Variational Autoencoder for Image Deraining

Firdaus
School of Electrical Engineering and Informatics
Bandung Institute of Technology (ITB)
Bandung, Indonesia
13221013@std.stei.itb.ac.id

Muhammad Raihan Fahlevi
School of Electrical Engineering and Informatics
Bandung Institute of Technology (ITB)
Bandung, Indonesia
13221027@std.stei.itb.ac.id

Ilham Nalendra Adji
School of Electrical Engineering and Informatics
Bandung Institute of Technology (ITB)
Bandung, Indonesia
13221035@std.stei.itb.ac.id

Ade Irman Budi Hendriawan
School of Electrical Engineering and Informatics
Bandung Institute of Technology (ITB)
Bandung, Indonesia
13221058@std.stei.itb.ac.id

M. Zhafran Arrafi Anwar
School of Electrical Engineering and Informatics
Bandung Institute of Technology (ITB)
Bandung, Indonesia
13221079@std.stei.itb.ac.id

***Abstract*—This paper implements a Variational Autoencoder (VAE) for image deraining on FPGA. The implementation is divided into two stages: a basic test using 3x3 pixel patterns and an image deraining application using 4x4 patches of 224x224 grayscale images. We propose a hardware-efficient VAE architecture that uses 16-bit fixed-point representation with activation functions linearized for FPGA implementation. For the basic test case, our architecture successfully reconstructs patterns with reasonable accuracy at 1MHz clock frequency. However, for the deraining application, while achieving efficient resource utilization with 11,905 LUTs and 13,830 FFs, timing constraints and deraining performance remain challenging. The results suggest that while VAE can be efficiently implemented on FPGA, more complex architectures may be needed for effective deraining applications. This implementation provides insights into deploying deep learning models on resource-constrained embedded systems.**

Keywords—Denoising, Variational Autoencoder, Deraining

I. Introduction

Variational autoencoders (VAEs) are a type of generative model in machine learning (ML) designed to create new data by producing variations of the input data on which they are trained. They also perform typical autoencoder tasks, such as denoising [1]. Like other autoencoders, VAEs are deep learning models consisting of an encoder, which identifies key latent variables from the training data, and a decoder, which reconstructs the input data using those latent variables.

Unlike most autoencoder designs that use a fixed, discrete representation for latent variables, VAEs employ a continuous, probabilistic representation of the latent space. This allows VAEs not only to reconstruct the original input precisely but also to use variational inference to generate new data samples that resemble the input data.

This paper presents the implementation of a Variational Autoencoder (VAE) for a deraining application. The deraining task focuses on minimizing errors caused by rain obscuring a camera's view, which often results in blurred images. VAEs are highly effective for denoising and identifying anomalies, making them well-suited to address the challenges associated with deraining.

The ability to remove the effects of rain from images has become increasingly important in modern computer vision applications, especially in autonomous vehicles, where accurate visual perception is critical for safety. Rain can significantly degrade image quality and affect the performance of vision-based systems, potentially leading to errors in critical tasks such as object detection and scene understanding.

While traditional image processing approaches often struggle to cope with the complex and non-linear nature of rain patterns, VAEs offer a promising solution through their ability to learn and model the distribution of rain patterns. Our implementation focuses on developing a hardware-efficient VAE architecture that can perform real-time deraining on FPGA platforms, demonstrating that sophisticated deep learning models can be effectively implemented on resource-constrained embedded systems without compromising performance.

II. Design

A. ReLU Activation Function

ReLU, or Rectified Linear Unit, is one of the most commonly used activation functions in deep learning, especially for hidden layers. ReLU works in a simple manner, defined as f(x) = x if x>0, and f(x) = 0 if x ≤ 0. ReLU is often used as the activation function in the encoder part of an Autoencoder (AE).

B. Sigmoid Activation Function

Sigmoid activation function has equation f(x) = 1 / (1 + e^-x). Sigmoid is commonly used for output layer on deep learning architecture. Typically, sigmoid used for binary classification as it works by predicting 1 or 0 if it passes threshold or not. In AE or VAE, sigmoid used on decoder layer especially on output layer for predicting number from 0 to 1.

