engineering proceedings

MDPI

Proceeding Paper

Hardware Implementation of an Autoencoder on a Field Programmable Gate Array †

Minh-Hieu Vo [1,2], Thien-Van Nguyen [1,2], Trong-Nhan Huynh [1,2], Tan-Phat Dang [1,2] and Huu-Thuan Huynh [1,2,*]

[1] Faculty of Electronics and Telecommunications, University of Science, Ho Chi Minh City 700000, Vietnam; 23c42004@student.hcmus.edu.vn (M.-H.V.); 20200414@student.hcmus.edu.vn (T.-V.N.); 20200286@student.hcmus.edu.vn (T.-N.H.); dtphat@hcmus.edu.vn (T.-P.D.)
[2] Vietnam National University, Ho Chi Minh City 720325, Vietnam
* Correspondence: hhthuan@hcmus.edu.vn
† Presented at the 9th Eurasian Conference on Educational Innovation 2026 (ECEI 2026), Da Nang City, Vietnam, 30 January–2 February 2026.

Abstract

An autoencoder is an unsupervised deep learning architecture designed to compress input data, extract meaningful features, and reconstruct the original input for applications such as anomaly detection and data compression. However, CPU-based implementations often suffer from limited performance and high power consumption. To address these challenges, this paper presents an FPGA-based autoencoder with a hardware-friendly neural network architecture optimized for both resource utilization and processing performance. In addition, optimization techniques such as network size reduction, quantization, and pipelining are applied to improve efficiency in real-time applications. The proposed autoencoder accelerator is integrated into a Nios II system to evaluate its effectiveness. Implemented on a Cyclone V 5CSXFC6D6F31C6 FPGA (Intel Corporation, San Jose, California, United States) at 50 MHz, the system occupies 81% of logic resources, 3% of memory blocks, and 3% of digital signal processing blocks. Experimental results show that, while an Intel Xeon CPU at 2.2 GHz requires more than 0.2 s to process a single handwritten digit from the Modified National Institute of Standards and Technology dataset, the proposed system performs the same task in approximately 4.5 milliseconds, providing a 44× speedup. This demonstrates the effectiveness of the proposed FPGA-based autoencoder accelerator.

Keywords: autoencoder; neural network; FPGA

1. Introduction

Autoencoders are unsupervised feed-forward neural networks designed to learn meaningful representations of input data without requiring labeled information [1]. By encoding high-dimensional inputs into compact latent representations and reconstructing them at the output, autoencoders aim to preserve essential information with minimal reconstruction loss. Owing to these capabilities, popular deep learning techniques are widely applied to data compression, anomaly detection, noise reduction, and data generation.

Despite their promising applications, implementing autoencoders on traditional CPUs and graphics processing units (GPUs) presents limitations. CPU-based implementations typically rely on multi-threaded execution and optimized numerical libraries, which provide flexibility and ease of deployment. However, their performance is constrained by limited parallelism in large-scale matrix operations, leading to high inference latency for

Check for updates

Academic Editors: Teen-Hang Meen, Chun-Yen Chang and Cheng-Fu Yang

Published: 10 June 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

computationally intensive autoencoder models [2]. In contrast, GPU-based implementations exploit massive parallelism to accelerate matrix multiplications and activation functions, achieving substantial speedups over CPU-based approaches [3]. Nevertheless, these designs often require substantial hardware resources and incur high power consumption, while additional data transfer overhead between memory and computing units further reduces efficiency [3,4]. As a result, GPU-based autoencoders are generally ill-suited for energy-constrained embedded and edge systems [4].

Recent autoencoder implementations on CPUs rely on optimized software execution but suffer from limited parallelism and high inference latency. GPU-based designs improve throughput through massive parallelism, at the cost of increased power consumption and memory overhead. In contrast, FPGA-based approaches achieve better energy efficiency using pipelining and reduced-precision arithmetic; however, most existing works lack system-level integration and quantitative analysis of accuracy trade-offs between fixed-point hardware and floating-point software models. This research gap motivates the proposed FPGA-based design.

