to floating-point arithmetic, a straightforward approach to implementing Posit arithmetic operators involves decoding the number representation, performing the integer arithmetic operations, and encoding the result back into the Posit number representation. Some of the more significant works in the context of the hardware implementation of Posit arithmetic operators include [8], [9], and [10]. In [8], the authors employ a three-stage process that involves Posit data extraction, core arithmetic processing, and Posit construction to perform parameterized Posit arithmetic, including multiplication and division. However, the circuit implementations proposed in [8] has two main drawbacks: firstly, the decoding logic utilizes both a Leading Zero Detector (LZD) and Leading One Detector (LOD), and secondly, a negative Posit number has to be complemented before it is decoded. While the authors in [9] eliminated the first problem, the authors in [10] obtained the best utilization and power by solving both these implementation deficiencies. For more optimized implementations, the authors in [11] propose a Posit multiplier in which the mantissa multiplier is divided into smaller width modules which can be enabled depending on the mantissa width. Their proposed design achieves an 8% reduction in power for an 8-bit Posit-based multiplier, and over 20% for Posit numbers with $n > 16$. To enable better throughput, the work in [12] proposed an algorithm for a 5-stage pipeline for the Posit FMA unit, which takes three n-bit inputs and a 3-bit control signal. However, this implementation is not fully optimized with regard to the decode stage and utilizes more resources than those reported in [8].

Few works in literature have focused on energy-aware and approximation-oriented hardware implementations for Posit arithmetic. In [13], the authors propose a Posit multiplier implementation utilizing Mitchell's approximate multiplication algorithm along with variable truncation bits and an iterative approach to reduce approximation errors. Compared to [8], their design achieves a 44% decrease in Look-Up Table (LUT) utilization for 32-bit Posit arithmetic. The work in [14] uses a logarithm-approximate multiplier as well, resulting in a 30% improvement in utilization compared to [8] and 15-20% compared to [9], as well as a 40-60% improvement in power compared to other works. Furthermore, the authors in [14] have trained different ANNs using the Posit(16,1) format for various datasets, and during inference, they replace the accurate Posit multipliers with their proposed approximate multipliers. Their experimental results show that both accurate and approximate multipliers-based inferences produce similar output accuracy. The authors in [6] proposed an application-specific energy-aware Posit to Fixed-point hardware, which reduces memory utilization by up to 46%. It also leverages the range of weights in neural networks, $[-1, 1]$, to make further optimizations to this hardware. Another kind of approximation is proposed in [15] and [16], where the Posit structure was leveraged to make approximations in activation functions. The behavioral results show that the proposed functions outperformed existing approximations such as FastSigmoid; however, these functions have not been characterized on hardware. An interesting approach to decrease the decoder power is proposed in [17], where the authors use a fixed regime length in the Posit format. Tuning the regime length allows the tuning of tapered precision. This optimization is carried out for $n = 16$ to $n = 32$ Posits, with a regime length that results in a similar dynamic range to FP32. The authors report 47% improvement in power compared to traditional Posit, and up to 70% improvement when compared to IEEE 754.

The majority of state-of-the-art Posit arithmetic operator designs focus on custom modifications to the algorithm, thereby optimizing the circuit complexity. Additionally, since the decoder deals with modules such as LZD and *shifters* which are required to be accurate, to determine the scale factor, there is limited scope for approximation in the decoder. Hence, the approximation-based methods primarily target the integer arithmetic module. However, most of the approximation methods focus on 16-bit Posit numbers where the integer module is much larger and consumes comparable power to the decoder and encoder. However, for the 8-bit Posit operations, the overall improvements seen while approximating integer modules may not be very high and hence motivate the exploration of novel approximations and optimizations. Further, state-of-the-art Posit approximations have not leveraged the various types of integer approximation methods available, primarily due to the lack of a framework for exploring the effect of such methods both at the operator- and application-level. Our current work attempts to enable such DSE by enabling the implementation of various approximations across different stages of computation and characterization of the resulting approximate designs for ANN-based applications.

III. Operator Modelling for Exploring Approximations

A. The Posit Arithmetic Flow

A general Posit arithmetic block as shown in Fig. 3 consists of a decoder, the integer arithmetic module (adder, multiplier, etc.), and an encoding block that takes care of the packaging and rounding, converting it back to a Posit.

The output of each module is an intermediate representation including parameters such as the *sign*, *scale*, *fraction*(frac), *isZero* and *isNaR* (the *point at infinity*). The intermediate representation in our current work follows the convention used in [10]. In general, a Posit operand must go through a 2's complement module before the decoder and/or after the encoder if it is a negative number. Hence, the 4 possibilities of using the 2's complement operation in a Posit addition are illustrated in Fig. 4. The *Adder* could be replaced with any

