Double-Word Decomposition in a Combined FP16, BF16 and FP32 Dot Product Add Operator

Orégane Desrentes*†, Benoît Dupont de Dinechin* and Florent de Dinechin†
* Kalray S.A., 38330 Montbonnot-Saint-Martin, France, {odesrentes, bddinechin}@kalrayinc.com
† INSA Lyon, Inria, CITI, UR3720, 69621 Villeurbanne, France, florent.de-dinechin@insa-lyon.fr

***Abstract*—This work presents a floating-point Fused Dot Product Add operator designed for two use cases: mixed-precision matrix multiply add for deep learning and single-precision arithmetic for numerical computing. We build on classic double-word techniques to emulate FP32 arithmetic as the unevaluated sum of lower-precision floating-point numbers. Specifically, we introduce E9S12, a 9-bit exponent, 12-bit significant, non-normalized floating-point format, to represent either the FP16 and BF16 multiplicands or the results of the FP32 multiplicand decomposition inside a Fused Dot Product Add operator. The resulting operator correctly rounds the FP16 products accumulated in FP32 and also matches the standard FP32 FMA arithmetic when the size-4 E9S12 dot product operates on decomposed FP32 multiplicands. We evaluated the implementation of this combined FP16, BF16 and FP32 Dot Product Add operator and compare it to correctly rounded Fused Dot Product Add operators by synthesis for a 4nm technology node.**

***Index Terms*—Dot Product, FP16, BF16, FP32**

I. Introduction

Processing units for deep learning applications rely on mixed-precision Matrix Multiply Add (MMA) units to accelerate convolutions and other linear operations. These units may be built from Dot Product Add (DPA) operators: $R = Z + \sum X_i \times Y_i$. The multiplicands are 16-bit IEEE 754 [1] floating-point numbers (FP16), Bfloat16 (BF16) [2] or 8-bit floating-point numbers [3]. Their product is accumulated into a 32-bit IEEE 754 binary floating-point (FP32) number [4], [5], [6], [7]. The performance and energy efficiency of these MMA units motivate their use for numerical computing in FP32 or higher precision.

A. Double-Word Arithmetic

The core idea (Fig.1) is to adapt the principles of double-word arithmetic [8] to linear algebra [9].

To accelerate an FP32 matrix multiplication $C = AB$ using an FP16 MMA unit, A and B are decomposed as the unevaluated sum of FP16 matrices $A \simeq A^h + A^l$ and $B \simeq B^h + B^l$. A mixed-precision MMA then computes an approximation of the FP32 matrix C as $C \approx A^hB^h + A^hB^l + A^lB^h + A^lB^l$. The decomposition of an FP32 matrix A into FP16 matrices A^h, A^l is obtained by [4]:

$$\begin{aligned} A^h &= \textbf{toFP16}(A) \quad , \\ A^l &= \textbf{toFP16}(A - \textbf{toFP32}(A^h)) \quad . \end{aligned} \tag{1}$$

The **toFP16()** operation converts each FP32 element to FP16. Likewise, **toFP32()** converts each FP16 element to FP32.

This approach has several drawbacks:

1) The dynamic range of the FP16 representation leads to extreme precision losses for large FP32 values [4], since the 8-bit exponent of the FP32 representation is larger than the 5-bit exponent of the FP16 representation. In other words, (1) often over/underflows.
2) Even if the FP32 multiplicands fit in the dynamic range of the FP16 format, there is still a loss of precision as the number of bits in two FP16 significands $(2 \times (10 + 1))$ is strictly less than in an FP32 significand $(23 + 1)$. In other words, (1) is usually inexact.
3) An additional loss of precision arises from the addition of subproduct matrices $A^hB^h, A^hB^l, A^lB^h, A^lB^l$ in FP32 arithmetic, which in the case of NVIDIA Tensor Cores only supports round to zero [4].
4) The explicit decomposition of the FP32 multiplicand matrices A, B into the FP16 matrices (A^h, A^l, B^h, B^l) increases the complexity of application software.

A solution to issues 1 and 2 is to use triple-BF16 [10], but it makes issues 3 and 4 worse. The motivation of the present work is to address all these drawbacks in hardware.

B. Contributions

This paper presents the architecture of a fused Dot Product Add (DPA) operator that targets both the mixed-precision operations used by machine learning and the FP32 linear operations used in numerical computing. The main enabler is the definition of an intermediate floating-point format called E9S12 (9 exponent bits, 12 significand bits), which is used inside the operators to represent the FP16 or BF16 multiplicands, and also enables an exact double-word decomposition of FP32 multiplicands.

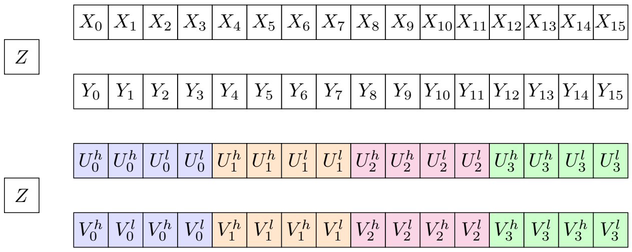
X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8 X_9 X_{10} X_{11} X_{12} X_{13} X_{14} X_{15}
Z
Y_0 Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7 Y_8 Y_9 Y_{10} Y_{11} Y_{12} Y_{13} Y_{14} Y_{15}
U_0^h U_0^h U_0^l U_0^l U_1^h U_1^h U_1^l U_1^l U_2^h U_2^h U_2^l U_2^l U_3^h U_3^h U_3^l U_3^l
Z
V_0^h V_0^l V_0^h V_0^l V_1^h V_1^l V_1^h V_1^l V_2^h V_2^l V_2^h V_2^l V_3^h V_3^l V_3^h V_3^l

Fig. 1. FP16 dot product of size 16 $R = Z + \sum X_i \times Y_i$ (top), used as an FP32 dot product of size 4 (bottom).

