Implementing Convolutional Auto Encoder on FPGA using High Level Synthesis

Son Dao-Xuan*, Khoi Nghiem-Tuan*, Anh Ong-Tung*, Chi Hoang-Phuong*†, Minh Nguyen-Duc*
*School of Electrical and Electronic Engineering, Hanoi University of Science and Technology, Vietnam
†Corresponding Author: chi.hoangphuong@hust.edu.vn

***Abstract*—This study introduces a CNN-based autoencoder implementation on FPGA using High-Level Synthesis (HLS), focusing on optimizing convolutional computing for reduced latency and enhanced resource efficiency. Implemented on the FPGA, the design utilizes 69,407 Flip-Flops (FFs), 104,467 Look-Up Tables (LUTs), and 3,716 Digital Signal Processors (DSPs), achieving an operational frequency of 100 MHz and a latency of 0.15 milliseconds. Through techniques such as pipelining, parallelization, and loop unrolling, we navigate the balance between resource utilization, computational speed, and image reconstruction fidelity. Our findings demonstrate the potential of HLS for efficiently deploying deep learning models on hardware, offering valuable insights for advancements in real-time processing applications.**

***Index Terms*—FPGA, Convolutional Neural Network, HLS, Auto encoders, Optimization**

I. Introduction

In recent years, Convolutional Neural Networks (CNNs) have marked significant advancements across a wide spectrum of domains including computer vision [1]–[3], speech recognition [4], and natural language processing [5], as evidenced by seminal works in these areas. Among the diverse architectures within the CNN paradigm, autoencoders, in particular, have attracted considerable attention due to their proficiency in capturing detailed representations of input data through a dual-network architecture comprising an encoder and a decoder. This architecture enables many applications such as data compression, inpainting, and image reconstruction, demonstrating its versatility and effectiveness [6].

Despite the impressive achievements of CNN-based autoencoders in software, their deployment on hardware platforms, especially Field Programmable Gate Arrays (FPGAs), introduces a set of challenges primarily centered around minimizing latency and optimizing resource allocation [7]–[9]. FPGAs, with their inherent parallel processing capabilities, offer a promising avenue for enhancing the performance of convolutional networks. High-Level Synthesis (HLS) emerges as a vital technology in addressing the aforementioned challenges. HLS facilitates the automatic transformation of high-level algorithmic descriptions into hardware configurations, streamlining the development and refinement of complex computational models on hardware platforms. This approach is particularly advantageous for CNN-based autoencoder implementations, offering a rapid prototyping pathway to intricate algorithm designs.

This paper enroll in the exploration of a novel design strategy for the efficient implementation of CNN-based autoencoders on FPGA platforms using HLS. Focusing on image reconstruction application, we propose an optimized convolutional computing method tailored for the FPGA series. Our research delves into the utilization of HLS for the synthesis, another way for computing convolutional and fine-tuning of a CNN autoencoder, aiming to achieve optimal hardware performance. By employing optimization techniques such as pipelining, parallelization, and loop unrolling, our study seeks to diminish latency and enhance resource efficiency while preserving the fidelity of image reconstruction. Through this investigation, we aim to illuminate the complex interplay between hardware resource constraints, computational latency, and reconstruction quality, thereby contributing valuable insights into the design and optimization of hardware-accelerated CNN autoencoders.

In evaluating our design's deployment on FPGA platforms, we observed a resource allocation comprising 69,407 Flip-Flops (FFs), 104,467 Look-Up Tables (LUTs), and 3,716 Digital Signal Processors (DSPs). The system achieved an operational frequency of 160 MHz. Additionally, the implementation showcased an exceptionally low latency, measuring 0.15 milliseconds.

The rest of this paper is as follows. Section II introduces the HLS design methodology and tools being used. In section III, the proposed architecture and optimization solutions are described in detail. Experiment results on the MINIST dataset are shown in Section IV, and the last conclusion is proposed in Section V.