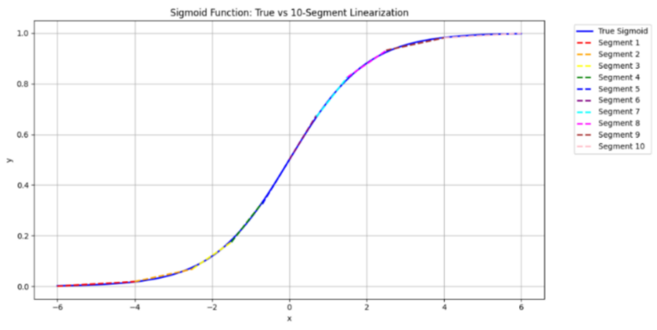


Fig. 1. Linearize Sigmoid Function

Exponential function is quite hard for hardware to synthesize. One optimal and simple approach was to linearize that nonlinear function. In this case, sigmoid function is linearized to 10 linear segments ranging from –3 to 3. Those segments will have equation f(x) = ax + b.

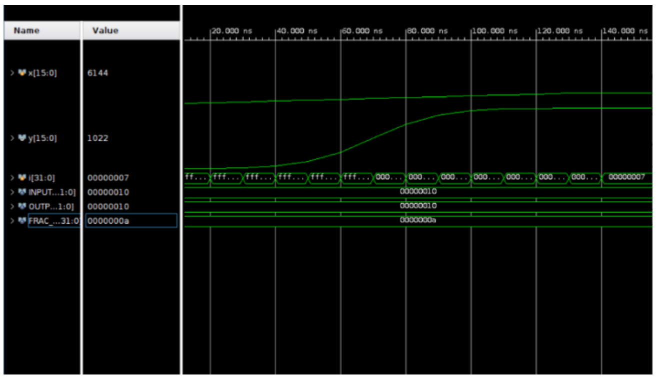


Fig. 2. Sigmoid RTL Simulation

In all designs, 16 bits fixed point number is chosen. 10 bits for fractional, 5 bits for integer, and 1 bit for sign. In figure 2, sigmoid is simulated using Vivado. For implementing sigmoid in verilog, basically it just need to make muxes.

C. *Softplus Activation Function*

Softplus is one alternative for hidden layer activation function. ReLU is so simple and widely used but it has singularity on x = 0, so softplus comes in as an alternative so it will be not singular. Softplus basically ReLU with some nonlinearities. Softplus on VAE is used on encoder layer. Another reasons why choose softplus instead of ReLU is softplus will generate better reconstruction loss and latent loss compared to ReLU.

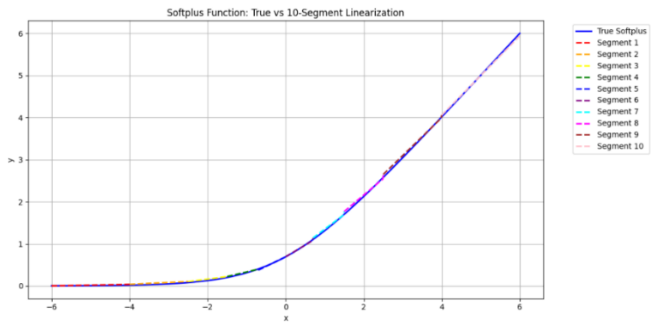


Fig. 3. Linearize Softplus Function

Softplus has equation of f (x) = ln (1 + e^x). Same with softmax, with exponential function and logarithmic exist, it is really hard for hardware to synthesize, so again, linearize it. Softplus is linearized into 10 linear segments from –6 to 6 with equation f (x) = ax + b. For number x > 6, the equation will be f (x) = x, because for such number, it is linear. The RTL simulation on Vivado, it can be seen on figure 4. Same as softmax, to implement in vivado, just make muxes.

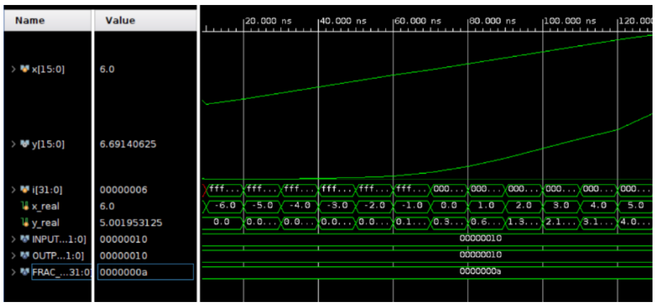


Fig. 4. Softplus RTL Simulation

D. *Random Number Generator*

The random normal distribution, also known as Gaussian distribution, is a fundamental probability concept characterized by its symmetric bell-shaped curve, defined by its mean (μ) and standard deviation (σ). In the context shown by the graph, it demonstrates a non-linear transformation process that converts uniform random values (ranging from 0 to 255) into normally distributed values (bounded between -3 and +3 standard deviations) using an inverse cumulative distribution function (inverse CDF). This transformation is essential because while most random number generators produce uniform distributions, many applications in statistics, machine learning, and scientific simulations require normally distributed values, where data points are more likely to cluster around the mean with decreasing probability towards the extremes, as evidenced by the S-shaped curve in the transformation graph..

