DyRecMul: Fast and Low-Cost Approximate Multiplier for FPGAs using Dynamic Reconfiguration

SHERVIN VAKILI, Institut national de la recherche scientifique, Energie Materiaux Telecommunications Centre, Montreal, Canada
MOBIN VAZIRI, Polytechnique Montréal, Montreal, Canada
AMIRHOSSEIN ZAREI, Institut national de la recherche scientifique, Energie Materiaux Telecommunications Centre, Montreal, Canada
J.M. PIERRE LANGLOIS, Polytechnique Montréal, Montreal, Canada

Multipliers are widely-used arithmetic operators in digital signal processing and machine learning circuits. Due to their relatively high complexity, they can have high latency and be a significant source of power consumption. One strategy to alleviate these limitations is to use approximate computing. This paper thus introduces an original FPGA-based approximate multiplier specifically optimized for machine learning computations. It utilizes dynamically reconfigurable lookup table (LUT) primitives in AMD-Xilinx technology to realize the core part of the computations. The paper provides an in-depth analysis of the hardware architecture, implementation outcomes, and accuracy evaluations of the multiplier proposed in INT8 precision. The paper also facilitates the generalization of the proposed approximate multiplier idea to other datatypes, providing analysis and estimations for hardware cost and accuracy as a function of multiplier parameters. Implementation results on an AMD-Xilinx Kintex Ultrascale+ FPGA demonstrate remarkable savings of 64% and 67% in LUT utilization for signed multiplication and multiply-and-accumulation configurations, respectively when compared to the standard Xilinx multiplier core. Accuracy measurements on four popular deep learning (DL) benchmarks indicate a minimal average accuracy decrease of less than 0.29% during post-training deployment, with the maximum reduction staying less than 0.33%. The source code of this work is available on GitHub.

CCS Concepts: • **Hardware → Arithmetic and datapath circuits**; **Reconfigurable logic applications**; **Hardware accelerators**.

Additional Key Words and Phrases: approximate multiplier, field-programmable gate array, dynamic reconfiguration, artificial intelligence hardware

1 INTRODUCTION

Computational circuits may become a bottleneck in wireless communications, digital signal processing, multimedia, image processing, machine learning (ML), etc., due to their high energy consumption, long delay, and large circuit area utilization. Applications such as neural networks have inherent error resilience, which presents an opportunity to use approximation techniques for enhancing their efficiency [7, 18]. Multiplication is a fundamental operation in computations, and, to enhance its efficiency, various approximate multipliers have been proposed.

Authors' Contact Information: Shervin Vakili, Institut national de la recherche scientifique, Energie Materiaux Telecommunications Centre, Montreal, Quebec, Canada, shervin.vakili@inrs.ca; Mobin Vaziri, Polytechnique Montréal, Montreal, Quebec, Canada, mobin.vaziri@polymtl.ca; Amirhossein Zarei, Institut national de la recherche scientifique, Energie Materiaux Telecommunications Centre, Montreal, Quebec, Canada, 94.amir.zarei@gmail.com; J.M. Pierre Langlois, Polytechnique Montréal, Montreal, Quebec, Canada, pierre.langlois@polymtl.ca.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1936-7406/2024/5-ART
https://doi.org/10.1145/3663480

Just Accepted

These multipliers aim to reduce delay, energy consumption, and area [45]. In 1962, Mitchell [30] introduced the first logarithm-based multiplier. This multiplier converted multiplications into addition operations, and it deployed approximation methods to calculate logarithmic expressions. Since then, there have been significant advancements in logarithm-based approximate multipliers [3, 4, 27, 33]. Liu et al. used approximate adders to enhance accuracy and reduce power consumption in logarithm-based multipliers [27]. Given that multiplication is a nonlinear operation, several works proposed methods to use linearization for approximate multiplication [6, 16]. An approximate linearization algorithm, called ApproxLP [16], utilized comparators to separate different sub-domains and allocate a proper linear function to each one instead of a nonlinear multiplication. This method, however, requires additional comparators to increase accuracy and reduce sub-domain size, which can result in longer delays and increased area [45]. Chen et al. introduced an optimally approximate multiplier (OAM) [6] to improve linearization and minimize the number of comparators, resulting in lower delay and improved performance. Other studies suggest utilizing a combination of accurate and approximate multipliers with varying precision levels to achieve the necessary accuracy, but this may come at the cost of higher energy consumption and circuit area [14, 15].

In non-logarithmic multiplication, the approximation can be introduced at four distinct stages: data input, partial product generation, accumulation, and Booth encoding [45]. Truncating input data is often used to reduce bitwidth, and it can be done through methods such as the dynamic segment method (DSM) and static segment method (SSM) [13, 32, 38, 39]. DSM keeps a specified number of bits from the most significant '1' in an n-bit operand but requires more hardware resources than SSM. SSM has limited and pre-determined options for truncating input data, resulting in more redundant bits [32, 45]. Some papers have introduced approximations in the partial product generation stage [21, 42, 46, 47]. However, the most beneficial stage to introduce approximation is during partial product accumulation, which typically requires the largest circuit area and computation time.

Compressors can be used for counting ones in tree-based partial product accumulations [45], such as the Wallace tree [43] and Dadda tree [8]. Instead of exact compressors, a variety of approximate compressors have been introduced to decrease circuit area and delay [1, 2, 10, 11, 23, 28, 29, 31, 34, 35, 42, 44, 47, 48]. These include the approximate 1-bit half adder, approximate 1-bit full adder, and approximate 4-2 compressors [2, 31, 34, 48]. To further decrease energy consumption, delay, and hardware utilization, high-order approximate compressors with more than five inputs have been proposed [1, 10, 29, 35]. Mahdiani et al. proposed a technique that removes the least significant partial products from the partial product matrix to decrease circuit area and delay, although it comes with some degree of error which can be adjusted by modifying the truncation level [28].

The existing approximate methods, which have been mainly designed to reduce energy consumption and area utilization in ASIC implementations, may not be as effective in FPGAs. This is because FPGA reconfigurable logic fabrics are typically based on fixed-size look-up tables (LUTs). Modern AMD-Xilinx and Intel FPGAs have hardwired DSP multipliers that offer faster and more energy-efficient multiplication than soft implementation on general-purpose LUT fabrics. The DSP-based multipliers are valuable resources that are situated in specific locations, which can result in long routing delays. On the other hand, LUTs are spread out across the chip, making them more easily routable. Moreover, there is a limitation on the number of DSP-based multipliers available. One solution is to deploy approximation methods to enhance the efficiency of LUT-based multiplication, in terms of speed, energy/power consumption, and hardware utilization. Ullah et al. [36, 37] implemented an optimization technique by truncating the least significant partial product of a 4×2 multiplier to reduce LUT utilization. Van Toan et al. [40] designed compact 3-2 and 4-2 compressors for use in different approximate multipliers with varying levels of accuracy. Kumm et al. [22] proposed dynamically reconfigurable FIR filters in Xilinx FPGAs using configurable look-up tables (CFGLUT5s).

This paper introduces DyRecMul, a cost-effective dynamically reconfigurable approximate multiplier for FPGAs. It is optimized for machine learning computation, has a short critical path, and uses a small number of LUTs. DyRecMul utilizes AMD-Xilinx technology's reconfigurable LUT primitives to approximate multiplication

Just Accepted

without significant accuracy degradation, even in post-training inference. To preserve dynamic range, DyRecMul utilizes a cost-effective encoder that transforms a fixed-point operand into an 8-bit floating-point format, and a decoder to revert the result back to fixed-point. The paper presents the design details and evaluation results of an INT8 version of DyRecMul since INT8 is a popular datatype in cost-effective machine learning computing. The multiplier introduces negligible accuracy loss while reducing significantly the number of required LUTs in FPGAs compared to the standard exact multiplier. Additionally, DyRecMul boasts very low latency, which allows for a faster clock frequency than that of typical AMD-Xilinx multiplier cores. The key contributions of this paper are summarized below.

- Utilization of dynamically reconfigurable LUTs (CFGLUT5s) in AMD-Xilinx FPGAs to make a low-cost and fast short-bitwidth multiplier.
- Internal conversion of fixed-point to a floating-point format to preserve dynamic range that is beneficial for machine learning and deep learning (DL) applications.
- For an INT8 case study, illustration of the design detail of a highly optimized encoder circuit to convert INT8 to 8-bit floating point format and a low-cost decoder to convert 8-bit floating point format to INT8.

The paper is organized as follows. Section 2 presents the analytical foundation of the proposed approach and illustrates the microarchitectural design detail of DyRecMul for the INT8 case study. Section 3 discusses generalized DyRecMul design considerations and guidelines, along with presenting hardware cost estimation functions. In Section 4, the reconfiguration time issue is discussed, and a rapidly reconfigurable variant of DyRecMul is introduced. Section 5 highlights the target applications where DyRecMul demonstrates its utility and efficacy, and Section 6 presents evaluation results, including error analysis, hardware evaluations, and accuracy measurements for DL applications. Finally, Section 7 concludes the paper. The source code of this work is available on https://github.com/INRS-ECCoLe/DyRecMul.

2 PROPOSED APPROXIMATE MULTIPLIER

This section provides an overview of the analytical basis of DyRecMul, followed by a detailed description of an INT8 DyRecMul design which also provides insight into the design decisions and considerations behind them.

2.1 Analytical Description

A precise N-bit signed integer multiplication of two operands, X and W, can be represented by:

$$Z = s_z.\sum_{i=0}^{2N-2} z_i.2^i = s_x.\sum_{i=0}^{N-2} x_i.2^i \times s_w.\sum_{i=0}^{N-2} w_i.2^i \tag{1}$$

where x_i, w_i and z_i denote the i^{th} bit of operand X and W and the result Z, respectively. S_x takes the value of -1 when X is negative, and 0 otherwise. The same logic applies to S_z and S_y. In INT8 representation, $N = 8$. When using multipliers in a computing system that only supports a single datatype, the output Z must be expressed in the same format as the input operands. This can be done using techniques like truncation.

