Vectorized Fused Dot Product

Thomas Ferrere, Jeremy Zolnai-Lucas, Tai Li and Adedotun Adeyemo
Imagination Technologies, Kings Langley, UK
{first.last}@imgtec.com

***Abstract*—We present a multiprecision dot product unit for accelerating matrix multiplications in a variety of number formats, such as INT8, FP4, FP8, FP16 and BF16. The architecture implements fused arithmetic, which increases the performance density by removing intermediate normalization and rounding steps. It natively supports floating-point and integer formats and allows for input and output precisions to be set independently. For half-width inputs, the number of terms in the dot product is doubled. The architecture achieves a high level of logic reuse through vectorization, enabling arithmetic operators to process as many terms as their width allows. Vectorization is commonly applied to elementwise operations like multiplication, and we show that this technique extends to reduction operations such as summation. A central idea is to represent each full-width term as two half-width terms in the alignment and summation stages. Accuracy analysis demonstrates widths of 32 and 16 bits are sufficient when accumulating FP16 products in FP32 precision and FP8 products in FP16 precision, respectively. The resulting design, which is formally verified, achieves an area saving of 31% compared to equivalent scalar designs processing each number format separately.**

***Index Terms*—computer arithmetic, matrix multiplication, floating point, mixed precision, logic minimization**

I. Introduction

Contemporary artificial intelligence (AI) workloads require the hardware acceleration of matrix multiplications in integer and floating-point number formats. The basic computational step is an accumulating dot product $a \cdot b + c = d$ where a, b vectors are low precision and c, d are higher precision. Formats for a, b include INT8, FP4, FP8 [1], FP16 [2] or BF16 [3] and formats for c, d include INT32, FP16 or FP32 [2]. Lower precision entails higher levels of acceleration. The problem is how to efficiently implement arithmetic over multiple formats while scaling the performance inversely with the bitwidth of operands.

Previous architectures addressed this problem in two different ways.

1) In vector units operating according to the single instruction, multiple data (SIMD) model [4], [5], processing elements operate over a fixed-width vector partitioned into a variable number of elements according to the selected number format. Such architectures can execute, for example, 1 FP64 multiplication, 2 FP32 multiplications, or 4 FP16 multiplications through the same datapath [6], [7]. The operation produces as many results as there are independent sets of operands. This can involve the reuse of logic elements within shifters, adders, etc. for computing bits associated with the same result or separate results, depending on the format. Each multiplication, addition or fused multiply-add (FMA) is performed in parallel and correctly rounded to the selected format.
2) In deep learning accelerators [8], [9], processing elements are optimized for specific number format combinations. Such units achieve high-density implementation of dot products for formats such as INT8 or FP8 by fusing several multiply-accumulates into one step [10], [11]. A fused dot product unit may compute, for example, $a_0b_0 + \ldots + a_3b_3 = c$, with a and b vectors in FP8 vectors, in a single step. These units typically omit or alter rounding steps of intermediate sums, such that a single result is computed and rounded to the destination format, for example FP32.

Partial results c can be added as part of the fused multiply-add or dot product (early accumulation), or separately through dedicated floating-point additions (late accumulation). With this, either solution can perform matrix multiplications of arbitrary size.

The first solution supports multiple formats through conversions and homogenous atomic operations. Supporting multiple formats promotes arithmetic reuse; however the lack of specialization still results in high circuit area. The second solution supports multiple formats through dedicated dot product units, sized for a given algorithm and number format combination. While each datapath implementation is adjusted according the bitwidth of its operands, dedicated units add to the overall circuit area.

We address above limitations by introducing a multiprecision fused dot product unit, with a vectorized datapath. The proposed architecture implements dot products over a, b in a variety of formats including INT8, FP8 and FP16, and outputs a result c in INT32 or FP32. It computes dot products twice as deep when the bitwidth is halved. When configured to operate over 256-bit vectors, it may compute $a_0b_0 + \ldots + a_{15}b_{15} = c$ for a, b in FP16 or $a_0b_0 + \ldots + a_{31}b_{31} = c$ for a, b in FP8. The dot product is broken down into stages, each of which is either dedicated to floating-point or has the ability to process both integer and floating-point through logic merging. Furthermore, each stage can process either full-width values, or vectors of two half-width values. This requires turning scalar operators into reconfigurable, scalar or vector operators, through a process we refer to as *vectorization*.[1] Target operators include multipliers, shifters, comparators and

[1]This is orthogonal to the concept of vectorization whereby code intended for a scalar architecture is transformed so as to make use of vector instructions.

adder structures. Vectorization is most beneficial whenever internal precision requirements are proportional to the bitwidth of operands, something we demonstrate relative to the target application.

Our implementation processes a superset of INT8, FP8, FP16 and BF16. Further formats are supported through conversion; for example FP4 is converted to an FP8 format. The architecture uses late accumulation with a short return path to reduce the cost of keeping the pipeline fully utilized. When the final output is requested in FP16, the result is computed as $c + d_0 = d_1$ with d_0, d_1 in FP16. Otherwise the result is natively accumulated in INT32 or FP32. The INT8, FP4 and FP8 formats are taken compatible with the Open Compute MX specification [1]. The FP16 and FP32 formats are as per the IEEE-754 standard [2].

