Journal of Systems Architecture 173 (2026) 103686

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

FP8ApproxLib: An FPGA-based approximate multiplier library for 8-bit floating point☆

Ruiqi Chen [a],*, Yangxintong Lyu [a], Han Bao [a], Shidi Tang [b], Jindong Li [c], Yanxiang Zhu [d], Ming Ling [b], Bruno da Silva [a]

[a] Department of Electronics and Informatics (ETRO), Vrije Universiteit Brussel, Brussels, 1050, Belgium
[b] School of Integrated Circuits, Southeast University, Nanjing, 214026, China
[c] School of Artificial Intelligence, University of Chinese Academy of Sciences, Beijing, 100049, China
[d] VeriMake Innovation Laboratory, Nanjing, 210088, China

ARTICLE INFO

Keywords:
8-bit floating point
Field-programmable gate array
Approximate multiplier
Approximate computing

ABSTRACT

The 8-bit floating-point (FP8) data format has been increasingly adopted in neural network (NN) computations due to its superior dynamic range compared to traditional 8-bit integer (INT8). Nevertheless, the heavy reliance on multiplication in neural network workloads leads to considerable energy consumption, even with FP8, particularly in the context of FPGA-based deployments. To this end, this paper presents FP8ApproxLib, an FPGA-based approximate multiplier library for FP8. Firstly, we conduct a bit-level analysis of the prior approximation method and introduce improvements to reduce the resulting computational error. Based on these, we implement a fine-grained optimized design on mainstream FPGAs (Altera and AMD) using primitives and templates combined with physical layout constraints. Moreover, an automated tool is developed to support user configuration and generate HDL code. We then evaluate the accuracy and hardware efficiency of the FP8 approximate multipliers. The results show that our proposed method achieves an average error reduction of 53.15% (36.74%~72.82%) compared to previous FP8 approximation method. Moreover, compared to prior 8-bit approximate multipliers, our FP8 designs exhibit the lowest resource utilization. Finally, we integrate the design into the inference phase of three representative NN models (CNN, LLM, and Diffusion), demonstrating its excellent power efficiency. This is the first FP8 approximate multiplier design with architecture-aware fine-grained optimization and deployment for modern FPGA platforms, which can serve as a benchmark for future designs and comparisons of FPGA-based low-precision floating-point approximate multipliers. The code of this work is available in our GitLab*.

1. Introduction

In recent years, the rapid development of neural networks has brought the issue of high energy consumption to the forefront [1]. An effective approach to address this challenge is the use of quantization techniques, which improves memory access and computational efficiency [2,3]. Among these, 8-bit floating-point (FP8) numbers offer better dynamic range and computational precision compared with traditional 8-bit integer (INT8), making them widely adopted in neural network computations [4,5]. To enhance the computational efficiency of FP8, specialized hardware modules for FP8 acceleration have become a new trend. For instance, designing application-specific integrated circuits (ASICs) [6,7] and integrating corresponding units into GPUs [8].

Nonetheless, this still cannot eliminate the multiplication operations in DNNs. As a fundamental computation in DNNs, various approximate multipliers have been proposed to improve efficiency and reduce energy consumption. These multipliers are designed to reduce latency, energy consumption, and area. Chen et al. [9] proposed an optimally multi-level architecture that seamlessly integrates runtime configurability with parallel module execution. An optimization strategy was applied to improve area efficiency, achieving a linear relationship with accuracy rather than the quadratic or exponential relationships seen in previous works. Ansari et al. [10] developed an 8 × 8 approximate

☆ gitlab.com/etrovub/embedded-systems/publications/fp8approxlib.

* Corresponding author.
E-mail addresses: ruiqi.chen@vub.be (R. Chen), yangxintong.lyu@vub.be (Y. Lyu), han.bao@vub.be (H. Bao), shidi@seu.edu.cn (S. Tang), lijindong2022@ia.ac.cn (J. Li), zhuyanxiang@verimake.com (Y. Zhu), trio@seu.edu.cn (M. Ling), bruno.da.silva@vub.be (B. da Silva).

https://doi.org/10.1016/j.sysarc.2026.103686
Received 14 July 2025; Received in revised form 14 November 2025; Accepted 1 January 2026
Available online 8 January 2026

multiplier tailored for NN designs by improving the design of logarithmic multipliers. HEAM [11] achieves automated design of approximate multipliers by minimizing the average error based on operand distribution and integrates these multipliers into DNN accelerators.

However, the aforementioned approximation methods are primarily aimed at reducing power consumption and area utilization in ASIC implementations and may perform suboptimally on field-programmable gate arrays (FPGAs). This is because FPGA reconfigurable logic is typically based on fixed-size lookup tables (LUTs). While FPGAs also integrate DSP hardware multiplier units, these units are limited or unavailable on some edge FPGAs (e.g., Lattice iCE40 and Gowin-GW1N). Consequently, there is a growing interest in DSP-free implementations across various edge applications, including computer vision [12], anomaly detection [13], and cryptographic processing [14]. Therefore, improving the efficiency of LUT-based multiplication in terms of speed, power consumption, and resource utilization becomes particularly critical. Ullah et al. [15–17] proposed a series of FPGA-based approximate multipliers covering data bit-widths from 4-bit to 32-bit. More recently, their AxO series [18,19] integrated the design of approximate multipliers into spiking neural network (SNN) accelerators. DyRecMul [20] introduced a dynamically reconfigurable INT8 approximate multiplier design, which includes a floating-point conversion unit. This design enables efficient floating-point conversion, reducing preprocessing operations and enhancing computational efficiency. Leon et al. [21] proposed a DSP-based approximate multiplier design for floating-point computations, which was integrated into a convolutional neural network (CNN) accelerator. This approach achieved more efficient computation within the accelerator framework.

In conclusion, existing works are limited by (1) model-specific FP8 approximate multipliers with poor generalization and accuracy trade-offs, and (2) most prior implementations target ASICs, with no known FP8 approximation designs developed specifically for FPGAs.

To this end, this paper presents FP8ApproxLib, an FPGA-based approximate multiplier library for FP8. Firstly, FP8ApproxLib performs a bit-level analysis of existing FP8 approximate multiplication methods and proposes a new hardware-friendly approach with reduced precision loss. Then, based on this, FP8ApproxLib provides a set of approximate multiplier templates targeting mainstream FPGAs, including Altera and AMD. All templates are supported by an automatic code generation tool, which outputs the corresponding hardware description language (HDL) code based on user configurations. Finally, the energy efficiency of FP8ApproxLib is validated during the inference stages of several representative neural network models. To the best of our knowledge, this is the first FP8 approximate multiplier design with architecture-aware fine-grained optimization and deployment for modern FPGA platforms, which can serve as a benchmark for future designs and comparisons of FPGA-based low-precision floating-point approximate multipliers. The core project of FP8ApproxLib is clear and open source, made available to users and developers for further customization and development. This work is built on our previous work [22] and main contributions are summarized as follows:

- A novel approximation method for FP8 multiplication is proposed, along with a bit-level analytical study of its behavior. Compared to prior approaches, the proposed method reduces the error by 36.74% (E4M3) .
- Based on this analysis, the design is implemented on three mainstream FPGA platforms (Altera and AMD). Fine-grained hardware-level optimization is achieved by combining primitive-based design, parameterized templates, and physical layout constraints, resulting in reduced resource usage and power consumption.
- A comparison with prior FPGA-based 8-bit approximate multiplier designs demonstrates that FP8ApproxLib reduces LUT resource usage by an average of 49.9% , without sacrificing performance.
- Case studies on CNN, large language model (LLM) and Diffusion are conducted, showing the practicality of FP8ApproxLib application in both the algorithm level and implementation level.

Table 1
Comparison of INT8 and FP8 (E4M3)

Data Type	INT8	FP8 (E4M3)
Bit Width	8 bits	8 bits
Minimum Value	−128	−448
Maximum Value	127	448
Decimal Precision	Fixed (1)	Dynamic

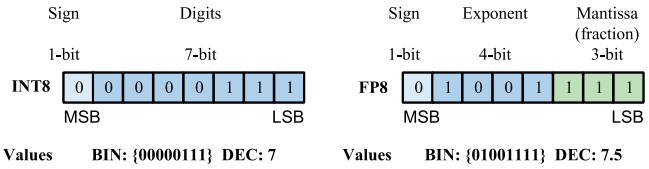

Fig. 1. The demonstration of the INT8 and FP8 (E4M3) adheres to IEEE 754. MSB stands for most significant bit and LSB stands for least significant bit.

The rest of this paper is organized as follows. Section 2 introduces the preliminaries of this work. Section 3 explains the modified approximation method of proposed FP8ApproxLib. Section 4 describes the customized hardware implementation in FP8ApproxLib. Section 5 illustrates the comprehensive experimental results of FP8ApproxLib. Section 6 provides a series case studies to verify the practicality, and Section 7 gives the conclusion.

2. Preliminaries

2.1. FP8 formats

FP8 is a natural progression from the FP16 representations, effectively reducing memory consumption and improving memory access and computational efficiency [5]. Compared to traditional INT8, FP8 offers a larger dynamic range (the commonly used E4M3 format, as shown in Table 1 and Fig. 1). Moreover, FP8 achieves less accuracy loss during NN inference [23]. The FP8 format adheres to IEEE-754 conventions, where a real numbers is encoded by using a 1-bit sign S, an e-bit integer exponent E and an m-bit fractional (mantissa M),

$$x_{\mathrm{DEC}} = (-1)^S \times 2^E \times M, \tag{1}$$

where $E = e - bias$ and $M = 1 + m$. The $bias$ in this context varies with the number of bits in the exponent and is determined by the following formula:

$$bias = 2^{e-1} - 1. \tag{2}$$

Note that an implicit 1, namely the $hiddenbit$, is concatenated to the fraction as an integer bit and forms the significant. A FP number with $E = 0$ has no implicit 1 in the significant, so zero and subnormal values can be represented. In addition, exponent $E = 2^{e-1}-1$ is reserved for the representation of $\pm\infty$ and $NaNs$.

