Deep Positron: A Deep Neural Network Using the Posit Number System

Zachariah Carmichael[§], Hamed F. Langroudi[§], Char Khazanov[§], Jeffrey Lillie[§],
John L. Gustafson[*], Dhireesha Kudithipudi[§]
[§]Neuromorphic AI Lab, Rochester Institute of Technology, NY, USA
[*]National University of Singapore, Singapore

***Abstract*—The recent surge of interest in Deep Neural Networks (DNNs) has led to increasingly complex networks that tax computational and memory resources. Many DNNs presently use 16-bit or 32-bit floating point operations. Significant performance and power gains can be obtained when DNN accelerators support low-precision numerical formats. Despite considerable research, there is still a knowledge gap on how low-precision operations can be realized for both DNN training and inference. In this work, we propose a DNN architecture, Deep Positron, with posit numerical format operating successfully at $\leq$8 bits for inference. We propose a precision-adaptable FPGA soft core for exact multiply-and-accumulate for uniform comparison across three numerical formats, fixed, floating-point and posit. Preliminary results demonstrate that 8-bit posit has better accuracy than 8-bit fixed or floating-point for three different low-dimensional datasets. Moreover, the accuracy is comparable to 32-bit floating-point on a Xilinx Virtex-7 FPGA device. The trade-offs between DNN performance and hardware resources, *i.e.* latency, power, and resource utilization, show that posit outperforms in accuracy and latency at 8-bit and below.**

***Index Terms*—deep neural networks, machine learning, DNN accelerators, posits, floating point, tapered precision, low-precision**

I. Introduction

Deep neural networks are highly parallel workloads which require massive computational resources for training and often utilize customized accelerators such as Google's Tensor Processing Unit (TPU) to improve the latency, or reconfigurable devices like FPGAs to mitigate power bottlenecks, or targeted ASICs such as Intel's Nervana to optimize the overall performance. The training cost of DNNs is attributed to the massive number of primitives known as multiply-and-accumulate operations that compute the weighted sums of the neurons' inputs. To alleviate this challenge, techniques such as sparse connectivity and low-precision arithmetic [1]–[3] are extensively studied. For example, performing AlexNet inference on Cifar-10 dataset using 8-bit fixed-point format has shown 6× improvement in energy consumption [4] over the 32-bit fixed-point. On the other hand, using 32-bit precision for an outrageously large neural network, such as LSTM with mixture of experts [5], will approximately require 137 billion parameters. When performing a machine translation task with this network, it translates to an untenable DRAM memory access power of 128 W[1]. For deploying DNN algorithms on the end-device (*e.g.* AI on the edge, IoT), these resource constraints are prohibitive.

[1]Estimated Power = (20 Hz × 10 G × 640 pJ (for a 32-bit DRAM access [1])) at 45nm technology node

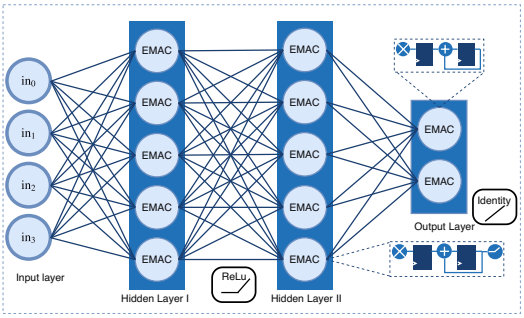

Fig. 1: An overview of a simple Deep Positron architecture embedded with the exact multiply-and-accumulate blocks (EMACs).

Researchers have offset these constraints to some degree by using low-precision techniques. Linear and nonlinear quantization have been successfully applied during DNN inference on 8-bit fixed-point or 8-bit floating point accelerators and the performance is on par with 32-bit floating point [3], [6], [7]. However, when using quantization to perform DNN inference with ultra-low bit precision ($\leq$8-bits), the network needs to be retrained or the number of hyper-parameters should be significantly increased [8], leading to a surge in computational complexity. One solution is to utilize a low-precision numerical format (fixed-point, floating point, or posit [9]) for both DNN training and inference instead of quantization. Earlier studies have compared DNN inference with low-precision (*e.g.* 8-bit) to a floating point high-precision (*e.g.* 32-bit) [4]. The utility of these studies is limited – the comparisons are across numerical formats with different bit widths and do not provide a fair understanding of the overall system efficiency.

