Design and Implementation of Low-Power Hardware Computation for Convolutional Variational Autoencoders (CVAEs) on FPGA

Nguyen Van Luu*, Nguyen Thi Thuy Linh*
Supervised by Assoc.Professor Ph.D Nguyen Duc Minh (EDABK Lab)
School of Electrical and Electronic Engineering, Hanoi University of Science and Technology, Vietnam
City: Ha Noi, Country: Viet Nam
Corresponding Author: luu.nv192993@sis.hust.edu.vn

Abstract—**Variational Autoencoders (VAEs) have gained significant attention in generative modeling due to their ability to model complex data distributions. By combining probabilistic inference with deep neural networks, VAEs can learn efficient representations of data, making them valuable tools in various applications such as image generation, anomaly detection, and data imputation. Their flexibility allows them to be used across diverse fields, including computer vision, natural language processing, and bioinformatics. Building upon the success of traditional VAEs, Convolutional Variational Autoencoders (CVAE) leverage the power of convolutional neural networks (CNNs) to improve the performance of VAEs, especially in tasks involving image and spatial data. By incorporating convolutional layers, CVAEs are able to better capture the spatial hierarchies and intricate patterns present in high-dimensional data, such as images and videos. However, deploying CVAEs on hardware poses challenges due to their high computational and memory requirements, along with the complex computations involved in variational inference and latent space modeling. Designing efficient hardware accelerators for CVAEs involves balancing speed, power consumption, and resource usage to optimize performance while managing the computational intensity of the model.**
In this research, the design and implementation of a computation unit for CVAE includes a convolutional accelerator, layers, sampling layer, fully connected layers. This study utilizes a data flow called weight stationary (WS) to minimize data movement and reuse partial sums based on spatial architecture with an array of processing elements. We propose a custom Gaussian sampling layer using the Mersenne Twister algorithm to generate random numbers and a Look-Up Table (LUT) to implement the Box-Muller transform for generating Gaussian random numbers, , produced by a Gaussian random number generator (GRNG). This system processes the convolutional process layers at a rate of 33.5 frames per second, with DRAM access per multiply-and-accumulate (MAC) operation being 0.0844 for the AlexNet model, 0.111 for the VGG-16 model (batch size N=1) and in CVAEs model is 0.095 while the total power consumption of the entire network is 4.87 W.

Index Terms—**Convolutional Variational Autoencoders (CVAEs), FPGA, weight stationay, spatial architecture, encoder, decoder, sampling.**

I. Introduction

Variational Autoencoders (VAEs) [1] have been widely recognized for their ability to learn complex data distributions through a probabilistic framework, enabling the generation of realistic synthetic data. However, while VAEs are powerful in modeling data in lower-dimensional spaces, their performance in handling high-dimensional, structured data, such as images or videos, can be limited. The inherent limitation lies in their reliance on fully connected layers, which may struggle to capture the spatial dependencies and hierarchical patterns that are present in such data. To address this, Convolutional Variational Autoencoders (CVAE)[2] have been introduced as an extension of VAEs, incorporating convolutional layers into the architecture to better capture spatial structures and local dependencies inherent in image data.

CVAEs combine the probabilistic framework of VAEs with the spatially-aware capabilities of convolutional neural networks (CNNs)[3], allowing for more effective modeling of complex data distributions, particularly in tasks such as image generation, denoising, and image-to-image translation. By leveraging the power of convolutional layers, CVAEs are able to learn and generate high-quality, realistic images with finer details, making them highly suitable for applications in computer vision. However, the convolutional layers introduce significant computational complexity. Convolution operations, especially when applied to high-resolution images, require intensive matrix multiplications and large amounts of memory to store the intermediate feature maps. These operations become increasingly expensive as the network deepens and the resolution of the input data increases. This added computational burden results in longer processing times, higher power consumption, and increased memory bandwidth requirements, particularly when the model is deployed on resource-constrained hardware. Despite these challenges, the advantages of CVAEs in terms of model accuracy and the ability to capture spatial hierarchies make them a powerful tool, with efforts being made to optimize their deployment through hardware acceleration techniques such as FPGAs, aiming to balance performance, latency, and energy efficiency in real-time applications..
The demands substantial data movement between on-chip and off-chip memory to support computation. Since data movement can consume more energy than the computation itself

[4], optimizing CNN processing involves not only achieving high parallelism for increased throughput but also enhancing the efficiency of data movement across the system. To address these challenges, it is crucial to design a compute scheme, called a dataflow, that can support a highly parallel compute paradigm while optimizing the energy cost of data movement from both on-chip and off-chip. The cost of data movement is reduced by exploiting data reuse in a multilevel memory hierarchy. Some key matrics when implement a DNN accelerator as below:

- ***Throughput and Latency***: Avoid MACs (Multiply–accumulate) unescessary to avoid latency.
- ***Energy and Power Consumption***: Exploit the reuse data, reduce the energy of Processing Element.
- ***Flexiblity***: Use flexible dataflow to exploit reuse in any dimension of DNN to increase energy efficiency.
- ***Scablity***: Increate how performance(latency, energy, power) scales with in amount of resource.

Researchers have introduced various computational acceleration architectures to tackle the significant demands of computation and memory. These architectures are based on Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs) [5], [6], [7], [8], [9], [10].
In this study, a hardware computation unit for CVAE is implemented, including convolutional computation (CONV), fully connected layers (FC), and Gaussian Sampling layer (including Pseudo Random Number Generator-PRNG and Gaussian Random Number Generator-GRNG) . Most of the computations in CVAE come from the convolutional and convolutional tranpose layers. To optimize performance, the key contributions of this work are:

(1) A data flow called ***weight stationary*** base on spartial architecture is employed, where weights are kept fixed within an array of Processing Elements (PEs).
(2) The utilization of hierarchical memory structure and FIFO asynchronous on-chip buffer reduces the off-chip memory access and reuse data.
(3) A Pseudo Random Number Generator (PRNG) generates sequences of numbers that simulate randomness. We implemented the Mersenne Twister (MT) algorithm for efficient random number generation in hardware applications.
(4) The proposed Gaussian sampling layer requires Gaussian random numbers ϵ generated by a Gaussian random number generator (GRNG) using the Box-Muller transform, with Look-Up Tables (LUTs) used to implement the $\cos()$, $\ln()$, and e^x functions.
(5) The IPs are parameterized to be adaptable for deployment across different configurations of the CVAE, ensuring flexibility in hardware implementations.