The proposed approach utilizes an encoder to convert the first operand, X, to a floating-point representation, $\hat{X}_{float}$, of format:

$$\begin{gathered} float\left(sign_{BW}, exp_{BW}, mnt_{BW}\right), \\ \text{sign}_{BW} \in \{0,1\},\ exp_{BW} > 0,\ mnt_{BW} < N-1 \end{gathered} \tag{2}$$

where $sign_{BW}$, exp_{BW}, and mnt_{BW} denote the bitwidth of the sign, exponent, and mantissa elements, respectively. In this section, centered on signed multiplication, a single bit is designated to represent the sign, i.e., $sign_{BW} = 1$.

Just Accepted

To enable covering the entire dynamic range of X, exp_{BW} and mnt_{BW} must fulfill the following:

$$2^{exp_{BW}} + mnt_{BW} > N - 1. \tag{3}$$

The mantissa is a segment of mnt_{BW} bits from $|X|$, with the leftmost bit in $|X|$ that contains '1' being the most significant bit. Since the mantissa is shorter than the magnitude bits of X, this conversion involves an approximation. The conversion can be expressed as:

$$\hat{X}_{float} = \left[\hat{X}_{sign},\ \hat{X}_{exp}\ ,\ \hat{X}_{mnt}\ \right] \tag{4}$$

$$\begin{cases} \hat{X}_{sign} = x_{N-1}, \\ \hat{X}_{exp} = N - i\ where \begin{cases} 2^i < |X| \leq 2^{i-1},\ X \geq 2^{mnt_{BW}} \\ 0,\ X < 2^{mnt_{BW}}, \end{cases} \\ \hat{X}_{mnt} = Round\ (\frac{|X|}{2^{exp}}) \end{cases}$$

The mantissa $\hat{X}_{mnt}$ is then multiplied by $|W|$ to generate the mantissa of the result. To keep the result in the format of Eq. 2, the product is quantized to mnt_{BW} bits using:

$$\hat{Z}_{mnt} = Q\left(\hat{X}_{mnt} \times |W|,\ mnt_{BW}\right), \tag{5}$$

where Q is the quantization function. The result must be converted back to integer format. For this purpose, the $(N-1)$-bit absolute value of Z is first calculated by:

$$|Z| = 2^{\hat{X}_{exp}} \times \hat{Z}_{mnt} \tag{6}$$

The result, Z, in integer format is obtained by applying a two's complement function when Z is negative. The sign of Z is determined by XORing the sign bits of X and W.

$$Z = S_z.\,|Z|, \begin{cases} S_z = +1,\ x_{N-1} \bigoplus w_{N-1} = 0 \\ S_z = -1,\ x_{N-1} \bigoplus w_{N-1} = 1 \end{cases} \tag{7}$$

2.2 DyRecMul for INT8 Multiplication: Architecture and Components

INT8 quantization is a supported feature in machine learning frameworks such as TensorFlow Lite and PyTorch, as well as hardware toolchains like AMD-Xilinx DNNDK. Moreover, INT8 is a popular datatype in machine learning hardware accelerators and ML-optimized GPUs designed for embedded and edge applications [24], [20]. The INT8 DyRecMul described in this section can be used in place of INT8 multipliers for pre- or post-training inference. This multiplier is intended primarily for single-datatype INT8 architectures, meaning that it calculates $Z = X \times W$, where X, W and Z are all INT8. Fig. 1 depicts its architecture, which consists of four main components: (1) a cost-effective INT8 to floating-point encoder; (2) an ultra-low-cost mantissa multiplier using dynamically reconfigurable LUTs; (3) a floating-point to INT8 decoder; (4) a two's complement logic. The following subsections describe each component in detail.

2.2.1 Integer to Floating-Point Encoder. The proposed architecture converts its first operand, X, from INT8 to $float\,(1, 2, 5)$ representation, where 1, 2, and 5 indicate the number of allocated bits to the sign, exponent, and mantissa, respectively. This conversion corresponds to Eq. 4 in Section 2.1. The mantissas are multiplied and the exponents are added, and this conversion limits the binary multiplication to five bits while maintaining the dynamic range, which is crucial for accurate DL computations. As will be discussed later, DyRecMul deploys a CGFLUT5-based unsigned multiplier in which one operand must be five bits wide to achieve optimal efficiency.

Just Accepted

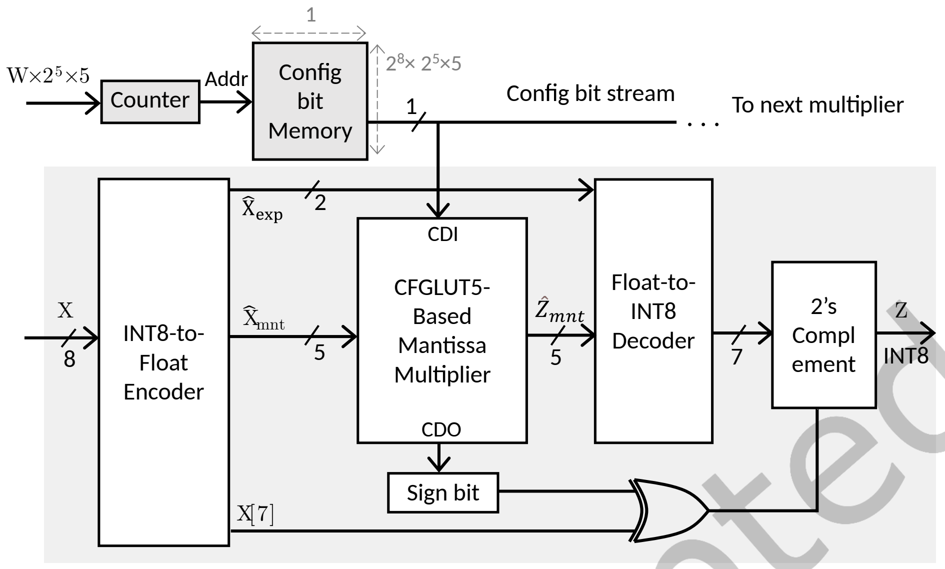


Fig. 1. DyRecMul architecture for INT8 multiplication of $Z = X \times W$, using $float\,(1, 2, 5)$ internal floating-point format.

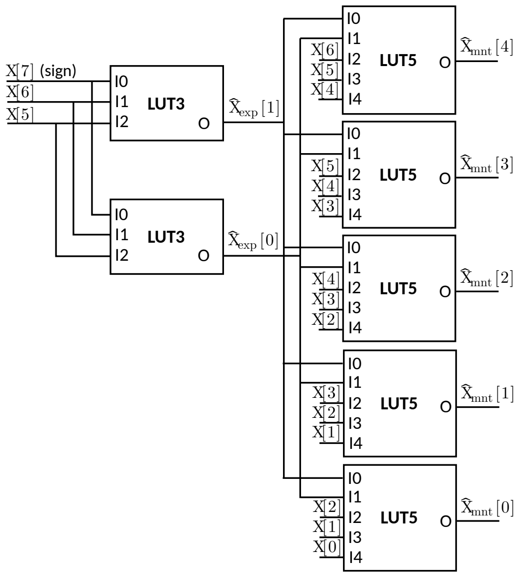
Just Accepted

When this multiplier is used in a weight stationary DL accelerator, input feature maps are fed as the first operand, X, to the multipliers. Without a floating-point conversion, the feature maps would need to be directly quantized to five bits. That would significantly limit the supported dynamic range, which could cause important accuracy loss for small activation values and ultimately lead to a prohibitive level of inaccuracy in ML inference. More precisely, such a quantization would limit the range of supported values to $[-2^5, 2^5 - 1]$, while $float\,(1, 2, 5)$ expands the range to $[-2^3 \times 2^5, 2^3 \times 2^5 - 1]$ and demands the same unsigned multiplier size for mantissa multiplication. The experimental results in Section 6.3 demonstrate that this conversion greatly helps in maintaining the precision of DL calculations.

The conversion logic from INT8 to $float\,(1, 2, 5)$ is designed to be low-cost and efficient, requiring only seven LUT5 elements. Fig. 2 depicts the truth tables and the corresponding LUT5 allocations for this encoder. The 2-bit exponent is acquired from the sign bit and the two most significant bits, while each mantissa bit is obtained from the exponent and three corresponding input bits. The encoder's critical path consists of two LUT5 units and their corresponding routing circuits.

2.2.2 Dynamically Reconfigurable Mantissa Multiplier. This component implements mantissa multiplication according to Eq. 5. The mantissa multiplier which serves as the core component of DyRecMul, is a cost-effective unsigned integer multiplier based on AMD-Xilinx CFGLUT5 primitives. CFGLUT5 offers a distinctive capability allowing the logical function of the LUT5 to be altered during circuit operation. This reconfiguration is achieved by programming new configuration bits into the LUT5, enabling it to adopt a new logic function with a maximum of 5-bit inputs and one output. The design subsystem controls the programming of new configuration bits, and this reconfiguration process does not necessitate external partial or complete reprogramming of the FPGA, ensuring uninterrupted operation. The programming of new configurations is carried out serially through a single-bit CDI input port, while CDI/CDO ports facilitate the cascading of multiple CFGLUT5 units in a chain.