The architecture of the multiprecision fused dot product is presented in Section II and its detailed implementation exposed in Section III. The verification of our implementation is discussed in Section IV and an exploration of its accuracy is performed in Section V. Synthesis results are reported and analysed in Section VI, followed by a discussion of related works and perspectives in Section VII.

II. ARCHITECTURE

The proposed architecture is centred on a fused dot product unit, which is made to process integer and floating-point, and multiple precisions by the vectorization of its datapath. The inputs to the dot product unit are converted to a set of internal formats whose purpose is to unify the packed encodings and simplify the dot product implementation by normalizing the floating-point data. The outputs of the dot product unit are accumulated through a dedicated integer and floating-point adder, which also handles multiple precisions as required.

A. Fused dot product

The computation of the proposed fused dot product architecture is organized into five stages:

- **Multiplication**: The integers or mantissas (including the implied one) of floating-point inputs are multiplied.
- **Exponent maximum**: Exponents of floating-point inputs are added into product exponents and their maximum is computed.
- **Mantissa alignment**: The mantissa products are shifted according to their respective exponent differences to the maximum.
- **Summation**: The integer and aligned mantissa products are summed using a shared carry-save and carry-propagate adder structure.
- **Normalization**: The mantissa sum is renormalized according to its number of leading zeroes and its associated exponent adjusted accordingly.

The proposed architecture builds on a fused floating-point dot product [12] but adds the ability to process integers. This is achieved through the two key stages of multiplication and summation. When operating over integers, the maximum exponent stage is disabled and the alignment and normalization stages are bypassed, e.g. by forcing the associated shifts to a constant.

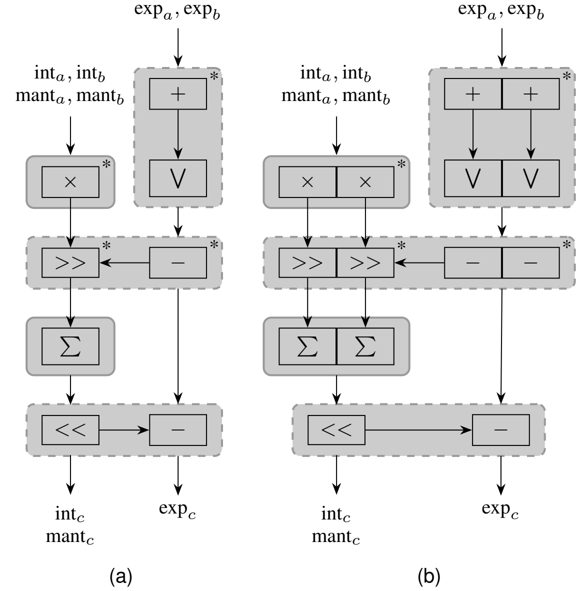


Fig. 1. Fused dot product architecture with (a) merged integer and floating-point, and (b) vectorized datapath. Elementwise operations are starred to denote multiple instances, one per input pair. Reducing operations are shown as $\bigvee$ for maximum and $\sum$ for summation. Stages are overlaid in gray and those specific to floating point have a dashed outline.

Another notable feature of our architecture is its reduced internal precision, which can result in some products being truncated or rounded off. This happens whenever any of their significant bits underflow the internal representation. A reduced precision is practically required for number formats with a large exponent range[2], and reduces the overall circuit area by closely matching the accuracy requirements.

Figure 1a shows the architecture of a merged integer and floating-point fused dot product.

B. Vectorization

Vectorizing a stage of processing may follow one of two strategies:

- Implementing two separate sets of arithmetic operators, each processing k inputs. Processing smaller-width inputs in a wider operator is trivial and is achieved, e.g., by zero-padding. One set of operators is designed to process both full-width and half-width values, while the other to process half-width values only.
- Implementing a single set of operators, processing either k full-width or $2k$ half-width values. Processing packed half-width inputs in a full-width operator could corrupt the data if information was to flow between unrelated half-width values. This is addressed by gating off selected bits in half-width mode.

[2]At least $16 + 508 = 524$ bits are required to represent all BF16 products in signed fixed point.

These strategies are applicable to each stage, with the second strategy most beneficial when half-width values can be densely packed into full-width values.

Multiplication and alignment are elementwise operations, generating one output per input element, and can be vectorized by using k operators, each capable of processing either a single full-width value or two half-width values under the second strategy. Exponent maximum and summation are reducing operations. Their vectorization employs a further reduction step to combine the two results from the vectorized operator into a single result. Normalization is a scalar operation in our architecture and vectorization is not applicable.

Figure 1b shows the architecture of the fused dot product unit with vectorization applied.

III. Implementation

We implement a dot product unit with depth $k = 16$ for 16-bit input formats and depth $2k = 32$ for 8-bit input formats. This enables the processing of 16-bit values at single rate and 8-bit values at double rate. A large depth k increases the dot product unit density by removing intermediate normalizations, and reduces the required number of downstream accumulations.

