Low Latency Variational Autoencoder on FPGAs

Zhiqiang Que, Minghao Zhang, Hongxiang Fan, He Li, Ce Guo, and Wayne Luk, *Fellow, IEEE*

***Abstract*—Variational Autoencoders (VAEs) are at the forefront of generative model research, combining probabilistic theory with neural networks to learn intricate data structures and synthesize complex data. However, designs targeting VAEs are computationally intensive, often involving high latency that precludes real-time operations. This paper introduces a novel low-latency hardware pipeline on FPGAs for fully-stochastic VAE inference. We propose a custom Gaussian sampling layer and a layer-wise tailored pipeline architecture which, for the first time in accelerating VAEs, are optimized through High-Level Synthesis (HLS). Evaluation results show that our VAE design is respectively 82 times and 208 times faster than CPU and GPU implementations. When compared with a state-of-the-art FPGA-based autoencoder design for anomaly detection, our VAE design is 61 times faster with the same model accuracy, which shows that our approach contributes to high performance and low latency FPGA-based VAE systems.**

I. Introduction

Variational Autoencoders (VAEs) are powerful generative models that combine probabilistic and neural network approaches to learn and generate complex data. The foundational concept behind VAEs is their ability to infer and distill the latent distribution that underlies the data, which facilitates the generation of new, representative samples. This makes VAEs a potent tool for generative tasks, allowing them to create and reproduce the intricate structures within data. VAEs have been used in applications such as image generation [1], natural language processing [2], and anomaly detection [3], [4].

Deep Neural Network (DNN) models, including VAEs, while powerful, face limitations in real-world applications due to their inference times, which can range from tens to hundreds of milliseconds [5]. VAEs, as a specialized subset of DNNs, are designed for generative tasks. Their unique architecture, which includes an encoder-decoder structure, and a distinct learning objective to effectively model a probabilistic latent space, distinguish them from other DNNs. To address the inference time issues, FPGAs have been used to speed up the inference of DNN [6]–[8], which offer advantages of reduced latency and lower power consumption compared to CPUs or GPUs, making them a viable solution for enhancing DNN deployment efficiency.

However, existing research on accelerating VAEs on FPGAs is limited. A notable recent contribution [9] involves a low-latency VAE network on FPGAs to detect outliers in particle collision signals. Although their full VAE model achieves best model accuracy in all the designs, they avoid the complexity of Gaussian sampling in hardware and use only the encoder part in hardware. There is another FPGA-based VAE design [5] for attack detection to secure 5G communication, where ultra-low latency is essential. In the design, Gaussian random numbers are generated in large batches on CPUs and then fed into the FPGA, resulting in increased latency. More details can be found in Section II-B on related work

Z. Que, M. Zhang and C. Guo are with Imperial College London, UK. E-mail:{z.que, minghao.zhang17, c.guo}@imperial.ac.uk. H. Fan is with Samsung AI Center & University of Cambridge Cambridge, UK. H. Li is with Southeast University, China. W. Luk is with Imperial College London, UK. E-mail:w.luk@imperial.ac.uk

As explained above, research into FPGA-based VAE acceleration remains scarce. Some existing studies avoid full VAE implementations and adopt simplified versions that omit critical steps like Gaussian sampling. This paper introduces a novel low-latency hardware pipeline on FPGAs for fully-stochastic VAE inference. In this work, a custom layer of random sampling from a Gaussian distribution in the latent space is proposed and designed to enable full stochastic processing in hardware for low latency. A Pseudo Random Number Generator (PRNG) is used to ensure the reproducibility of the generated random number sequence. Moreover, our approach splits the entire VAE into several sub-layers and adopts a layer-wise tailor-made pipeline architecture with independently optimized components in each layer of the VAE using High-Level Synthesis (HLS). The layer-wise pipeline architecture has been used to speedup Convolutional Neural Networks (CNNs) [10]–[12], Recurrent Neural Networks (RNNs) [13], [14], Graph Neural Networks (GNNs) [15], [16] and transformer neural networks [17], but it has not been used in accelerating VAEs.

To realize the above tailor-made pipeline architecture, we propose novel code transformation to control design unrolling due to limited hardware resources on a given FPGA. It restructures the HLS-based VAE inference design into a perfect loop using a Finite-State Machine (FSM) based structure to improve the overall latency and initiation interval, enabling real-time operation within the bounds of limited hardware resources. This work uses VAE-based anomaly detection algorithm for detecting gravitational waves as an end-to-end application. Improving both accuracy and speed of VAEs are challenging, as higher accuracy usually requires a more complicated model and longer processing time. Accelerating VAE inference using FPGA-based accelerators would enable sophisticated algorithms, such as anomaly detection, to run in real time on input data streams, leading to improved model accuracy and low processing latency with a fast response.

To the best of our knowledge, this work is the first to propose a low latency hardware pipeline design of VAEs with full stochastic processing in hardware, enabling sophisticated real-time analysis on streaming data. Our contributions in this paper are as follows:

1) A low latency hardware architecture of VAEs with full Gaussian sampling in hardware and a layer-wise tailored pipeline architecture.
2) An HLS-based code transformation which restructures the VAE inference into a perfect loop using an FSM-

based structure to achieve low latency.

3) A comprehensive evaluation of the proposed method and hardware architecture.

While the design and parameters of the VAE are tailored for anomaly detection in gravitational wave observation, the optimizations for low latency would benefit many other applications, especially those requiring real-time response such as VAE-based Learned Image Compression (LIC) system [18], adaptive radiotherapy [19] and electronic trading [20].

The rest of this paper is as follows: Section II details the foundational concepts behind VAEs and the related work. Section III describes the VAE design, including the encoder-decoder structure, Gaussian Sampling Layer, Pseudo Random Number Generator, Gaussian Random Number Generator, Quantization of the GRNG, Layer-wise Architecture, and Parallelization. Section IV evaluates the performance and scalability of the proposed VAEs. We discuss related work in Section II-B and conclude in Section V.

II. Background

A. Variational Autoencoder

A conventional Autoencoder contains an encoder and a decoder, as shown in Fig. 1 (left). The encoder encrypts inputs into a compressed representation (latent layer), while the decoder tries to reconstruct the compressed representation back to the original inputs [21]. 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 [21], [22]. To address this issue, VAE is proposed in Ref. [22] 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, shown Fig. 1 (right). This Stochastic process improves the design robustness and forces the latent vectors to encode meaningful representations. The latent vector Z has a Gaussian distribution:

$$Z = \mu + (\sigma \odot \epsilon) \tag{1}$$

where μ and σ are mean and standard deviation of the input data. $\odot$ denotes element-wise multiplication. ϵ is a random vector sampled from a standard Gaussian distribution. This equation is referred as Gaussian sampling.