In this study, we implement an autoencoder on an FPGA by carefully exploring hardware-friendly network architectures. Furthermore, applying hardware optimization techniques, such as network size reduction, fixed-point quantization, and pipelining, enables efficient utilization of FPGA resources. The proposed approach achieves a favorable balance among reconstruction accuracy, hardware resource usage, inference latency, and power consumption on an FPGA platform.

2. Autoencoder Model Development

2.1. Autoencoder Model Architecture

The autoencoder model is constructed using the sequential architecture in the Keras framework [5] (Figure 1). It consists of multiple dense (fully connected) layers arranged symmetrically to form an encoder–decoder structure. The detailed configuration of the model is described as follows:

- The input layer has a dimensionality of 784, corresponding to the 784 pixels of the input image (28×28);
- The first hidden layer (Encoder) is a Dense layer with 128 neurons, employing the ReLU activation function;
- The second hidden layer (Encoder) is a Dense layer with 64 neurons, using the ReLU activation function;
- The third hidden layer (Latent layer) contains 32 neurons with the ReLU activation function and serves as the compressed latent representation of the input data;
- The fourth hidden layer (Decoder) is a Dense layer with 64 neurons, using the ReLU activation function;
- The fifth hidden layer (Decoder) is a Dense layer with 128 neurons, employing the ReLU activation function;
- The output layer consists of 784 neurons and uses the sigmoid activation function to reconstruct the input data.

The model is compiled using the Adam optimizer [6] and the binary cross-entropy loss function [7]. It is trained on the Modified National Institute of Standards and Technology (MNIST) dataset for 100 epochs with a batch size of 256. During training, the data are shuffled to improve generalization, and a validation dataset is used to evaluate the model at each epoch.

https://doi.org/10.3390/engproc2026141011

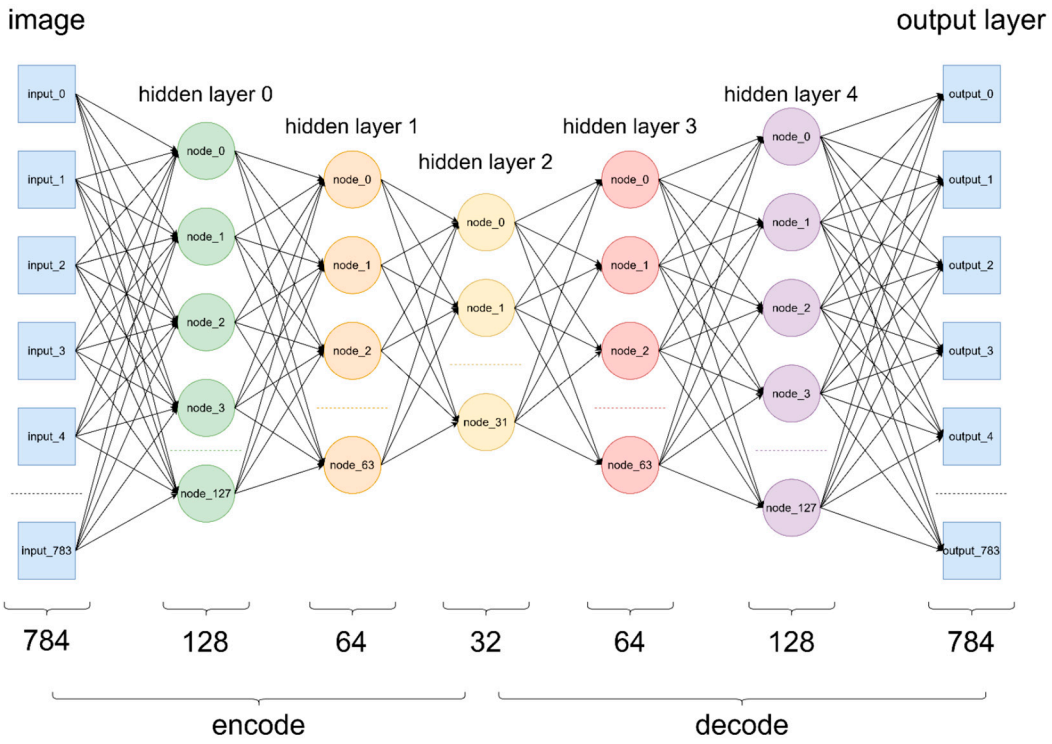


Figure 1. Autoencoder model constructed in this study.

