High Flexibility Designs of Quantized Runtime Reconfigurable Multi-precision Multipliers

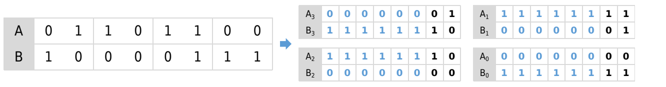
Yuhao Liu, Shubham Rai, Salim Ullah, Akash Kumar
Chair of Processor Design, Center for Advancing Electronics Dresden (CfAED), TU Dresden, Germany
Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI Dresden/Leipzig), Germany
Email: {yuhao.liu1, shubham.rai, salim.ullah, akash.kumar}@tu-dresden.de

***Abstract*—Recent research widely explored the quantization schemes on hardware. However, for recent accelerators only supporting 8 bits quantization, such as Google TPU, the lower-precision inputs, such as 1/2-bit quantized neural network models in FINN, need to extend the data width to meet the hardware interface requirements. This conversion influences communication and computing efficiency. To improve the flexibility and throughput of quantized multipliers, our work explores two novel reconfigurable multiplier designs that can repartition the number of input channels in runtime based on input precision and reconfigure the signed/unsigned multiplication modes. In this manuscript, we explored two novel runtime reconfigurable multi-precision multipliers based on the multiplier-tree and bit-serial multiplier architectures. We evaluated our designs by implementing a systolic array and single-layer neural network accelerator on the Ultra96 FPGA platform. The result shows the flexibility of our implementation and the high speedup for low-precision quantized multiplication working with a fixed data width of the hardware interface.**

***Index Terms*—Multiplier, Multi-precision, Runtime Reconfiguration, Quantization**

I. Introduction

Continuously extending demands for large-scale computing in recent state-of-the-art research and applications requires high-performance acceleration hardware platforms. For instance, in the research of neural network accelerators, the rapidly growing size of network models causes a huge hardware resource consumption in model storage, transmission, and inference computing. This trend prompted related researchers to explore various optimization subjects, such as quantization: one widely applied solution that can highly decrease the memory and computing resource consumption on accelerator design. To satisfy the requests on the inference of different network models, the industry community published various generic neural network accelerator products supporting quantization computing, for example, the *Google TPU* on Google Cloud [2]. However, these designs only support the quantization of network models to 8 bits. As opposed to the design strategy in the industry community, the other previous works have deeply explored the low-precision quantization (<8 bits), such as *FINN* [11], which can highly compress the parameter storage and reduce the resource consumption compared with 8-bit quantization with accuracy loss. This accuracy loss can be improved by selecting hyperparameters and retraining. Therefore, for instance, when researchers want to accelerate 2-bit quantized network models trained for *FINN* on the *Google TPU* hardware, as shown in Figure 1, the 2-bit input data need to extend the bit-width by filling placeholder bits (blue) to 8 bits. As a result, eight 2-bit inputs become eight 8-bit inputs. This conversion influences inference efficiency and costs more hardware resources to implement. Furthermore, because low-precision quantization causes an accuracy loss, previous works, such as *FILM-QNN* [9] and *MP-OPU* [13], explored the mixed low-precision quantization to trade off the inference accuracy and resource consumption by setting different precision for layers in the network. However, as shown in Table III, our experiment shows that the regular fixed-precision multiplier, such as Vivado IP, is not a good platform to achieve speedup for mixed-precision models based on its low precision in different layers because all data need to be unified to the data width of highest precision.



