A Posit8 Decompression Operator for Deep Neural Network Inference

Orégane Desrentes[1,2], Diana Resmerita[2], and Benoît Dupont de Dinechin[2(✉)]

[1] ENS Lyon, Lyon, France
[2] Kalray S.A, Montbonnot-Saint-Martin, France
bddinechin@kalray.eu

Abstract. We propose a hardware operator to decompress Posit8 representations with exponent sizes 0, 1, 2, 3 to the IEEE 754 binary 16 (FP16) representation. The motivation is to leverage the tensor units of a manycore processor that already supports FP16.32 matrix multiply-accumulate operations for deep learning inference. According to our experiments, adding instructions to decompress Posit8 into FP16 numbers would enable to further reduce the footprint of deep neural network parameters with an acceptable loss of accuracy or precision. We present the design of our decompression operator and compare it to lookup-table implementations for the technology node of the targeted processor.

Keywords: Posit8 · FP16 · Deep learning inference

1 Introduction

Various approaches for reducing the footprint of neural network parameters have been proposed or deployed. Mainstream deep learning environments support rounding of the FP32 parameters to either the FP16 or BF16 representations. They support further reduction in size of the network parameters by applying linear quantization techniques that map FP32 numbers to INT8 numbers [14]. In this paper, we follow an alternate approach to FP32 parameter compression by rounding them to Posit8 numbers. As reported in [4,9], exponent sizes (es) of 0, 1, 2, 3 are useful to compress image classification and object detection network parameters (es=*2* is now the standard for Posit8 [1]).

Unlike previous work [17,19] that apply Posit arithmetic to deep learning inference, we do not aim at computing directly with Posit representations. Rather, we leverage the capabilities of the Kalray MPPA3 processor for deep learning inference [10] whose processing elements implement 4× deep FP16.32 dot-product operators [2,3]. The second version of this processor increases 4× the number of FP16.32 dot-product operators that become 8× deep. As a result, the peak performance for FP16.32 matrix multiply-accumulate operations increases 8×. For this processor, we designed decompression operators that expand Posit8 into FP16 multiplicands before feeding them to the FP16.32 dot-product operators. These Posit8 decompression operators have to be instanced 32 times in

J. Gustafson and V. Dimitrov (Eds.): CoNGA 2022, LNCS 13253, pp. 14–30, 2022.
https://doi.org/10.1007/978-3-031-09779-9_2

order to match the PE load bandwidth of 32 bytes per clock cycle. The corresponding instruction provides the *es* parameter as a two-bit modifier.

In Sect. 2, we evaluate the effects of compressing the IEEE 754 binary 32 floating-point representation (FP32) deep learning parameters to Posit8 representations on classic classification and detection networks, then discuss the challenges of decompressing Posit8 representations to FP16. In Sect. 3, we describe the design of several Posit8 to FP16 decompression operators and compare their area and power after synthesis for the TSMC 16FFC technology node.

2 Compression of Floating-Point Parameters

2.1 Floating-Point Representations Considered

A floating-point representation uses a triplet (s, m, e) to encode a number x as:

$$x = (-1)^s \cdot \beta^{e-bias} \cdot 1.m, \tag{1}$$

where β is the radix, $s \in \{0, 1\}$ is the sign, $m = m_1 ... m_{p-1}$ is the mantissa with an implicit leading bit m_0 set to 1, p is the precision and $e \in [e_{min}, e_{max}]$ is the exponent. The IEEE-754 standard describes binary representations ($\beta = 2$) and decimal representations ($\beta = 10$). Binary representations such as FP32 and FP16 are often used in neural network inference. Let us denote the encoding of x in a representation F with x_F, so we write x_{FP32} and x_{FP16} as follows:

$$x_{\text{FP32}} = (-1)^s \cdot 2^{e-127} \cdot 1.m, \text{ with } p = 24, \tag{2}$$

$$x_{\text{FP16}} = (-1)^s \cdot 2^{e-15} \cdot 1.m, \text{ with } p = 11. \tag{3}$$

A first alternative to IEEE-754 floating-point for deep learning is the BF16 representation, which is a 16-bit truncated version of FP32 with rounding to nearest even only and without subnormals [13]. It has a sign bit, an exponent of 8 bits and a mantissa of 7 bits. A number represented in BF16 is written as:

$$x_{\text{BF16}} = (-1)^s \cdot 2^{e-127} \cdot 1.m, \text{ with } p = 8. \tag{4}$$

A second alternative is the floating-point representation introduced by Microsoft called MSFP8 [8], which is equivalent to IEEE-754 FP16 truncated to 8 bits. This representation has a sign bit, a 5-bit exponent and a 2-bit mantissa:

$$x_{\text{MSFP8}} = (-1)^s \cdot 2^{e-15} \cdot 1.m, \text{ with } p = 3. \tag{5}$$

The third alternative considered are the 8-bit Posit representations [11]. Unlike FP, Posit representations have up to four components: sign, regime, exponent and mantissa. A Posit$n.es$ representation is fully specified by n, the total number of bits and es, the maximum number of bits dedicated to the exponent. The components of a Posit representation have dynamic lengths and are determined according to the following priorities. Bits are first assigned to the sign and the regime. If some bits remain, they are assigned to the exponent and lastly, to the mantissa. The regime is a run-length encoded signed value (Table 1).

Table 1. Regime interpretation (reproduced from [5]).

Binary	0001	001	01	10	110	1110
Regime value	–3	–2	–1	0	1	2

Table 2. Comparison of components and dynamic ranges of representations. Note that the components of Posit numbers have dynamic length. The indicated values of exponent and mantissa for Posit represent the maximum number of bits the components can have. The regime has priority over the mantissa bits.