The C model is reimplemented in C, based on the autoencoder model originally developed in Python 3.5. This reimplementation follows each computational step of the inference process, serving as a reference model for subsequent hardware design. According to the C model, the computation of each hidden layer in the autoencoder consists of matrix operations, including weight multiplication and bias addition, followed by the application of an activation function to generate the output, which is then used as the input for the next layer. This process is repeated sequentially until the input data propagates through all layers, producing the autoencoder's final output.

2.2. Quantization Aware Training (QAT)

QAT is used to prepare deep learning models for deployment on resource-constrained hardware platforms such as FPGAs, microcontrollers, and mobile devices [8]. The objective of QAT is to reduce numerical precision in arithmetic operations, typically converting 32-bit floating-point representations to 16-bit integers or lower. By explicitly modeling quantization effects during training, QAT enables deep learning models to maintain high performance when executed on hardware with limited computational and memory resources. By compensating for quantization-induced errors during training, QAT plays a critical role in optimizing models for practical real-world applications.

QAT is applied to the autoencoder model using the TensorFlow Model Optimization Toolkit to enable 16-bit integer quantization for efficient FPGA deployment. The model is initially trained using 32-bit floating-point. In QAT, quantization operations are inserted between layers to simulate the effects of reduced precision on weights and activations. The model is then retrained to compensate for quantization-induced errors, thereby improving hardware efficiency while maintaining reconstruction performance.

3. Autoencoder Design

3.1. Autoencoder Core

The autoencoder hardware consists of several tightly coupled functional blocks (Figure 2). The FSM block controls and coordinates the entire system's operation. The W_REG_ARRAY block stores the weights and biases transferred from the DMA via a FIFO interface, while the REGN block holds the input data. The PE_ARRAY block, comprising 128 processing elements, performs the core computations of the autoencoder. Intermediate results are stored in the BACKUP_ARRAY and SHIFT_ARRAY blocks to support data reuse and serialization. Finally, the activation function block applies a nonlinear activation and produces the output data.

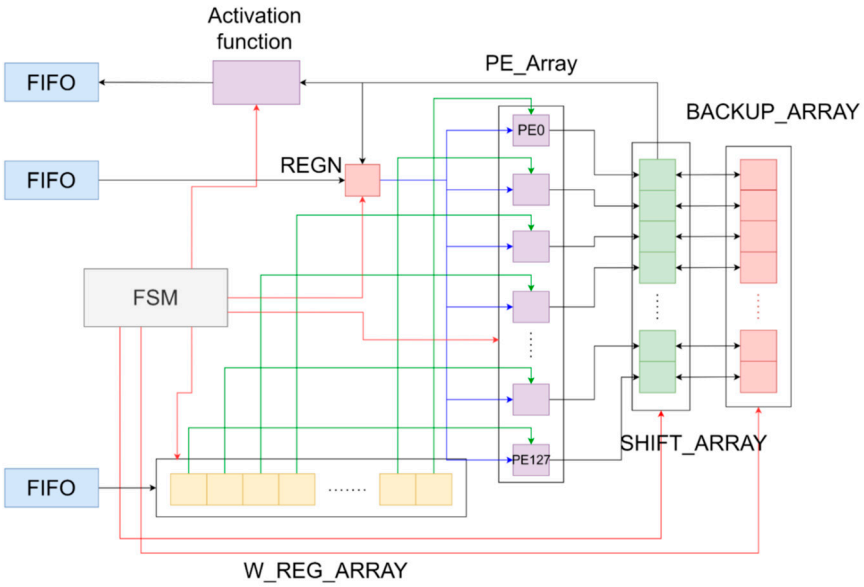


Figure 2. Autoencoder core is used in this study. Note: Signal paths are color-coded as follows: red lines represent control signals from the FSM; green lines indicate weight and bias loading paths; blue lines denote the input image data path; and black lines signify the primary datapath.