A	0	1	1	0	1	1	0	0
B	1	0	0	0	0	1	1	1

A_3	0	0	0	0	0	0	0	1
B_3	1	1	1	1	1	1	1	0

A_1	1	1	1	1	1	1	1	1
B_1	0	0	0	0	0	0	0	1

A_2	1	1	1	1	1	1	1	0
B_2	0	0	0	0	0	0	0	0

A_0	0	0	0	0	0	0	0	0
B_0	1	1	1	1	1	1	1	1

Fig. 1: Quad Channel 2-bit Signed Inputs (left) and Their Extended 8-bit Format with Placeholder for Recent Hardware (right)

Therefore, aiming to improve the flexibility of applying different precision and increase the throughput in network inference on hardware, we explored two designs of quantized *Runtime Reconfigurable Multi-precision Multipliers*. The key features and contributions of this work are:

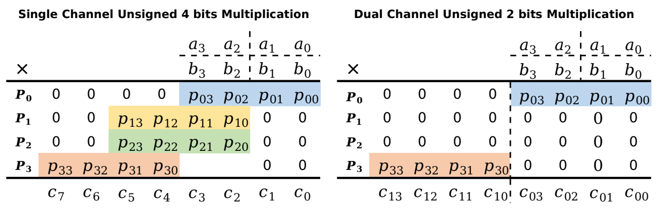
- This paper explored two multi-precision multiplier architecture, *Multiplier-Tree* and *Bitshifter* architectures, extended from *Partial Product Array Multiplier* and *BISMO Matrix Multiplier* separately.
- Our multiplier designs can reconfigure the number of input channels with a fixed input width in runtime. For example, assuming one instance of our work is designed as a 32 bits multiplier. The implementation can be configured as a 1/2/4/8/16/32 channels 32/16/8/4/2/1-bit multiplier.
- Our designs also support the runtime reconfiguration of switching between signed and unsigned multiplication modes.

II. Background

A. Partial Product Array Multiplier

Equation 1 shows the principle of an unsigned *Partial Product Array Multiplier*. For two inputs, $\mathbf{A}$ and $\mathbf{B}$, assuming the value of bit i in $\mathbf{A}$ and $\mathbf{B}$ is a_i and b_i, these inputs can be represented as the binarized format: $\langle a_{n-1}\cdots a_1 a_0\rangle_{bin}$ and $\langle b_{n-1}\cdots b_1 b_0\rangle_{bin}$. Because the n-bit multiplication in array multiplier has n partial products, the partial products P_i $(0 \leqslant i < n, i \in \mathbb{Z})$ can be computed with $\mathbf{A}$ and b_i as $P_i = (\mathbf{A} \times b_i) \times 2^i$. The final output $\mathbf{C}$ of multiplication is the accumulation result of all partial products, $\mathbf{C} = \sum_{i=0}^{n-1} P_i$. Therefore, for *Partial Product Array Multiplier*, if the input precision is n bits, the multiplier needs to compute n times partial products and add all these $2n$-bit results together as output.

$$\begin{aligned}\mathbf{A}\times\mathbf{B} =& \langle a_{n-1}a_{n-2}a_{n-3}a_{n-4}\cdots a_1 a_0\rangle_{bin}\\ &\times \langle b_{n-1}b_{n-2}b_{n-3}b_{n-4}\cdots b_1 b_0\rangle_{bin}\\ =&\mathbf{A}\times(2^{n-1}b_{n-1}+2^{n-2}b_{n-2}+...+2^1 b_1+2^0 b_0)\\ =&\mathbf{A}\times\sum_{i=0}^{n-1}2^i b_i=\sum_{i=0}^{n-1}[(\mathbf{A}\times b_i)\ll i]\end{aligned} \tag{1}$$


Single Channel Unsigned 4 bits Multiplication
Dual Channel Unsigned 2 bits Multiplication
P_0
P_1
P_2
P_3
Fig. 2: From Single Channel 4bits Unsigned Multiplier to Dual Channel 2bits Unsigned Channel in Previous Works

B. *BISMO Matrix Multiplier*

BISMO [10] implemented a novel design of a ***Bit-Serial Matrix Multiplication Overlay***, which convert the dot matrix multiplication as the bitwise-shifting and *AND* operations. This work is evaluated on FPGA and replaced all DSPs resources as LUTs in matrix calculation implementation. Equation 2 describes the principle of this design: For instance, assuming two input matrices, A and B, as two 2×2 2-bit matrices, both of these matrices can be converted as the sum of two 1-bit sub-matrices, $2^1A_1 + 2^0A_0$ and $2^1B_1 + 2^0B_0$. Therefore, the dot matrix multiplication of $A \cdot B$ can be represented as $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}[(A_i \cdot B_j) \ll (i+j)]$. Because A_i and B_j are 1-bit matrices, the result of $A_i \cdot B_j$ can be computed with bitwise operators. If we replace the 1-bit matrices A_i and B_j as the 1-bit value a_i and b_j, the *BISMO* can be converted from the matrix multiplication overlay to an unsigned integer multiplication.

