Exact Dot Product Accumulate Operators for 8-bit Floating-Point Deep Learning

Orégane Desrentes, Benoît Dupont de Dinechin, Julien Le Maire

Exact Dot Product Accumulate Operators for 8-bit Floating-Point Deep Learning

Orégane Desrentes*†, Benoît Dupont de Dinechin* and Julien Le Maire*
*Kalray S.A., Montbonnot, France, †Univ Lyon, INSA Lyon, Inria, CITI, France
Email:{odesrentes, bddinechin, jlemaire}@kalrayinc.com

***Abstract*—Low bit-width floating-point formats appear as the main alternative to 8-bit integers for quantized deep learning applications. We propose an architecture for exact dot product accumulate operators and compare its implementation costs for different 8-bit floating-point formats: FP8 with five exponent bits and two fraction bits (E5M2), FP8 with four exponent bits and three fraction bits (E4M3), and Posit8 formats with different exponent sizes. The front-ends of these exact dot product accumulate operators take 8-bit multiplicands, expand their full-precision products to fixed-point, and sum terms into wide accumulators. The back-ends of these operators round down the wide accumulators contents first to FP32 and then to one of the 8-bit floating-point formats. We synthesize the proposed 8-bit floating-point exact dot product accumulate operators targeting the TSMC 16FFC node and compare their area and power to a baseline of operators with FP16 and INT8 multiplicands.**

***Index Terms*—FP8, Posit8, Deep Learning.**

I. Introduction

Advances in low bit-width data representations for deep learning indicate that quantization using 8-bit floating-point formats may be preferable to using 8-bit integers. In case of inference, 8-bit integer quantization requires post-training quantization (PTQ) and sometimes quantization-aware training (QAT) to maintain acceptable accuracy [1]. Even with PTQ and QAT, using 8-bit integer quantization on models such as MobileNet remains problematic [2]. By comparison, the wider dynamic range of 8-bit floating-point formats allows to directly quantize a pre-trained model down to 8 bits without losing accuracy by fine-tuning batch normalization statistics [2]. Moreover, the 8-bit floating-point formats appear suitable for training, provided different bit allocations for exponent and fraction [2] and/or different exponent biases [3], [4] are used for the activations, weights, gradients, and accumulations.

In this paper, we compare the hardware implementation costs of different 8-bit floating-point formats for the matrix multiply-accumulate (GEMM) operations of deep learning applications. These formats are: E5M2 (5-bit exponent, 2-bit fraction), E4M3 [1] (4-bit exponent, 3-bit fraction) and the Posit8 representations [5]. Although E5M2 was first considered as the IEEE 754 8-bit binary floating-point format of choice (called msfp8 in [6]), major industry players [7], [1] propose to include one of the E4M3 representations as a FP8 standard. For comparison, we also include FP16 and INT8 among the input and output formats evaluated.

The remainder of this paper is organized as follows. We discuss the state of the art of low bit-width data representations for deep learning in Section II, which motivates our work. Section III describes the architecture of our exact dot product accumulate operators, with focus on the features of the wide accumulator. Section IV presents our synthesis results. We summarize our findings and conclude in Section V.

II. State of the Art

A. Floating-Point Representations

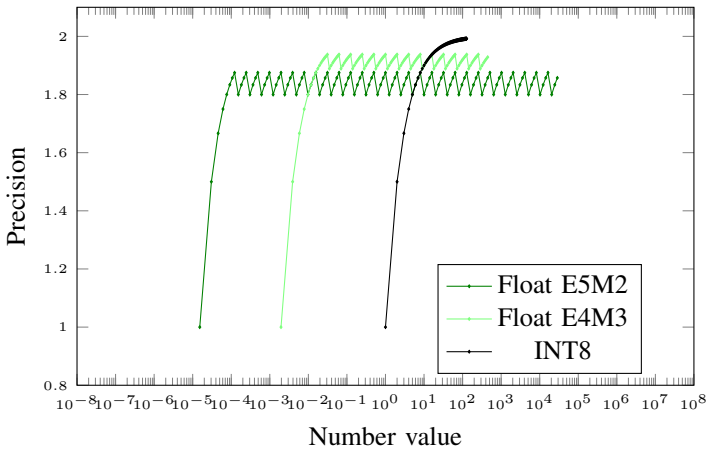
Most investigations of 8-bit floating-point formats for deep learning focus on variants of the IEEE 754 binary representation, which encodes real number values with a sign bit S, an integer exponent E on e bits and a fraction M on m bits as:

$$x = \begin{cases} (-1)^S \times 2^{E-B}(1 + \frac{M}{2^m}) & \text{if some } e \text{ bits differ} \\ (-1)^S \times 2^{1-B}\frac{M}{2^m} & \text{if all } e \text{ bits are } 0 \\ (-1)^S \text{ Inf or NaN} & \text{if all } e \text{ bits are } 1 \end{cases}$$

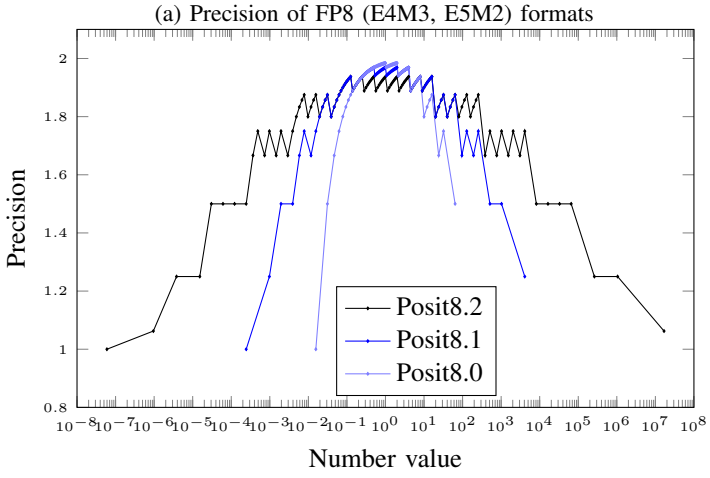
When the implicit bit is concatenated to the fraction, it is called the significand. Here, B is the bias of the exponent set to $2^{e-1} - 1$ in the IEEE 754 binary floating-point representations. When $E = 0$, there is no implicit 1 in the significand so zero and subnormal values can be represented.