3.1.1. REGN Block

The REGN block functions as a register with two data input sources. The first source is the input data provided by the DMA, while the second source is data returned from the SHIFT_ARRAY block, which is used as input to subsequent computational stages (Figure 3).

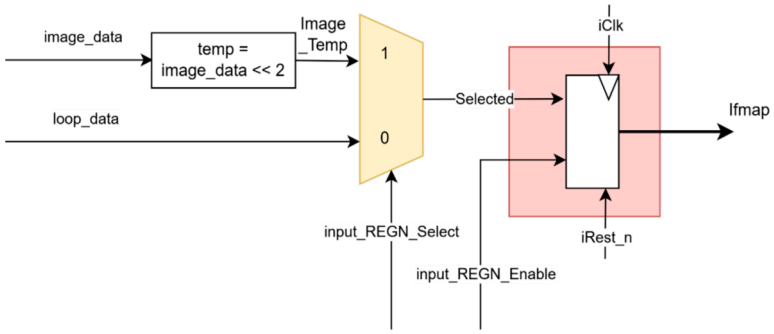


Figure 3. REGN block.

3.1.2. PE_Array Block

PE_array consists of 128 processing elements (PEs), as illustrated in Figure 4, and is responsible for performing the core computations of the autoencoder. Each PE receives

input data, weights, and bias values provided by the W_REG_ARRAY, SHIFT_ARRAY, and REGN blocks, respectively. The computation process involves multiplying the input data by the corresponding weights, accumulating the result with the bias term, and storing the intermediate result in an accumulator register.

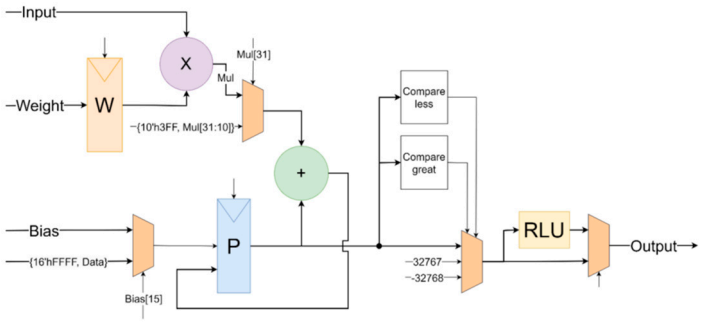


Figure 4. PE_Array block. Note: The notation [] indicates bit-selection (for a single bit) and part-selection (for a bit-string).

The PEs are designed to operate on 16-bit fixed-point data. Internally, each PE includes a 16-bit integer multiplier and a 32-bit integer adder. Additionally, two comparator units are integrated to detect and handle potential overflow conditions. After computation, the output data are optionally passed through a ReLU activation block when enabled. The computed results are also written to the SHIFT_ARRAY and BACKUP_ARRAY blocks, depending on the control signals generated by the finite state machine.

3.1.3. Activation Function Block

The Activation Function block is responsible for performing the activation operation and producing the output of the autoencoder. This block is designed to operate with 16-bit fixed-point arithmetic (Figure 5). It incorporates integer comparison units to determine the appropriate activation interval. Once the corresponding interval is selected, integer adders are used to compute and generate the final output value.