Repres	FP32	FP16	BF16	MSFP8	Posit8.0	Posit8.1	Posit8.2	Posit8.3
Exponent	8	5	8	5	0	1	2	3
Mantissa	23	10	7	2	5	4	3	2
Regime	–	–	–	–	2–7	2–7	2–7	2–7
Range	83.38	12.04	78.57	9.63	3.61	7.22	14.45	28.89

The numerical value of a Posit number is given by (6) where r is the regime value, e is the exponent and m is the mantissa:

$$x_{\text{Posit}n.es} = (-1)^s \cdot (2^{2^{es}})^r \cdot 2^e \cdot m, \text{ with } p = m. \tag{6}$$

In order to choose a suitable arithmetic representation for a set of values, one needs to consider two aspects: dynamic range and precision. The dynamic range is the decimal logarithm of the ratio between the largest representable number to the smallest one. The precision is the number of bits of the mantissa, plus the implicit one. The total size and exponent size determine the dynamic range of a given representation. Table 2 summarizes the components of the floating-point representations considered along with their dynamic range.

The FP32 representation has a wide dynamic range and provides the baseline for DNN inference. The BF16 representation preserves almost the same dynamic range as FP32, while FP16 has a smaller dynamic range and higher precision than BF16. MSFP8 has almost the same dynamic range as FP16, however it comes with a much reduced precision. Concerning Posit representations, not only they offer tapered precision by distributing bits between the regime and the following fields, but also they present the opportunity of adjusting es to adapt to the needs of a given application. An increase of the es decreases the number of bits available for the fractional part, which in turn reduces the precision.

2.2 Effects of Parameter Compression

We use pre-trained classic deep neural networks, compress their FP32 parameters to FP16 and to each of the alternative floating-point representations, after which we analyse the impact on the results of different classification and detection networks. In the following experiments, all computations are done in FP32,

however we simulate the effects of lower precision by replacing the parameters with the values given by the alternative representations. Conversion from FP32 to BF16 is done by using FP32 numbers with the last 16 bits set to 0. Similarly, for the MSFP8 we use FP16 numbers with last 8 bits are cleared.

Table 3. Classification networks. Compression is applied to all parameters.

DNN	Criterion	FP32	FP16	BF16	MSFP8	Posit			
						8.0	8.1	8.2	8.3
VGG16	ACC-1	70.6	70.6	70.8	69.7	10.2	70.8	70.5	70
	ACC-5	91.3	91.3	91.2	90.3	25.2	91.0	91.0	90
VGG19	ACC-1	70.1	70.1	70.3	67.9	4.8	70.1	69.9	70.6
	ACC-5	90.4	90.4	90.5	89.4	16.3	90	90	90.4
ResNet50	ACC-1	75.7	71.3	75.5	62.8	0.0	27.7	73.2	66
	ACC-5	93.3	90.2	93.5	83.8	0.0	91.4	91.4	88.7
InceptionV3	ACC-1	71.1	71.1	71.3	44.8	65.1	69.4	69.7	63.1
	ACC-5	89.9	89.9	90.0	67.9	86.1	91.0	89.5	85.3
Xception	ACC-1	73.5	73.4	73.6	37.5	70.6	72.4	72.1	63.8
	ACC-5	92.1	92.2	91.7	60.6	90.9	91.4	90.9	86.0
MobileNetV2	ACC-1	71.2	71.2	71	0.2	12.7	12.3	11.0	3.2
	ACC-5	90.0	90.0	89.6	0.6	24.7	25.7	24.4	9.9

Table 4. Detection network. Compression is applied to all parameters.

DNN	Criterion	FP32	FP16	BF16	MSFP8	Posit			
						8.0	8.1	8.2	8.3
YOLO v3	mAP	0.41595	0.41595	0.41585	0.3022	0.4025	0.4155	0.411	0.394

Regarding the Posit representations, even at small size they encode numbers with useful precision and dynamic range. Thus, in our experiments, we evaluate Posit8 representations with *es* between 0 and 3. A dictionary containing the 255 values given by each Posit8 type is first obtained by relying on a reference software implementation [11]. We observe that all Posit8.0 and Posit8.1 values can be represented exactly in FP16. The Posit8.2 representation has 8 values of large magnitude which are not representable in FP16, but can be represented in BF16. For the Posit8.3 representation, 46 values are not representable in FP16 and 12 values are not representable in BF16. In our experiments, compression is done by replacing the parameters with their nearest values in the dictionary.

We experiment with six classification networks and one object detection network. The evaluation criteria are: Accuracy Top 1 (ACC-1), Accuracy Top 5 (ACC-5) for classification and Mean Average Precision (mAP) for detection.

Table 3 contains the results for the classification networks VGG16 [23], VGG19, ResNet50 [12], InceptionV3 [24], Xception [7], MobileNetV2 [22], which have different architectures. We also display the results obtained with FP32 and FP16 in order to compare with the standard floating-point representations. The mAP results for the object detection network (YOLO v3 [21]) are shown in Table 4.

Overall, compression with BF16 gives better results than with FP16. Despite its lower precision than FP16, BF16 appears to be well suited to deep neural network inference. On the other hand, the reduced precision of MSFP8 leads to a significant loss of performance for all tested networks. For the Posit8.0 and Posit8.3 representations, a significant loss of performance is also observed in both conventional classification (VGG16) and detection networks (YOLO).

Table 5. Classification networks. Compression is only applied to parameters of convolutions and of fully connected operators.

