Novel, Configurable Approximate Floating-point Multipliers for Error-Resilient Applications

Vishesh Mishra[1], Sparsh Mittal[2], Rekha Singhal[3], Manoj Nambiar[3]
[1]*IIT Kanpur,* [2]*IIT Roorkee,* [3]*TCS Research, India*
vishesh@cse.iitk.ac.in,sparsh.mittal@ece.iitr.ac.in,{rekha.singhal,m.nambiar}@tcs.com

2023 24th International Symposium on Quality Electronic Design (ISQED) | 979-8-3503-3475-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISQED57927.2023.10129296

***Abstract*—By exploiting the gap between the user's accuracy requirement and the hardware's accuracy capability, approximate circuit design offers enormous gains in efficiency for a minor accuracy loss. In this paper, we propose two approximate floating point multipliers (AxFPMs), named DTCL (decomposition, truncation and chunk-level leading-one quantization) and TDIL (truncation, decomposition and ignoring LSBs). Both AxFPMs introduce approximation in mantissa multiplication. DTCL works by rounding and truncating LSBs and quantizing each chunk. TDIL works by truncating LSBs and ignoring the least important terms in the multiplication. Further, both techniques multiply more significant terms by simply exponent addition or shift-and-add operations. These AxFPMs are configurable and allow trading off accuracy with hardware overhead. Compared to exact floating-point multiplier (FPM), DTCL(4,8,8) reduces area, energy and delay by 11.0%, 69% and 61%, respectively, while incurring a mean relative error of only 2.37%. On a range of approximate applications from machine learning, deep learning and image processing domains, our AxFPMs greatly improve efficiency with only minor loss in accuracy. For example, for image sharpening and Gaussian smoothing, all DTCL and TDIL variants achieve a PSNR of more than 30dB. The source-code is available at https://github.com/CandleLabAI/ApproxFloatingPointMultiplier.**

I. Introduction

The ever-increasing need for data processing has drawn the attention of researchers toward alternative design paradigms for efficient computing. Several applications, especially in image processing, information extraction, and artificial intelligence [1] are error-resilient. In other words, these applications can tolerate errors in their computations. This allows trading accuracy for improving performance and area/energy efficiency. This has motivated the researchers to propose several approximate computing techniques. All computational units invoke the basic arithmetic circuits at the hardware level. Since multipliers (and adders) are at the core of any arithmetic circuit, the design of approximate multipliers can provide significant energy and latency benefits to a broad range of applications.

Contributions: This paper proposes two novel approximate FPM (AxFPM) designs for IEEE-754 floating-point multiplication. We demonstrate our technique for single-precision (FP32) format, although it can also be applied for double-precision format. In an FPM, processing of sign bit requires only an XOR operation; hence, making it approximate offers no benefit. For exponent addition, using an approximate adder can lead to high error. Some works use an external error compensating unit to reduce the error; however, these circuits nullify the area and power savings obtained from approximate addition. Mantissa multiplication accounts for 81% of the total power in an FPM [2]. Further, in the mantissa multiplication phase, the multiplication of 24-bit mantissa produces a 48-bit result. This result is finally truncated and rounded off to a 24-bit value. Since not all 48 bits get reflected in the final result, we can compute the most significant bits accurately and the least significant bits approximately. Clearly, there is a scope for approximation in the mantissa multiplication stage, and hence, we introduce approximation in the mantissa multiplication stage. The key idea of our designs is dividing the extended mantissa (i.e., mantissa with leading 1-bit) into various regions and then introducing significance-aware approximation in processing various regions.

Our first technique, named decomposition, truncation and chunk-level leading-one quantization (DTCL) based multiplication works in following stages (1) Segmenting the 24-bit extended mantissa into three regions: exact (Re), approximate (Ra) and truncation (Rt) (2) rounding and truncating Rt, the T-bit region of truncation (3) division of Ra into chunks of size K each. Then, in every K-bit chunk, the leading-one bit (LOB) is detected, and the remaining bits (if any) are reset to zero. (4) Now, multiplication of $(24-T)$ bit mantissas of two operand starts. It is performed such that (i) regions of exactness (Re) of two operands are multiplied using exact multiplier (ii) $Re_1 * Ra_2$ and $Re_2 * Ra_1$ multiplications are realized using shift-and-add operations and (iii) $Ra_1 * Ra_2$ multiplications are realized by summing the exponents. By adding the results of (i), (ii) and (iii), we obtain the final result. Refer Section III for more details.