This paper is organized as follows. Part II provides fundamental knowledge of CVAE operation, part III covers the system design, and part IV describes experimental setup and result.

II. Background of CVAE

Variational Auto Encoders: A conventional Autoencoder contains an encoder and a decoder, as shown in Fig. 4. The encoder encrypts inputs into a compressed representation (latent layer), while the decoder tries to reconstruct the compressed representation back to the original inputs. In practice, such classical autoencoders do not lead to particularly useful or nicely structured latent spaces. The conventional AE does not lead to nicely structured latent spaces, as the parameters in the latent vector are **selected randomly** when generating the output. To address this issue, VAE is proposed in [11] by Kingma and Welling, which converts the inputs into a representation of a Gaussian distribution: a mean and a variance. These parameters are used to randomly sample an element that is then decoded back to the original inputs. This Stochastic process improves the design robustness and forces the latent vectors to encode meaningful representations. The latent vector Z has a Gaussian distribution:

$$Z = \mu + (\sigma \odot \epsilon) \quad (1)$$

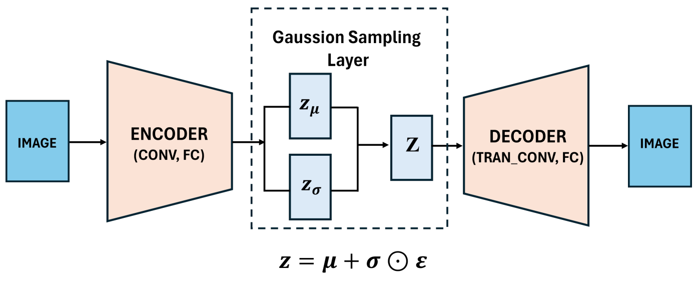


Fig. 1. Convolutional VAE architecture overview.

Convolutional Variational Auto Encoders: A generative model which combines the strengths of convolutional neural networks and variational autoencoders. Variational Autoencoder (VAE) [12] works as an unsupervised learning algorithm that can learn a latent representation of data by encoding it into a probabilistic distribution and then reconstructing back using the convolutional layers which enables the model to generate new, similar data points. The key working principles of a CVAE include the incorporation of convolutional layers, which are adept at capturing spatial hierarchies within data, making them particularly well-suited for image-related tasks. Additionally, CVAEs utilize variational inference, introducing probabilistic elements to the encoding-decoding process. Instead of producing a fixed latent representation, a CVAE generates a probability distribution in the latent space, enabling the model to learn not just a single deterministic representation but a range of possible representations for each input. The architecture of

CVAE is described in figure below:

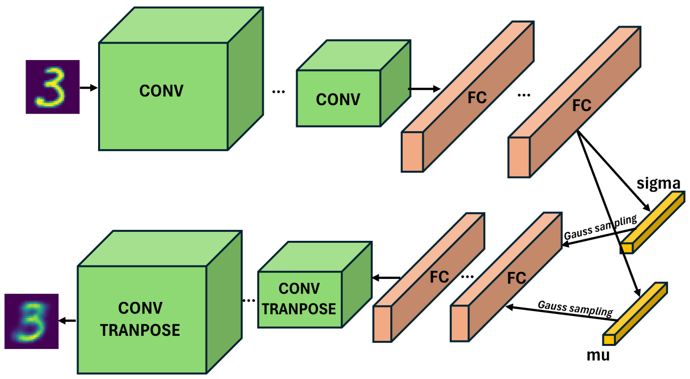


Fig. 2. Convolutional VAE architecture overview.

CNNs: are constructed from multiple computational layers organized as a directed acyclic graph (DAG)[13]. Each layer extracts an abstraction of data provided by the previous layer, which is referred to as a feature map (fmap). The most common layers in CNNs are convolution (CONV), pooling (POOL), and fully connected (FC) layers [14]. In CONV layers, as illustrated in figure 6, two-dimensional (2-D) filters slide over the input images or feature maps (Ifmaps), performing convolution operations to extract feature characteristics from local regions and generating output images or feature maps (Ofmaps). In the case of three-dimensional (3D) convolution, a batch of 3-D ifmaps is processed by a group of 3-D filters in a layer.

In addition, there is a 1-D bias that is added to the filtering results. Given the shape parameters in Table I, the computation of a layer is defined as

$$\mathbf{O}[z][u][x][y] = \textbf{ReLU}\Bigg(B[u] + \sum_{k=0}^{C-1}\sum_{i=0}^{R-1}\sum_{j=0}^{S-1} I[z][k][U_x+i][U_y+j]\mathbf{W}[u][k][i][j] \Bigg), \quad (2)$$

$$0 < z < N,\ 0 < u < M,\ 0 < y < E,\ 0 < x < F,$$

$$E = \frac{H - R + U}{U},\ F = \frac{W - S + U}{U}$$

where **O**, **I**, **W**, and **B** are the matrices of the of maps, ifmaps, filters, and biases, respectively. U is a given stride size. Fig. 6 shows a visualization of this computation (ignoring biases). After the convolutions, activation functions, such as the rectified linear unit (ReLU) [15], are applied to introduce nonlinearity.

III. System Design

A. Convolutional Layer Architecture

Figure 4 illustrates the block diagram of the architecture and memory hierarchy of the convolutional accelerator, which includes a PE array, global buffer, controller block and ReLU activation function. This block is responsible for convolution operations, max pooling, ReLU, and fully connected layers. The weights, biases, and input feature maps are stored in off-chip DRAM and are read into the accelerator via buffers to reduce latency when accessing off-chip memory. The memory hierarchy consists of three types: off-chip DRAM, a global buffer (FIFO buffer), and registers within each PE.