VAEs calculate the joint probability distribution P(X,Z) for input X and latent variable Z. Joint probability $P(X = x_1, Z = z_1)$ is the probability of $X = x_1$ and $Z = z_1$ happening at the same time. Joint probability distribution $P(X, Z)$ refers to all the possible value pairs of X and Z; from Bayes Theorem, it can be calculated by:

$$P(X, Z) = P(X|Z)P(Z) \tag{2}$$

$P(X|Z)$ is the probability distribution of X given the value of Z. P(Z) is the probability distribution of latent variable Z and is defined to be a standard Gaussian.

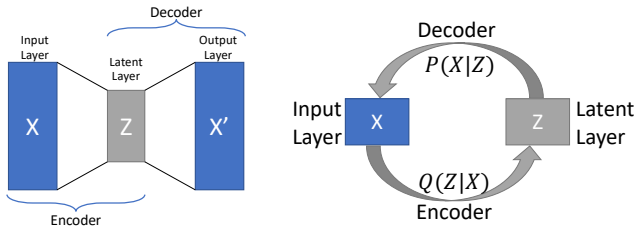


Fig. 1. An AE (left) maps inputs to a fixed latent representation and then decodes it back; a VAE (right) maps inputs to a probability distribution which samples an element and then decodes it back.

The loss function of the VAE combines the reconstruction loss (ELBO, Evidence Lower Bound) and KL (Kullback-Leibler) loss:

$$\begin{aligned} Loss(X;\theta,\lambda) &= ELBO(X;\theta,\lambda) + D_{KL}[Q(Z|X)||P(Z)] \\ &= \frac{1}{L}\sum_{l=1}^{L} \log P_\theta(X|Z_l) + \frac{1}{2}\sum_{j=1}^{J} 1 - \mu_j^2 - \sigma_j^2 + \log \sigma_j^2 \end{aligned} \tag{3}$$

The reconstruction loss calculates the difference between input X and output X'. $P(X|Z)$ is equivalent to $P(X|X')$, as X' is constructed from latent variable Z. The KL loss calculates the distance between variational posterior distribution $Q(Z|X)$ and the output distribution $P(Z)$. $Q(Z|X)$ is an approximation of the posterior distribution $P(Z|X)$, which is the probability distribution of Z given evidence X. θ and λ are hyperparameters to reduce the loss.

B. Related Work

Existing research on accelerating VAEs on FPGAs is scarce. A low-latency FPGA-based VAE [9] is designed to detect particle collision signals in the massive data from the Large Hadron Collider (LHC) in the Level-1 Trigger (L1T) system. To minimize latency, their approach omits Gaussian random number generation, which is typically a crucial step in VAEs. Instead, KL divergence D_{KL} and z-score R_Z (defined as the summed ratio of squared mean and squared standard deviation) are used to fine-tune a custom parameter (β), which ranges between 0 and 1. This parameter helps balance the reconstruction loss (ELBO) and KL loss. They achieve nice accuracy in multiple collision scenarios (AUC $> 80\%$) but the proposed VAE is incomplete, as it lacks the capability to reconstruct the input. In contrast, our work implements a full VAE with Gaussian random number generation on FPGAs while still maintaining low latency.

Another low-latency FPGA-based VAE design [5] focuses on attack detection for secure 5G communication. In this design, normally distributed random numbers are generated in large batches on CPUs and then fed into the FPGA as a secondary input, resulting in increased latency. The latency achieved by their design is several hundred microseconds, significantly higher than our system; it fails to meet the deployment requirements for scientific applications such as particle detection in LHC [9]. The latency introduced by random number generation on CPUs and data transfer can significantly hinder real-time processing capabilities, which is critical for

applications requiring fast response times. By implementing Gaussian sampling directly on the FPGA, we eliminate the need for data transfer from the CPU, thereby streamlining the inference process and enhancing the efficiency of the design.

VAEs also have been used in a fundamental framework in end-to-end learning-based image coding schemes [23], [24]. More recently, the F-LIC framework [18] leverages an FPGA-based Learned Image Compression (LIC) system with a Fine-grained Pipeline, utilizing the VAE framework. The low-latency hardware architecture of VAE developed in this work has the potential to be adapted to support such systems. Exploring this possibility is a promising direction for our future work.

Generative Adversarial Networks (GANs) are another widely recognized type of generative network. Similar to VAEs, GANs comprise two principal components: a generator and a discriminator. These components work together to learn the distribution of input data, enabling the generation of new, synthetic data. There has been significant research on accelerating GANs on FPGAs, as highlighted in these studies [25]–[30]. Despite the advancements in GANs, our current research is specifically concentrated on enhancing the efficiency of VAEs on FPGAs.

III. Design, Optimizations and Implementations

A. The overview of the VAE design

Figure 2 shows an overview of the VAE model used in this work. It consists of three main components: the encoder and the decoder, with a Gaussian sampling layer in between that connects the two. The details of the Gaussian sampling layer is shown in Figure 3.

The encoder network processes input data through hidden layers, which vary based on the data type and desired model complexity. Dense layers, also known as Fully Connected (FC) layers, are generic and provide a strong baseline for many tasks. LSTM (Long Short-Term Memory) layers are specialized for sequential data, capturing long-term dependencies in time series or text. CNN layers excel with spatial data, such as images, by leveraging their structure to reduce the number of parameters and improve efficiency. Each type of layer brings distinct advantages to the encoder network, optimizing it for different kinds of input data and learning tasks.

One branch of the encoder network's final layer outputs the mean (μ) of the latent space representation. Another branch outputs the log variance ($\log(\sigma^2)$). These parameters are learned for each data point and represent the encoder's belief about where the input data should be encoded in the latent space.

The mean and log variance are used in the Gaussian sampling layer, where a random sample Z is drawn. This is where the reparameterization trick comes into play, which allows for the gradient of the loss function to backpropagate through this stochastic process during training. The sampled latent vector Z is then fed into the decoder network that attempts to reconstruct the input data. The decoder network also consists of several hidden layers, which can be of the same types as the encoder network (Dense, LSTM, CNN).

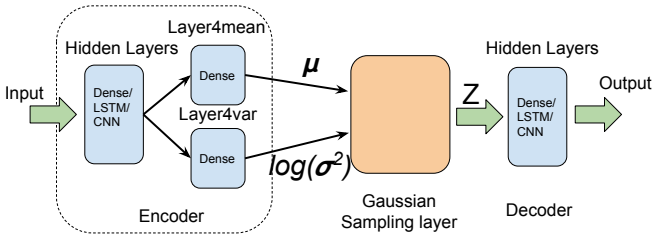


Fig. 2. The overall block diagram of the proposed design.

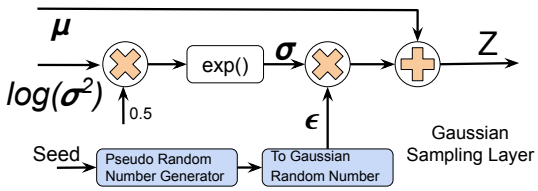


Fig. 3. The Gaussian sampling layer.