More recently, the posit format has shown promise over floating point with larger dynamic range, higher accuracy, and better closure [10]. The goal of this work is to study the efficacy of the posit numerical format for DNN inference. An analysis of the histogram of weight distributions in an AlexNet DNN and a 7-bit posit (Fig. 2) shows that posits can be an optimal representation of weights and activations. We compare the proposed designs with multiple metrics related to performance and resource utilization: accuracy, LUT utilization, dynamic range of the numerical formats, maximum operating frequency, inference time, power consumption, and

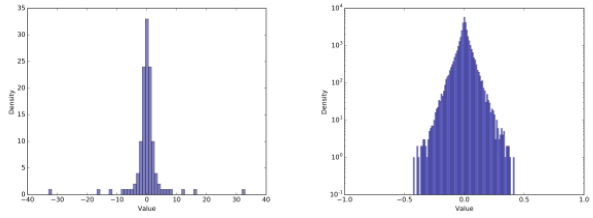

Fig. 2: (a) 7-bit posit ($es = 0$) and (b) AlexNet weight distributions. Both show heavy clustering in [-1,1] range.

energy-delay-product.

This paper makes the following contributions:

- We propose an exact multiply and accumulate (EMAC) algorithm for accelerating ultra-low precision ($\leq$8-bit) DNNs with the posit numerical format. We compare EMACs for three numerical formats, posit, fixed-point, and floating point, in sub 8-bit precision.
- We propose the Deep Positron architecture that employs the EMACs and study the resource utilization and energy-delay-product.
- We show preliminary results that posit is a natural fit for sub 8-bit precision DNN inference.
- We conduct experiments on the Deep Positron architecture for multiple low-dimensional datasets and show that 8-bit posits achieve better performance than 8-bit fixed or floating point and similar accuracies as the 32-bit floating point counterparts.

II. Background

A. Deep Neural Networks

Deep neural networks are biologically-inspired predictive models that learn features and relationships from a corpus of examples. The topology of these networks is a sequence of layers, each containing a set of simulated neurons. A neuron computes a weighted sum of its inputs and produces a nonlinear *activation function* of that sum. The connectivity between layers can vary, but in general, it has feed-forward connections between layers. These connections each have an associated numerical value, known as a *weight*, that indicates the connection strength. To discern correctness of a given network's predictions in a supervised environment, a *cost function* computes how wrong a prediction is compared to the truth. The partial derivatives of each weight with respect to the cost are used to update network parameters through backpropagation, ultimately minimizing the cost function.

Traditionally 32-bit floating point arithmetic is used for DNN inference. However, the IEEE standard floating point representation is designed for a very broad dynamic range; even 32-bit floating point numbers have a huge dynamic range of over 80 orders of magnitude, far larger than needed for DNNs. While very small values can be important, very large values are not, therefore the design of the numbers creates low information-per-bit based on Shannon maximum entropy [11]. Attempts to address this by crafting a fixed-point representation for DNN weights quickly runs up against the *quantization error*.

The 16-bit (half-precision) form of IEEE floating point, used by Nvidia's accelerators for DNN, reveals the shortcomings of the format: complicated exception cases, gradual underflow, prolific NaN bit patterns, and redundant representations of zero. It is not the representation to design from first principles for a DNN workload. A more recent format, *posit arithmetic*, provides a natural fit to the demands of DNNs both for training and inference.

B. Posit Number System

The posit number system, a Type III unum, was proposed to improve upon many of the shortcomings of IEEE-754 floating-point arithmetic and to address complaints about the costs of managing the variable size of Type I unums [10]. (Type II unums are also of fixed size, but require look-up tables that limits their precision [12].) Posit format provides better dynamic range, accuracy, and consistency between machines than floating point. A posit number is parametrized by n, the total number of bits, and es, the number of exponent bits. The primary difference between a posit and floating-point number representation is the posit *regime* field, which has a dynamic width like that of a unary number; the regime is a run-length encoded signed value that can be interpreted as in Table I.

TABLE I: Regime Interpretation

Binary	0001	001	01	10	110	1110
Regime (k)	-3	-2	-1	0	1	2

Two values in the system are reserved: 10...0 represents "Not a Real" which includes infinity and all other exception cases like $0/0$ and $\sqrt{-1}$, and 00...0 represents zero. The full binary representation of a posit number is shown in (1).

$$\underbrace{\overbrace{s}^{\text{Sign}}\ \overbrace{r\ r\ ...\ r\ \bar{r}}^{\text{Regime}}\ \overbrace{e_1\ e_2\ e_3\ ...\ e_{es}}^{\text{Exponent, if any}}\ \overbrace{f_1\ f_2\ f_3\ ...}^{\text{Mantissa, if any}}}_{n\text{ Bits}} \tag{1}$$

For a positive posit in such format, the numerical value it represents is given by (2)

$$(-1)^s \times \left(2^{2^{es}}\right)^k \times 2^e \times 1.f \tag{2}$$

where k is the regime value, e is the unsigned exponent (if $es > 0$), and f comprises the remaining bits of the number. If the posit is negative, the 2's complement is taken before using the above decoding. See [10] for more detailed and complete information on the posit format.