DNN	Criterion	FP32	Posit			
			8.0	8.1	8.2	8.3
ResNet50	ACC-1	75.7	71.3	75.0	75	73.6
	ACC-5	93.3	9.8	92.7	92.8	92.6
InceptionV3	ACC-1	71.1	66.0	70.9	70.1	69.9
	ACC-5	89.9	86.8	90.7	89.1	88.5
Xception	ACC-1	73.5	72.1	72.6	72.8	68.8
	ACC-5	92.1	91.3	91.7	91.3	89.4
MobileNetV2	ACC-1	70.8	25.3	53.5	52.7	39.4
	ACC-5	89.8	47.0	76.9	77.3	63.1

For networks containing normalization operators (ResNet50, InceptionV3, Xception and MobileNetV2), the loss of performance is significant on at least one of the Posit8 representation. This motivates a second round of experiments on the four the networks that have batch normalization operators. As reported in Table 5, not compressing the parameters of the batch normalization operators improves the performance on all these networks. However, despite the improvement, MobileNetV2 remains with a significant accuracy loss.

To summarize the effects of parameter compression to 8-bit representations, using Posit8.es with $0 \leq es \leq 3$ appears interesting. We expect that accuracy and precision could be further improved by selecting the compression representation (none, FP16, Posit8.es) individually for each operator. This motivates the design and implementation of a Posit8.es to FP16 decompression operator.

2.3 Decompression Operator Challenges

Previous implementations of Posit operators [6,15,16,18,20,25–27] include a decompressing component, often called data extraction or decoding unit, that

transforms a Posit number into an internal representation similar to a floating-point number of non-standard size, into which the Posit can be exactly represented. While the structure of these units provides inspiration for our work, the design of our decompression operator faces new challenges.

- Unlike Posit operator implementations of previous work which support a single Posit representation after synthesis, our decompression operator receives the exponent size from the instruction opcode. Support of variable *es* is interesting for the MPPA3 processor as its deep learning compiler may adapt the number representation of each tensor inside a network in order to provide the best classification accuracy or detection precision.
- Posit numbers have a symmetric representation with respect to the exponent, unlike IEEE 754 floating-point that has an asymmetry tied to gradual underflow, so previous works do not deal with subnormal numbers. Support of subnormal numbers is important for the decompression of the Posit8.2 and Posit8.3 representations to FP16. The smallest Posit8.2 numbers are exactly represented as FP16 subnormals, ensuring the conversion does not underflow. Although the decompression of the Posit8.3 representation to FP16 may underflow, the range added by gradual underflow is crucial for networks that compute with FP16 parameters.
- For the Posit8 numbers with $es \in \{2, 3\}$ that are not exactly representable in FP16, our decompression operator supports the four IEEE 754 standard rounding modes (Round to Nearest Even, Round Up, Round Down and Round to Zero) by extrapolating the flag setting convention described by IEEE 754 standard when narrowing to a smaller representations.
- For these Posit8.*es* number that cannot be exactly represented in FP16, our decompression operator also has to raise the IEEE 754 overflow or underflow flags. Likewise, the Not a Real (NaR) value cannot be expressed in the IEEE 754 standard, so this conversion should raise and invalid flag and return a quiet Not a Number (NaN) [1].

Lemma 1. *With the Posit*8.2 *and Posit*8.3 *representations, knowing the regime is sufficient to pre-detect conversion underflow or overflow to FP16.*

Proof. Let us call w_E the width of the floating-point exponent. For FP16 $w_E = 5$, so maximum exponent value is $e_{max} = 2^{w_E-1} - 1 = 15$. Similarly, the smallest possible exponent (counting subnormals) is $e_{min\_sn} = -2^{(w_E-1)} + 1 - w_M + 1 = -24$ (where w_M is the width of the mantissa).

The Posit$n$ representation combined exponent (in the floating point sense) is $c = r \times 2^{es} + e$ where r is the regime, with $-n+1 \leq r \leq n-2$, and e is the posit exponent $0 \leq e \leq 2^{es} - 1$. A Posit number overflows the FP16 representation when $c > e_{max}$ i.e. $r \times 2^{es} + e \geq e_{max} + 1$. This can be detected irrespective of e when $e_{max} + 1$ is a multiple of 2^{es}. This is the case when $es \leq 4$.

A Posit number underflows the FP16 representation when $c < e_{min\_sn}$. Since $c = r \times 2^{es} + e < 0$ and $e \geq 0$, then we need for the detection that $e_{min\_sn} - 1$ is a value with the maximum e, which is $2^{es} - 1$. This can be detected irrespective of e when e_{min_sn} is a multiple of 2^{es}. This is the case when $es \leq 3$.

By application of Lemma 1, Posit8.2 overflows FP16 when regime ≥ 4, while Posit8.3 overflows when regime ≥ 2 and underflows when regime ≤ -4.

Lemma 1 still holds for other values of n if the goal is to compute the IEEE 754 overflow flag, however it does not work as well for the underflow as this can only detect the underflow to zero, and not the loss of significand bits in a subnormal result. The Posit8.3 representation works here since it is small enough to not have significand bits when its value is converted to a small FP16 subnormal. The regime r should be large enough to imply no mantissa bits, i.e. $1 + r + 1 + es \geq n$, for the r corresponding to the small FP subnormals.

For example, with the FP32 representation $e_{max} = 127$ so Lemma 1 applies for conversion overflow pre-detection if $es \leq 6$. For the FP32 underflow pre-detection, $e_{min\_sn} = 154 = 2 \times 77$ so Lemma 1 does not apply in case $es > 1$.

3 Design and Implementation

3.1 Combinatorial Operator Design

Our first Posit8 decompression operator design is combinatorial (Fig 1), with steps similar to those of previously proposed Posit hardware operators. First the two's complement of the Posit number is computed when its sign bit is 1. The regime is then decoded with a leading digit counter combined with a shifter. Since the maximum exponent size es is variable, three more small shifters (Fig. 2) are used to separate the Posit exponent field e from the mantissa and to combine e with the regime value in binary to compute the unbiased FP16 exponent. The FP16 bias is then added, and the three parts of the FP16 number are combined. Additional details are needed to decompress special Posit8 numbers, or when a Posit value overflows or underflows the FP16 representation.

