Efficient Gaussian Random Number Generators in HLS4ML

Atharva Mattam

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington
2024

Committee:
Scott Hauck
Shih-Chieh Hsu

Program Authorized to Offer Degree:
Department of Electrical and Computer Engineering

University of Washington

Abstract

Efficient Gaussian Random Number Generators in HLS4ML

Atharva Mattam

Chair of the Supervisory Committee:

Scott Hauck

Department of Electrical and Computer Engineering

Efficient hardware implementation of neural networks, such as Variational Autoencoders (VAEs), often relies on FPGAs for their balance of performance and energy efficiency. VAEs require accurate Gaussian distributions for latent space sampling, but traditional methods like the Central Limit Theorem (CLT) are resource-intensive. The Multihat method combines combinational logic and CLT to achieve high tail accuracy with reduced hardware costs. This thesis discusses the Multihat method implemented using High-Level Synthesis (HLS), optimized for scalability and integrated into HLS4ML as a custom layer for FPGA deployment. Results show the Multihat GRNG generates statistically accurate Gaussian distributions, with improved resource efficiency and performance compared to CLT-based approaches.

Contents

1 Introduction **4**

2 Background **4**

2.1 Neural Networks 4

2.2 Autoencoders 6

2.3 Variational Autoencoders 8

2.4 Vivado HLS Flow 10

2.5 HLS4ML 12

3 Multihat **13**

3.1 Uniform Random Number Generator 14

3.2 Hat Generation 16

3.2.1 The Standard Hat 17

3.2.2 The Narrow Hat 18

3.2.3 The Wide Hat 19

3.2.4 The Tall and Short Hat 21

3.2.5 The "Multi" Hat 22

3.3 Central Limit Theorem 23

4 HLS Implementation **25**

5 Integration into HLS4ML **26**

6 Results **27**

6.1 Configurable Bitwidths 27

6.2 Statistical Properties 29

6.3 Resource Utilization and Performance 30

6.4 Comparing a Second Implementation 32

7 Applications **33**

8 Conclusion **35**

9 My Contributions **36**

1 Introduction

This research project explores the development and integration of a Gaussian random number generator (GRNG) using the Multihat method [16] into the HLS4ML framework [7] for use in Variational Autoencoders (VAEs). The motivation stems from the need to efficiently generate Gaussian distributions in hardware to support neural network models such as VAEs. VAEs use Gaussian distributions to sample latent space variables, which are key to generating and reconstructing data. Traditional methods for generating Gaussian distributions on hardware, such as the Central Limit Theorem (CLT), are resource-intensive, especially when targeting extreme tail accuracy. The Multihat method offers a resource-efficient alternative, combining several layers of transformation to approximate a Gaussian curve at a lower hardware cost, while maintaining high accuracy in the tails of the distribution.

The work discusses in detail the structure and components of the Multihat algorithm, including the uniform random number generator, the process of generating "hats" from uniform random distributions, and the use of the CLT to refine the accuracy of the final Gaussian distribution. The work also examines the challenges and advantages of integrating this GRNG into HLS4ML, a framework that automates the deployment of machine learning models on FPGAs.

2 Background

2.1 Neural Networks

As the name suggests, neural networks are inspired by the brain and mimics some behaviors of the human brain. Neural networks compose of several layers of interconnected data processing nodes, or neurons, which are typically organized

into one input layer, one output layer, and several hidden layers based on the complexity of the network and its goal. According to IBM [9], each individual node can be thought of as being its own linear regression model, composed of input data, weights, a bias, and an output.

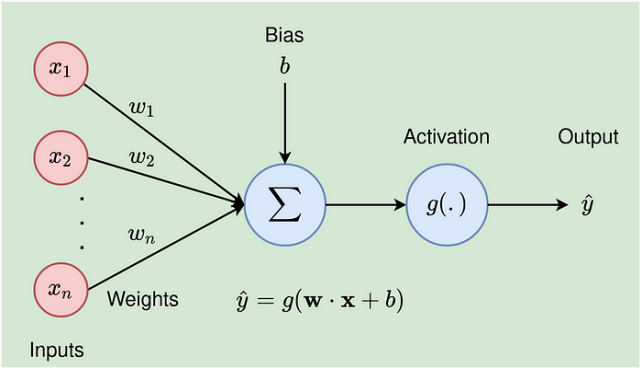


Figure 1: Example of a Neural Network [13]

Figure 1 shows the mathematical computation that takes place within each neuron [13]. Once the input to the neuron is established, each input is multiplied by its assigned weight. Inputs with larger magnitude weights typically contribute more significantly to the final output of the layer, helping determine the importance of a given neuron. Once the weighted inputs are calculated, a summation of all the weighted inputs takes place, after which, the bias value of the corresponding neuron is added to the final value. This result passes through an activation function, which determines the final output of the neuron. If the output surpasses a certain threshold, the neuron "fires", allowing the data to move to the next layer of the network. In this way, the output of one node becomes the input for the next. This sequential process of passing data from layer to layer is what characterizes the neural network as a feedforward network.

2.2 Autoencoders

Autoencoders aim to identify key components of an input (the latent variables) by compressing, or encoding, the input data through a central bottleneck layer, forcing the encoder to focus on extracting only the most critical information required to reconstruct, or decode, the original input. While different autoencoder types adjust pieces of their architecture to suit specific data types or objectives, they all share a core structure [2], which can be seen in figure 2.

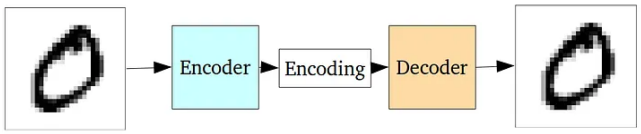


Figure 2: A Standard Autoencoder [20]

The encoder consists of layers that progressively reduce the dimensionality of the input data, forming a compressed representation. Typically, as the data moves through the encoder, each hidden layer contains fewer nodes than the previous one, resulting in a compression of the input data.

At the center of the network, the bottleneck, or "code," holds the most compressed version of the input and serves as the final output of the encoder as well as the input to the decoder. A key design challenge in autoencoders is determining the smallest set of features or dimensions needed to accurately reconstruct the input. This latent space representation, or code, is passed into the decoder.

The decoder is made up of layers with progressively more nodes, expanding or "decoding" the compressed data back into its original form. The final reconstructed output is then compared to the original input (referred to as the "ground truth") to evaluate the autoencoder's performance. The difference between the output and the original input is known as the reconstruction error.

There is one fundamental problem with autoencoders. The applications of standard autoencoders are fairly limited, because the latent space of standard encoders do not allow for easy interpolation. For example, when an autoencoder is trained on the MNIST dataset, visualizing the encodings from its latent space will reveal distinct clusters, as seen in figure 3. This is beneficial when the goal is to simply replicate the input images [20].