$$
\begin{aligned}
&\begin{cases} A = \begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} = \begin{bmatrix} \langle a_{00}a_{01}\rangle_{bin} & \langle a_{10}a_{11}\rangle_{bin} \\ \langle a_{20}a_{21}\rangle_{bin} & \langle a_{30}a_{31}\rangle_{bin} \end{bmatrix} \\ B = \begin{bmatrix} b_0 & b_1 \\ b_2 & b_3 \end{bmatrix} = \begin{bmatrix} \langle b_{00}b_{01}\rangle_{bin} & \langle b_{10}b_{11}\rangle_{bin} \\ \langle b_{20}b_{21}\rangle_{bin} & \langle b_{30}b_{31}\rangle_{bin} \end{bmatrix} \end{cases} \\
\Rightarrow &\begin{cases} A = 2^1 \begin{bmatrix} a_{00} & a_{10} \\ a_{20} & a_{30} \end{bmatrix} + 2^0 \begin{bmatrix} a_{01} & a_{11} \\ a_{21} & a_{31} \end{bmatrix} \\ B = 2^1 \begin{bmatrix} b_{00} & b_{10} \\ b_{20} & b_{30} \end{bmatrix} + 2^0 \begin{bmatrix} b_{01} & b_{11} \\ b_{21} & b_{31} \end{bmatrix} \end{cases} \\
\Rightarrow &\begin{cases} A = 2^1 \cdot A_1 + 2^0 \cdot A_0 \\ B = 2^1 \cdot B_1 + 2^0 \cdot B_0 \end{cases} \\
\Rightarrow &\ A \cdot B = A_1 \cdot B_1 \times 2^2 + (A_1 \cdot B_0 + A_0 \cdot B_1) \times 2^1 \\
&\qquad + A_0 \cdot B_0 \times 2^0 \\
\Rightarrow &\ A \cdot B = \sum_{i=0}^{1}\sum_{j=0}^{1}[(A_i \cdot B_j) \ll (i+j)]
\end{aligned} \tag{2}
$$

C. *Previous Multi-precision Multiplier Designs*

Various previous works explored different designs of multi-channel multipliers: Shun et al. [8] promised a *Radix-4 Booth* multiplier-based multi-channel multiplier. Pfänder et al. [6] serialized the design in [8] to reduce hardware resource consumption. *PIR-DSP* [7] achieves a reconfigurable multi-precision architecture for multiplier by applying the DSP block design on FPGA. Neda et al. [5] and Guo et al. [1] explored the implementations of the approximate multi-precision multiplier with different methods to generate the partial products. The works mentioned above are developed based on a similar principle: We assume two $2N$-bit inputs for unsigned multiplication, for instance, the 4-bit inputs shown in Figure 2. We can divide these two inputs into four N-bit data, and one $2N$-bit multiplication can be converted to four N-bit multiplications. The outputs of four N-bit multiplications are the four partial products P_{0-3}. If we ignore the partial products, P_1 and P_2, the single channel $2N$-bit multiplication can be a dual channel N-bit multiplication. Compared with the above-mentioned works, our *Multiplier-Tree* architecture also applies the partial product method to achieve the reconfiguration and additionally supports the reconfiguration between signed/unsigned multiplication modes. Moreover, our *Bitshifter* architecture applies a different mathematics theory to create the runtime reconfigurable multi-channel multiplier by bitwise AND and shifting operations.