2.2. FPGA structure

State-of-the-art FPGAs from AMD (Xilinx) and Altera (Intel) utilize basic logic cells such as multi-input LUTs, carry chains (adders), multiplexers, and D flip-flops to implement both combinational and sequential logic circuits. The method proposed in this paper is general, relying on LUTs, adders (carry chains), multiplexers, and D flip-flops. However, there are subtle differences depending on the characteristics of different devices. Therefore, we present three types of mainstream FPGA devices for illustration, as shown in Fig. 2.

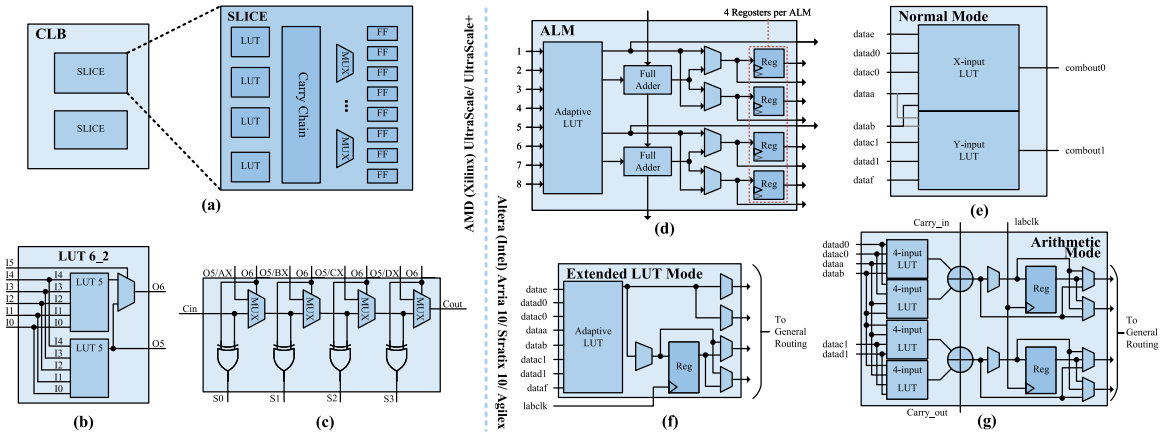

Fig. 2. The state-of-the-art FPGA basic structure from AMD (Xilinx) and Altera (Intel). (a) Typical AMD FPGA configurable logic block (CLB) structure [24]. (b) LUT6 structure. (c) Carry chain structure. (d) Typical Altera FPGA adaptive logic module (ALM) structure [25]. (e) The ALM runs in normal mode. (f) The ALM runs in extended LUT mode. (g) The ALM runs in arithmetic mode.

2.2.1. AMD

The Configurable Logic Block (CLB) is the basic logic unit in AMD's UltraScale and UltraScale+ FPGAs, and each CLB contains two slices. As shown in Fig. Fig. 2(a), each slice consists of four 6-input LUTs (LUT6_2), two 4-bit carry chains, eight flip-flops (FFs), and several multiplexers (MUXs). Each LUT6_2 can be configured to implement either a single 6-input combinational function using the O6 output, or two independent 5-input functions using the O5 and O6 outputs, as shown in Fig. 2(b). The desired logic functionality can be implemented by specifying the corresponding INIT value [17]. Besides the implementation of single 6-bit combinational functions, these LUT6_2 are also used for controlling the associated carry chain, as shown in Fig. 2(c). The carry chain implements a carry-lookahead adder by using O5 as the carry-generate signal and O6 as the carry-propagate signal. The carry-generate signals for the carry chain can also be provided by the external bypass signals AX – DX.

2.2.2. Altera

The adaptive logic module (ALM) in Altera's Stratix 10, Arria 10, and Agilex series primarily includes an 8-input fracturable LUT, two dedicated embedded adders, and four dedicated registers, as shown in Fig. 2(d). Unlike AMD's CLB, Altera's ALM supports three different modes: normal mode, extended LUT mode, and arithmetic mode. In the normal mode, the adaptive LUT is divided into two multi-input LUTs: an X-input LUT and a Y-input LUT, as shown in Fig. 2(e). The two LUTs are connected through logical interconnections, allowing for different X:Y combinations. When the inputs are independent, the LUTs can support configurations such as 4:4 or 5:3, or even less inputs. However, when X:Y is 5:4 or 5:5, the inputs need to be shared between the two LUTs. In the extended LUT mode, the ALM can support a maximum of an 8-input LUT, as shown in Fig. 2(f). The ALM in arithmetic mode uses two sets of two 4-input LUTs along with two dedicated full adders, as shown in Fig. 2(g). The dedicated adders allow the LUTs to perform pre-adder logic. Therefore, each adder can add the output of two 4-input functions.

3. Approximate multiplication for FP8

This section introduces our improved FP8 approximate multiplication method. We conduct a bit-level analysis to show that the proposed method not only reduces precision loss relative to existing approach, but also offers theoretical improvements in hardware resource efficiency.

3.1. L-Mul approximation

According to the Eq. (1), the FP8 multiplication process of x and y can be represented as:

$$\begin{aligned} Mul(x,y) &= M_x \cdot 2^{E_x} \times M_y \cdot 2^{E_y} \\ &= (1+m_x)\cdot 2^{E_x} \times (1+m_y)\cdot 2^{E_y} \\ &= (1+m_x+m_y+m_x\cdot m_y)\cdot 2^{E_x+E_y}, \end{aligned} \tag{3}$$

we omit the sign bit as it can be handled through an XOR operation. One can note that in Eq. (3), only $m_x \cdot m_y$ involves a multiplication operation for hardware circuit design. The remaining operations can be implemented by using addition or other linear operations such as shift. To alleviate the potential bottleneck caused by mantissa multiplication, Luo et al. [26] propose the *L-Mul* algorithm, which can be designed to approximate the FP8 multiplication process:

$$\begin{aligned} &L\text{-}Mul(x,y) = (1+m_x+m_y+2^{-l(m)})\times 2^{E_x+E_y}, \\ &l(m) = \begin{cases} m & \text{if } m \le 3, \\ 3 & \text{if } m = 4, \\ 4 & \text{if } m \ge 4, \end{cases} \end{aligned} \tag{4}$$

where m denotes the bit-width of the mantissa. By using this piecewise function approximation, the original multiplication operation can be transformed into shift and addition operations. The combination of the *L-Mul* algorithm (Eq. (4)) and the FP8 format conversion relationships (Eq. (1) and (2)) provides the bit-level representation of the *L-Mul* algorithm in binary operations:

$$\begin{aligned} L\text{-}Mul_{\text{BIN}}(x,y) = &\left(1+\frac{x[m-1:0]+y[m-1:0]}{2^m}+2^{-l(m)}\right) \\ &\times 2^{x[6:m]+y[6:m]-\text{bias}_x-\text{bias}_y}. \end{aligned} \tag{5}$$

3.2. Improved FP8 approximate multiplication

As shown in Eq. (4), *L-Mul* replaces the original multiplication operation with a piecewise linear approximation using addition. However, similar to prior works [10,27], *L-Mul* still separates the floating-point number into exponent and mantissa components, which is not well-suited for direct hardware implementation. To overcome this limitation, we propose another linear optimization (*Add-Mul*), a more flexible and hardware-friendly approximation scheme designed to maintain precision across various FP8 format configurations. Inspired by prior work on FP32 and FP16 formats [28], we first reformulate the entire FP8 multiplication directly in terms of binary operands as an expression of the form as follow,

$$Add\text{-}Mul(x,y) = x + y - offset. \tag{6}$$

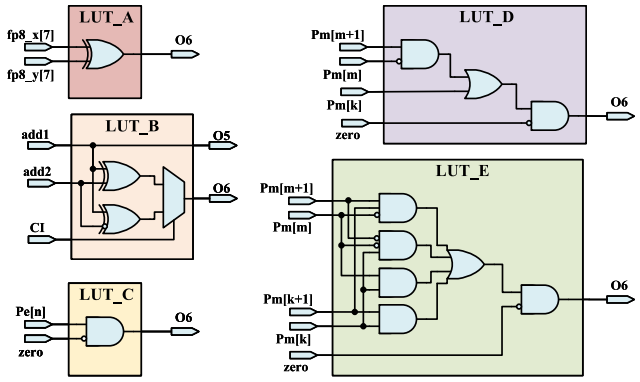

Fig. 3. The five LUT-based basic components required for approximate multiplication. Type LUT_A, _B, and _C components are shared by both *L-Mul* and *Add-Mul*, while types of LUT_D and LUT_E are specifically designed for use in the *L-Mul* architecture.

In contrast to prior work, we derive a generalized expression for the *offset* term. Earlier method often relied on empirically tuned offsets, requiring manual adjustment. However, due to the limited bit-width of FP8 formats, the *offset* exhibits a consistent structure, which is defined as follows:

$$Add\text{-}Mul(x,y)_{\text{BIN}} = x[6:0] + y[6:0] - (2^e - 1)\cdot 2^m. \tag{7}$$

Where e and m represent the bit-widths of exponent and mantissa fields, respectively. Owing to the hardware-friendly nature of the Add-Mul algorithm, the resulting hardware design is similar to L-Mul and can be implemented using only N-bit adders and LUTs. The hardware implementation will be introduced in Section 4. The evaluation of *Add-Mul* in terms of both computational accuracy and hardware deployment will be presented in Section 5. Furthermore, in Section 6, we integrate *Add-Mul* into representative neural network models to demonstrate its effectiveness and advantages in practical scenarios.

4. Hardware implementation

In this section, we first introduce the basic component designs corresponding to the two approximate methods, *L-Mul* and *Add-Mul*. These designs are compatible with mainstream FPGA platforms, including those from AMD, Altera, Lattice, Microchip, GOWIN, Anlogic, and others. Then, we detail how these components are mapped onto different FPGA hardware architectures (AMD and Altera). Finally, we present the automated register-transfer level (RTL) generation flow that supports these designs.