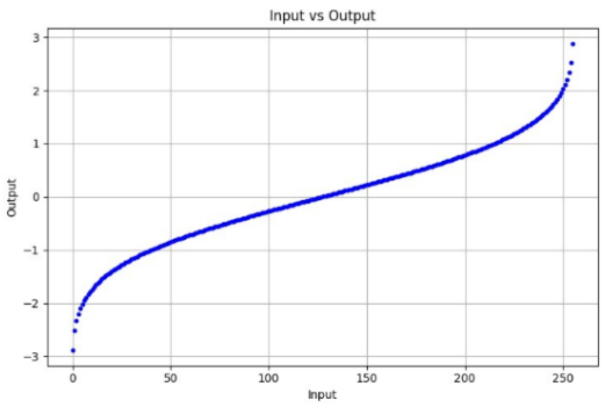


Fig. 5. Input vs Output Plot

The graph depicts a non-linear transformation function (inverse CDF or quantile function) that converts uniform inputs (0-255) into normally distributed outputs (-3 to +3 standard deviations). Its characteristic S-shaped curve features a steeper slope in the middle and flatter slopes at the extremes, effectively mapping inputs to create a bell-shaped probability distribution. This smooth, continuous transformation ensures that values cluster more densely around the mean while maintaining appropriate probability distribution in the tails, covering 99.7% of possible values within the ±3 standard deviation range. This implementation is fundamental in various fields including statistical modeling, machine learning, and signal processing, where converting uniform random inputs into normal distributions is essential for accurate data simulation and analysis.

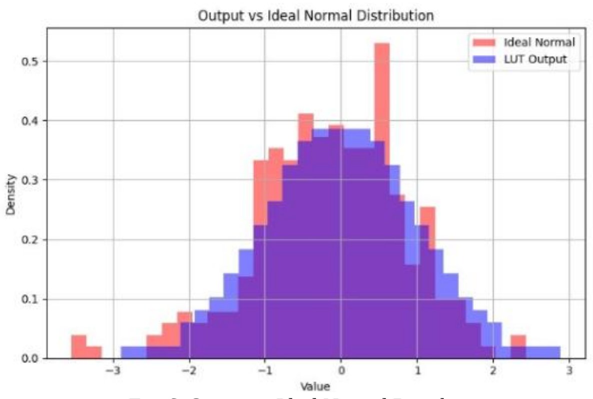


Fig. 6. Output vs Ideal Normal Distribution

The graph compares the generated normal distribution (purple histogram) against an ideal normal distribution (red overlay), demonstrating the effectiveness of the transformation function. The generated data closely matches the theoretical distribution, with both peaks correctly aligned at the center (mean) and similar density patterns throughout. The symmetric bell shape is well-preserved in the generated data, and the tail regions on both sides show appropriate tapering that matches the ideal distribution's characteristics. The overlap between the two distributions indicates that the implementation successfully produces random numbers that follow normal distribution properties, with the histogram's shape, spread, and density closely approximating the expected theoretical values.

E. *Variational Auto Encoder for Circle and Cross (3x3 Pixels)*

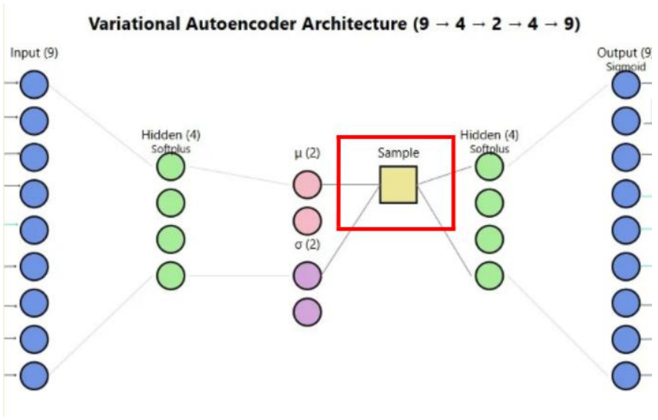


Fig. 7. VAE 3x3 Image Architecture

Our VAE Architecture has 9 inputs, because the image is 3x3 pixels. Our training image consists of cross and circle image. In encoder layer, 9 inputs is compressed into 4 neurons. In encoder layer, softplus activation is used. After that, it sampled using equation $z = \mu + \sigma \odot \varepsilon$. ε is random noise N(0,1) and σ is standard deviation that has formula of $\exp(0.5 \times \log \sigma^2)$. Then, it decompressed again into 4 neurons and has softplus activation function. Finally it decompressed again into 9 outputs so it will make sure the same number of inputs-outputs. In output layer, because of the output value should lies within value of 0 and 1. Sigmoid activation function is used.