B. Gaussian Sampling layer

Figure 3 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 ϵ). This transformation is necessary because PRNGs generate numbers that are uniformly distributed, not normally distributed. The input log variance $\log(\sigma^2)$ is passed through an exponential function to get the standard deviation (σ). The generated Gaussian random number ϵ is then multiplied by the standard deviation (σ). 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 (σ).

The proposed Gaussian sampling layer requires Gaussian random numbers ϵ generated by a Gaussian random number generator (GRNG). The traditional method for producing Gaussian random numbers requires two components: a uniform 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.

C. Pseudo Random Number Generator

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 $2^{19937} - 1$ 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. Figure 4 shows the hardware archi-

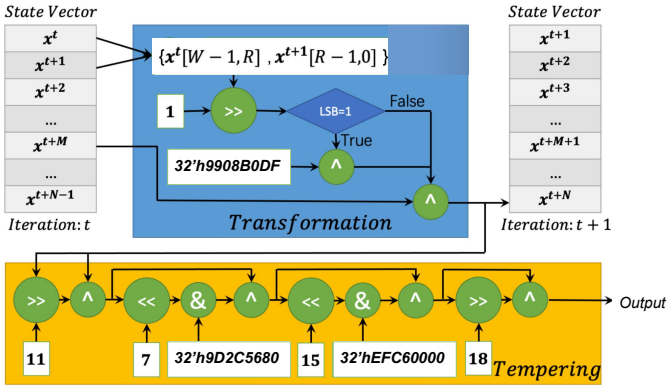


Fig. 4. The hardware architecture of the transformation and tempering stages in the Mersenne Twister PRNG Algorithm.

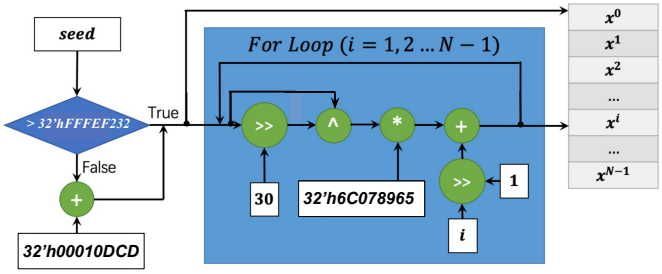


Fig. 5. The hardware architecture of the state vector initialization.

tecture of the MT19937 algorithm using HLS, specifically the transformation and tempering stages.

The transformation stage mainly includes a conditional bitwise operation based on the Least Significant Bit (LSB) of the right-shifted value of the concatenation of the (32-R) most significant bits of x_t and the R least significant bits of x_{t+1}. If the LSB is 1, the result of the right-shift is combined (using a bitwise XOR operation, indicated by the symbol $\wedge$.) with a constant. The result of the transformation is then used to update the state vector, creating the state vector for the next iteration (t+1).

The output of the PRNG is not taken directly from the state vector but is instead 'tempered' to improve the statistical properties of the output. This process involves several bitwise operations, such as bitwise left shift and bitwise AND operations as well as bitwise XOR, with different parts of the internal state being combined with constants to produce a tempered value. The result of the tempering process is the output of the PRNG for that iteration.

The initialization step shown in Figure 5 is crucial for the Mersenne Twister, as the quality of the pseudorandom number sequence it generates depends on the initial state vector being properly seeded and diversified, ensuring a high degree of randomness and uniform distribution in the generated sequence.

During each iteration, three words are read from position 0, 1, and M (called bias) and one word is written at position N. However, reading three values from one array causes difficulty in running multiple reads concurrently. Therefore, this work separates the 1024-depth state vector into two 512-depth arrays (even and odd), which doubles the performance with little increase in hardware resource usage.

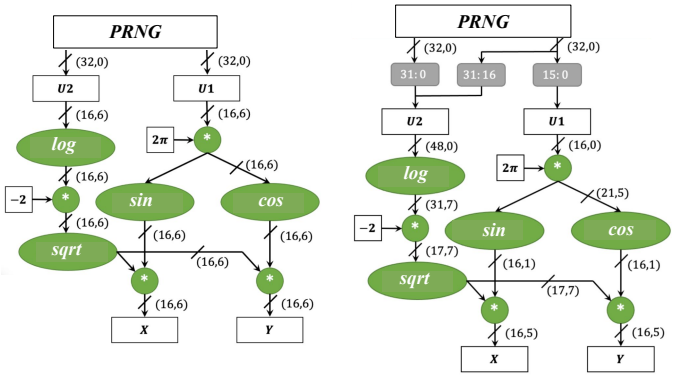


Fig. 6. The architecture of the Gaussian Random Number Generator (GRNG), which converts uniform random numbers generated by a Pseudo-Random Number Generator (PRNG) into Gaussian-distributed random numbers. U1 and U2 are two independent uniform random numbers while X and Y are the generated pair of independent Gaussian random numbers. The left diagram shows the GRNG implemented with a uniform 16-bit precision, while the right diagram illustrates a mixed-precision GRNG for optimized performance.

D. 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 process starts with two independent random numbers from a uniform distribution, U1 and U2, as shown in Figure 6. 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 2π to convert the uniform random number into an angle, as the 2π 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. All these functions are implemented by using HLS math library.

E. Quantization of the GRNG

The original implementation is based an unified 32-bit floating-point representation for all the variables in GRNG. Although floating-point numbers are accurate and flexible, they are less efficient than fixed-point numbers when implemented on FPGAs. To improve the hardware efficiecny, quantization is applied. Figure 6(Left) shows a fixed-point GRNG with unified 16-bit width, including 6 integer bits. However, setting all variables to a unified 16-bit representation results in low model accuracy since some functions/modules may require more integer bits while others may need more fractional bits.

To improve the model accuracy and maintain the low latency, a mixed-precision is proposed, as shown in Figure 6(Right). We concatenate and split the two 32-bit random numbers into a 48-bit variable and a 16-bit variable, which have the $[0, 1-2^{-48}]$ and $[0, 1-2^{-16}]$ respectively. The 48-bit variable is used for logarithm (log) while the short one is used for sin and cos. The output of logarithm function requires 7 bits for the integer as the output is in the range of $(0, 66.5421]$. The lower bound is determined by $-2ln(1-2^{-48})$ while the upper bound is calculated from $-2ln(2^{-48})$. The 16-bit

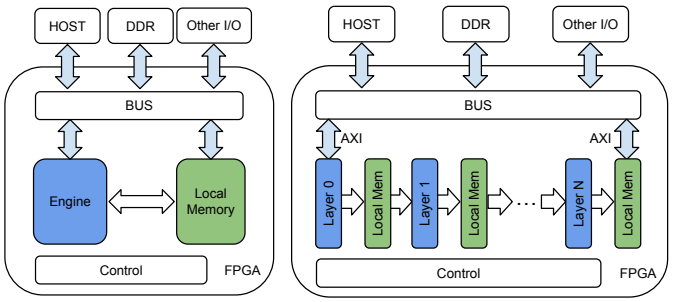


Fig. 7. Comparison of architectures. (a) A single engine hardware architecture with a single computation engine capable of processing multiple layers in an DNN model. (b) A stream-based layer-wise pipeline hardware architecture with several custom engines, each processing a whole layer or even multiple layers in an DNN model.

random number (U1) is passed to the sin and cos functions. The bit-width of U1 is much lower than U2 because the output range of sin and cos is $(0, 1]$, much smaller than the output range of log, which is $(0, \infty)$. Only 1-bit is reserved for the integer of the output for sin and cos.