II. HLS Design Methodology

High-Level Synthesis (HLS) is a digital design method that converts a high-level description of a hardware system into a *Register-Rransfer Level (RTL)* implementation. This approach allows designers to work more abstractly, using languages like C, C++, or SystemC to automatically generate hardware descriptions, usually in Verilog or VHDL. HLS tools are designed to bridge the gap between software and hardware design, enabling quicker development cycles and more efficient optimization. HLS also provides automated optimization through compiler pragmas, which direct the HLS engine to produce RTL code that conforms to specific implementation styles, such as loop and tiling structures, function interfaces, pipelining, inlining, and resource instantiations. This automation in code transformation through HLS enables designers to efficiently implement complex functions on FPGAs.

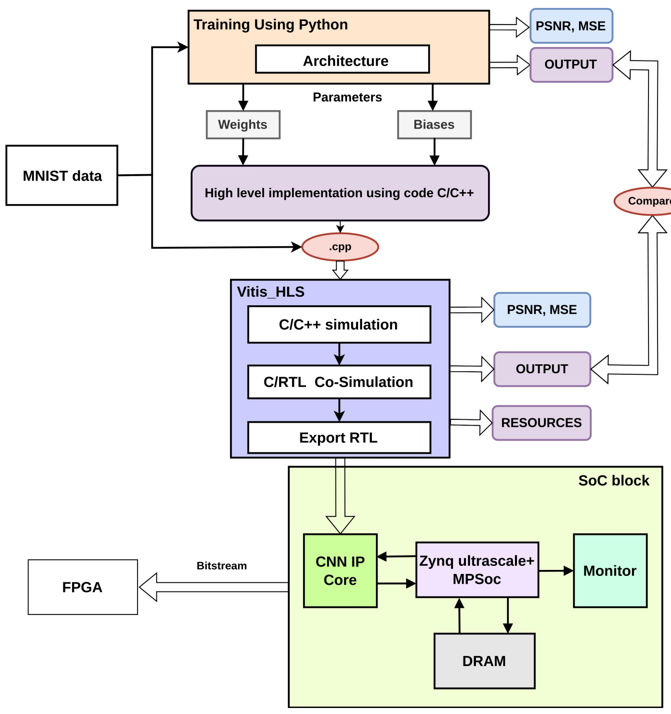


Fig. 1. High Level Synthesis for CNN implementation on FPGAs

Utilizing the High-Level Synthesis (HLS) approach for deploying CNNs on FPGAs, our proposed design methodology, depicted in the block diagram of Figure 1, encompasses critical stages: (1) construction of the CNN model; (2) formulation of a high-level description; (3) execution of high-level synthesis. It is imperative to conduct thorough validation checks between adjacent stages to confirm the equivalence of descriptions across the steps. The following provides an in-depth examination of each stage.

Initially, a CNN-based autoencoder model is developed and refined utilizing the Python-based Keras framework. Optimization of the model is conducted with an emphasis on key performance metrics, specifically *Peak Signal-to-Noise Ratio (PSNR)* and *Mean Square Errors (MSE)*. This phase culminates in the derivation of essential CNN parameters, including weights and biases. Moreover, this stage generates the 'golden outputs' of the CNN, which serve as benchmarks for validating the accuracy and integrity of subsequent processes.

Following the initial construction of the CNN-based autoencoder architecture, a corresponding C++ description is crafted in accordance with high-level synthesis (HLS) guidelines. This C++ representation is then processed using an HLS synthesis tool to evaluate performance metrics, including PSNR and MSE, ensuring its alignment with the original Python-based CNN model. Subsequently, this C++ description undergoes synthesis into a Register Transfer Level (RTL) description utilizing Verilog/SystemVerilog. A co-simulation of the RTL description with the C++ code is performed to verify the equivalence of these two representations. After this verification, the RTL description is synthesized and deployed onto an FPGA, allowing for the estimation of hardware resources such as the number of Look-Up Tables (LUTs), Flip-Flops (FFs), and Digital Signal Processors (DSPs). This iterative process, from C++ description to RTL synthesis, may be repeated to refine hardware efficiency, operational speed, and power consumption.