III. Methodology

A. Exact Multiply-and-Accumulate (EMAC)

The fundamental computation within a DNN is the multiply-and-accumulate (MAC) operation. Each neuron within a network is equivalent to a MAC unit in that it performs a weighted sum of its inputs. This operation is ubiquitous across many

DNN implementations, however, the operation is usually inexact, *i.e.* limited precision, truncation, or premature rounding in the underlying hardware yields inaccurate results. The EMAC performs the same computation but allocates sufficient padding for digital signals to emulate arbitrary precision. Rounding or truncation within an EMAC unit is delayed until every product has been accumulated, thus producing a result with minimal local error. This minimization of error is especially important when EMAC units are coupled with low-precision data.

In all EMAC units we implement, a number's format is arbitrary as its representation is ultimately converted to fixed-point, which allows for natural accumulation. Given the constraint of low-precision data, we propose to use a variant of the Kulisch accumulator [13]. In this architecture, a wide register accumulates fixed-point values shifted by an exponential parameter, if applicable, and delays rounding to a post-summation stage. The width of such an accumulator for k multiplications can be computed using (3)

$$w_a = \lceil \log_2(k) \rceil + 2 \times \left\lceil \log_2 \left(\frac{max}{min} \right) \right\rceil + 2 \tag{3}$$

where max and min are the maximum and minimum values for a number format, respectively. To improve the maximum operating frequency via pipelining, a D flip-flop separates the multiplication and accumulation stages. The architecture easily allows for the incorporation of a bias term – the accumulator D flip-flop can be reset to the fixed-point representation of the bias so products accumulate on top of it. To further improve accuracy, the *round to nearest* and *round half to even* scheme is employed for the floating point and posit formats. This is the recommended IEEE-754 rounding method and the posit standard.

B. Fixed-point EMAC

The fixed-point EMAC, shown in Fig. 3, accumulates the products of k multiplications and allocates a sufficient range of bits to compute the exact result before truncation. A weight, bias, and activation, each with q fraction bits and $n-q$ integer bits, are the unit inputs. The unnormalized multiplicative result is kept as $2n$ bits to preserve exact precision. The products are accumulated over k clock cycles with the integer adder and D flip-flop combination. The sum of products is then shifted right by q bits and truncated to n bits, ensuring to clip at the maximum magnitude if applicable.

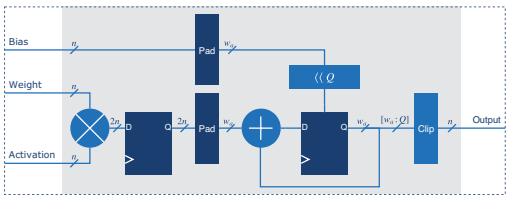

Fig. 3: A precision-adaptable (DNN weights and activation) FPGA soft core for fixed-point exact multiply-and-accumulate operation.

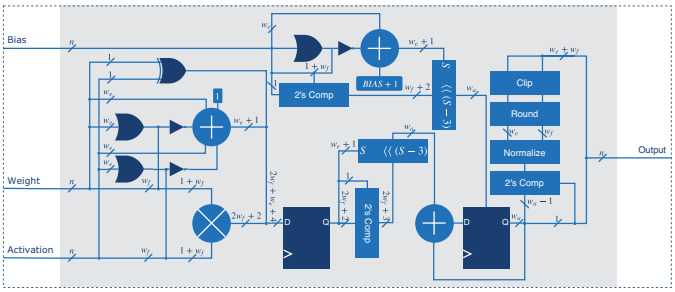

Fig. 4: A precision-adaptable (DNN weights and activation) FPGA soft core for the floating-point exact multiply-and-accumulate operation.

C. Floating Point EMAC

The floating point EMAC, shown in Fig. 4, also computes the EMAC operation for k pairs of inputs. We do not consider "Not a Number" or the "$\pm$ Infinity" as inputs don't have these values and the EMAC does not overflow to infinity. Notably, it uses a fixed-point conversion before accumulation to preserve precision of the result. Inputs to the EMAC have a single signed bit, w_e exponent bits, and w_f fraction bits. Subnormal detection at the inputs appropriately sets the hidden bits and adjusts the exponent. This EMAC scales exponentially with w_e as it's the dominant parameter in computing w_a. To convert floating point products to a fixed-point representation, mantissas are converted to 2's complement based on the sign of the product and shifted to the appropriate location in the register based on the product exponent. After accumulation, inverse 2's complement is applied based on the sign of the sum. If the result is detected to be subnormal, the exponent is accordingly set to '0'. The extracted value from the accumulator is clipped at the maximum magnitude if applicable.