The special cases of Posit8 decompression to FP16 are pre-detected in the operator. Testing for Posit zero and Posit NaR are done in parallel and return floating-point zero or floating-point NaN (with the inexact flag) at the end of the operator, adding two multiplexers to the end of the operator. Those were not drawn to save space. Similarly, since all the other flag values are coded into tables, they are not drawn on the operator and not described in the figures. A few one-bit multiplexers are added to set the IEEE 754 flags.

Lemma 1 enables to check a table as soon as the regime is known to determine if the value will overflow or underflow FP16. The rest of the exponent construction is done in parallel with the table lookup. Moreover, since the mantissa of Posit8 is always smaller than the mantissa of FP16, there is no precision lost in the normal and subnormal cases. However, the rounding mode may change the values returned when there is a conversion underflow or an overflow.

The special case table (Fig. 3) outputs four different values: ∞ (0x7C00), Ω (0x7BFF), smallest subnormal (SN) (0x0001) and zero (0x0000). The sign is concatenated after the table. Two extra bits are used, the first to encode if the value in the table should be used, and the second to know if this is an overflow or an underflow, in order to use this for the flags.

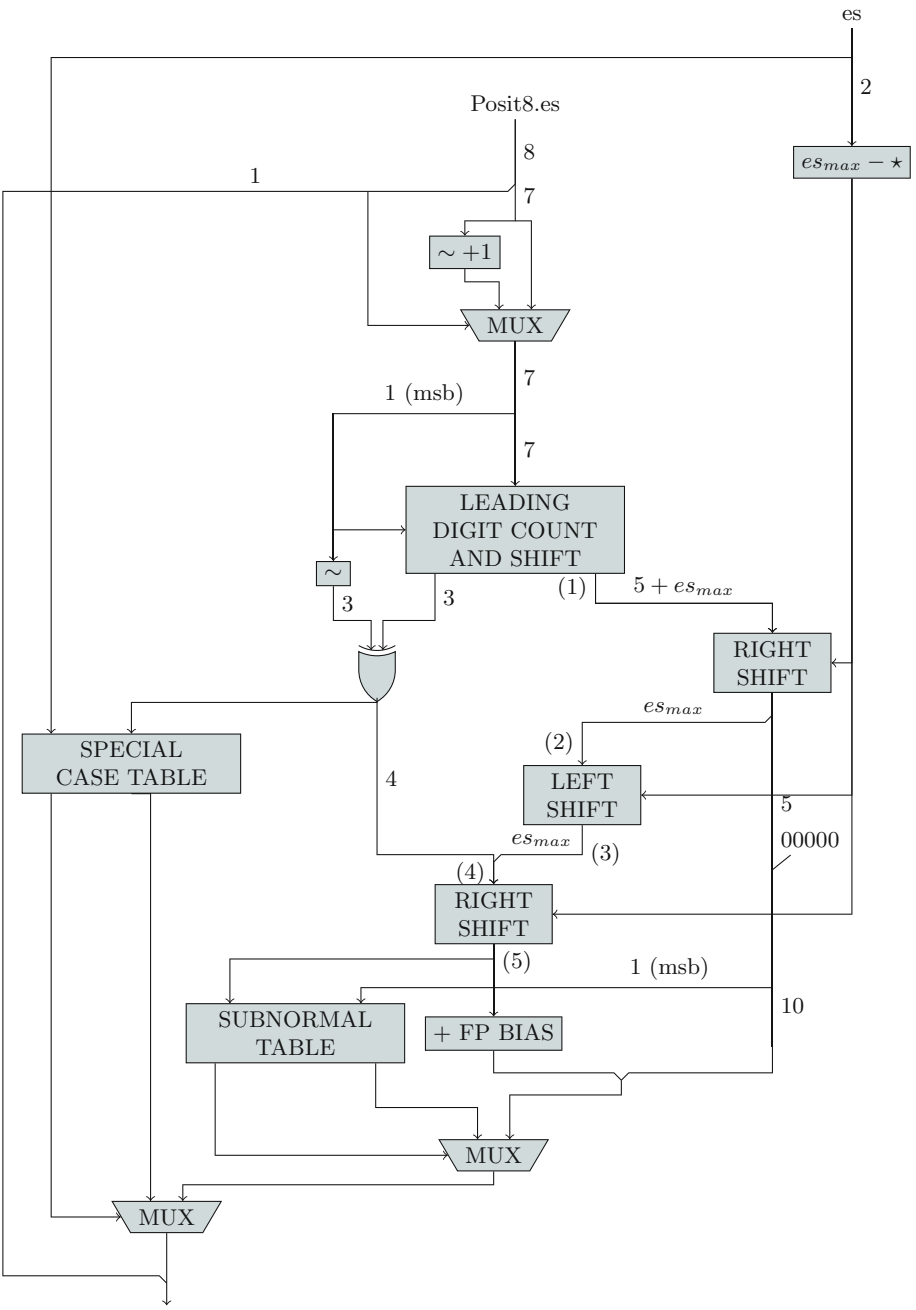

Fig. 1. Posit8 to floating-point 16 decompression operator.

The table for subnormal values (Fig. 4) is used as soon as the unbiased FP16 exponent in known. For the Posit8.2 representation, the numbers that convert to subnormals do not have any mantissa bits. For Posit8.3 however, this may happen in two cases, so the most significant bit of the mantissa is input to the table. The values returned by the table are always exact, so no inexact or underflow flags are needed. This table does not output the sign but instead a bit indicating if the value of the table should be used or not.