Among the IEEE 754 derived 8-bit floating-point formats, the most promising identified are those with $e = 5, m = 2$, and $e = 4, m = 3$ [1], [2], [4]. While the 5-bit exponent format definition follows the IEEE 754 rules, depending on authors the details of the 4-bit exponent format definitions differ:

- [1] A format with e exponent bits and m fraction bits is denoted EeMm. Format E5M2 has bias $B = 15$, NaNs and Infs follow the IEEE 754 rules. Format E4M3 has bias $B = 7$, NaNs = $s.1111.111_2$ and no Infs.
- [4] A format with e exponent bits and m fraction bits is denoted 1.e.m. Formats 1.5.2 and 1.4.3 use the IEEE 754 codeword for -0 as Inf/NaN and reuse the code space of the all-ones exponent. Biases are adjusted per tensor.
- [2] Formats 1.5.2 and 1.4.3 follow the IEEE 754 rules, except that format 1.4.3 has bias $B = 4$ (instead of 7).
- [7] A format with e exponent bits, m fraction bits and bias value B is denoted mMeEBB. This allows the bias to be adapted per-channel (weights) or per-tensor (activations), as well a be fixed for a whole network.
- [3] Two FP8 formats similar to E5M2 and E4M3 called CFloat8_1_5_2 and CFloat8_1_4_3 are introduced, without encoding for Inf or Nan. The bias is configurable and encoded separately as a six-bit unsigned integer.



(a) Precision of FP8 (E4M3, E5M2) formats



(b) Precision of Posit8 (*es* = 0, 1, 2) formats.

Fig. 1: Precision of FP8 (E4M3, E5M2), INT8 and Posit8 (*es* = 0, 1, 2) formats. Precision is the absolute difference between two successive values divided by the one of largest magnitude.

In all cases, the subnormal IEEE 754 numbers remain encoded in the lowest binade ($E = 0$), as they allow a graceful representation of the values close to zero.

The posit representation encodes a real number value x with a sign $S \in \{0, 1\}$, a regime value R, an exponent value $E \in [0, 2^{es} - 1]$ and a fraction value $F \in [0, 1)$ as:

$$x = \begin{cases} 0 & \text{if } S = 0 \text{ and all other bits are } 0 \\ \text{NaR} & \text{if } S = 1 \text{ and all other bits are } 0 \\ ((1 - 3S) + F) \times 2^{(1-2S)\times(2^{es}\times R + E + S)} & \\ & \text{otherwise} \end{cases}$$

The regime value R is encoded in a bit field of $k > 0$ identical bits and a bit whose value differs from the previous one. If the first regime bit is zero, then $R = -k$ else $R = k - 1$. The exception value is called NaR (Not a Real). Given that the minimum number of regime bits is two, while the exponent field takes *es* bits before any fraction bits are encoded, the maximum number of fraction bits for the Posit8.*es* representations is $5 - es$.

The posit encoding is easier to understand by first considering case $S = 0$. For the bit patterns different from 0 and from NaR, then $x = useed^R \times 2^E \times (1+F)$, where $useed = 2^{2^{es}}$, so the exponent is actually $2^{es} \times R + E$. In case $S = 1$, one may take the two's complement of the posit bit pattern, decode it as in case $S = 0$, then negate the resulting value. This procedure is typically implemented in hardware to produce the internal floating-point representation of a posit number [8].

The posit standard [5] requires the support of a wide fixed-point accumulator in two's complement format called the *quire* that enables the exact accumulation of posit products. For n-bit multiplicands, the quire provides one sign bit, 31 guard bits, $8n - 16$ integer bits and $8n - 16$ fractional bits for a total of $16n$ bits. The quire representation also encodes NaR with the most significant bit set and all the other bits cleared.

B. Suitability of 8-bit Quantization for Deep Learning

The benefits of FP8 compared to INT8 quantization are explained by the non-linear sampling of FP8 values [1], which is a better match for Gaussian-like distributions that have more density around zero [7]. In the setting of inference with Post-Training Quantization (PTQ), [7] concludes that many networks benefit from the FP8 formats provided their fraction size and bias values can be set per channel. In cases the bias can be adapted per-channel (weights) or per-tensor (activations), formats 5M2E, 4M3E (and in one case 3M4E) perform best for PTQ. When the bias is fixed, formats 4M3E3B, 3M4E9B and 3M4E8B perform best.

Presumably, the benefits of these FP8 representations in deep learning could be realized by other 8-bit floating-point representations not derived from IEEE 754, including log formats [9] and posit formats [10]. Among these, the standard Posit8 representation [5] whose *es* = 2 appears especially interesting [11], as it combines a dynamic range larger than the E5M2 representation with a precision comparable to the E4M3 representation around zero (Fig. 1). Previous work also confirmed that Posit8.2 performs significantly better than msfp8 (E5M2), Posit8.0 and Posit8.1 on detection and classification networks, while Posit8.3 performed similarly to Posit8.2 [12]. However, Posit8.0 and Posit8.1 arithmetic is less expensive to implement than Posit8.2 arithmetic, so we also include these formats in the scope of our study.