A. Internal Precision

Integer inputs are given as INT8, which can be signed or unsigned. Floating-point inputs are given as BF16, FP16, or FP8, which can be E5M2 or E4M3. Each stage of the design must be sized to accommodate the widest inputs the stage may encounter, under both full- and half-width inputs. To this end, it is convenient to first convert each input into a superset of the 16-bit and 8-bit floating-point formats above.

Furthermore, it is beneficial to normalize the internal representations, i.e. extend the format to be able to represent denormals as normals. This ensures that no additional precision is lost in alignment, where denormal operands would have otherwise caused more significant bits to be shifted out.

We therefore introduce the formats E8N10 and E5N3, where N is used in place of M as the mantissa is strictly normalized. Their data widths are defined as follows:

- E8N10 has an exponent of $\max(5, 8) = 8$ bits and a mantissa of $\max(7, 10) = 10$ bits.
- E5N3 has an exponent of $\max(4, 5) = 5$ bits and a mantissa of $\max(2, 3) = 3$ bits.

The BF16 format has denormals flushed to zero[3] and FP16 denormals are representable with an 8-bit exponent, so 8 bits are sufficient for the superset E8N10.

An exponent of 6 bits is avoided when normalising E5M2, by noticing that its non-exceptional values lie in the range $[2^{-16}, 2^{15})$. With an exponent bias of 16, the entire non-exceptional range becomes representable within the 5-bit exponent.

[3]In the absence of a universally ratified BF16 standard, and consistent with design choices of major alternative hardware implementations, this architecture does not support subnormals for the BF16 format.

TABLE I
Bitwidth of integer and floating-point internal data.

Format	E8N10	E5N3	INT8
$\exp_a$, $\exp_b$	8	5	–
mant_a, mant_b, int_a, int_b	11	4	8
$\exp_p$	9	6	–
mant_p, int_p	22	8	16
mant_q	32	16	–
mant_r, int_r	36	21	21

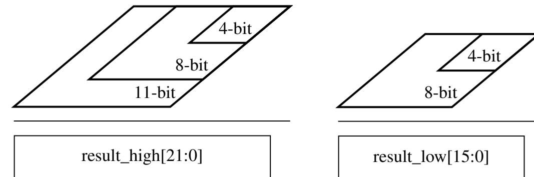


Fig. 2. Partial product array reuse in a vectorized multiplication stage.

The full representation uses flags *isnan*, *isinf*, and *iszero* to keep track of exceptional inputs. These flags are private to each input but can be combined to inform exceptional outputs. The *isnan* and *isinf* flags are discarded once exceptional outputs are determined and the *iszero* flags discarded after further forcing product mantissas and exponents to zero.

Following the discussion in Section II-B, an internal precision for accumulation that is close to a factor of two larger for full-width inputs than for half-width inputs is beneficial to the vectorization of the alignment and summation stages. The minimum internal precision required to represent the product of two 8-bit integers is 16 bits. We select this width of 16 bits as the internal precision for half-width inputs, and an internal precision of 32 bits for full-width inputs therefore follows. We demonstrate in Section V that these widths are sufficient to ensure good accuracy.

Internal data widths in each processing stage are summarized in Table I and derived as follows:

- The exponents $\exp_a$ and $\exp_b$ are added to form product exponents $\exp_p$, which are one bit wider.
- The integer values int_a and int_b are multiplied to form products int_p, which are twice as wide. Likewise the mantissas mant_a and mant_b are multiplied to form product mantissas mant_p, which are twice as wide.
- The unsigned product mantissas mant_p are extended and aligned to the target internal precision with the sign bit applied to form mant_q, which are 32-bits signed values for E8N10 and 16-bit signed values for E5N3. The 16-bit products int_p for INT8 are left unchanged.
- The terms int_p and aligned mantissas mant_q are summed into int_r and mant_r, whose width increases by $\log(k)$ for k-deep dot products, and therefore extend to a total to 36 bits for E8N10 and 21 bits for E5N3 or INT8.

B. Processing stages

1) Multiplication: The multiplication stage consists of $k = 16$ processing elements, which can each process either a single 11-bit multiplication for E8N10, two 4-bit multiplications for E5N3, or two 8-bit multiplications for INT8 inputs. Each processing element reuses a single 11-bit multiplier for the first term of all three cases, while an additional 8-bit multiplier is used for the second term when present, as illustrated in Figure 2.

2) Exponent maximum: The exponent maximum stage consists of exponent additions and maximum reduction.

The 9-bit adders that perform the exponent additions in E8N10 are reused for one half of the terms in E5N3, with dedicated 6-bit adders used for the other half.

The exponent maximum is processed by independent maximum reduction trees with depth $k = 16$ and widths of 9 and 6 bits, respectively. The first reduction tree inputs are either the 9-bit product exponents of E8N10 or half of the 6-bit product exponents of E5N3, and the second reduction tree inputs are the other half of the 6-bit exponents of E5N3. A final reduction step combines the two outputs for E5N3.

3) Mantissa alignment: The mantissa alignment stage consists of exponent differences and mantissa shifts, each requiring k or $2k$ independent operations depending on the number format.