This work was partially supported by the PEPR IA HOLIGRAIL project of the Agence Nationale de la Recherche, ANR-23-PEIA-0010 and the European High- Performance Computing Joint Undertaking (EPI SGA2)

2576-2265/25/$31.00 ©2025 IEEE
DOI 10.1109/ARITH64983.2025.00021

On the implementation side, the starting point is a correctly rounded fused DPA operator with FP16 multiplicands and FP32 addend. This operator is extended to support E9S12 multiplicands, as this also enables the use of BF16 and FP32 multiplicands. The FP32 multiplicands are decomposed in hardware as double-E9S12. The resulting operator is still correctly rounded for FP16 multiplicands, and is IEEE compliant when emulating an FP32 Fused Multiply Add (FMA) operator. The proposed operator is not only faster, but also more accurate than software solutions based on decompositions of FP32 multiplicands into pairs of FP16 or BF16 numbers.

Specifically, for an integer N, let $(X_i)_{i\in[0,4N-1]}$, $(Y_i)_{i\in[0,4N-1]}$ be FP16 numbers, Z and R be FP32 numbers. The proposed DPA operator computes the sum of a dot product of X_i, Y_i and the addend Z with a single rounding $\circ$:

$$R = \circ(X_0 \times Y_0 + \ldots + X_{4N-1} \times Y_{4N-1} + Z) \quad .$$

This requires an internal accumulator slightly larger than 80 bits (see Section IV-B). If the inputs $(X_i)_{i\in[0,4N-1]}$, $(Y_i)_{i\in[0,4N-1]}$ are BF16 numbers, the same precision is used to compute their dot product. This process and the corresponding truncation is noted ϕ. The DPA operator then computes

$$R = \circ(\phi(X_0 \times Y_0 + \ldots + X_{4N-1} \times Y_{4N-1}) + Z) \quad .$$

Finally, the same DPA operator may compute, for FP32 numbers $(U_i)_{i\in[0,N-1]}$, $(V_i)_{i\in[0,N-1]}$,

$$R = \circ(\phi(U_0 \times V_0 + \ldots U_{N-1} \times V_{N-1}) + Z) \quad .$$

This operator provides the building block of a large pipelined MMA accelerator, the details of which are beyond the scope of this work.

C. Outline

This paper is organized as follows. Section II discusses background and related work. Section III introduces the intermediate format that supports the proposed operator features. Section IV describes the proposed DPA operator architecture and variants. Section V presents validation details and synthesis results. Section VI concludes the paper.

II. Background and Related Work

A. Double-Word Floating-Point Arithmetic

A double-word number [11] X is defined as the unevaluated sum of two floating-point numbers X^h and X^l such that $X = X^h + X^l$ and $X^h = \circ(X)$ using round to nearest. Thus X^l represents the signed rounding error of X: $X^l = X - \circ(X)$.

The exact result of floating-point additions or multiplications can be represented as a double word, which is useful for error-free transforms [8]. Hardware operators that input two floating-point numbers and output the exact sum or product as a double word have been studied in [12].

In a binary floating-point format with p bits for the fraction, the significand has $p+1$ bits thanks to the implicit bit. Interestingly, a double-word decomposition can always represent at least $2p+3$ consecutive bits: $p+1$ bits for X^h, $p+1$ bits for X^l, and an extra bit encoded in the sign of X^l as follows:

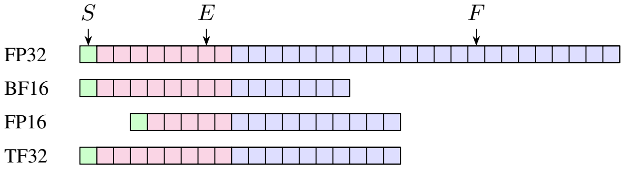

Fig. 2. Floating-point formats supported by NVIDIA Tensor Cores [13].

- If X was a midpoint between two exact floating-point values, then only $p+2$ bits are needed to represent its significand.
- Otherwise, X^l is strictly smaller than the half-ulp [8] of X, therefore there is a gap of at least 1 bit between X^h and X^l.

B. Multiword Techniques for NVIDIA Tensor Cores

Tensor Cores in NVIDIA GPGPUs are mixed-precision fused MMA units [4]. They multiply matrices with FP16 elements and add the products to a matrix with FP32 elements. Since the NVIDIA Ampere architecture, they also support other multiplicand formats, including BF16 and TF32 [13], a combination of FP16 and BF16 formats (Fig. 2).

Although designed to accelerate deep learning kernels, Tensor Cores are also used in numerical analysis to improve the performance of matrix multiplications in FP32 arithmetic. One approach is to rely on iterative refinement techniques [14], while others adapt multi-word arithmetic techniques. In this setting, the TF32 format is appealing because it has the same exponent size as the FP32 format. This alleviates the dynamic range problem of double-FP16 decompositions of FP32 numbers. However, the limited accuracy of double-FP16 decompositions remains with double-TF32 decompositions.

C. Relevant Previous Work in Hardware Floating Point

a) Floating-Point Addition: The classic way to compute a floating-point addition of two numbers is to align the significands on the one with the largest magnitude [15]. The size of the adder is about the size of a significand. The significand with the smallest magnitude is shifted right relative to the largest one, with the bits shifted out ORed into a sticky bit for further rounding. After the addition, a leading zero count (LZC) and a second shift are needed for the possible renormalization of the result.

b) Dot product in a Long Accumulator: For a sum of many terms, the Long Accumulator (often referred to as a Kulisch accumulator) was proposed as a way to avoid the error due to shifted-out bits [16], [17]. It can also be extended to accumulate the exact products in a dot product, thus also removing rounding errors due to floating-point multipliers. The exact dot product is converted back into a floating-point number only at the end of the computation.

A Long Accumulator large enough to hold exact dot products requires ≈ 4200 bits for FP64 numbers and ≈ 550 for FP32 numbers. Early Long Accumulators were iterative, accumulating one product per instruction [16], [17]. The corresponding large fixed-point addition can be sped up thanks to parallel execution [18]. A two's complement, high-radix carry-save representation of the accumulator allows for high frequency operation at low hardware cost [19].