The relevant characteristics of a float number are computed as follows.

$$\begin{aligned} bias &= 2^{w_e - 1} - 1 \\ exp_{max} &= 2^{w_e} - 2 \\ max &= 2^{exp_{max} - bias} \times (2 - 2^{-w_f}) \\ min &= 2^{1 - bias} \times 2^{-w_f} \end{aligned}$$

Algorithm 1 Posit data extraction of n-bit input with es exponent bits

procedure DECODE(in) ▷ Data extraction of in
  nzero $\leftarrow$ |in ▷ '1' if in is nonzero
  sign $\leftarrow$ in$[n-1]$ ▷ Extract sign
  twos $\leftarrow$ $(\{n-1\{\text{sign}\}\} \oplus \text{in}[n-2:0]) + \text{sign}$ ▷ 2's Comp.
  rc $\leftarrow$ twos$[n-2]$ ▷ Regime check
  inv $\leftarrow$ $\{n-1\{\text{rc}\}\} \oplus$ twos ▷ Invert 2's
  zc $\leftarrow$ LZD(inv) ▷ Count leading zeros
  tmp $\leftarrow$ twos$[n-4:0] \ll (\text{zc} - 1)$ ▷ Shift out regime
  frac $\leftarrow$ $\{\text{nzero}, \text{tmp}[n-es-4:0]\}$ ▷ Extract fraction
  exp $\leftarrow$ tmp$[n-4:n-es-3]$ ▷ Extract exponent
  reg $\leftarrow$ rc ? zc-1 : $-$zc ▷ Select regime
  return sign, reg, exp, frac
end procedure

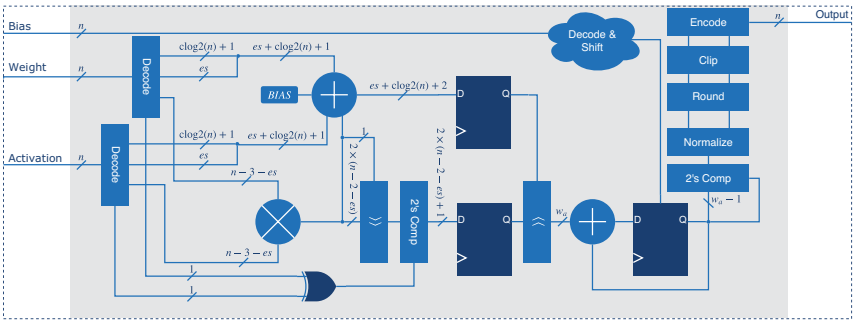

Fig. 5: A precision-adaptable (DNN weights and activation) FPGA soft core for the posit exact multiply-and-accumulate operation.

D. Posit EMAC

The posit EMAC, detailed in Fig. 5, computes the operation of k pairs of inputs. We do not consider "Not a Real" in this implementation as all inputs are expected to be real numbers and posits never overflow to infinity. Inputs to the EMAC are decoded in order to extract the sign, regime, exponent, and fraction. As the regime bit field is of dynamic size, this process is nontrivial. Algorithm 1 describes the data extraction process. To mitigate needing both a leading ones detector (LOD) and leading zeros detector (LZD), we invert the two's complement of the input (line 5) so that the regime always begins with a '0'. The regime is accordingly adjusted using the regime check bit (line 11). After decoding inputs, multiplication and converting to fixed-point is performed similarly to that of floating point. Products are accumulated in a register, or quire in the posit literature, of width $qsize$ as given by (4).

$$qsize = 2^{es+2} \times (n-2) + 2 + \lceil \log_2(k) \rceil, \; n \geq 3 \quad (4)$$

To avoid using multiple shifters in fixed-point conversion, the scale factor $\texttt{sf}_{\texttt{mult}}$ is biased by $bias = 2^{es+1} \times (n-2)$ such that its minimum value becomes 0. After accumulation, the scale factor is unbiased by $bias$ before entering the convergent rounding and encoding stage. Algorithm 2 gives the procedure for carrying out these operations.