In the mantissa multiplier, the logic circuit for multiplication with the second operand is programmed into CFGLUT5 units. Whenever the second operand, W, undergoes changes, the CFGLUT5 units within the mantissa



$\mathbf{X}$	$\hat{\mathbf{X}}_{\mathbf{exp}}$
[7:5]	**[1:0]**
000	00
001	01
01x	10
10x	10
101	01
111	00

$\hat{\mathbf{X}}_{\mathbf{exp}}$	$\mathbf{X}$	$\hat{\mathbf{X}}_{mnt}$
[1:0]	**[6:4]**	**[4]**
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

$\hat{\mathbf{X}}_{\mathbf{exp}}$	$\mathbf{X}$	$\hat{\mathbf{X}}_{mnt}$
[1:0]	**[5:3]**	**[3]**
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

$\hat{\mathbf{X}}_{\mathbf{exp}}$	$\mathbf{X}$	$\hat{\mathbf{X}}_{mnt}$
[1:0]	**[4:2]**	**[2]**
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

$\hat{\mathbf{X}}_{\mathbf{exp}}$	$\mathbf{X}$	$\hat{\mathbf{X}}_{mnt}$
[1:0]	**[3:1]**	**[1]**
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

$\hat{\mathbf{X}}_{\mathbf{exp}}$	$\mathbf{X}$	$\hat{\mathbf{X}}_{mnt}$
[1:0]	**[2:0]**	**[0]**
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

Fig. 2. INT8 to $float\,(1,2,5)$ encoder: LUT mapping and corresponding truth tables.

multiplier are reconfigured to establish a constant multiplication logic with the updated value. The first operand is fed into the 5-bit input port of CFGLUT5s. Hardcoding the multiplication logic for the second operand directly into the LUTs, instead of employing a traditional double-operand multiplier, significantly reduces the hardware cost of the multiplication circuit.

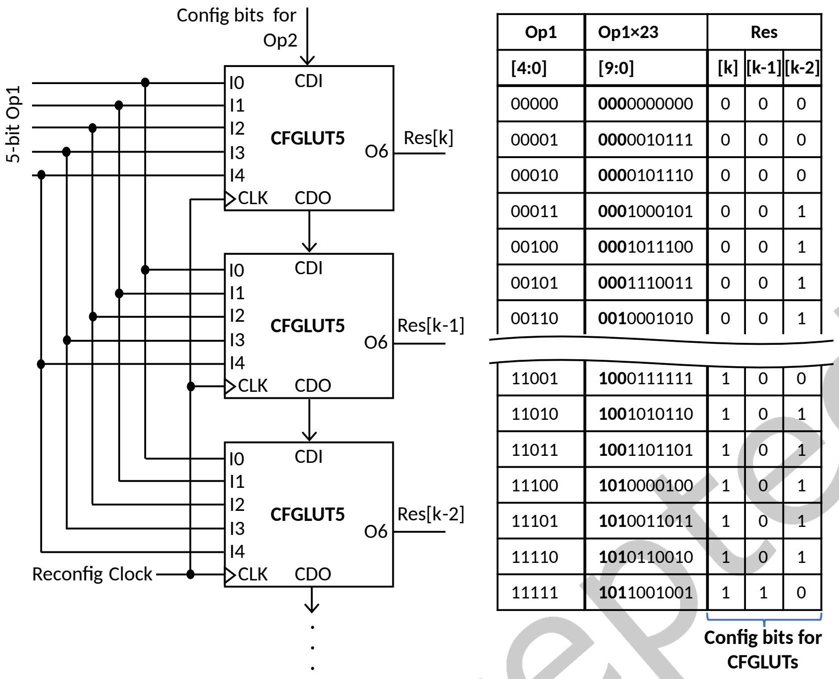
Fig. 3 illustrates the LUT mapping and configuration bits for an example 5-bit unsigned multiplier. The output consists of the quantized k most significant bits of the result. This means that only k CFGLUT5s are needed, each generating one output bit. In this example, a value of 23 is assumed for the second operand, W, configuring the CFGLUT5s to realize the logic circuit of a constant multiplier calculating the product of a 5-bit quantized first operand, and 23. Fig. 3 depicts the process of calculating the configuration bits for three CFGLUT5s which produce the three most significant bits of the product. Rounding is used in these calculations to minimize quantization error. The configuration bits are loaded into the CFGLUT5s serially through a cascaded CDI and CDO chain. As the bitwidth of the first operand surpasses five, the number of CFGLUT5s required for each output bit grows exponentially. More precisely, the required number of CFGLUT5s is:

$$\#CFGLUT5 = 2^{b_1-5} \times k \tag{8}$$

where b_1 and k denote the bitwidth of the first operand, X, and the result, Z, respectively.

As an example, implementing an exact 8×8 unsigned multiplier requires $2^3 \times 16 = 128$ CFGLUT5s. Although the LUT utilization may appear to be high, a more serious obstacle lies in the significant number of reconfiguration bits - a total of 4096 (128×2^5). In order to fully exploit the CFGLUT5s while minimizing their quantity, we limited the first operand, to five bits, resulting in one CFGLUT5 utilization per result bit. Thus, using k parallel CFGLUT5s allows for calculating the k-bit result of a 5-bit multiplication. If the bitwidth is shorter, some CFGLUT5s may remain partially unused, while increasing it to over five bits will exponentially increase the number of CFGLUT5s. Additionally, for the INT8 DyRecMul, we set the result bitwidth k to five, restricting the number of CFGLUT5s to only five units and the number of reconfiguration bits to 160. When this multiplier is used in a weight stationary

Just Accepted



Op1	Op1×23	Res		
[4:0]	[9:0]	[k]	[k-1]	[k-2]
00000	**0000**000000	0	0	0
00001	**0000**010111	0	0	0
00010	**0000**101110	0	0	0
00011	**0001**000101	0	0	1
00100	**0001**011100	0	0	1
00101	**0001**110011	0	0	1
00110	**0010**001010	0	0	1
11001	**1000**111111	1	0	0
11010	**1001**010110	1	0	1
11011	**1001**101101	1	0	1
11100	**1010**000100	1	0	1
11101	**1010**011011	1	0	1
11110	**1010**110010	1	0	1
11111	**1011**001001	1	1	0

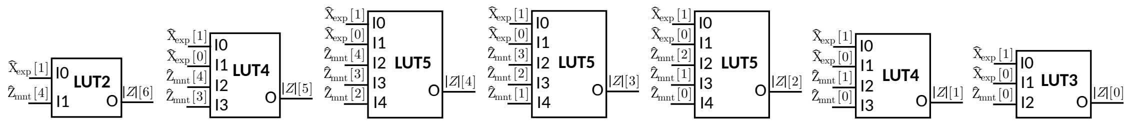
Fig. 3. 5-bit dynamically reconfigurable multiplier with 3-bit result using CFGLUT5 primitives. The second operand, $Op2 = 23$ in this example.

Just Accepted

DL accelerator, weights are translated into the configuration bits in CFGLUT5s, and input feature maps are fed as operands to the multipliers. As there is no interdependence between CFGLUT5 elements, they can operate entirely in parallel, resulting in a short critical path of just one CFGLUT5.

The configuration bits for CFGLUT5s are stored in a shared BRAM-based memory, as illustrated in Fig. 1. This memory functions like a read-only memory, storing all configuration bits for every possible 2^N value of the second operand. A counter generates the read address for the configuration memory during reconfiguration. When the second operand, W, changes, this counter is initialized to the address of the first configuration bit corresponding to the new W. The counter then increments with each clock cycle, and during each cycle, it programs one bit into the chain of CFGLUT5s inside the mantissa multiplier through CDI/CDO pins. In the case of the INT8 DyRecMul, this process continues for 160 clock cycles to write all 160 new configuration bits.

When multiple DyRecMuls are utilized in a design, the reconfiguration BRAM can be dual-ported to facilitate the parallel reprogramming of two DyRecMuls. Furthermore, a single reconfiguration memory can be employed for the sequential reprogramming of more than two DyRecMuls, albeit at the expense of a longer reconfiguration time. The reconfiguration time is an important factor in the efficiency of DyRecMul. As previously mentioned, DyRecMul is most suitable for applications where one of the operands does not change frequently. In other words, DyRecMac gains an edge when one of the operands remains unchanged significantly longer than the reconfiguration time. The influence of reconfiguration time on the overall processing time primarily relies on the dataflow and architecture of the DL accelerator. Section 5 will review some of the main target applications that can effectively take advantage of DyRecMul.



$\hat{X}_{exp}$	$\hat{Z}_{mnt}$	$\lvert Z \rvert$
[1]	**[4]**	**[6]**
0	x	0
1	0	0
1	1	1

$\hat{X}_{exp}$	$\hat{Z}_{mnt}$	$\lvert Z \rvert$
[1:0]	**[4:3]**	**[5]**
00	xx	0
01	0x	0
01	1x	1
1x	x0	0
1x	x1	1

$\hat{X}_{exp}$	$\hat{Z}_{mnt}$	$\lvert Z \rvert$
[1:0]	**[4:2]**	**[4]**
00	0xx	0
00	1xx	1
01	x0x	0
01	x1x	1
1x	xx0	0
1x	xx1	1

$\hat{X}_{exp}$	$\hat{Z}_{mnt}$	$\lvert Z \rvert$
[1:0]	**[3:1]**	**[3]**
00	0xx	0
00	1xx	1
01	x0x	0
01	x1x	1
1x	xx0	0
1x	xx1	1

$\hat{X}_{exp}$	$\hat{Z}_{mnt}$	$\lvert Z \rvert$
[1:0]	**[2:0]**	**[2]**
00	0xx	0
00	1xx	1
01	x0x	0
01	x1x	1
1x	xx0	0
1x	xx1	1

$\hat{X}_{exp}$	$\hat{Z}_{mnt}$	$\lvert Z \rvert$
[1:0]	**[1:0]**	**[1]**
00	0x	0
00	1x	1
01	x0	0
01	x1	1
1x	xx	0

$\hat{X}_{exp}$	$\hat{Z}_{mnt}$	$\lvert Z \rvert$
[1:0]	**[0]**	**[0]**
00	0	0
00	1	1
01	x	0
1x	x	0

Fig. 4. $float\ (1, 2, 5)$ to INT8 decoder: LUT mapping and corresponding truth tables.

2.2.3 Floating-point to Integer Decoder. The decoder converts the result of mantissa multiplication along with the exponent of the first operand into a 7-bit unsigned format. The decoder needs seven LUT5 units, with each unit producing one output bit using two exponent bits and a maximum of three mantissa bits. The truth table of each output bit is shown in Fig. 4. Since all LUT5 units function in parallel, the critical path is only one LUT5.