Table I
Shape parameter of a CNN layer

Shape parameter	Description
N	batch size
M	number of filter/ofmap channel
C	channel ifmap
H/W	ifmap height/width
R/S	filter height/width
E/F	ofmap height/width

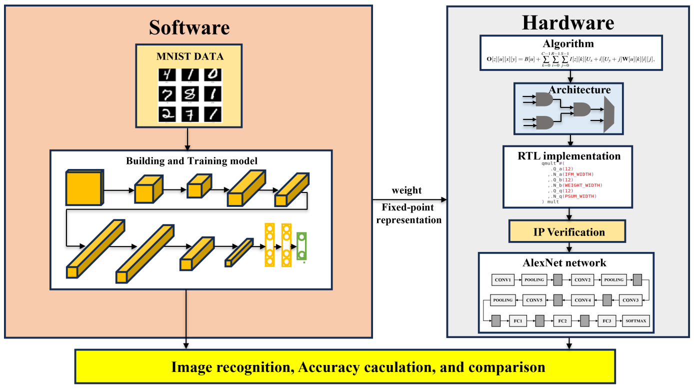

Software
MNIST DATA
Building and Training model
weight
Fixed-point representation
Hardware
Algorithm
Architecture
RTL implementation
IP Verification
AlexNet network
Image recognition, Accuracy caculation, and comparison

Fig. 3. Software and Hardware co-design flow.

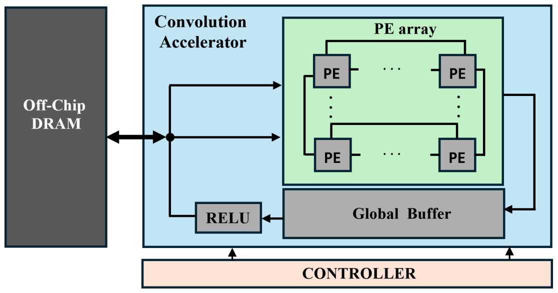


Fig. 4. System architecture overview.

Each PE in the PE array is responsible for computing a convolution operation or max-pooling and accumulating the result through the internal PE register and a global buffer. The FIFO buffer is closely associated with the PE array in rows. The accelerator is controlled by finite state machine (FSM) in controller block.

Dataflow: Data flow is a major challenge when designing computing units for convolutional layers, as computations in these layers are highly complex and involve a large amount of memory access. To optimize data movement, we use a dataflow called ***weight stationary***. In this dataflow, the weight filters are stored statically in small local memory such as registers in PE, forming a PE array of size RxS, corresponding to the size of the kernel matrix. The input feature map (activation) is streamed row by row with a bandwidth of 1 pixel per cycle, broadcasting activations and accumulating partial sums spatially across the PE array. Each activation is multiplied and accumulated with the weight stored statically in the PE. Each primitive multiplication result needs to be stored and accumulated with others to form partial sums. By using asynchronous FIFO, we can store and reuse the primitive results for subsequent references. The number of buffers needed is equal to the number of rows of the weight matrix and the size of the FIFO depends on the row size of the input feature map (IFM). The size of the buffer is calculated using the following:

$$\text{Fifo size} = \frac{W + 2p - k}{s} + 1 \tag{3}$$

This architecture reduces the energy required for weight reads, maximizes convolutional operations, and enables efficient reuse of the filter.

- *Filter reuse*: Each filter weight is reused E x F times within one input feature map (ifm) channel.
- *IFM reuse*: Each input feature map (ifm) is reused R x S times.

The PE array will perform a 2-D convolution between the kernel and the IFM window, with each row of the PE array executing a 1-D convolution multiplication as described in Fig 8. Initially, the first pixel, ifm1, from row 1 is pushed into the PE array, at which point psum_in in all PEs is initialized to 0. The result of the multiplication is stored in the register within each PE and passed to the adjacent PE via the psum_in signal at each PE. After W cycles, the first row has been fully read, and the FIFOs are filled, ready to push one value per cycle to the psum_in input of the first PE in the row below. The sliding window will shift downward until all H rows of the IFM are read, at which point the values in FIFO_END represent the result of the 2-D convolution of the kernel (RxS) with the IFM (RxW). However, this is not the final result, as in 3-D convolution, accumulation occurs along the depth dimension, using a buffer of size ExF to temporarily store the 2-D convolution results for each channel. The mux will select input from the FIFO for the first channel's computation and alternate for the other channels.

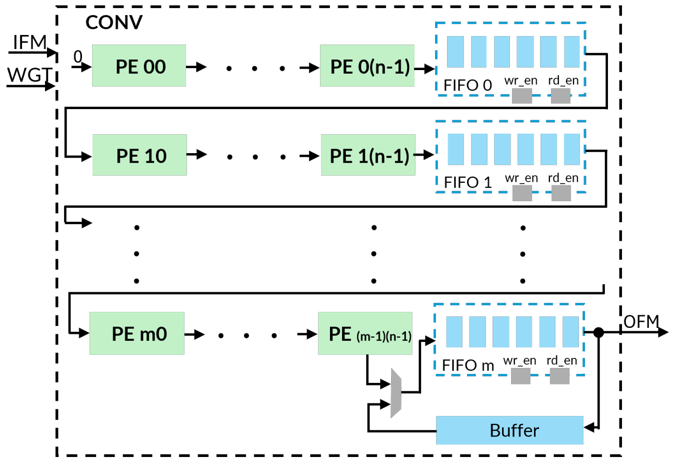


Fig. 7. Convolutional architecture hardware implementation for configurable PE array size.

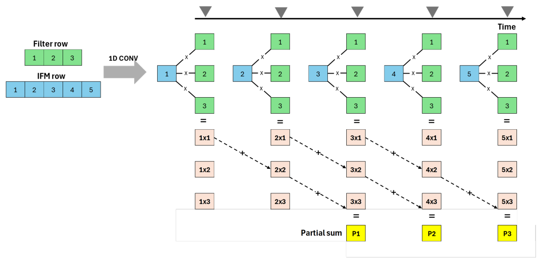


Fig. 8. Processing sequence of a 1-D convolution primitive in a PE. In this example, the filter row size (S) and the ifmap row size (W) are 3 and 5, respectively.

1-D Convolution Primitive PE array: the weight stationary dataflow first divides the computation in (1) into 1-D convolution primitives that can all run in parallel. Each primitive operates on one value of filter weights and one row of ifmap values and generates a row of psum. The result from different primitives are then further accumulated to generate the partial sum and ofmap value.