Our second technique, named truncation, decomposition and ignoring LSBs (TDIL) works in following steps: (1) decomposing the 24-bit extended mantissa into two regions: C-bit region of computation (Rc) and T-bit region of truncation (Rt). Now, Rt is truncated. (2) Rc_i is expressed as $x_i + \delta x_i$, where $x_i = 2^{C-1}$. Then, the multiplication of Rc_1 and Rc_2 has four terms (i) we compute the term $x_1 * x_2$ simply by adding the exponents. (ii) We use shift operations for computing $x_1\delta x_2$ and $x_2\delta x_1$. (c) We discard the last term i.e. $\delta x_1 * \delta x_2$. We obtain the final result by adding the results of (i) and (ii). Refer Section IV for more details.

Sparsh is the corresponding author. Vishesh contributed to this paper while working as an intern at IIT Roorkee.

By varying the value of E, A, T, K and C, a designer can adapt our designs to meet the desired accuracy/hardware-cost targets. Previous techniques re-execute inputs causing a high error on an exact FPM, which leads to high overhead and variable latency. By contrast, we do not again multiply inputs causing higher errors.

Compared to exact FPM, our techniques reduce area, delay and energy at the cost of minor loss in accuracy. Compared to exact FPM, DTCL(4,8,8) reduces area, energy and delay by 11.0%, 68.8% and 61%, respectively, while incurring a mean relative error distance (MRED) of only 2.37%. TDIL(12) increases the area by 4% but reduces energy by 2.8% and delay by 39%, with an MRED of 4.08%. Overall, DTCL is superior to TDIL due to its lower error and hardware overhead. Further, we evaluate the impact of our AxFPM designs on three categories of applications: image processing, machine learning and deep learning. We observe that our designs provide negligible or acceptable loss in output quality, and thus, our proposal has the potential to benefit an extensive range of applications.

II. Background and Related Works

An IEEE-754 single precision FP number has three parts: a sign (1 bit), exponent (8 bits), and mantissa (23 bits). In the IEEE-754 system, the normal numbers have a hidden "1" bit to the left of the binary point. We denote 23b mantissa as M and the 24b extended mantissa as $m = 1.M$. To denote the first or second operand of multiplication, we use subscript 1 or 2, e.g., m_1.

Exact floating point multiplier: Figure 1 shows the steps in the multiplication of two FP numbers, A and B. The overall computation can be divided into three parallel steps. (1) An XOR operation is performed between the sign bits. (2) The exponent bits are added, and then the bias is subtracted. (3) The two mantissas viz., $1.M_1$ and $1.M_2$ are multiplied. Here, 24 partial products are generated to get a 48-bit intermediate result. (4) After truncation and rounding, the final mantissa value is attained. If required, the exponent may be adjusted, which is referred to as result normalization.

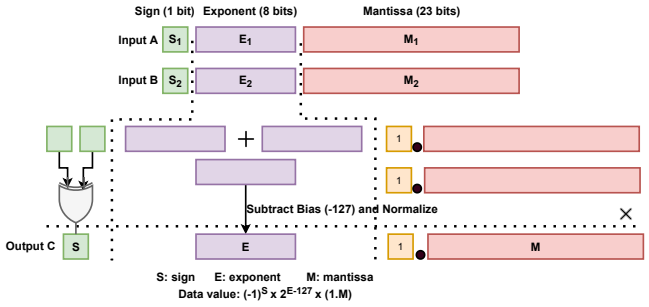

Fig. 1: Exact floating-point multiplication

Related works: We now review several recent approximate FPMs. Many of these designs allow switching between approximate and exact mode; however, we only summarize the approximate FPM design.

The CFPU [3] design discards one of the input mantissae and directly uses the second mantissa for the output. In this way, it obviates the need for multiplication. Inputs that can lead to high output error are multiplied using an accurate multiplier. However, due to directly discarding one mantissa, it incurs very high error. The RMAC [4] technique substitutes the multiplication of FP numbers with the addition of input mantissa values. This is because a multiplication operation involves repeated shift and add operations. The multiplication is re-executed on the exact hardware if the estimated error is high. Again, RMAC leads to high errors since addition is a coarse approximation of multiplication.