The 10-bit adders that perform subtraction (in two's complement) of E8N10 exponents are reused for one half of the E5N3 exponents, with dedicated 7-bit adders for the other half.

The processing elements for mantissa right-shifts are vectorized shifters, as illustrated in Figure 3. The output of each shifter is 32 bits wide, representing either a single 32-bit result for E8N10 or two 16-bit results for E5N3. For 16-bit inputs, the mantissa product entering the shifter is 22 bits wide, and the exponent difference (i.e., the shift amount) is a 9-bit unsigned value. The input is zero padded at the least significant end, so that input and output widths into the shifter are the same, at 32 bits. Similarly, for 8-bit inputs, the mantissa product is 8 bits wide (zero padded to 16 bits on input), with a 6-bit exponent difference.

The shifter follows a standard barrel shifter structure with $\lceil \log_2(n) \rceil$ shifting stages for an n-bit input, as shown in Figure 3a. Each shifting stage i shifts the data by 2^i bit positions, for $i = 0, 1, \dots, \lceil \log_2(n) \rceil - 1$. A sticky bit is also produced by the shifter, set whenever any nonzero bit is shifted out; it is later used to inform the rounding of the shifted mantissa.

When operating in half-width mode (for E5N3), the control signal *vec* shown in Figure 3b is deasserted. This disables cross-lane propagation by forcing shifted-in bits from the upper half lane into the lower half lane to zero at each stage of the barrel shifter. This ensures that the upper and lower 16-bit operands operate independently within the same 32-bit lane.

A sticky bit associated to each shifting stage i is generated by OR'ing together the bits that are shifted out. A further sticky bit is set when the shift amount is greater than the overall output width of the shifter and the input data is non-zero. The final output sticky bit is produced by OR'ing together the results from all shifting stages of the barrel shifter with the additional sticky bit, as shown in Figure 3a.

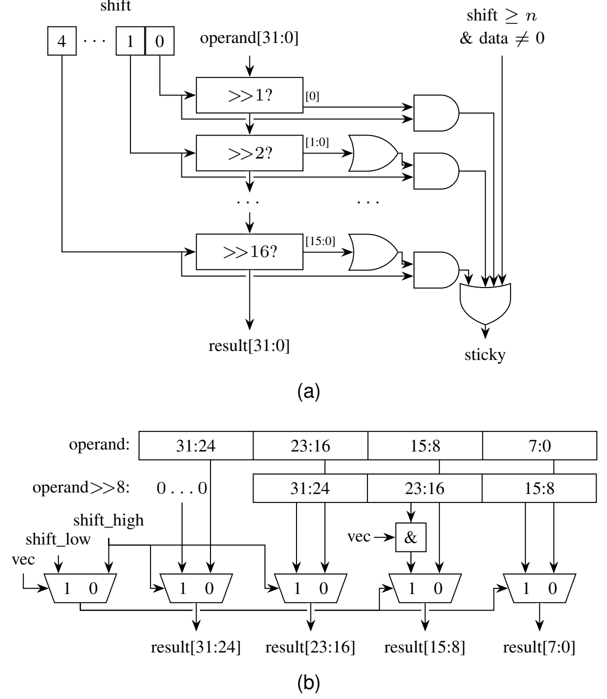


Fig. 3. Architecture of (a) scalar barrel shifter with sticky bit generation logic, and (b) vectorized barrel shift layer (shown for a shift by 8 bit positions).

The sticky bit logic is shared across precision modes; the lower half of the sticky bit path is shared between 16-bit and 32-bit operations, while the upper half is used exclusively for the upper 16-bit operand in 16-bit mode. Since 32-bit operands can require a larger shift range, the 32-bit datapath includes one additional shifting stage.

After the alignment shifter, each unsigned product is put into two's complement form by applying the product sign bit. When negating the product, an additional increment bit is introduced due to one's complement inversion.

An intermediate rounding is also performed to reduce the aligned product to the accumulator's intermediate precision. This rounding may also introduce an increment when rounding up. Notably, the increment from two's complement conversion and the increment from rounding are mutually exclusive, and so at most a single increment needs to be added alongside each product term.

For INT8, the alignment stage is skipped and products feed the summation stage directly.

4) Summation: The summation stage consists of four processing elements: two carry-save adder (CSA) trees, each computing the sum of $k = 16$ (signed) numbers that are 16 bits wide, one configurable 4:2 reduction stage, and one carry-propagate adder (CPA).