Algorithm 2 Posit EMAC operation for n-bit inputs each with es exponent bits

1: **procedure** PositEMAC(weight, activation)
2: $\texttt{sign}_w, \texttt{reg}_w, \texttt{exp}_w, \texttt{frac}_w \leftarrow$ Decode(weight)
3: $\texttt{sign}_a, \texttt{reg}_a, \texttt{exp}_a, \texttt{frac}_a \leftarrow$ Decode(activation)
4: $\texttt{sf}_w \leftarrow \{\texttt{reg}_w, \texttt{exp}_w\}$ ▷ Gather scale factors
5: $\texttt{sf}_a \leftarrow \{\texttt{reg}_a, \texttt{exp}_a\}$

Multiplication

6: $\texttt{sign}_{\texttt{mult}} \leftarrow \texttt{sign}_w \oplus \texttt{sign}_a$
7: $\texttt{frac}_{\texttt{mult}} \leftarrow \texttt{frac}_w \times \texttt{frac}_a$
8: $\texttt{ovf}_{\texttt{mult}} \leftarrow \texttt{frac}_{\texttt{mult}}[\texttt{MSB}]$ ▷ Adjust for overflow
9: $\texttt{normfrac}_{\texttt{mult}} \leftarrow \texttt{frac}_{\texttt{mult}} \gg \texttt{ovf}_{\texttt{mult}}$
10: $\texttt{sf}_{\texttt{mult}} \leftarrow \texttt{sf}_w + \texttt{sf}_a + \texttt{ovf}_{\texttt{mult}}$

Accumulation

11: $\texttt{fracs}_{\texttt{mult}} \leftarrow \texttt{sign}_{\texttt{mult}} \; ? \; -\texttt{frac}_{\texttt{mult}} : \texttt{frac}_{\texttt{mult}}$
12: $\texttt{sf}_{\texttt{biased}} \leftarrow \texttt{sf}_{\texttt{mult}} + bias$ ▷ Bias the scale factor
13: $\texttt{fracs}_{\texttt{fixed}} \leftarrow \texttt{fracs}_{\texttt{mult}} \ll \texttt{sf}_{\texttt{biased}}$ ▷ Shift to fixed
14: $\texttt{sum}_{\texttt{quire}} \leftarrow \texttt{fracs}_{\texttt{fixed}} + \texttt{sum}_{\texttt{quire}}$ ▷ Accumulate

Fraction & SF Extraction

15: $\texttt{sign}_{\texttt{quire}} \leftarrow \texttt{sum}_{\texttt{quire}}[\texttt{MSB}]$
16: $\texttt{mag}_{\texttt{quire}} \leftarrow \texttt{sign}_{\texttt{quire}} \; ? \; -\texttt{sum}_{\texttt{quire}} : \texttt{sum}_{\texttt{quire}}$
17: $\texttt{zc} \leftarrow \text{LZD}(\texttt{mag}_{\texttt{quire}})$
18: $\texttt{frac}_{\texttt{quire}} \leftarrow \texttt{mag}_{\texttt{quire}}[2\times(n-2-es)-1+\texttt{zc} : \texttt{zc}]$
19: $\texttt{sf}_{\texttt{quire}} \leftarrow \texttt{zc} - bias$