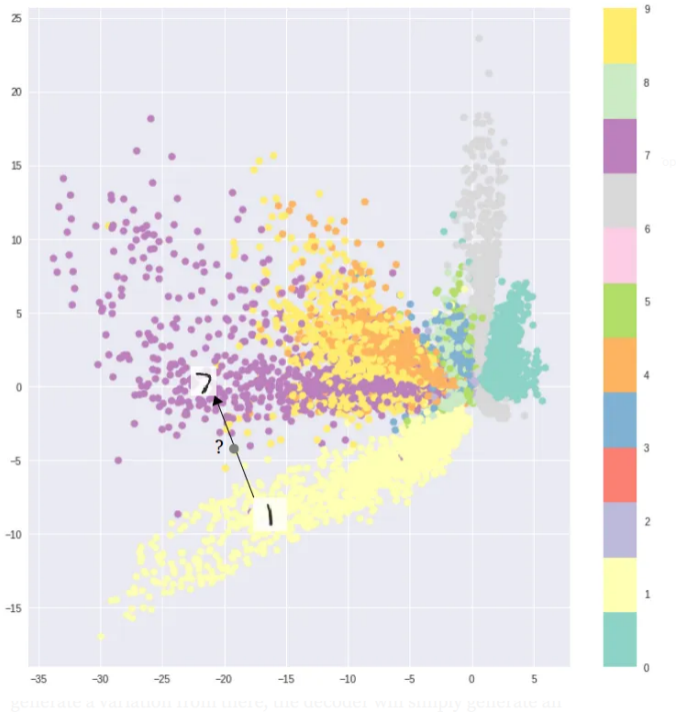


Figure 3: Latent Space Representation of an autoencoder trained on the MNIST dataset. [20]

However, when developing a generative model, the objective shifts. Rather than simply replicating input images, the focus is on randomly sampling from the latent space or generating variations of an image. To achieve this, the latent space must be continuous and smooth. Discontinuities in the latent space, such as gaps between clusters, create challenges. Samples or variations taken from

these untrained regions typically produce unrealistic outputs, as the decoder has not learned to generate meaningful data from these areas during training [20].

2.3 Variational Autoencoders

A variational autoencoder (VAE) provides a probabilistic manner for describing an observation in the latent space [11]. This means that, instead of mapping the output of an encoder to a single vector, the VAE's encoder maps its output to a distribution. The VAE's encoder achieves this by outputting two vectors: the mean μ and the standard deviation σ, which is represented by figure 4.

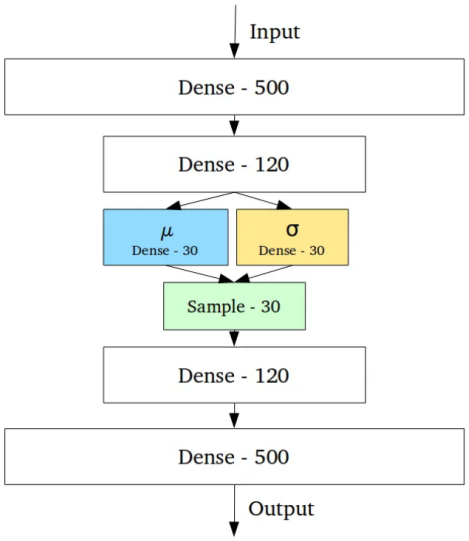


Figure 4: An Example Variational Autoencoder [20]

After obtaining the mean μ and the standard deviation σ, the VAE incorporates a key step: sampling from the latent distribution. Instead of directly using the encoded mean μ and standard deviation σ as inputs to the decoder, the VAE introduces a random variable, z, from a Gaussian distribution. This

sampling step, also outlined in figure 4, allows for the generation of a variety of potential outputs, even when encoding similar inputs.

However, there is one major problem with this architecture of a VAE. Training the VAE using backpropagation becomes impossible because calculating the gradient with respect to the random variable z is mathematically undefined. This is where the reparameterization trick comes into action. Instead of sampling directly from the Gaussian distribution, an additional random variable, ϵ, is introduced. The random variable ϵ is in charge of handling and creating a Gaussian distribution with a mean μ of 0 and a standard deviation σ of 1, typically notated as $\epsilon \sim N(0, 1)$. This simple yet powerful trick, when implemented in the VAE, can be described by the below equation:

$$z = \mu + \sigma * \epsilon \tag{1}$$

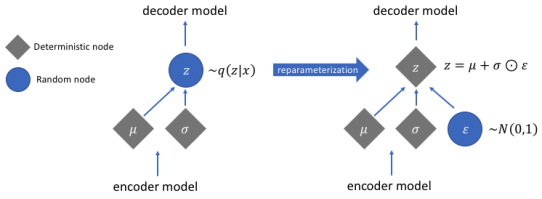


Figure 5: The Reparameterization Trick [10]

Equation 6 shows that the random variable z is calculated by first scaling the Gaussian distribution ϵ with the model's standard deviation σ. Then, the scaled distributing is shifted by the model's mean μ. This trick, as seen in figure 5, achieves the exact same functionality as before, with the crucial difference being that the model's mean μ and standard deviation σ are deterministic values [23], i.e, the mean and standard deviation do not change between runs when the input data is consistent between runs. Not only are the mean and standard deviation deterministic now, but the random variable ϵ can be controlled

separately. The reparameterization trick captures the essence of the original Gaussian distribution while maintaining the relationship between the mean and standard deviation defined by the encoder [23]. Finally, this trick enables the ability to backpropagate throughout the entire network, thus, allowing the VAE to be trained end-to-end using standard gradient-based optimization techniques like Adam [12].

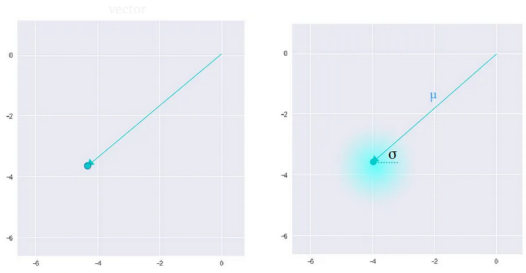


Figure 6: The latent space representation of an autoencoder vs a variational autoencoder [20]

This sampling mechanism is what distinguishes the VAE from traditional autoencoders. Figure 6 shows the difference in the latent space representation between an autoencoder and a variational autoencoder. From the figure, a standard autoencoder is directly encoded to specific coordinates, whereas a variational autoencoder contains a probability distribution around the encoding. It introduces an element of stochasticity, allowing the VAE to learn a more robust and continuous latent space, as well as enabling the generation of novel data points by sampling different latent variables.

A more in depth understanding of variational encoders can be found in Carl Doersch's "Tutorial on Variational Autoencoders." [5]

2.4 Vivado HLS Flow

HLS or High-Level Synthesis is a hardware design methodology that automates the process of converting high level programs into digital circuits rather than

using HDLs such as Verilog or VHDL. HLS allows for quicker and rapid prototyping compared to HDLs, by allowing designers to work in programming languages such as C/C++ or SystemC. HLS tools, then, synthesize the high level programs into appropriate RTL code to be deployed onto FPGAs. Vivado HLS is one such tool, developed by Xilinx, that eases the deployment of HLS programs onto Xilinx FPGAs.

Once an algorithm is written in C/C++ or SystemC, HLS translates its behavioral description into a hardware model that can be synthesized into digital logic. By focusing on functionality rather than hardware implementation details, HLS simplifies the traditionally complex RTL design process. This process allows designers to explore various micro-architectural options, such as partitioning resources, scheduling operations, or parallelizing tasks, without needing to manually adjust RTL code. Such flexibility is especially valuable in the early design phases, where rapidly evaluating different alternatives is key to finding the best solution.