F. Layer-wise Architecture

Many existing neural network accelerators on FPGAs are designed using a single engine architecture like GPUs, as shown in the Figure 7(a). It employs a single computational engine with multiple Processing Elements (PEs) to process one sub-layer (block) or an entire layer at a time, such that the entire DNN is processed by repeatedly running the engine [34], [35]. This method, while straightforward, forces each network layer to adapt to a uniform level of parallelism since there is only one single engine, which limits the flexibility and fails to leverage the full customizability potential of FPGAs [14].

This work splits the whole VAE into several sub-layers and adopts a stream-based layer-wise tailor-made pipeline hardware architecture. Unlike the single engine architecture, this architecture allocates each layer or a few cascaded layers to a dedicated hardware engine, and these engines are connected together to form a seamless pipeline, as shown in Figure 7(b). This architecture reduces off-chip memory access by retaining data on-chip, thereby reducing latency. In addition, it enables dedicated and layer-specific optimizations for each layer in VAEs since each layer is implemented independently in this architecture. Specifically, this work maps each layer of the VAE shown in Figure 2 on-chip and performs the computation for different layers on their own hardware unit. It accepts AXI bus based streamed data and outputs the results also via AXI bus. Inspiring by FPGA-based accelerators for low-precision quantized neural networks or binarized neural networks, such as those created using FINN [36] and HLS4ML [37], [38] frameworks, this works adopts a tailored accelerator design strategy aimed at using as many hardware resources as possible to achieve low latency. This work fully flattens computations within each component and implements each operation physically on-chip using dedicated hardware circuits, leading to significant low latency and high throughput. For example, all multiplication operations are conducted in parallel, utilizing the maximum possible number of multipliers on hardware.

Algorithm 1: The pseudocode of a VAE inference.

```
1 Function VAE_Inference():
2     #pragma HLS PIPELINE
3     VAE_Encoder;
4     for j = 0 to N_S − 1 do
5         #pragma HLS PIPELINE
6         VAE_Gaussian;
7         VAE_Decoder;
8     end
9 End Function
```

G. Parallelization

Algorithm 1 illustrates the pseudocode of a typical VAE inference design. The function VAE_Encoder performs the encoder network and outputs a statistical distribution with a mean and log variance. The function VAE_Gaussian performs the Gaussian sampling and outputs the sampled latent vector Z which is then fed to the decoder, VAE_Decoder. The N_S denotes the number of samples.

To pipeline the whole VAE inference, one could set the #pragma of "PIPELINE" at line 2, but it leads to automatically unrolling all loops in the hierarchy below. As a result, this would completely unroll the inner loop at line 4, resulting in N_S hardware copies of VAE_Decoder and VAE_Gaussian, significantly consuming hardware resources. If the required hardware resources exceed the given budget, one must limit the number of instances of VAE_Decoder and VAE_Gaussian. For example, if the total budget can support only $\frac{N_S}{2}$ instances of VAE_Decoder and VAE_Gaussian, this loop can only be unrolled by a factor of $\frac{N_S}{2}$. However, using a #pragma of unrolling with a factor for the inner loop will not manage to reduce the number of copies since the "PIPELINE" at line 2 has priority and will force the full unrolling of the inner loop. Thus, one has to move the #pragma of "PIPELINE" to line 5 to only pipeline the inner loop, excluding pipelining the VAE_Encoder, which leads to a poor design with large latency as well as large initiation interval.

To address this issue, this work proposes a code transformation which transforms the VAE inference design into a perfect loop using a FSM-based structure with a target initiation interval (II_{new}), as shown in Algorithm 2. In Xilinx HLS, it is a perfect loop when only the innermost loop has loop body content and there is no logic specified between the loop statements and all the loop bounds are constant [39]. With this transformation, now all the HLS code can be run in pipeline while before the transformation the VAE_Encoder is not pipelined as described above, which increases the overall latency.

If the number of instances of VAE_Decoder and VAE_Gaussian that can be deployed under the given resource budget is N_D, then II_T equals $Ceiling(\frac{N_S-1}{N_D})$. The loop can now run with an initiation interval of one, but the equivalent initiation interval is the target initiation interval. With this code transformation, we can deploy as many instances as permitted by the hardware budget, thereby improving the design performance of the VAE and reducing the overall design latency. Such code transformation, which involves finding the

Algorithm 2: Code transformation with an FSM-based structure to improve the design latency and initiation interval.

```
Function VAE_Inference_New():
    for i = 0 to II_new do
        #pragma HLS PIPELINE
        // The loop II will be 1 but the equivalent II is the II_new
        switch (curr_state) do
            case 0 do
                // Encoder is used only once
                VAE_Encoder;
                for j = 0 to Ceiling((N_S-1)/II_new) do
                    VAE_Gaussian;
                    VAE_Decoder;
                end
                curr_state ++;
                break;
            end
            case 1 do
                .....
            end
            .....
            case (II_new - 1) do
                for
                 j = (Ceiling((N_S-1)/II_new) × (II_new - 1))
                 to (N_S - 1) do
                    VAE_Gaussian;
                    VAE_Decoder;
                end
                curr_state = 0;
                break;
            end
            Default: break;
        end
    end
End Function
```

optimal number of instances to enhance performance, can be automated [40].

IV. Evaluation and Analysis

This section presents the evaluation results of the VAE models across two generations of Xilinx FPGAs, demonstrating the scalability of the proposed optimizations for low latency VAEs with good hardware efficiency.

A. Experimental Setup

Events related to Gravitational Wave (GW) signal dataset are created by simulating GW production from compact binary coalescences using the PyCBC toolkit [41], which incorporates algorithms from LIGO's LAL Suite [42]. Signal events containing GWs were created overlaying simulated GWs, with the SEOBNRv4 Approximant, on top of detector noise. Noise events occur when there is no GW signal but only detector noise. The detector noise is generated at a specified Power Spectral Density (PSD) [42] to mimic normal detector background conditions using PyCBC [41]. This method of event generation does not consider glitches, blips, or other transient noise sources in the detector. Following generation, the data are whitened and band-passed, and are then normalized. Further details about this dataset are available in [21].

The dataset for training comprises 240k events, while the validation and test sets contain 60k and 50k events, respectively. The training is performed on a Nvidia GPU 2080Ti (CUDA 11.8) based on Tensorflow 2.6 framework. During each training iteration, both the reconstruction loss and the Kullback-Leibler loss are computed and minimized to update the model's weights and biases. An early stopping mechanism with a patience setting of 10 epochs is employed to enhance training efficiency. To study the performance and limitations of the proposed optimizations and the hardware architecture, the designs are implemented using Vivado HLS 20.1. Two different generations of Xilinx FPGAs, the ZYNQ 7045 and the U250, are evaluated. The operating frequencies are 142MHz on the ZYNQ 7045 and 200MHz on the U250.