Each half feeds its carry-save adder tree directly from respective upper and lower half-lanes of vectorized shifters. In full-width mode (for E8N10), each 32-bit aligned mantissa

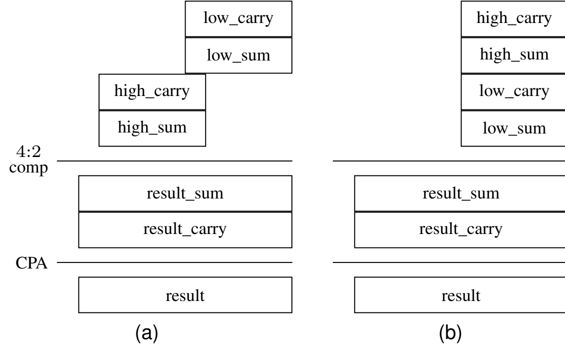


Fig. 4. Resolving carry-save halves into a single result in (a) full-width mode and (b) half-width mode.

product is therefore partitioned into a most significant and least significant half and driven to respective 16 bits wide CSA trees. In half-width mode (for E5N3 or INT8), each 16-bit aligned mantissa product is driven into one of the 16 bits wide CSA tree directly. The CSA outputs are then combined to create a single sum, as shown in Figure 4. In half-width mode, the two carry-save halves are aligned vertically, so that the calculated total is the sum of all $2k$ aligned mantissas of 16 bits each. In full-width mode, the carry-save result from the upper half is left-shifted by the width of the lower half, i.e. 16 bit positions, so that the final result is the sum of all k 32-bit aligned mantissa products. In both scenarios, the carry-save half results are reduced again into a single carry-save result using a 4:2 compressor, and then summed into a final result using a shared CPA.

5) Normalization: The normalization stage consists of two processing elements, a left-shift based on the number of leading zeroes, and an exponent adjustment, which subtracts the number of leading zeroes from the previously computed maximum exponent. Finally the result is rounded to FP32. These operators are scalar, i.e. they process a single value. The later accumulation unit supports accumulating to FP32 or FP16 natively using mixed-precision arithmetic. When an integer result is required, the normalization stage is bypassed and the result accumulated to INT32.

IV. Verification

Design verification is a key consideration when introducing an optimized architecture. The large input space makes exhaustive verification of floating-point dot product hardware infeasible when using simulation-based techniques. This design was verified using formal verification techniques with the aid of industry-leading tools. Proof convergence is one area of difficulty with formal verification and there are two key features of this design critical in addressing this problem. First is the staged approach to computation used in this architecture. With this, we could use intermediate results to aid the formal tool in reaching full proofs. Second, the vectorization of the datapath does not alter the arithmetic behaviour of the dot product, so creating behavioural models for equivalence checking is straightforward. The overall verification is broadly organized in two stages:

1) Arithmetic correctness: check that each dot product transaction is computed correctly, by performing an equivalence check of the design's output against a validated C-model (also used for empirical error measurement) and a behavioural implementation;
2) Data integrity: check that the datapath calculation cannot be corrupted by the state of the hardware over an infinite time period, i.e. proving that the integrity of input and output data between transactions is preserved.

This strategy hinges on the availability of a clear, unambiguous specification. The present design is fully specified by the choice of internal precision, normalization and rounding method at each stage.

V. Accuracy

The accuracy of our implementation is evaluated using theoretical, backward error analysis [13] and empirical, forward error measurements on random data. Backward error analysis is useful to derive worst-case bounds, where forward error is theoretically unbounded.

Given vectors a, b, the forward error compares the computed result $\hat{c}$ to the exact result $a \cdot b = c$, while a backward error compares the effective arguments a and b to exact arguments $\hat{a}$ and $\hat{b}$ such that $\hat{a} \cdot \hat{b} = \hat{c}$. Errors are measured in units in the last place (ulp), defined as the difference between the nearest floating-point number and the next one away from zero.[4] The ulp of a floating-point value with an exponent e and p mantissa fraction bits is 2^{e-p}. When the ulp error is less than 0.5, the exact value rounds to the nearest floating-point value, and therefore its ulp derives straightforwardly from the computed exponent.

A. Backward error analysis

In this analysis, we only account for internal rounding errors and leave out the final rounding error due to normalization in the destination format, which can dominate and is present regardless of the choice of summation method. Assuming k terms, only $k-1$ terms may be subject to underflow in alignment. This is because the shared format is wide enough to hold the product mantissas before shift, and the term with the largest exponent will not be shifted. For a product with p fraction bits aligned relative to a shared exponent g, the rounding error due to alignment is at most 2^{g-p-1}, assuming round to nearest. Hence, the combined absolute error is at most $(k-1)2^{g-p-1}$.

Let $a_i = m2^e$ and $b_i = n2^f$ be the two factors occurring in one term of the dot product. For factors with q fraction bits, an ulp error $\varepsilon < 0.5$ affects their product by $b_i 2^{e-q}\varepsilon$ or $a_i 2^{f-q}\varepsilon$, which is at least $2^{e+f-q}\varepsilon$ when a_i and b_i are normalized, meaning $1 \leq m, n < 2$. Thus dividing an absolute

[4] This definition is easy to understand and yields uniform error bounds. A more mathematically rigorous definition could be considered [14].

error on $a_i b_i$ by 2^{e+f-q} yields a bound on an equivalent ulp error on factor a_i or b_i. The combined absolute error may be transferred to the term with the largest exponent, and in that case $e+f=g$. Hence assuming k terms, for aligned products with p bits and factors with q bits, the backward error is at most $(k-1)2^{g-p-1-e-f+q} = (k-1)2^{q-p-1}$ ulp. That is, for all vectors a and b there exists $\hat{a}$, $\hat{b}$ such that the algorithm computes $a \cdot b \approx \hat{c}$ and $\hat{a} \cdot \hat{b} = \hat{c}$ with $\hat{a}$ and $\hat{b}$ having an error at most $(k-1)2^{q-p-1}$ ulp relative to a and b. Note that such backward errors only alter one component of a or b and tighter bounds may be derived if multiple components of a and b are altered.