4.1. Basic component designs

Fig. 3 illustrates the five LUT-based basic components required for approximate multiplication. Among them, the LUT_A, _B, and _C components are shared by both *L-Mul* and *Add-Mul*, while types of LUT_D and LUT_E are specifically designed for use in the *L-Mul* architecture. The LUT_A is configured as an XOR gate to determine the sign bit of the product. Accordingly, its inputs are fixed to the MSB of the two FP8 operands. LUT_B is configured as a half-adder. When combined with the carry chain, this configuration enables efficient implementation of addition. From the above discussion, it can be seen that all five LUT-based components require no more than five inputs, ensuring full compatibility with a wide range of modern FPGA architectures that support multi-input LUT designs. This enables seamless deployment across devices from AMD, Altera, Lattice, Microchip, GOWIN, Anlogic, other vendors.

The detailed mapping of these components onto various FPGA architectures will be discussed in the following sub-sections.

Table 2
The representation of the mantissa for different carry.

[m+1,m]	Mantissa
2'b00	$1.x_m$
2'b01	$10.x_m$
2'b10	$11.x_m$
2'b11	$100.x_m$

Table 3
The *bias* $*$ configurations in the Exponent-Adder of *L-Mul* for different FP8 formats.

FP8 Type	[m+1,m]	bias*	FP8 Type	[m+1,m]	bias*
E6M1	2'b00	−31	E5M2	2'b00	−15
	2'b11	−29		2'b11	−13
	others	−30		others	−14
E4M3	2'b00	−7	E3M4	2'b00	−3
	2'b11	−5		2'b11	−1
	others	−6		others	−2
E2M5	2'b00	−1	E1M6	2'b00	0
	2'b11	1		2'b11	2
	others	0		others	1

4.2. Mapping onto AMD FPGA

4.2.1. L-Mul implementation

The design for *L-Mul* primarily consists of three parts: the Exponent-Adder, the Mantissa-Adder and the Post-Processing unit, as shown in Fig. 4(a).

The Exponent-Adder and Mantissa-Adder are implemented using LUT_B and CARRY8. The Exponent-Adder includes an m-bit adder and an m+1-bit adder, while the Mantissa-Adder includes an e-bit adder and an e+1-bit adder. LUT_B primarily functions as a half-adder. The output $O5$ corresponds to the sum (S). When the LSB of carry-in (CI) is 0, the operation is $O_5 = add1 \oplus add2$. Otherwise the operation is $O_5 = add1 \oplus (\sim add2)$. The output ($O_6$) corresponds to the C in a half-adder. In other words, $O_6 = add1 \cdot add2$. CARRY8 is used to implement addition operations. Each CARRY8 unit contains eight basic units (CC), and each CC can combine with LUT_B to function as a full adder. The CI represents the carry input from the previous stage. When the CC unit is the LSB, $CI = 0$ indicates addition and $CI = 1$ indicates subtraction. The O corresponds to the sum (S) in the full adder which can be calculated as: $O = (add1 \oplus add2) \oplus CI$. In summary, a total of N LUT_B and CC units can be combined to form an N-bit adder.

The Post-Processing Unit is primarily responsible for handling the sign bit and managing the carry of the mantissa. The LUT_A is used to determine the sign bit of the product. Specifically, it processes the bit of the inputs $x[7]$ and $y[7]$. There might be a carry occur, requiring the carry value from the mantissa to be added to the exponent. For the mantissa, we follow the carry principles of typical FP multipliers, representing the mantissa in the form of $1.x_m$. The corresponding carry handling is shown in Table 2. When the carry value is $2'b00$, the final product's mantissa is $P_m[m-1:0]$, and no carry is added to the exponent. When the carry value is $2'b01$, the mantissa is represented as $10.x_m$, requiring the decimal point to shift left by one position, i.e. the exponent is incremented by 1. In this case, the mantissa is $P_m[m-1:0]$. Similarly, when the carry value is $2'b10$, the exponent is incremented by 1, and the mantissa becomes $1'b1, P_m[m-1:1]$. For a carry value of $2'b11$, the exponent is incremented by 2, and the mantissa is $P_m[m-1:0]$. Since the product of 0 and any number is 0, the final product's mantissa and exponent can be expressed using the following formulas,

$$P'_m[m-1:0] = \begin{cases} 0, & \text{zero} = 1 \\ \{1'b1, P_m[m-1:1]\}, & P_m[m+1:m] = 2'b10 \\ P_m[m-1:0], & \text{others} \end{cases} \tag{8}$$

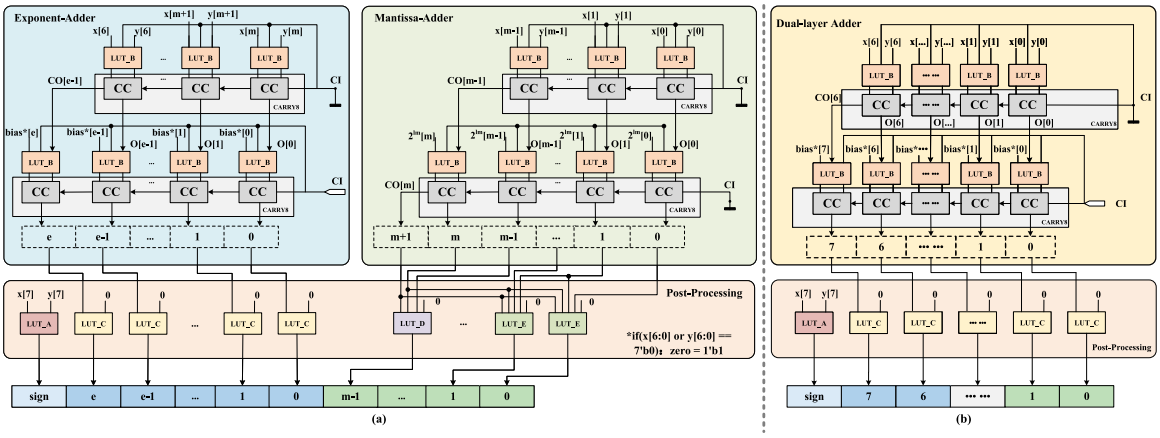

Fig. 4. Fine-grained design targeting AMD FPGAs for various FP8 approximate multipliers. (a) Fine-grained design for *L-Mul*-based FP8 approximate multiplication. (b) The fine-grained design for *Add-Mul* employs only LUT_A, LUT_B, and LUT_C components.

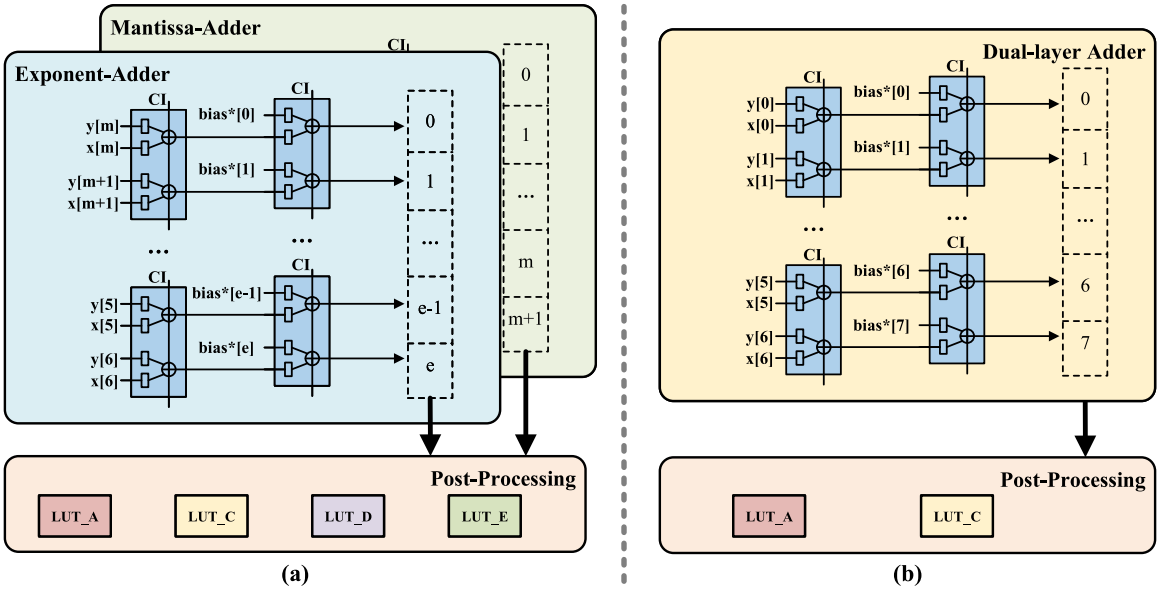

Fig. 5. Fine-grained design targeting AMD FPGAs for various FP8 approximate multipliers. (a) Fine-grained design for *L-Mul*-based FP8 approximate multiplication. (b) The fine-grained design for *Add-Mul* employs only LUT_A, LUT_B, and LUT_C components.

$$P'_e[e:0] = \begin{cases} 0, & \text{zero} = 1 \\ P_e, & P_m[m+1:m] == 2'b00 \\ P_e + 2, & P_m[m+1:m] == 2'b11 \\ P_e + 1, & \text{others.} \end{cases} \tag{9}$$

To reduce the usage of adders, we combine the bias with various carry scenarios from Table 2 and treat it as a constant, $bias*$. The corresponding values are shown in Table 3 for the types of FP8 formats. LUT_C, LUT_D and LUT_E are used to implement Eq. (8) and (9), and the remaining corresponding operations. These operations compute the final product's exponent bits, the highest mantissa bit, and the remaining mantissa bits excluding the highest bit, respectively.