2.2.4 Two's Complement Logic. The output of the decoder is the absolute value of the result. As described in Section 2.1, the sign of the product can be obtained by exclusive OR operation between the sign bit of the two operands. If the product is negative, a two's complement function must be applied. Two's complement logic can be realized using eight LUT5 units. The carry signal is expected to be a major source of latency in this multiplier. To create an unsigned version of DyRecMul, we only need to remove the two's complement stage, modify the floating-point format to $float\ (0, 2, 5)$, and make slight adjustments to the encoder and decoder stages. The results in section 6 indicate that the unsigned version is more efficient than the signed version. This is because the two's complement stage, which consumes a significant amount of hardware resources, is present only in the signed DyRecMul.

3 GENERALIZED DYRECMUL

Section 2 detailed the design of an INT8 multiplier with internal $float\ (1, 2, 5)$ conversion. A crucial question arises regarding the effectiveness of DyRecMul with other data types and floating-point formats. The cost and accuracy of a DyRecMul multiplier depend on four key parameters: mnt_{BW}, the mantissa bitwidth in the floating-point encoding of *Op1*; exp_{BW}, the exponent bitwidth; $m\hat{X}_{exp} = max\left(\hat{X}_{exp}\right) + 1$, the maximum value that the exponent may attain plus one; and $\hat{Z}_{BW}$, the number of mantissa product bits generated by the mantissa multiplier unit. Altering these parameters results in different trade-offs between hardware cost and accuracy. The main task in designing a DyRecMul multiplier entails selecting suitable values for these parameters. This section provides an estimation of the hardware cost and accuracy of an $N \times N$ DyRecMul as a function of these four parameters. These estimation functions provide insight into the overall efficiency of the DyRecMul method and facilitate the process of designing DyRecMul multipliers for various data types.

3.1 Hardware Cost Estimation

In this section, we provide a conservative estimation of the hardware cost for three crucial components of unsigned DyRecMul—namely, the encoder, mantissa multiplier, and decoder—measured in terms of the number of utilized LUT5 units. To establish a uniform metric, we estimate the number of LUT5s, striving to map all

Just Accepted

logic onto LUT5 resources, excluding other elements such as F7, F8, and F9 MUX primitives. Since each LUT6 in modern AMD-Xilinx technology consists of two LUT5 units, the estimations provided in this section can be converted to LUT6 utilization by dividing the estimated number of LUT5s by two. Moreover, the estimations can be readily extended to the signed DyRecMul by applying minor modifications and incorporating the LUT utilization for the two's complement component.

3.1.1 Integer to Floating-Point Encoder. As illustrated in Fig. 2 for the INT8 case, the integer to floating-point encoder comprises two components: (1) an *exponent identifier* circuit, and (2) *mantissa extractor*.

The *exponent identifier* examines the $m\hat{X}_{exp}$ most significant bits to determine the exp_{BW} bits of the exponent. This is realized by a circuit that bears similarities to a $m\hat{X}_{exp}$-to-exp_{BW} priority encoder. As long as $m\hat{X}_{exp} \leq 5$, each output bit can be generated by a single LUT5. However, when $m\hat{X}_{exp}$ exceeds 5, a conservative and simplified estimation approach involves dividing the exponent identifier into two steps. The first step employs a logic circuit to convert the $m\hat{X}_{exp} - 1$ most significant bits, retaining only the most significant '1' bit and masking other less significant bits to zero. This step requires almost $m\hat{X}_{exp} - 1$ LUTs. The second step resembles a binary decoder with exp_{BW} output bits. In decoders, each output bit typically depends on a maximum of $2^{exp_{BW}}/2$ input bits. One LUT5 is adequate to process a maximum of five input bits in this stage. Since exp_{BW} is not supposed to be too large in DyRecMul, we can estimate that $\lceil 2^{exp_{\mathrm{BW}}}/10 \rceil$ LUT5s are required for each output bit. In summary, the estimated total number of LUT5s for the exponent identifier can be expressed as:

$$\begin{cases} exp_{LUT5} = exp_{BW}, \quad when\ m\hat{X}_{exp} \leq 5. \\ exp_{LUT5} = \lceil 2^{exp_{BW}}/10 \rceil \times exp_{BW} + m\hat{X}_{exp} - 1, \quad when\ m\hat{X}_{exp} > 5. \end{cases} \tag{9}$$

Recalling that $m\hat{X}_{exp} = max\left(\hat{X}_{exp}\right) + 1$, the exponent bitwidth, exp_{BW}, is directly derived from $m\hat{X}_{exp}$ as follows:

$$exp_{BW} = \left\lceil \log_2 \left(m\hat{X}_{exp} \right) \right\rceil. \tag{10}$$

The *mantissa extractor* functions as a multiplexer, with the exponent acting as its selector and $m\hat{X}_{exp}$ bits of the first operand as its input. Each bit of the mantissa is derived from the exp_{BW}-bit exponent and $m\hat{X}_{exp}$ bits of the input, X. As long as $m\hat{X}_{exp} + exp_{BW} \leq 5$, each output bit can be generated by one LUT5. However, if $m\hat{X}_{exp} + exp_{BW}$ exceeds 5, a multilayer tree of LUTs implements this multiplexer. Every 4×1 multiplexer can be implemented within a single LUT6 or two LUT5. Thus, we conservatively estimate the number of LUT5 in the first layer as $\lceil m\hat{X}_{exp}/4 \rceil \times 2$. Every four LUTs from the first layer, along with two exponent bits, feed one LUT6 in the second layer. Therefore, we estimate the number of LUT5s in the second layer as $\lceil m\hat{X}_{exp}/16 \rceil \times 2$. A third layer will be needed only when $m\hat{X}_{exp}$ exceeds 16 and will need more than one LUT when it exceeds 64, which is too large to consider. Hence, for simplicity, we discard the third layer here. Overall, the obtained estimation for LUT5 utilization in the mantissa extractor will be as follows:

$$\begin{cases} mnt_{LUT5} = mnt_{BW}, \quad when\ m\hat{X}_{exp} < 4. \\ mnt_{LUT5} = mnt_{BW} \times 2, \quad when\ m\hat{X}_{exp} = 4. \\ mnt_{LUT5} = \left(\lceil m\hat{X}_{exp}/4 \rceil + \lceil m\hat{X}_{exp}/16 \rceil \right) \times 2 \times mnt_{BW}, \quad when\ m\hat{X}_{exp} > 4. \end{cases} \tag{11}$$

The total estimated number of LUT5 units required for integer to floating-point encoding is:

$$encoder_{LUT5} = exp_{LUT5} + mnt_{LUT5} \tag{12}$$

Just Accepted

3.1.2 Floating-Point to Integer Decoder. The decoder converts the product result to integer format. Similar to the mantissa extractor in the encoder, the decoder operates as a multiplexer, with each output bit chosen from one of the $m\hat{X}_{exp}$ mantissa product bits, $\hat{Z}_{exp}$. Consequently, each output bit of the decoder requires a multiplexer, employing the exponent as the selector to choose among $m\hat{X}_{exp}$ bits of the mantissa product. As long as $m\hat{X}_{exp} + exp_{BW} \leq 5$, a single LUT5 is sufficient for implementing the multiplexer of each bit. If it exceeds 5, a similar approximation to Eq. 11 is applied. The resulting estimation function for the total number of LUT5s for the decoder is as follows:

$$\begin{cases} decoder_{LUT5} = N, \quad when\ m\hat{X}_{exp} < 4. \\ decoder_{LUT5} = N \times 2, \quad when\ m\hat{X}_{exp} = 4. \\ decoder_{LUT5} = \left(\lceil m\hat{X}exp/4 \rceil + \lceil m\hat{X}_{exp}/16 \rceil\right) \times N \times 2, \quad when\ m\hat{X}_{exp} > 4. \end{cases} \tag{13}$$

For the signed version, Eq. 13 needs to be adjusted by replacing N with $N - 1$.

3.1.3 CFGLUT5-Based Mantissa Multiplier. As detailed in Section 2.2.2, an estimate of the required number of LUT5 units for mantissa multiplication can be obtained as follows:

$$\begin{cases} mnt\_multiply_{LUT5} = \lceil \hat{Z}_{BW} \rceil, \quad when\ mnt_{BW} \leq 5. \\ mnt\_multiply_{LUT5} = \lceil 2^{mnt_{BW}-5} \times \hat{Z}_{BW} \rceil, \quad when\ mnt_{BW} > 5 \end{cases} \tag{14}$$

Increasing $\hat{Z}_{\text{BW}}$ enhances accuracy at the expense of a larger mantissa multiplication unit and an extended reconfiguration time.

As an example, for a UINT8 DyRecMul multiplier with $float\,(0, 2, 5)$ internal floating-point format, and the parameter values of $N = 8$, $exp_{BW} = 2$, $mnt_{BW} = 5$, $m\hat{X}_{exp} = 4$, and $\hat{Z}_{\text{BW}} = 5$, the estimated total number of LUT5s using Eq. 12, 13, and 14 is as follows:

$$total_{LUT5} = encoder_{LUT5} + decoder_{LUT5} + mnt\_multiply_{LUT5} = 12 + 16 + 5 = 33 \tag{15}$$

From Eq. 15, the estimated number of LUT6 units required for this UINT8 DyRecMul is approximately 17. It is worth noting that the synthesizer tools typically have the capability to reduce LUT utilization through optimization.

3.2 Accuracy Considerations

In addition to hardware costs, accuracy is another crucial factor to consider when determining DyRecMul parameter values. Specifically in DL applications, relative error serves as a crucial accuracy metric. This importance arises from the prevalence of small values in a significant portion of features and weights in DL computing. Consequently, a simple binary quantization may cause numerous features to be truncated to zero, leading to a substantial loss of accuracy. Hence, there is a growing tendency to use floating-point datatypes, which offer lower relative errors, in ML computation.