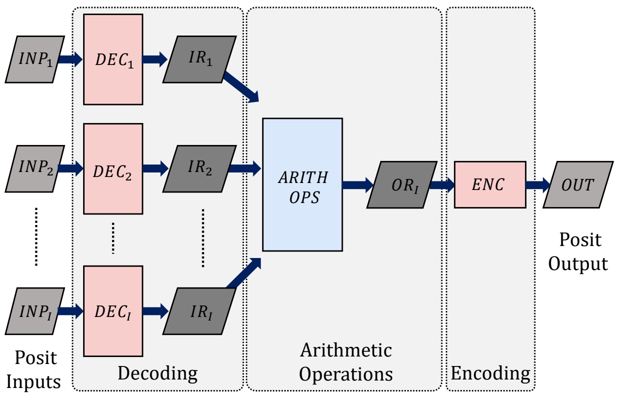

Fig. 3: Generic representation of posit arithmetic

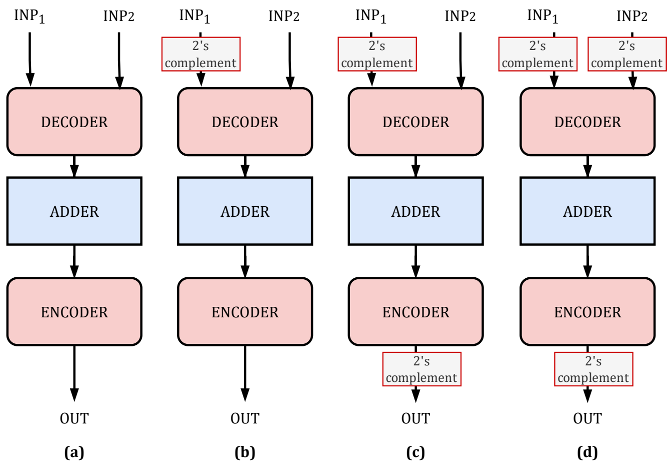

Fig. 4: The possibilities of 2's complementing

integer module for a different operator. It can be seen that for cases such as (d) the 2's complement can be repetitive and redundant, especially when we look into the finer details of 2's complementing that takes place in the decoder. The authors in [10] resolved this redundancy through a more complex but efficient implementation known as smallPosit [18]. We thus utilize *smallPosit* as a starting point for further approximations and optimizations.

B. The Posit Decoder

The main modules of the Posit decoder are shown in Fig. 5. The most components of the decoder are the LZD (in this case, Inverse LZD (ILZD)) and the *Shifter*. In [11], the authors eliminated the need for both LZD and LOD by performing an XOR operation on the bits $inp(n-2,1)$ and $inp(n-3,0)$, where inp is the input to the decoder. When the output of the XOR operation is passed through the ILZD, the resulting value is one less than the regime length. The regime length, which is the flipped output of ILZD, is an input to the Shifter. Normally, the shift operation is $inp(n-3,0) >> r\_length$, as the sign bit and flip bit must be taken into account. However, since the regime length is 1 less than what it is supposed to be, the shift operation, in this case, is $inp(n-4,0) >> r\_length$, as seen in the figure. Finally, the Shifter outputs the exponent-fraction

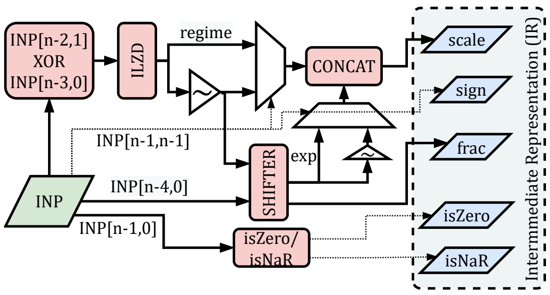

Fig. 5: The Posit Decoder

value which can be separated into the respective components. The exponent has to be 2's complemented if the number is negative. In this case, a 1's complement suffices in most cases, and the exponent is concatenated with the regime to form the scale. However, the 1's complement would bring about errors in cases where the fraction bits are 0. This problem is resolved by defining the mantissa as: $mantissa = \{sign, !sign, frac\}$. This mantissa format offsets the scale value in anomalous cases.

C. Posit Adder and Multiplier

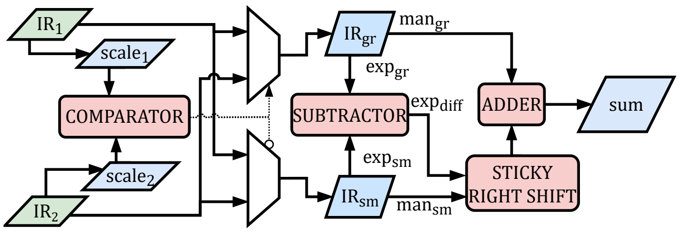

Fig. 6: The Posit Adder

The Posit Adder is quite similar to the standard floating-point adder, given the intermediate representation inputs. The main modules, shown in Fig. 6, are the *comparator* that is used to find the larger scale, *subtractor* to find the difference of the scales, *sticky right shift* module and the 2's complement *signed adder*. Post addition the required modules are the ILZD and Shifter for normalizing and the overflow unit, as shown in Fig. 7.

The Posit multiplier flow is similar to that of floating-point as well. As shown in Fig. 8, the integer multiplier unit receives the *mantissas*, and the output goes to a bias unit, which consists of various XOR operations to evaluate the added exponent bias, depending on the value of sig_{prod}. Then the exponent bias is added to the two scales to compute the final scale. Since there is a possibility of overflow or underflow as well, the scale must be compared with the maximum and minimum scales to obtain the final output.

D. The Posit Encoder

A basic schematic of the Posit encoder is shown in Fig. 9. The main modules of the encoder involve a shifter to extract the regime, another shifter for packaging the Posit, multiplexers, and finally a rounding unit that consists of AND and OR