TABLE I
Resource utilization on a Xilinx Zynq 7045 and U250 FPGAs.

Task	Z7045	LUT	FF	BRAM	DSP
	Available	218600	437200	1090	900
VAE	Used [↓]	12708	12929	9	175
($N_{vae} = 2$)	Utili. [%, ↓]	5.8%	3.0%	0.8%	19.4%
VAE	Used [↓]	62590	63817	53	860
($N_{vae} = 10$)	Utili. [%, ↓]	28.6%	14.6%	4.9%	95.6%
Task	**U250**	**LUT**	**FF**	**BRAM**	**DSP**
	Available	1728000	3456000	5376	12288
VAE	Used [↓]	188350	29496	62	1032
($N_{vae} = 12$)	Utili. [%, ↓]	10.9%	0.9%	1.2%	8.4%

B. Resource Utilization

Table I shows the resource utilization of the proposed VAE designs on Xilinx Zynq 7045 and U250 FPGAs with various parallelism parameters. The encoder is comprised of three FC layers. The initial FC layer has 64 neurons, providing output to the subsequent two FC layers, named Layer4mean and Layer4var, dedicated to producing the mean and log variance. Both the layers accept an input size of 64 and output a dimension of 1 to match the specified mean and log variance size in this design. Meanwhile, the decoder is designed with a single FC layer, mirroring the initial encoder layer with 64 neurons, ensuring a symmetrical architecture for effective decoding. The encoder contains 3 FC layers. The first FC layer contains 64 neurons and output to the other two FC layer which generates the mean and log variance. The decoder has a single FC layer with size of 64. The first design is a baseline which has two VAE units, including one GRNG unit. The second design involves 10 VAE units with 5 GRNG units. The big design consumes most of the available DSP resources with good utilization on ZYNQ 7045 FPGAs. It consumes 4.9× more DSP blocks than the first design, showing the good scalability of our optimization. With more VAE units, a larger FPGA can be selected, such as the U250 FPGA. Table I also

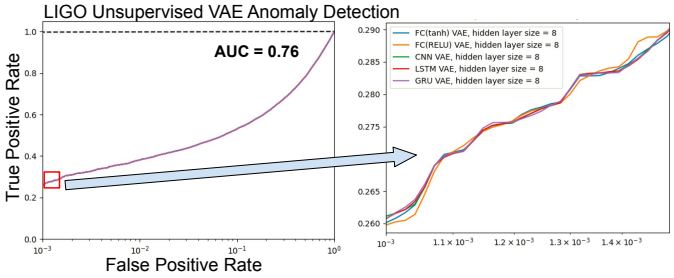


Fig. 8. AUCs and ROC curves for variational autoencoder with various types of neural network layers. The right figure shows a zoomed-in view of the red frame in the left figure. The X-axis denotes the False Positive Rate on a logarithmic scale, which is the ratio of incorrect positive predictions to the total number of negative cases

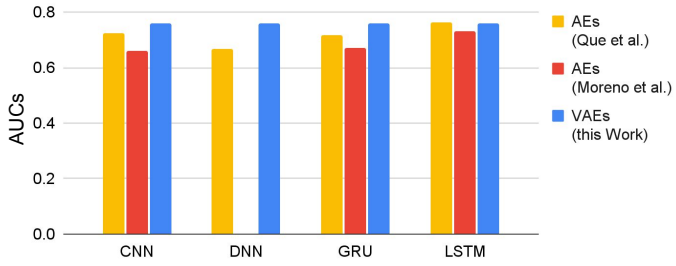


Fig. 9. Comparison of the AUCs of our design with other two designs (Que *et al.* [13] and Moreno *et al.* [21]).

shows a design which contains 12 VAE units and 6 GRNG units. This design consumes 1026 DSPs which are more than the given DSP numbers on a ZYNQ 7045 FPGA. By deploying more optimized VAE units on the given FPGA, we can achieve higher throughput, reduced latency, and improved hardware efficiency. In addition, both the encoder and decoder in the VAE model are quantized to 16-bit, which achieves the same model accuracy as the one using single-precision float-point representation.

C. Model Accuracy

To quantify the performance of the proposed variational autoencoders for anomaly detection in detecting gravitational waves, which are implemented by various neural networks, we use the AUC metric, or area under the Receiver Operating Characteristic (ROC) curve, as shown in Figure 8 and Figure 9, with higher AUC corresponding to better performance. AUC is a common metric for evaluating models as it is classification-threshold-invariant. The threshold for detecting an anomaly is determined by setting a False Positive Rate (FPR) on the noise events. The higher the threshold, the lower the FPR will be. Then the corresponding True Positive Rate (TPR) is calculated with this threshold on gravitational signal events.

We evaluate the AUCs of VAE networks with different hyperparameters, including the type of hidden layers and the size of hidden layer, using the same performance matrix (ROC and AUC) as [13] and [21]. We observe that with the proposed VAEs, the designs show good consistency and are not affected by the type of hidden layers, as shown in Fig. 8. The ROC curves (TPR vs FPR) of all the VAEs with various type of hidden layers are highly overlapped. The differences are only

TABLE II
COMPARISON OF TRUE-POSITIVE RATES (TPR) FOR GW DETECTION AT 10% AND 1% FALSE-POSITIVE RATES (FPR) WITH OTHER WORK. THE HIGHER, THE BETTER.

FPR	AE [21] (LSTM)	This work	AE [21] (GRU)	This work	AE [21] (CNN)	This work
0.1	0.461	0.531	0.348	0.531	0.302	0.531
0.01	0.304	0.381	0.186	0.381	0.120	0.381
0.001	-	0.261	-	0.261	-	0.261

TABLE III
THE LATENCY AND HARDWARE USAGE OF THE VAES WITH DIFFERENT GRNGS ON U250

Types	Latency	Initiation Interval	LUT	DSP	BRAM
Float32	24	1	16128	147	4
Unified 16-bit	16	1	15045	109	5
Mixed Precision	17	1	15570	113	12

visible when zooming in the low FPR part of the ROCs. In addition, the ROC curves of different sizes of hidden layers show the same feature. Furthermore, our VAEs achieve best AUCs for all the existing autoencoder-based designs for anomaly detection in detecting gravitational waves, as shown in Fig. 9.

In practice, the TPR values at particular FPR values are important for gravitational wave detection. An FPR of 0.01 corresponds to about 100 false alarms a day, while an FPR of 0.001 corresponds to 10 false alarms. The TPRs of the proposed VAEs are much higher than the conventional AE-based designs [21], as shown in Table II. This table demonstrates that the VAE (this work) consistently outperforms LSTM, GRU, and CNN based AE in gravitational wave detection across all false positive rates, indicating a more robust and generalized model with superior detection capabilities, particularly at lower false positive thresholds.