3.2 General Operator Efficiency

The synthesis are done with Synopsys Design Compiler NXT for the TSMC 16FFC node. We compare our combinatorial operator implementation to a baseline lookup-table and track the area (Fig. 5) and power (Fig. 6) according to the operating frequency. A pipelined version of this operator is obtained by adding a stage between the leading digit count and the first shift. This enables the use

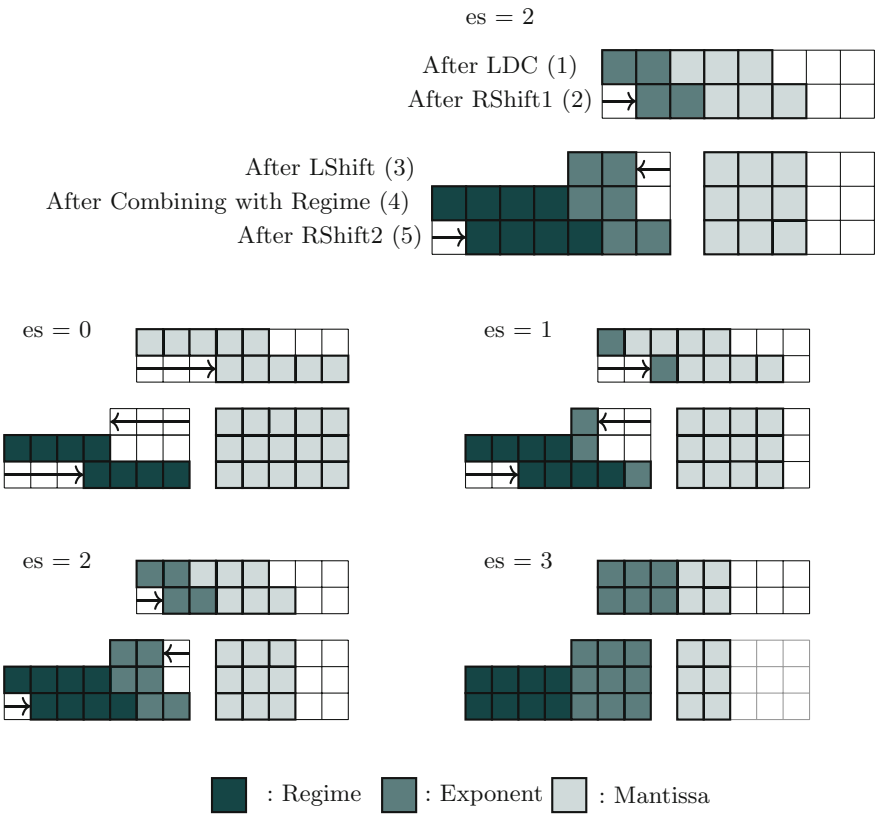

Fig. 2. Shifters to separate exponent and mantissa

sign	es	regime	rounding mode	output
0	2	4 to 6	10 or 11 (RD or RZ)	10&Ω
0	2	4 to 6	00 or 01 (RN or RU)	10&∞
1	2	4 to 6	01 or 11 (RU or RZ)	10&Ω
1	2	4 to 6	00 or 10 (RN or RD)	10&∞
0	3	2 to 6	10 or 11 (RD or RZ)	10&Ω
0	3	2 to 6	00 or 01 (RN or RU)	10&∞
1	3	2 to 6	01 or 11 (RU or RZ)	10&Ω
1	3	2 to 6	00 or 10 (RN or RD)	10&∞
0	3	-4 to -6	01 (RU)	11&SN
0	3	-4 to -6	else	11&0
1	3	-4 to -6	10 (RD)	11&SN
1	3	-4 to -6	else	11&0
*	*	*	*	00&0

Fig. 3. Table for the special values

exponent	mantissa msb	output
0	0	1 & 000 0010 0000 0000
0	1	1 & 000 0011 0000 0000
-1	0	1 & 000 0001 0000 0000
-1	1	1 & 000 0001 1000 0000
-2	0	1 & 000 0000 1000 0000
-3	0	1 & 000 0000 0100 0000
-4	0	1 & 000 0000 0010 0000
-5	0	1 & 000 0000 0001 0000
-6	0	1 & 000 0000 0000 1000
-7	0	1 & 000 0000 0000 0100
-8	0	1 & 000 0000 0000 0010
-9	0	1 & 000 0000 0000 0001
*	*	0 & 000 0000 0000 0000

Fig. 4. Table for the subnormal values

of a faster clock and reduces the area, the trade-off being that the conversion takes 2 clock cycles. The throughput is still of one value per clock cycle.

The lookup table implementation uses as inputs {Posit, *es*, rounding mode} so it has $2^{12} = 4096$ entries. The Posit sign is needed to compute the overflow and underflow IEEE 754 flags in Round Up and Round Down modes.

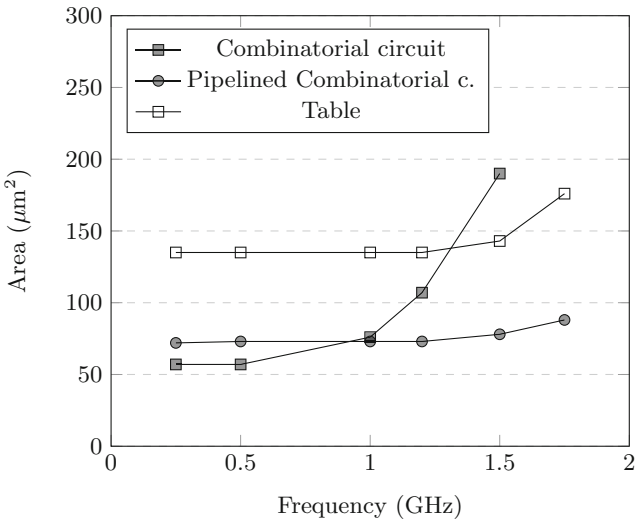

Fig. 5. Area depending on frequency for the general operators

At the target frequency of our processor (1.5 GHz), the pipelined combinatorial implementation has $\frac{2}{3}$ the area and half the leakage power of a baseline table-based implementation. It also has a lower dynamic power consumption.