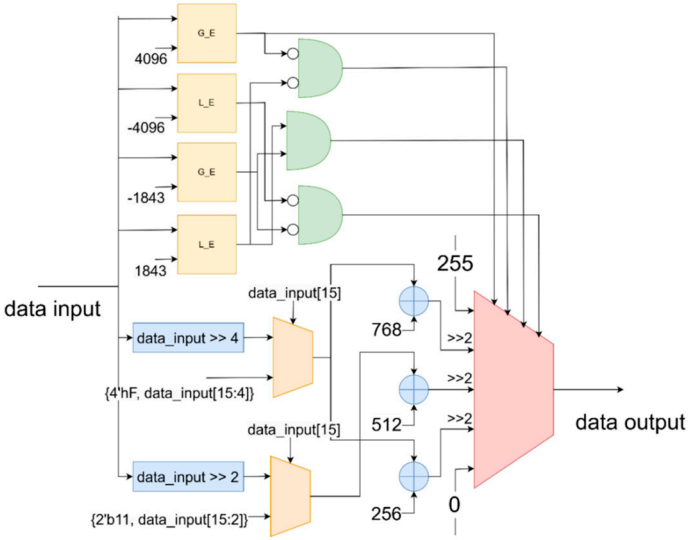


Figure 5. Hardware implementation of the sigmoid approximation function. Note: The notation [] indicates bit-selection (for a single bit) and part-selection (for a bit-string).

Implementing complex mathematical functions such as the sigmoid function on FPGA platforms presents several challenges. FPGAs employ fixed-point arithmetic to reduce hardware resource consumption, whereas the sigmoid function requires high numerical

precision, particularly within the output range of [0, 1]. This discrepancy necessitates the design of sophisticated arithmetic algorithms to maintain acceptable accuracy. Moreover, accurate sigmoid computation requires substantial hardware resources, including lookup tables, flip-flops, and multipliers, which can significantly increase the FPGA's area utilization and power consumption. In addition, the computational complexity of the sigmoid function can introduce significant latency, affecting processing speed and overall system performance.

3.2. System Integration and Operational Flow

3.2.1. System Integration

The developed autoencoder core is integrated into a Nios II system and deployed on the Cyclone V SE 5CSXFC6D6F31C6N FPGA platform (Figure 6). This integrated system is designed to evaluate the autoencoder hardware module in an SOPC environment, particularly its interactions with other system components and its ability to produce and collect processing results.

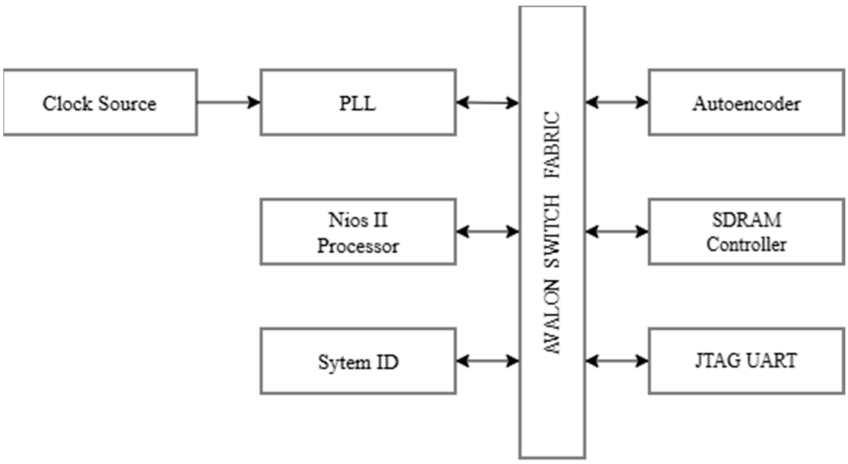


Figure 6. Integrated system in this study.

A system configuration flow is established as follows.

1. Initialize the input image array required for the autoencoder computation. Since a large portion of the 784 input image pixels are zero, computations involving zero-valued inputs are skipped to improve processing speed. This optimization reduces unnecessary weight memory accesses and Avalon Switch Fabric transactions, eliminating up to 128 multiply–accumulate operations and 256 bytes of weight data reads per zero-valued pixel;
2. Create two pointers corresponding to the hardware module offset address and the SDRAM offset address, storing the weight and bias parameters;
3. Configure the autoencoder hardware module for operation;
4. Set the memory addresses of the input image array and the output image buffer;
5. Configure the memory addresses for the weight and bias parameters stored in SDRAM.
6. Write a value of 1 to the control register to trigger the hardware computation.