D. Performance of the Optimized GRNG

The most important part of the VAE is the Gaussian random number generator, GRNG. This section evaluates the quality of the presented GRNG.

The accuracy of the GRNG can be assessed by the high sigma test, which examines the number of sigmas that the Gaussian tail can accurately reaches. Each GRNG is used to generate 100,000,000 Gaussian random samples to conduct the high sigma test. The probability distribution histogram generated by the GRNGs is shown in blue, while the ideal Gaussian probability distribution function is shown in orange, as shown in Figure 10.

The GRNG with unified precision (Figure 6(left)) can reach an accuracy of 3.7 sigmas. The value of the Probability Density Function (PDF) at sigma = 3.7 is higher than the ideal curve, as the probability density of the Gaussian tail beyond 3.7 sigmas is added to it. The GRNG with mixed precision (Figure 6(right)) improves the accuracy to 5.5 sigma as shown in Figure 10(lower). It is lower than the 8.2 sigma achieved in [43]. [43] implements custom logarithm, square root and

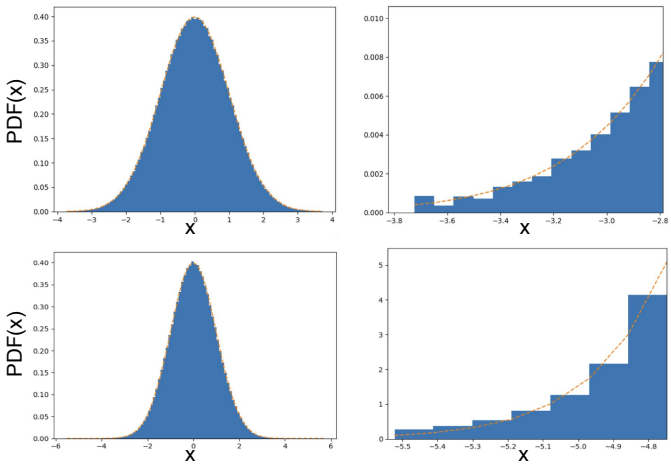


Fig. 10. Probability Density Function (PDF) of Gaussian Random Numbers. X-axis denotes the random number value and the Y-axis shows the probability density. The upper figures show the GRNG with unified precision while the lower ones illustrate the GRNG with mixed precision.

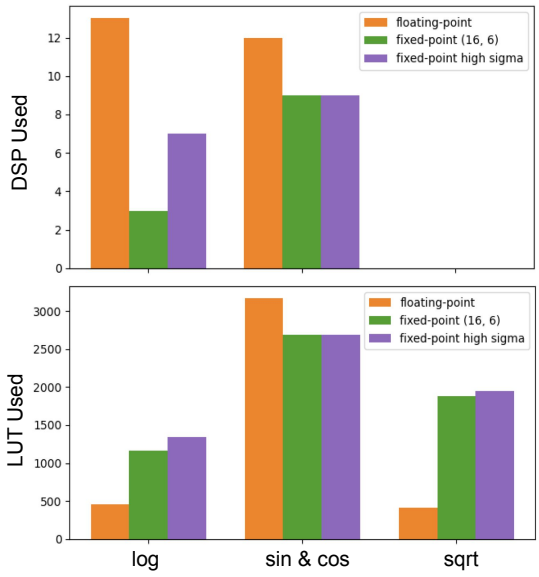


Fig. 11. The hardware usage of the main functions within the GRNG with different precision configurations.

sine/cosine functions, specific to the input precision. As these functions are approximated by polynomials, the coefficients of each term are also optimized in terms of their implementations. In this work, these functions are implemented using the Vivado HLS package due to flexibility. Also the GRNG in [43] only has a period of 2^{88}. More GRNGs in hardware can be found in this survey [44].

As shown in Table III, changing the floating-point GRNG to fixed-point reduces the inference latency by 50% and also reduces the DSP, FF, and LUT usage. The latency and initiation interval of the VAE excludes the initialization of the PRNG since the initialization only run once. The fixed-point high-sigma GRNG has consumed a little more BRAMs but still maintains the low latency. Moreover, we achieve an initiation interval of 1 for all the three cases.

The hardware usage of the main functions within the GRNG is shown in Figure 11. The *log* function used by floating-point GRNG consumes more DSPs to compensate for the low LUT usage. Compare to the custom *log* in the fixed-point 16-bit GRNG, the *log* in the high sigma GRNG increase DSP consumption from 3 to 7 because the input precision of hls::log rises from 16 bits to 48. In summary, the fixed-point high sigma GRNG reaches a high accuracy without the significant extra cost of hardware resources. The strategy of allocating more bits for *log* and fewer bits for sin/cos arranges the hardware resource efficiently and boosts the model performance.

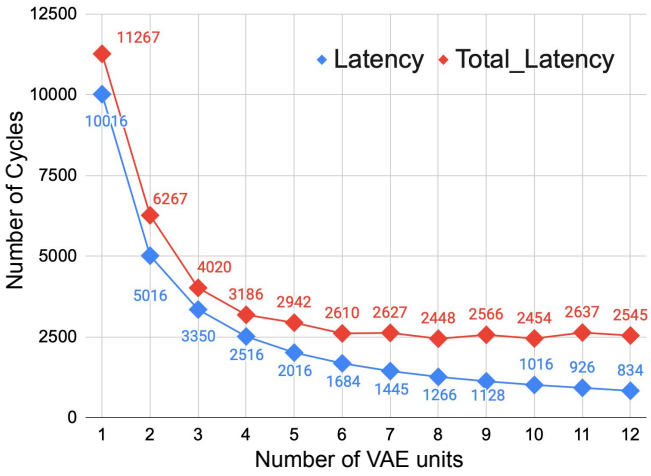


Fig. 12. The latency of the designs with multiple VAE units

E. Performance and Efficiency Comparison

Figure 12 shows performance improvement when deploying multiple VAE units in parallel for running 10,000 inferences on an U250 FPGA platform. The Latency (represented by blue diamonds) covers the VAE inference units after the PRNG initialization while the Total Latency (represented by red diamonds) also includes the latency of such initialization. As the number of VAE units increases from 1 to 12, both the Latency and the Total Latency decrease significantly. However, the decline in latency is larger when going from 1 to around 6 VAE units. Beyond 6 units, the decline flattens out, indicating that adding more VAE units does not significantly reduce the latency further. With a design frequency of 200MHz, where each cycle is 5ns, the Total Latency for 10,000 VAE inferences using a single VAE unit is 11,267 cycles as shown in Figure 12, resulting in an average latency of 5.6ns per VAE inference. As more VAE units are added, the Total Latency decreases, reaching a minimum of 2,448 cycles when there are 8 units. Moreover, the total latency is 17 cycles (85ns) for a single VAE unit with a single inference.