3.3 Specialized Operator Efficiency

As discussed earlier, the rounding mode has no effects on the decompression for the majority of Posit8 numbers, and may only change the result in cases of overflow or underflow. This motivates specializing the operators to convert Posit8 to FP16 representations for the Round-to-Nearest (RN) mode only.

The combinatorial decompression operators are built the same way as before, the only changes being on the special case table which is significantly reduced since the rounding mode and sign no longer intervene, while the number of possible outputs is halved.

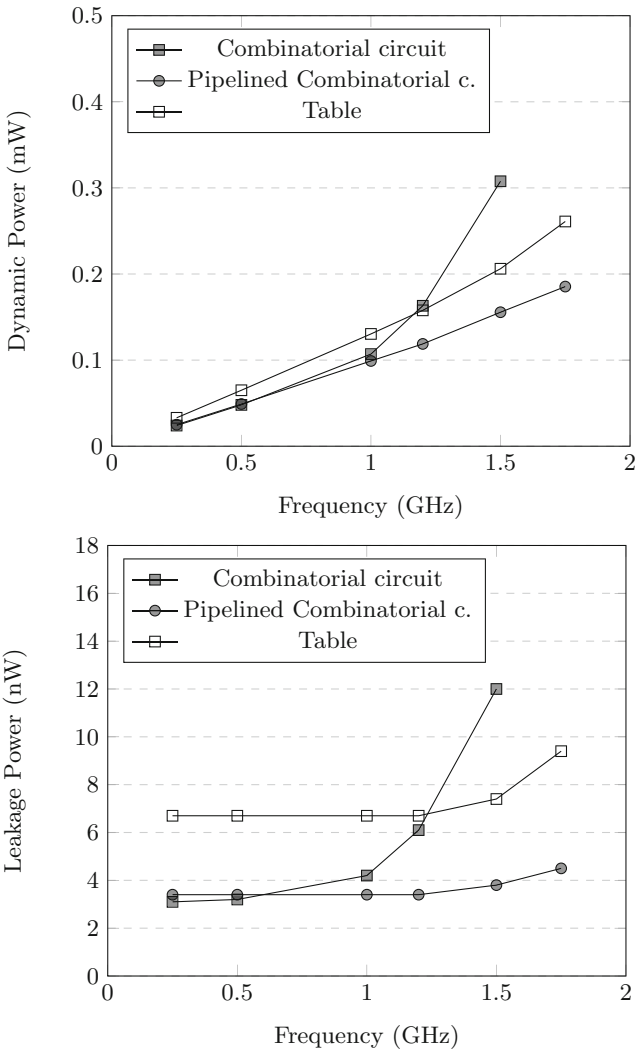

Fig. 6. Power depending on frequency for the general operators

The baseline table implementation now uses inputs {Posit, es} so it has $2^{10} = 1024$ entries. Moreover, as the sign is no longer needed for producing the IEEE 754 overflow and underflow flags, the table implementation can be factored as illustrated in Fig. 7. If the Posit is negative, its two's complement is used for accessing the table, and the sign is appended to the output of the table.

Those specialized operators show similar result in area (Fig. 8) and leakage power (Fig. 9). The factored table implementation however is significantly better

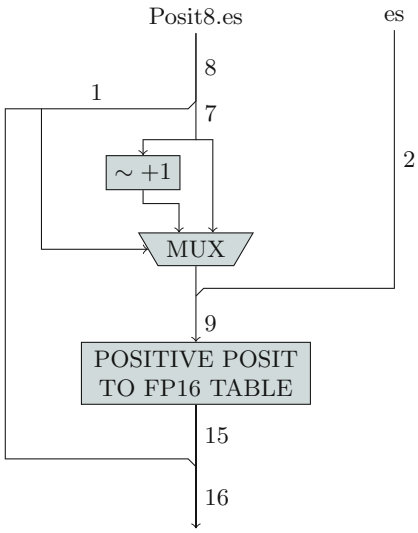

Fig. 7. Posit8 to FP16 decompression operator based on a factored table.

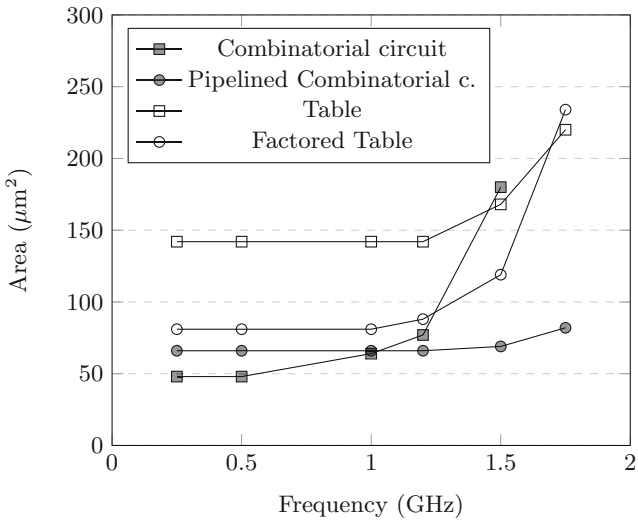

Fig. 8. Area depending on frequency for the RN only operators

than the baseline table as it exploits a symmetry that was not apparent. At our target frequency, our pipelined operator implementation has $\frac{5}{6}$ the area and $\frac{2}{3}$ the leakage power of the optimised table-based implementation, and has lower dynamic power consumption (Fig. 9).