For the applications of posit arithmetic to deep learning, work by [13] is among the first to use a Posit8 format for training. However, this work relies on an early version of the posit standard where the *es* parameter is set respectively to 0, 1, 2 for the posit bit sizes of 8, 16, 32. The work of [11] describes training with Posit8 and Posit16 formats, in both cases with *es* = 1. Work by [14] compares the inference accuracy for Post-Training Quantisation (PTQ) using the bfloat8 format (E5M2) and the Posit8 format with *es* = 0.

Posit8.0 arithmetic presents satisfactory inference results on smaller networks [14] with QAT. However, using the cheaper INT8 arithmetic for LeNet-5 on the MNIST data set results in a very similar network accuracy [15].

C. Matrix Multiply-Accumulate Operations

In previous work, matrix multiply-accumulate operations have been realized in different ways:

- Accumulate the outer-product of row and column vectors, as in the IBM POWER 10 matrix-multiply assist (MMA) facility [16] or the Armv9 scalable matrix extensions (SME) [17]. The elementary operations performed on each matrix element are fused multiply-add, possibly in mixed-precision [18], so a rounding step is implied at each product accumulation.
- While performing a dot product operation, align/truncate products, accumulate and round to the accumulator precision. The Intel Nervana Neural Network Processor-T [19] implements a 32-product BF16 dot product operator that accumulates to FP32. Its products are aligned to the maximum product exponent then truncated to an internal 37-bit datapath before being summed. A similar operator with up to 32 products is presented in [20], whose design focuses on the maximum exponent and global alignment computations which become a bottleneck for large numbers of products. Reverse-engineering of the NVIDIA V100 GPU tensor cores also reveals they adopted a similar term truncation approach [21].
- Sum a specific number of products in full precision, accumulate and round to the accumulator precision. This exact fused approach called chunk-based accumulation [22] is applicable to two-term dot products in cases of floating-point formats that have a wider exponent than FP16 [23], [24]. Conversely, using FP16 or representations with smaller exponent ranges for the multiplicands enables summing more than two full-precision terms in fixed-point before rounding back to an accumulator floating-point representation [25]. This approach is implemented in the FP16.32 matrix multiply-accumulate operators of the Kalray MPPA3 tensor coprocessors [26], [12].
- Accumulate the full precision products into a fixed-point accumulator wide enough to eliminate all rounding errors. Kulisch suggested using a sign-magnitude representation for the exact product accumulator [27], but recent literature [28], [20], [19], [25] motivates the choice of fixed-point two's complement representation for the w-bit wide datapath before feeding the products into a carry-save compression tree. This is similar to the quire required by the posit standard [5] for summing products. However hardware support for the posit quire is not very common and previous work in this area only implements the accumulation of one product at a time [28], [29].

In the area of posit arithmetic, Carmichael et al. [8] propose Exact Multiply-and-Accumulate (EMAC) operators for floating-point, fixed-point and Posit$n$ with configurable *es*. Cococcioni et al. [30] describe the hardware implementation of a posit processing unit (PPU) inside the Ariane RISC-V core that supports the Posit16 and Posit8 formats with *es* = 2. The CVA6 RISC-V core, which is an evolution of the Ariane RISC-V core maintained by the OpenHW Group, is augmented with a Posit32 arithmetic unit (PAU) that includes a quire accumulator and operates on 32 posit registers [29].

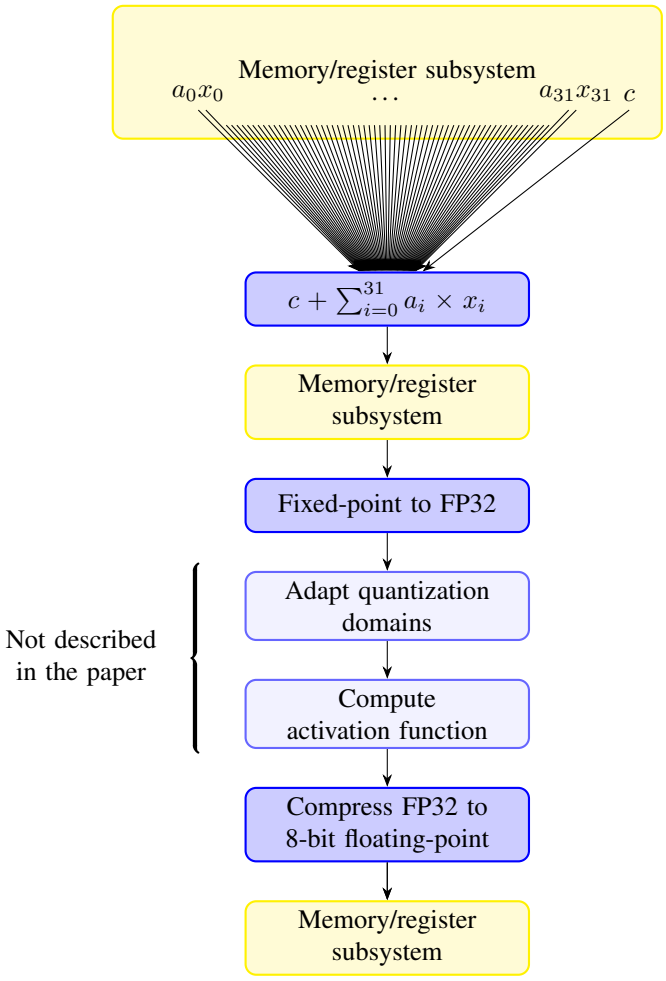


Fig. 2: Computation steps of a GEMM neural network layer.