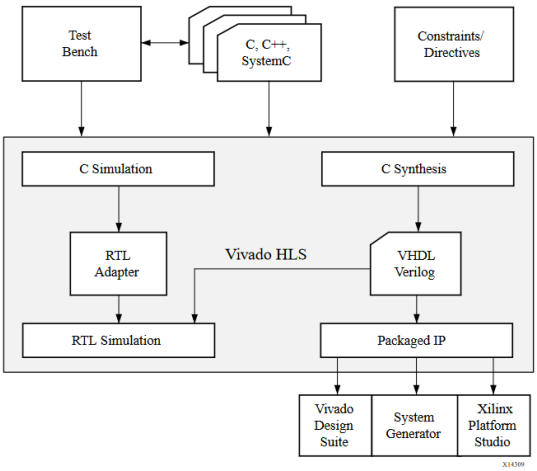


Figure 7: Vivado HLS Design Flow [24]

From Xilinx's User Guide for High-Level Synthesis [24], the Vivado HLS

design flow is as follows: first, compile, execute and debug the C algorithm. Second, Synthesize the C algorithm into an RTL implementation. Third, generate comprehensive reports and analyze the design. Fourth, verify the RTL implementation. Finally, package the RTL implementation into a selection of IP formats. The Vivado HLS tool requires a few inputs during the design flow. For the first step, the tool requires the C function written in C/C++, or SystemC which may contain subfunctions. Along with the C function, the tool also requires constraints with information about the clock and FPGA, the C testbench and other utility files, and optionally directives that can direct the tool to implement a specific behavior. After synthesis, the tool outputs the RTL implementation files in VHDL (IEEE 1076-2000) and Verilog (IEEE 1364-2001) (as of Vivado HLS 2019.2) [24] along with report files with information about area (LUTs, registers, BRAMs and DSPs), latency, and initiation interval. The C testbench is then used to verify the RTL output using C/RTL Cosimulation and ends with its own report files. Other tools in the Xilinx Design Flow can use the implementation files as IP blocks. Using logic synthesis, the packaged IP can be synthesized into a bitstream to be deployed on an FPGA.

2.5 HLS4ML

HLS4ML [7] takes the design flow one level higher by supporting Python. HLS4ML works by translating models from traditional open-source machine learning packages into HLS. This Python package is geared towards machine learning inference in FPGAs.

Figure 8 shows the HLS4ML workflow. An ML model built using Keras, for example, is converted into HLS, which then synthesizes into RTL in one automated process. Additionally, HLS4ML allows for fine-tuning in every step for better optimization or performance of the design. While HLS enables rapid pro-

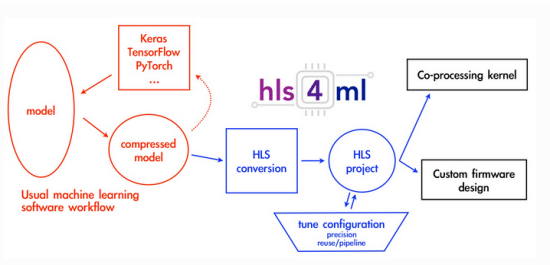


Figure 8: HLS4ML Workflow [6]

totyping compared to traditional RTL design, HLS4ML further accelerates the process, specifically for neural networks. In addition to quicker prototyping and deployment of neural networks on FPGAs, HLS4ML also provides users with control over size/compression, precision, dataflow/ resource reuse and quantization aware training for their models.

3 Multihat

A survey on various hardware architectures on Gaussian random number generators [16] introduced a new architecture called the Multihat. The study shows that the Multihat algorithm for Gaussian random number generation was not only the best in terms of hardware resource utilization, but also achieved a tail accuracy of 8σ, meaning, the algorithm generated accurate values 8 standard deviations away from the mean. The motivation to create a Gaussian random number generator comes from here.

As mentioned in the previous sections, equation 6 represents the math required within a VAE, where μ represents the mean, σ represents the standard deviation, ϵ represents the generated random number, and z represents the sampled latent representation. This is where the Multihat method [16] comes in with its ability to generate Gaussian random numbers with a long tail at a very low hardware cost. A long tail refers to the ability to generate rare, extreme

values that occur far from the mean, which is important for accurately modeling distributions with seldomly occurring yet significant outliers. The architecture of the Multihat algorithm can be divided into three distinct parts: the uniform random number generator, the hat generation, and the central limit theorem.

3.1 Uniform Random Number Generator

Data and noise generation in digital systems typically requires a sequence of random numbers. A widely used algorithm for generating these sequences is the multiplicative congruential generator (MCG), which produces a pseudorandom sequence with a uniform distribution and a long repeating period. A common hardware implementation of the MCG is the linear feedback shift register (LFSR), which, as seen in figure 9 consists of interconnected single-bit storage registers and a feedback logic network with an additional combinational XOR between two or more bits, also known as the LFSR's tap [4]. This feedback network and tap locations are typically modeled by what is known as the generating polynomial.

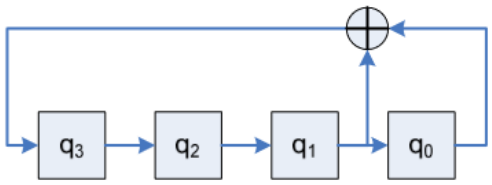


Figure 9: 4-bit LFSR with a tap point between 0 and 1 [4]

However, there is one major flaw with a typical implementation of LFSR, when used for the MCG algorithm: successive values in the generated sequence are correlated. From figure 9, it can be seen why values are correlated; with simple bitwise shifting and one XOR computation. This correlation is also seen in larger LFSRs, for example, a 168 bit LFSR contains similar bitwise shifting

and only four XOR computations [1]. Out of the many ways to mitigate this issue with the traditional LFSR, the Multihat method utilizes a technique known as the multi-bit skip-ahead. While the multi-bit skip-ahead technique is more efficient in terms of performance, it consumes more resources as the feedback network must implement the generating polynomial multiple times. Figure 10 shows a 4-bit LFSR with a skip-ahead of 3. This skipping ahead effectively improves the randomness of the pseudorandom number sequence of the LFSR, making it more uniform and less predictable.

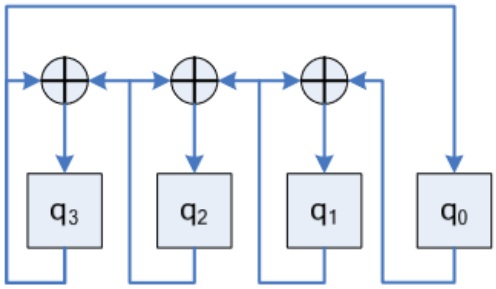


Figure 10: 4-bit LFSR with a skip-ahead of 3 [4]

Colavito's and Silage's "Efficient PGA LFSR Implementation Whitens Pseudorandom Numbers" [4] contains a deeper dive into the workings behind the architecture of a skip-ahead LFSR. Considering a 20-bit LFSR whose generating polynomial is as follows:

$$q_0 = q_{19} \oplus q_{16} \tag{2}$$

Equation 2 says that the 0th bit of this LFSR is an XOR computation of the 20th and 17th bit. For this LFSR, a skip-ahead of 5 would cause the generating polynomial to be represented by the below equations:

$$\begin{aligned} q_4 &= q_{19} \oplus q_{16} \\ q_3 &= q_{18} \oplus q_{15} \\ q_2 &= q_{17} \oplus q_{14} \\ q_1 &= q_{16} \oplus q_{13} \\ q_0 &= q_{15} \oplus q_{12} \end{aligned} \tag{3}$$

As seen in equation 3, the skip-ahead of 5 causes the original XOR computation from equation 2 to move bits in front. Along with moving the original XOR computation, the skip-ahead creates XOR computations that trail the jump from the 20th bit to the 5th bit.