VAE needs correct weights and biases value, so we created similar architecture using PyTorch to train and test. We were using Adam optimizer with learning rate of 10^-3 and used 2000 epochs. The loss function we were using is combination of reconstruction loss and Kullback-Leibler divergence loss. The weights and biases then used in verilog code but we should make sure that we have converted the data type using 16 bits fixed point data.

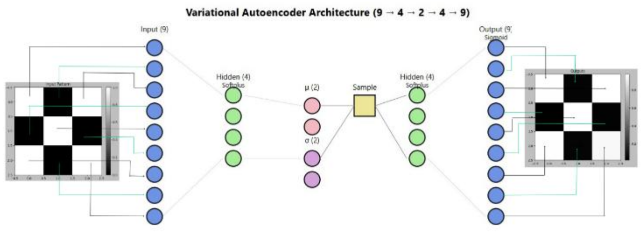


Fig. 8. VAE Output for Cross Input

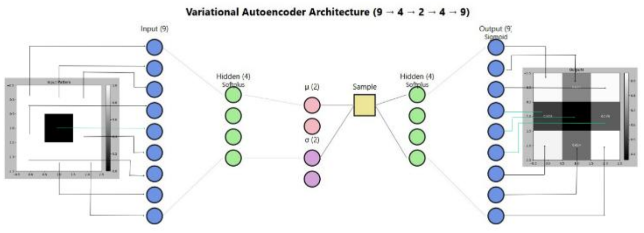


Fig. 9. VAE Output for Circle Input

Figure 8 and 9 shows output of VAE using verilog, for cross input, the output shows promising output. But, when input is circle, there are some errors on another pixels, but if the outputs are limited to just 0 and 1 not within range [0,1], the output of circle will show the exact same pixels as inputs are.

F. *Variational Auto Encoder for Image Deraining*

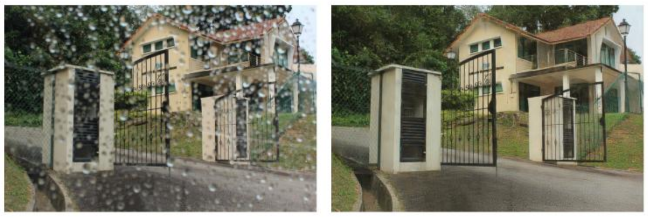


Fig. 10. GAN For Raindrop Removal [1]

VAE has so many applications, starting from image generation until image reconstruction. Image deraining is one of the application for VAE. Our architecture is designed so that droplets on left picture will be removed like right picture. Same with VAE for 3x3 pixels, our architecture was trained using PyTorch using image raindrop datasets, consists of ~1000 images. The original image has size of 224 x 224 x 3 (RGB) but then it preprocessed to grayscale, so now it has size of 224 x 224 x 1.

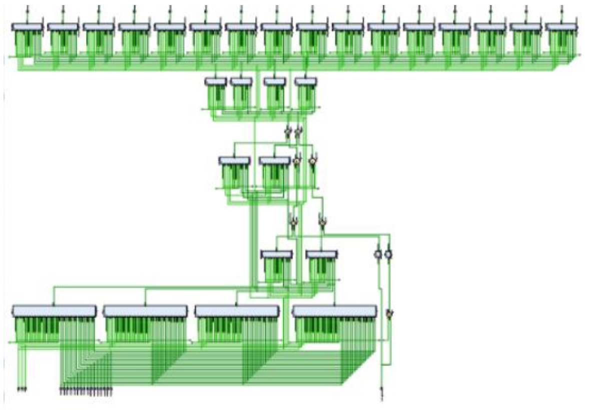


Fig. 11. VAE for Image Deraining Architecture

Our architecture now has inputs of 16 (4 x 4) instead of 9 (3 x 3). So the input image is patched into many patches with size of 4 x 4 and put into our architecture. Our model was trained using Adam optimizer and trained using 1000 epochs. The weights and biases also sent to VAE model on verilog.