Zhang. et al. [5] propose an approximate FPM, which uses Mitchell's algorithm. Mitchell's algorithm for multiplication works in three steps: (1) Convert every operand to its $\log_2$ value by applying a piecewise linear approximation in [0,1]. (2) Add these values (3) Perform piecewise linear approximation to find the antilog value, which is the final result. They also note that applying Mitchell's algorithm directly to $(1 + M_1)*(1+M_2)$ multiplication leads to high error. Hence, in the multiplication $(1+M_1)*(1+M_2) = 1+M_1+M_2+M_1*M_2$, they apply Mitchell's algorithm for computing $M_1 * M_2$ and use adders for computing $1 + M_1 + M_2$. Thus, the multiplier is replaced by three adders. They further apply bit-truncation to improve hardware efficiency.

Li et al. [6] propose an approximate FPM based on approximate mantissa multiplication. They divide the mantissa into exact and approximate multiplication parts. For the approximate part, multiplication is replaced by an addition and shifting approach, which improves hardware efficiency. The FPCAM design [7] truncates certain bits in the mantissa M_2 and multiplies this truncated M_2 mantissa with M_1. This multiplication is further implemented as shift-and-add operations, which avoids the need for a multiplier.

Leon et al. [8] perform mantissa multiplication using the Booth algorithm and propose two ideas for approximation: perforation for reducing the depth of the partial product matrix (PPM) and rounding to reduce the width of PPM. Specifically, in perforation, P consecutive partial products beginning with the least-significant ones are eliminated. Then, rounding is performed, whereby, in the remaining partial products, R-1 LSBs are truncated, and the next untruncated bits are added to the remaining partial products for rounding.

Our proposed DTCL is distinct from all previous works such that, apart from truncation, it involves chunk-level quantization of the mantissa to powers of 2 such that the multiplication of parts of the product is achieved by shift operations and a narrow-width exact multiplier. Thus, it is expected to improve hardware efficiency without a significant impact on accuracy.

III. Proposed AxFPM Design 1: DTCL

In the *exact FP multiplication*, the sign-bits are XORed, exponent bits are added (and a bias is subtracted), and the extended mantissa (1.M) bits are multiplied. The 48b intermediate result of mantissa multiplication is truncated and rounded off to obtain the final mantissa. We propose novel

approximate FP multipliers, which work by approximating the multiplication of mantissa bits. Similar to previous works, we focus on the multiplication of normal floating-point numbers. Figure 2 shows the block-diagram of DTCL. DTCL operates in four stages.

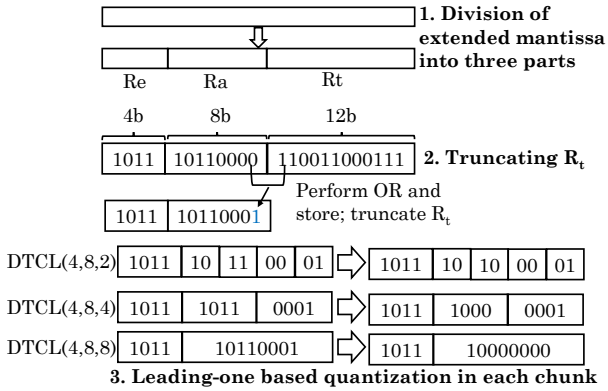

Fig. 2: Block diagram of DTCL technique

A. Stage 1: Division of extended mantissa

We first divide the extended mantissa into three regions so as to process them with different degrees of accuracy. We divide the extended mantissa into three regions of length E, A and T bits, such that $E + A + T = 24$. These are as follows:

- **Region 1: Region of Exactness:** To avoid a high amount of error, we process E most significant bits of extended mantissa in an exact manner. These bits, shown as Re_i are taken as $Re_i = m_i[1 : E]$, for $i = 1$ and 2.
- **Region 2: Region of Approximation** From the remaining $(24-E)$ bits, the most significant A bits are referred to as the region of approximation. $Ra_i = m_i[E + 1 : E + A]$.
- **Region 3: Region of Truncation** The LSBs have almost no impact on the final output since the 48-bit product is anyway truncated later to obtain a 24-bit product. As such, it is better to truncate them before multiplication, which improves hardware efficiency. The least significant T bits are referred to as region of truncation, thus, $Rt_i = m_i[24 - T + 1 : 24]$.

Our technique performs exact computation over the region of exactness (most significant bits), approximation of the region of approximation (middle bits) and rounding and truncation of the region of truncation (LSB bits).