The Multihat algorithm utilizes a 130-bit LFSR with a skip-ahead of 120. This 130-bit LFSR has an effective repetition period of 2^{123} [17]. Even though this skip-ahead LFSR has a smaller period when compared with the traditional 130-bit LFSR, the great advantage comes in the form of uncorrelated uniform random numbers, thus finishing the first step in the Multihat algorithm of creating a uniform random number generator.

3.2 Hat Generation

The fundamental idea behind the Multihat method is that it is possible to change the probability density of a uniformly distributed random sequence by introducing an extra bit and using a multiplexer [16]. The modified distribution begins to represent the shape of a hat, which can be adjusted using basic logic cells such multiplexers and simple gates. Taking a randomly generated 2's complement 16-bit number into consideration, forcing the 15th bit of the number

to be equal to the 16th bit, drops the range by a factor of 2. Assuming this number is distributed between (-1, 1), this 16-bit number would contain one integer bit and 15 fractional bits. When the 15th bit is forced to be equal to the 16th bit, the new range of the number becomes (-0.5, 0.5). A similar principle, used in conjunction with the muxes help in the generation of "hats," which will be described in the following sections in further detail.

3.2.1 The Standard Hat

Figure 11 shows the conversion of a uniform distribution to a hat distribution. A 16-bit number represented by x_{15}:x_0, uniformly distributed between (-1, 1) is directly mapped to the output z_{15}:z_0. Since this 16-bit number is distributed between (-1, 1), it is a fixed-point number that can be represented in the Q format, where Q(m.n) denotes m bits for the integer part and n bits for the fractional part. For this Q(1.15) number, the probability of all values occurring are the same: 2^{-16}, therefore, creating a probability density function with a height of 0.5 as seen in figure 11.

This 16-bit number is manipulated such that a mux decides z_{14}'s value between x_{15} and x_{14} where the select bit of the mux is x_{16}. Assuming every bit has a probability of 50%, the distribution of z will change such that 50% of the edge values will map to the center, increasing the height of the PDF in the center.

Table 1 helps to visualize the shift in probabilities. The mux transforms the input bits $x_{15}x_{14}$ such that, the 3/8 of all output values begin with output $z_{15}z_{14}$ being 00 or 11, and 1/8 of all output values begin with $z_{15}z_{14}$ as 01 or 10. This means that values from 16'b11_00000000000000 to 16'b11_11111111111111 (-0.5, $\sim$0) and values from 16'b00_00000000000000 to 16'b00_11111111111111 (0, $\sim$0.5) now have an updated probability of $3/8 * 2^{-14}$ with the remaining values having a probability of $1/8 * 2^{-14}$. This change in probabilities can be seen in

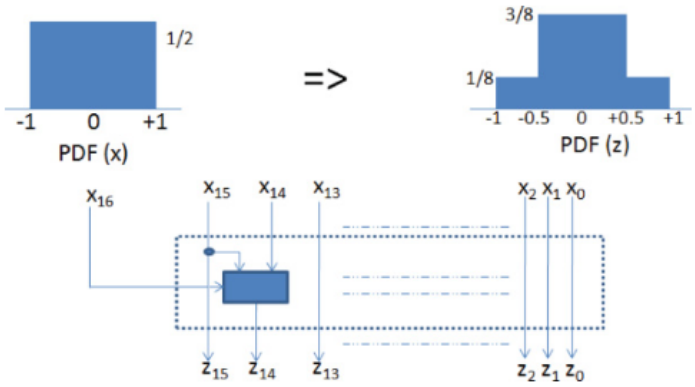


Figure 11: Converting a uniform distribution to a hat [16]

Table 1: Transformed Outputs of the Standard Hat from Figure 11

Output $z_{15}z_{14}$	Mux bit x_{16}	Input $x_{15}x_{14}$
00	0	00
	1	00
	1	01
11	0	11
	1	10
	1	11
01	0	01
10	0	10

the transformed probability density function in figure 11 where the middle part has a height of 3/8 and the edges have a height of 1/8.

3.2.2 The Narrow Hat

Building off the standard hat distribution, it is possible to further decrease the width of the middle to create the narrow hat distribution. As seen in figure 12, rather than having one mux between the input x and output z, there are now two muxes. Similar to the standard hat, the first mux drives z_{14}, whereas now the second mux drives the 14th bit z_{13}.

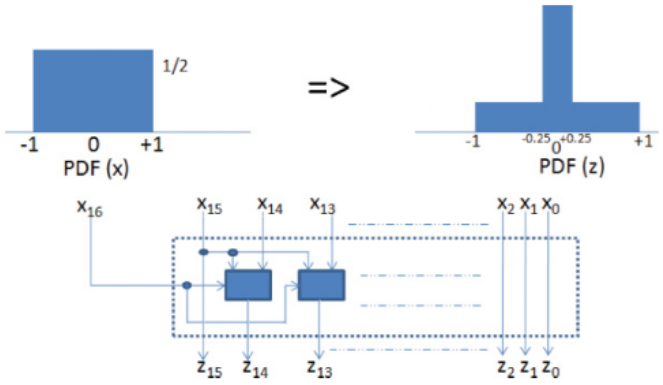


Figure 12: Converting a standard hat distribution to a narrow hat [16]

This effectively creates new probabilities which can be visualized from table 2. For a 16 bit output with the same representation as the standard hat of Q(1.15), 5/16 of all output values begin with output $z_{15}z_{14}z_{13}$ as 000 or 111, and 1/16 of all output values begin with $z_{15}z_{14}z_{13}$ as 001, 010, 011, 100, 101, or 110. Therefore, all values from 16'b000_0000000000000 to 16'b000_1111111111111 (0, ∼0.25) and all values from 16'b111_0000000000000 to 16'b111_1111111111111 (-0.25, ∼0) now have an updated probability of $5/16 * 2^{-13}$ with the remaining values having a probability of $1/16 * 2^{-13}$. This change in probabilities can be seen in the transformed probability density function in figure 12.

3.2.3 The Wide Hat

On the other hand, it is also possible to widen the distribution of the middle part, in what is called the wide hat distribution. As noticed in the narrow hat distribution, adding more muxes will only decrease the width, so to widen the distribution of the hat from (-0.5, ∼0.5) to (-0.75, ∼0.75) the wide distribution has the first instance of elementary logic gates in addition to two additional bits.