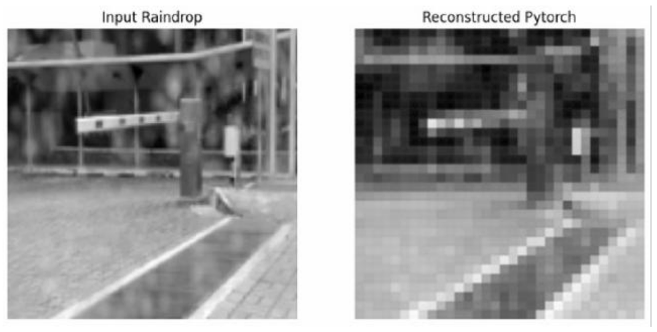


Fig. 12. VAE Inputs vs Outputs from PyTorch

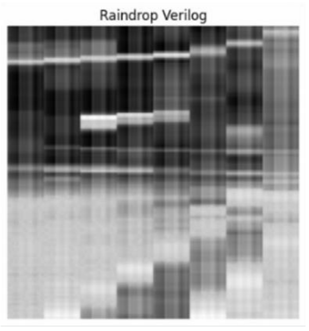


Fig. 13. Outputs from Verilog

Based on the result, the approach by using 4x4 patches is not actually correct. Also, the VAE architecture is not enough to remove rain on picture. It needs more complex architecture by adding CNN, ResNet, or even Attention network.

III. IMPLEMENTATION

A. *Variational Auto Encoder for Circle and Cross (3x3 Pixels)*

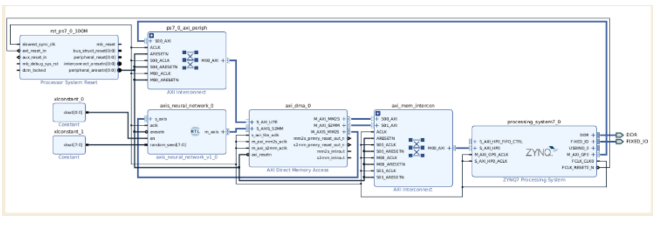
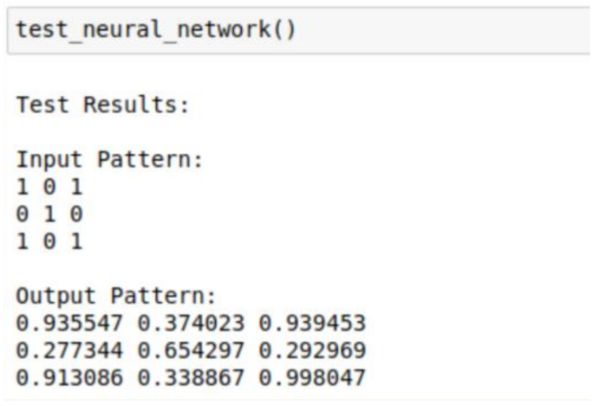


Fig. 14. Block Design of VAE for Circle and Cross

The implementation of VAE level 1 architecture on FPGA is shown in the form of an interconnected block diagram. The architecture consists of a memory controller block at the beginning which functions to store input data as well as weights and biases from training results from PyTorch. The main part of the architecture contains the VAE neural network implementation which includes an encoder with a softplus activation function, a sampling block to implement the reparameterisation trick with the formula $z = \mu + \sigma \odot \varepsilon$, and a decoder that uses a sigmoid activation function on the output layer. The final part is connected to the processing system on the ZYNQ FPGA which plays a role in controlling the data flow and performing final processing. The implementation uses a 1MHz clock because it is limited by the use of combinational circuits, with a 16-bit fixed point data format divided into 10 fractional bits, 5 integer bits, and 1 sign bit.



```
test_neural_network()

Test Results:

Input Pattern:
1 0 1
0 1 0
1 0 1

Output Pattern:
0.935547 0.374023 0.939453
0.277344 0.654297 0.292969
0.913086 0.338867 0.998047
```

Fig. 15. VAE Output for Circle Input

In the implementation of VAE for the case of circle and cross 3x3 pixels, input is used in the form of a cross pattern represented in the form of a 3x3 matrix. This input uses binary values where 0 represents black pixels and 1 represents white pixels. The test results show that the implemented VAE architecture successfully reconstructs the pattern well, where the resulting output has significant similarity with the original input pattern.

The test results show that the VAE output produces floating point values between 0 and 1, in contrast to the input which is a binary value. This can be seen from the output values such as 0.935547, 0.374023, 0.939453, and so on. This output characteristic is normal in VAE architecture due to its probabilistic nature. When these output values are rounded to the nearest value (0 or 1), the cross pattern can still be reconstructed accurately.

The implementation uses a 1MHz clock, which is relatively slow due to the limitations of combinational circuits. Even so, this result is enough to prove that the designed VAE architecture can work well for reconstructing simple patterns. The successful implementation in this simple case is an important basis before further development for deraining cases that require image processing with much larger dimensions.