III. Implementation

A. *Computing Patterns*

Because the key features of our *R2M2* design support the runtime reconfiguration of i) multi-precision multi-channel multiplication restructure from 1- to N-bit; ii) switching between signed and unsigned modes; iii) applying different precision of two inputs, the implementations in this work support four different computing patterns:

1) 1×1 *bit Multiplication*: Because in *Binarized Neural Network* (BNN), the -1 is represented as '0' and 1 is represented as '1', the binarized multiplication is the bitwise *XNOR* operation.
2) $1 \times N$ *and* $N \times N$ *bits Unsigned Multiplication*: Implementing a regular unsigned multiplier can support this computation.
3) $N \times N$ *bits Signed Multiplication*: Implementing a regular signed multiplier can support this computation.
4) $1 \times N$ *bits Signed Multiplication*: Because binarized data represents the -1 as '0', the result of '0'$\times A$ is not 0, but $-A$.

Furthermore, to simplify the processing, if the precision n of one input is $2^{m-1} < n < 2^m$, these inputs need to be extended to 2^m bits with placeholders. If the precision n_0 and n_1 of two inputs are different, both inputs need to be unified to the same precision, the higher one. And the input with the higher original precision should be placed as the second input for our multipliers.

B. *Multiplier-Tree Architecture*

In subsection II-C, previous work [8, 6, 7, 5, 1] explored how to build a $2N$-bit multiplier with four N-bit multipliers to support the runtime reconfiguration between $2N$- and N-bit. Therefore, if we design a reconfigurable 2-bit multiplier as the basic unit, any 2^n-bit multi-precision multipliers can be created as a nested structure. However, to achieve the runtime reconfiguration between signed and unsigned multiplication, we need to extend the methods applied in previous works. As shown in Equation 3, suppose we compute a 4-bit signed multiplication:

$$
\begin{aligned}
\mathbf{A} \times \mathbf{B} &= \langle a_3a_2a_1a_0\rangle_{bin} \times \langle b_3b_2b_1b_0\rangle_{bin} \\
&= (-2^3a_3 + 2^2a_2 + 2^1a_1 + 2^0a_0) \times \\
&\quad (-2^3b_3 + 2^2b_2 + 2^1b_1 + 2^0b_0) \\
&= \left[\mathbf{(\text{-}2^1a_3{+}2^0a_2)} \times \mathbf{(\text{-}2^1b_3{+}2^0b_2)}\right] \ll 4 + \\
&\quad \left[\mathbf{(\text{-}2^1a_3{+}2^0a_2)} \times (2^1b_1 + 2^0b_0)\right] \ll 2 + \\
&\quad \left[\mathbf{(\text{-}2^1b_3{+}2^0b_2)} \times (2^1a_1 + 2^0a_0)\right] \ll 2 + \\
&\quad \left[(2^3a_1 + 2^2a_0) \times (2^3b_1 + 2^2b_0)\right] \ll 0
\end{aligned} \tag{3}
$$

The two 4-bit signed inputs, $\mathbf{A}$ and $\mathbf{B}$, can be broken down into two 2-bit signed sub-inputs (bold parts) and two 2-bit unsigned inputs. Therefore, one *'signed 2-bit* $\times$ *signed 2-bit'* multiplier, two *'signed 2-bit* $\times$ *unsigned 2-bit'* multipliers, and one *'unsigned 2-bit* $\times$ *unsigned 2-bit'* multiplier can build a signed 4-bit signed multiplier. If we implement the basic 2-bit multipliers, supporting the runtime reconfiguration of *'signed* $\times$ *signed'*, *'signed* $\times$ *unsigned'*, and *'unsigned* $\times$ *unsigned'* multiplications, the 4-bit multiplier built based on it can support the runtime reconfiguration of signed/unsigned modes: When four 2-bit multipliers work in unsigned mode, this

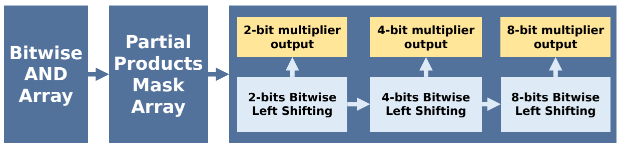

Bitwise AND Array
Partial Products Mask Array
2-bit multiplier output
4-bit multiplier output
8-bit multiplier output
2-bits Bitwise Left Shifting
4-bits Bitwise Left Shifting
8-bits Bitwise Left Shifting