With FP16, the size of the Long Accumulator is only 80 bits, and it becomes an efficient way to implement a mixed-precision FP16×FP16+FP32 Fused Multiply and Add (FMA) [20]. This insight was also applied to a dot product operator in [21], where several FP16 products are computed in parallel and accumulated into an FP32 number every clock cycle.

For even smaller formats such as FP8, a Long Accumulator is the best option to implement dot product operators [22], [7]. For ranges larger than FP16, an option is to compress the redundant parts of the Long Accumulator [23], [24] while still ensuring correctly rounded dot product computations.

c) Other dot product implementations: For $z = \circ(x_0 \times y_0 + x_1 \times y_1)$, the two products can simply be computed in parallel and added as a floating-point sum of two numbers [25].

Another operator [26] implements the operation $z = \circ(\circ(x_0 \times y_0 + x_1 \times y_1) + \circ(x_2 \times y_2 + x_3 \times y_3))$ with all intermediate roundings as one instruction. This dot product operator also computes multiple products in parallel.

The MMA units of mainstream GPUs [4], [5], [6], [27], [28] use a variant of the floating-point addition architecture: All products are aligned relative to the one with the largest magnitude then added to a floating-point accumulator that can be of an arbitrary size, the popular choice being 24 bits.

D. Baseline Technique for the Present Work

In the FP16 mixed-precision FMA operator of [20], an FP16 product $A \times B$ is first converted to a fixed-point number, which is then added with correct rounding to an FP32 addend Z. The fixed-point format that contains all FP16 values has a Most Significant Bit (MSB) of 30 and a Least Significant Bit (LSB) of -48, hence the total size 80 bits. The operator of [20] implements the addition between the FP32 addend and the fixed-point product in a way that ensures a correctly rounded result to FP32, with fewer bits than a Long Accumulator matching the FP32 range (which would be 277 bits from MSB 127 down to LSB -149).

The FP32 addend is first shifted relative to the FP16 accumulator. Fig. 3 (not to scale) summarizes the main alignment cases in this addition.

- Case 1 in Fig. 3 is similar to the way classic floating-point addition is computed. The FP32 addend is shifted relative to the FP16 accumulator and shifted into a sticky bit if needed (Case 1').
- In Case 2 of Fig. 3, the FP32 addend has a larger exponent than the MSB of the fixed-point FP16 product. The result of the multiply-add is the FP32 addend, possibly modified only with a rounding contribution from the fixed-point product. The alignment of the FP32 addend is limited to its LSB being one bit larger than the MSB of the FP16 product. As the guard bit stays clear, this allows for a correctly rounded sum.

The two numbers are added, and the FP32 exponent is used as the reference exponent for the result.

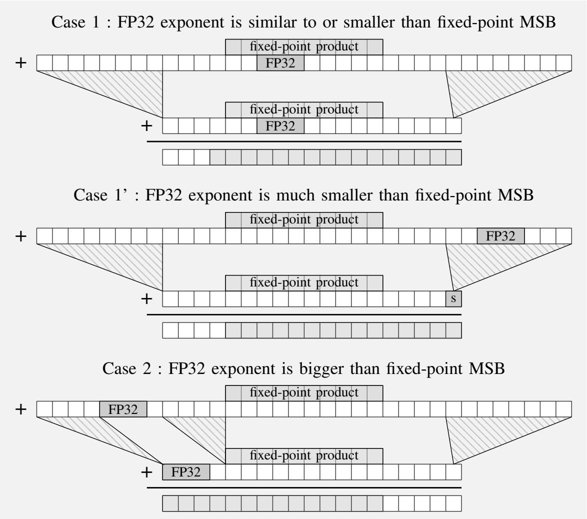
Case 1 : FP32 exponent is similar to or smaller than fixed-point MSB
fixed-point product
+
FP32
fixed-point product
+
FP32
Case 1' : FP32 exponent is much smaller than fixed-point MSB
fixed-point product
+
FP32
fixed-point product
+
S
Case 2 : FP32 exponent is bigger than fixed-point MSB
fixed-point product
+
FP32
fixed-point product
+
FP32

Fig. 3. Addition of an FP32 number to a fixed-point (30, -48) number [20].

The Leading Zero Count (LZC) of the sum (the lower lines of Fig. 3) is computed, and then there are three cases to consider to determine the output exponent:

- In case 1, the exponent of the result is $30 + 25 - \text{LZC} + \text{FP32bias}$, where $30 + 25$ corresponds to the (unbiased) weight of the MSB of the sum.
- In case 1', when the FP16 product is 0 (which can be determined from the LZC), the FP32 addend which has been rounded off in the alignment must be restored by wiring it from the input to the output without modification.
- If the exponent of the FP32 addend is larger than $30 + 25 + \text{FP32bias}$, it is used as the result exponent (Case 2).

In the first and last cases, the significand is the normalized and rounded result of the sum.

The minimum exponent of an FP16 product (-48) is within the range of normal FP32 numbers. The only way this operator can produce a subnormal result is if the FP16 accumulator is 0 and the FP32 addend Z is subnormal, so it is wired unchanged to the output (second case). Thus, there is no need for the rounding logic to handle subnormal outputs.

This method can be extended to a $\text{DP}_{\text{FP16}}\text{A}$ operator, as the sum of several fixed-point FP16 products is still a fixed-point number. The fixed-point format must be extended with a few bits to absorb possible overflows. This $\text{DP}_{\text{FP16}}\text{A}$ operator architecture was described in [21].

III. Intermediate Floating-Point Format

In order to operate on FP16, BF16 and double-word decompositions of FP32 multiplicands, we first define an intermediate format to represent all of them and be suitable for implementation of the downstream calculations. This intermediate format is operational, and thus does not need to be frugal in bits, unlike a storage format. Moreover, conversion to this format should be easy to implement in hardware.