III. Operator Architecture

A. Matrix Multiply Accumulate Operations

We select the use of a wide fixed-point accumulator in two's complement for computing exact dot products. This approach is well suited to 8-bit floating-point formats, as their reduced dynamic range enables to implement accumulators of practical size (Table I). In case of the Posit8.2 format, the structure of such exact accumulator is standardized as a 128-bit quire [5].

A full-precision dot product cannot generate Inf values (there is no rounding), while NaNs can only be generated by operating on Inf values. How to deal with the NaN and Inf inputs depends on the approach to quantization. According to [4], quantization should never round to Inf/NaN and clip to the maximal FP8 representable number instead. This behavior does not conform to the IEEE 754 round to nearest, ties to even (RNE) that can produce Inf values. So in the cases of quantization from FP32 using RNE, Inf values may be produced and will be propagated in the dot product operator. In the operator output, any Inf or NaN flags an invalid input.

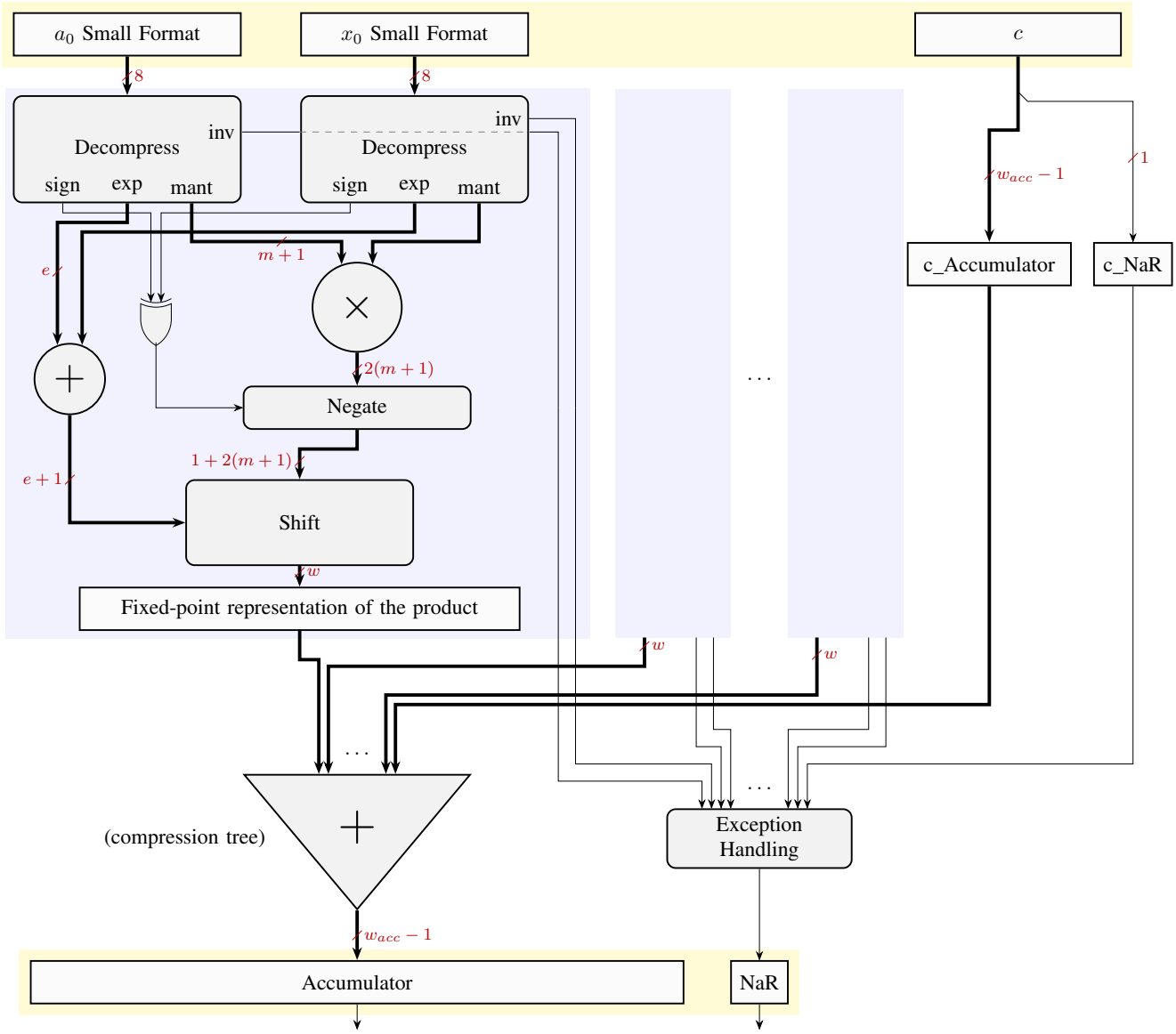


Fig. 3: Architecture of the exact dot product accumulate operator.

In the specific case of deep learning applications, distinguishing between Inf and NaN is useless; this agrees with the fact that the main proposals for FP8 with 4-bit exponent (E4M3 [1], 1.4.3 [4]) do not encode Inf. Accordingly, in our decompressor the Infs and NaNs are converted to a single NaR bit which has the same meaning as in the posit standard [5]. To save logic when converting the accumulator contents back to FP32, we use the least significant bit of the accumulator to represent the value of a NaN/Inf flag.