3.2.2. Operational Flow

In the first layer, input image data are loaded into the REGN register, while weights and biases are transferred from memory to PEs through the W_REG_ARRAY block. Each PE performs multiply–accumulate operations, accumulating partial sums until all input data

are processed. The resulting outputs of the first layer are then stored in the SHIFT_ARRAY register array.

For subsequent layers, weights and biases are loaded in the same manner, whereas the input data for REGN is sequentially fetched from the values stored in SHIFT_ARRAY. In the final layer, because the output dimension (784) exceeds the number of available PEs, the computation is executed in 7 iterations, producing 128 outputs per iteration, except for the last iteration, which generates 16 outputs. Before being written back to memory, the computed outputs pass through the activation function block and are buffered in a FIFO. Intermediate results of the previous layer are stored in the BACKUP_ARRAY and restored to the SHIFT_ARRAY when required to complete the final layer computation.

4. Results and Discussions

The samples processed by the autoencoder hardware in simulation produce results that are consistent with those obtained from the developed C model (Figure 7).

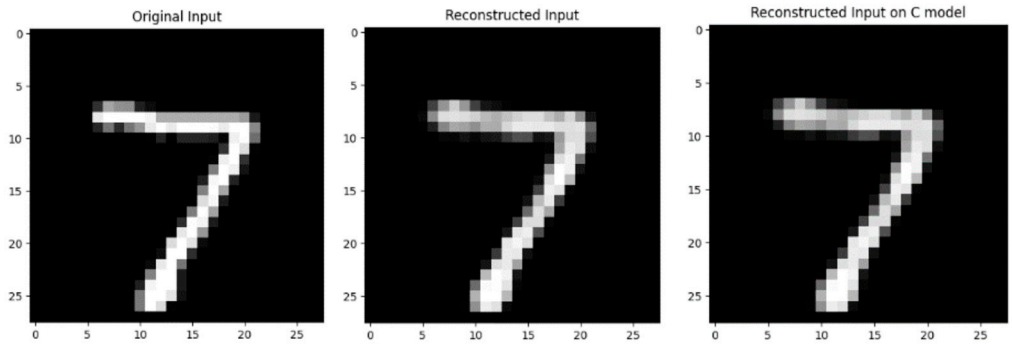


Figure 7. Hardware simulation output and corresponding C reference model output.

The peak signal-to-noise ratio (PSNR) measured across 100 test samples between the 16-bit fixed-point autoencoder hardware and the 32-bit floating-point software implementation indicates that the proposed autoencoder hardware achieves an accuracy of 98% (Figure 8).

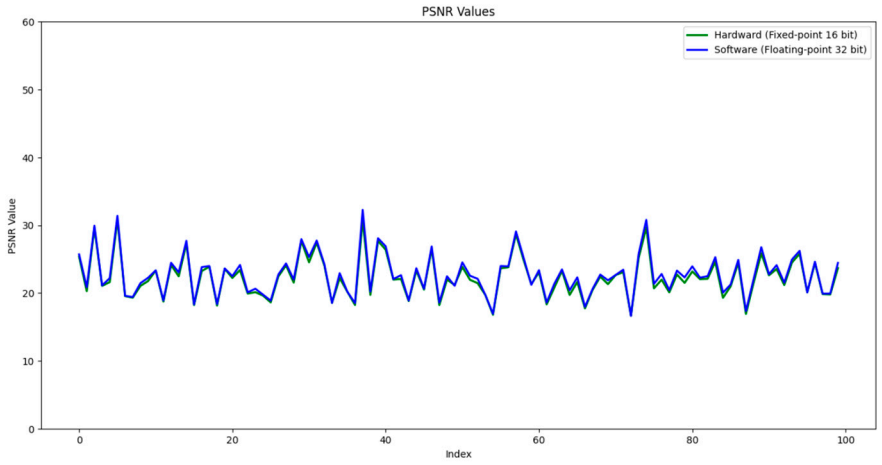


Figure 8. PSNR values of the hardware and software autoencoder models over 100 test samples.