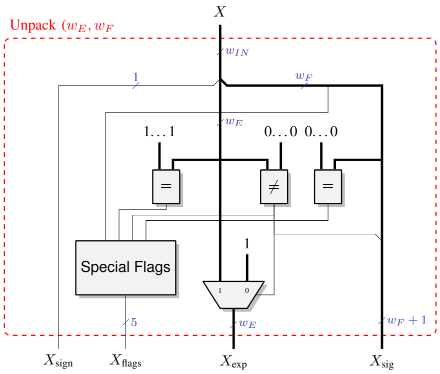

Fig. 4. Unpacking a floating-point number with exponent size w_E and fraction size w_F.

A. FP16 and BF16 Multiplicands

Let us first define a generic "Unpack(w_E, w_F)" operation (Fig. 4) that expands a floating-point number X with w_E exponent bits and w_F fraction bits into its sign X_{sign}, its biased exponent X_{exp}, its significand X_{sig}, and a vector of flags X_{flags} (isnormal, isinf, isnan, issignan, iszero). The implicit bit is made explicit in the significand, so there is no need to normalize subnormal inputs, they will just have a non-normalized significand.

Unpacking an FP16 number requires an Unpack(5, 10), outputting 5 exponent bits and 11 significand bits. Unpacking a BF16 number requires an Unpack(8, 7), outputting 8 exponent bits and 8 significand bits. Therefore, an unpacked format supporting both FP16 and BF16 numbers requires 1 sign bit, 8 exponent bits, 11 significand bits, and 5 bits for X_{flags}.

B. Decomposed FP32 Multiplicands

We now introduce a hardware-friendly approach to the idea of reusing an FP16 / BF16 datapath to compute an FP32 dot product using double-word arithmetic.

An FP32 number U is decomposed into an unevaluated sum of U^h and U^l, that is, $U = U^h + U^l$. The key to making this decomposition hardware-friendly is to adapt the intermediate floating-point format to represent U^h and U^l (Fig. 5). Since an FP32 number has a significand of 24 bits, it should be split into two significands of 12 bits There is no need for the rounding trick of Section II-A that requires $U^h = \circ(U)$. Instead, it is possible to just split the mantissa U_{sig} into its 12 most significant bits that become U^h_{sig}, and its 12 least significant bits which become U^l_{sig} (Fig. 5). Since the significands in the intermediate format may be not normalized, it is not necessary either to normalize U^l_{sig} if its leading bit is 0. The signs of U^h and U^l are the same (Fig. 5).

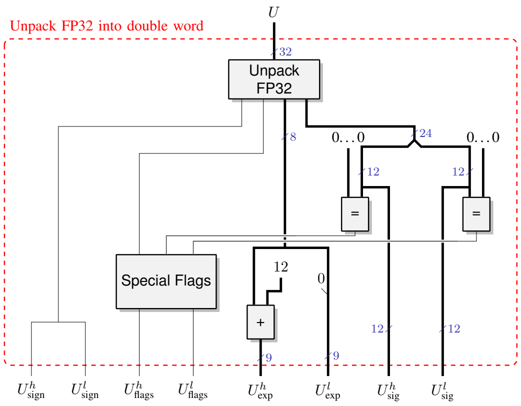

Fig. 5. Unpacking an FP32 number into two E9S12.

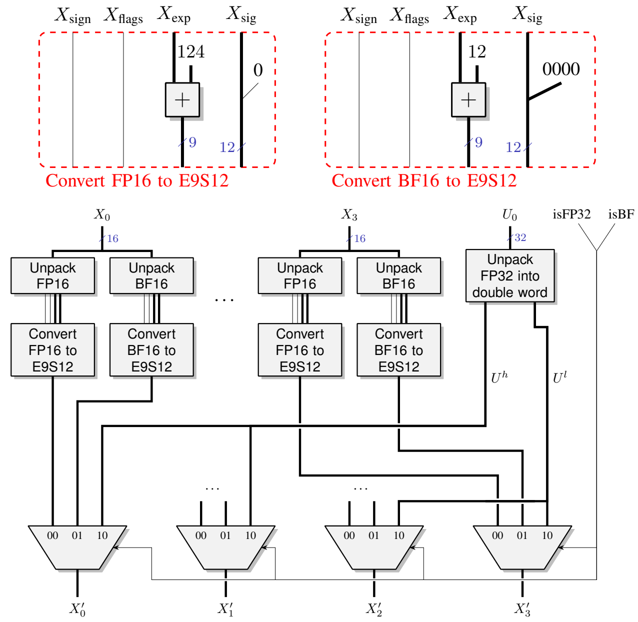

Fig. 6. Unpacking an FP16, a BF16 or an FP32 number into E9S12.

The exponent of U^l, U^l_{exp} is the exponent of U minus the constant 12, which is simple to implement. To ensure that 0 encodes the smallest possible exponent, the intermediate format uses an encoding with an exponent bias of $139 = 127{+}12$. Without this bias, negative exponents $U^l_{\text{exp}} = U_{\text{exp}} - 12$ would appear during the decomposition of subnormal FP32 numbers and those with exponent $U_{\text{exp}} < 12$.

Using the bias of 139, the hardware sets $U^l_{\text{exp}} = U_{\text{exp}}$ and $U^h_{\text{exp}} = U_{\text{exp}} + 12$. If the input U is subnormal, then $U^l_{\text{exp}} = 0$ and $U^h_{\text{exp}} = 12$. This bias change is compensated later in the operator when computing the final exponent. However, it requires one additional exponent bit in the intermediate format.

Special care is needed when dealing with this non-normalized, non-standard format. For example, U^l can be zero with a nonzero exponent. In general, this approach is simpler and more energy efficient than implementing (1) which enforces $U^h = \circ(U)$.

C. The Common Intermediate Format

The intermediate floating-point format that captures all these multiplicands is called E9S12 in this work. It is encoded using 27 bits: 1 sign bit, 9 exponent bits, 12 bits for the significand and the 5 flag bits. In other words, the format needed to decompose the FP32 numbers is also large enough for the unpacking of the FP16 and BF16 numbers as well.

As shown in Fig. 6, the outputs of the BF16 Unpack(8,7) and the FP16 Unpack(5, 10) are converted to the E9S12 format by trivial zero extension of the significand and update of exponent bias.