Finally, the validated and refined RTL description of the autoencoder is encapsulated into an Intellectual Property (IP) block, tailored for incorporation into the Multi-Processor System-on-Chip (MPSoC) architecture of the FPGA. This crucial step involves the creation of the IP block, which comprises the autoencoder and its bus interface, facilitating seamless integration and software control within the MPSoC environment. This integration enables the demonstration and operational validation of the autoencoder, showcasing its effectiveness and efficiency within the FPGA-based system.

III. CNN-based Autoencoder High Level Design

A. *CNN-based Autoencoder Architecture*

An autoencoder is a particular neural network type primarily crafted to encode input data into a compressed and meaningful representation and subsequently decode it back to reconstruct the input as closely as possible to the original. Its principal objective is to learn, unsupervised, a "meaningful" representation of the data that can be leveraged for diverse applications such as clustering. The problem, formally defined in [10], entails learning the functions $A : \mathbb{R}^n \to \mathbb{R}^p$ (encoder) and $B : \mathbb{R}^p \to \mathbb{R}^n$ (decoder) that satisfy as Eq. 1.

$$\arg\min_{A,B} E[\Delta(\mathbf{x}, B \circ A(\mathbf{x})] \tag{1}$$

where E is the expectation over the distribution of x and Δ is the reconstruction loss function, which measures the distance between the output of the decoder and the input.

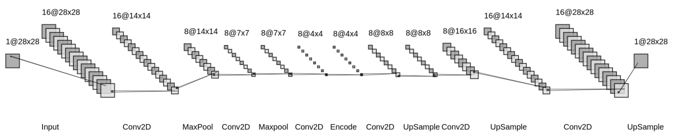


Fig. 2. CNN-based Autoencoder Architecture

In this work, we propose a CNN-based autoencoder architecture, as shown in Figure 2, which encompasses seven convolutional layers, three pooling layers, and three upsampling layers, systematically organized into two principal segments: the encoder and the decoder. Each layer within this network applies the Rectified Linear Unit (ReLU) activation function, transitioning to the sigmoid activation function for the concluding layers. The process begins with an input image dimension of 28×28, where the encoding phase efficiently compresses the data to a representation nearly ten times smaller than its original size, setting the stage for precise reconstruction in the decoding phase. This autoencoder model was trained on Keras's MNIST handwritten digit dataset with $60,000$ training images and $10,000$ test images. The training and test results were $PSNR = 62.08dB$ and $MSE = 0.008$.

To develop a high-level C++ description that adheres to HLS guidelines, several critical aspects must be considered: (1) streaming data into the design and between the computational layers, and (2) efficiently computing the convolution, max pooling, and upsampling layers. In the sections that follow, we will explore a range of strategies designed to refine the high-level C++ description. These approaches are targeted at optimizing the efficiency of the resultant RTL description, focusing on improving both hardware resource efficiency and overall performance.

B. Data streaming using FIFO buffers

In our design, both the input to the system and the data transferred between consecutive layers are structured as 3-D matrices, with the Z-axis representing the number of input channels and the X, Y axes representing the length and width of a 2D input data layer respectively as illustrated in Figure 3. However, processing a complete 3-D matrix in the hardware design would require extensive signal connections between layers, which is impractical. Additionally, employing buffer memory storage at the input of each layer is not viable due to the significant amount of underutilized memory storage it would require. Data synchronization between layers also presents a considerable challenge in the design process. To address these issues, we employ data streaming to facilitate the transmission of data into the design and between layers. This approach is implemented in hardware using first-in-first-out (FIFO) buffers, allowing for efficient and orderly data handling. The input data stream will be organized sequentially by each channel, meaning the data will be extracted along the Z-axis as depicted in Figure 3 . The data within the input stream will be represented as 000, 001, ..., 00Z with different colors for each channel. Subsequently, the data will continue to push through the rows and columns until the entire data set is transmitted.