Table 2: Transformed Outputs of the Narrow Hat from Figure 12

Output $z_{15}z_{14}z_{13}$	Mux bit x_{16}	Input $x_{15}x_{14}x_{13}$
000	0	000
	1	000
	1	001
	1	010
	1	011
111	0	111
	1	100
	1	101
	1	110
	1	111
001	0	001
010	0	010
011	0	011
100	0	100
101	0	101
110	0	110

Figure 13 shows a simple digital circuit consisting of three gates - an XNOR, an XOR, an AND. Taking the top three bits of the 16-bit input in Q(1.15) format, the circuitry outputs true for two only cases - when $x_{15}x_{14}x_{13}$ is 011 or 100. For the case of $x_{15}x_{14}x_{13}$ being 011, the circuit outputs true for all values greater than or equal to 0.75. Similarly, for the case of when $x_{15}x_{14}x_{13}$ being 100, the circuit outputs true for all values less than -0.75. This circuit then drives a mux that either replaces the 14th and 15th bit with two new bits of 50% probability or feeds through the input to the output. These additional bits decreases the probability of numbers greater than 0.75 and numbers less than -0.75 from $1/4 * 2^{-13}$ to $1/4 * 0.5 * 0.5 * 2^{-13}$ [16]. Similar to the narrow hat of adding more muxes, the wide hat can be made wider by using more XOR, XNOR, and AND circuits recursively.

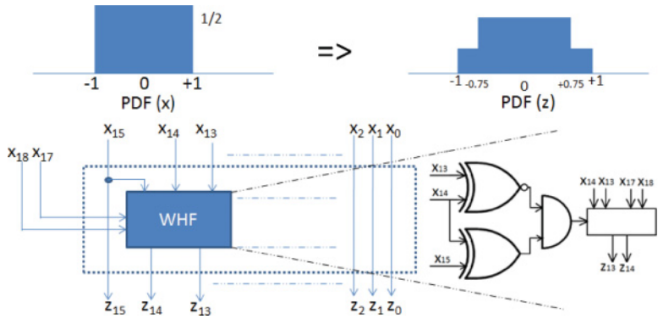


Figure 13: Converting a standard hat distribution to a wide hat [16]

3.2.4 The Tall and Short Hat

In addition to widening and narrowing the middle part of the hat, it is also possible to shorten and lengthen the middle part of the hat. As seen in the standard hat section and table 1, a mux bit with a 50% probability changed the probabilities of the edges and the middle by forcing outer samples of the uniform distribution inward [16]. As seen from the previous sections, a URNG typically outputs each bit with a 50% probability. However, using combinational circuitry, this 50% probability can be manipulated.

$$P(AandB) = P(A) * P(B) \tag{4}$$

$$P(AorB) = 1 - (P(A) * P(B)) \tag{5}$$

From equation 4 and 5, it is apparent that using an AND gate will lower the probability at the output, whereas using an OR gate will raise the probability at the output. In figure 14, the muxes of narrow hat distribution receives a second bit x_{17} which is AND-ed with the existing x_{16}. As seen above, AND-ing two bits will decrease the probability, thus decreasing the height of the narrow hat.

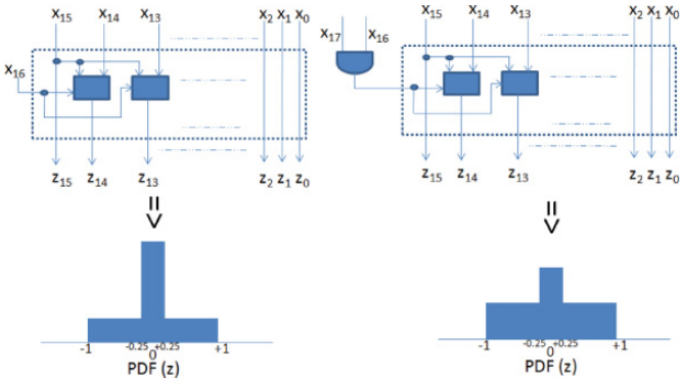


Figure 14: Converting a narrow hat distribution to a narrow and short hat [16]

Nevertheless, more complex combinational circuitry using ANDs and ORs can be created to achieve an arbitrary probability at the output. For example, in figure 15, assuming the inputs to the gates are from the URNG, with a probability of 50% each, the output of the AND gate will contain a probability of 12.5%, and the output of the first OR gate will contain a probability of 75%. Together, the final output will contain a probability of 90.625%.

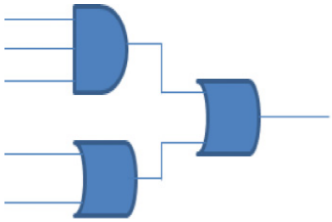


Figure 15: Boolean logic to manipulate bit probabilities [16]

3.2.5 The "Multi" Hat

Finally, the "Multi" hat stage combines multiple hat distribution techniques to create a prototype Gaussian curve. The Multihat algorithm, aptly, receives its name from the combination of all the different hat distribution techniques.

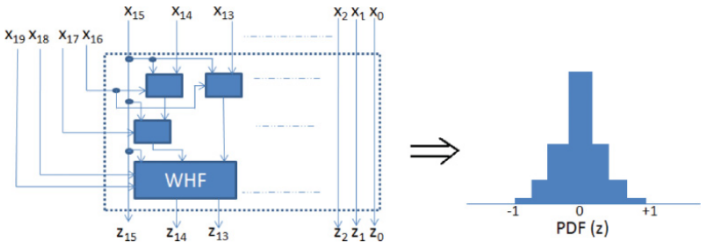


Figure 16: Combining hat distributions to generate a Gaussian-looking curve [16]

From figure 16, the combination of the narrow, standard, and wide hats yield a pyramid looking distribution that starts to resemble a Gaussian curve. Arbitrarily increasing and editing the steps can lead to a much more Gaussian-looking curve compared to figure 16. However, one obvious downside to increasing the complexity of the "Multi" hat is the explosion of hardware resource utilization. This explosion is due to the resources of the combinational circuit becoming larger with increased complexity. The creators of the Multihat algorithm, chose to limit the "Multi" hat stage to four steps and use the Central Limit Theorem to generate high tail Gaussian Random numbers [16].

3.3 Central Limit Theorem

"The Central Limit Theorem is one of those beautiful examples in nature, where a mathematical relationship can be directly associated to real-life observations" [17]. The CLT offers a cost-effective method of transforming uniform random numbers into a Gaussian distribution. According to the CLT, the sum of n independent random variables with a finite mean and variance approximates a Gaussian distribution as n increases [18]. Even though the CLT is as simple as solely relying on addition operations, CLT is rarely used to generate high tail Gaussian random numbers. "Error in tail regions of the Probability Density Function (PDF), is inversely proportional to the number of samples to

be added." [18] While multiple studies show the extent of using a CLT-based approach to generate Gaussian Random Numbers, the common issue is high resource usage that arises from the need to correct the error in the tail regions.

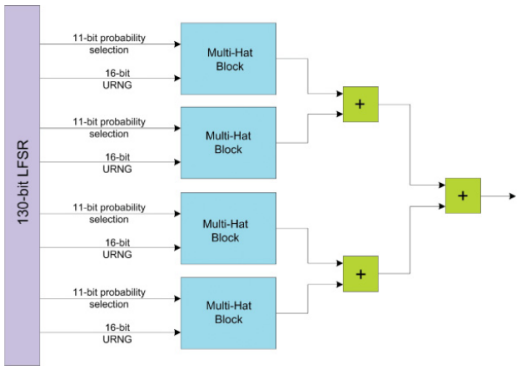


Figure 17: Multihat Block Diagram [16]

However, as mentioned in the previous section, the Multihat algorithm as a whole, uses the best of both worlds. Using the "Multi" hat approach combined with the power of the Central Limit Theorem, the Multihat algorithm is able to generate high-tail Gaussian Random Numbers at relatively low resource utilization. From figure 17, we can see that four Multihat blocks convert the LFSR's uniform distribution to a pyramid distribution. Each Multihat block uses a total of 27 bits, 16 bits for the input and 11 bits that dictate the overall shape of the pyramid distribution. Breaking it down, 4 bits generate a thin hat, 3 bits generate a standard hat, and 4 bits generate a wide hat with relevant shortening and raising. The adder blocks then sum up all the Multihat block, covering the CLT part of the Multihat algorithm. Additionally, if the variance of an individual block is unity, the variance of the resultant distribution remains unity if the final number is divided by $\sqrt{n}$ [17][16]. Since four Multihat blocks are added together, the final output needs to be divided by 2, which is a trivial task in fixed point representation.