To enhance the performance of the FP8 approximate multiplier, we implemented the hardware design using LUTs and carry chain primitives. Furthermore, to shorten the connection paths between LUTs and carry chains, we applied strict physical placement constraints. Specifically, as described in Section 2.2, each carry chain can connect directly to four LUTs. Therefore, we constrained the physical placement at the CLB level, ensuring that the LUTs are connected to the carry chain within the same CLB. The input FFs are placed in adjacent CLBs to guarantee the shortest possible data paths.

Table 4
The $bias*$ configurations of *Add-Mul* for different FP8 formats.

FP8 Type	bias*	FP8 Type	bias*
E6M1	−62	E3M4	−48
E5M2	−60	E2M5	−32
E4M3	−56	E1M6	0

4.2.2. Add-Mul implementation

The design of *Add-Mul* consists of two main components: a dual-layer adder and a post-processing unit, as shown in Fig. 4(b). Compared to *L-Mul*, *Add-Mul* utilizes only three types of LUT-based basic component (LUT_A, LUT_B, and LUT_C), further reducing resource consumption.

The Dual-layer Adder comprises a 7-bit and an 8-bit adder constructed using multiple *CC*s and LUT_Bs. *CC* manages carry propagation, with the CI input set to 0 for addition and 1 for subtraction at the LSB. The sum output $O = (add1 \oplus add2) \oplus CI$, and the carry output $CO = add1 + CI \cdot (add1 \oplus add2)$. When $CI = 0$, $O5 = add1 \oplus add2$ (addition); when $CI = 1$, $O5 = add1 \oplus \sim add2$ (subtraction). The carry

Table 5
Comparative evaluation of approximation errors across various approximations and data formats.

Data format	INT8	E6M1		E5M2		E4M3		E3M4		E2M5		E1M6	
Approx	[20]	*L-Mul*	*Add-Mul*	*L-Mul*	*Add-Mul*	*L-Mul*	*Add-Mul*	*L-Mul*	*Add-Mul*	*L-Mul*	*Add-Mul*	*L-Mul*	*Add-Mul*
EP ↓	5.16E-01	1.00E+00	2.50E-01	9.38E-01	5.63E-01	9.68E-01	7.64E-01	9.92E-01	8.78E-01	9.97E-01	9.38E-01	9.99E-01	9.69E-01
MAE ↓	3.97E+02	2.10E+15	3.00E+14	8.58E+05	3.73E+05	1.41E+02	8.92E+01	3.04E+00	1.70E+00	9.91E-01	4.60E-01	7.65E-01	3.43E-01
MRE ↓	6.80E-02	3.19E-01	2.78E-02	1.11E-01	3.57E-02	6.84E-02	3.81E-02	6.87E-02	3.88E-02	7.19E-02	3.91E-02	7.26E-02	3.96E-02
MSE ↓	9.63E+04	2.00E+33	1.54E+32	2.89E+13	9.12E+13	7.56E+05	3.71E+05	9.07E+01	2.69E+01	3.23E+00	5.57E-01	1.18E+00	1.84E-01

output $O6 = add1$. The values of $bias * $ are shown in Table 4 across different FP8 formats.

The Post-Processing Unit integrates LUT_A for sign determination and LUT_C for selective output gating. LUT_A computes the XOR of the operand sign bits, while LUT_C outputs $O6 = P[n] \cdot \overline{zero}, \quad (0 \le n \le 7)$. Here, $zero$=1 indicates that either of the two input operands is zero.

4.3. Mapping onto Altera FPGA

4.3.1. L-Mul implementation

The implementation of L-Mul on Altera FPGAs follows the same design principles as its deployment on AMD devices. Which based on bit-level Eq. (5) is divided into three parts: the Exponent-Adder, the Mantissa-Adder, and the Post-Processing unit, as shown in Fig. 5. As introduced in Section 2.2, due to the integration of full adders within the ALMs of Altera FPGAs, the designs of Exponent-Adder and Mantissa-Adder need to be modified, while the Post-Processing unit can be directly reused. In other words, only the LUT_B design needs adjustment, while other LUT-based basic components (as shown in Fig. 3) can be directly applied to Altera FPGAs. We configure the ALMs to operate in arithmetic mode, where each ALM includes two full adders. Thus, the combination of LUT_B + CC from AMD FPGAs can be directly replaced by the ALM implementation. Additionally, since carry cascading is supported between ALMs, the cascade can be achieved by enforcing physical placement constraints on adjacent ALMs during layout. Under this configuration, one ALM serves as an equivalent replacement for two LUT_B and CC combinations. Therefore, Exponent-Adder and Mantissa-Adder consume e ALMs and m ALMs, respectively.

4.3.2. Add-Mul implementation

Similarly, the *Add-Mul* deployment on the Altera platform can reuse most of the design from the AMD implementation. Specifically, eight ALMs are configured in arithmetic mode to replace the combination of carry chains and LUT_B in AMD's dual-layer adder module. For the post-processing module, the designs of LUT_A and LUT_C can be directly mapped onto ALMs configured in normal mode.

4.4. Automated generation flow

The proposed FP8 approximate multipliers are designed to be parameterizable, with the core structure remaining consistent across formats. Only the bias parameter adapts to the bit-widths of the exponent and mantissa defined by the selected FP8 format. To improve usability, we developed an automated generation tool, as shown in Fig. 6. This tool features an intuitive graphical user interface (GUI) that allows users to customize settings based on their specific requirements, including the target device, choice between the two approximation methods, and FP8 data format. Once configured, FP8ApproxLib automatically generates the corresponding Verilog HDL code based on these parameters. In Section 6, we demonstrate the effectiveness of FP8ApproxLib by integrating it into three neural network inference accelerators, including CNN, LLM, and Diffusion models.

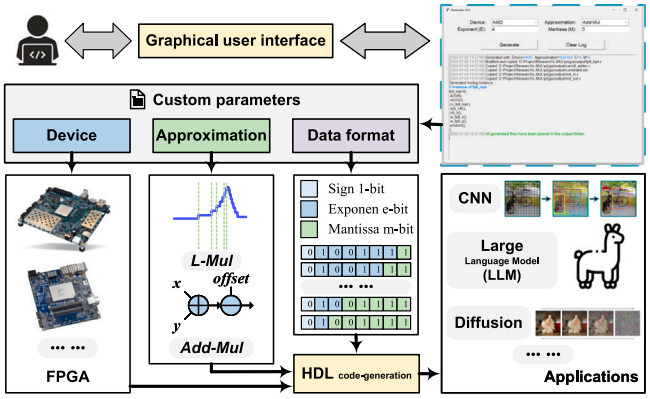

Fig. 6. The overall generation flow and applications of FP8ApproxLib.

5. Evaluation

5.1. Experimental setup

We first evaluate the error of *L-Mul* and *Add-Mul* using four metrics: Error Probability (EP), Mean Absolute Error (MAE), Mean Relative Error (MRE), and Mean Squared Error (MSE). The corresponding formulas are defined as follows:

$$EP = \frac{1}{2^N} \sum_{i=0}^{2^N-1} ED_i \neq 0, \tag{10}$$

$$MAE = \frac{1}{2^N} \sum_{i=0}^{2^N-1} ED_i, \tag{11}$$

$$MRE = \frac{1}{2^N} \sum_{i=0}^{2^N-1} \frac{ED_i}{Exact_i}, \tag{12}$$

and

$$MSE = \frac{1}{2^N} \sum_{i=0}^{2^N-1} (ED_i)^2. \tag{13}$$

In the above equations, ED denotes the Error Distance, and ED_i represents the absolute difference between the approximate and exact outputs for the ith input combination.

The FP8 approximate multipliers are design by Verilog and implemented using AMD Vivado 2022.2 and Quartus Prime Pro 23.3 for logic synthesis and placement constraints. The designs are deployed and validated on the ZCU104 Evaluation Kit of UltraScale+ FPGA and Apollo Agilex SOM, respectively. We perform multiple synthesis iterations, applying different critical path constraints in each iteration to implement each design multiple times. This approach ensures accurate measurements of area and maximum operating frequency. The Vivado simulator and power analysis tools (XPE [29]) are used to calculate power consumption. As this is the first FPGA-based FP8 approximate multiplier design, we ensure fairness by selecting previous FPGA-based INT8 approximate multipliers for comparisons of resource consumption, power consumption, and critical path delay.

Table 6
Implementation results of *L-Mul* and *Add-Mul* under different FP8 formats on AMD UltraScale+ and Altera Agilex FPGAs.

	AMD UltraScale+ FPGA								Altera Agilex FPGA					
	L-Mul				*Add-Mul*				*L-Mul*			*Add-Mul*		
	LUTs	FFs	CARRY	Frq	LUTs	FFs	CARRY	Frq	ALMs	Regs	Frq	ALMs	Regs	Frq
E6M1	22	25	4	606	24	25	2	631	27	25	655	15	25	646
E5M2	21	25	4	610	24	25	2	631	26	25	674	16	25	685
E4M3	22	25	4	617	24	25	2	691	27	25	694	15	25	691
E3M4	22	25	4	610	25	25	2	702	27	25	623	15	25	689
E2M5	22	25	4	568	25	25	2	716	24	25	519	15	25	706
E1M6	22	25	4	585	25	25	2	687	25	25	577	14	25	720

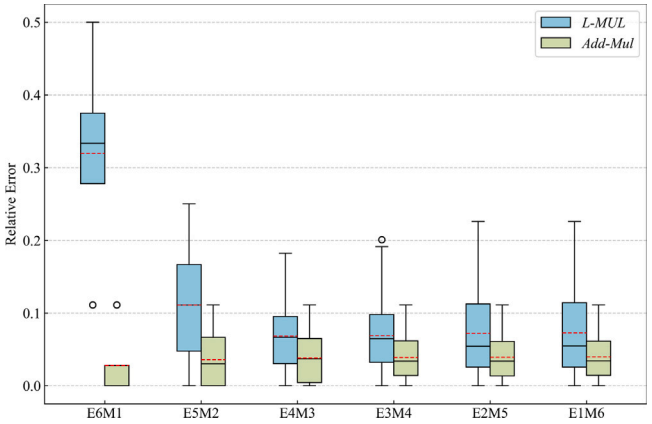

Fig. 7. Relative error box plots of *L-Mul* and *Add-Mul* under different FP8 formats. Black lines indicate the median, red dashed lines indicate the mean, and dots represent the outlier.