To enable continuous data processing with streamed data, a buffer system is essential for recycling data for future computations. This buffer is capable of tracking the position of the input data, enabling the establishment of a sliding window mechanism. This simplifies the execution of convolutional operations by effectively managing the flow and processing of data within the system.

C. Convolutional Layers

The convolutional layer extracts features from input matrix data and transform into other images. In the convolutional layers of 2D CNNs, both input and output feature data consist of multiple channels. The output $Y[m][h][w]$ will be computed from the input through the function:

$$\varphi\left(\sum_{c=0}^{c-1}\sum_{r=0}^{k-1}\sum_{c'=0}^{k-1} W[m][c][r][c']\cdot X[c][h+r][w+c'] + B[m]\right)$$

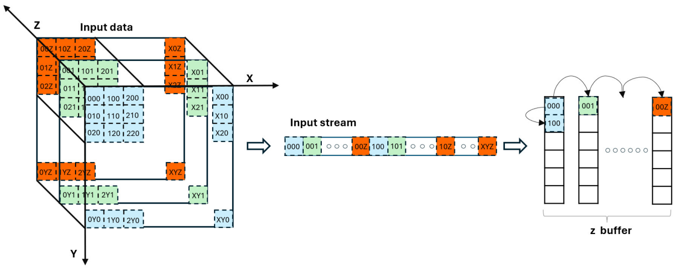


Fig. 3. The arrangement of data input in the buffer

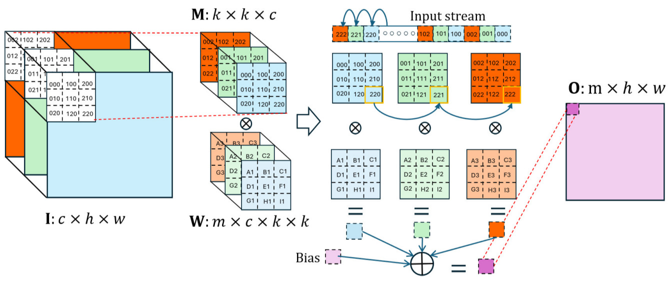
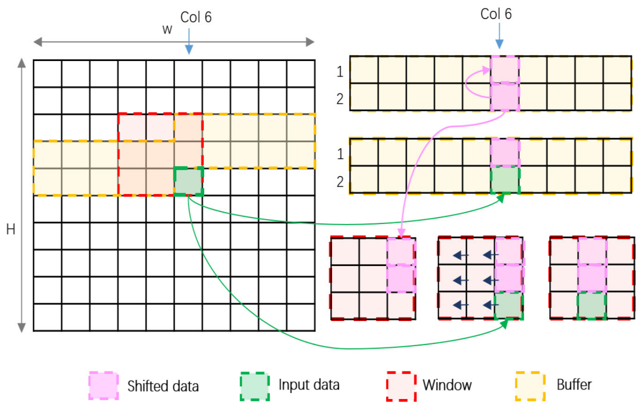


Fig. 4. Compute in the 3 dimension of the convolution layer

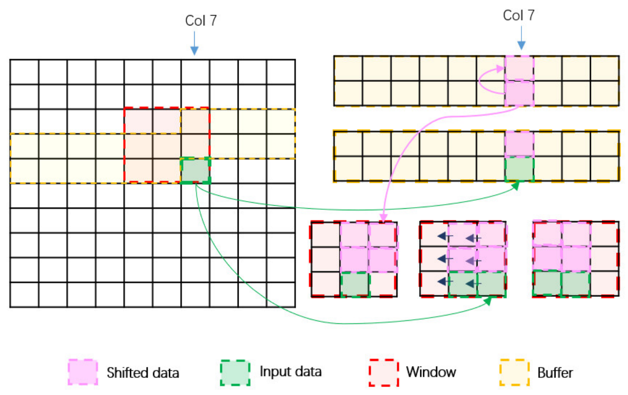
where I represents the input feature with a size of $c\times h\times w$. M represents the window extracted from the input with a size of $k\times k\times c$, and W denotes the weights used in this layer to convolve with the window, with a size of $m\times c\times k\times k$. O represents the output with a size of $m\times h\times w$. Here, w denotes the number of input channels, m denotes the number of output channels, h and w denote the height and width of each feature, and $k\times k$ represents the size of a kernel or window. B represents the bias and φ is a ReLU function. The 2D convolution operation will be performed sequentially on each channel until all data is processed, which will be elaborated on in detail in the subsequent section.