With a single VAE unit, the PRNG initialization contributes approximately 11.1% to the Total Latency. However, as the number of VAE units increases to 12, the PRNG initialization accounts for roughly 67% of the Total Latency. This indicates that the initialization latency becomes more significant as the system scales up. We leave this for our future work since it has little impact on the conclusion of this work.

The proposed VAE design, as a gravitational wave detection approach, uses significantly less hardware resource than pre-

TABLE IV
COMPARISON OF THE FPGA, CPU AND GPU DESIGNS

Platform	CPU E5-2620	GPU GeForce 2080Ti	FPGA U250 [13]	FPGA U250 This work [a]
Application Domain	Anomaly Detection			
Model	VAE	VAE	AE	VAE
AUC	0.76	0.76	0.76	0.76
Precision	Float32	Float32	Fixed 16	Fixed 16
DSP	-	-	2221	113
Average Late. (ns)	461.7	1162.5	343	5.6
Latency ratio with ours	82	208	61	1

[a] For a fair comparison, the design with a single VAE unit is used.

vious approaches such as LSTM-based Autoencoder (LSTM-AE) [13]. The LSTM-AE optimized in [13] consumes many more DSPs (2k to 9k) and LUTs (more than 449k), while our design with a single optimized VAE unit (including one GRNG) only requires 113 DSPs, however, our design is 61 times faster than [13] and achieves the same model accuracy with an AUC of 0.76, as shown in Table IV. Expanding our architecture to 12 VAE units scales up the resource demand modestly to 1,032 DSPs and 188k LUTs, but it delivers an additional 4.4 times speedup. This significant performance advantage shows the strength of our VAE-based design for real-time anomaly detection.

Power consumption is a vital metric for evaluating the efficiency of FPGA-based accelerators. For the proposed VAE design with 12 units (denoted as $N_{vae} = 12$) on the U250, the Xilinx Vivado tool reports on-chip power consumption of 5.755W, which includes a static power of 2.997W. Please note that this power consumption only reflects the energy consumed by the entire computational kernel in the U250's dynamic region. These numbers are reported by the Xilinx Vivado tool after the place & route stage.

To compare the performance of the proposed design on FPGA with other platforms, we implement the same VAE on Intel CPU and NVIDIA GPU. The CuDNN libraries are used for optimizing GPU performance. As GPUs always show advantages in multi-batch workload, we set the batch size to be 10k events on all hardware platforms for a fair comparison. Compared with the design running on CPU and GPU, our FPGA design is 82 and 208 times faster, as shown in Table IV.

Although Hardware implementation of Gaussian sampling does indeed consume more FPGA resources than conducting it on CPUs, and may potentially increase the critical path delay, affecting the maximum clock frequency, the benefits of reduced latency are considerable. This is particularly true for applications where inference speed is critical. In such cases, the trade-off is justified as the performance gain in latency outweighs the cost in resource consumption. Furthermore, the critical path delay can be managed and minimized through careful design and optimization techniques. Modern FPGA architectures offer substantial flexibility and resources that can be leveraged to mitigate the impact on clock frequency.

V. CONCLUSIONS AND FUTURE WORK

This paper presents a new low-latency hardware architecture for VAE inference. The novel aspects of this architecture include a custom Gaussian sampling layer and a layer-wise tailored pipeline architecture, both optimized for latency reduction. The proposed VAE design, when implemented in an FPGA, is shown to be 208 times faster than a GPU and 82 times than a CPU implementation. Our FPGA design is also 61 times faster than a recent FPGA-based autoencoder design for the same anomaly detection application with the same model accuracy. Future work includes extending the proposed VAEs beyond anomaly detection to application domains such as real-time image and video processing and other time-sensitive tasks. Additionally, we aim to automate the proposed optimizations using techniques such as meta-programming [45], applying it to a wide variety of applications targeting FPGAs and ASICs, particularly those that can benefit from the low-latency features of our architecture.

ACKNOWLEDGEMENT

The support of the United Kingdom EPSRC (grant numbers EP/V028251/1, EP/L016796/1, EP/N031768/1, EP/P010040/1, EP/S030069/1, and EP/X036006/1), CERN, AMD and SRC is gratefully acknowledged.

REFERENCES

[1] A. Vahdat and J. Kautz, "NVAE: A deep hierarchical variational autoencoder," *Advances in neural information processing systems*, vol. 33, pp. 19 667–19 679, 2020.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Z. Que, H. Nakahara, E. Nurvitadhi, H. Fan, C. Zeng, J. Meng, X. Niu, and W. Luk, "Optimizing Reconfigurable Recurrent Neural Networks," in *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 10–18.

[8] 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.

[9] E. Govorkova, E. Puljak, T. Aarrestad, T. James, V. Loncar, M. Pierini, A. A. Pol, N. Ghielmetti, M. Graczyk, S. Summers *et al.*, "Autoencoders on field-programmable gate arrays for real-time, unsupervised new physics detection at 40 MHz at the Large Hadron Collider," *Nature Machine Intelligence*, vol. 4, no. 2, pp. 154–161, 2022.

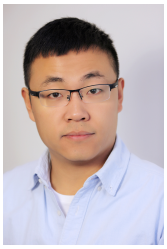
[10] S. Tridgell, M. Kumm, M. Hardieck, D. Boland, D. Moss, P. Zipf, and P. H. Leong, "Unrolling ternary neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 4, pp. 1–23, 2019.