5.2. Error evaluation

Table 5 presents the error metrics of *L-Mul* and *Add-Mul* in different formats of FP8. Across all data, *L-Mul* exhibits the worst error on the E6M1 format, highlighted in red. In contrast, the proposed *Add-Mul* achieves the lowest error on E6M1 and E1M6, marked in green. The yellow represents second-lowest error. To be more specific, Add-Mul achieves a 53.13% average MAE reduction over L-Mul, with the largest gain on E6M1 (72.82%) and a 36.74% improvement on the widely adopted E4M3 format. In addition, both L-Mul and Add-Mul outperform the latest INT8-based approximate design, DyRecMul [20], in terms of MAE and MRE under the E4M3 format. This is an acceptable outcome given the better data range of the FP8 format. Moreover, it is important to note that when the exponent is allocated a larger bit-width, the range of representable numbers increases, which can lead to significantly larger MAE and MSE values due to the greater magnitude of errors.

To provide a more intuitive illustration of the error differences between *L-Mul* and *Add-Mul*, Fig. 7 depicts a box plot comparison under different FP8 formats. The black line within each box represents the median, and the red dashed line indicates the mean. We observe that reducing exponent bits while increasing mantissa bits leads to reduced relative error. Notably, *Add-Mul* achieves tighter boxes, lower medians, and smaller gaps between the mean and median, suggesting better precision and robustness. By comparison, *L-Mul* performs poorly in exponent-dominant formats, with both high error variance and significant skew in distribution.

5.3. Hardware implementation and evaluation

We implemented *L-Mul* and *Add-Mul* across different FP8 formats on both AMD UltraScale+ and Intel Agilex platforms. The resource consumption and corresponding frequency results are detailed in Table 6.

For AMD UltraScale+ FPGA implementation of *L-Mul*, by leveraging fine-grained primitive designs and physical placement and routing constraints, all components are mapped to the resources within six adjacent CLBs, as shown in Fig. 8(a). Meanwhile, the carry chain consumption aligns with the design intent, requiring four carry chains. It can be observed that our design consumes fewer than 22 LUTs on average. Furthermore, as shown in Fig. 8(b)(b), all components of *Add-Mul* are similarly mapped to four adjacent CLBs. The design utilizes only two carry chains, which aligns with the original design intent. As observed from the deployment results, *Add-Mul* requires fewer carry chains than *L-Mul*. Meanwhile, *L-Mul* achieves fewer LUT consumption by leveraging LUT6_2-based logic sharing in its Exponent-Adder and Mantissa-Adder.

In the case of Altera Agilex FPGA, we utilize templates and logic region constraints to compensate for the absence of fine-grained primitives. For the implementation of *L-Mul*, all components are mapped to the resources within three adjacent logic array blocks, as shown in Fig. 9(a). Using the resource property viewer tool, we confirm that the ALMs function in arithmetic mode and achieve carry cascading, as shown in Fig. 9(b). Similarly, the *Add-Mul* deployment implements the Dual-layer Adder using four ALMs operating in arithmetic mode, as shown in Fig. 9(c).

To further highlight the advantages of our designs in terms of resource utilization and power consumption, we compare it with previous FPGA-based approximate multipliers and AMD-Xilinx's intellectual property (IP) core. It should be noted that previous work has primarily been based on AMD (Xilinx) FPGAs, and there are certain differences in resource calculation between AMD and Altera FPGAs. Therefore, in this comparison, we only present the results for AMD FPGAs. We use the deployment results under the E4M3 format, which is the most commonly used FP8 format. Although the multiplication rules for FP8 and INT8 differ, we consider this comparison valuable due to their identical data bit-width. Table 7 presents the comparison results, these values are reported or estimated from original reports. It should be noted that the data in the Table 7 is sourced from the original reports of each respective paper. As a result, the comparisons involve different devices and operating frequencies. For each item, it can be observed that our designs, *L-Mul* and *Add-Mul*, exhibit the lowest resource consumption. Compared to previous 8-bit approximate designs, the LUT usage is reduced by an average of 49.9%. Additionally, compared to the FP8-compatible DyRecMul [20], our designs achieves a lower delay. It is important to note that Refs. [16,30], and [17] are implemented on AMD-Xilinx 7-series FPGAs, which introduces some power consumption differences. Additionally, the power data reported in [31] is recorded at 1.2 V supply and 100 MHz frequency, which is significantly lower than the results from other design. To better demonstrate the power-efficiency advantages of our design, we performed a Pareto analysis to visually illustrate the differences between our design and others, as shown in Fig. 10. The Pareto analysis is constructed following the method proposed by previous work [17], by connecting non-dominated design points in the area (LUTs)–delay space. A design point is considered standing on Pareto frontier if no other point offers both lower area and lower delay simultaneously. The result shows that both of our

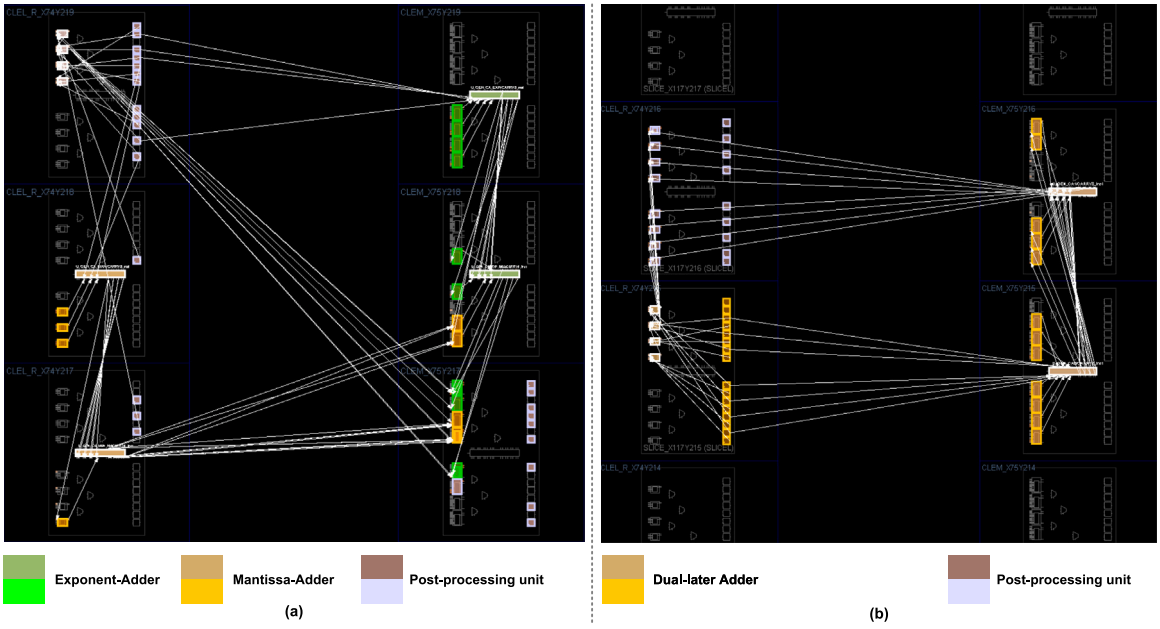

Fig. 8. (a) The layout of *L-Mul* on the AMD UltraScale+ FPGA. (b) The layout of *Add-Mul* on the AMD UltraScale+ FPGA.

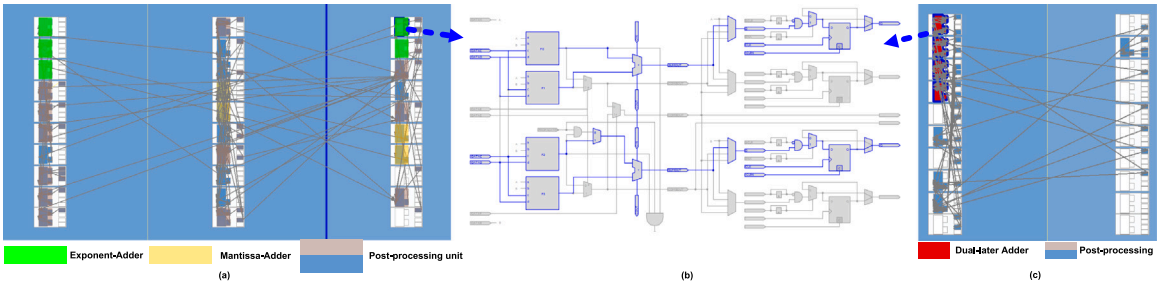

Fig. 9. (a) The layout of *L-Mul* on the Altera Agilex FPGA. (b) The demonstration of ALM runs in arithmetic mode. (c) The layout of *Add-Mul* on the Altera Agilex FPGA.

Table 7
Resource and performance comparison of 8-bit approximate multipliers based on reported data in prior work.

Designs	Year	Device Family	LUTs	FFs	CARRY	DSPs	Max Frq (MHz)	Delay (ns)	Power (mW)
AMD-Xilinx[1] (Exact INT8 Signed)	2015	UltraScale+ (16 nm)	69	16	14	0	**730**	3.54	2.32
Ullah [30] (INT8 Unsigned)	2018	Virtex-7 (28 nm)	56	**0**	4	0	/	6.95	1.68
Van Toan [31] (INT8 Unsigned)	2020	Spartan-6 (45 nm)	59	/	4	0	/	4.65	**0.432**[2]
Ullah [16] (INT8 Unsigned)	2020	Virtex-7 (28 nm)	37	35	4	0	/	**3.41**	1.65[3]
Ullah [17] (INT8 Signed)	2022	Virtex-7 (28 nm)	54	32	9	0	/	4.37	1.66
DyRecMul [20] (FP8 to INT8 Signed)	2024	UltraScale+ (16 nm)	35	/	**0**	0	699	5.72	1.20[3]
L-Mul (FP8 E4M3)	2025	UltraScale+ (16 nm)	**22**	25	4	0	617	4.85	1.34[4]
Add-Mul (FP8 E4M3)	2025	UltraScale+ (16 nm)	**24**	25	2	0	691	4.34	1.34[4]

[1] The IP version is Multiplier v12.0 and values reported at 200 MHz.
[2] Values reported from the original report (operating at 1.2V and 100 MHz).
[3] Values estimated from the reported PDP in the original report.
[4] Values reported at 200MHz.