Relative error, RE_i when presenting value i is defined as follows:

$$RE_i = \frac{ED_i}{Exact_i}, \tag{16}$$

where

$$ED_i = |Exact_i - Approx_i|\,, i \in \mathbb{N}. \tag{17}$$

Just Accepted

Mean relative error (MRE) serves as an illustrative metric for measuring relative errors, and it is calculated as follows in unsigned multiplication:

$$MRE = \frac{1}{2^N} \sum_{i=0}^{2^N-1} \frac{ED_i}{Exact_i}, \tag{18}$$

where N denotes the input bitwidth. A floating-point representation with an mnt_{BW}-bit mantissa can always preserve the mnt_{BW} significant bits from the leftmost '1' (in positive numbers). In the worst case, where only the most significant mantissa bit is a '1', the maximum possible relative error is equal to $2^n/\left(2^{mnt_{BW}+n} + 2^n\right) = 1/2^{mnt_{BW}} + 1$. Conversely, in integer quantization, even quantizing a single least significant bit can lead to a worst-case relative error of 1.

Despite these advantages, a major drawback of floating-point representations is the excessive hardware cost of their adder/subtracter circuits, limiting their efficiency. DyRecMul addresses this issue by internally converting to floating point while allowing the subsequent add/subtract operations, typically following multiplication in DL, to be performed in integer format.

Two key factors significantly impact the relative error in DyRecMul: (1) the mantissa bitwidth, mnt_{BW}, and (2) the maximum exponent value, $m\hat{X}_{exp} - 1$. A longer mantissa allows for the preservation of more least significant bits, thereby improving resolution. Increasing mnt_{BW} reduces the worst-case relative error. On the other hand, the maximum exponent value, coupled with the mantissa bitwidth, defines the supported dynamic range of the floating-point representation. Specifically, when converting an N-bit integer to a $float\ (sign_{BW}, exp_{BW}, mnt_{BW})$ format, the floating-point representation can encompass maximum $m\hat{X}_{exp} + mnt_{BW} - 1$ bits of the input. If N is greater than $m\hat{X}_{exp} + mnt_{BW} - 1$, then the remaining $N - m\hat{X}_{exp} - mnt_{BW} + 1$ bits would be discarded.

The recommended approach to determine suitable parameter values for an $N \times N$ DyRecMul multiplier begins with identifying the two most crucial parameters: $m\hat{X}_{exp}$ and mnt_{BW}. As highlighted in the above discussions, there is a delicate balance between hardware cost and accuracy when selecting values for these parameters. The hardware cost estimation functions and MRE, described in Section 3, can be employed to determine the optimal values for these parameters in accordance with the design constraints. The exponent bitwidth, exp_{BW}, is then simply obtained using Eq. 10. The only remaining parameter is $\hat{Z}_{\mathrm{BW}}$, the bitwidth of the reconfigurable mantissa multiplier. This parameter significantly influences the hardware complexity of the mantissa multiplier, decoder, and, most importantly, the reconfiguration time. It is advisable to fine-tune this parameter after establishing other parameter values, using a simple exhaustive search. This process involves assessing hardware and reconfiguration costs, conducting an accuracy analysis of several options, and selecting the value that provides the optimal trade-off and meets the specified requirements.

4 RECONFIGURATION TIME AND RAPIDLY RECONFIGURABLE DYRECMUL

The reconfiguration time of CFGLUT5 units in the mantissa multiplier is a crucial parameter for the practicality of DyRecMul. As illustrated in Fig. 1, the reconfiguration bits are stored in a BRAM-based *Config bit Memory*, which functions as look-up tables. When CFGLUT5s need to be updated with a new W value, the new W serves as an address to read a total of $\#CFGLUT5 \times 2^5$ reconfiguration bits to be programmed into the CFGLUT5s, where $\#CFGLUT5$ has been calculated in Eq. 8. Since the programming is executed serially through a cascaded chain, the entire reconfiguration of the mantissa multiplier will take $\#CFGLUT5 \times 2^5$ clock cycles. In the case of the INT8 DyRecMul presented in Section 2.2, which utilizes 5 CFGLUT5s, this corresponds to $5 \times 2^5 = 160$ clock cycles. Throughout the reconfiguration time, the multiplier remains non-functional. Therefore, a standard DyRecMul architecture, as discussed thus far, is suitable for applications where the frequency of updating one of the operands is slow enough to ensure that the reconfiguration stall time does not introduce a significant latency overhead in overall processing time.

Just Accepted

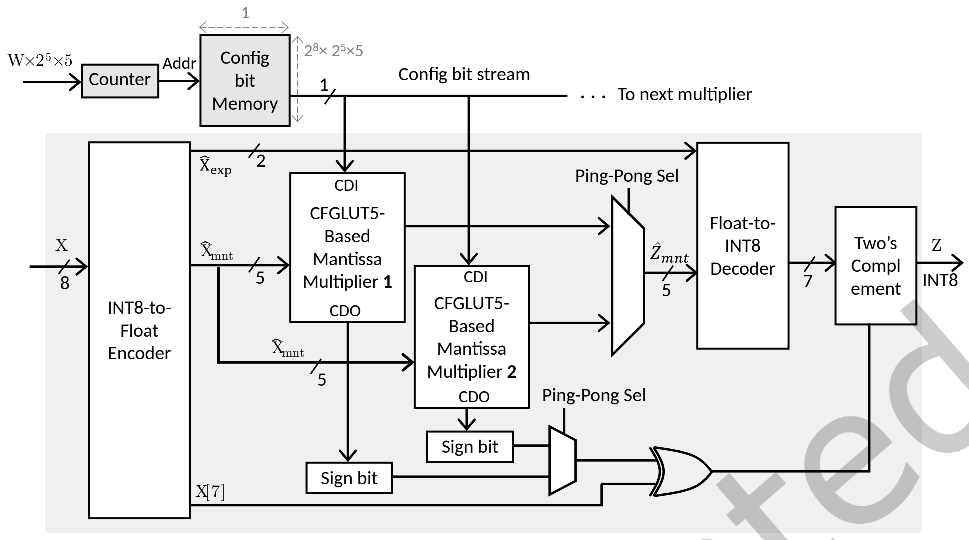


Fig. 5. Architecture of RR_DyRecMul utilizing a ping-pong scheme to reduce the reconfiguration latency.

To mitigate this limitation, this section introduces a Rapidly Reconfigurable version of DyRecMul, termed RR_DyRecMul. RR_DyRecMul employs a ping-pong scheme to conceal the reconfiguration time. As illustrated in Fig. 5, the RR_DyRecMul architecture incorporates a second CFGLUT5-based mantissa multiplier. A multiplexer is utilized to select the output of one of the mantissa multipliers at any given time. While the first mantissa multiplier is operational, the second one can be reconfigured with the next W value. The transition from the old W value to the new one can then occur by simply adjusting the multiplexer selector, and activating the second multiplier while putting the first one in reconfiguration mode. In RR_DyRecMul, if one of the operands remains unchanged for at least $\#CFGLUT5 \times 2^5$ clock cycles, the reconfiguration time overhead ideally becomes zero. The cost overhead comprises an additional mantissa multiplier, the LUT cost of which is estimated in Eq. 14, as well as a 2-to-1 multiplexer that would require a maximum of $\hat{Z}_{\mathrm{BW}}$ LUTs.

5 TARGET APPLICATIONS

As Section 6 will unveil, DyRecMul achieves a significant enhancement in efficiency, particularly with its innovative CFGLUT5-based mantissa multiplier. The reconfiguration time is a crucial factor that may introduce latency overhead, particularly when the hardcoded operand experiences frequent variations. Consequently, DyRecMul emerges as an effective solution for applications in which one operand remains relatively constant. This characteristic aligns well with various ML computations and signal and image processing applications, where the infrequency of changes in one operand is a common property. This section provides an overview of some key applications for the DyRecMul multiplier. The utilization of DyRecMul has the potential to significantly enhance the efficiency of FPGA-based computing in these applications.

5.1 Low-cost Hardware Accelerators for Updatable Supervised Learning

Hardware accelerators are necessary to yield the required inference throughput, latency, and energy efficiency when using supervised machine learning models for several applications. In an expanding array of applications across domains such as wireless communication and the inference of tiny ML models, where throughput is crucial, a fully parallelized accelerator architecture assigns a dedicated hardware multiplier to each multiplication.

Just Accepted

When model parameters remain unchanged, employing a constant multiplier can considerably enhance efficiency. However, occasional updating of ML parameter values is a crucial requirement for numerous supervised learning applications. Even in specific ML approaches, like federated learning and incremental learning, the periodic adjustment of model parameters is a fundamental feature. The support for such occasional parameter updates requires non-constant multipliers, which, in turn, incurs significant costs and energy overhead. DyRecMul serves as a solution that bridges the gap between constant and non-constant multipliers, delivering markedly enhanced efficiency compared to non-constant multipliers while accommodating occasional updates.

5.2 Deep Reinforcement Learning Hardware

Reinforcement learning (RL) methods encompass two distinct phases: exploration and exploitation. In the exploration phase, the agent actively seeks the optimal action selection policy for each state within the environment. This policy undergoes iterative refinement until a near-optimal solution is achieved. Subsequently, the RL transitions to the exploitation phase, during which the agent applies its learned policy to make decisions within the environment, aiming to maximize its cumulative reward. In deep reinforcement learning techniques like deep Q-learning, a deep neural network (DNN) is utilized to fully approximate quality values (Q-values) or state-action policies. Throughout the exploration phase, the DNN undergoes retraining, whereas, in the considerably longer exploitation phase, the DNN's parameter weights remain constant. DyRecMul is a fitting choice for such applications characterized by relatively short periods of coefficient changes and where constant multipliers suffice for the majority of operational time. DyRecMul efficiently addresses the implementation of high-performance DNNs by supporting weight updates during the exploration phase through reconfiguration. Afterward, it seamlessly operates as a cost-effective constant multiplier during the exploitation phase.