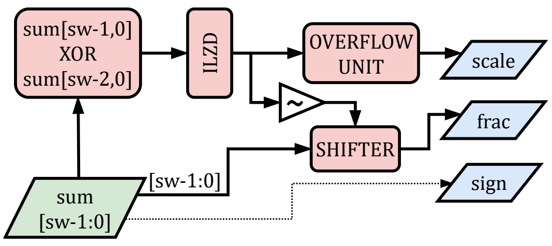

Fig. 7: Post Adder Operations

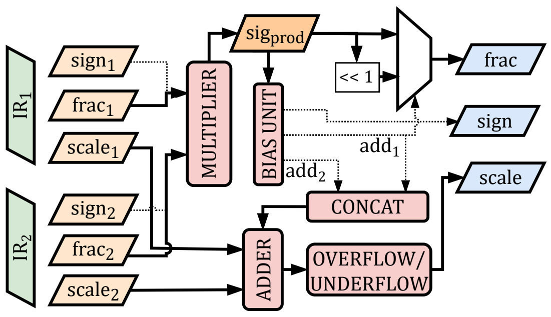

Fig. 8: Posit Multiplier

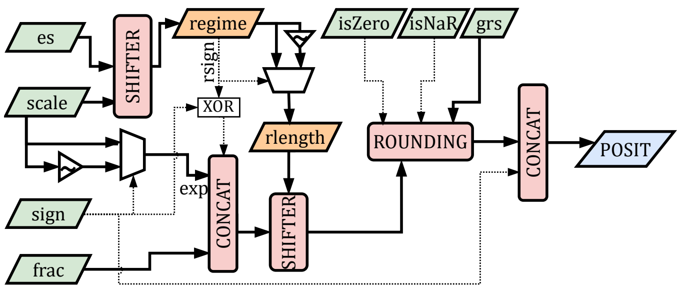

Fig. 9: Posit Encoder

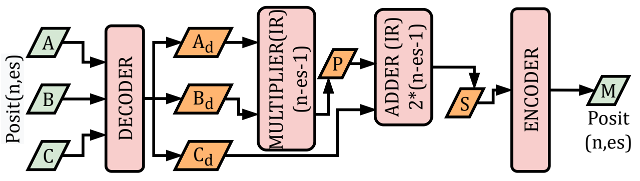

Fig. 10: Posit FMA

operations as well as an adder. The rounding unit must also output the bit sequences for isZero and isNaR depending on the value of those parameters for the input operands. The final rounded bit sequence is concatenated with the sign to obtain the final Posit result. As seen in the figure, there is no 2's complementing required when the sign is negative, as this is taken care of much earlier with the multiplexers.

E. The Fused Multiply-Adder

The Posit FMA essentially combines the multiplier and adder modules to create a pipelined MAC unit. Fig. 10 shows the basic modules of the FMA unit. There is no encoding between the multiplier and adder stages, which leaves the rounding to the end, thus reducing the possibilities for an error. The FMA in the *smallPosit* design is a two-stage pipelined architecture with a latency of two clock cycles. In the first stage, the three inputs are decoded and A and B are multiplied to obtain the product. In the second stage, the product(P) and C are added and the output is encoded back into a Posit. As a result, the final output is obtained after two clock cycles.

IV. Exploring Optimizations and Approximations

The proposed models of the Posit-based arithmetic operators, with a modular representation of the constituent operations, allows the characterization and optimization of the operator implementations. Further, it also enables the exploration of possible approximations at different hierarchies. In our current work, we utilize the model to propose algorithm-level optimizations over state-of-the-art implementations. Additionally, we explore the impact of standard approximations in the arithmetic operators—both at operator- and application-level. While in the current work, we have limited such exploration to generic approximate methods, the proposed model can be used for exploring Posit-specific approximations as well. However, such an approach is beyond the scope of the current article.

A. Algorithm-level optimizations

1) Removing Adder/Subtractor Multiplexing

Posit adder implementations involve addition or subtraction at the arithmetic operation stage, which is determined based on the sign of the intermediate representation of both operands. Similar signs indicate that an addition operation is to be performed whereas dissimilar signs indicate a subtraction operation is to be performed. Thus, PACoGEN's [8] Posit adder implementation uses standalone instances of both an adder and a subtractor module in the arithmetic operation stage, whose inputs are the magnitude of the operands. Instead, we use a unified binary adder-subtractor module that intrinsically 2's complements the second operand when a subtraction operation is required to reduce resource utilization.

2) Removing Redundant Comparator

Observing the schematic of the Posit Adder (in Fig. 6), we notice that a comparator is utilized to obtain the larger scale, which is then fed into the subtractor as an input with the smaller scale. We can eliminate this comparator by directly subtracting the two scales and using the sign as a select line to the multiplexer. In this case, we do not require the multiplexer to output the smaller scale, but instead to 2's complement the difference in case it is negative.

B. Exploring Approximations

Various approximations, similar to the work in [13] and [14], could be introduced to optimize the resource utilization of Posits. However, these works are mainly focused on 16-bit and 32-bit Posits, where the integer arithmetic module, especially the multiplier, consumes a comparable amount of power to the decoder and encoder. According to [14], the multiplier consumes 70% of the power in these cases. Also, they limit their exploration to posit-specific approximations. One current work is orthogonal to these approaches and complements them by leveraging the large body of research into integer approximation methods. We limit our exploration to low-precision (8-bits) arithmetic only and focus on the following methods.

1) Using Truncated Multipliers

Truncation is one of the more generic and widely used methods for implementing low-cost arithmetic. For the truncated multiplier, we eliminate 1-2 bits from the LSB of both input operands and append the appropriate number of zeros to the multiplier result.

2) Using Approximate Signed Operators

A large body of research is dedicated to implementing approximate operators by partial removal of the circuit logic. Such approaches have resulted in the design of both generic [19] and application specific operators [4]. In one current work, we use the approximate operators proposed in [19] as the integer multiplier. Since the work in [19] is only designed for even multipliers currently, this approximation could not be applied to the case (8,2).

3) Using Approximate XOR Multiplier

Using XOR operations for a subset of the bits forms an extreme way of implementing approximate multiplication. We used the case where the input operand bits are XORed to generate the MSBs of the integer multiplication result, wth the LSBs being set to all zeros.

C. PyTorch-based Application-level Exploration

To evaluate approximate operators, they are evaluated in application-specific contexts. Applications with implicit error tolerance present ample scope for implementing approximate arithmetic while providing disproportionate gain in implementation cost. AI/ML-based systems form one primary example of such applications. Multiple works have focused on building frameworks for Posit-based Neural Network inference [20]–[23]. While [21] performs the accumulation in single-precision, [20] and [22] utilize existing Posit frameworks for the operations. The work in [23] presents FP, FxP and Posit-based EMACs (exact multiply and accumulate operations) with hardware characterizations. However the resource utilization clearly shows limitations in 8-bit Posit when compared to FP, FxP operations of the same precision. None of the aforementioned works utilize approximate operators. In this work, we present a novel PyTorch-based framework for running Neural Network inference with Posits. The features of this framework allow for selecting approximate operators to estimate the application-level accuracy when such operators are used in the MAC units. Currently, the framework only supports linear layers. PyTorch allows for the creation and integration of custom operators in C++. The Posit operations are carried out using *Universal* by Stillwater [24], a C++ header library. The operators are written in a language called TorchScript [25].