D. Intermediate Format Applications

The simplest application of E9S12 is to implement an FP32 Multi-Addition (MA) operator. Each FP32 input U is decomposed into $U^h + U^l$, each in E9S12 format. The MA2N operator for E9S12 can be used as a MAN operator for FP32:

$$\begin{aligned} R &= \circ(U_0^h + U_0^l + \ldots + U_{(N-1)}^h + U_{(N-1)}^l) \\ &= \circ(U_0 + \ldots + U_{N-1}) \quad . \end{aligned}$$

When unpacking the inputs for the Multi-Addition, each FP32 input is decomposed into two E9S12.

However, the main application of E9S12 is to implement a DPA operator with FP16, BF16 or FP32 multiplicands, FP32 addend and FP32 result. In case of FP16 or BF16, each 16-bit multiplicand X_i, Y_i is unpacked into the E9S12 format (Fig. 6), and the DP4NA operator computes:

$$R = \circ(Z + X_0' \times Y_0' + \ldots + X_{4N-1}' \times Y_{4N-1}') \quad .$$

In case of FP32 multiplicands U (resp. V) is decomposed (Fig. 5) into U_h and U_l in E9S12 format such that $U = U_h + U_l$ (resp. $V = V_h + V_l$). The product $U \times V$ is rewritten as:

$$\begin{aligned} U \times V &= (U^h + V^l) \times (U^h + V^l) \\ &= U^h \times V^h + U^h \times V^l + U^l \times V^h + U^l \times V^l \quad . \end{aligned}$$

The DP4NA for E9S12 multiplicands can then be used as a DPNA for FP32 multiplicands:

$$\begin{aligned} R &= \circ(Z + \sum_{i=0}^{N-1} (U_i^h \times V_i^h + U_i^h \times V_i^l + U_i^l \times V_i^h + U_i^l \times V_i^l) \\ &= \circ(Z + U_0 \times V_0 + \ldots + U_{N-1} \times V_{N-1}) \quad . \end{aligned}$$

The unpack circuit for this operator is computed in blocks of 4 16-bit inputs. A block is shown in Fig. 6 for X. In the corresponding block for Y, the outputs Y_1' and Y_2' are switched compared to this figure.

IV. Dot Product Add Operators

We first present in Sect. IV-A a baseline Dot Product Add Operator for sums of FP16 products. Here an exact multi-addition of 81-bit numbers allows for correct rounding. Then we show in Sect. IV-B how this architecture can be also used for E9S12 Multiplicands. In this case, the wider exponent range makes the exact sum of products too costly, so the architecture keeps the 81-bit sum, but attaches an exponent to it before rounding it to FP32.

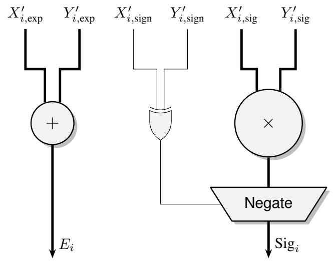

Fig. 7. Product of two floating-point numbers. X_i', Y_i' can be FP16 or E9S12.

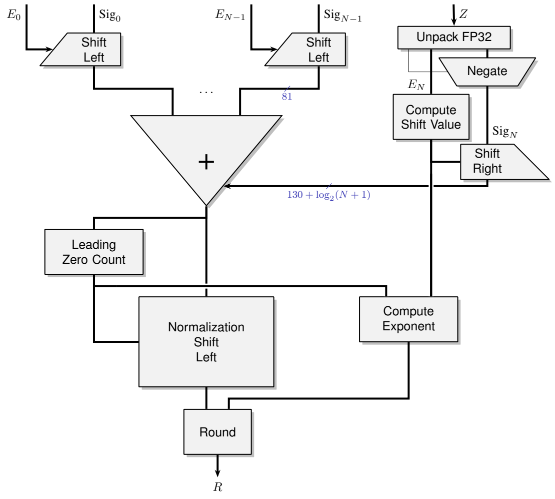

Fig. 8. Architecture of a correctly-rounded FP16 Dot Product Add operator.

A. Baseline FP16 Dot Product Add Operator

The baseline operator (Fig. 8) reuses the techniques of [21] (see Section II-D) to implement the DPA operation $R = \circ(X_0' \times Y_0' + \ldots + X_{N-1}' \times Y_{N-1}' + Z)$, where X', Y' are vectors of numbers in the FP16 format and Z is an FP32 number. The first steps in this DPA operator is the computation of the significand products $X_i' \times Y_i'$ and their conversion into two's complement fixed-point format. This involves XORing the signs, adding the exponents, multiplying the significands, and optionally negating the result (Fig. 7).

The products Sig_i are then shifted by their exponent E_i to obtain their fixed-point value. The sum of these fixed-point products is computed exactly by a fixed-point multi-adder. This is where two's complement is more efficient than keeping the sign-magnitude representation of the inputs [19].

The sum also includes the contribution of the FP32 addend Z, which after unpacking has a different format than the products. The relative shift of the fixed-point sums of products and the FP32 addend follows the principles of Sect. II-D [20]. Consequently, the conversion from the exact fixed-point sum to FP32 ensures a correctly rounded result.

B. Dot Product Add Operator with E9S12 Multiplicands

The E9S12 format has a notably larger dynamic range than FP16, which would involve shifting and accumulating fixed-point numbers over 540 bits (MSB 254 and LSB -298).

This makes an exact fixed-point summation impractical, so it is replaced by an accurate floating-point summation method described in Fig. 9.

a) Architecture overview: The products are aligned to the product with the largest exponent E_{max}. The accumulator is reduced to an arbitrary size L, but it is now needed to store its exponent E_{Acc}.

To avoid loss of precision in the case of partial cancellation, the sum size should be at least twice the size of the significand of the result (48 bits for FP32). We chose here the sum size that enables the computation of a correctly rounded FP16 Dot Product Add: $L = 81 + \log_2(N)$ bits. This has proven to be useful for formats with a small dynamic range [7]. Additionally, this larger accumulator results in additional precision for BF16 and FP32.