B. Stage 2: Rounding and truncation

Compared to only performing truncation, we observe that performing rounding followed by truncation provides lower error since it keeps a signature of MSB of Rt in the LSB of the Ra. Hence, we implement these two steps:

1. Rounding: we perform an OR operation between the least significant bit of the Ra region and the most significant bit of Rt region. Therefore, the rounded LSB of Ra region is given by $Ra[E + A] = Ra[E + A]|Rt[24 - T + 1]$. In other words, we round the least significant bit of Ra with the help of the most significant bit of Rt.

2. Truncation: Now, we truncate all the bits in the truncation region. After truncation, we are left with only exact and approximate regions (Re and Ra). Changing the number of truncated bits allows for configuring the accuracy of FPM.

C. Stage 3: Chunk formation and quantization

We now divide the bits in the region of approximation (Ra) into multiple chunks and perform quantization. Note that this quantization is an approximation technique.

1. Chunk formation: We first divide the region of approximation into multiple chunks of size K-bits each. Specifically, $CHUNK[j]$ is created as $Ra[(j - 1) * K + 1, j * K]$.

2. Quantization: We perform quantization in a manner to reduce the hardware overhead of multiplication without degrading accuracy. We note that the multiplication by a power-of-two binary number can be implemented by simply performing a left-shift operation. This avoids the need for a multiplication operation. Also, if two power-of-two numbers are multiplied, the operation becomes equivalent to exponent addition. Based on these observations, we detect the leading-one bit (LOB) in a chunk and set the remaining bits of the chunk to zero.

Let $A = 8$. Then, for a sample number $[11010010]$, Table I shows chunking and quantization process for various values of K.

TABLE I: Chunking and LOB quantization for the number [11010010]

Chunk size (K)	Chunks	After quantization
2	[11,01,00,10]	[10,01,00,10]
4	[1101,0010]	[1000,0000]
8	[11010010]	[10000000]

D. Stage 4: Multiplication

After truncation, the $(24 - T)$ bit number can be written as $m_i = (Re_i * 2^A + Ra_i) * 2^T$. On multiplying m_1 with m_2, we get

$[Re_1 * Re_2 * 2^{2A} + [Re_1 * Ra_2 + Re_2 * Ra_1] * 2^A + Ra_1 * Ra_2] * 2^{2T}$

In the above equation, for computing the first term in hardware, we use an E-bit accurate multiplier and then left-shift the result by $2A+2T$ bits. The shift and add technique is used to compute the second and third terms. This is because, after quantization, each chunk is a power-of-two number. Finally, since the last term consists of only powers-of-two, it is computed by exponent addition, and then the bits are arranged to get the result. Finally, the four terms are combined to generate the final result.

Summary: The DTCL variants can be denoted as DTCL(E, A, K), where E/A are the number of bits in the region of exactness/approximation (respectively) and K is the chunk-size. Clearly, the number of chunks is A/K. A designer can tune the values of E, A, K to exercise a tradeoff between accuracy and efficiency. As an example, we take $E = 4$,

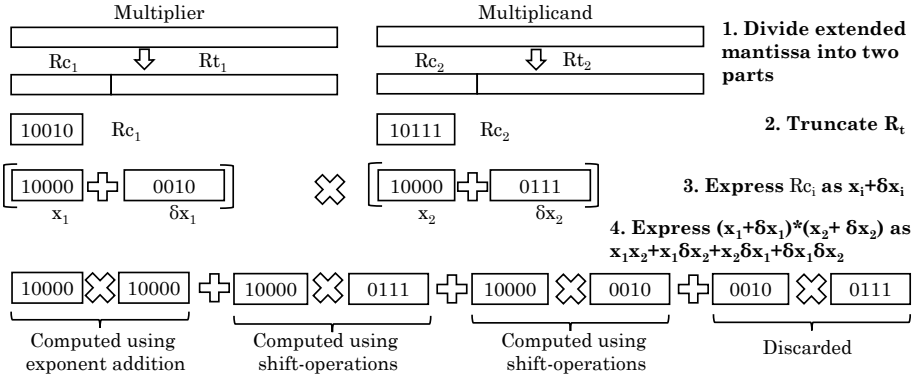

Fig. 3: Block diagram of TDIL technique

$A = 8, T = 12$. For $A = 8$, the possible chunk size values (K) can be 2, 4 or 8. This corresponds to three versions denoted by DTCL(4,8,2), DTCL(4,8,4) and DTCL(4,8,8).

IV. Proposed AxFPM Design 2: TDIL

We now discuss our second proposed technique, named TDIL (truncation, decomposition and ignoring LSBs). Here, we divide the extended mantissa into only two regions: a region of computation (Rc) and a region of truncation (Rt). If the region of computation has C bits, then this version of TDIL is denoted as TDIL(C).