5.3 DL Hardware Accelerators With Weight-Stationary Dataflows

DL hardware accelerators are increasingly utilized for computing the inference of DL models. These accelerators are commonly designed to handle computations across various types and sizes of DL algorithms. Given that multiply-and-accumulate (MAC) operations form the predominant majority and bottleneck, these accelerators typically feature a core component consisting of a matrix of processing elements (PEs) each containing a MAC operator. Weights and features are provided to this PE matrix using a predefined dataflow, wherein matrix multiplications are typically carried out for various DL layers such as multilayer perceptron layers or convolutional layers. A commonly used dataflow in DL accelerators is the weight-stationary scheme, where weights are stored in local memories within the PEs, and input features are sequentially uploaded and circulated across PEs [19]. DyRecMul is an effective solution for weight-stationary dataflows, particularly when dealing with large-sized input features that necessitate infrequent updates of weights. In addition to offering reduced hardware cost and latency compared to standard multipliers, DyRecMul eliminates the need for internal registers to store weights. This is achieved by directly hardcoding and programming the weights into the reconfigurable mantissa multiplier. In scenarios where the input feature size is substantial, the number of clock cycles during which weights must persist on PEs is notably greater than the reconfiguration time calculated in Section 4. This long stability period for weights eliminates the need for reconfiguration time overhead in RR_DyRecMul, as it affords sufficient time to program the next weights into the second mantissa multiplier unit while the current weights are still in use. The transition to the new weights can then be seamlessly executed within a single cycle. Even with the baseline DyRecMul, a larger input feature size results in less frequent reconfiguration, leading to a correspondingly lower reconfiguration time overhead.

Just Accepted

5.4 Updatable Digital Filters

Beyond its applications in ML hardware, DyRecMul can be effectively utilized in digital filters for signal and image processing. This is particularly the case in scenarios where the filter coefficients remain constant but require occasional updates. DyRecMul achieves markedly higher efficiency by hardcoding coefficients into CFGLUT5s compared to non-constant multipliers, all while allowing for periodic coefficient updates. Alternative solutions for achieving updatable filters involve using non-constant multipliers with additional registers to store coefficients or resorting to constant multipliers, demanding the reprogramming of the entire FPGA when weights are updated. The former solution often results in diminished efficiency concerning hardware cost and latency, while the latter necessitates a service interruption for FPGA reprogramming. DyRecMul serves as a solution that bridges the gap between non-constant and constant multiplier methods, offering enhanced efficiency while allowing dynamic coefficient updates.

6 RESULTS

We evaluated INT8/UINT8 DyRecMul by conducting error analysis, measuring hardware efficiency, and running accuracy tests on popular benchmark DL models. This section presents the results, discussions, and comparisons.

6.1 Error Analysis and Evaluation

One important method for assessing the accuracy of approximate calculations is error measurements and analysis. These assessments gauge the difference between the results of approximate calculations to those of exact calculations to determine any discrepancies [41]. This section presents the accuracy of DyRecMul with five error metrics, including MRE (defined in Eq. 18), Error Probability (EP), Mean Absolute Error (MAE), Mean Squared Error (MSE), and Normalized Error Distance (NED). For unsigned arithmetic, these metrics are defined as follows:

$$EP = \frac{1}{2^N} \sum_{i=0}^{2^N-1} ED_i \neq 0, \tag{19}$$

$$MAE = \frac{1}{2^N} \sum_{i=0}^{2^N-1} ED_i, \tag{20}$$

$$MSE = \frac{1}{2^N} \sum_{i=0}^{2^N-1} (ED_i)^2, \tag{21}$$

$$NED = \frac{1}{2^N} \sum_{i=0}^{2^N-1} \frac{ED_i}{max(ED)}, \tag{22}$$

where N denotes the input bitwidth and ED_i has been defined in Eq. 17.

Table 1 compares DyRecMul with existing approximated multipliers [5, 9, 11, 25, 26, 36, 37, 40, 42, 46]. Our primary focus lies on the signed version of the DyRecMul, tailored specifically for ML applications. Nonetheless, we present the unsigned version of the multiplier, which is suitable for error-tolerant applications within the domain of image processing, such as image sharpening and smoothing [17]. Our method outperforms [9] by a significant margin in error analysis. The approximate multipliers proposed by [40] and [26] offer reconfigurability based on an approximation factor P representing the number of encoders employed in the partial product generation. DyRecMul yields comparable accuracy compared to this method in the low approximation ($P = 8$) mode. In the mid-level approximation mode ($P = 10$), our method outperforms the other designs and proves its robustness and effectiveness.

Just Accepted

Table 1. Error Analysis and Comparisons

Multiplier	**EP**	**MAE**	**MRE**	**MSE**	**NED**
DyRecMul (signed)	0.5157	397	0.0680	96336	0.00005
Danopoulos [9]	0.7480	464	0.1259	5515991	0.01160
Liu [26] ($P = 8$)	-*	-	0.0525	-	-
Liu [26] ($P = 10$)	-	-	0.1991	-	-
Van Toan [40] ($P = 8$)	-	100	0.2299	-	-
Van Toan [40] ($P = 10$)	-	512	0.9320	-	-
DyRecMul (unsigned)	0.7380	336	0.0194	260528	0.1210
Ha [11]	-	3490	0.3676	-	-
Ullah [36] $Approx1$	-	-	0.016	-	-
Ullah [36] $Approx2$	-	-	0.030	-	-
Ullah [36] $Approx3$	-	-	0.027	-	-
Ullah [37] Ca	-	-	0.0029	-	-
Ullah [37] Cc	-	-	0.1293	-	-
Yang [46]	-	220	0.0196	-	-
Venkatachalam [42]	-	101	0.0548	-	-
Liu [25]	-	130	0.0062	-	-
Ansari [5]	-	1530	0.1336	-	-

* Not reported in the reference

Just Accepted

Although error analysis can be useful in evaluating the accuracy of approximated circuits, it may not give a complete picture. In Section 6.3, we will examine how the approximation in DyRecMul affects the inference accuracy of some benchmark convolutional neural network (CNN) models in order to gain a more comprehensive understanding.

6.2 Hardware Implementation Results

DyRecMul was modeled in VHDL, then synthesized and implemented with AMD-Xilinx Vivado ML 2022.2 for a xcku5p Kintex Ultrascale+ FPGA with a speed grade of -3. For comparisons, we also evaluated the default AMD-Xilinx multiplier circuit. To ensure fairness, the synthesis tool was instructed to avoid using DSP resources.

Table 2 presents the implementation costs and maximum supported clock frequency of INT8/UINT8 DyRecMul and the Xilinx standard multiplier cores in three configurations: signed multiplier, unsigned multiplier, and signed MAC. The metrics include the utilization of LUTs and CARRY8 primitives, as well as the maximum clock frequency that is supported. Table 2 also presents the results of four unsigned approximate multipliers from existing works. These data have been extracted from the reported implementation results on Xilinx Spartan-6 in [40], and normalized based on exact multiplier results. As all the compared designs are combinational multipliers, none of them utilizes flip-flops. Additionally, the utilization of other resources, such as BRAMs and DSPs, is also zero.

The results indicate significant reductions in LUT usage with 64%, 80%, and 67% savings for the signed, unsigned, and MAC cases, respectively, compared to the exact 8×8 multiplier. DyRecMul also archived a higher frequency than all existing exact and approximate multipliers. The ping-pong scheme facilitating rapid reconfiguration in RR_DyRecMul incurs an additional cost of ten LUTs, a still considerably more cost-effective option compared to standard multipliers.

Furthermore, from the discussions in Section 2, it can be inferred that the total size of this configuration bit memory in terms of the number of bits is:

$$config\_bit\_memory_{size(bits)} = \hat{Z}_{BW} \times 2^{N} \times 2^{mnt_{BW}}. \tag{23}$$

As each BRAM36K unit can be configured and used as a 32K by 1-bit wide memory, mapping the configuration bit memory onto BRAM36K elements, the total number of required BRAM36K units is calculated as follows:

$$config\_bit\_memory_{\#BRAMs} = \left\lceil \left(\hat{Z}_{BW} \times 2^{N} \times 2^{mnt_{BW}} \right) / 32768 \right\rceil. \tag{24}$$

From Eq. 24, we can determine that two BRAM36K units are required for the proposed INT8/UINT8 DyRecMul as $\lceil (5 \times 2^{8} \times 2^{5}) / 32768 \rceil = 2$.

In addition to incorporating 8-bit multipliers, we expanded our implementation to include 12-bit and 16-bit multipliers, utilizing distinct internal floating-point formats. This extension enabled us to experimentally assess the efficiency of DyRecMul when applied to larger datatypes. Table 3 presents the hardware implementation results for 12-bit and 16-bit multipliers, comparing DyRecMul, RR_DyRecMul, and AMD-Xilinx multiplier cores. For the INT12/UINT12 versions, we allocated three bits to the exponent bitwidth, $exp_{BW} = 3$, and five bits to the mantissa bitwidth, $mnt_{BW} = 5$, resulting in the format $float\,(1/0, 3, 5)$. Consequently, $m\hat{X}_{exp}$ equals eight and seven in these UINT12 and INT12 multipliers, respectively. In the case of INT16/UINT16 multipliers, we assigned four bits and five bits to the exponent and mantissa bitwidths, respectively, yielding the format $float\,(1/0, 4, 5)$. The findings highlight that the hardware efficiency superiority of DyRecMul becomes more pronounced with larger datatypes. Additionally, the results indicate that unsigned multipliers exhibit progressively lower hardware costs compared to their signed counterparts. This disparity is primarily attributed to the two's complement logic in signed DyRecMul, which incurs escalating costs as the input and output bitwidth, N, increases.

Despite considerable utilization of CARRY8 fast carry primitives in standard multipliers, DyRecMul achieves a significantly higher maximum clock frequency thanks to its optimized datapath. DyRecMul yields similar results in both signed multiplier and MAC setups. This is because the two's complement logic at the end of the architecture needs the same amount of LUTs as an adder/subtractor in MAC. In other words, an adder/subtractor can be incorporated into the two's complement LUT-based circuit with almost no cost overhead. Among the three tested setups, DyRecMul achieves the highest performance in the unsigned multiplication setup. This is because the two's complement logic in signed multiplication and the add/sub logic in MAC operations constitute a significant portion of both hardware and latency.