The implementation uses the internal representations introduced in Section III-A. Input mantissas range over the interval $[1, 2)$. They have 1 integer bit, 3 fraction bits for E5N3, and 10 fraction bits for E8N10. Aligned product mantissas therefore range over the interval $(-4, 4)$ after application of the sign. They have 3 signed integer bits, 13 fraction bits for E5N3, and 29 fraction bits for E8N10. Round to nearest is used in alignment. From the above, we can derive the following backward error bounds:

- For E4M3, errors on c are commensurate with an error on a or b chosen to be at most $31 \times 2^{3-13-1} < 0.0152$ ulp.
- For FP16, errors on c are commensurate with an error on a or b chosen to be at most $15 \times 2^{10-29-1} < 0.0000144$ ulp.

These bounds become $31 \times 2^{2-14} < 0.00757$ ulp for E5M2 and $15 \times 2^{7-30} < 0.00000179$ ulp for BF16. Therefore internal rounding errors are expected to be small relative to quantization errors, which may be as large as 0.5 ulp.

B. Empirical error measurement

An empirical measurement of the forward error is obtained in simulation. We drive random vectors into the C-model of our unit, and compare its output against the infinitely precise result. The C-model is formally verified to match the bit-level behaviour of the register transfer level (RTL) implementation.

We compare our unit against alternative summation methods [13], which use exact multiplications and two-term, correctly-rounded additions in the accumulation format:

- *Recursive summation*: Each term is added to a floating-point accumulator in sequence.
- *Pairwise summation*: Pairs of terms are added together recursively until one result remains.

Models of these methods, as well as a calculation of the infinitely precise result, are written in Python using the mpmath [15] library for arbitrary precision arithmetic. The internal representation allows for unbounded exponents, such that partial products never require rounding before summation in either model, matching our unit's behaviour.

We generate $100,000$ random inputs for BF16, FP16, and the variants E5M2 and E4M3 of FP8. A depth of $k = 16$ is used for BF16 and FP16, and a depth of $k = 32$ is used for FP8. Accumulation is assumed to be in FP32 for BF16 and FP16 multiplications and in FP16 for FP8. Inputs are generated using the implied floating-point distribution, where bits in the packed representation are independent and identically distributed (i.i.d.) and set with probability 0.5. We exclude exceptional outputs $\pm\infty$ and not-a-number (NaN), and exact zeroes, for which the error is either infinite or undefined.

TABLE II
AVERAGE FORWARD ERROR IN ULP OF ALTERNATIVE SUMMATION METHODS FOR DOT PRODUCTS OF VARIOUS PRECISION AND DEPTH.

Multiply	Accum.	Depth	Recursive	Pairwise	**Ours**	Exact
FP16	FP32	16x	1.373	1.310	**0.259**	0.251
BF16	FP32	16x	0.186	0.182	**0.145**	0.145
E5M2	FP16	32x	1.160	1.058	**0.406**	0.246
E4M3	FP16	32x	2.690	1.744	**0.490**	0.250

Table II shows the average forward error in ulp for each of the output formats. We include results for an *exact* dot product implementation, which accumulates the partial products exactly with a single correct rounding on output, and therefore provides a lower bound for the error achievable by any implementation under the given input distribution. It can be seen that our implementation results in a superior average accuracy when compared with alternative implementations across all formats simulated, with less than 0.5 ulp on average. Further, the accuracy of our implementation is close to exact for FP16 and BF16.

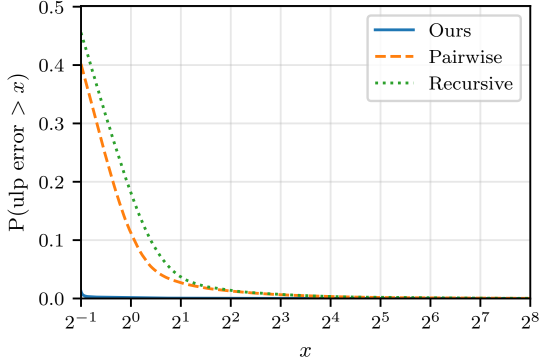
We quantify the likelihood of larger errors by measuring the complementary cumulative distribution function (CCDF) of the ulp error, defined as $\mathrm{P}(\text{ulp error} > x)$ for error threshold x. Figure 5 plots the CCDF of the ulp error of our implementation compared to recursive and pairwise summation methods, for the more precise FP16 and E4M3 on a semi-logarithmic scale.