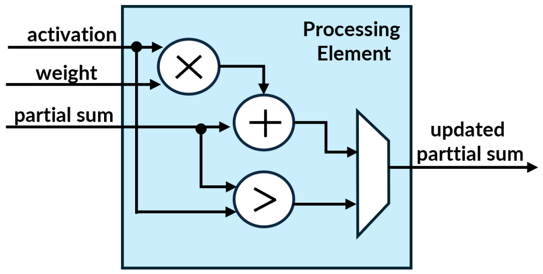
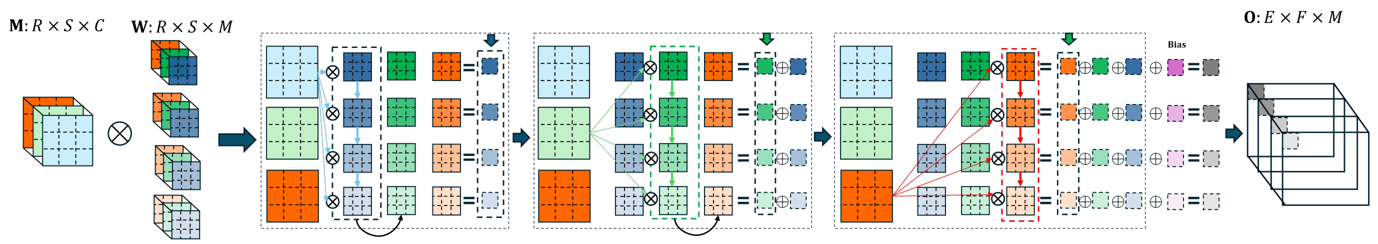


Fig. 9. Convolutional architecture hardware implementation for configurable PE array size.

By mapping each primitive to one PE for processing, the computation of each value stays stationary in the PE. Due to the sliding window processing of each primitive, as shown in Fig. 8, each PE can utilize local scratchpad memory (spads) for both convolutional data reuse and psum accumulation. Since



$\mathbf{M}: R \times S \times C$
$\mathbf{W}: R \times S \times M$
Bias
$\mathbf{O}: E \times F \times M$

Fig. 5. 3-D convolutional operator: transforming multi-channel input to multi-channel output.

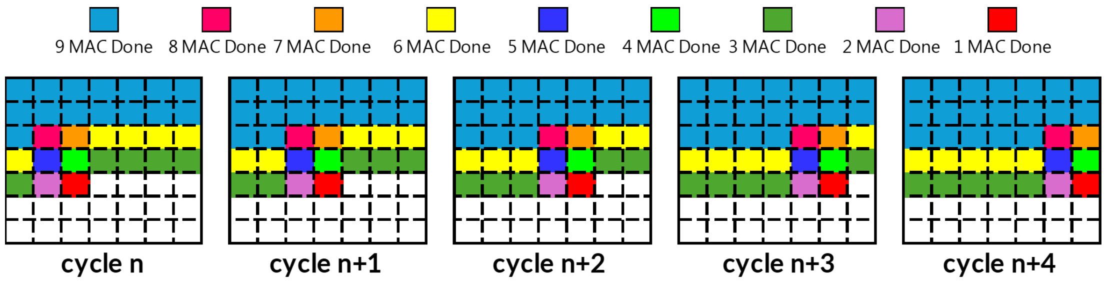


Fig. 6. 2-D convolutional operator with pipeline computing: completion level of each partial sum per cycle, the colored cells represent partial sums, where partial sums with the same color share the same completion level.

only a sliding window of data needs to be retained at any given time, the required is one in a PE.

2-D Convolution Primitive PE array: A 2-D convolution is composed of multiple 1-D convolution operations. In the 2-D convolution process, the input feature map (IFM) is processed row by row, with each row undergoing a 1-D convolution using a k×k kernel. Each row is subjected to multiply-accumulate (MAC) operations, and the intermediate results are stored in buffers. To maximize efficiency, the 2-D convolution with a k×k kernel is computed in a pipeline.
Each partial sum is the result of multiplying a window matrix and a kernel matrix. Each element in the *n* FIFO represents the result of multiplying the *n* row of the input feature map (IFM) window with the *n* row of the kernel matrix. The first FIFO holds the result of a 1-D convolution between the first row of the IFM and the first row of the kernel matrix, while the last FIFO contains the partial sum resulting from the convolution of k rows of the IFM with the kernel matrix.

For example, in cycle n, one IFM is loaded into the processing element (PE) array. In the very next clock cycle, 9 MAC operations are performed, with each MAC contributing to its respective partial sum. In the subsequent cycles, these partial sums are gradually accumulated, with 9 MAC operations completed per cycle, as shown by the changes in the partial sums' states in the figure 6. It can be observed that the partial sums are progressively completed in the order of the IFM's processing sequence.

B. Convolutional Tranpose Layer

A transposed convolutional layer is an upsampling layer that generates the output feature map greater than the input feature map. It is similar to a deconvolutional layer. A deconvolutional layer reverses the layer to a standard convolutional layer. If the output of the standard convolution layer is deconvolved with the deconvolutional layer then the output will be the same as the original value, While in transposed convolutional value will not be the same, it can reverse to the same dimension.

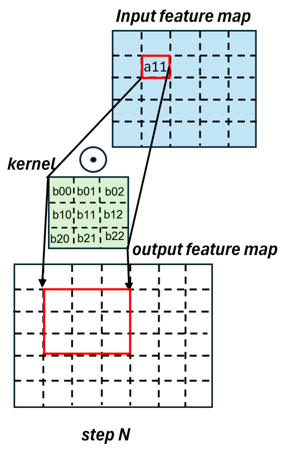


Fig. 10. Convolutional Tranpose architecture.

Transposed convolutional layers are used in a variety of tasks, including image generation, image super-resolution, and image segmentation. They are particularly useful for tasks that involve upsampling the input data, such as converting a low-resolution image to a high-resolution one or generating an image from a set of noise vectors.
Unlike the integration layer, which usually reduces the size of the input, the forward convolution layer increases the size of the output, the size of the output is calculated according to the following formula: The architecture hardware we propose to publish is quite similar to the tick layer with some different

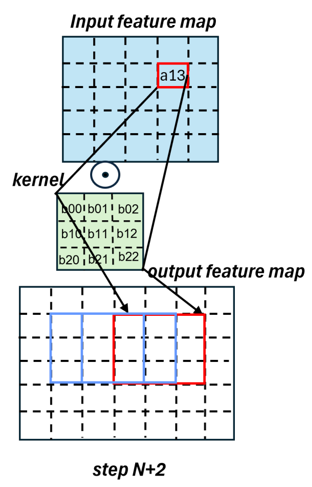


Fig. 11. Convolutional Tranpose architecture.

changes in data flow, the data flow is described in 2 figure 10 and 11 the architecture hardware is depicted in the figure below