Fig. 3: Hardware Structure of 8-bit Bitshifter Multiplier

4-bit multiplier is an unsigned reconfigurable multi-precision multiplier. Else, when all four 2-bit multipliers are reconfigured as Equation 3, this 4-bit multiplier works in the signed mode. Therefore, based on the same principle, we can extend this 4-bit multiplier to 8 bits. However, we also need the reconfigurable 4-bit multiplier to support *'signed × unsigned'* multiplications.

$$\begin{aligned}\mathbf{A}\times\mathbf{B} =& \langle a_3a_2a_1a_0\rangle_{bin} \times \langle b_3b_2b_1b_0\rangle_{bin} \\ =&(-2^3a_3+2^2a_2+2^1a_1+2^0a_0)\times \\ &(2^3b_3+2^2b_2+2^1b_1+2^0b_0) \\ =&\left[(\mathbf{-2^1a_3{+}2^0a_2})\times(2^1b_3+2^0b_2)\right] \ll 4+ \\ &\left[(\mathbf{-2^1a_3{+}2^0a_2})\times(2^1b_1+2^0b_0)\right] \ll 2+ \\ &\left[(2^1a_1+2^0a_0)\times(2^1b_3+2^0b_2)\right] \ll 2+ \\ &\left[(2^3a_1+2^2a_0)\times(2^3b_1+2^2b_0)\right] \ll 0\end{aligned} \tag{4}$$

As shown inEquation 4: one 4-bit *'signed × unsigned'* multiplier can be broken down to two *'signed 2-bit × unsigned 2-bit'* multipliers and two *'unsigned 2-bit × unsigned 2-bit'* multipliers. Furthermore, Equation 5 shows how to apply two *1-bit × signed 2-bit* multipliers and one offset, 2^2b_1, to achieve one *1-bit × signed 4-bit* multipliers.

$$\begin{aligned}\pm\mathbf{1}\times\mathbf{B} =& \pm\mathbf{1}\times\langle b_3b_2b_1b_0\rangle_{bin} \\ =&\pm1\times(-2^3b_3+2^2b_2+2^1b_1+2^0b_0) \\ =&\left[\pm1\times(\mathbf{-2^1b_3{+}2^0b_2})\right] \ll 2+ \\ &\left[\pm1\times(\mathbf{-2^1b_1{+}2^0b_0})\right] \ll 0 \pm 2^2b_1\end{aligned} \tag{5}$$

Therefore, based on a basic 2-bit multiplication unit, supporting the runtime reconfiguration of *'signed × signed'*, *'signed × unsigned'*, *'unsigned × unsigned'*, *'1-bit × signed'*, and *'1-bit × 1-bit'* multiplications modes, we can create any 2^n-bit reconfigurable multipliers with a nesting structure. These 2-bit multiplier modules are built with five constant arrays by Verilog to look up the corresponding output based on the inputs.

C. Bitshifter Design

subsection II-B introduced the principle of BISMO [10] and pointed out that it can be extended to design a regular multiplier. Therefore, based on *BISMO*, assuming two input matric have only one element, the unsigned multiplication can be converted as Equation 6:

$$\begin{aligned}\mathbf{A}\times\mathbf{B} =&< a_{n-1}a_{n-2}...a_1a_0 >_{bin} \times < b_{n-1}b_{n-2}...b_1b_0 >_{bin} \\ =&(2^{n-1}a_{n-1}+2^{n-2}a_{n-2}+...+2^1a_1+2^0a_0)\times \\ &(2^{n-1}b_{n-1}+2^{n-2}b_{n-2}+...+2^1b_1+2^0b_0) \\ =&[(a_{n-1}b_{n-1}) << (2n-2)] \\ &+[(a_{n-1}b_{n-2}+a_{n-2}b_{n-1}) << (2n-3)] \\ &+... \\ &+[(a_1b_0+a_0b_1) << 1] \\ &+[(a_0b_0) << 0]\end{aligned} \tag{6}$$