The general inference flow of operators is shown in Fig. 11. Approximations can currently be applied only at the layer level, and not for individual nodes. The parameters (including n_{mult} and n_{add} for parallel units) are sent to the Posit MAC function

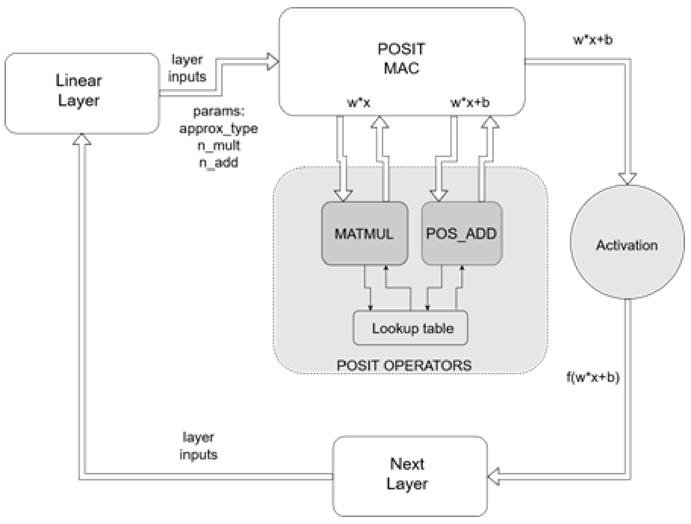

Fig. 11: Operator flow for Neural Network inference

(scripted in Python), which calls the individual Posit matrix multiplication and matrix add operators. The output is then passed through the activation function, which currently does not use Posit. Similarly, each of the next layer is called to run through the same loop.

V. Experiments and Results

A. Experiment Setup

To make approximations, an important step is to characterize the various constituent operations of the implementation. To do so, we sub-divide the Posit adder, multiplier and MAC unit into modules similar to the flow in Fig. 3. Therefore, the decoder is the point up to main adder/multiplier, and the encoder starts right after the integer module. Note that the term *decoder* and *encoder* here are not the standard Posit decoder and encoder as they are performing additional operations required for each specific operator. For example, the decoder module for the adder contains the sticky right shift, comparator, and subtractor. The decoder for the multiplier does not include these modules. Since *smallPosit* [18] is written in Chisel language, we proceed to subdivide using the same script. The designs were implemented in Chisel language [26]. Once each sub-module was implemented, we obtained the Verilog code for all cases of n=8 and below. Chisel first converts to FIRRTL [27] and then to Verilog. Each of these modules is instantiated in an external wrapper that connects the required input and output wires. The modules require a “DON’T TOUCH” condition to view hierarchical power and utilization for each module. Finally, the external wrapper is further instantiated in an outer wrapper with clock, reset and registers.

The modular designs were characterized in Xilinx Vivado 2020.1 [] on a Zynq Ultrascale Board xczu7ev-ffvc1156-2-e. The clock constraints were optimized over 10 iterations using a Tool Command Language (TCL) script. To obtain the power, we require a Switching Activity Interchange Format (SAIF) which is fed to the power report. The SAIF file can be generated by simulating the design with all possible testcases in random order. For n=8, we obtain 216 such testcases. These cases were also verified with the results obtained from the original *smallPosit* design. Once the synthesis, implementation, and post-implementation behavioral simulation are complete, we obtain the power, delay and utilization of the design.

B. Modular Characterization

The power and utilization of the adder can be seen in Fig. 12 and Fig. 13. It is evident that most of the power dissipation and LUT utilization come from the decoder and encoder. The decoder of the adder seems to dominate in the case of power and utilization, which can be attributed to the comparator, subtractor and shifter modules. The encoder consists of an additional ILZD and shifter, which also contribute to the power. A similar distribution can be seen in the case of the multiplier, although here the power consumed by the integer multiplier module is more comparable to that of the decoder. The multiplication module power and utilization reduce with *es*, as seen in Fig. 14 and Fig. 15. This is due to the decrease in fraction width with an increase in *es*, which reduces the multiplier size. The encoder module in the multiplier consists of two comparators which are a part of the overflow/underflow module. As the size of the scale increases with *es*, so does the comparator size, which gives rise to an increase in encoder power with an increase in *es*.