The handling of the FP32 addend is similar to a floating-point adder: The sum of products is indeed a floating-point number (possibly non-normalized) of exponent E_{Acc} and of significand size L. However, contrary to a classic FP adder, the proposed architecture always shifts the FP32 significand, since it is much smaller than L. This is actually similar to the addition performed in the baseline operator [20], with the notable difference that the shift amount of the addend Z is not computed from a constant (MSB 30 of the accumulator), but from the exponent E_{Acc}.

Finally, a Leading Zero Count and a Shift are performed to normalize the sum and prepare it for the final rounding.

b) Overhead over the baseline FP16 DPA: Compared to the baseline implementation, the size of the multipliers increases to 12×12 to support E9S12. The accumulation size of $L = 81$ is kept to continue to allow for exact FP16 calculations, which implies that the alignment shifters and adder trees are the same size as in the baseline operator.

E_{max} must be computed with a comparator tree, it can be done in parallel with the multiplication of significands. As the significands are aligned with E_{max}, it is simpler to use a right shifter instead of the previously used left shifter. The significand product Sig_i is shifted by $E_{\text{max}} - E_i$.

c) Subnormal handling: As discussed in Sect. II-D, the baseline FP16 operator does not need subnormal output logic, as the range for the FP16 product is much smaller than the range for the FP32 result.

However, with the E9S12 format, it becomes possible to create a subnormal output from normal inputs. Denormalization logic is added to the operator, where the final shift computation takes into account the Leading Zero Count and Exponent. The shift performs an incomplete normalization of the accumulator so that the significand is correctly aligned for the rounding.

d) Zero detection: When splitting an FP32 number into two E9S12 numbers, it is important to independently detect if the two separate parts are zero. Although most flags are duplicated (i.e. $X = +\infty \rightarrow X^h = +\infty \wedge X^l = +\infty$), accurately detecting zeros helps catching bugs in the rest of the operator. The case $X^l = 0, X \neq 0$ can happen when the fraction of X is empty on the lower half. The case $X^h = 0, X \neq 0$ can happen when X is a small subnormal.

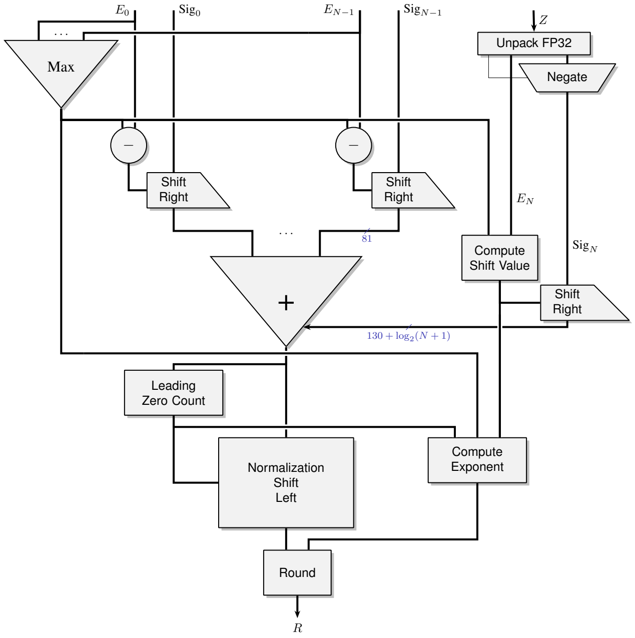

Fig. 9. Dot product operator for E9S12.

C. Support of the FP32 Fused-Multiply Add

A dot product operator in the E9S12 format

$$R = \circ(\phi(X'_0 \times Y'_0 + \ldots X'_{N-1} \times Y'_{N-1}) + Z)$$

as described in the previous section can be used to emulate a correctly rounded FP32 Fused Multiply-Add (FMA) operation.

If all FP32 products except one $U_k \times V_k$ are zero, the result is correctly rounded. Indeed, the product is split into four terms $U_k \times V_k = U_k^h \times V_k^h + U_k^h \times V_k^l + U_k^l \times V_k^h + U_k^l \times V_k^l$ but those terms have a maximum exponent difference of 24. Since the size of each term is also 24, the four products can be exactly represented on 48 bits (the size of the FP32 significand multiplication). Therefore, as soon as $L > 48$, no information is lost in the sum, and the product $U_k \times V_k$ is exact. Implementing the method of [20] then ensures the correct rounding of the addition of Z.

D. Alternative Operator Architectures

a) Accumulator-based architecture for arbitrary size dot product: When the DPA operator is used to emulate a larger dot product operator with an arbitrary size D, an alternative architecture, depicted in Fig. 10, can be used where the addend is not exposed to the user. It is replaced with an accumulator that can only be reset to 0. This accumulator is here considered as a product; for instance, its exponent can be chosen as the maximum exponent. In this architecture, the final normalization shift and round components are only used when the D products have been added, to round the result into an FP32 number.

Compared to an FP32 addend, this approach increases accuracy for large dot products: the whole accumulator of

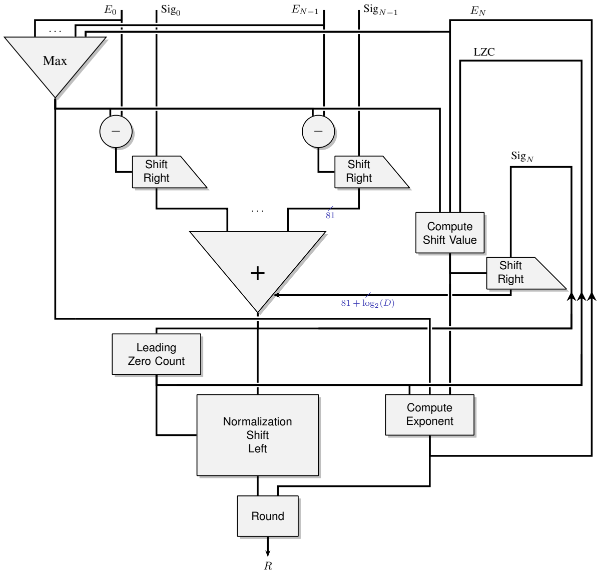

Fig. 10. Dot product operator for E9S12, with loop on the accumulator.