B. *Variational Auto Encoder for Image Deraining*

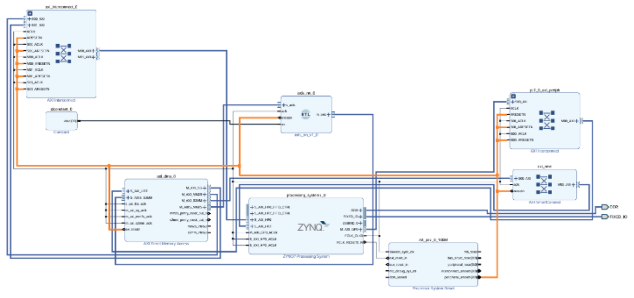


Fig. 16. Block Design of VAE for Image Deraining

The architecture of the VAE implementation for image deraining is similar in structure to the previous implementation but modified to handle more complex input images. The block diagram shows the system consisting of several interconnected main components. The first block serves as a memory controller that manages the input image data and training parameters. The middle part is the core of the VAE architecture which includes an encoder for data compression, a sampling block with the implementation of the reparameterisation trick, and a decoder for image reconstruction. The system is connected to the ZYNQ FPGA as the main processing unit through the PS-PU (Processing System - Processing Unit) interface. Unlike the previous implementation, this architecture is designed to process image data in 224×224 grayscale format divided into 4×4 patches for more efficient processing.

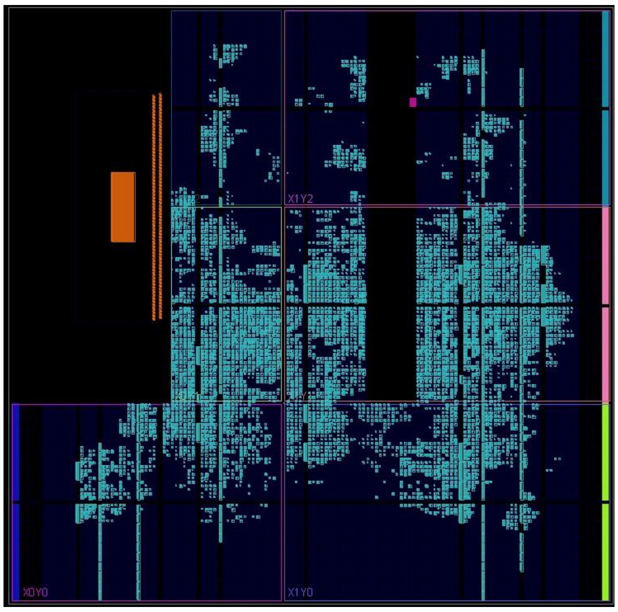
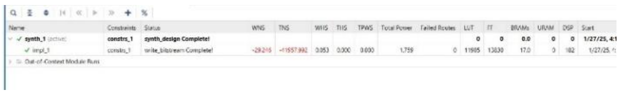


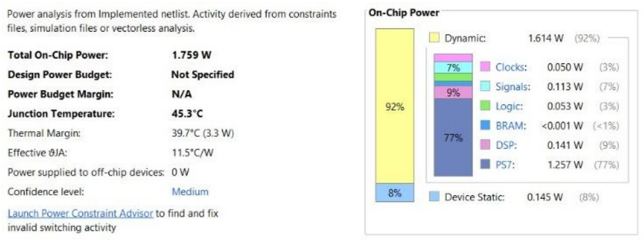
Fig. 17. Device of Block Design

Device block design visualises the layout of components on the FPGA after the implementation process. The layout shows the distribution and placement of various components such as LUTs, FFs, BRAMs, and DSPs used in the VAE implementation. Different colours in the figure represent different types of resources on the FPGA.



Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start
✓ synth_1 (active)	constrs_1	synth_design Complete!								0	0	0.0	0	0	1/27/25, 4:1
✓ impl_1	constrs_1	write_bitstream Complete!	-29.246	-41957.992	0.053	0.000	0.000	1.759	0	11905	13830	17.0	0	182	1/27/25, 4:
Out-of-Context Module Runs															

Fig. 18. Implementation of Block Design



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.759 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**45.3°C**
Thermal Margin:	39.7°C (3.3 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	1.614 W	(92%)
Clocks:	0.050 W	(3%)
Signals:	0.113 W	(7%)
Logic:	0.053 W	(3%)
BRAM:	<0.001 W	(<1%)
DSP:	0.141 W	(9%)
PS7:	1.257 W	(77%)
Device Static:	0.145 W	(8%)