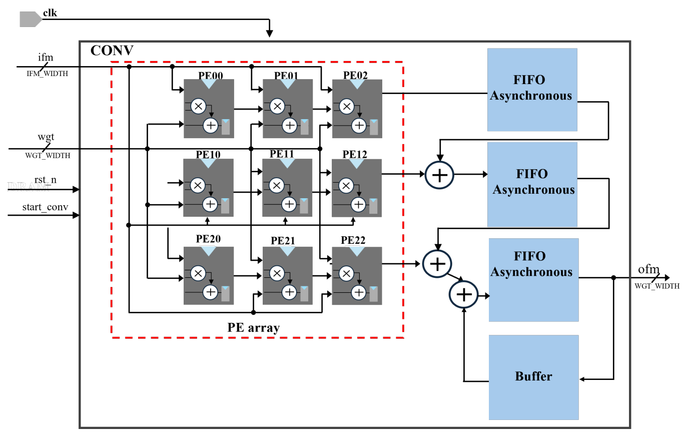


Fig. 12. Convolutional Tranpose architecture.

C. Gaussian Sampling Layer

Figure 14 shows the hardware architecture of the Gaussian sampling layer. This layer starts with a seed value that is input to a PRNG. The seed ensures the reproducibility of the random number sequence generated by the PRNG. The output from the PRNG is then transformed into a Gaussian random number (denoted by).

This transformation is necessary be cause PRNGs generate numbers that are uniformly distributed, not normally distributed. The input log variance log(2) is passed through an exponential function to get the standard deviation (). The generated Gaussian random number is then multiplied by the standard deviation (). This scales the random number by the desired spread of the distribution. The scaled random number is then added to the mean (μ), which effectively shifts the distribution so that it is centered around the mean. Finally, the output of the Gaussian Sampling Layer (Z) is a sample from a Gaussian distribution with the specified mean (μ) and standard deviation (). The proposed Gaussian sampling layer requires Gaussian random numbers generated by a Gaussian random number generator (GRNG). The traditional method for producing Gaussian random numbers requires two components: a uni form PRNG and a transform to the Gaussian distribution. This work first produces a uniform random number (step 1) and then transforms it into a Gaussian random number (step 2) in hardware. The Mersenne Twister is used in step 1, while the Box-Muller transform is used in step 2. Details are described in the following two subsections.

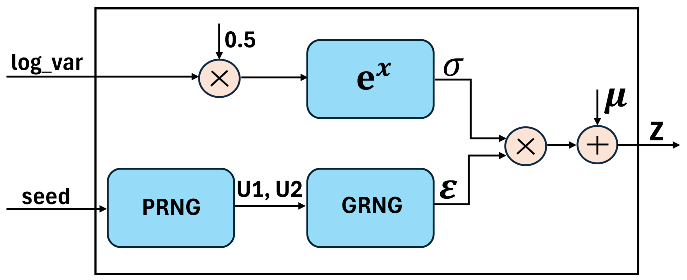


Fig. 13. Gaussion Sampling Layer architecture overview.

Pseudo Random Number Generator: In Variational Autoencoders (VAEs), pseudo-random number generators (PRNGs) are critical for enabling stochastic processes such as sampling from probability distributions during both training and inference phases. Specifically, VAEs rely on the reparameterization trick to sample from the latent space, where the latent variables are assumed to follow a Gaussian distribution. This sampling step requires the generation of high-quality random numbers to accurately model the latent variable's distribution.
The Mersenne Twister is a low latency, long period, hardware-efficient PRNG [31], [32]. The most commonly used Mersenne Twister, MT19937, has a period of 219937 1 and uses a 1024-depth array (state vector) to hold 624 word-sized (32-bit) elements. As a twisted GFSR (Generalized Feedback Shift Register), the Mersenne Twister updates the state vector by twisting recurrently

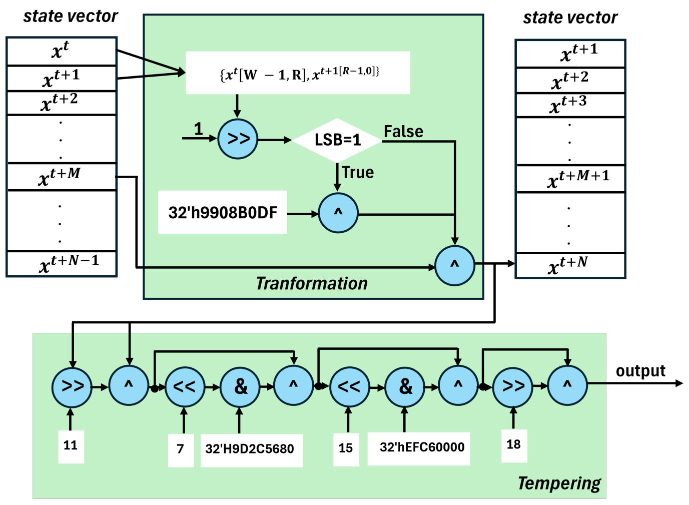


Fig. 14. Gaussion Sampling Layer architecture in hardware.

Step by step of MIT:

Step 0. Create bitmask for upper and lower bits

$$u \leftarrow \underbrace{1\ldots1}_{w-r}\underbrace{0\ldots0}_{r}, \quad \text{bit mask of upper } w-r \text{ bits,}$$

$$ll \leftarrow \underbrace{0\ldots0}_{w-r}\underbrace{1\ldots1}_{r}, \quad \text{bit mask of lower } r \text{ bits,}$$

$$a \leftarrow a_{w-1}a_{w-2}\ldots a_1a_0, \quad \text{the last row of matrix } A.$$

Step 1. Initialize the x array with seeds of nonzero values.

$$x[0], x[1], \ldots, x[n-1]$$

Step 2. Compute $(x_i^u \mid x_{i+1}^l)$, where the upper bits of $x[i]$ are concatenated with the lower bits of $x[i+1]$

$$y \leftarrow (x[i] \text{ AND } u) \text{ OR } (x[(i+1) \mod n] \text{ AND } ll)$$

Step 3. Calculate the next state

$$x[i] \leftarrow x[(i+m) \mod n] \oplus (y \gg 1) \oplus \begin{cases} 0 & \text{if LSB of } y = 0, \\ a & \text{if LSB of } y = 1 \end{cases}$$

Step 4. Multiply $x[i]$ by the tempering matrix T for better equidistribution

$$y \leftarrow x[i]$$

$$y \leftarrow y \oplus (y \gg u)$$

$$y \leftarrow y \oplus ((y \ll s) \text{ AND } b)$$

$$y \leftarrow y \oplus ((y \ll t) \text{ AND } c)$$

$$z \leftarrow y \oplus (y \gg l)$$

Output y.