Just Accepted

Table 2. Implementation Costs and Performance Comparison for INT8 and UINT8 datatypes

Function	Multiplier	Size	#LUT	#CARRY8/4	Max Freq. (MHz)
Signed Multiplier	DyRecMul	8×8	25	0	770
	RR_DyRecMul	8×8	35	0	699
	AMD-Xilinx	8×8	69	8	730
	(Exact)	7×7	61	6	660
Signed MAC	DyRecMul	8×8	25	1	769
	AMD-Xilinx	8×8	76	10	571
	(Exact)	7×7	69	8	585
Unsigned Multiplier	DyRecMul	8×8	16	0	950
	RR_DyRecMul	8×8	26	0	847
	AMD-Xilinx	8×8	82	6	684
	(Exact)	7×7	55	6	725
	Venkatachalam [42] *	8×8	108	4	690
	Ha [11] *	8×8	73	4	664
	Ansari [5] *	8×8	63	3	757
	Yang [47] *	8×8	76	3	729
	Ullah [36] *Approx*1†	8×8	64	2	-
	Ullah [36] *Approx*2†	8×8	54	0	-
	Ullah [36] *Approx*3†	8×8	54	0	-
	Ullah [37] *Ca*†	8×8	67	7	424
	Ullah [37] *Cc*†	8×8	59	4	452
	Van Toan [40] $P = 8$*	8×8	59	4	759
	Van Toan [40] $P = 10$*	8×8	56	4	849

* Results are obtained from [40] and normalized based on the exact multiplier results.

† The target device was Virtex-7.

Just Accepted

6.3 Accuracy in Deep Learning Computation

To ensure the usability of DyRecMul in DNN accelerators, we evaluated its accuracy for the benchmark models ResNet18, ResNet50, VGG19, and DenseNet121 using the CIFAR-10 dataset. For these experiments, we used the AdaPT framework which provides a rapid emulation environment to measure the accuracy of new approximate multipliers in the CNNs, LSTMs, and GANs inferences [9]. Fig. 6 compares the inference accuracy and the hardware utilization costs offered by an INT8 DyRecMul with those of standard AMD-Xilinx multipliers with different bitwidths. The accuracy is measured for post-training deployment with no re-training applied. Fig. 6 clearly indicates that DyRecMul offers a significantly superior accuracy-hardware cost trade-off compared to the

Table 3. Implementation Costs and Performance Comparison for INT/UINT12 and INT/UINT16 Multipliers

Datatype	Multiplier	Size	#LUT	#CARRY8	Max Freq. (MHz)
INT12 *	DyRecMul	12×12	62	0	470
	RR_DyRecMul	12×12	70	0	425
	AMD-Xilinx (Exact)	12×12	168	18	525
UINT12 *	DyRecMul	12×12	36	0	615
	RR_DyRecMul	12×12	48	0	580
	AMD-Xilinx (Exact)	12×12	186	15	575
INT16 †	DyRecMul	16×16	104	0	355
	RR_DyRecMul	16×16	114	0	320
	AMD-Xilinx (Exact)	16×16	279	32	355
UINT16 †	DyRecMul	16×16	45	0	510
	RR_DyRecMul	16×16	56	0	475
	AMD-Xilinx (Exact)	16×16	337	28	400

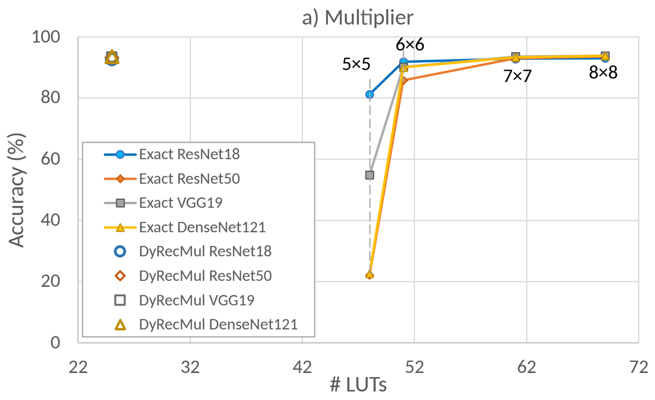
* For INT12/UINT12, $exp_{BW} = 3$ and $mnt_{BW} = 5$.

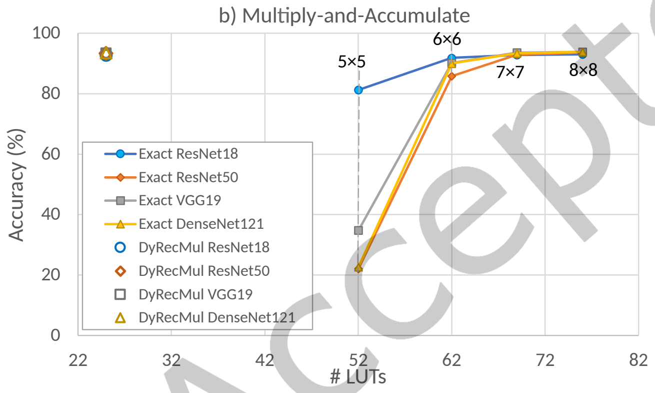
† For INT12/UINT16, $exp_{BW} = 4$ and $mnt_{BW} = 5$.

Table 4. Inference Accuracy and Hardware Metrics Comparison With Previous Studies

Multiplier	Dataset	Accuracy (%)				Area (LUTs)		Max Freq. (MHz)	
		ResNet18	ResNet50	VGG19	DenseNet121	MUL	MAC	MUL	MAC
Exact (5×5)	CIFAR-10	81.20	22.08	34.83	22.72	48	52	740	724
Exact (6×6)	CIFAR-10	91.83	85.81	90.17	89.98	51	62	770	700
Exact (7×7)	CIFAR-10	92.83	92.95	93.60	93.37	61	69	660	585
Exact (8×8)	CIFAR-10	92.98	93.57	93.78	93.85	69	76	730	571
[9] (8×8)	CIFAR-10	-	82.70	90.70	-	-	-	-	-
[12] SSM (8×8)	ImageNetV2	-	-	92.28	-	-	-	-	-
[12] DSM (8×8)	ImageNetV2	-	-	92.15	-	-	-	-	-
DyRecMul	CIFAR-10	92.72	93.32	93.45	93.54	25	25	770	769

Just Accepted





Just Accepted

Fig. 6. Trade-offs between hardware utilization and inference accuracy when running four benchmark CNNs using DyRecMul and standard exact multipliers of different sizes: (a) signed multiplier results, and (b) MAC results.

standard AMD-Xilinx multiplier cores. Table 4 provides detailed results of INT8 DyRecMul and selected previous works [12], including inference accuracy, maximum supported clock frequency and hardware utilization. Based on the findings, DyRecMul provides an average accuracy loss that is only 0.29% lower than that of 8×8 exact multipliers and 0.07% higher than 7×7. The worst-case accuracy distance from the exact 8×8 multiplier is only 0.33%. Table 4 also shows that DyRecMul offers slightly higher accuracy compared to Danopoulos [9], SSM, and DSM [12] INT8 approximate multipliers in the VGG19 test and significantly higher accuracy compared to Danopoulos in ResNet50. Also, as reported in Section 6.2, DyRecMul uses fewer LUTs, with a reduction of 64% and 67% in signed and MAC setups, respectively. Furthermore, it can support clock frequencies that are up to 5.5% and 34.6% higher than an exact 8×8 multiplier in signed and MAC setups, respectively.

7 CONCLUSION

This paper introduces DyRecMul, an approximate multiplier meticulously optimized for machine learning computations on FPGAs. Leveraging dynamically reconfigurable logic, this multiplier achieves remarkable hardware efficiency. Additionally, it employs a cost-effective internal floating-point conversion technique to preserve a wide dynamic range, thereby enhancing the precision of machine learning calculations. The results demonstrate that for INT8 precision, DyRecMul requires 64%, 80%, and 67% fewer LUTs compared to the Xilinx standard multiplier core in signed, unsigned, and MAC setups, respectively. Moreover, the maximum supported clock frequency remains notably higher than that of Xilinx multipliers. DyRecMul also provides a substantial advantage over four existing approximate multipliers in terms of both hardware utilization and frequency. Additionally, the results indicate that employing this multiplier for post-training DL inference leads to a minimal average accuracy degradation of less than 0.29% compared to exact INT8 multiplication.

8 ACKNOWLEDGMENTS

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Just Accepted

REFERENCES

[1] Mohammad Ahmadinejad, Mohammad Hossein Moaiyeri, and Farnaz Sabetzadeh. 2019. Energy and area efficient imprecise compressors for approximate multiplication at nanoscale. *AEU-International Journal of Electronics and Communications* 110 (2019), 152859.

[2] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2017. Dual-quality 4: 2 compressors for utilizing in dynamic accuracy configurable multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 4 (2017), 1352–1361.

[3] Mohammad Saeed Ansari, Bruce F. Cockburn, and Jie Han. 2019. A Hardware-Efficient Logarithmic Multiplier with Improved Accuracy. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 928–931. https://doi.org/10.23919/DATE.2019.8714868

[4] Mohammad Saeed Ansari, Bruce F Cockburn, and Jie Han. 2020. An improved logarithmic multiplier for energy-efficient neural computing. *IEEE Trans. Comput.* 70, 4 (2020), 614–625.

[5] Mohammad Saeed Ansari, Honglan Jiang, Bruce F Cockburn, and Jie Han. 2018. Low-power approximate multipliers using encoded partial products and approximate compressors. *IEEE journal on emerging and selected topics in circuits and systems* 8, 3 (2018), 404–416.

[6] Chuangtao Chen, Sen Yang, Weikang Qian, Mohsen Imani, Xunzhao Yin, and Cheng Zhuo. 2020. Optimally approximated and unbiased floating-point multiplier with runtime configurability. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 121, 9 pages. https://doi.org/10.1145/3400302.3415702

[7] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference* (Austin, Texas) *(DAC '13)*. Association for Computing Machinery, New York, NY, USA, Article 113, 9 pages. https://doi.org/10.1145/2463209.2488873

[8] Luigi Dadda. 1965. Some schemes for parallel multipliers. *Alta frequenza* 34 (1965), 349–356.