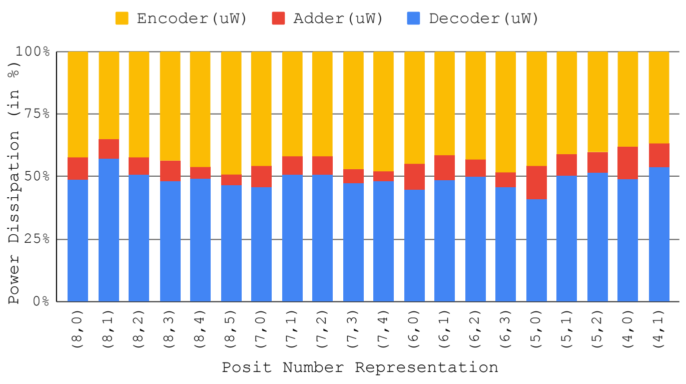

Fig. 12: Modular power dissipation for Posit adder

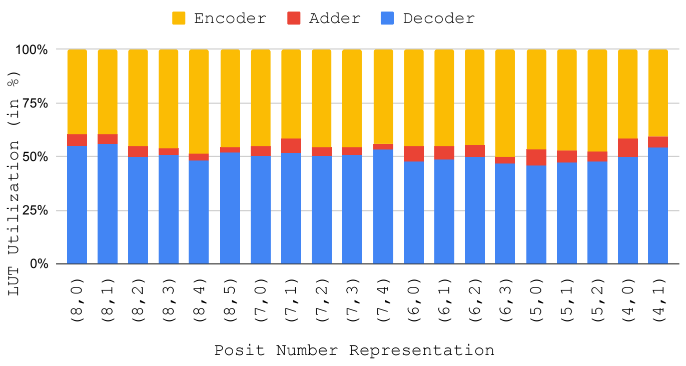

Fig. 13: Modular utilization for Posit adder

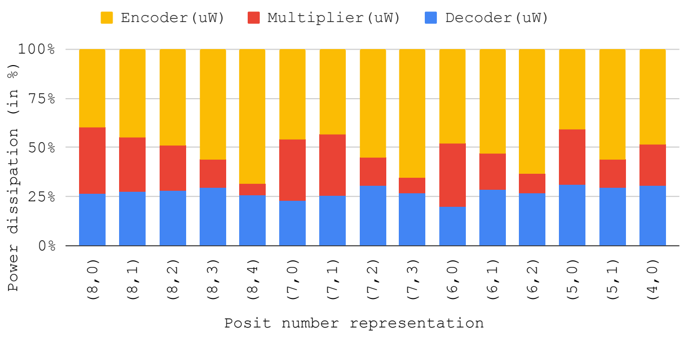

Fig. 14: Modular power dissipation for multiplier

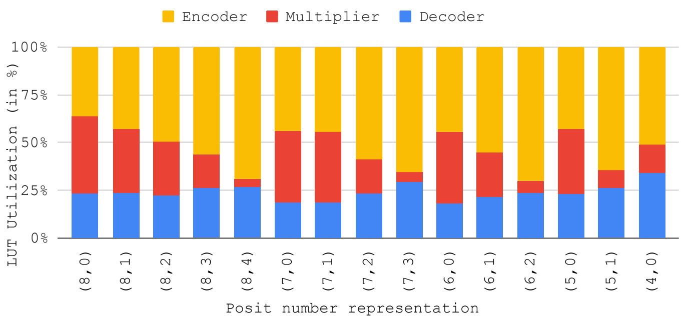

Fig. 15: Modular utilization for multiplier

C. Circuit-level Optimization

The proposed improvements over PACoGEN [8], by removing the adder-subtractor multiplexing, resulted in reduced resource utilization and reduced CPD. For instance, in the case of Posit(8,2) adder, the LUT utilization reduced from 189 to 178 and the CPD from 12.99ns to 11.96ns. However, the smallPosit [18] implementation reports LUT utilization of 143 and CPD of 12.851ns. Consequently, for the rest of the article, we use the *smallPosit* results for comparison.

The comparator removal in the Posit adder saves power and utilization, as shown in Fig. 16 and Fig. 17. As seen in Fig. 16, the power saved is most prominent for cases (8,0), (8,1), (8,5), and (6,0). Although the subtractor width increases with an increase in es, as shown in TABLE II, since the size of the scale increases as well, the additional 2's complementing could offset the decrease in power, depending on switching activity. However, for the most part, this elimination of the comparator reduces power dissipation by up to 17% and utilization by up to 40%.

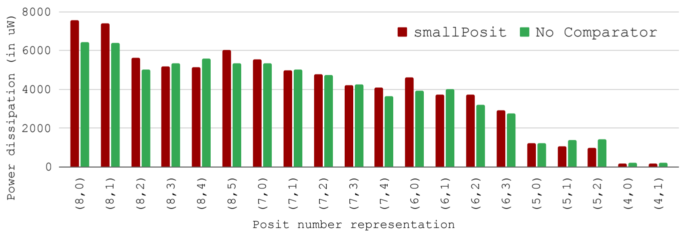

Fig. 16: Comparing power dissipation with comparator removal

The comparator removal can be applied to the Posit FMA. As seen in Fig. 18, more than 50% of the power is consumed by the integer multiply-accumulate component of the FMA. Hence, there is some scope for optimizing the integer modules

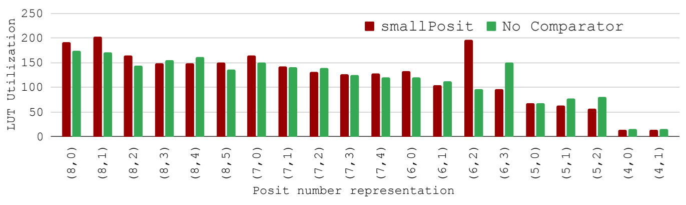

Fig. 17: Comparing utilization with comparator removal

TABLE II: Adder and subtractor widths for multipliers

Design	Posit(8,0)	Posit(8,1)	Posit(8,2)	Posit(8,3)	Posit(8,4)	Posit(8,5)
Adder width	7	6	5	4	3	2
Subtractor width	4	5	6	7	8	9

in this case. Having applied the modification, we observe a power reduction in many cases, as seen in Fig. 19. There is a 16-20% improvement in cases such as (8,1) and (7,1). There is an improvement in the LUT count in almost all cases as seen in Fig. 20. Therefore the power increase in some cases in Fig. 19 can be attributed to bit-switches which result from 2's complementing the scale. Further, it is important to note that the No comparator versions are modularized versions of the smallPosit hardware.