Programming from high-level languages requires that the accumulator be loaded from and stored to memory, likely using the wide memory access instructions which are usually provided for the SIMD data types; this motivates that the accumulator size be a power of 2. Loading and storing the accumulator also enables to parallelize the dot product computation across multiple processing cores while ensuring that a final reduction across cores yields the exact result.

Once accumulation is complete, the fixed-point accumulators are converted to FP32 for use by non-linear activation functions. The results of the FP32 activation functions may then be compressed back to a 8-bit floating-point representation. This yields a functional decomposition of the deep learning matrix multiply-accumulate operations into three hardware operators, illustrated in dark blue in Fig. 2. In the following, we focus on the implementation of these hardware operators for INT8, FP16 and the 8-bit floating-point formats.

B. Accumulator to FP32 Compression

Compressing the results of GEMM operations to FP32 is useful for adding the bias, whose range may exceed those of the low bit-width representations, and also for feeding the activation functions that depend on a math library. The wide

accumulator is an internal representation that is too large to be efficiently moved around for computations.

The design of the accumulator to FP32 compression operators is adapted from the one implemented in a production deep learning coprocessor [26]. The main modifications are to adjust to the different accumulator bit-widths, and to the fact that the least significant bit of the accumulator flags NaN/Inf values. Moreover, this compression operator does not need to detect special values (NaN is explicit in the accumulator).

The LSB of the FP32 range is -149 (Table I), so none of the accumulators can underflow this format. The MSB of the FP32 range is 127, so none of the accumulators can overflow this format except for Posit8.3. However this is not relevant in practice as this would require the accumulation of over 2^{31} ($\approx 10^9$) products of maxPosit. With the NaR information stored in a single bit of the accumulators, most of the logic of the operators is dedicated to a wide leading sign bit counter and shifter. Compression from the accumulator of the Posit8 dot product operators is virtually the same as the compression from the FP formats of same accumulator size.

C. FP32 to 8-Bit Floating-Point Compression

For the FP32 compression operators to the IEEE 754 derived FP8 formats, we adapted the design of floating-point narrow operators of a production FPU [26]. These compression operators have been developed directly in VHDL.

We only support rounding to nearest ties to even (RNE), which may produce Inf values. However format E4M3 has no codeword for the Inf values and only two for the NaN values: $s.1111.111_2$ where $s \in \{0, 1\}$ is the sign bit [1]. A design choice would be to use those NaN values to represent the signed Infs. However, as discussed earlier in the context of product accumulation, it is more important to return NaN in the case where $\infty \times 0$ occurs in the computation. As a result, we have chosen to use $0.1111.111_2$ for the E4M3 quiet NaN.

Producing value -0 when compressing FP32 to E4M3 is useless when feeding the dot product to wide accumulator operators located further down the computation graph, as these accumulate terms in fixed-point two's complement representation. We implemented compression to -0 for E4M3, but observe that the 1.4.3 format of [4] appears more interesting than the E4M3 format thanks to its two extra representable values. We anticipate that the corresponding compression operator from FP32 would be implemented very similarly.

The FP32 to Posit8 compression operators are derived from the compressors used for the hardware posit implementations of [31]. As the range of the FP32 format is larger than the one the standard Posit8 format, we do not worry about compressing the FP32 subnormal significands since the corresponding values will always be rounded to the Posit8 minPosit.

D. Exact Dot Product to Wide Accumulator

The exact dot product to wide accumulator operator implements the following steps (Fig. 3):

- Decompress the multiplicands.
- Multiply the unsigned significands.

TABLE I: Multiplier sizes, product sizes and accumulator sizes for the exact dot product accumulate operator.

Format	Product				Accumulator	
	size	LSB	MSB	w (bits)	MSB	w_{acc} (bits)
INT8	8×8	0	15	16	31	32
E4M3	4×4	-18	16	36	44	63+1
E5M2	3×3	-32	30	64	94	127+1
Posit8.0	6×6	-6	6	26	50	63+1
Posit8.1	5×5	-24	24	50	38	63+1
Posit8.2	4×4	-48	48	98	78	127+1
Posit8.3	3×3	-96	96	194	158	255+1
FP16	11×11	-48	30	80	78	127+1

- Convert the product to two's complement representation.
- Arithmetic shift right the product into a w-bit wide datapath.
- Sum the products using a compression tree and add to the accumulator.

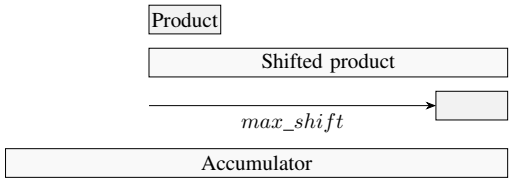
1) Decompression Step: The decompression step refers to translating the input number into sign, exponent and significand bit-fields that can be used in the remainder of the operator. Decompression is straightforward in case of the E4M3 and E5M2 formats, even for subnormal values; these do not need special casing other than clearing the implied significand leading bit. If required by power saving mechanisms, the decompression step could also identify the zero values to avoid useless term accumulation.