Convergent Rounding & Encoding

20: $\texttt{nzero} \leftarrow |\texttt{frac}_{\texttt{quire}}$
21: $\texttt{sign}_{\texttt{sf}} \leftarrow \texttt{sf}_{\texttt{quire}}[\texttt{MSB}]$
22: $\texttt{exp} \leftarrow \texttt{sf}_{\texttt{quire}}[es-1:0]$ ▷ Unpack scale factor
23: $\texttt{reg}_{\texttt{tmp}} \leftarrow \texttt{sf}_{\texttt{quire}}[\texttt{MSB}-1 : es]$
24: $\texttt{reg} \leftarrow \texttt{sign}_{\texttt{sf}} \; ? \; -\texttt{reg}_{\texttt{tmp}} : \texttt{reg}_{\texttt{tmp}}$
25: $\texttt{ovf}_{\texttt{reg}} \leftarrow \texttt{reg}[\texttt{MSB}]$ ▷ Check for overflow
26: $\texttt{reg}_{\texttt{f}} \leftarrow \texttt{ovf}_{\texttt{reg}} \; ? \; \{\{\lceil \log_2(n) \rceil - 2\{1\}\}, 0\} : \texttt{reg}$
27: $\texttt{exp}_{\texttt{f}} \leftarrow (\texttt{ovf}_{\texttt{reg}} | \sim\texttt{nzero} | (\&\texttt{reg}_{\texttt{f}})) \; ? \; \{es\{0\}\} : \texttt{exp}$
28: $\texttt{tmp1} \leftarrow \{\texttt{nzero}, 0, \texttt{exp}_{\texttt{f}}, \texttt{frac}_{\texttt{quire}}[\texttt{MSB}-1:0], \{n-1\{0\}\}\}$
29: $\texttt{tmp2} \leftarrow \{0, \texttt{nzero}, \texttt{exp}_{\texttt{f}}, \texttt{frac}_{\texttt{quire}}[\texttt{MSB}-1:0], \{n-1\{0\}\}\}$
30: $\texttt{ovf}_{\texttt{regf}} \leftarrow \&\texttt{reg}_{\texttt{f}}$
31: **if** $\texttt{ovf}_{\texttt{regf}}$ **then**
32: $\texttt{shift}_{\texttt{neg}} \leftarrow \texttt{reg}_{\texttt{f}} - 2$
33: $\texttt{shift}_{\texttt{pos}} \leftarrow \texttt{reg}_{\texttt{f}} - 1$
34: **else**
35: $\texttt{shift}_{\texttt{neg}} \leftarrow \texttt{reg}_{\texttt{f}} - 1$
36: $\texttt{shift}_{\texttt{pos}} \leftarrow \texttt{reg}_{\texttt{f}}$
37: **end if**
38: $\texttt{tmp} \leftarrow \texttt{sign}_{\texttt{sf}} \; ? \; \texttt{tmp2} \gg \texttt{shift}_{\texttt{neg}} : \texttt{tmp1} \gg \texttt{shift}_{\texttt{pos}}$
39: $\texttt{lsb}, \texttt{guard} \leftarrow \texttt{tmp}[\texttt{MSB}-(n-2) : \texttt{MSB}-(n-1)]$
40: $\texttt{round} \leftarrow \sim(\texttt{ovf}_{\texttt{reg}} | \texttt{ovf}_{\texttt{regf}}) \; ?$ (guard & (lsb | (|tmp[MSB−n : 0]))) : 0
41: $\texttt{result}_{\texttt{tmp}} \leftarrow \texttt{tmp}[\texttt{MSB} : \texttt{MSB}-n+1] + \texttt{round}$
42: $\texttt{result} \leftarrow \texttt{sign}_{\texttt{quire}} \; ? \; -\texttt{result}_{\texttt{tmp}} : \texttt{result}_{\texttt{tmp}}$
43: **return** result
44: **end procedure**

The relevant characteristics of a posit number are computed as follows.

$$useed = 2^{2^{es}}$$
$$max = useed^{n-2}$$
$$min = useed^{-n+2}$$

E. Deep Positron

We assemble a custom DNN architecture that is parametrized by data width, data type, and DNN hyperparameters (*e.g.* number of layers, neurons per layer, *etc.*), as shown in Fig. 1. Each layer contains dedicated EMAC units

with local memory blocks for weights and biases. Storing DNN parameters in this manner minimizes latency by avoiding off-chip memory accesses. The compute cycle of each layer is triggered when its directly preceding layer has terminated computation for an input. This flow performs inference in a parallel streaming fashion. The ReLU activation is used throughout the network, except for the affine readout layer. A main control unit controls the flow of input data and activations throughout the network using a finite state machine.

IV. Experimental Results

A. EMAC Analysis and Comparison

We compare the hardware implications across three numerical format parameters that each EMAC has on a Virtex-7 FPGA (xc7vx485t-2ffg1761c). Synthesis results are obtained through Vivado 2017.2 and optimized for latency by targeting the on-chip DSP48 slices. Our preliminary results indicate that the posit EMAC is competitive with the floating point EMAC in terms of energy and latency. At lower values of $n \leq 7$, the posit number system has higher dynamic range as emphasized by [10]. We compute dynamic range as $\log_{10}\left(\frac{max}{min}\right)$. While neither the floating point or posit EMACs can compete with the energy-delay-product (EDP) of fixed-point, they both are able to offer significantly higher dynamic range for the same values of n. Furthermore, the EDPs of the floating point and posit EMACs are similar.

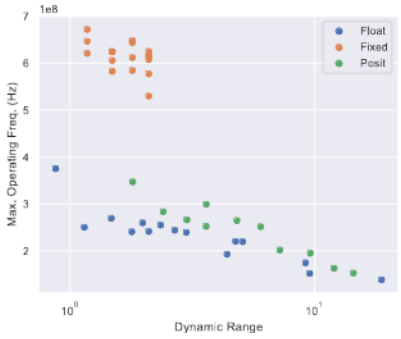

Fig. 6: Dynamic Range vs. Maximum Operating Frequency (Hz) for the EMACs implemented on Xilinx Virtex-7 FPGA.