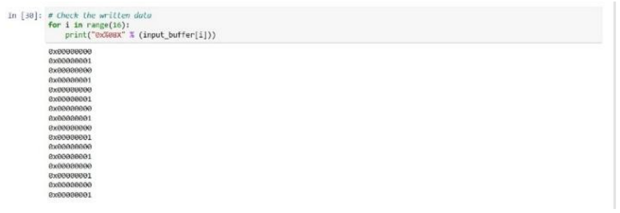
Fig. 19. Power Consumption of Block Design

The image displays the results of synthesis (synth_1) and implementation (impl_1) from an FPGA design process, showcasing metrics related to timing, resource utilization, and power consumption. The synthesis process completed successfully, as indicated by the status "synth_design Complete!" with no visible errors or critical warnings. During implementation, the status "write_bitstream Complete!" confirms that the final FPGA bitstream generation was successful, but there are significant timing issues that need to be addressed.

From the timing analysis, the Worst Negative Slack (WNS) is -29.246 ns, which indicates that certain paths fail to meet the required timing constraints by this margin. The Total Negative Slack (TNS) is -41,957.992 ns, highlighting a cumulative timing failure across multiple paths. On the positive side, the Worst Hold Slack (WHS) is 0.053 ns, and the Total Hold Slack (THS) is 0.000 ns, confirming that hold constraints are satisfied.

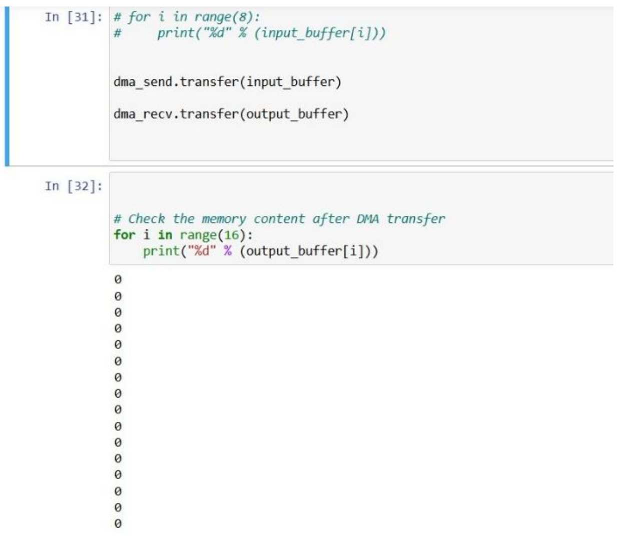
In terms of resource utilization, the design efficiently uses the FPGA resources. Specifically, 11,905 Look-Up Tables (LUTs) and 13,830 Flip-Flops (FFs) are utilized, which are well within the available limits. The usage of BRAMs (17) and DSPs (182) indicates that the design includes some memory and significant signal processing operations. These numbers suggest that the resource utilization is balanced and far below the device's maximum capacity, demonstrating hardware efficiency.

The total power consumption is reported at 1.759 W, which appears reasonable for the design. However, while the resource utilization is efficient, the negative timing slack remains a critical issue. To address this, further optimization is needed, such as improving critical path timing, reconfiguring placement and routing, or reducing clock frequency. Overall, the design demonstrates good resource usage but requires timing fixes to ensure it operates reliably at the desired clock speed.



```
In [30]: # Check the written data
         for i in range(16):
             print("0x%08X" % (input_buffer[i]))

         0x00000000
         0x00000001
         0x00000000
         0x00000001
         0x00000000
         0x00000001
         0x00000000
         0x00000001
         0x00000000
         0x00000001
         0x00000000
         0x00000001
         0x00000000
         0x00000001
         0x00000000
         0x00000001
```

Fig. 20. Input Jupyter

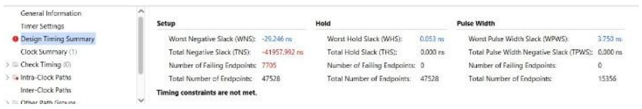


```
In [31]: # for i in range(8):
         #     print("%d" % (input_buffer[i]))


         dma_send.transfer(input_buffer)

         dma_recv.transfer(output_buffer)

In [32]:

         # Check the memory content after DMA transfer
         for i in range(16):
             print("%d" % (output_buffer[i]))

         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
```

Fig. 21. Output Jupyter

The provided images illustrate an issue with a DMA (Direct Memory Access) transfer process where the input buffer contains valid data, but the output buffer remains empty or zeroed after the transfer. The input buffer is populated with 16 values, each represented as 32-bit integers, indicating that the data is written correctly. However, the output buffer, which is expected to match the input data after the DMA transfer, shows only zeros, suggesting that the transfer either failed or the data was not handled properly. This issue could arise from several potential problems. First, there might be a buffer misalignment; if the DMA hardware expects data in 256-bit chunks (8 uint32_t values), the buffers must be structured accordingly. Second, there may be a mismatch in the transfer size, with the DMA configuration not aligning with the input or output buffer size. Third, the DMA controller itself might not be correctly configured for the desired transfer size or burst settings. Finally, there could be a software logic issue where the data is not correctly divided into 256-bit chunks or is incorrectly interpreted during the transfer process.