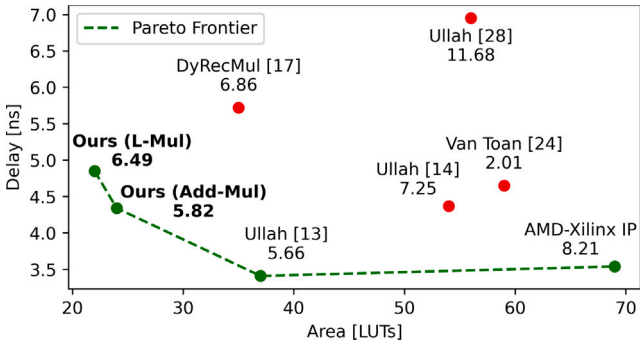

Fig. 10. The Pareto analysis under area and delay. The data under the design point represents the Power-Delay Product (PDP).

Table 8
Evaluation of accuracy loss for different data formats across various datasets.

Data Formats		MNIST	CIFAR-10	ImageNetV2
Exact	FP32	0	0	0
	FP8 (E4M3)	−0.04%	−0.36%	−0.18%
	INT8	−0.1%	−1.69%	−0.35%
Approx	Ullah [30] (INT8 Unsigned)	−1.81%	−3.1%	−1.52%
	Ullah [16] (INT8 Unsigned)	−0.88%	−2.83%	−5.01%
	Ullah [17] (INT8 Signed)	−1.33%	−1.56%	−0.49%
	DyRecMul [20] (INT8 Unsigned)	−1.66%	−7.25%	−0.21%
	L-Mul (E4M3)	−0.96%	−1.56%	−0.83%
	Add-Mul (E4M3)	−0.82%	−0.6%	−0.21%

designs, *L-Mul* and *Add-Mul*, lie on the Pareto frontier. Additionally, due to their outstanding power-efficiency, both of our designs, achieves the second smallest Power-Delay Product (PDP) among the compared implementations.

6. Case studies

To demonstrate the practical applicability of the proposed approximate multipliers, we conduct a case study using FP8ApproxLib. Specifically, both *L-Mul* and *Add-Mul* are integrated into three representative neural network inference accelerators: CNN, LLM, and diffusion model. For comparison, prior open-source 8-bit approximate multipliers are also integrated into the same accelerator designs.

6.1. CNN accelerator integration

We integrate our approximate multipliers into a CNN accelerator. The CNN model is built using PyTorch and quantized using a post-training quantization (PTQ) strategy [32]. The backbone of the model is Faster R-CNN, which consists of 13 convolutional layers, 1 Region Proposal Network, 1 RoI pooling layer, and 2 fully connected layers. We evaluate the inference accuracy on three representative CNN datasets (MNIST, CIFAR-10, and ImageNetV2). Table 8 shows the average accuracy loss for the corresponding models under different data formats. FP8, with its superior dynamic range, incurs less accuracy loss compared with INT8. For the comparison of 8-bit approximation methods, both *L-Mul* and *Add-Mul* demonstrate better robustness. Specifically, across three datasets, *Add-Mul* achieves the lowest average accuracy drop (−0.54%) among all methods.

To demonstrate the high energy efficiency of our approximate multiplier in hardware deployment, we integrate it into a CNN accelerator to replace the original DSP blocks, and evaluate the corresponding resource usage and power consumption. For comparison, prior works are also integrated and deployed using the same methodology. Considering that most baseline designs are developed on the Virtex-7 platform, we adopt the VC709 Connectivity Kit as the target platform. This choice does not affect our implementation, as introduced in Section 2.2, the

Table 9
Hardware deployment results for neural network inference using different multipliers, obtained through our own experiments on a unified platform (VC709 Connectivity Kit)

Multiplier	LUT	FF	DSP	Power (W)	WNS (ns)
INT8 (Exact)	117k	95k	1k	9.46	0.459
Ullah [30]	188k	136k	0	10.33	−0.78
Ullah [16]	166k	152k	0	9.56	−0.208
Ullah [17]	186k	157k	0	10.21	−0.347
DyRecMul [20]	250k	169k	0	12.69	0.012
L-Mul (E4M3)	143k	122k	0	9.08	0.035
Add-Mul (E4M3)	142k	122k	0	9.08	0.042

[1] The data is collected using the XPE tool, with the operating frequency uniformly set to 200MHz.
[2] The timing constraint is uniformly set to 5 ns.

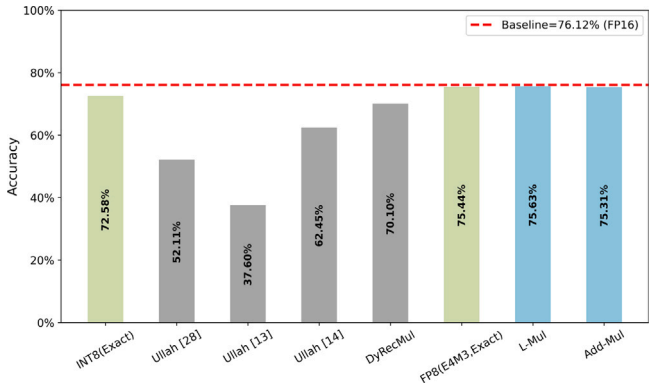

Fig. 11. Comparison of inference accuracy on the GSM8K using LLaMA3.1-8B under different data formats and 8-bit approximate multipliers.

Virtex-7 and UltraScale+ devices feature the same CLB architecture. The synthesis and implementation are conducted using AMD Vivado 2022.2. As shown in Table 9, All of resource utilization, power consumption, and the worst negative slack (WNS) are obtained from the post-implementation reports generated by Vivado. Power consumption is obtained as the sum of GTH, hard IP, and dynamic components. Among them, GTH and hard IP are used for peripheral Component interconnect express (PCIe) communication, and this portion of the power overhead is identical across all accelerators. The data shows that all 8-bit approximate multipliers enable DSP-free designs. However, only our design and DyRecMul are able to maintain the original operating frequency of 200 MHz. Among all 8-bit approximate designs, our approach consumes the lowest LUTs and FFs, leading to reduced power consumption. Specifically, compared to the original INT8 design, our method achieves a 4.02% reduction in power, and compared to the DyRecMul, power is reduced by 28.45%. In addition, the table includes a default accelerator implementation as part of the ablation study. This design is directly derived from Eq. (5) for RTL implementation, without applying any primitives-based fine-grained optimizations or physical placement constraints. The results show that under the default strategy, the default implementation incurs higher resource usage and exhibits inferior operation frequency. To sum up, these results demonstrate the superiority of our design especially in power efficiency.

6.2. LLM accelerator integration

Although LLM is the initial target for the *L-Mul* [26] approximation, no hardware implementation exists to evaluate its practical value. To address this, we incorporate FP8ApproxLib into an LLM inference accelerator, showcasing the strengths of FP8 approximate multiplier in real deployment. We use LLaMA-3.1-8B as the evaluation model and test various 8-bit approximate multipliers and data formats on GSM8K [33], a dataset of 8.5k grade-school math problems. On the hardware side,

Table 10
Hardware deployment results for LLM inference using different multipliers, obtained through our own experiments on an unified platform (ZCU104)

Multiplier	LUT	FF	DSP	Power (W)[1]	WNS (ns)[2]
Original (Exact)	102k	150k	310	3.99	0.157
Ullah [30]	118k	145k	176	3.71	0.133
Ullah [16]	110k	147k	176	3.68	0.201
Ullah [17]	129k	146k	176	3.69	0.134
DyRecMul [20]	/	/	/	/	/
L-Mul (E4M3)	106k	146k	176	3.65	0.242
Add-Mul (E4M3)	106k	146k	176	3.65	0.305

[1] The data is collected using the XPE tool, with the operating frequency uniformly set to 266MHz.
[2] The timing constraint is uniformly set to 3.75 ns.

Table 11
Comparing Performance with Different Data Formats on the LSUN Bedroom Dataset with 256×256 Image Resolution.

Data Formats	DDIM 100 steps	DDIM 250 steps			
		FID↓	sFID↓	FID↓	sFID↓
Exact	FP32	25.03	216.08	22.88	216.91
	FP8 (E4M3)	29.81	226.51	26.58	225.60
	INT8	32.04	229.33	34.61	238.59
Approx	Ullah [30] (INT8 Unsigned)	48.94	273.17	69.24	290.0
	Ullah [16] (INT8 Unsigned)	44.39	262.84	58.21	272.33
	Ullah [17] (INT8 Signed)	80.89	362.46	92.95	363.48
	DyRecMul [20] (INT8 Unsigned)	39.22	250.60	40.67	247.97
	L-Mul (E4M3)	34.20	239.21	37.74	239.04
	Add-Mul (E4M3)	33.73	232.0	32.75	230.77

we integrate the approximate multipliers into our previous accelerator framework [34].

Fig. 11 compares the resulting accuracy across configurations, with the red dashed line marking the FP16 baseline. The results show that negligible accuracy loss is observed for exact FP8, *L-Mul*, and *Add-Mul* when compared to FP16. Conversely, INT8 approximate multipliers exhibit substantial degradation, with some variants rendering the model unusable.