$81 + \log_2(N)$ bits loops back into the computation instead of the 24 bits of precision of an FP32 addend.

An FP32 number can still be added by first splitting it as two E9S12 and inputting as a product with 1. This is trivial in a single dot product, and takes multiple operations in a matrix multiply add operator to initialize every addend independently. This also does not always provide the correctly rounded sum of an FP32 and an FP16 dot product (consider the case where this FP32 has a very large or very small exponent compared to the accumulator exponent, leading to bits being lost in the shift). For the same reason, this alternative architecture cannot emulate the FP32 FMA.

In short, this variant is a trade-off, as it improves the overall accuracy over many iterations but loses precision over one unique computation. The lack of correct rounding also reduces predictability in distributed systems and software simulation.

b) Multi-addition operator: The architecture of Fig. 10 can also be used to compute the sum of D FP16 or FP32 terms: $R = \sum X_i$. In this case, the floating-point inputs X_i are unpacked (Fig. 6) and the significands are converted to two's complement before being input into the architecture of Fig. 10.

c) Complex matrix multiplication: Mixed-mode matrix multiplication without addend is also useful to accelerate FP16 Fast Fourier Transforms [29]. This is illustrated with the radix-4 decimation in frequency butterfly computation [30]:

$$\begin{aligned} y_l &= (x_l + x_{l+\frac{N}{2}}) + (x_{l+\frac{N}{4}} + x_{l+\frac{3N}{4}}) \quad , \\ z_l &= ((x_l - x_{l+\frac{N}{2}}) - j(x_{l+\frac{N}{4}} - x_{l+\frac{3N}{4}}))\omega_N^l \quad , \\ g_l &= ((x_l + x_{l+\frac{N}{2}}) - (x_{l+\frac{N}{4}} + x_{l+\frac{3N}{4}}))\omega_N^{2l} \quad , \\ h_l &= ((x_l - x_{l+\frac{N}{2}}) + j(x_{l+\frac{N}{4}} + x_{l+\frac{3N}{4}}))\omega_N^{3l} \quad . \end{aligned}$$

For a complex FFT of N samples, each of the $\log_4 N$ stages executes $\frac{N}{4}$ independent butterfly computations. With mixed-mode arithmetic, the $\{x_i\}$ are FP16 complex numbers, while the $\{y_l, x_l, g_l, h_l\}$ are FP32 complex numbers that are converted back to FP16 before being stored to memory.

Each radix-4 butterfly can be implemented as a $[1, 8] \times [8, 8]$ real matrix multiplication. A $[k, 8] \times [8, 8]$ multiplication performs k simultaneous butterfly computations. Assuming interleaved real and imaginary parts, the extension of the operator then expands the ω multiplicands as $[2, 2]$ submatrices.

V. Experimental Results

A. Operator Validation

This operator was implemented within the FloPoCo framework [31], which includes MPFR-powered test bench generation. When the operator takes FP16 multiplicands, a correctly rounded result is expected. With BF16 and FP32 multiplicands, the test case is accepted when the result is a faithful rounding of the exact result computed in MPFR, and rejected otherwise. When rejected, the case is checked by hand to verify that it is not a bug but a normal behavior of the operator.

For every input mode, the FMA is expected to be correctly rounded: if the number of non-zero product is 1 or less, the test bench expects a correctly rounded result.

The FloPoCo framework encourages the definition of standard test cases. Here, these include the tests of negative zeros and subnormals, as well as debug tests for development purposes. It also allows for directed random tests, where the random number generator is biased towards increasing the probability of some rare but important situations. Here, directed random tests enable the verification of the FMA mode, forcing all products but one to be zero. The probability of cancellation cases (where products can have very close exponents and different signs) is also increased. Finally, directed random tests are also used to increase the frequency of subnormal inputs and outputs, as this is often an error-prone part of the operators.

B. Synthesis Results

This section compares DPA operators of size 16 for FP16 and BF16, and thus of size 4 for FP32 multiplicands. The actual pipelining of the chosen operator will be highly dependent of its integration in the larger context of a MMA unit. For this reason, we prefer to compare combinatorial operators, which are synthesized for one clock cycle at 333MHz to allow for later pipelining in 3 clock cycles at 1GHz. The operators have been synthesized with the Synopsys Design Compiler NXT for the TSMC 4FFC node.

Here, DP16$_{\text{FP16-BF16}}$4$_{\text{FP32}}$A is the combined FP16, BF16 and FP32 operator based on the E9S12 format. It is compared to an alternative with two separate operators: DP16$_{\text{FP16-BF16}}$A, a combined FP16 and BF16 dot-product operator dedicated to 16-bit operands, and DP4$_{\text{FP32}}$A, a correctly rounded FP32 dot product operator with subnormal support as described in [24]. The results of the synthesis are shown in Table I.

The combined operator has a significantly smaller area compared to having two operators DP16$_{\text{FP16-BF16}}$A and DP4$_{\text{FP32}}$A (-40%), and this is correlated with the leakage power (-39%).

		Power	
Operator	Area	Leakage	Total
	(μm^2)	(nW)	(mW)
DP16$_{\text{FP16}}$A [21]	1796	477	1.83
DP16$_{\text{FP16-BF16}}$A	2343	602	2.11
DP4$_{\text{FP32}}$A [24]	1865	476	1.87
DP4$_{\text{FP32}}$A [24] & DP16$_{\text{FP16-BF16}}$A	4208	1078	2.11
DP16$_{\text{FP16-BF16}}$4$_{\text{FP32}}$A	2504	657	2.62

TABLE I

SYNTHESIS RESULTS FOR THE DIFFERENT OPERATORS CONFIGURATIONS.

However, total power consumption increases (+24%) as the operator is less specialized.