Work in [12] indicates that a table that maps every input to a sign-exponent-fraction format is more efficient than a combinatorial circuit to decompress 8-bit posits. This is even more the case here as the table does not need to output the FP16 subnormals, which require 10 fraction bits. The Posit8.2 format expands to an intermediate floating format with a sign bit, $e = 6$ and $m = 3$. Indeed, Posit8.2 decompression needs one more exponent bit than FP16 decompression to represent the eight values that do not fit into the FP16 range [12]; conversely it only needs at most 3 fraction bits instead of 10. The Posit8.1 format expands to a format $e = 5, m = 4$. The Posit8.0 format expands to a format $e = 4, m = 5$.

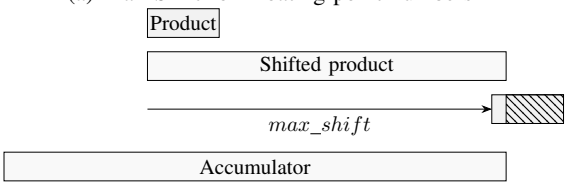
2) Shifted Product Size: For the multiplication of the significands and their insertion into a w-bit fixed-point datapath, the differences are only quantitative, depending on the input format as summarized in Table I. For each format, we compute the position of the least-significant bit (LSB) and the most-significant bit (MSB) of the shifted product.

The value of the LSB can be obtained by squaring the smallest representable number, while the MSB is obtained by squaring the largest number:

- For the IEEE-754 floating-point formats: LSB $= -2 \times (2^{e-1} - 2 + m)$; MSB $= 2 \times (2^{e-1} - 1 + \text{one\_nan})$ where one_nan is one when the highest exponent binade that normally contains Infs and NaNs also encodes normal values (case of E4M3) and zero otherwise.
- For the Posit formats, we take minPosit and maxPosit from the standard [5], compute the MSB and LSB which give the results in Table I. For Posit8.2, this corresponds to the $8n - 16$ specified by the quire of a Posit$n$ format.



(a) Max Shift for Floating-point numbers



(b) Max Shift for Posits

Fig. 4: Alignment of the product to a shifted product

An extra sign bit is needed in all cases, so the total size of the fixed-point two's complement value after alignment of the product is $w = \text{MSB} - \text{LSB} + 2$.

3) Shift Value: Fig. 4 illustrates the positioning of the products in the case of the smallest possible product, where the shift is *max_shift*. For each format family, we tune the shifter logic to the specific range of the fixed point representation described in Table I.

For the IEEE 754 derived formats, insertion of the (possibly negated) $2(m+1)$ bits of the product into the w bits of the fixed-point accumulator is performed by an arithmetic right shift of *max_shift* − sum_of_biased_exponents bits, with *max_shift* $= w - 2 \times (1 + m) - 1$.

Posits have no significand bits at the edges of their absolute value range, so there is no need to provide space for them. The computation of *max_shift* for Posits has a corrective constant to account for the biased exponent of the internal float representation having more range than the Posit. The shift is performed by an arithmetic right shift of *max_shift* + corrective_term − sum_of_biased_exponents bits, with *max_shift* $= w - 2$ and corrective_term $= LSB + 2 \times (2^{e-1} - 2)$.

4) Accumulator Size: Finally, we select the accumulator size to provide guard bits for the sum compression tree ($\log_2$ of the number of products), and round it up to the next power of two to match standard datatype sizes. We consider around 4000 products to add, which correspond to 12 extra bits.

E. RTL Design and Optimisations

We designed the dot product to wide accumulator based on a fork of the FloPoCo VHDL generator [32], [33]. The conversion operators have been written in VHDL.

The compression tree has optimisations implemented in the FloPoCo VHDL generator as described in [34]. The framework implementing the optimal compression enables to select different compressors optimised for the target. The standard cell Full Adder is the only compressor that was kept.

Since the accumulator is not immediately converted into a floating-point format, there is no need for a Leading Zero Anticipator circuit.

Dual-path optimisations [35] are not relevant, as our operators do not compute the difference between exponents and position all the numbers into the accumulator independently.

IV. Operator Synthesis and Evaluation

A. Operator Validation

The conversion operator from accumulator to FP32 was adapted from a production FPU integer to FP32 operator [26], enabling to reuse its validation workflow. The NaR bit extension has been specifically tested.

The conversion of FP32 to the 8-bit formats was tested with directed random tests. Those include testing of special values, overflow and underflow, the exact values of the 8-bit formats, as well as randomly selected values in between each 8-bit value to test rounding behavior. The posit conversions were verified against the official lookup table generator [36].

The dot product component starting after the decompression step was tested with the FloPoCo test bench, whose reference data is provided by MPFR [37]. Floating-point have a trivial decompression, and posit decompression was tabulated with the lookup table generator.

B. Compression to FP32 and to Smaller Formats

1) Accumulator to FP32 Compression: The accumulator to FP32 compression operator is synthesized with the Synopsys Design Compiler NXT for the TSMC 16FFC node with a target frequency of 1.25 GHz, fully pipelined in 2 cycles. The results appear in Table II.

2) FP32 to 8-Bit Floating-Point Compression: Table III reports the synthesis results obtained with the same workflow as in Table II. None of the FP32 to low bit-width compression operators need pipelining, while their power consumption and area appear one order of magnitude smaller than those of the accumulator to FP32 compression operators.

TABLE II: Synthesis results for the accumulator to FP32 compression operator, pipelined in 2 cycles at 1.25 GHz.

Formats	Acc (bits)	Area (μm^2)	Power (mW)
Posit8.3	255	2488	3.81
FP16, E5M2, Posit8.2	127	1542	2.41
E4M3, Posit8.1, Posit8.0	63	743	1.22
INT8	32	427	0.74

TABLE III: Synthesis results for the FP32 to low bit-width floating-point compression operators, with a single-cycle latency (no pipelining) at 1.25 GHz.