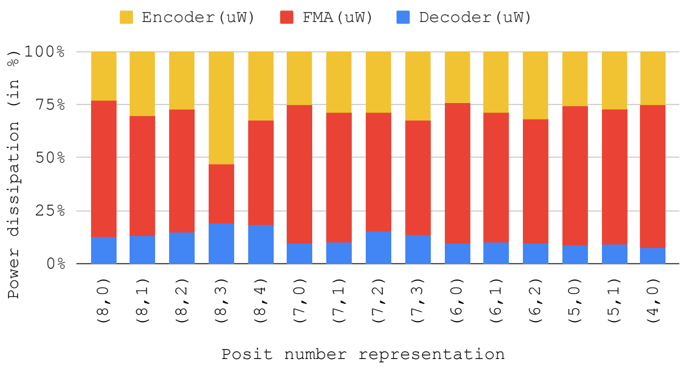

Fig. 18: Modular power dissipation of Posit FMA designs

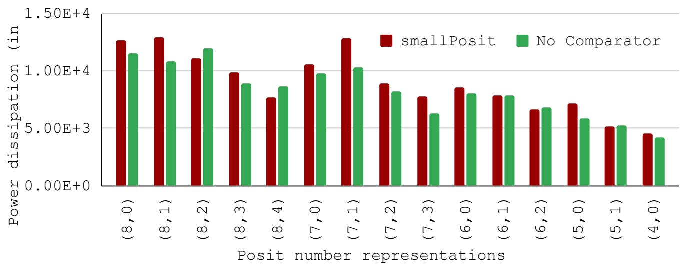

Fig. 19: FMA Power dissipation with comparator removal

D. Using Approximate Components

The results for power and utilization for the accurate (smallPosit) and approximate Posit multiplier are shown in Fig. 21 and Fig. 22. As seen in the figures, the Approximate-XOR multiplier achieves the minimum power in many cases, but also the maximum error as shown in Fig. 23. The best power-accuracy trade-off is obtained by the 1-bit reduce precision multiplier. For cases like (8,3) which are useful for applications such as ML inference, there is a 33% improvement in power with only a 4% error. For cases such as (7,1) and (6,0), we obtain an improvement of over 40% with less than 4% error.

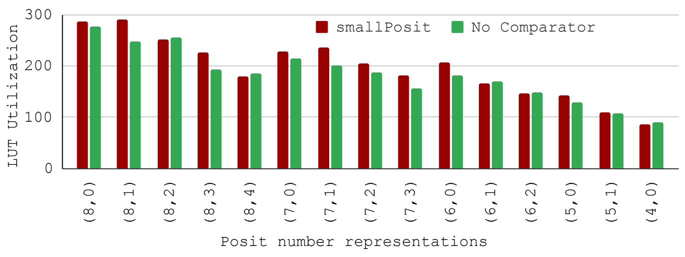

Fig. 20: FMA LUT utilization with comparator removal

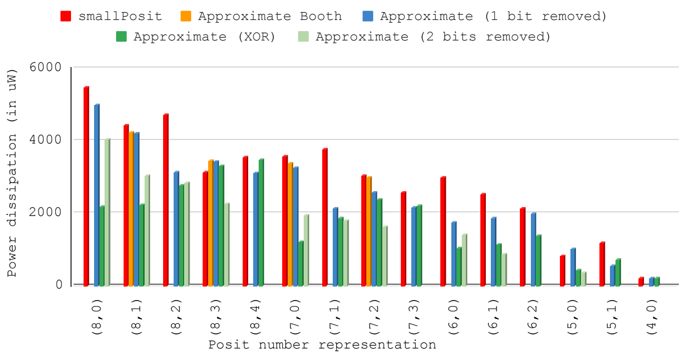

Fig. 21: Comparing power dissipation with approximation

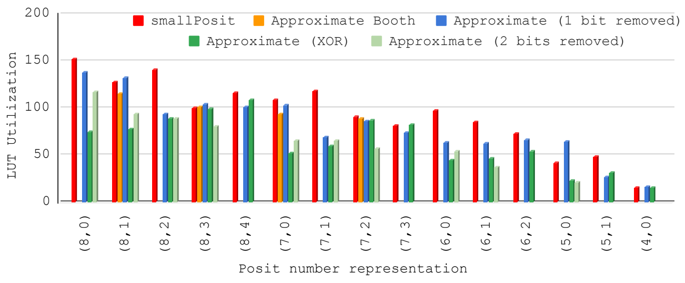

Fig. 22: Comparing utilization with approximation

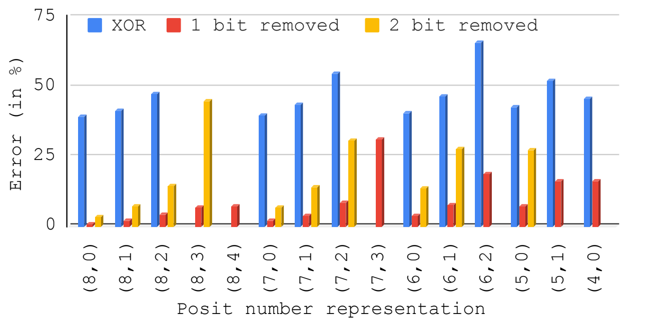

Fig. 23: Multiplier error comparison

Fig. 24 illustrates the error and resource improvement for a Posit Multiplier with 1-bit reduced precision. As seen in the figure, replacing the multiplier can increase the resources as well, in smaller cases of n. This can be attributed to the overheads incurred which offset the small improvement