For hardware implementation, we replace the DSPs in the Vector Processing Unit (VPU) design with 8-bit approximate multipliers. To accommodate the increased LUT usage, we migrate the hardware platform from KV260 to ZCU104. Power consumption is estimated using the XPE tool under identical conditions, with the clock constraint uniformly set to 266 MHz. The deployment results for different 8-bit approximate multipliers are summarized in Table 10. We attempted several integration strategies for DyRecMul, including the use of synthesis constraints such as the DONT_TOUCH attribute, but were unable to achieve a successful implementation. As observed in Table 10, all other integrated designs arec able to meet timing constraints on the ZCU104 board, owing to the ample resources of the UltraScale+ FPGA. Furthermore, the two FP8 approximate multipliers in FP8ApproxLib exhibit optimal power efficiency and WNS performance.

6.3. Diffusion accelerator integration

We also integrate the proposed approximate multipliers into our previously developed diffusion accelerator (Diff-Acc) [35] and perform comparisons at both the software and hardware levels. On the software side, we apply PTQ using both INT8 and FP8 (E4M3) formats [36]. Under the same quantized operators, we directly substitute the corresponding multiplication operations with their approximate approaches during inference. On the hardware side, the original Diff-Acc directly employed DSP blocks to perform 8-bit multiplications without any specialized optimization. To ensure a fair comparison, we preserve the same data path architecture from Diff-Acc and replace the original DSP units with our proposed approximate multipliers. It is worth noting that

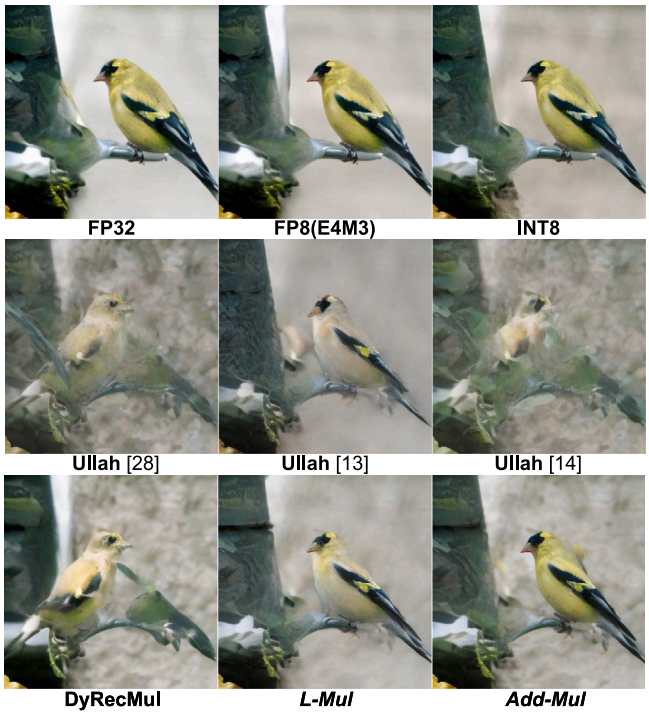

Fig. 12. Example images generated by different data formats and approximate multipliers.

Table 12
Hardware deployment results for Diffusion inference using different multipliers, obtained through our own experiments on a unified platform (XCZU19EG within ALINX Z19-P board)

Multiplier	LUT	FF	DSP	Power (W)[1]	WNS (ns)[2]
INT8 (Exact)	213k	175k	1k	3.84[3]	0.022
Ullah [30]	371k	271k	127	4.63	−3.26
Ullah [16]	348k	232k	127	4.33	−0.91
Ullah [17]	375k	273k	127	4.75	−0.952
DyRecMul [20]	376k	227k	127	4.56	−0.556
L-Mul (E4M3)	299k	201k	127	3.82	−0.495
Add-Mul (E4M3)	299k	201k	127	3.82	−0.464

[1] The data is collected using the XPE tool, with the operating frequency uniformly set to 200MHz.
[2] The timing constraint is uniformly set to 5 ns.
[3] Power differences stem from platform variation: XCZU9EG (0.63 W static) in the original work [35] and XCZU19EG (1.13 W static) in this work.

the original Diff-Acc implementation already utilized nearly 80% of the LUT resources on the ZCU-102. As a result, we migrate the deployment to the ALINX Z19-P board, which is also based on the UltraScale+ MPSoC architecture but offers a larger resource budget. In particular, approximate multipliers are used to replace DSP-based multiplications in the conv and linear modules of Diff-Acc, while the multiplication units in normalization and nonlinear functions are left unmodified.

We first evaluate the performance of different data formats on the LSUN [37] bedroom dataset using the DDIM [38] sampling method with 100 and 200 steps, respectively. Table 11 presents the accuracy differences of the diffusion algorithm after quantization, measured in terms of Fréchet Inception Distance (FID) and sliced FID (sFID). It can be observed that the approximate FP8 methods achieve accuracy comparable to, or even better than INT8. Given that, FP8 approximation eliminates the multiplication operations, it offers the potential for improved hardware energy efficiency. Furthermore, to provide a more intuitive comparison of the generative image quality, Fig. 12 illustrates the output states under different data formats. As observed from the generated images, the visual quality aligns with the metrics reported

in Table 11. L-Mul and Add-Mul deliver the best performance among all approximate multipliers.

Table 12 reports the resource utilization, power consumption, and WNS of different approximate multipliers integrated into Diff-Acc. Although the ALINX Z19-P offers a larger resource budget, integrating approximate multipliers still results in LUT utilization exceeding 65%. This high utilization poses significant placement and routing challenges for Diff-Acc, which lacks a unified architecture for large-scale matrix computation arrays. As a result, none of the approximate multipliers can operate at the originally specified clock frequency. However, thanks to our fine-grained design, FP8ApproxLib still achieves the lowest LUT utilization and corresponding power consumption among all 8-bit approximate multipliers.

7. Conclusion

This paper presents FP8ApproxLib, to the best of our knowledge, this is the first FP8 approximate multiplier design with architecture-aware fine-grained optimization and deployment for modern FPGA platforms. Specifically, we begin by conducting a bit-level analysis of existing FP8 approximation methods and introduce a novel improved approach named Add-Mul. Building upon this, we perform fine-grained hardware deployment tailored to different FPGA architectures. Furthermore, the proposed designs are integrated into an automated HDL generation tool, enabling users to generate synthesizable code under various configurations. Experimental results show that Add-Mul achieves lower approximation error and more favorable hardware mapping compared to the previously proposed L-Mul. Compared to prior 8-bit approximate multipliers, our FP8 designs demonstrate the lowest LUT usage and power consumption. Finally, we integrate FP8ApproxLib into several representative neural network accelerators, further validating its practical applicability.

CRediT authorship contribution statement

Ruiqi Chen: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Yangxintong Lyu:** Writing – review & editing, Validation, Software, Methodology, Formal analysis. **Han Bao:** Writing – review & editing, Validation, Software, Methodology, Formal analysis. **Shidi Tang:** Writing – review & editing, Validation, Software. **Jindong Li:** Writing – review & editing, Validation, Software. **Yanxiang Zhu:** Writing – review & editing, Visualization, Resources. **Ming Ling:** Writing – review & editing, Supervision, Project administration. **Bruno da Silva:** Writing – review & editing, Supervision, Project administration, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported in part by the National Natural Science Foundation of China under Grants 92464301. This research work is also supported by the Big Data Computing Center of Southeast University.

Data availability

Data will be made available on request.

References

[1] M. Dampfhoffer, T. Mesquida, A. Valentian, L. Anghel, Backpropagation-based learning techniques for deep spiking neural networks: A survey, IEEE Trans. Neural Netw. Learn. Syst. 35 (9) (2024) 11906–11921, http://dx.doi.org/10.1109/TNNLS.2023.3263008.

[2] Q. Han, Y. Hu, F. Yu, H. Yang, B. Liu, P. Hu, R. Gong, Y. Wang, R. Wang, Z. Luan, D. Qian, Extremely low-bit convolution optimization for quantized neural network on modern computer architectures, in: Proceedings of the 49th International Conference on Parallel Processing, ICPP, ACM, New York, NY, USA, 2020, pp. 1–12, http://dx.doi.org/10.1145/3404397.3404407.

[3] Z. Yao, R. Yazdani Aminabadi, M. Zhang, X. Wu, C. Li, Y. He, ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers, in: Advances in Neural Information Processing Systems, vol. 35, Curran Associates, Inc., 2022, pp. 27168–27183.

[4] H. Shen, N. Mellempudi, X. He, Q. Gao, C. Wang, M. Wang, Efficient Post-training Quantization with FP8 Formats, in: Proceedings of Machine Learning and Systems, vol. 6, 2024, pp. 483–498.

[5] D.R. Lutz, A. Saini, M. Kroes, T. Elmer, 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.

[6] S.K. Lee, A. Agrawal, J. Silberman, M. Ziegler, M. Kang, S. Venkataramani, N. Cao, B. Fleischer, M. Guillorn, M. Cohen, et al., A 7-nm four-core mixed-precision AI chip with 26.2-TFLOPS hybrid-FP8 training, 104.9-TOPS INT4 inference, and workload-aware throttling, IEEE J. Solid-State Circuits 57 (1) (2021) 182–197.

[7] S.K. Venkataramanaiah, J. Meng, H.-S. Suh, I. Yeo, J. Saikia, S.K. Cherupally, Y. Zhang, Z. Zhang, J.-S. Seo, A 28-nm 8-bit floating-point tensor core-based programmable CNN training processor with dynamic structured sparsity, IEEE J. Solid-State Circuits 58 (7) (2023) 1885–1897.

[8] A.C. Elster, T.A. Haugdahl, Nvidia hopper GPU and grace CPU highlights, Comput. Sci. Eng. 24 (2) (2022) 95–100.

[9] C. Chen, S. Yang, W. Qian, M. Imani, X. Yin, C. Zhuo, Optimally Approximated and Unbiased Floating-Point Multiplier with Runtime Configurability, in: Proceedings of the 39th International Conference on Computer-Aided Design, ICCAD, ACM, 2020, pp. 1–9.