To address this, it is crucial to ensure that the input and output buffers are properly aligned to the DMA's expected data size. For example, the buffers should be divided into 256-bit chunks if the DMA requires such alignment. Additionally, the DMA configuration must be verified to ensure the transfer size and burst settings are consistent with the buffer structure. Debugging tools or status checks should also be implemented to detect any errors during the transfer process. Furthermore, output verification can be enhanced by displaying the buffer contents in groups of 256-bit chunks to ensure proper formatting. If the input data size does not match the required size for the DMA, padding can be added to align the data correctly. By implementing these fixes, the output buffer should match the input buffer after the DMA transfer, resolving the issue and ensuring successful data handling.



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-29.246 ns	Worst Hold Slack (WHS):	0.053 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	-41957.992 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	7705	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	47528	Total Number of Endpoints:	47528	Total Number of Endpoints:	15356

Timing constraints are not met.

Fig. 22. Timing of Block Design

The image presents a timing summary for a digital design, focusing on the Setup, Hold, and Pulse Width constraints. Here's an analysis in paragraph form:

The timing summary indicates that the design does not meet its timing constraints, as shown by the negative Worst Negative Slack (WNS) of -29.246 ns under the Setup category. This value represents the longest delay on any critical path, exceeding the allowed timing window by 29.246 ns, which contributes to a significant Total Negative Slack (TNS) of -41,957.992 ns. This large TNS implies that multiple paths are failing, with 7,705 failing endpoints out of a total of 47,528. While the Setup timing is critical and must be addressed, the Hold constraints are met, as evidenced by the positive Worst Hold Slack (WHS) of 0.053 ns and a Total Hold Slack (THS) of 0.000 ns, with no failing endpoints. Additionally, the Pulse Width Slack (WPWS) is 3.750 ns, with no violations, indicating that pulse-width constraints are satisfied across all 15,356 endpoints.

IV. Conclusion

This paper presents the implementation of Variational Autoencoder (VAE) architecture for deraining on FPGA. The implementation is done in two stages: first with a simple 3x3 pixels test case and then with a more complex image deraining case. Based on the results obtained, several conclusions can be drawn below.

The VAE implementation for the 3x3 pixels case shows promising results, where the architecture successfully reconstructs simple patterns. Using a 16-bit fixed-point representation (10 fractional bits, 5 integer bits, and 1 sign bit), the system achieves fairly good accuracy in pattern reconstruction, despite operating at a relatively low clock frequency of 1MHz due to combinational circuit limitations.

For image deraining applications, our implementation reveals some limitations. The 224×224 grayscale image processing approach using 4×4 patches proved not effective enough to remove the rain effect. Although the hardware implementation achieves efficient resource usage, with 11,905 LUTs and 13,830 FFs still well below the FPGA capacity limit, timing constraints remain a significant challenge, as seen from the Worst Negative Slack (WNS) of -29,246 ns.

Power consumption analysis shows a reasonable total consumption of 1,759W, indicating efficient hardware usage. However, our experiments show that the basic VAE architecture alone is not sufficient for complex image deraining tasks. Future development should consider using more advanced architectures such as CNN, ResNet, or attention mechanisms to improve deraining performance while maintaining hardware efficiency.

This implementation provides valuable insights into the challenges and considerations in applying deep learning models to resource-constrained embedded systems, especially highlighting the trade-off between architectural complexity and hardware limitations.

References

[1] R. Qian, R. T. Tan, W. Yang, J. Su, and J. Liu, "Attentive generative adversarial network for raindrop removal from A single image," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, IEEE, Jun. 2018, pp. 2482–2491. Accessed: Jan. 28, 2025. [Online]. Available: https://doi.org/10.1109/cvpr.2018.00263

[2] Y. Du, J. Xu, Q. Qiu, X. Zhen, and L. Zhang, “Variational Image Deraining,” in *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*, IEEE, Mar. 2020, pp. 2395–2404. Accessed: Jan. 28, 2025. [Online]. Available: https://doi.org/10.1109/wacv45572.2020.9093393