[9] Dimitrios Danopoulos, Georgios Zervakis, Kostas Siozios, Dimitrios Soudris, and Jörg Henkel. 2023. AdaPT: Fast Emulation of Approximate DNN Accelerators in PyTorch. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 6 (June 2023), 2074–2078. https://doi.org/10.1109/TCAD.2022.3212645

[10] Darjn Esposito, Antonio Giuseppe Maria Strollo, Ettore Napoli, Davide De Caro, and Nicola Petra. 2018. Approximate multipliers based on new approximate compressors. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 12 (2018), 4169–4182.

[11] Minho Ha and Sunggu Lee. 2018. Multipliers With Approximate 4–2 Compressors and Error Recovery Modules. *IEEE Embedded Systems Letters* 10, 1 (2018), 6–9. https://doi.org/10.1109/LES.2017.2746084

[12] Issam Hammad, Ling Li, Kamal El-Sankary, and W Martin Snelgrove. 2021. CNN inference using a preprocessing precision controller and approximate multipliers with various precisions. *IEEE Access* 9 (2021), 7220–7232.

[13] Soheil Hashemi, R. Iris Bahar, and Sherief Reda. 2015. DRUM: A Dynamic Range Unbiased Multiplier for Approximate Applications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (Austin, TX, USA) *(ICCAD '15)*. IEEE Press, 418–425.

[14] Mohsen Imani, Ricardo Garcia, Saransh Gupta, and Tajana Rosing. 2018. RMAC: Runtime Configurable Floating Point Multiplier for Approximate Computing. In *Proceedings of the International Symposium on Low Power Electronics and Design* (Seattle, WA, USA) *(ISLPED '18)*. Association for Computing Machinery, New York, NY, USA, Article 12, 6 pages. https://doi.org/10.1145/3218603.3218621

[15] Mohsen Imani, Daniel Peroni, and Tajana Rosing. 2017. CFPU: Configurable Floating Point Multiplier for Energy-Efficient Computing. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 76, 6 pages. https://doi.org/10.1145/3061639.3062210

[16] Mohsen Imani, Alice Sokolova, Ricardo Garcia, Andrew Huang, Fan Wu, Baris Aksanli, and Tajana Rosing. 2019. ApproxLP: Approximate Multiplication with Linearization and Iterative Error Control. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 159, 6 pages. https://doi.org/10.1145/3316781.3317774

[17] Honglan Jiang, Cong Liu, Fabrizio Lombardi, and Jie Han. 2018. Low-power approximate unsigned multipliers with configurable error recovery. *IEEE Transactions on Circuits and Systems I: Regular Papers* 66, 1 (2018), 189–202.

[18] Honglan Jiang, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu, and Jie Han. 2020. Approximate arithmetic circuits: A survey, characterization, and recent applications. *Proc. IEEE* 108, 12 (2020), 2108–2135.

[19] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[20] Sumin Kim, Gunju Park, and Youngmin Yi. 2021. Performance Evaluation of INT8 Quantized Inference on Mobile GPUs. *IEEE Access* 9 (2021), 164245–164255. https://doi.org/10.1109/ACCESS.2021.3133100

[21] Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. 2011. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In *2011 24th Internatioal Conference on VLSI Design*. IEEE, 346–351. https://doi.org/10.1109/VLSID.2011.51

[22] Martin Kumm, Konrad Möller, and Peter Zipf. 2013. Dynamically reconfigurable FIR filter architectures with fast reconfiguration. In *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, 1–8. https://doi.org/10.1109/ReCoSoC.2013.6581517

[23] Chia-Hao Lin and Ing-Chao Lin. 2013. High accuracy approximate multiplier with error correction. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 33–38. https://doi.org/10.1109/ICCD.2013.6657022

[24] Ye Lin, Yanyang Li, Tengbo Liu, Tong Xiao, Tongran Liu, and Jingbo Zhu. 2021. Towards fully 8-bit integer inference for the transformer model. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence* (Yokohama, Yokohama, Japan) *(IJCAI'20)*. Article 520, 7 pages.

[25] Cong Liu, Jie Han, and Fabrizio Lombardi. 2014. A low-power, high-performance approximate multiplier with configurable partial error recovery. In *Proceedings of the Conference on Design, Automation & Test in Europe* (Dresden, Germany) *(DATE '14)*. European Design and Automation Association, Leuven, BEL, Article 95, 4 pages.

[26] Weiqiang Liu, Liangyu Qian, Chenghua Wang, Honglan Jiang, Jie Han, and Fabrizio Lombardi. 2017. Design of approximate radix-4 booth multipliers for error-tolerant computing. *IEEE Transactions on computers* 66, 8 (2017), 1435–1441.

[27] Weiqiang Liu, Jiahua Xu, Danye Wang, Chenghua Wang, Paolo Montuschi, and Fabrizio Lombardi. 2018. Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 9 (2018), 2856–2868.

[28] Hamid Reza Mahdiani, Ali Ahmadi, Sied Mehdi Fakhraie, and Caro Lucas. 2009. Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 57, 4 (2009), 850–862.

[29] R Marimuthu, Y Elsie Rezinold, and Partha Sharathi Mallick. 2016. Design and analysis of multiplier using approximate 15-4 compressor. *IEEE Access* 5 (2016), 1027–1036.

[30] John N. Mitchell. 1962. Computer Multiplication and Division Using Binary Logarithms. *IRE Transactions on Electronic Computers* EC-11, 4 (Aug 1962), 512–517. https://doi.org/10.1109/TEC.1962.5219391

[31] Amir Momeni, Jie Han, Paolo Montuschi, and Fabrizio Lombardi. 2014. Design and analysis of approximate compressors for multiplication. *IEEE Trans. Comput.* 64, 4 (2014), 984–994.

[32] Srinivasan Narayanamoorthy, Hadi Asghari Moghaddam, Zhenhong Liu, Taejoon Park, and Nam Sung Kim. 2014. Energy-efficient approximate multiplication for digital signal processing and classification applications. *IEEE transactions on very large scale integration (VLSI) systems* 23, 6 (2014), 1180–1184.

[33] Hassaan Saadat, Haseeb Bokhari, and Sri Parameswaran. 2018. Minimally biased multipliers for approximate integer and floating-point multiplication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2623–2635.

[34] Antonio Giuseppe Maria Strollo, Ettore Napoli, Davide De Caro, Nicola Petra, and Gennaro Di Meo. 2020. Comparison and extension of approximate 4-2 compressors for low-power approximate multipliers. *IEEE Transactions on Circuits and Systems I: Regular Papers* 67, 9 (2020), 3021–3034.

[35] Che-Wei Tung and Shih-Hsu Huang. 2019. Low-Power High-Accuracy Approximate Multiplier Using Approximate High-Order Compressors. In *2019 2nd International Conference on Communication Engineering and Technology (ICCET)*. 163–167. https://doi.org/10.1109/ICCET.2019.8726875

[36] Salim Ullah, Sanjeev Sripadraj Murthy, and Akash Kumar. 2018. SMApproxlib: library of FPGA-based approximate multipliers. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 157, 6 pages. https://doi.org/10.1145/3195970.3196115

[37] Salim Ullah, Semeen Rehman, Bharath Srinivas Prabakaran, Florian Kriebel, Muhammad Abdullah Hanif, Muhammad Shafique, and Akash Kumar. 2018. Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 159, 6 pages. https://doi.org/10.1145/3195970.3195996

Just Accepted

[38] Shaghayegh Vahdat, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2017. LETAM: A low energy truncation-based approximate multiplier. *Computers & Electrical Engineering* 63 (2017), 1–17.

[39] Shaghayegh Vahdat, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2019. TOSAM: An energy-efficient truncation-and rounding-based scalable approximate multiplier. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 5 (2019), 1161–1173.

[40] Nguyen Van Toan and Jeong-Gun Lee. 2020. FPGA-based multi-level approximate multipliers for high-performance error-resilient applications. *IEEE Access* 8 (2020), 25481–25497.

[41] Zdenek Vasicek. 2019. Formal methods for exact analysis of approximate circuits. *IEEE Access* 7 (2019), 177309–177331.

[42] Suganthi Venkatachalam and Seok-Bum Ko. 2017. Design of power and area efficient approximate multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 5 (2017), 1782–1786.

[43] C. S. Wallace. 1964. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers* EC-13, 1 (Feb 1964), 14–17. https://doi.org/10.1109/PGEC.1964.263830

[44] Xuan Wang and Weikang Qian. 2022. MinAC: Minimal-Area Approximate Compressor Design Based on Exact Synthesis for Approximate Multipliers. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. 677–681. https://doi.org/10.1109/ISCAS48785.2022.9938008

[45] Ying Wu, Chuangtao Chen, Weihua Xiao, Xuan Wang, Chenyi Wen, Jie Han, Xunzhao Yin, Weikang Qian, and Cheng Zhuo. 2024. A Survey on Approximate Multiplier Designs for Energy Efficiency: From Algorithms to Circuits. *ACM Trans. Des. Autom. Electron. Syst.* 29, 1, Article 23 (jan 2024), 37 pages. https://doi.org/10.1145/3610291

[46] Tongxin Yang, Tomoaki Ukezono, and Toshinori Sato. 2017. Low-Power and High-Speed Approximate Multiplier Design with a Tree Compressor. In *2017 IEEE International Conference on Computer Design (ICCD)*. 89–96. https://doi.org/10.1109/ICCD.2017.22

[47] Tongxin Yang, Tomoaki Ukezono, and Toshinori Sato. 2018. A low-power high-speed accuracy-controllable approximate multiplier design. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 605–610. https://doi.org/10.1109/ASPDAC.2018.8297389

[48] Zhixi Yang, Jie Han, and Fabrizio Lombardi. 2015. Approximate compressors for error-resilient multiplier design. In *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. 183–186. https://doi.org/10.1109/DFT.2015.7315159

Received 21 December 2023; revised 5 March 2024; accepted 15 April 2024

Just Accepted