Format	Area (μm^2)	Power (mW)
FP16	117	0.22
INT8	99	0.18
E4M3	66	0.17
E5M2	65	0.16
Posit8.0	103	0.20
Posit8.1	122	0.21
Posit8.2	207	0.29
Posit8.3	110	0.21

C. Exact Dot Product to Wide Accumulator

The target frequency of the operator is 1.25 GHz with a latency of 5 cycles. Since we are targeting an ASIC rather than a FPGA, we do not use FloPoCo's auto-pipelining but instead synthesise the operator for one stage at a clock period of 5 times the desired clock period. The operator is therefore synthesized with the Synopsys Design Compiler NXT for the TSMC 16FFC node with a target frequency of 250 MHz. Synthesis results appear in Table IV. Operations per watt figures are normalized relative to those of the INT8 operator as it is the most energy-efficient.

The main takeaway from Table IV is that least expensive floating-point operator to implement is the one with the E4M3 multiplicands. Specifically, it requires only $1.64\times$ the power and $1.19\times$ the area of the corresponding INT8 operator. Although we designed for the E4M3 format [1], using the arguably better 1.4.3 format [4] would only incur minor changes in the decompression and compression steps with no impacts on the power or area.

Another observation is that the operator with Posit8.2 multiplicands is not better than the one with FP16 multiplicands, whether on dynamic power or on area. This can be explained by the fact that the wider range of the Posit8.2 values compared to FP16 leads to a larger fixed-point representation of the shifted products (96 bits versus 80 bits from Table I), which in turn implies wider shifters and compression tree. The effects of the increases of the fixed-point datapath appear to compensate those of the decrease of the multiplier sizes.

The cost of the decompression from the Posit format to a sign-exponent-fraction format also appears significant, having apparently a bigger impact on hardware complexity than expanding to fixed-point representation. Indeed, the Posit8.0 and Posit8.1 have a fixed-point representation of the same size as the E4M3 floating point, but have an extra cost of 47% for Posit8.0 and 88% for Posit8.1. The cost of Posit8.1 is even 12% higher than E5M2, even though E5M2 has an accumulator twice the size. Posit8.3 requires the widest accumulator, which explains its poor energy efficiency.

With regards to the power and area figures of the compression operators, they appear as $5\times$ to $10\times$ lower than those of the corresponding dot product operators. Compression operators are normally used only once per element of the matrix multiply-accumulate computation. This further lowers their relative contribution to the total energy consumption.

TABLE IV: Synthesis results of the exact dot product accumulate operator, with a target frequency of 250 MHz.

Format	# of products	Area (μm^2)	Power (mW)	OPs/W ratio
FP16	16	8040	4.12	0.4
INT8	32	4107	1.67	1.0
FP16	32	15482	7.84	0.21
E4M3	32	4896	2.73	0.61
E5M2	32	8266	4.67	0.36
Posit8.0	32	7188	3.93	0.42
Posit8.1	32	9222	5.19	0.32
Posit8.2	32	17821	9.85	0.17
Posit8.3	32	26217	17.23	0.1

V. Summary and Conclusions

We propose a generic dot product accumulate operator architecture targeting 8-bit floating-point arithmetic in deep learning applications. Its basic principle is to exactly compute and accumulate the dot products of the matrix multiply-accumulate operations with full precision, an appropriate option for the 8-bit floating-point formats considered. These 8-bit floating-point formats include the FP8 family, which follows or adapts the rules of the IEEE 754 binary floating-point representations, and the Posit8 formats (with *es* = 0,1,2,3).

The proposed dot product operators are complemented by two families of conversion operators, one for the compression of wide accumulators to FP32, and the other for the compression of FP32 values to 8-bit floating-point formats.

Besides operator architectures, our contribution exposes the implementation costs of using different 8-bit number formats for operators that exactly compute a 32-term dot product for 8-bit floating-point vectors and accumulate them into 64-bit or 128-bit full precision fixed-point accumulators.

Synthesis for the TSMC 16FFC node at a 1.25 GHz target frequency shows the interest of the FP8 format with four bits of exponent and three bits of fraction (E4M3) in terms of dynamic power and area. Compared to the INT8.32 dot product accumulate operator, the corresponding E4M3 operator only requires $1.64\times$ more power and $1.19\times$ more area.

It also appears that computing full-precision dot products using Posit8.2 multiplicands leads to an operator comparable in power and size to the one using FP16 multiplicands, both of them being two to three times more expensive than the operator with the E4M3 multiplicands.

However, the Posit8.2 format encodes a wider value range than the IEEE 754 inspired FP8 and standard FP16 formats. Although there could be issues with the effects of the posit geometric rounding at the extremes of the Posit8.2 range, when constrained to 8-bit data this wider range could provide decisive benefits compared to the FP8 formats. Further experimentation with deep learning inference and training is required to confirm or invalidate this hypothesis.

Acknowledgment

This work has received funding from the European High Performance Computing Joint Undertaking (JU) (EPI SGA2).

The authors thank Florent de Dinechin for fruitful discussions and support for adapting the FloPoCo VHDL generator to our ASIC synthesis workflow.

References

[1] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu, and H. Wu, "Fp8 formats for deep learning," 2022.

[2] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[3] "Tesla Dojo Technology — A Guide to Tesla'sConfigurable Floating Point Formats & Arithmetic Tesla Dojo TechnologyA Guide to Tesla's Configurable Floating Point Formats & Arithmetic."