[11] 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.
[12] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.
[13] Z. Que, E. Wang, U. Marikar, E. Moreno, J. Ngadiuba, H. Javed, B. Borzyszkowski, T. Aarrestad, V. Loncar, S. Summers *et al.*, "Accelerating recurrent neural networks for gravitational wave experiments," in *IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 117–124.
[14] 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.
[16] Z. Que, H. Fan, M. Loo, H. Li, M. Blott, M. Pierini, A. D. Tapper, and W. Luk, "LL-GNN: Low Latency Graph Neural Networks on FPGAs for High Energy Physics," *ACM Transactions on Embedded Computing Systems (TECS)*, 2024, (Accepted).
[17] F. Wojcicki, Z. Que, A. D. Tapper, and W. Luk, "Accelerating Transformer Neural Networks on FPGAs for High Energy Physics Experiments," in *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2022, pp. 1–8.
[18] H. Sun, Q. Yi, F. Lin, L. Yu, J. Katto, and M. Fujita, "F-LIC: FPGA-based Learned Image Compression with a Fine-grained Pipeline," in *IEEE Asian Solid-State Circuits Conference (A-SSCC)*, 2022, pp. 1–3.
[19] D. Thorwarth and D. A. Low, "Technical Challenges of Real-Time Adaptive MR-Guided Radiotherapy," *Frontiers in Oncology*, vol. 11, p. 332, 2021.
[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] E. A. Moreno, B. Borzyszkowski, M. Pierini, J.-R. Vlimant, and M. Spiropulu, "Source-agnostic gravitational-wave detection with recurrent autoencoders," *Machine Learning: Science and Technology*, 2022.
[22] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.
[23] 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.
[24] Z. Cheng, H. Sun, M. Takeuchi, and J. Katto, "Learned image compression with discretized gaussian mixture likelihoods and attention modules," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 7939–7948.
[25] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "FlexiGAN: An end-to-end solution for fpga acceleration of generative adversarial networks," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 65–72.
[26] S. Liu, C. Zeng, H. Fan, H.-C. Ng, J. Meng, Z. Que, X. Niu, and W. Luk, "Memory-efficient architecture for accelerating generative networks on FPGA," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 30–37.
[27] N. Shrivastava, M. A. Hanif, S. Mittal, S. R. Sarangi, and M. Shafique, "A survey of hardware architectures for generative adversarial networks," *Journal of Systems Architecture*, vol. 118, p. 102227, 2021.
[28] 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.
[29] A. S. Rakin, S. Angizi, Z. He, and D. Fan, "PIM-TGAN: A processing-in-memory accelerator for ternary generative adversarial networks," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 266–273.
[30] K. Nakamura and H. Nakahara, "A Consideration on Ternary Adversarial Generative Networks," in *2023 IEEE 53rd International Symposium on Multiple-Valued Logic (ISMVL)*. IEEE, 2023, pp. 1–6.
[31] H. Gabbard, C. Messenger, I. S. Heng, F. Tonolini, and R. Murray-Smith, "Bayesian parameter estimation using conditional variational autoencoders for gravitational-wave astronomy," *Nature Physics*, vol. 18, no. 1, pp. 112–117, 2022.
[32] M. Matsumoto and Y. Kurita, "Twisted GFSR generators," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 2, no. 3, pp. 179–194, 1992.
[33] G. E. Box and M. E. Muller, "A note on the generation of random normal deviates," *The annals of mathematical statistics*, vol. 29, no. 2, pp. 610–611, 1958.
[34] H. Fan, S. Liu, M. Ferianc, H.-C. Ng, Z. Que, S. Liu, X. Niu, and W. Luk, "A Real-Time Object Detection Accelerator with Compressed SSDLite on FPGA," in *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 14–21.
[35] Z. Que, H. Nakahara, E. Nurvitadhi, A. Boutros, H. Fan, C. Zeng, J. Meng, K. H. Tsoi, X. Niu, and W. Luk, "Recurrent neural networks with column-wise matrix–vector multiplication on FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 2, pp. 227–237, 2021.
[36] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "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.
[37] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.
[38] J. Ngadiuba, V. Loncar, M. Pierini, S. Summers, G. Di Guglielmo, J. Duarte, P. Harris, D. Rankin, S. Jindariani, M. Liu *et al.*, "Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 1, p. 015001, 2020.
[39] Xilinx, "Vivado design suite user guide, high-level synthesis; 2020," *UG902*.
[40] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance modeling and directives optimization for high-level synthesis on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 7, pp. 1428–1441, 2019.
[41] A. Nitz, I. Harry, D. Brown, C. M. Biwer, J. Willis, T. D. Canton, C. Capano, L. Pekowsky, T. Dent, A. R. Williamson, G. S. Davies, S. De, M. Cabero, B. Machenschalk, P. Kumar, S. Reyes, D. Macleod, F. Pannarale, dfinstad, T. Massinger, M. Tápai, L. Singer, S. Khan, S. Fairhurst, S. Kumar, A. Nielsen, SSingh087, shasvath, I. Dorrington, and B. U. V. Gadre, "gwastro/pycbc: Pycbc release v1.16.9," Aug. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3993665
[42] LIGO Scientific Collaboration, "LIGO Algorithm Library - LALSuite," free software (GPL), 2018.
[43] D.-U. Lee, J. D. Villasenor, W. Luk, and P. H. W. Leong, "A hardware Gaussian noise generator using the Box-Muller method and its error analysis," *IEEE transactions on computers*, vol. 55, no. 6, pp. 659–671, 2006.
[44] J. S. Malik and A. Hemani, "Gaussian random number generation: A survey on hardware architectures," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, pp. 1–37, 2016.
[45] Que, Zhiqiang and Liu, Shuo and Rognlien, Markus and Guo, Ce and Coutinho, Jose G. F. and Luk, Wayne, "MetaML: Automating Customizable Cross-Stage Design-Flow for Deep Learning Acceleration," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023, pp. 248–252.

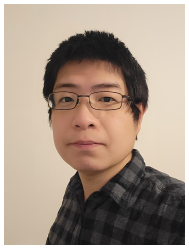
Zhiqiang Que is a research associate in the department of Computing, Imperial College London, UK. He received his B.S in Microelectronics and M.S in CS from Shanghai Jiao Tong University in 2008 and 2011, and the PhD from the Department of Computing, Imperial College London, U.K. in 2023. He worked on microarchitecture design and verification of ARM CPUs with Marvell Semiconductor (2011-2015) and low latency FPGA systems with CFFEX (2015-2018). His research interests include computer architectures, embedded systems, high-performance computing and computer-aided design tools for hardware design optimization.

Minghao Zhang received the master's degree from the Department of Computing, imperial College London UK in 2022.

Hongxiang Fan received the B.S. degree in electronic engineering from Tianjin University, Tianjin, China, in 2017, and the M.Res. and D.Phil. degrees from the Department of Computing, Imperial College London, London, U.K., in 2018 and 2022. He is currently a research scientist at Samsung AI Cambridge and an affiliated postdoctoral researcher at the University of Cambridge. His current research focuses on computer architecture, machine learning and quantum computing.



He Li He Li holds a full professorship and serves as the head of Circuits and Systems Department and the head of Heterogeneous Intelligent and Quantum Computing Lab in the School of Electronic Science and Engineering, Southeast University as a Professor. Before joining SEU, Dr. Li was a research assocaite for quantum information at the Univeristy of Cambridge, and a teaching staff at Trinity College Cambridge. Before joining Cambridge, He received the PhD degree at Imperial College London, 2020. He serves on technical programme committees of the top-tier EDA and reconfigurable computing conferences (DAC, IC- CAD, ICCD, FCCM, FPL, FPT, ASP-DAC, ASAP and SOCC), and the editorial board of Frontiers in Electronics. Dr. Li is the FPT'17 best paper presentation award recipient. He serves as organisation committee members for multiple IEEE/ACM international conferences.



Ce Guo Ce Guo is a Research Associate in Accelerator Computing in the Department of Computing at Imperial College London. His research focuses on developing efficient computing systems for large-scale temporal data analytics, agent-based simulation, and causal structural learning. Ce holds a master's degree in Artificial Intelligence and a PhD in Custom Computing, both from Imperial College London.

Wayne Luk received the M.A., M.Sc., and D.Phil. Degrees in Engineering and Computing Science from the University of Oxford, Oxford, U.K. He is a Professor of Computer Engineering with Imperial College London, London, U.K. He was a Visiting Professor with Stanford University, Stanford, CA, USA. His current research interests include theory and practice of customizing hardware and software for specific application domains, such as multimedia, networking, and finance.