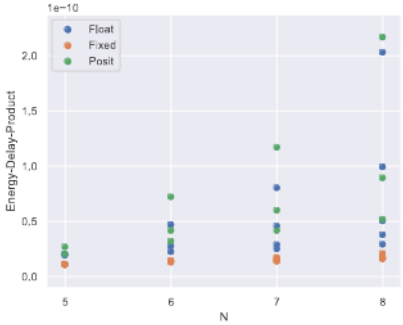

Fig. 7: n vs. energy-delay-product for the EMACs implemented on Xilinx Virtex-7 FPGA.

Fig. 6 shows the synthesis results for the dynamic range of each format against maximum operating frequency. As expected, the fixed-point EMAC achieves the lowest datapath latencies as it has no exponential parameter, thus a narrower accumulator. In general, the posit EMAC can operate at a higher frequency for a given dynamic range than the floating point EMAC. Fig. 7 shows the EDP across different bit-widths and as expected fixed-point outperforms for all bit-widths.

The LUT utilization results against numerical precision n are shown in Fig. 8, where posit generally consumes a higher amount of resources. This limitation can be attributed to the more involved decoding and encoding of inputs and outputs.

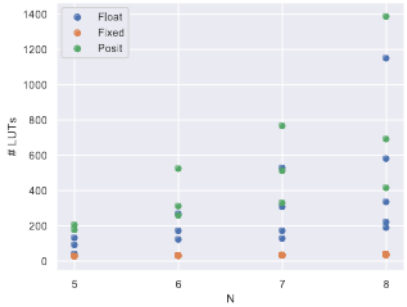

Fig. 8: n vs. LUT Utilization for the EMACs implemented on Xilinx Virtex-7 FPGA.

B. Deep Positron Performance

We compare the performance of Deep Positron on three datasets and all possible combinations of [5,8] bit-widths for the three numerical formats. Posit performs uniformly well across all the three datasets for 8-bit precision, shown in Table II, and has similar accuracy as fixed-point and float in sub 8-bit. Best results are when posit has $es \in \{0, 2\}$ and floating point has $w_e \in \{3, 4\}$. As expected, the best performance drops sub 8-bit by [0-4.21]% compared to 32-bit floating-point. In all experiments, the posit format either outperforms or matches the performance of floating and fixed-point. Additionally, with 24 fewer bits, posit matches the performance of 32-bit floating point on the Iris classification task.

Fig. 9 shows the lowest accuracy degradation per bit width against EDP. The results indicate posits achieve better performance compared to the floating and fixed-point formats at a moderate cost.

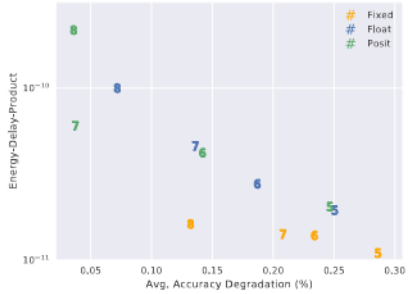

Fig. 9: Average accuracy degradation vs. energy-delay-product for the EMACs implemented on Xilinx Virtex-7 FPGA. Numbers correspond with the bit-width of a numerical format.

TABLE II: Deep Positron performance on low-dimensional datasets with 8-bit EMACs.

			Accuracy		
Dataset	Inference size	Posit	Floating-point	Fixed-point	32-bit Float
Wisconsin Breast Cancer [14]	190	85.89%	77.4%	57.8%	90.1%
Iris [15]	50	98%	96%	92%	98%
Mushroom [16]	2708	96.4%	96.4%	95.9%	96.8%

V. Related Work

Research on low-precision arithmetic for neural networks dates to circa 1990 [17], [18] using fixed-point and floating point. Recently, several groups have shown that it is possible to perform inference in DNNs with 16-bit fixed-point representations [19], [20]. However, most of these studies compare DNN inference for different bit-widths. Few research teams have performed a comparison with same bit-widths across different number systems coupled with FPGA soft processors. For example, Hashemi *et al.* demonstrate DNN inference with 32-bit fixed-point and 32-bit floating point on the LeNet, ConvNet, and AlexNet DNNs, where the energy consumption is reduced by $\sim$12% and $<$1% accuracy loss with fixed-point [4]. Most recently, Chung *et al.* proposed an accelerator, Brainwave, with a spatial 8-bit floating point, called *ms-fp8*. The *ms-fp8* format improves the throughput by 3$\times$ over 8-bit fixed-point on a Stratix-10 FPGA [3].