4 HLS Implementation

The Multihat algorithm was recreated in HLS, with the original VHDL design taken into consideration. Using a class-based approach, the HLS Multihat algorithm consisted of its three core components: The LFSR-based uniform random number generator, the "Multi" hat pyramid distribution generator, and the central limit theorem. Next, further modifications were made to the overall design that allowed the creation of multiple Multihat Gaussian Random Number Generators. Each "GRNG" block is made out of the core Multihat algorithm components, as visualized in figure 17. Multiple "GRNG"s create their own instance of the "Multihat block." However, one issue in creating multiple GRNGs is the initial state for each of the LFSRs. N different GRNGs would require N different initial states for the N LFSRs. In addition to requiring N different initial states, each state must be uncorrelated to ensure each GRNG outputs an uncorrelated Gaussian random number. This was tackled by giving one input "SEED" to the system, which, uses Katio Udagawa's [22] SplitMix32 algorithm to create N different initial seeds for N GRNGs. The SplitMix32 is an extremely cheap function that is based on an algorithm known as SplitMix included in Java JDK8 [3]. By performing several XORs and shifts, this function essentially the seeds inputted to the GRNGs.

Additionally, the implementation taking place in Xilinx's Vivado HLS 2019.2 allows for easier optimization in comparison to writing HDL. Various pragmas such as array partitioning, loop unrolling, and pipelining aided in designing a highly optimized N Gaussian Random Number Generators.

5 Integration into HLS4ML

As mentioned in previous sections, the motivation to create an efficient Gaussian random number generator comes from the need to create and deploy a Variational Autoencoder on an FPGA. With HLS4ML's ability to translate traditional open-source machine learning package models into HLS, the next step was to integrate the Multihat Gaussian random number generator into HLS4ML by creating a custom layer.

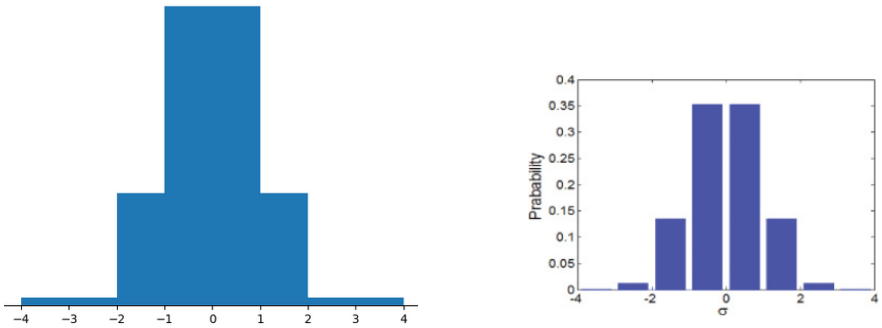
The custom Gaussian layer, modeled after Keras' implementation of a sampling layer, gives the HLS4ML Gaussian layer two inputs that aid in producing the accurate value at the output of the Gaussian layer. Equation 6 from a previous section is slightly modified to the below equation:

$$z = \mu + e^{(0.5 * \ln(\sigma^2))} * \epsilon \tag{6}$$

The Gaussian layer receives two inputs - the mean, μ, and the log_var, $\ln(\sigma^2)$, for a certain input. Since the layer receives a log_var value, to account for accurate scaling, the value needs to be converted to its respective standard deviation. This was achieved in the HLS implementation of Multihat by creating a look-up table to perform the exponential calculation. With a default size of 1024 entries and a binary representation of Q(8.10), the LUT produces a good approximation for the exponential calculation.

6 Results

This section will discuss in more detail the output bitwidth of the Multihat Gaussian random number generator, the statistical properties of the generated Gaussian, resource utilization and performance, and a comparison with a different implementation of a GRNG.



(a) Generated output of "Multi" hat block

(b) Output of "Multi" hat block from the Multihat paper [16]

Figure 18: Comparison of 1 million HLS generated output of one "Multi" hat block vs the paper's output.

6.1 Configurable Bitwidths

The first step in verifying the functionality of the Multihat algorithm was to verify the functionality of the "Multi" hat block itself.

Figure 18 shows a comparison of the output at one "Multi" hat block. Figure 18a contains 1 million outputs from the HLS code, whereas Figure 18b is the pyramid distribution from an ideal Gaussian curve described in the Multihat paper [16]. As can be seen, the HLS output very closely mimics the shape of the pyramid distribution from an ideal Gaussian curve. This single "Multi" hat block achieved a mean, μ, of 0.000389, a standard deviation, σ, of 1.08535, and a variance, σ^2 of 1.17800. This implies, the 16-bit output of the "Multi"

hat block is in a binary representation of Q(3.13) with a maximum possible value of 3.9998779296875 and a minimum value of -4. Looking back at figure 17, this means that the final output after all the CLT additions has a binary representation of Q(5.11). However, the last step as part of the CLT process - "if the variance of an individual block is unity, the variance of the resultant distribution remains unity if the final number is divided by $\sqrt{n}$ [17][16]." Note, the variance is not exactly unity, showcasing the Multihat method is not an exact method, but, rather an approximate method. Since the design has four blocks, the final Q(5.11) output is right shifted by one, to divide by two. This effectively means that the final output of the Gaussian random number generator is in the format Q(4.12).

Looking back at figure 17, one can notice that each "Multi" hat block receives 27 bits, totaling to 108 bits, whereas the LFSR is 130 bits wide. In reality, each "Multi" hat block receives 32 bits in total. The base engine for each block uses only 27 bits, but the HLS algorithm has been designed in a way to achieve variable bitwidths, specifically for three cases - extending the fractional bits up to 17 (by the 5 unused bits) fractional bits for a representation Q(4.17), truncating the fractional bits to reduce precision, and truncating the integer bits. The primary use case of this added support is for greater flexibility for quantization. HLS4ML users typically quantize their designs, which is the process of reducing the numerical precision of the weights, biases, and activations in a neural network model.

However, extending the integer bits has not been implemented. The base Multihat Algorithm generates an output of Q(4.12) which accounts for a tail accuracy of 8σ. Increasing the bitwidth to Q(5.12), for example, may seem desirable but is "only symbolic"[17]. From one CLT-based GRNG study that achieved a tail accuracy of 12σ, they emphasized "the probability of occurrence

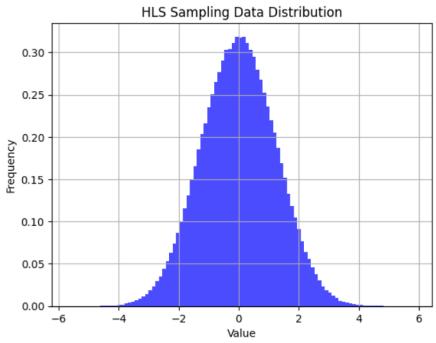
of an event around 12σ is so low that a GRNG producing 1-G samples per second will take roughly 10^{24} **years** to exhibit". With a bitwidth of Q(5.12), a tail accuracy of 16σ could be achieved, however, the occurrence of an event would generally only be seen after an unimaginably long period of time.