We first truncate the region of truncation. Then, we express the extended mantissa in the following way:

$m_1 = x_1 + \delta x_1$ and $m_2 = x_2 + \delta x_2$

Here, the value of x_1 and x_2 are fixed such that $x_1 = x_2 = 2^{C-1}$. Also, $\delta x_i = m_i[2 : C]$. Table II illustrates this with an example.

TABLE II: Decomposition in TDIL technique for two operands ($C = 5$)

Step	First operand	Second operand
1	m_1 = [1 0 0 1 0]	m_2 = [1 0 1 1 1]
2	x_1= 2^4 =[1 0 0 0 0]	x_2= 2^4= [1 0 0 0 0]
3	δx_1 = [0 0 1 0]	δx_2 =[0 1 1 1]

Now, the product $m_1 \times m_2$ can be written as

$x_1 * x_2 + x_2 * \delta x_1 + x_1 * \delta x_2 + \delta x_1 * \delta x_2$

Here, we compute $x_1 * x_2$ simply by using the addition of exponents. Further, we use shift operations for computing $x_1\delta x_2$ and $x_2\delta x_1$. Finally, we discard the last term i.e. $\delta x_1 * \delta x_2$. By adding $x_1 * x_2$, $x_2 * \delta x_1$ and $x_1 * \delta x_2$, we obtain the final result. Thus, TDIL avoids the need for any multiplier for performing mantissa multiplication. Figure 3 shows the flow-diagram of TDIL technique.

V. Experimental setup

Evaluation approach: We evaluate at both the software and hardware levels. At the software level, we compute the error metrics using MATLAB. We generate 10^6 uniformly distributed random inputs and feed them to AxFPMs to evaluate the "mean relative error distance" (MRED) and "maximum relative error distance" (MaxRED). For hardware-level evaluation, we implemented the proposed design in Verilog and synthesized it using SynopsysDesign Compiler with NanGate 45nm Open Cell Library at an operating condition of P:1.0, V:1.10V, T:25.0°C. Also, a uniform clock period of 1ns is used. We now define the metrics.

(1) $MRED$: Let $ED = r - \overline{r}$, where r is the accurate result, $\overline{r}$ is the approximate multiplication result. Also, num corresponds to the number of random inputs. $MRED$ measures the mean of relative error in each computation of random inputs. $MRED = \frac{1}{num} \sum_{i=1}^{num} \frac{ED_i}{r} \times 100$

(2) $MaxRED = max\left(\frac{ED_i}{r}\right) \times 100$

(3) Area: We show the post-layout area based on physical synthesis in DC, placement and routing on ICC2. (4) Power: The synthesized netlist is annotated with RTL VCD for switching activity and analyzed using Synopsys PrimeTime/PrimePower. (5) Delay: The critical path delay of the entire circuit workflow is chosen as the delay of the design. To simulate our and the previous techniques, we first set the clock period to $1ns$. After that, we measure the delay incurred by these techniques.

VI. Evaluation

A. Error metrics

We evaluate multiple versions of our techniques (DTCL, and TDIL), along with several previous works and show the results in Table III.

Comparison of DTCL/TDIL variants: Among the variants of DTCL(E, A, K), the error reduces with decreasing value of chunk size (K). For lower values of K, quantization has a lower impact on the value of Ra, whereas a high value of K leads to coarse approximation. At a low value of K, the area, power and delay values are higher, but error values are also higher. This shows the tradeoff between hardware cost and error metrics. Further, we observed (results omitted due to page limit) that the error reduces with increasing values of E and A. This is expected because for higher values of E, more bits go through exact multiplication and fewer bits are part of the region of approximation or truncation. A designer can select the proper value of E, A and K to exercise this tradeoff and meet the application-level quality constraints.