Step 5. Increment i by 1

$$i \leftarrow (i+1) \mod n$$

Step 6. Repeat the process, Go to step 2.

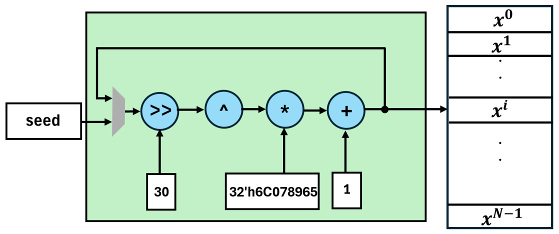


Fig. 15. The architecture of initial phase in Pseudo Random Number Generator(PRNG).

Table II
PARAMETERS OF 32-BIT MT19937

TABLE I
PARAMETERS OF 32-BIT MT19937

Parameter	Quantity
n	624
w	32
r	31
m	397
a	9908B0DF
u	11
s	7
t	15
l	18
b	9D2C5680
c	EFC60000

This paper discusses Mersenne Twister(MT) which is a PRNG and which satisfies all the requirements to be certified as a good PRNG. MT is proposed in 1997 by Makoto Matsumoto and Takuji Nishimura. It provides for fast generation of very high-quality pseudorandom numbers with a long period length which is chosen to be a Mersenne prime, high order of dimensional equidistribution, speed and reliability.

Gaussian Random Number Generator: To transform random numbers from a uniform distribution into numbers from a normal distribution, this work adopts the Box-Muller transform [33], a method for generating pairs of independent standard normally distributed (Gaussian) random numbers, given pairs of uniform random numbers. The transformation is mathematically represented as follows:

$$X_1 = (-2\log_e U_1)^{1/2} \cos 2\pi U_2$$

$$X_2 = (-2\log_e U_1)^{1/2} \sin 2\pi U_2$$

The process starts with two independent random numbers from a uniform distribution, U1 and U2, as shown in Figure 16. For U2, the logarithm (log) is taken and is then square-rooted (sqrt). This part of the transformation ensures that the variance of the normal distribution is correct. For U1, it is multiplied by 2 to convert the uniform random number into an angle, as the 2 represents the full circle in radians. It is then fed into sine (sin) and cosine (cos) functions. These functions are periodic and will convert the uniformly distributed U1 into two variables that follow the standard normal distribution.

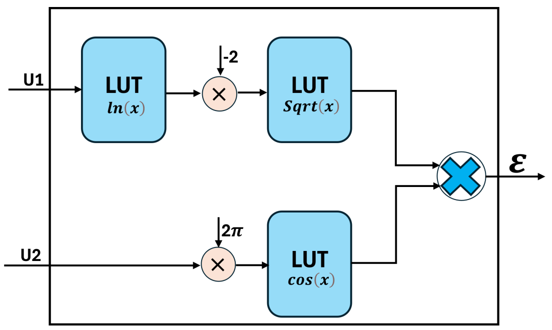


Fig. 16. The architecture of the Gaussian Random Number Generator (GRNG), which converts uniform random numbers generated by a Pseudo-Random Number Generator (PRNG) into Gaussian-distributed random numbers. U1 and U2 are two independent uniform random numbers while X and Y are the generated pair of independent Gaussian random numbers.

D. Fully conected Layer

The Fully Connected Layer (FC Layer), also known as the Dense Layer, is one of the fundamental components of artificial neural networks. In this layer, each neuron in the output layer is connected to every neuron in the input layer, allowing the model to learn complex representations.
Assume we have:

- **Input**: A vector $\mathbf{x} \in \mathbb{R}^N$, where N is the number of input features.
- **Weights**: A matrix $\mathbf{W} \in \mathbb{R}^{M \times N}$, where M is the number of neurons in the output layer.
- **Bias**: A bias vector $\mathbf{b} \in \mathbb{R}^M$.
- **Activation function**: $f(\cdot)$, such as ReLU, Sigmoid, or Tanh.
- **Output**: A vector $\mathbf{y} \in \mathbb{R}^M$.

The computation formula for a Fully Connected layer is:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Where:

- $\mathbf{W}\mathbf{x}$ is the matrix multiplication between the weight matrix and the input vector.
- $\mathbf{b}$ is the bias vector that allows the model to learn nonlinear shifts.
- $f(\cdot)$ is the activation function that ensures the output falls within the desired range.

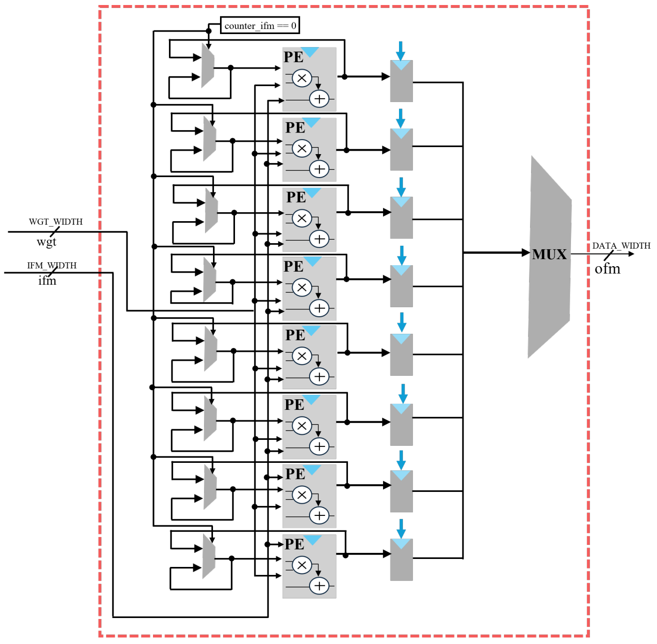


Fig. 17. The architecture of the fully connected layers.

In the FC layer, since the size of each kernel is equal to the size of the ifmap, implementing the FC layer is simpler than the CONV layer. To reduce latency for the FC layer, I will perform parallel computation with 8 kernels multiplying the ifmap simultaneously, utilizing a bandwidth of 8 weight pixels read in one clock cycle.