[10] M.S. Ansari, B.F. Cockburn, J. Han, An improved logarithmic multiplier for energy-efficient neural computing, IEEE Trans. Comput. 70 (4) (2020) 614–625.

[11] S. Zheng, Z. Li, Y. Lu, J. Gao, J. Zhang, L. Wang, HEAM: High-efficiency approximate multiplier optimization for deep neural networks, in: 2022 IEEE International Symposium on Circuits and Systems, ISCAS, IEEE, 2022, pp. 3359–3363.

[12] L. Xuan, K.-F. Un, C.-S. Lam, R.P. Martins, An FPGA-based energy-efficient reconfigurable depthwise separable convolution accelerator for image recognition, IEEE Trans. Circuits Syst. II: Express Briefs 69 (10) (2022) 4003–4007.

[13] A. Rangsikunpum, S. Amiri, L. Ost, BIDS: An efficient intrusion detection system for in-vehicle networks using a two-stage binarised neural network on low-cost FPGA, J. Syst. Archit. 156 (2024) 103285.

[14] A. Ahmed, N. Sheybani, D. Moreno, N.B. Njungle, T. Gong, M. Kinsy, F. Koushanfar, AMAZE: Accelerated MiMC hardware architecture for zero-knowledge applications on the edge, in: Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, 2024, pp. 1–9.

[15] S. Ullah, S.S. Murthy, A. Kumar, SMApproxLib: Library of FPGA-based Approximate Multipliers, in: Proceedings of the 55th Annual Design Automation Conference (DAC), ACM, New York, NY, USA, 2018, pp. 1–6.

[16] S. Ullah, H. Schmidl, S.S. Sahoo, S. Rehman, A. Kumar, Area-optimized accurate and approximate softcore signed multiplier architectures, IEEE Trans. Comput. 70 (3) (2020) 384–392.

[17] S. Ullah, S. Rehman, M. Shafique, A. Kumar, High-performance accurate and approximate multipliers for FPGA-based hardware accelerators, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (2) (2022) 211–224.

[18] Y. Liu, S. Ullah, A. Kumar, BitSys: Bitwise systolic array architecture for multi-precision quantized hardware accelerators, in: 2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2024, 220–220.

[19] S. Ullah, S.S. Sahoo, A. Kumar, AxOSpike: Spiking neural networks-driven approximate operator design, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 43 (11) (2024) 3324–3335.

[20] S. Vakili, M. Vaziri, A. Zarei, J.P. Langlois, DyRecMul: Fast and low-cost approximate multiplier for FPGAs using dynamic reconfiguration, ACM Trans. Reconfigurable Technol. Syst. (2024).

[21] V. Leon, T. Paparouni, E. Petrongonas, D. Soudris, K. Pekmestzi, Improving Power of DSP and CNN Hardware Accelerators Using Approximate Floating-point Multipliers, ACM Trans. Embed. Comput. Syst. 20 (5) (2021) 1–21.

[22] R. Chen, Y. Lyu, H. Bao, J. Liu, Y. Zhu, S. Tang, M. Ling, B. Da Silva, FPGA-Based Approximate Multiplier for FP8, in: 2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2025, pp. 01–09, http://dx.doi.org/10.1109/FCCM62733.2025.00079.

[23] M. van Baalen, A. Kuzmin, S.S. Nair, Y. Ren, E. Mahurin, C. Patel, S. Subramanian, S. Lee, M. Nagel, J. Soriaga, et al., FP8 versus INT8 for efficient deep learning inference, 2023, arXiv preprint arXiv:2303.17951.

[24] AMD Xilinx, UltraScale Architecture Configurable Logic Block User Guide, 2017, URL https://docs.amd.com/v/u/en-US/ug574-ultrascale-clb. (Accessed: 11 Novenber 2024).
[25] Altera Intel, Agilex™ 7 FPGAs and SoCs device overview, 2024, URL https://www.intel.com/content/www/us/en/docs/programmable/683458/current/adaptive-logic-module-in-fpgas-and-socs.html. (Accessed: 11 Novenber 2024).
[26] H. Luo, W. Sun, Addition is all you need for energy-efficient language models, 2024, arXiv preprint arXiv:2410.00907.
[27] H. Saadat, H. Bokhari, S. Parameswaran, Minimally Biased Multipliers for Approximate Integer and Floating-Point Multiplication, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 37 (11) (2018) 2623–2635.
[28] O. Gustafsson, N. Hellman, Approximate floating-point operations with integer units by processing in the logarithmic domain, in: 2021 IEEE 28th Symposium on Computer Arithmetic, ARITH, IEEE, 2021, pp. 45–52.
[29] AMD Xilinx, AMD power estimator (XPE), 2025, URL https://www.amd.com/en/products/adaptive-socs-and-fpgas/technologies/power-efficiency/power-estimator.html. (Accessed: 29 May 2025).
[30] S. Ullah, S. Rehman, B.S. Prabakaran, F. Kriebel, M.A. Hanif, M. Shafique, A. Kumar, Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators, in: Proceedings of the 55th Annual Design Automation Conference, DAC, ACM, 2018, pp. 1–6.
[31] N. Van Toan, J.-G. Lee, FPGA-based multi-level approximate multipliers for high-performance error-resilient applications, IEEE Access 8 (2020) 25481–25497.
[32] M. Nagel, M. Fournarakis, R.A. Amjad, Y. Bondarenko, M. Van Baalen, T. Blankevoort, A white paper on neural network Quantization0, 2021, arXiv preprint arXiv:2106.08295.
[33] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al., Training verifiers to solve math word problems, 2021, arXiv preprint arXiv:2110.14168.
[34] J. Li, T. Li, G. Shen, D. Zhao, Q. Zhang, Y. Zeng, Pushing up to the limit of memory bandwidth and capacity utilization for efficient LLM decoding on embedded FPGA, in: 2025 Design, Automation & Test in Europe Conference, DATE, IEEE, 2025, pp. 1–7.
[35] S. Tang, R. Chen, R. Liu, Y. Lv, P. Zheng, H. Li, M. Ling, Diff-acc: An efficient FPGA accelerator for unconditional diffusion models, ACM Trans. Embed. Comput. Syst. (TECS) 24 (5) (2025) 1–21.
[36] C. Chen, Low-bitwidth Floating-Point Quantization for Diffusion Models, University of Toronto (Canada), 2024.
[37] A.Q. Nichol, P. Dhariwal, Improved denoising diffusion probabilistic models, in: Proceedings of the 38th International Conference on Machine Learning, PMLR, 2021, pp. 8162–8171.
[38] J. Song, C. Meng, S. Ermon, Denoising diffusion implicit models, in: The 9th International Conference on Learning Representations, ICLR, 2021, pp. 1–22.

Ruiqi Chen received the B.S. degree in Electronic Science and Technology from Southeast University Chengxian College in 2017, and the M.S. degree in Integrated Circuit Engineering from Fuzhou University in 2020. He served as a research assistant (2020–2023) at VeriMake Innovation Lab, Fudan University, and Southeast University, respectively.

He is currently pursuing the joint Ph.D. degree in Engineering Sciences and Computer Science Engineering at the Vrije Universiteit Brussel and Universiteit Gent. His research interests include domain-specific architecture and reconfigurable technologies.

Yangxintong Lyu received her Ph.D. degree in Engineering Science from Vrije Universiteit Brussel, Belgium, in 2024. Previously, she earned the B.Eng. degree in Software Engineering from Xidian University, Xi'an, China, in 2017, and the M.Sc. degree in Applied Computer Science from Vrije Universiteit Brussel, Belgium, in 2018. Her current research interests include Machine Learning and its applications in Computer Vision and Smart Mobility Systems.

Han Bao received his B.S. degree in Microelectronics from Northwestern Polytechnical University, China, in 2023, and the M.S. degree in Electrical Engineering from Vrije Universiteit Brussel, Belgium, in 2024. He is pursuing the Ph.D. degree in Engineering Science at Vrije Universiteit Brussel, Belgium. His research interests include FPGA-based edge AI and NN model implementation.

Shidi Tang received the B.S. degree in communication engineering from Central China Normal University, Wuhan, China, in 2020, and the MS degree in biomedical engineering from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2023. He is currently working toward the PhD degree at the Southeast University, Nanjing, China.

His current research interests include domain specific accelerator, efficient machine learning and parallel computing.

Jindong Li received the bachelor's degree from Sun Yat-sen University, Guangzhou, Guangdong, China, in 2022. He is currently pursuing the master's degree with the Brain-inspired Cognitive Intelligence Lab, Institute of Automation, Chinese Academy of Sciences, Beijing, China, under the supervision of Prof. Qian Zhang and Prof. Yi Zeng. His research focuses on hardware acceleration of brain-inspired algorithms, domain-specific architecture, and FPGA system design.

Yanxiang Zhu received the B.S. and M.S. degrees from Southeast University, Nanjing, China, in 2004 and 2007, respectively.

He is the founder of VeriMake, Nanjing Renmian Integrated Circuit Company Ltd., Nanjing. His current research interests include human–computer interaction and domain-specific architecture.

Ming Ling received the B.S., M.S., and Ph.D. degrees from Southeast University, Nanjing, China, in 1994, 2001, and 2011, respectively.

He is currently a Professor with the National ASIC System Engineering Technology Research Center, Southeast University. His current research interests include memory subsystem of system-on-chip (SoC), processor architecture, and domain-specific architecture.

Bruno da Silva obtained his Ph.D. degree in 2019 from the Vrije Universiteit Brussel and Universiteit Gent. Since 2022 he has been an Assistant Professor with the Engineering Sciences Faculty of the Vrije Universiteit Brussel. As a member of the Electronics and Informatics department (ETRO), he is responsible for research in reconfigurable technologies,low-power biomedical devices, and edge AI.