For two n-bit inputs $A\,\langle a_{n-1}...a_1a_0\rangle$ and $B\,\langle b_{n-1}...b_1b_0\rangle$, the product can be computed with the sum of $2n-1$ partial products, $P_i,(0 < i < 2n-1)$. And P_i consists of sub partial products a_jb_k as $P_i = \sum(a_jb_k) \ll i,(j+k=i, 0<j,k<n)$. The values of partial products P_i need to apply a i-bit

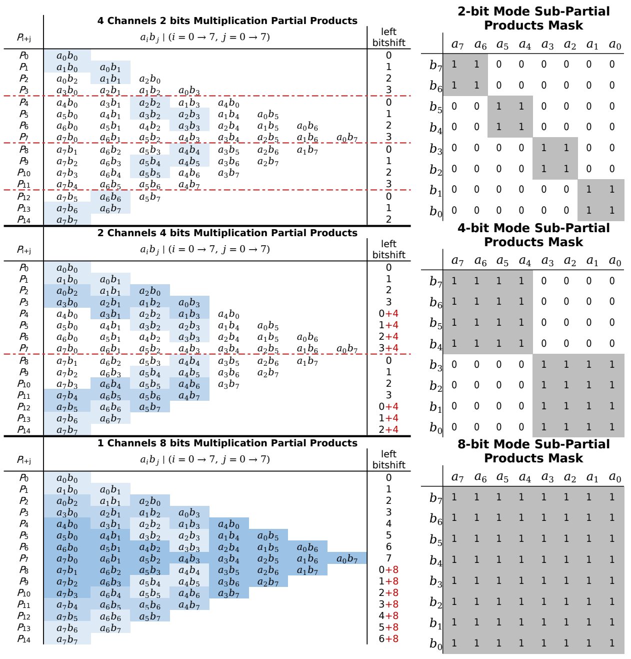

4 Channels 2 bits Multiplication Partial Products
$a_ib_j \mid (i=0\rightarrow7, j=0\rightarrow7)$
left bitshift
2 Channels 4 bits Multiplication Partial Products
1 Channels 8 bits Multiplication Partial Products
2-bit Mode Sub-Partial Products Mask
4-bit Mode Sub-Partial Products Mask
8-bit Mode Sub-Partial Products Mask

Fig. 4: 1/2/4 Channels Unsigned 8/4/2 bits Bitshifter Multiplication and Sub-Partial Products Masks

bitwise left shifting. Therefore, this multiplier only contains adders, bitwise left shifting, and bitwise *AND* operators. To design a multi-channel reconfigurable multiplier: As the instance shown in Figure 4, assuming the input width of one *Bitshifter* architecture-based multiplier is 8 bits, it needs to create a 15x8 register array to store sub partial products, (a_ib_j). Red lines divide this array into different channels by precision. From the 2/4/8-bit modes shown in Figure 4, we can find that if the applied input precision is M when $|i-j| \geqslant M$ or $(i \mod M) \neq (j \mod M)$, sub-partial products a_ib_j can be set as zero. The blued a_ib_j means it is available for the applying precision and channel setting, which can be detected by a sub-partial product mask on the right side. Besides, from the left bitshift column in Figure 4, the left bit-shifting numbers for 2/4/8-bit multiplication are constant values: For 2-bit multiplication, the bitwise left shifting bits always repeat from $0 \rightarrow 3$; For 4-bit multiplication, the bitwise left shifting bits need to add 4 based on the shifting of 2-bit multiplication when $[(i+j) \mod 8] > 3$; For 8-bit multiplication, the bitwise left shifting bits add 8 based on the shifting of 4-bit multiplication when $[(i+j) \mod 16] > 7$. Therefore, we can first implement the hardware for 2 bits multiplication and extend it to 4, 8, or higher precision as shown in Figure 3. Besides, because the *Bitshifter* architecture focuses on unsigned multiplication, the signed inputs must be converted to an absolute value before the multiplication, and the final output will be converted back to the signed value. XNOR multipliers are implemented as one individual module for binarized inputs.