IV. EXPERIMENTAL SETUP AND RESULT

The design, training, and extraction of post-training parameters for the network were carried out on Google Colab with GPU support (Tesla T4), using the PyTorch library, and all network weights are of the float data type. The model used for this experimental task is based on the CVAE architecture, which has been fine-tuned to meet the requirements of the task, as described in Figure 4. The details of the model are shown in Table 5.3.

The model was trained using the Stochastic Gradient Descent (SGD) method with the following configuration parameters: Image dataset: MNIST, Number of training samples: 60,000 images, Learning rate: 0.005, Momentum: 0.8, Batch size: 64, Epochs: 100.

An independent test dataset, separate from the training set, consisting of 10,000 images containing digits from 0 to 9, is used for evaluation. The software testing is performed by a Python-based program.

The deployment architecture operates at a frequency of 100 MHz, utilizing the Zynq UltraScale+ ZCU104 FPGA board, achieving a maximum processing speed of 33.5 FPS with a power consumption of 4.8W.

Table IV provides a detailed summary of the resource utilization results for each convolution and fully connected layer of the CVAE.

Table III

MODEL ARCHITECTURE AND PARAMETERS.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 16, 16]	320
Conv2d-2	[-1, 64, 8, 8]	18,496
Conv2d-3	[-1, 128, 4, 4]	73,856
Flatten-4	[-1, 2048]	0
Linear-5	[-1, 2]	4,098
Linear-6	[-1, 2]	4,098
Sampling-7	[-1, 2]	0
Encoder-8	[[-1, 2], [-1, 2], [-1, 2]]	0
Linear-9	[-1, 2048]	6,144
ConvTranspose2d-10	[-1, 64, 8, 8]	73,792
ConvTranspose2d-11	[-1, 32, 16, 16]	18,464
ConvTranspose2d-12	[-1, 1, 32, 32]	289
Decoder-13	[-1, 1, 32, 32]	0
Total params:		199,557
Trainable params:		199,557
Non-trainable params:		0

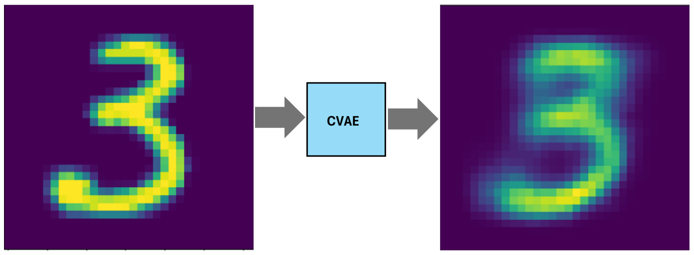


Fig. 18. The image illustrates the process of using a Convolutional Variational Autoencoder (CVAE) to reconstruct an image from the MNIST dataset. The left image represents the input digit ”3,” which is passed through the CVAE model (depicted by the box in the center). The right image is the output, showing the reconstructed version of the original digit, which retains the overall structure but exhibits slight variations due to the encoding and decoding process of the CVAE.

Table IV

SUMMARY OF TOTAL RESOURCE USAGE FOR THE ENTIRE NETWORK ON FPGA

Resource	Result		
	Use	Avaiable	Utilization
LUT	93727	230400	40.68%
LUTRAM	624	101760	0.61%
FF	125021	460800	27.13%
BRAM	162.5	312	52.08%
I/O	269	360	74.72%
BUFG	2	544	0.37%
DSP	183	1728	10.59%

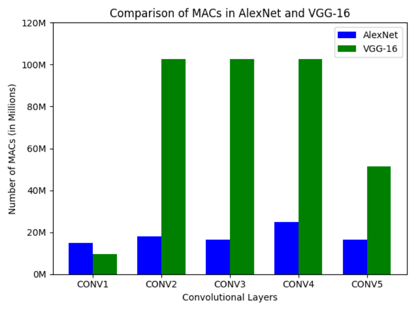


Fig. 20. Number of DRAM access in WS dataflow.

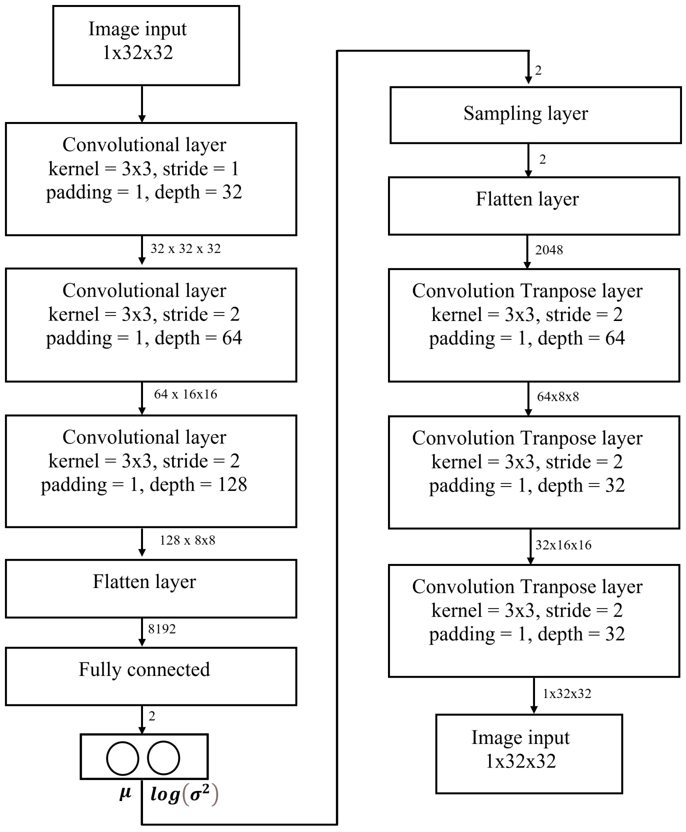


Fig. 19. Fine-tuned CVAE Network Architecture.