[4] B. Noune, P. Jones, D. Justus, D. Masters, and C. Luschi, "8-bit numerical formats for deep neural networks," 2022.

[5] "Standard for Posit Arithmetic (2022) Release 5.0," 2022.

[6] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, 2018.

[7] A. Kuzmin, M. Van Baalen, Y. Ren, M. Nagel, J. Peters, and T. Blankevoort, "FP8 quantization: The power of the exponent," 2022.

[8] Z. Carmichael, S. H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep positron: A deep neural network using the posit number system," *CoRR*, vol. abs/1812.01762, 2018.

[9] M. Christ, F. de Dinechin, and F. Petrot, "Low-precision logarithmic arithmetic for neural network accelerators," in *2022 IEEE 33rd Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, 2022.

[10] J. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, 2017.

[11] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," *IEEE Trans. on Computers*, 2021.

[12] O. Desrentes, D. Resmerita, and B. Dupont de Dinechin, "A posit8 decompression operator for deep neural network inference," in *Conf. for Next Generation Arithmetic*, 2022.

[13] R. Murillo, A. A. Del Barrio, and G. Botella, "Deep PeNSieve: A deep learning framework based on the posit number system," *Digital Signal Processing*, 2020.

[14] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "Small reals representations for deep learning at the edge: A comparison," in *Conf. for Next Generation Arithmetic* (J. Gustafson and V. Dimitrov, eds.), 2022.

[15] D. M. Rifqie, D. F. Surianto, N. M. Abdal, W. H. M., and H. Ramli, "Post training quantization in lenet-5 algorithm for efficient inference," *J. of Embedded Systems, Security and Intelligent Systems*, vol. 3, no. 1, pp. 60–64, 2022.

[16] J. a. P. L. de Carvalho, J. E. Moreira, and J. N. Amaral, "Compiling for the IBM matrix engine for enterprise workloads," *IEEE Micro*, vol. 42, pp. 34–40, sep 2022.

[17] A. Limited, "Arm architecture reference manual supplement, the scalable matrix extension (SME), for Armv9-A," 2022.

[18] N. Brunie, F. de Dinechin, and B. Dupont de Dinechin, "A mixed-precision fused multiply and add," in *Record of the Forty Fifth Asilomar Conf. on Signals, Systems and Computers (ASILOMAR)*, 2011.

[19] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, and S. Avancha, "Intel nervana neural network processor-t (NNP-T) fused floating point many-term dot product," in *IEEE 27th Symp. on Computer Arithmetic (ARITH)*, 2020.

[20] H. Kaul, M. Anders, S. Mathew, S. Kim, and R. Krishnamurthy, "Optimized fused floating-point many-term dot-product hardware for machine learning accelerators," in *IEEE 26th Symp. on Computer Arithmetic (ARITH)*, 2019.

[21] B. Hickmann and D. Bradford, "Experimental analysis of matrix multiplication functional units," in *IEEE 26th Symp. on Computer Arithmetic (ARITH)*, 2019.

[22] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *32nd Int. Conf. on Neural Information Processing Systems (NIPS)*, 2018.

[23] H. Zhang, D. Chen, and S.-B. Ko, "Efficient multiple-precision floating-point fused multiply-add with mixed-precision support," *IEEE Trans. on Computers*, 2019.

[24] L. Bertaccini, G. Paulin, T. Fischer, S. Mach, and L. Benini, "MiniFloat-NN and ExSdotp: An ISA extension and a modular open hardware unit for low-precision training on RISC-V cores," in *IEEE 29th Symp. on Computer Arithmetic (ARITH)*, 2022.

[25] N. Brunie, "Modified fused multiply and add for exact low precision product accumulation," in *IEEE 24th Symp. on Computer Arithmetic (ARITH)*, 2017.

[26] B. D. de Dinechin, J. Hascoët, J. L. Maire, and N. Brunie, "Deep Learning Inference on the MPPA3 Manycore Processor," in *Embedded World Conference*, 2020.

[27] U. Kulisch and V. Snyder, "The exact dot product as basic tool for long interval arithmetic," *Computing*, vol. 91, p. 307–313, mar 2011.

[28] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanovic, "A hardware accelerator for computing an exact dot product," in *IEEE 24th Symp. on Computer Arithmetic (ARITH)*, 2017.

[29] D. Mallasén, R. Murillo, A. A. D. Barrio, G. Botella, L. Piñuel, and M. Prieto-Matias, "PERCIVAL: Open-source posit RISC-V core with quire capability," *IEEE Trans. on Emerging Topics in Computing*, 2022.

[30] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "A lightweight posit processing unit for RISC-V processors in deep neural network applications," *IEEE Trans. on Emerging Topics in Computing*, 2021.

[31] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the hardware cost of the posit number system," in *29th Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2019.

[32] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, 2011.

[33] F. de Dinechin, "Reflections on 10 years of FloPoCo," in *26th IEEE Symp. of Computer Arithmetic (ARITH-26)*, 2019.

[34] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for FPGAs," *IEEE Trans. on Computers*, vol. 67, no. 8, pp. 1078–1091, 2018.

[35] Y. Tao, A. Jianfeng, G. Deyuan, and F. Xiaoya, "Dual-path architecture of floating-point dot product computation," in *Proceedings of 2011 Int. Conf. on Computer Science and Network Technology*, vol. 4, pp. 2272–2276, 2011.

[36] "Posit Lookup Table Generator." `https://posithub.org/widget/lookup`. Used for tests in first semester of 2023.

[37] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. on Mathematical Software (TOMS)*, 2007.