IV. Evaluation

To evaluate the two multiplier architectures approached in this manuscript, we implemented their 8-, 16, and 32-bit instances, three simple systolic arrays, and three single-layer neural network accelerators on *Ultra96-V2* FPGA platform. Table I listed the hardware resource consumption of multiplier instances compared with Vivado-IP and the designs of Neda et.al [5]. Table II listed the resource consumption and computing latency of three systolic arrays and Table III list the evaluation result of three

TABLE I: Resource Consumption Evaluation of *Multiplier-Tree* and *Bitshifter* Architecture

Design	Input Width	Available Precision	Accurate/ Approximate	Unsigned/ Signed	LUTs
Vivado IP	32 bits	32	Accurate	Unsigned	1012
	16 bits	16			249
	8 bits	8			60
mpDNN-Approx3 [5]	16 bits	4/8/16	Approximate	Signed	324
	8 bits	4/8			74
mpDNN-AO [5]	16 bits	4/8/16			261
	8 bits	4/8			59
Multiplier-Tree	32 bits	1/2/4/8/16/32	Accurate	both	6601
	16 bits	1/2/4/8/16			1670
	8 bits	1/2/4/8			333
Bitshifter	32 bits	1/2/4/8/16/32		both	3446
	16 bits	1/2/4/8/16			1002
	8 bits	1/2/4/8			285

TABLE II: Systolic Array Evaluation on Ultra96-V2

Implementation		Vivado IP		Multiplier Tree		Bitshifiter	
Input Width		32 bits					
Systolic Size		2×2					
PL Frequency		100 MHz		50 MHz		60 MHz	
Predicted Power		2.164W		2.114W		2.109W	
LUTs		122333	17.34%	33076	46.88%	21848	30.96%
FFs		8532	6.05%	6982	4.95%	6121	4.34%
LUTRAMs		692	2.40%	475	1.65%	363	1.36%
BRAMs		85	39.35%	71	32.87%	71	32.87%
Matrix Multiplication Latency	32 bits	52.97ms		105.93ms		88.29ms	
	8 bits			1.72ms		1.42ms	
	1 bit			15.27μs		12.14μs	

TABLE III: Single-Layer Accelerator on Ultra96-V2

Design	Precision	LUT		FF		BRAM		Frequency	Latency
Vivado IP	8/8/8/8	24090	34.14%	22175	15.71%	135	62.50%	150 MHz	137.654μs
	1/2/4/8								131.059μs
Bitshifter	1/2/4/8	42952	60.87%	22486	15.93%	138	63.89%	125 MHz	56.658μs
Multiplier Tree		37020	52.47%	22500	15.94%			100 MHz	69.274μs