Table V

SUMMARY TABLE OF METRICS FOR FINE-TUNED ALEXNET NETWORK

Layer	Power (mW)	Num.of MAC	Num.of PE	LUT	FF	DSP
CONV1	333	11.71M	121	34303	73793	121
CONV2	84	37.32M	25	8166	18143	25
CONV3	26	12.46M	9	2301	4551	9
CONV4	26	24.92M	9	2301	4551	9
CONV5	26	12.46M	9	2301	4549	9
Sum (all)	**4.8W**	**104.65M**	**183**	**93727**	**125021**	**183**

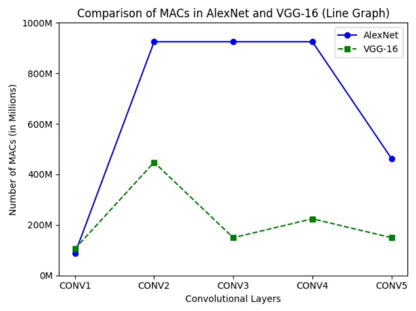


Fig. 21. Number of MAC in ALexNet and VGG-16 model.

Finally, the DRAM access speed per MAC is 0.0844 access/MAC of convolution layers measured on the AlexNet network and 0.111 access/MAC on the VGG-16 network and 0.095 in CVAE model.

V. CONCLUSION

In this study, we present a hardware architecture tailored for Convolutional Variational Autoencoders (CVAEs) with a focus on efficient computation, data movement reduction, and resources management. By implementing a weight-stationary data flow on a spatially arranged array of processing elements, we minimized data transfers and maximized filter reuse, leading to a significant reduction in energy requirements. This system processes the convolutional process layers at a rate of 33.5 frames per second, with DRAM access per multiply-and-accumulate (MAC) operation being 0.0844 for the AlexNet model, 0.111 for the VGG-16 model (batch size N=1) and in CVAEs model is 0.095 while the total power consumption of the entire network is 4.87 W.. The proposed CNN accelerator efficiently utilizes available FPGA resources, as demonstrated by the experimental results, which confirms it potential in deploying CVAEs on low-power embedded systems, paving the way for further advancements in energy-efficient AI hardware solutions.

REFERENCES

[1] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, Fellow, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” IEEE Journal of Solid-State Circuits, vol. 52, no. 1, pp. 127–138, 2017.

[2] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.W. Tai, “Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas,” in Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE, 2017, pp. 1–6.

[3] S. I. Venieris and C.-S. Bouganis, “fpgaconvnet: Automated mapping of convolutional neural networks on fpgas,” in Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2017, pp. 291–292.

[4] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, “Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks,” in Proceedings of the 2017 ACM/SIGDA International Symposium on FieldProgrammable Gate Arrays (FPGA), 2017, pp. 45–54.

[5] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song et al., “Going deeper with embedded fpga platform for convolutional neural network,” in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2016, pp. 26–35.

[6] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu, and T. Delbruck, ”Nullhop: A flexible convolutional neural network accelerator based on sparse repre.

[7] R. Zhao, X. Niu, and W. Luk, ”Automatic optimising cnn with depthwise separable convolution on fpga: (abstact only),” in Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2018, pp. 285–285.

[8] Xiao Dong, Xiaolei Zhu, and De Ma, ”Hardware Implementation of Softmax Function Based on Piecewise LUT” 2019 IEEE International Workshop on Future Computing (IWOFC).

[9] A. Vahdat and J. Kautz, “NVAE: A deep hierarchical variational autoen coder,” Advances in neural information processing systems, vol. 33, pp. 19667–19679, 2020.

[10] W. Xu, H. Sun, C. Deng, and Y. Tan, “Variational autoencoder for semi supervised text classification,” in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 31, no. 1, 2017.

[11] A. A. Pol, V. Berger, C. Germain, G. Cerminara, and M. Pierini, “Anomaly detection with conditional variational autoencoders,” in 18th IEEE international conference on machine learning and applications (ICMLA), 2019, pp. 1651–1657.

[12] L. Valente, L. Anzalone, M. Lorusso, and D. Bonacorsi, “Joint Variational Auto-Encoder for Anomaly Detection in High Energy Physics,” in International Symposium on Grids and Clouds (ISGC), vol. 19, no. 31, 2023.

[13] C. Coldwell, D. Conger, E. Goodell, B. Jacobson, B. Petersen, D. Spencer, M. Anderson, and M. Sgambati, “Machine learning 5g attack detection in programmable logic,” in IEEE Globecom Workshops (GC Wkshps), 2022, pp. 1365–1370.

[14] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi et al., “A Configurable Cloud-Scale DNN Processor for Real-Time AI,” in Proceedings of the 45th Annual International Symposium on Computer Architecture. IEEE Press, 2018, pp. 1–14.

[15] H. Fan, S. Liu, Z. Que, X. Niu, and W. Luk, “High-performance acceleration of 2-D and 3-D CNNs on FPGAs using static block floating point,” IEEE Transactions on Neural Networks and Learning Systems, vol. 34, no. 8, pp. 4473–4487, 2021.

[16] H. Nakahara, Z. Que, and W. Luk, “High-Throughput Convolutional Neural Network on an FPGA by Customized JPEG Compression,” in IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2020, pp. 1–9.

[17] Z. Que, E. Wang, U. Marikar, E. Moreno, J. Ngadiuba, H. Javed, B. Borzyszkowski, T. Aarrestad, V. Loncar, S. Summers et al., “Accel erating recurrent neural networks for gravitational wave experiments,” in IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2021, pp. 117–124.

[18] Z. Que, “Reconfigurable acceleration of Recurrent Neural Networks,” PhD dissertation, 2023. [15] Z. Que, M. Loo, H. Fan, M. Pierini, A. Tapper, and W. Luk, “Optimizing Graph Neural Networks for Jet Tagging in Particle Physics on FPGAs,” in 32nd International Conference on Field-Programmable Logic and Applications (FPL). IEEE, 2022, pp. 327–333.

[19] F. Wojcicki, Z. Que, A. D. Tapper, and W. Luk, “Accelerating Trans former Neural Networks on FPGAs for High Energy Physics Ex periments,” in 2022 International Conference on Field-Programmable Technology (ICFPT). IEEE, 2022, pp. 1–8.

[20] S. Denholm, H. Inoue, T. Takenaka, T. Becker, and W. Luk, “Low latency fpga acceleration of market data feed arbitration,” in 2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors. IEEE, 2014, pp. 36–40.

[21] L. Zhou, C. Cai, Y. Gao, S. Su, and J. Wu, “Variational autoencoder for low bit-rate image compression,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2018, pp. 2617–2620.

[22] K. Nakamura and H. Nakahara, “Optimizations of Ternary Generative Adversarial Networks,” in 2022 IEEE 52nd International Symposium on Multiple-Valued Logic (ISMVL). IEEE, 2022, pp. 158–163.