This paper also relates to three previous works that use posits in DNNs. The first DNN architecture using the posit number system was proposed by Langroudi *et al.* [21]. The work demonstrates that, with $<$1% accuracy degradation, DNN parameters can be represented using 7-bit posits for AlexNet on the ImageNet corpus and that posits require $\sim$30% less memory utilization for the LeNet, ConvNet, and AlexNet neural networks in comparison to the fixed-point format. Secondly, Cococcioni *et al.* [22] discuss the effectiveness of posit arithmetic for application to autonomous driving. They consider an implementation of the Posit Processing Unit (PPU) as an alternative to the Floating point Processing Unit (FPU) since the self-driving car standards require 16-bit floating point representations for the safety-critical application. Recently, Jeff Johnson proposed a log float format as a combination of the posit format and the logarithmic version of the EMAC operation called the exact log-linear multiply-add (ELMA). This work shows that ImageNet classification using the ResNet-50 DNN architecture can be performed with $<$1% accuracy degradation [23]. It also shows that 4% and 41% power consumption reduction can be achieved by using an 8/38-bit ELMA in place of an 8/32-bit integer multiply-add and an IEEE-754 float16 fused multiply-add, respectively.

This paper is inspired by the earlier studies and demonstrates that posit arithmetic with ultra-low precision ($\leq$8-bit) is a natural choice for DNNs performing low-dimensional tasks. A precision-adaptable, parameterized FPGA soft core is used for comprehensive analysis on the Deep Positron architecture with same bit-width for fixed, floating-point, and posit formats.

VI. Conclusions

In this paper, we show that the posit format is well suited for deep neural networks at ultra-low precision ($\leq$8-bit). We show that precision-adaptable, reconfigurable exact multiply-and-accumulate designs embedded in a DNN are efficient for inference. Accuracy-sensitivity studies for Deep Positron show robustness at 7-bit and 8-bit widths. In the future, the success of DNNs in real-world applications will equally rely on the underlying platforms and architectures as much as the algorithms and data. Full-scale DNN accelerators with low-precision posit arithmetic will play an important role in this domain.

References

[1] S. Han, H. Mao, and W. J. Dally, "Deep Compression - Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *Iclr*, pp. 1–13, 2016.

[2] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," *arXiv preprint arXiv:1802.04680*, 2018.

[3] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[4] S. Hashemi, N. Anthony, H. Tann, R. Bahar, and S. Reda, "Understanding the impact of precision quantization on the accuracy and energy of neural networks," in *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2017, pp. 1478–1483.

[5] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le *et al.*, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*, 2017.

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit TM," pp. 1–17, 2017.

[7] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 267–278.

[8] A. Mishra and D. Marr, "Wrpn & apprentice: Methods for training and inference using low-precision numerics," *arXiv preprint arXiv:1803.00227*, 2018.

[9] P. Gysel, "Ristretto: Hardware-oriented approximation of convolutional neural networks," *CoRR*, vol. abs/1605.06402, 2016. [Online]. Available: http://arxiv.org/abs/1605.06402

[10] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[11] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, pp. 623–656, 1948.

[12] W. Tichy, "Unums 2.0: An interview with john l. gustafson," *Ubiquity*, vol. 2016, no. September, p. 1, 2016.

[13] U. Kulisch, *Computer arithmetic and validity: theory, implementation, and applications*. Walter de Gruyter, 2013, vol. 33.

[14] W. N. Street, W. H. Wolberg, and O. L. Mangasarian, "Nuclear feature extraction for breast tumor diagnosis," in *Biomedical Image Processing and Biomedical Visualization*, vol. 1905. International Society for Optics and Photonics, 1993, pp. 861–871.

[15] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[16] J. C. Schlimmer, "Concept acquisition through representational adjustment," 1987.

[17] A. Iwata, Y. Yoshida, S. Matsuda, Y. Sato, and N. Suzumura, "An artificial neural network accelerator using general purpose 24 bits floating point digital signal processors," in *IJCNN*, vol. 2, 1989, pp. 171–182.

[18] D. Hammerstrom, "A vlsi architecture for high-performance, low-cost, on-chip learning," in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. IEEE, 1990, pp. 537–544.

[19] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014. [Online]. Available: http://arxiv.org/abs/1412.7024

[20] Y. Bengio, "Deep learning of representations: Looking forward," in *International Conference on Statistical Language and Speech Processing*. Springer, 2013, pp. 1–37.

[21] S. H. F. Langroudi, T. Pandit, and D. Kudithipudi, "Deep learning inference on embedded devices: Fixed-point vs posit," in *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, March 2018, pp. 19–23.

[22] M. Cococcioni, E. Ruffaldi, and S. Saponara, "Exploiting posit arithmetic for deep neural networks in autonomous driving applications," in *2018 International Conference of Electrical and Electronic Technologies for Automotive*. IEEE, 2018, pp. 1–6.

[23] J. Johnson, "Rethinking floating point for deep learning," *arXiv preprint arXiv:1811.01721*, 2018.