6.2 Statistical Properties

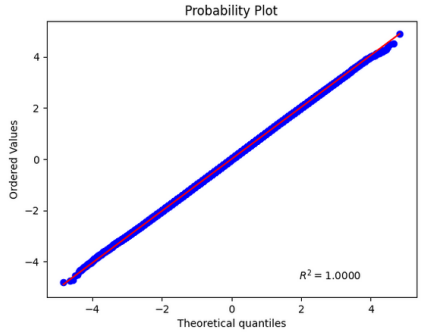
Several tests were conducted to validate the quality of the GRNG. Figure 19a shows the distribution of one million random numbers generated using the GRNG. Even though the figure clearly shows a normal distribution, it is not reliable because the bin sizes in such plots are arbitrary. This can lead to normal distribution-shaped plots without the properties of normal distribution.

Next, Figure 19b shows the QQ plot of the same one million values. The QQ plot compares the generated values against the theoretical normal distribution, which is represented by the red line. While this is another visual check, the QQ plot allows for a better check of the tail behavior compared to the histogram from Figure 19a. Here, we can see some slight discrepancies from the theoretical normal distribution. As mentioned in the previous section, the Multihat method is not an exact method; however, it is possible to further reduce the discrepancy by increasing the number of steps in the "Multi" hat pyramid distribution, and increasing the addition operations in the CLT part.

Statistically, there are two numerical values that can test the normality of data: skewness and Kurtosis. According to the skewness test, for normally distributed data, the skewness should be about 0. This is because skewness is a measure of the asymmetry of the probability distribution of a random variable about its mean. On the other hand, the Kurtosis test compares the height and sharpness of the central peak, relative to that of a bell curve. The Kurtosis of a normal distribution is 0, if following Fisher's definition of Kurtosis. For



(a) Histogram Representation.



(b) QQ Plot

Figure 19: Two different visual representations of one million numbers generated by GRNG

the same data used in Figure 19a and Figure 19b, the skewness and Kurtosis tests returned values of -0.00086 and -0.04518, respectively. Some studies suggest that skewness and Kurtosis values of less than 1.0 indicate "slight non-normality," whereas other studies suggest values up to absolute 1.0 indicate normality [19]. So, based on the skewness and Kurtosis results, we can conclude that the generated data are Gaussian.

6.3 Resource Utilization and Performance

Another property of HLS4ML that comes into play during resource utilization and performance analysis is the reuse factor. The reuse factor is an important

configuration parameter of HLS4ML that is set for each layer, including the new Gaussian layer. The reuse factor, essentially, determines how many resources of a certain type are used.

Table 3: Resource utilization and Performance based on the reuse factor for 32 GRNGs

Reuse Factor	DSP	FF	LUT	Latency	Interval
1	32	15594	16232	4	1
2	32	10138	8264	5	2
4	32	7412	4280	7	4
8	32	6056	2300	11	8
16	32	5390	1380	19	16
32	32	5081	989	35	32

Table 3 shows the resource utilization of the Multihat algorithm, with the inclusion of the additional math required to accurately scale the mean and standard deviation. With a reuse factor of 1, the system uses the most amount of resources, but has the best performance with an interval of only 1. This means that for 32 GRNGs, the system is instantiating 32 different GRNG blocks that are each capable of generating a new random number every single cycle. On the other hand, as expected, an increase in reuse factor decreases the total resource with an almost-half decrease for the LUTs. The drawback of a higher reuse factor comes in the form of performance where the system needs four cycles, in case of reuse factor four, per random number because the system generates eight GRNGs and uses each 4 times.

To achieve minimal latency, this designed contained several optimization techniques in HLS such as array partitioning, loop unrolling, and pipelining. However, due to these techniques, the DSP utilization remained consistent no matter the reuse factor. Other techniques to force the reusing of the DSP involved non-pipelining the design, which exploded the latency and interval, proving to be very inefficient.

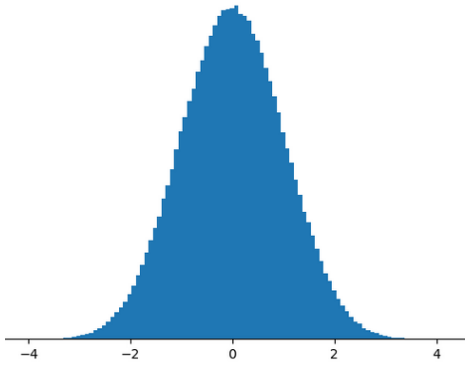
6.4 Comparing a Second Implementation

An HLS GRNG implementation by CERN [15] allows for a deeper understanding of not only the Multihat algorithm, but also the HLS tools. The CERN GRNG approach is a CLT-based approach which uses a pseudo random number generator to generate an array of uniform samples. Then, the samples are summed up, according to the Central Limit Theorem. Finally, the output is divided by the variance to get the proper range.

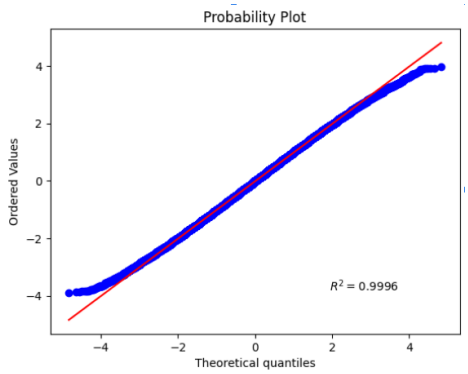
Table 4: Resource utilization and Performance Comparison at 32 Samples per Cycle

Approach	DSP	FF	LUT	Clock	Latency	Interval
Multihat	32	15594	16232	5ns	4 Cycles	1 Cycles
CLT-based	128	9378	18129	10ns	2 Cycles	2 Cycles

Table 4 shows a detailed comparison between the two approaches. For both designs outputting 32 samples per cycle, the CLT-based approach uses fewer flip-flops than the Multihat algorithm, the Multihat algorithm utilizer fewer DSPs, LUTs, and has a faster performance. In terms of statistical properties, it can be seen from figure 20a that the data is in a Gaussian shaped curve. However, when figure 20b comes into play, the significant discrepancy in the tail can be seen, especially when compared to the Multihat algorithm's QQ Plot 19b. Additionally, the skewness and Kurtosis tests returned values of -0.0021 and -0.1525, respectively. This may be considered Gaussian data, however, the Multihat algorithm outperforms it once again.



(a) Histogram Representation.