TABLE III: Hardware metrics of FPMs (PDP/ADP/EDP = power/area/energy delay product)

Design	MRED	MaxRED	Area(um^2)	Power (mW)	Delay (ns)	Energy (PDP)	EDP	ADP
Exact	0	0	7690	0.15	3.59	0.538	1.933	4141.1
DTCL(4,8,2)	1.72	4.86	7990	0.25	2.2	0.55	1.21	4394.5
DTCL(4,8,4)	2.05	6.3	7884	0.21	1.84	0.386	0.711	3046.3
DTCL(4,8,8)	2.37	6.48	6842	0.12	1.4	0.168	0.235	1149.4
TDIL(5)	6	19	7824	0.2	1.8	0.36	0.648	2816.6
TDIL(6)	5.88	18.35	7992	0.21	2	0.42	0.84	3356.6
TDIL(12)	4.08	14.01	8012	0.24	2.18	0.523	1.141	4191.9
CFPU	2.04	6.42	7951	0.17	1.6	0.272	0.435	2162.7
FPCAM	3.16	9.28	7698	0.19	1.76	0.334	0.588	2574.2

In TDIL(C), with an increasing value of C, the error values are reduced. This is expected since for higher values of C, fewer bits are truncated and $(x_1 * \delta x_2 + x_2 * \delta x_1)$ term becomes more accurate. We now compare DTCL with TDIL for the same number of total bits, i.e., $E + A$ (for DTCL) equals C (for TDIL). For example, DTCL(4,8,2) has E=4, A=8; hence, it can be compared with TDIL(12), which has C=12. TDIL(12) has an MRED of 4.08%, and DTCL(4,8,8) has an MRED of 2.37%. Thus, DTCL gives a lower error than TDIL. TDIL ignores the $\delta x_1.\delta x_2$ term, which leads to a large error. TDIL requires higher area due to the extra logic required to generate δx_1 and δx_2.

Comparison with previous works: Since the exact FPM is designed to reduce hardware resource utilization, it uses lower area and power than all AxFPM designs except DTCL(4,8,8). However, AxFPMs seek to reduce the delay. Except for DTCL(4,8,2), our other AxFPMs consume lower energy (PDP) than the exact FPM. In fact, DTCL(4,8,8) provides the lowest area, delay, PDP, EDP and ADP among all our designs and previous designs. DTCL(4,8,8) is superior to FPCAM on all metrics. Also, it has comparable error metrics to CFPU and is superior to CFPU on all hardware metrics.

VII. Application-level evaluation

We now provide an application-level evaluation of our AxFPMs for three categories of error-tolerant applications: image processing, machine learning and deep learning. For evaluating image quality, we use Peak Signal to Noise Ratio (PSNR) in dB and Structural Similarity Index (SSIM) [9] values of images obtained from AxFPMs compared with those obtained from accurate FPMs. PSNR values higher than 30 dB are considered good [9].

1. Gaussian Smoothing: We take a 5×5 Gaussian kernel (G), shown as

$$G = \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \quad (1)$$

We evaluate three *256×256* grayscale input images, viz. Lenna, monkey and female. In convolution, exact multiplications have been replaced by approximate multiplications. Table IV shows the results. Evidently, all AxFPMs achieve more than 30dB PSNR, and DTCL provides better results than TDIL. Figure 4(top) provides the visual comparison of "female" output images obtained by accurate FPM and our proposed AxFPMs.

TABLE IV: Gaussian smoothing results

Image	Lenna		Baboon		Female	
Design	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
DTCL(4,8,2)	42.27	0.9981	42.21	0.998	42.1	0.998
DTCL(4,8,4)	40.68	0.9975	40.16	0.9974	39.36	0.9971
DTCL(4,8,8)	38.46	0.9972	38.31	0.9975	37.08	0.9946
TDIL(5)	33	0.9876	33.09	0.9932	31.32	0.9811
TDIL(6)	33.01	0.9876	33.09	0.9932	31.32	0.9811
TDIL(12)	36.31	0.9944	35.88	0.9935	35.82	0.9935

2. Image Sharpening: Table V shows the numerical results and Figure 4(bottom) provides visual results. For all designs, PSNR is higher than 30dB, and for all DTCL designs, the SSIM remains close to 1. TDIL versions have lower quality than DTCL versions.

TABLE V: Image sharpening results

Design	PSNR	SSIM	Design	PSNR	SSIM
DTCL(4,8,2)	42.47	0.9986	TDIL(5)	33	0.9876
DTCL(4,8,4)	40.85	0.9981	TDIL(6)	33.01	0.9876
DTCL(4,8,8)	39.02	0.9975	TDIL(12)	36.42	0.9972

3. K-Means clustering: We use datasets containing 150 to 4500 points uniformly distributed into 3 clusters. At each iteration of the algorithm, the cluster centers are updated in order to find the best possible cluster. We approximate the square operation required for Euclidean distance computation. Figure 5 shows the outcome of the K-means algorithm when evaluated using an accurate FPM (left) and DTCL(4,8,2) (right). Evidently, the output of DTCL is identical to that of the accurate FPM.