single-layer accelerators. The input width of multipliers applied in systolic arrays and single-layer accelerators are 32-bit and 8-bit. As shown in Table I, all multi-precision multiplier designs in Neda et.al [5] and our work consume more hardware resources than regular design in Vivado IP. The two implementations in Neda et.al [5] show less hardware overhead to achieve the runtime reconfiguration of multi-precision than our designs. However, their work executes the approximate multiplication and supports less precision. Their multiplier designs also can not support the switching between signed and unsigned modes in runtime. For the two multiplier architectures proposed in our work, *Bitshifter* architecture costs fewer resources than *Multiplier-Tree*: For their 8/16/32-bit instance, *Bitshifter* architecture consumes 85.6%, 60.0%, and 52.2% LUTs of the *Multiplier-Tree* instances in same precision. Besides, the multiplier designs based on Vivado IP, works of Neda et.al [5], and *Multiplier-Tree* architecture cost about four times LUT resources in doubling the input width. However, *Bitshifter* architecture only consumes about 3.3 times the LUT resources to double the input width. Furthermore, considering systolic array is widely applied in FPGA-based accelerator designs and explored in previous works like *HiMap* [12] and *ChordMap* [4], we implement three 2×2 systolic arrays as evaluation applying 32-bit Vivado-IP, *Multiplier-Tree*, and *Bitshifter* architectures based multipliers. Table II listed the average latency after 1000 times matrix multiplication between two 256×256 matrices based on three systolic arrays. The result shows that *Bitshifter* architecture has the advantage in timing performance and resource consumption compared with *Multiplier-Tree* architecture: it is always faster than *Multiplier-Tree*-based systolic array in different precision because of it supports a higher frequency. The Vivado-IP-based systolic array is faster than our two architecture designs when the values in matrices are 32-bit integers. However, when matrices are quantized to 1/8 bits, the matrix computation based on our two architectures can achieve about 60× and 7000× speedup in the same systolic array instances because Vivado-IP-based systolic array doesn't support multi-precision reconfiguration in runtime. Moreover, to evaluate the acceleration of our multiplier designs on network inference, we trained two four-layer quantized MLP models containing four 8-bit quantized layers and four 1/2/4/8-bit mixed-precision quantized layers separately. The neuron numbers in the four layers are 64/64/64/10. Because the inference accuracy on the MNIST dataset [3] of regular quantized and mixed-precision models is 97.74% and 95.96%, and the mixed-precision model reduced the 83.1% of network weight storage, mixed-precision quantization shows its advantage in the trade-off between accuracy and resource consumption. However, Table III shows that, unlike our multi-precision multiplier designs, the regular multiplier, such as Vivado IP, can not speed up the inference of mixed-precision models. Based on our two multiplier designs, the accelerators reduced 56.8% and 47.1% latency in inference.

V. Conclusion

This manuscript presents two architectures, *Multiplier Tree* and *Bitshifter*, for runtime reconfigurable multi-precision multiplier design supporting the runtime repartition of the input channel according to the input precision and reconfiguration of signed/unsigned computation mode. We separately implement one 8-, 16-, and 32-bit instance for these two multiplier architectures. All instances have been tested with $1 \sim 32$-bit inputs. Moreover, we implemented three 2×2 systolic arrays and three single-layer neural network accelerators based on *Vivado* multiplier IP and our two multiplier architectures for evaluation. Results show that our designs can achieve high flexibility with multi-precision and multi-channel reconfiguration in runtime and speed up the computation of low-precision inputs.

References

[1] Chuliang Guo et al. "A Reconfigurable Approximate Multiplier for Quantized CNN Applications". In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2020, pp. 235–240.

[2] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: 2017.

[3] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[4] Zhaoying Li et al. "ChordMap: Automated Mapping of Streaming Applications Onto CGRA". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.2 (2022), pp. 306–319.

[5] Negar Neda et al. "Multi-Precision Deep Neural Network Acceleration on FPGAs". In: *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2022, pp. 454–459.

[6] Oliver A. Pfänder et al. "Configurable Blocks for Multi-precision Multiplication". In: *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*. 2008, pp. 478–481.

[7] SeyedRamin Rasoulinezhad et al. "PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, pp. 35–44.

[8] Zhou Shun et al. "A VLSI architecture for a Run-time Multi-precision Reconfigurable Booth Multiplier". In: *2007 14th IEEE International Conference on Electronics, Circuits and Systems*. 2007, pp. 975–978.

[9] Mengshu Sun et al. "FILM-QNN: Efficient FPGA Acceleration of Deep Neural Networks with Intra-Layer, Mixed-Precision Quantization". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. Virtual Event, USA: Association for Computing Machinery, 2022, 134–145.

[10] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander. "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2018, pp. 307–3077.

[11] Yaman Umuroglu et al. "Finn: A framework for fast, scalable binarized neural network inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 65–74.

[12] Dhananiaya Wijerathne et al. "HiMap: Fast and Scalable High-Quality Mapping on CGRA via Hierarchical Abstraction". In: *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2021, pp. 1192–1197.

[13] Chen Wu et al. "MP-OPU: A Mixed Precision FPGA-based Overlay Processor for Convolutional Neural Networks". In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2021, pp. 33–37.