(b) QQ Plot

Figure 20: Two different visual representations of one million numbers generated by GRNG

7 Applications

As mentioned previously, the primary motivation for the Gaussian random number generator comes from the need of creating VAE through HLS4ML. One such VAE is the Latent Factor Analysis via Dynamical Systems (LFADS) [14]. Neuroscience is currently undergoing a data revolution, with the ability to record the activity of hundreds or even thousands of neurons simultaneously. LFADS is a new approach designed to reveal latent dynamics from high-dimensional, single-trial neural spiking data recorded across multiple neurons. LFADS is a se-

quential model based on a variational autoencoder framework, which integrates a dynamical systems perspective to model the generation of the observed data. By reducing the high-dimensional spiking data into a set of low-dimensional temporal factors, trial-specific initial conditions, and inferred external inputs, LFADS provides a more interpretable representation [21]. An existing non-Gaussian HLS implementation of LFADS was modified to account for the new Gaussian layer. The inclusion of this layer also saw the inclusion of several other layers to account for the extra computation. This, obviously, led to an increase in resources Table 5 represents the extra hardware costs from these additional layers.

Table 5: LFADS Hardware Cost with GRNG and additional layers post place-and-route for device xcu50fsvh2104-2L.

Gaussian Layer	BRAM18K	DSP48E	FF	LUT
Yes	31.29%	37.90%	14.49%	29.16%
No	29.09%	35.75%	12.82%	26.65%

Overall, VAEs have become a powerful tool in various domains of machine learning and artificial intelligence due to their ability to generate new data that resembles the distribution of the training data. In image processing, VAEs are widely used for tasks like image generation, inpainting (filling in missing parts of images), and style transfer. By learning a compressed latent representation of an image, VAEs can sample new data points from this latent space to generate novel images, enabling creative applications in art, fashion, and design. Additionally, VAEs are used in anomaly detection, where they model the normal data distribution and can flag outliers or unusual data points that deviate from this learned distribution.

In natural language processing, VAEs are applied in text generation, machine translation, and dialogue systems, where they help generate relevant and appropriate sentences by learning low-dimensional representations of text. Their

ability to model uncertainty in data and generate diverse outputs has led to their use in drug discovery, where they can explore chemical space by generating novel molecular structures with desired properties. VAEs have also been applied in time-series prediction, such as for forecasting stock prices or weather patterns, by learning complex dependencies in sequential data [25][8].

8 Conclusion

In conclusion, the Multihat method proves to be an effective approach for Gaussian random number generation in FPGA implementations of machine learning models. By leveraging simple combinational logic and the Central Limit Theorem, the Multihat algorithm can generate highly accurate Gaussian distributions with efficient resource utilization.

The Multihat method stands out for its ability to balance resource constraints and performance. While other methods like the CLT-based approach offer competitive performance, the Multihat method's combination of simplicity and accuracy ensures its relevance in applications where hardware efficiency is the biggest priority. As VAEs and similar models continue to gain prominence, the role of efficient hardware-implemented GRNGs like the Multihat method will likely expand, contributing to faster, more energy-efficient deployments in various fields such as image processing, time-series analysis, and anomaly detection.

9 My Contributions

The development of the Multihat algorithm was originally started by Jeffery Xu who translated the Multihat algorithm's original VHDL design to SystemVerilog.

After taking over, I implemented the Multihat algorithm in HLS with the aforementioned features. Once I achieved a working GRNG, I edited the algorithm to generate an array of N GRNGs while performing HLS-related optimization techniques to create a design with minimal latency.

After synthesizing the design and verifying the statistical properties, I integrated the design into HLS4ML as a custom layer. This included minor code adjustments to fit in the HLS4ML framework, and the creation of a Python notebook that registered the custom layer as an HLS4ML layer.

Additionally, to test the design within LFADS, I modified the LFADS design with the help of the LFADS team (Chi-Jui Chen and Xiaohan Liu) to use the Multihat algorithm and performed various comparisons of LFADS with and without the GRNG.

References

[1] AMD. Efficient shift registers, lfsr counters, and long pseudo-random sequence generators. XAPP052, July 1996. Document ID: XAPP052, Revision 1.1, English.

[2] Dave Bergmann and Cole Stryker. What is an autoencoder?, 2024.

[3] Bryc. Prngs - pseudorandom number generators.

[4] Leonard Colavito and Dennis Silage. Efficient pga lfsr implementation whitens pseudorandom numbers. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 308–313, 2009.

[5] Carl Doersch. Tutorial on variational autoencoders, 2021.

[6] Javier Duarte et al. Fast inference of deep neural networks in FPGAs for particle physics. *JINST*, 13(07):P07027, 2018.

[7] FastML Team. fastmachinelearning/hls4ml, 2023.

[8] Saba Hesaraki. Navigating the world of variational autoencoders: From architecture to applications, 2024.

[9] IBM. What is a neural network?, Oct 2024.

[10] Dilith Jayakody. The reparameterization trick clearly explained, Dec 2023.

[11] Jeremy Jordan. Variational autoencoders., Mar 2023.

[12] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 12(4):307–392, 2019.

[13] Kiprono Elijah Koech. The basics of neural networks (neural network series)-part 1, May 2022.

[14] Xiaohan Liu, ChiJui Chen, YanLun Huang, LingChi Yang, Elham E Khoda, Yihui Chen, Scott Hauck, Shih-Chieh Hsu, and Bo-Cheng Lai. Fpga deployment of lfads for real-time neuroscience experiments, 2024.

[15] Vladimir Loncar. Personal communication, 2024.

[16] Jamshaid Sarwar Malik and Ahmed Hemani. Gaussian random number generation: A survey on hardware architectures. *ACM Comput. Surv.*, 49(3), November 2016.

[17] Jamshaid Sarwar Malik, Ahmed Hemani, Jameel Nawaz Malik, Ben Silmane, and Nasirud Din Gohar. Revisiting central limit theorem: Accurate gaussian random number generation in vlsi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(5):842–855, 2015.

[18] Jamshaid Sarwar Malik, Jameel Nawaz Malik, Ahmed Hemani, and N.D. Gohar. Generating high tail accuracy gaussian random numbers in hardware using central limit theorem. In *2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip*, pages 60–65, 2011.

[19] Fatih ORCAN. Parametric or non-parametric: Skewness to test normality for mean comparison. *International Journal of Assessment Tools in Education*, 7(2):255–265, Jun 2020.

[20] Irhum Shafkat. Intuitively understanding variational autoencoders, Oct 2021.

[21] David Sussillo, Rafal Jozefowicz, L. F. Abbott, and Chethan Pandarinath. Lfads - latent factor analysis via dynamical systems, 2016.

[22] UmiReon. splitmix32.c. https://github.com/umireon/my-random-stuff/blob/master/xorshift/splitmix32.c.

[23] Dagang Wei. Demystifying neural networks: Variational autoencoders, Mar 2024.

[24] Inc. Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis*, Oct 2019. UG902 (v2019.2).

[25] Renda Zhang. Variational autoencoders series 4: Beyond images - the multidomain applications of vaes, 2024.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Dr. Scott Hauck, whose guidance, encouragement, and insight have been invaluable both inside and outside the research world. Working with you over the past two years has been a truly rewarding experience. I am also deeply appreciative of Dr. Shih-Chieh Hsu for serving on my thesis committee, for your insightful feedback and support, and for having me as a part of the A3D3 institute.

A huge thank you to all the members of the ACME lab, past and present, and HLS4ML community for helping me with the various questions I bombarded you with. A special thank you to Yilin Shen, Xiaohan Liu, Chi-Jui Chen, Zhixing "Ethan" Jiang, Jeffery Xu, Geoff Jones and Vladimir Loncar.

To my parents, siblings, grandparents, friends, and my biggest cheerleader, thank you all for your constant unwavering support and encouragement throughout this entire journey.