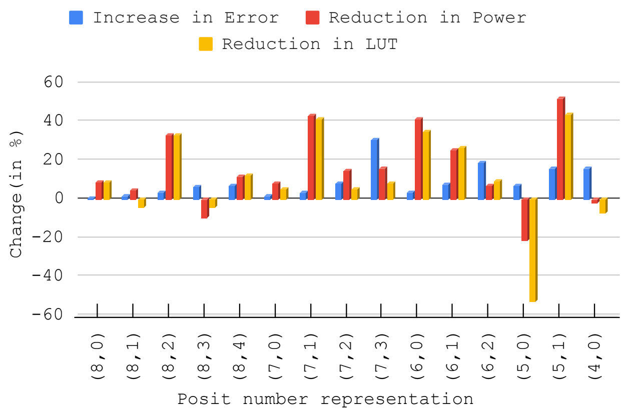

Fig. 24: Power-Accuracy trade-off for 1-bit truncation-based integer multiplier

brought about by replacing a minuscule multiplier (as the fraction width is much lower for n=4 and n=5). As discussed previously, the best improvement is obtained for (6,0) and (7,1). In future work, these approximations could be utilized in combination with other optimizations in the decoder to bring about a greater improvement in resource utilization. Similar approximations are applied to the Posit FMA unit, as shown in Fig. 25 and Fig. 26. In this instance, the integer multiplier with reduced precision is utilized. This brings about almost a 40% improvement in power in case of (7,1) and 19% in (8,3). In case of utilization, there is an improvement of over 20% in cases such as (8,3), (7,3) and (5,0). It is important to note that these approximations are applied to the design with the comparator. Greater improvements could come about by removing the comparator as well. Future work in approximating the FMA involves reducing the adder width in combination with various approximate multipliers. The Booth multiplier can only be applied to even cases. However it brings about great improvement in the power of the FMA: 40% in (8,1) and (7,0).

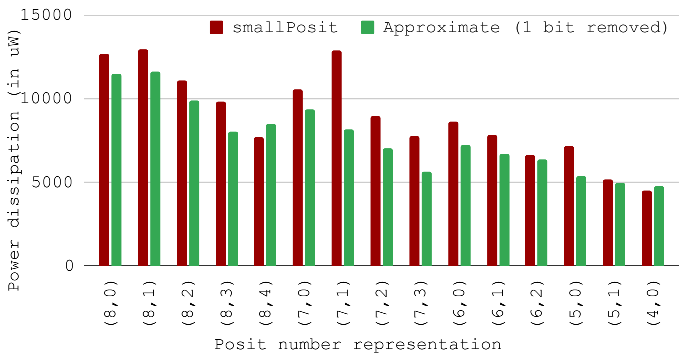

Fig. 25: FMA power with 1-bit truncation

E. Application-level Exploration

To test the framework with Posit, we use the MNIST dataset with a fully-connected network of 2 hidden layers. The overall layer sizes used for the MNIST network are 784, 100, 64 and 10. The network was trained in floating point, and the inference was run on 8196 images using 8-bit Posit

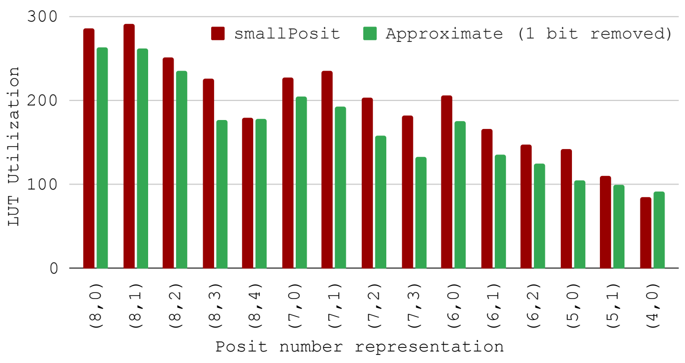

Fig. 26: FMA utilization with 1-bit truncation

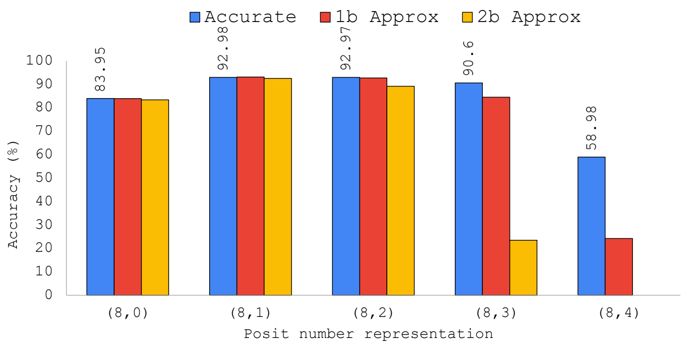

Fig. 27: MNIST Inference accuracy

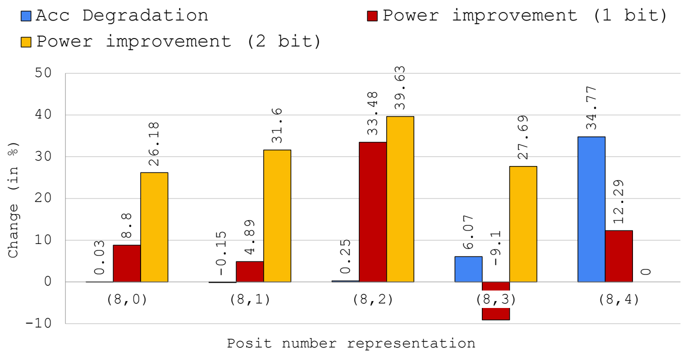

Fig. 28: MNIST power-accuracy trade-off

with a reduced precision approximated multiplier, as shown in Fig. 27. As seen in the figure, there is negligible accuracy degradation for Posit(8,0), and the accuracy degrades by a higher percent as *es* increases (due to the decrease in mantissa width). We also see that the highest accuracy is obtained for Posit(8,1). The accuracy comparison for exact operators is shown in TABLE III. Fig. 28 shows the power improvement alongside the accuracy degradation for a 1-bit reduced precision approximate multiplier. The best case is obtained in Posit(8,2), where 33% power is saved in a single multiplier with less than 0.3% accuracy degradation overall. Currently, the framework takes less than 1 second per image inference. This speed can be improved by utilizing direct operations in C++ instead of lookup tables. Future work involves improving the inference speed and integrating convolutional operators along with Posit-approximate activation functions. The network can then be tested on multiple other datasets such as CIFAR and Fashion-MNIST.