VI. CONCLUSIONS

This work is motivated by the extension of a Matrix Multiply Add (MMA) unit, originally designed for deep learning applications, for use in FP32 numerical applications. The building block for this MMA unit is a Dot Product Add (DPA) operator with FP16, BF16 or FP32 multiplicands, an FP32 addend and an FP32 result.

The proposed DPA operator performs a dot product between vectors of $4N$ 16-bit floating-point elements or N FP32 elements. This operator accepts FP32 multiplicands by decomposing each of them into a pair of numbers represented in a suitably designed internal format, consisting of 12 bits of significand and 9 bits of exponent.

This combined FP16, BF16 and FP32 DPA operator removes the need for implementing double-word arithmetic in software, while providing the full precision and dynamic range of FP32 arithmetic. In case of FP16 multiplicands and FP32 addend, this operator computes the correctly rounded result. It also correctly emulates the FP32 Fused Multiply and Add operation when all product terms except one are zeros.

The synthesis results of non-pipelined operators for a 4nm technology node show that significant savings in area (-40%) and leakage power (-39%) are achieved with the combined DPA operator, compared to the specializing of the two corresponding operators. However, operating the combined operator increases the total power by 24%.

REFERENCES

[1] IEEE, “754-1985 - IEEE Standard for Binary Floating-Point Arithmetic,” 10 1985.

[2] Intel, “BFLOAT16 – Hardware Numerics Definition Revision 1.0,” 11 2018.

[3] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, “Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks,” in *33rd International Conference on Neural Information Processing Systems (NIPS)*, 2019.

[4] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, “NVIDIA Tensor Core Programmability, Performance & Precision ,” in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2018.

[5] 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 *26th Symposium on Computer Arithmetic (ARITH)*, IEEE, 2019.

[6] B. Hickmann and D. Bradford, “Experimental Analysis of Matrix Multiplication Functional Units,” in *26th Symposium on Computer Arithmetic (ARITH)*, IEEE, 2019.

[7] D. R. Lutz, A. Saini, M. Kroes, T. Elmer, and H. Valsaraju, “Fused FP8 4-Way Dot Product With Scaling and FP32 Accumulation,” in *31st Symposium on Computer Arithmetic (ARITH)*, IEEE, June 2024.

[8] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhauser, 2018.

[9] M. Fasi, N. J. Higham, F. Lopez, T. Mary, and M. Mikaitis, “Matrix Multiplication in Multiword Arithmetic: Error Analysis and Application to GPU Tensor Cores,” *SIAM Journal on Scientific Computing*, 2023.

[10] G. Henry, P. T. P. Tang, and A. Heinecke, “Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations,” in *26th Symposium on Computer Arithmetic (ARITH)*, IEEE, 2019.

[11] T. J. Dekker, “A floating-point technique for extending the available precision,” *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[12] W. R. Dieter, A. Kaveti, and H. G. Dietz, “Low-cost microarchitectural support for improved floating-point accuracy,” *IEEE Computer Architecture Letters*, vol. 6, no. 1, pp. 13–16, 2007.

[13] P. Valero-Lara, I. Jorquera, F. Lui, and J. Vetter, “Mixed-Precision S/DGEMM Using the TF32 and TF64 Frameworks on Low-Precision AI Tensor Cores,” in *International Conference on High Performance Computing, Network, Storage, and Analysis*, p. 179–186, ACM, 2023.

[14] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, “Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers,” in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, IEEE, 2019.

[15] F. de Dinechin and M. Kumm, *Application-Specific Arithmetic*, ch. Basic Floating-Point Operators. Springer, 2024.

[16] R. Kirchner and U. Kulisch, “Accurate arithmetic for vector processors,” *Journal of parallel and distributed computing*, 1988.

[17] U. W. Kulisch, *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, 2002.

[18] A. Knofel, “Fast hardware units for the computation of accurate dot products,” in *10th Symposium on Computer Arithmetic (ARITH)*, pp. 70–71, IEEE, 1991.

[19] F. de Dinechin, B. Pasca, O. Creţ, and R. Tudoran, “An FPGA-specific approach to floating-point accumulation and sum-of-products,” in *Field-Programmable Technologies*, pp. 33–40, IEEE, 2008.

[20] N. Brunie, “Modified fused multiply and add for exact low precision product accumulation,” in *24th Symposium on Computer Arithmetic (ARITH)*, pp. 106–113, IEEE, 2017.

[21] N. Brunie, “Towards the basic linear algebra unit: replicating multi-dimensional FPUs to accelerate linear algebra applications,” in *54th Asilomar Conference on Signals, Systems, and Computers*, pp. 1283–1290, IEEE, 2020.

[22] O. Desrentes, B. D. de Dinechin, and J. Le Maire, “Exact dot product accumulate operators for 8-bit floating-point deep learning,” in *Euromicro Conference on Digital System Design (DSD)*, 2023.

[23] Y. Tao, G. Deyuan, F. Xiaoya, and J. Nurmi, “Correctly rounded architectures for floating-point multi-operand addition and dot-product computation,” in *24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 346–355, IEEE, 2013.

[24] O. Desrentes, B. D. de Dinechin, and F. de Dinechin, “Exact Fused Dot Product Add Operators,” in *30th Symposium on Computer Arithmetic (ARITH)*, IEEE, 2023.

[25] H. H. Saleh and E. E. Swartzlander, “A floating-point fused dot-product unit,” in *International Conference on Computer Design*, pp. 427–431, IEEE, 2008.

[26] D. Kim and L.-S. Kim, “A floating-point unit for 4D vector inner product with reduced latency,” *IEEE Transactions on computers*, vol. 58, no. 7, pp. 890–901, 2008.

[27] 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 *27th Symposium on Computer Arithmetic (ARITH)*, IEEE, 2020.

[28] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, “Numerical behavior of NVIDIA tensor cores,” *PeerJ Computer Science*, 2021.

[29] B. Li, S. Cheng, and J. Lin, “tcFFT: A Fast Half-Precision FFT Library for NVIDIA Tensor Cores,” in *International Conference on Cluster Computing (CLUSTER)*, pp. 1–11, IEEE, 2021.

[30] E. Chu and A. George, *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

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