Another specialisation of interest is to only decompress the standard Posit8.2 representation. This further reduces the size (Fig. 10) and the power consumption (Fig. 11) of the combinatorial implementations since it simplifies both the special tables and the small shifters. This operator is also smaller than the table-based implementations. It is interesting to note that for tables this small, at high

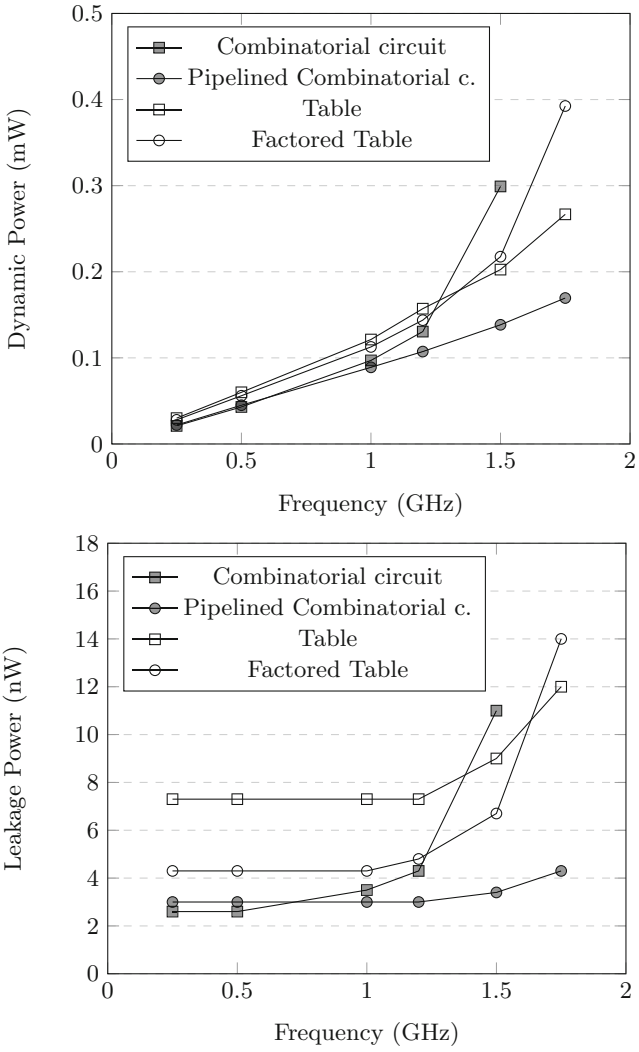

Fig. 9. Power depending on frequency for the RN only operators

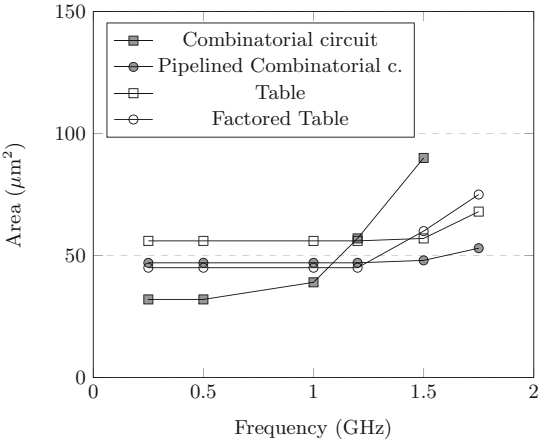

Fig. 10. Area depending on frequency for the RN and Posit8.2 only operators

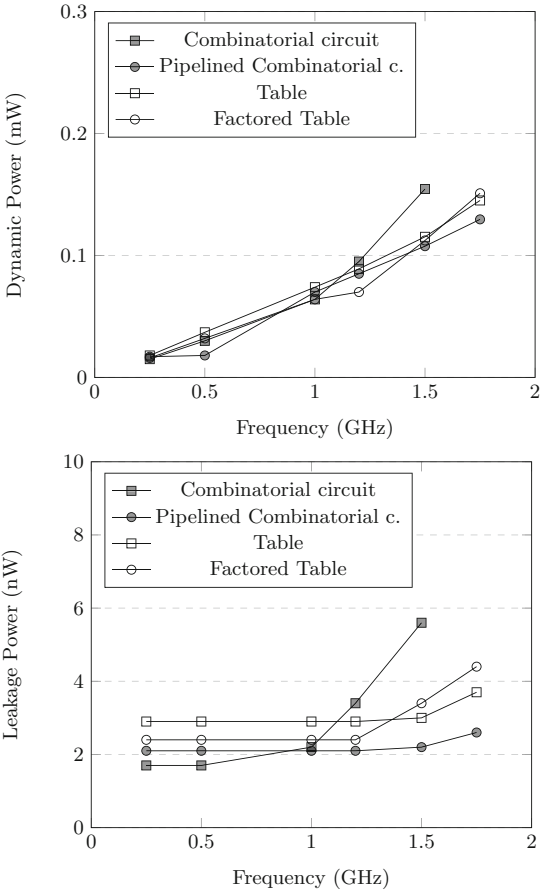

Fig. 11. Power depending on frequency for the RN and Posit8.2 only operators

frequencies the overhead of computing the absolute values may offset the benefit of halving the table size.

4 Summary and Conclusion

This paper proposes to use 8-bit floating-point representations for the compression of the IEEE 754 binary 32 (FP32) parameters of trained deep learning classification and detection networks. Traditional compression of FP32 parameters for inference rounds them to the IEEE 754 binary 16 (FP16) or to BF16 representations, where they are used as multiplicands before accumulation to FP32 or wider representations. Assuming that efficient mixed-precision FP16.32 matrix multiply-add operators are available, our objective is to select 8-bit representations suitable for floating-point parameter compression and to design the corresponding decompression operators to the FP16 representation.