4. kNN: We evaluate three datasets, namely, Monk's problem, Haberman's survival, and letter recognition [10]. We use the Euclidean distance metric for similarity calculations and implement it approximately. We have experimented with various configurations of DTCL. For Haberman's survival dataset, the accuracy degrades by 0.74 percentage points, whereas for the other two datasets, the accuracy was the same as that of the accurate FPM. Similarly, for all three datasets, the F1 score with DTCL AxFPMs was the same as that with accurate FPM designs.

5. Deep learning application: We take a CNN that has one convolution layer with 12 filters, followed by another convolution layer with 24 filters. The CNN is trained and evaluated

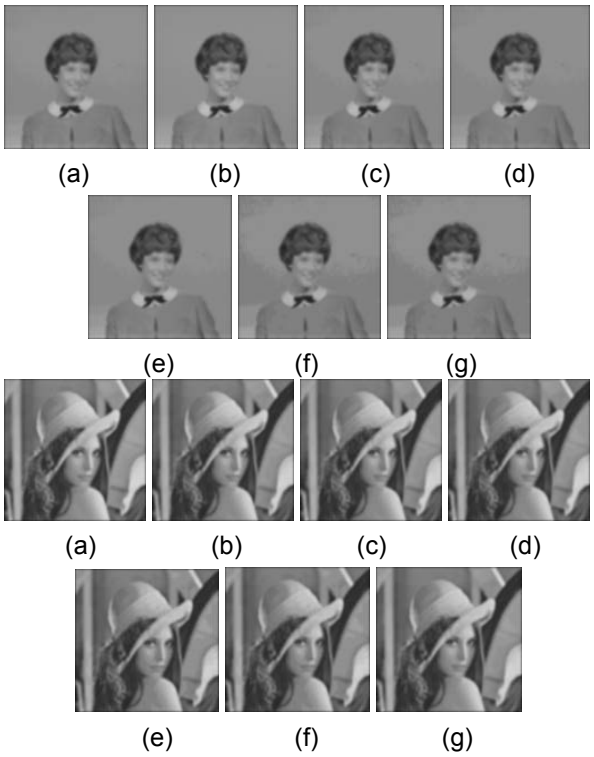

Fig. 4: (Top) Gaussian smoothing and (bottom) image sharpening results using (a) accurate FPM, (b) DTCL(4,8,2), (c) DTCL(4,8,4), (d) DTCL(4,8,8), (e) TDIL(5), (f) TDIL(6) and (g) TDIL(12). In Gaussian smoothing results, approximation leads to a halo behind the person in images (e) to (g). In other images, the impact of approximation is difficult to see visually.

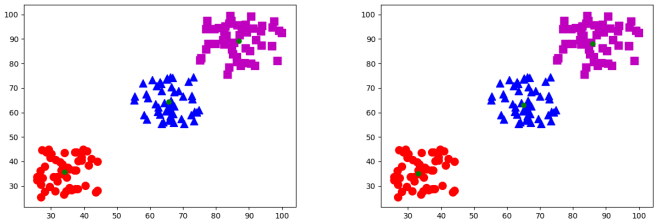

Fig. 5: K-Means clustering results using accurate FPM (left) and DTCL(4,8,2) (right).

on the MNIST dataset. We employ our AxFPMs to perform multiplications in its convolution layers. The accuracy results are shown in Table VI. Here "loss" refers to the accuracy difference between exact and approximate implementations. Clearly, DTCL variants incur minor accuracy loss, whereas TDIL variants incur a slightly higher loss.

VIII. Ablation Results: A Variant Based on MSB-based Quantization (DTCM)

We further present a variant, named DTCM, named decomposition, truncation and chunk-level most-significant bit based quantization. It differs from DTCL only in the way a chunk is quantized. In DTCL, the LOB of a chunk is detected, and the remaining bits are set to zero. In DTCM, quantization is done by setting all bits of a chunk except the MSB to zero. For a K-bit chunk, the quantized value of chunk $Quan[j]$ is set to 2^{K-1} if $CHUNK[j] \geq 2^{K-1}$, otherwise, $Quan[j]$ is set to zero. This can be implemented just by examining the MSB of the chunk. Thus, DTCM quantizes a chunk to either zero or 2^{K-1}.

Figure 6 contrasts the quantization results from DTCL and DTCM for K=2 and 4. It is evident that DTCM quantizes many chunk-value combinations to zero, and hence, it incurs a high amount of error, as evident from the results. By contrast, DTCL introduces a lower amount of error in each chunk value and hence, is preferred.