An ulp error less than or equal to 0.5 corresponds to a correctly rounded result (ignoring ties), and the probability of a correctly rounded result can be read from the graphs as the complement of the CCDF at 0.5. Each order of magnitude corresponds to the loss of an additional bit of precision. The FP16 and E4M3 formats exhibit worse error characteristics under all three implementations than their counterparts BF16 and E5M2, due to their smaller exponent range and larger mantissa width. Errors beyond 0.5 ulp are very rare for the FP16 format, such that our implementation's curve is largely coincident with the x axis at the scale of Figure 5a. These results demonstrate that our implementation has a significantly lower probability of ulp errors of all magnitudes than either pairwise or recursive implementations, under the implied floating-point distribution.

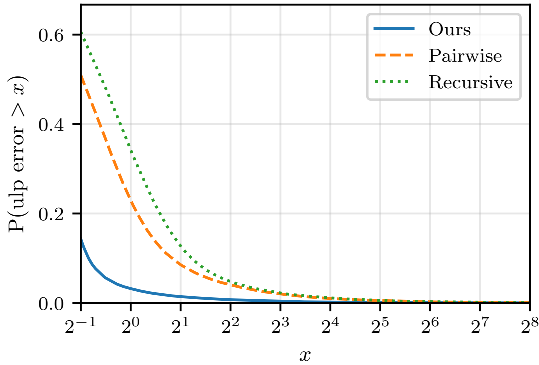
VI. Synthesis

The RTL design is fully pipelined over five clock stages. The selected technology is a 5 nm TSMC process and the target clock frequency is 1.1 GHz. We use industry-standard logic synthesis tools to produce a physical implementation of our design. Three versions of our design are considered:

1) A *scalar* design, processing either integer or floating-point data for one width;



(a)



(b)

Fig. 5. Estimated CCDF of ulp errors for different summation methods for (a) FP16 multiplication with FP32 accumulation and depth $k = 16$, and (b) E4M3 multiplication with FP16 accumulation and depth $2k = 32$.

2) A *merged* design, processing both integer and floating-point data for one width;
3) A *vectorized* design, processing both integer and floating point data for all widths.

The scalar version of our design is synthesized in three configurations that respectively compute 16-deep dot products of E8N10, 32-deep dot products of E5N3, and 32-deep dot products of INT8. Together they provide a baseline for comparison. The merged version of our design is synthesized in a configuration that computes 32-deep dot products of INT8 or E5N3. In this configuration, it shares logic between INT8 and E5N3 processing. The vectorized version of our design is synthesized in a configuration that computes 16-deep dot products of E8N10 and 32-deep dot products of INT8 and E5N3. In this configuration, it shares logic between all number formats using the approach presented in Section III. The arithmetic operators in each stage are internally vectorised, but the overall operation remains scalar.

We measure the combinational circuit area for each stage of our architecture as described in Section II-A. The area result and its breakdown per processing stage is reported in Table III, with other combinational and register area reported for the

TABLE III
CELL AREA TO THE NEAREST μm^2 OF DOT PRODUCT DESIGNS.

Configuration	Scalar	Scalar	Merged	Scalar	**Vectorized**
E5N3	32x	–	32x	–	**32x**
INT8	–	32x	32x	–	**32x**
E8N10	–	–	–	16x	**16x**
Multiplication	77	506	506	407	**673**
Exponent maximum	138	0	137	89	**153**
Mantissa alignment	147	0	147	196	**265**
Summation	174	162	216	175	**249**
Normalization	9	0	12	23	**25**
Other combinational	83	17	94	67	**132**
Registers	209	204	370	215	**493**
Total	836	889	1482	1172	**1990**

design as a whole. Comparing the area of the vectorized unit with the sum of the areas of each scalar unit, we quantify the total saving achieved by the use of a multiprecision, vectorized dot product unit to 31.3%. Comparing the sum of the areas of the merged unit and a single scalar E8N10 unit with that same total baseline area, we further observe that out of this 31.3% saving, 8.4% can be attributed to integer and floating-point merging and the remaining 22.9% to vectorization.

The sharing of registers is limited by low-power design considerations. Dynamic power will increase with vectorization, and bits unused in lower precisions should be gated off.

The stage benefiting the most from logic merging and vectorization is the summation stage. The same carry-save and carry-propagate adders perform the reduction for all required formats, with an optional shift in the 4:2 reduction step. Logic sharing in the multiplication stage also has a significant impact; logic sharing in the exponent maximum and mantissa alignment stages are relatively more modest, where native widths make vectorization less straightforward.

We also measure the area of auxiliary conversion and accumulation units. The conversion unit consumes one BF16 or FP16 value and produces an E8N10 value; or it consumes two FP8 values and produces two E5N3 values. The accumulation unit consumes one dot product result which is added to a single FP16, FP32 or INT32 accumulator value. Thirty-two instances of the conversion unit are sufficient to feed the dot product unit, and a single accumulation unit is sufficient to consume its results at full rate. The converters occupy 292 μm^2 and the accumulator 185 μm^2, bringing the total area per accumulating dot product to 2468 μm^2. In practice, reuse of the a and b operands inherent to matrix multiplication algorithms may allow for fewer conversion units to be used.

VII. DISCUSSION

The proposed multiprecision dot product unit scales by depth; this method of scaling simplifies the design of a matrix multiplication accelerator. The unit consumes 256 bits of matrix a and matrix b and produces 32 bits of matrix c, corresponding to 16 or 32 elements of one row of a and column of b and one element of c, a partial result for matrix

d. The accelerator can process matrices of arbitrary size based on this primitive, in a format-agnostic way.