The overall system synthesis results on the Cyclone V SE 5CSXFC6D6F31C6N FPGA are presented in Table 1. The implementation demonstrates a high demand for computational logic, with logic utilization reaching 34,004 units, which accounts for 81% of the total available resources. The design also utilizes a total of 88,751 registers. In terms of memory and arithmetic hardware, the system consumes 4,219,008 total block memory bits,

representing a 75% proportion, while the dedicated RAM and DSP blocks exhibit a minimal footprint of only 3% each.

Table 1. Resource utilization of the implemented autoencoder system on an FPGA.

Resource	Integrated System	Proportion
Logic utilization	34,004	81%
Total registers	88,751	-
Total block memory bits	4,219,008	75%
RAM	-	3%
DSP block	-	3%

We compare the performance of the results with those of CPU and GPU platforms, and other FPGA-based implementations. The comparison results are presented in Table 2.

Table 2. Comparison of the results of different platforms.

Platform	Technology	Frequency	Latency
Intel Xeon	CPU	2.20 GHz	0.2 s
NVIDIA GTX 1080 TI [3]	GPU	1.48 GHz	6.15 ms
Zhang et al. [3]	FPGA (Xilinx)	100 MHz	15.73 ms
Ours	FPGA (Cyclone V)	50 MHz	4.5 ms

5. Conclusions

We present an FPGA-based hardware design of an autoencoder for handwritten digit recognition using the MNIST dataset. In developing the system, we apply hardware-oriented optimization techniques, including network size reduction, fixed-point quantization, and pipelining, to improve execution efficiency and resource utilization. The proposed design, implemented on an Altera Cyclone V FPGA operating at 50 MHz, processes a single MNIST image in approximately 4.5 ms. In contrast, our CPU-based implementation on an Intel Xeon processor at 2.2 GHz takes more than 0.2 s for the same task, yielding a performance improvement of about 44×. These results demonstrate the effectiveness of the proposed FPGA-based approach for accelerating autoencoder inference on handwritten digit data. In addition to performance evaluation, the system also serves as an educational platform to illustrate the integration of machine learning algorithms with hardware design in SoPC-based FPGA systems.

Author Contributions: Conceptualization, T.-V.N. and T.-N.H.; software, M.-H.V., T.-V.N. and T.-N.H.; validation, M.-H.V., T.-V.N. and T.-N.H.; writing—original draft preparation, M.-H.V.; writing—review and editing, T.-P.D. and H.-T.H.; supervision, T.-P.D. and H.-T.H.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Hinton, G.E.; Salakhutdinov, R.R. Reducing the dimensionality of data with neural networks. *Science* **2006**, *313*, 504–507. [CrossRef]
2. Buber, E.; Banu, D.I.R.I. Performance analysis and CPU vs. GPU comparison for deep learning. In Proceedings of the 2018 6th International Conference on Control Engineering & Information Technology (CEIT), Istanbul, Turkey, 25–27 October 2018; pp. 1–6. [CrossRef]
3. Zhao, W.; Jia, Z.; Wei, X.; Wang, H. An FPGA implementation of a convolutional auto-encoder. *Appl. Sci.* **2018**, *8*, 504. [CrossRef]
4. Isik, M.; Oldland, M.; Zhou, L. An energy-efficient reconfigurable autoencoder implementation on FPGA. In *Proceedings of SAI Intelligent Systems Conference*; Springer Nature: Cham, Switzerland, 2023; pp. 212–222. [CrossRef]
5. Chollet, F. Building Autoencoders in Keras. *Keras Blog*, 14 May 2016. Available online: https://blog.keras.io/building-autoencoders-in-keras.html (accessed on 10 December 2025).
6. Kinga, D.; Adam, J.B. A method for stochastic optimization. In Proceedings of the 3rd International Conference on Learning Representations (ICLR), San Diego, CA, USA, 7–9 May 2015; Volume 5.
7. Bishop, C.M.; Nasrabadi, N.M. *Pattern Recognition and Machine Learning*; Springer: New York, NY, USA, 2006.
8. Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; Narayanan, P. Deep learning with limited numerical precision. In Proceedings of the 32nd International Conference on Machine Learning (ICML), Lille, France, 6–11 July 2015; pp. 1737–1746.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.