We first observe that compressing parameters from FP32 to MSFP8 (a FP16 representation truncated to 8 bits proposed by Microsoft) does not give acceptable inference results for the networks considered. Indeed, to achieve compression of the FP32 parameters without significant accuracy loss, a trade-off between the dynamic range and the precision is needed. Accordingly, parameter compression to the Posit8.1, Posit8.2 and Posit8.3 representations performs well for inference with the tested networks, with a few exceptions.

We then design and implement combinatorial and table-based Posit8 to FP16 decompression operators with increasing degrees of specialization. The combinatorial designs benefit from an insight on the conditions which leads to overflow or underflow when converting Posit8.2 or Posit8.3 to FP16. This enables to predetect those conditions by inspecting only the Posit regime bits.

The most general decompression operators presented receive as input a Posit8 value, the exponent size $0 \leq es \leq 3$, and one of the four IEEE 754 rounding modes. A first specialization considers only rounding to the nearest even, which in turn enables the table-based implementation to be factored relative to the sign. A second specialization only decompresses the standard Posit8 representation, whose exponent size is 2. In all cases, the pipelined version of our combinatorial decompression operator appears as the best option.

References

1. Standard for Posit Arithmetic (2022) Release 5.0
2. Brunie, N.: Modified fused multiply and add for exact low precision product accumulation. In: 24th IEEE Symposium on Computer Arithmetic, ARITH 2017, London, United Kingdom, pp. 106–113, July 2017
3. Brunie, N.: Towards the basic linear algebra unit : replicating multi-dimensional FPUs to accelerate linear algebra applications. In: 2020 54th Asilomar Conference on Signals, Systems, and Computers, pp. 1283–1290 (2020)
4. Carmichael, Z., Langroudi, H.F., Khazanov, C., Lillie, J., Gustafson, J.L., Kudithipudi, D.: Performance-efficiency trade-off of low-precision numerical formats in deep neural networks. In: Proceedings of the Conference for Next Generation Arithmetic 2019, pp. 3:1–3:9. CoNGA 2019, ACM, New York (2019)

5. Carmichael, Z., Langroudi, H.F., Khazanov, C., Lillie, J., Gustafson, J.L., Kudithipudi, D.: Deep positron: a deep neural network using the posit number system. In: DATE, pp. 1421–1426. IEEE (2019)
6. Chaurasiya, R., et al.: Parameterized posit arithmetic hardware generator. In: 2018 IEEE 36th International Conference on Computer Design (ICCD), pp. 334–341. IEEE (2018)
7. Chollet, F.: Xception: Deep learning with depthwise separable convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1251–1258 (2017)
8. Chung, E., et al.: Serving dnns in real time at datacenter scale with project brainwave. IEEE Micro **38**, 8–20 (2018)
9. Cococcioni, M., Rossi, F., Ruffaldi, E., Saponara, S., de Dinechin, B.D.: Novel arithmetics in deep neural networks signal processing for autonomous driving: challenges and opportunities. IEEE Sig. Process. Mag. **38**(1), 97–110 (2020)
10. de Dinechin, B.D.: Consolidating high-integrity, high-performance, and cybersecurity functions on a manycore processor. In: 56th ACM/IEEE Design Automation Conference (DAC 2019), p. 154 (2019)
11. Gustafson, J.L., Yonemoto, I.T.: Beating floating point at its own game: posit arithmetic. Supercomput. Frontiers Innov. **4**(2), 71–86 (2017)
12. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of IEEE conference on CVPR, pp. 770–778 (2016)
13. Intel: BFLOAT16 - Hardware Numerics Definition Revision 1.0, November 2018
14. Jacob, B., et al.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: 2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, pp. 2704–2713, June 2018
15. Jaiswal, M.K., So, H.K.H.: Universal number posit arithmetic generator on FPGa. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1159–1162. IEEE (2018)
16. Jaiswal, M.K., So, H.K.H.: Pacogen: a hardware posit arithmetic core generator. IEEE access **7**, 74586–74601 (2019)
17. Lu, J., Fang, C., Xu, M., Lin, J., Wang, Z.: Evaluations on deep neural networks training using posit number system. IEEE Trans. Comput. **70**(2), 174–187 (2020)
18. Murillo, R., Del Barrio, A.A., Botella, G.: Customized posit adders and multipliers using the FloPoCo core generator. In: 2020 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5. IEEE (2020)
19. Murillo, R., Del Barrio, A.A., Botella, G.: Deep pensieve: a deep learning framework based on the posit number system. Digital Signal Process. **102**, 102762 (2020)
20. Podobas, A., Matsuoka, S.: Hardware implementation of POSITs and their application in FPGAs. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 138–145. IEEE (2018)
21. Redmon, J., Farhadi, A.: Yolov3: an incremental improvement. CoRR abs/1804.02767 (2018)
22. Sandler, M., Howard, A.G., Zhu, M., Zhmoginov, A., Chen, L.: Mobilenetv 2: inverted residuals and linear bottlenecks. In: CVPR, pp. 4510–4520. IEEE Computer Society (2018)
23. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: 3rd ICLR (2015)
24. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2818–2826 (2016)

25. Uguen, Y., Forget, L., de Dinechin, F.: Evaluating the hardware cost of the posit number system. In: 2019 29th International Conference on Field Programmable Logic and Applications (FPL), pp. 106–113. IEEE (2019)
26. Xiao, F., Liang, F., Wu, B., Liang, J., Cheng, S., Zhang, G.: Posit arithmetic hardware implementations with the minimum cost divider and squareroot. Electronics **9**(10), 1622 (2020)
27. Zhang, H., He, J., Ko, S.B.: Efficient posit multiply-accumulate unit generator for deep learning applications. In: 2019 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5. IEEE (2019)