Alternative methods of scaling for driving dot products units from vector registers are studied by Brunie [16].

Our implementation is compatible with the Open Compute MX specification [1] and associated micro-scaling exponents. The specification introduces a block floating-point operation [17], where INT8 vectors a and b have their own scale and their dot product accumulated in FP32. Dot products in formats such as INT8 and FP8 are computed with a depth of 32 and the combined scale factor applied to the result itself. The unit presented in this paper enables such use-cases.

Alternative implementations of fused dot products are numerous [8], [12], [18]–[20].

A unit for computing dot products of FP8 numbers was proposed by Lutz et al. [21]. It processes 4 terms with FP32 accumulation or 2 pairs of terms with FP16 accumulation, enabling vectorization of the accumulation stage. The small depth of operation keeps the latency small and ensures every operand fits into the same word size.

Another unit for computing dot products of BF16, FP16 and FP32 numbers was proposed by Desrentes et al. [22]. It processes up to 32 terms with FP32 accumulation precision. The authors use double-word decompositions to compute k terms from FP32 operands or $4k$ terms from BF16 or FP16 operands. The double-word decomposition is advantageous for wider operands, where the cost of multiplication dominates.

By contrast our unit targets narrow data formats, where the cost of multiplication becomes small. The precisions selected for the mantissa alignment and summation stages are key to reach the highest performance density. Vectorization can then be applied at each stage and maximize logic reuse for the processing of integer and floating-point, half- and full-width values. Attempts have been made at further merging multipliers and exponent adders but the multiplexer overhead is significant at small widths and therefore did not result in area saving. The merging of alignment shifts, sticky bit calculation and carry-save additions achieves a significant area saving.

Internal rounding errors are limited to the alignment stage and fully specified by the associated normalization and rounding method. This rules out hardware-centric optimisations such as hybrid local/global alignment strategies [23]. However this promotes a hardware-independent specification of the numerical behaviour of inexact fused dot products.

Cross-platform reproducibility of matrix multiplications in small number formats is the subject of on-going research [24] and standardization [25].

Acknowledgments

The authors would like to thank Zige Zheng, Blake Davies, Valeriy Novikov and Alan Vines for their support and comments on earlier versions of this research.

References

[1] "OCP microscaling formats (MX) specification," Open Compute Project, September 2023.

[2] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std. 754, 2019.

[3] "BFLOAT16 – hardware numerics definition," Intel, November 2018.

[4] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu *et al.*, "The ARM scalable vector extension," *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.

[5] *Vector Extension*, RISC-V International Std., November 2021.

[6] A. Danysh and D. Tan, "Architecture and implementation of a vector/SIMD multiply-accumulate unit," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 284–293, 2005.

[7] L. Huang, L. Shen, K. Dai, and Z. Wang, "A new architecture for multiple-precision floating-point multiply-add fused unit design," in *18th IEEE Symposium on Computer Arithmetic (ARITH'07)*. IEEE, 2007, pp. 69–76.

[8] 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 *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2020, pp. 133–136.

[9] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for Google's training chips: TPUv2 and TPUv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021.

[10] M. Langhammer, "High performance matrix multiply using fused datapath operators," in *2008 42nd Asilomar Conference on Signals, Systems and Computers*. IEEE, 2008, pp. 153–159.

[11] O. Desrentes, B. D. de Dinechin, and F. de Dinechin, "Exact fused dot product add operators," in *2023 IEEE 30th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2023, pp. 151–158.

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

[13] N. J. Higham, *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[14] J.-M. Muller, "On the definition of ulp (x)," INRIA, LIP, 2005.

[15] *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.3.0)*, http://mpmath.org/, 2023.

[16] N. Brunie, "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*. IEEE, 2020, pp. 1283–1290.

[17] J. H. Wilkinson, *Rounding errors in algebraic processes*. SIAM, 2023.

[18] F. De Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *2008 International Conference on Field-Programmable Technology*. IEEE, 2008, pp. 33–40.

[19] 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.

[20] J. Sohn and E. E. Swartzlander, "A fused floating-point four-term dot product unit," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 3, pp. 370–378, 2016.

[21] D. R. Lutz, A. Saini, M. Kroes, T. Elmer, and H. Valsaraju, "Fused FP8 4-way dot product with scaling and FP32 accumulation," in *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*. IEEE, 2024, pp. 40–47.

[22] O. Desrentes, B. D. De Dinechin, and F. De Dinechin, "Double-word decomposition in a combined FP16, BF16 and FP32 dot product add operator," in *2025 IEEE 32nd Symposium on Computer Arithmetic (ARITH)*, 2025.

[23] 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 *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 84–87.

[24] P. Xie, Y. Gao, Y. Wang, and J. Xue, "Revealing floating-point accumulation orders in software/hardware implementations," in *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, 2025, pp. 1425–1440.

[25] *Standard for Arithmetic Formats for Machine Learning*, IEEE Std. P3109, February 2023.