TABLE III: Comparing accuracy for Posit formats

FP32	Posit(8,0)	Posit(8,1)	Posit(8,2)	Posit(8,3)	Posit(8,4)
93.79	83.95	92.98	92.97	90.6	58.98

VI. Conclusion

Posits offer many advantages over the IEEE-754 standard of floating Point with their dynamic range, tapered accuracy, and resolution (especially in the golden region). Their benefits can be leveraged in applications such as Deep Neural Network inference, replacing traditional fixed-point quantization in the context of low precision and offering significant memory savings. However, the hardware characterization of Posits shows that the decoding logic is more complex and may consume more power and LUTs than floating-point. We introduced an optimization by eliminating the comparator in the Posit Adder, bringing about improvements of up to 17% in power and 40% in utilization. To further reduce the overall power consumption, approximations were made in the integer arithmetic module of the Posit Multiplier and FMA. The results indicate that this may not always be advantageous in case of 8-bit Posits and lower, as the mantissa width is not significant (the maximum width is 7 bits for (8,0) including the hidden 1 and sign). However for multiple cases there is a reduction of 33-40% in power, respectively. It is imperative to select the type of approximation depending on the application and the selected parameters of n and es. Another possibility to explore is the fixed regime length, which could significantly simplify decoding logic. However, restricting the regime length also affects the range and accuracy of numbers, bringing us back to the notion that optimizations must be made for specific applications. To behaviorally test the operators, they were integrated into PyTorch using lookup tables. It is found that es=1 and es=2 obtained the best accuracy for n=8, with accuracy degradation being minimum at the lowest es. It is evident that Posit numbers can offer great benefits over floating-point, not just due to the error-resilience of applications but also due to their dynamic range and other hardware-friendly properties. It is worth observing and comparing the metrics above with fixed-point numbers to reduce the overall resource utilization for low-power systems. Future work looks towards optimizing and standardizing Posit hardware as far as possible to make them usable in multiple applications.

References

[1] A. Lacoste, A. Luccioni, V. Schmidt, and T. Dandres, "Quantifying the carbon emissions of machine learning," *arXiv preprint arXiv:1910.09700*, 2019.

[5] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.*, vol. 4, pp. 71–86, 2017.

[2] J. Lin *et al.*, "Mcunet: Tiny deep learning on iot devices," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 711–11 722, 2020.

[3] S. Gupta, S. Ullah, K. Ahuja, A. Tiwari, and A. Kumar, "Align: A highly accurate adaptive layerwise log_2_lead quantization of pre-trained neural networks," *IEEE Access*, vol. 8, pp. 118 899–118 911, 2020.

[4] S. Ullah, S. S. Sahoo, N. Ahmed, D. Chaudhury, and A. Kumar, "AppAxO: Designing Application-specific Approximate Operators for FPGA-based Embedded Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[6] S. Nambi *et al.*, "ExPAN(N)D: Exploring Posits for Efficient Artificial Neural Network Design in FPGA-Based Systems," *IEEE Access*, vol. 9, pp. 103 691–103 708, 2021.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[8] M. K. Jaiswal and H. K.-H. So, "Pacogen: A hardware posit arithmetic core generator," *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019.

[9] R. Chaurasiya *et al.*, "Parameterized posit arithmetic hardware generator," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 334–341.

[10] F. Xiao *et al.*, "Posit arithmetic hardware implementations with the minimum cost divider and squareroot," *Electronics*, vol. 9, no. 10, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/10/1622

[11] H. Zhang and S.-B. Ko, "Design of power efficient posit multiplier," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 861–865, 2020.

[12] S. Jean *et al.*, "P-fma: A novel parameterized posit fused multiply-accumulate arithmetic processor," in *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*, 2021, pp. 282–287.

[13] C. J. Norris and S. Kim, "An approximate and iterative posit multiplier architecture for fpgas," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[14] R. Murillo *et al.*, "Plam: a posit logarithm-approximate multiplier," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2021.

[15] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "Fast approximations of activation functions in deep neural networks when using posit arithmetic," *Sensors*, vol. 20, no. 5, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/5/1515

[16] ——, "A novel posit-based fast approximation of elu activation function for deep neural networks," in *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2020, pp. 244–246.

[17] V. Gohil, S. Walia, J. Mekie, and M. Awasthi, "Fixed-posit: A floating-point representation for error-resilient applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 10, pp. 3341–3345, 2021.

[18] B. Wu., "SmallPositHDL," https://github.com/starbrilliance/SmallPositHDL, 2020.

[19] S. Ullah, H. Schmidl, S. S. Sahoo, S. Rehman, and A. Kumar, "Area-optimized accurate and approximate softcore signed multiplier architectures," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 384–392, 2021.

[20] R. Murillo, A. A. Del Barrio, and G. Botella, "Deep pensieve: A deep learning framework based on the posit number system," *Digital Signal Processing*, vol. 102, p. 102762, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S105120042030107X

[21] T. Zhang, Z. Lin, G. Yang, and C. De Sa, "Qpytorch: A low-precision arithmetic simulation framework," in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, 2019, pp. 10–13.

[22] G. Raposo, P. Tomás, and N. Roma, "Positnn: Training deep neural networks with mixed low-precision posit," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 7908–7912.

[23] Z. Carmichael *et al.*, "Deep positron: A deep neural network using the posit number system," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1421–1426.

[24] E. T. L. Omtzigt, P. Gottschling, M. Seligman, and W. Zorn, "Universal Numbers Library: design and implementation of a high-performance reproducible number systems library," *arXiv:2012.11011*, 2020.

[25] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[26] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[27] A. Izraelevitz *et al.*, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 209–216.