TABLE VI: CNN accuracy results (pp=percentage point)

Design	Accuracy	Loss (pp)	Design	Accuracy	Loss (pp)
EXACT	98.61%	0.00%			
DTCL(4,8,2)	98.58%	0.03	TDIL(5)	96.48%	2.13
DTCL(4,8,4)	98.24%	0.37	TDIL(6)	96.81%	1.80
DTCL(4,8,8)	97.10%	1.51	TDIL(12)	97.11%	1.50
CFPU	97.11%	1.50	FPCAM	97.09%	1.52

	Original chunk values	DTCL (Leading-one based quantization)	DTCM (MSB-based quantization)
K=2	[00,01,10,11]	[00,01,10,10]	[00,00,10,10]
K=4	[0000,0001,0010,0011]	[0000,0001,0010,0010]	[0000,0000,0000,0000]
	[0100,0101,0110,0111]	[0100,0100,0100,0100]	[0000,0000,0000,0000]
	[1000,1001,1010,1011]	[1000,1000,1000,1000]	[1000,1000,1000,1000]
	[1100,1101,1110,1111]	[1000,1000,1000,1000]	[1000,1000,1000,1000]

Fig. 6: Comparison of quantization in DTCL and DTCM (red-entries show where quantization of DTCL and DTCM are different). For K=2 and 4, there are 4 and 16 (respectively) possible chunk values.

Table VII presents the results of DTCM. The only difference between DTCL and DTCM is that DTCL uses LOB-based quantization, and DTCM uses MSB-based quantization. Due to its more aggressive approximation approach, DTCM incurs higher errors than DTCL, whereas their hardware overheads are similar. Hence, DTCL is superior to DTCM.

IX. Conclusion

Basic arithmetic building blocks like adders and multipliers [11] are the ideal candidates for approximation to gain efficiency at the cost of accuracy. In this paper, we present two AxFPM designs. Experiments on several error-tolerant applications have confirmed that our proposed AxFPMs offer a significant boost in efficiency with only a minor loss in accuracy. Given the low area, power and delay obtained, our AxFPMs make an excellent alternative to be used in low-energy embedded devices running machine learning, deep learning and image processing workloads. As future work, we intend to synthesize larger chips with our AxFPMs to substantiate and measure the overall savings for a given use case.

TABLE VII: Hardware metrics of FPMs (PDP/ADP/EDP = power/area/energy delay product)

Design	MRED	MaxRED	Area(um^2)	Power (mW)	Delay (ns)	Energy (PDP)	EDP	ADP
Exact	0	0	7690	0.15	3.59	0.538	1.933	4141.1
DTCM(4,8,2)	2.13	7.47	7982	0.25	2.18	0.545	1.188	4350.2
DTCM(4,8,4)	3.02	10.28	7846	0.21	1.82	0.382	0.696	2998.7
DTCM(4,8,8)	3.18	10.76	6840	0.12	1.4	0.168	0.235	1149.1

REFERENCES

[1] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 62:1–62:33, 2016.

[2] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *TVLSI*, vol. 8, no. 3, pp. 273–286, 2000.

[3] M. Imani, D. Peroni, and T. Rosing, "CFPU: Configurable floating point multiplier for energy-efficient computing," 2017.

[4] M. Imani *et al.*, "RMAC: Runtime Configurable Floating Point Multiplier for Approximate Computing," *ISLPED*, 2018.

[5] H. Zhang *et al.*, "A low-power accuracy-configurable floating point multiplier," in *ICCD*, 2014, pp. 48–54.

[6] J. Li *et al.*, "Accuracy-configurable low-power approximate floating-point multiplier based on mantissa bit segmentation," in *TENCON*, 2020.

[7] C. K. Jha *et al.*, "FPCAM: Floating point configurable approximate multiplier for error resilient applications," in *ISCAS*, 2021.

[8] V. Leon *et al.*, "Improving power of dsp and cnn hardware accelerators using approximate floating-point multipliers," *ACM TECS*, 2021.

[9] D. Bull, *Communicating Pictures - A Course in Image and Video Coding*. Academic Press, 2014.

[10] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[11] V. Mishra, S. Mittal, S. Singh, D. Pandey, and R. Singhal, "MEGA-MAC: A Merged Accumulation based Approximate MAC Unit for Error Resilient Applications," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2022.