1) 2 line buffer: Regarding saving input data into buffer and forming window, a special technique used in our research is 2-line buffer. This approach optimizes data storage and computation compared to storing the entire input data. This technique involves storing the input values into a buffer and performing calculations on the subsequent data. The newly updated data will always be stored in the second row of the buffer and has the same column as the input data. The previously stored data will be shifted by one row, as illustrated in Figure 5(a).

For updating values for the window, it requires values from both the buffer and the input data. Additionally, the data in line 1 and line 2 in column 3 of the window, takes values from the two lines of the buffer corresponding to the column the same as the column of input data. Then, when the new input data is read, it is stored in line 2, column 2 of the window. In the next stage, as illustrated in Figure 5(b), when new input data is obtained, the window shifts its columns to the left and updates data With a detailed description of pseudo code in Algorithm 1.



(a)



(b)

Fig. 5. Performing 2-line buffer with high-level programming

Algorithm 1 2-line Buffer Pseudo Code

```
$i \leftarrow 0$
while $i \leq$ height do
    $j \leftarrow 0$
    while $j \leq$ width do
        $k \leftarrow 0$
        while $k \leq$ filter_in do
            // Shift values in the window horizontally
            $m \leftarrow 0$
            while $m \leq$ win_col do
                $n \leftarrow 0$
                while $n \leq$ win_row do
                    if win_row $< K - 1$ then
                        // Shift left and update data in window row 0,1
                    else if win_row $= K - 1$ then
                        // Shift left and update data in window row 2
                    end if
                    $n \leftarrow n + 1$
                end while
                $m \leftarrow m + 1$
            end while
            // Shift data vertically to update 2-line buffer row 1
            $i_{\text{range}} \leftarrow 0$
            while $i_{\text{range}} < (K - 1)$ do
                // Update data 2-line buffer in row 2
                $i_{\text{range}} \leftarrow i_{\text{range}} + 1$
            end while
            $k \leftarrow k + 1$
        end while
        $j \leftarrow j + 1$
    end while
    $i \leftarrow i + 1$
end while
```

2) Compute 3D Convolutional: The convolutional operation will be further explained as depicted in Figure 6. The 2D convolution is conducted using weights and a window size of $k \times k$, where $k = 3$ in this study. Starting from the window created using the 2-line buffer, the 2D convolutional operation is executed with the weights. Here, c denotes the input channel and m signifies the output channel, as previously discussed in the convolution layer section. Following convolution, the resulting data is temporarily stored in a buffer with a size of $1 \times m$. Subsequent input data updates the window in the next channels, with the green window continuing the convolution operation with the corresponding green kernel, and the resulting data is appended to the saved data buffer. This process iterates until the input data reaches the last channel. At this point, the sum of the data from the saved data buffer is concurrently updated to the output, and the subsequent cycle repeats the same steps until completion.

3) Max Pooling Layer: The maximum value of the input values within each pooling zone is the output value for that region in max pooling [11]. As a result, less pertinent data is discarded while the most notable traits in each pooling zone are preserved. High-level synthesis helps process data more efficiently by reading it sequentially into and storing it in buffers and windows, much like convolutional layers. The window will store the data values for comparison to determine the maximum value. The buffer in this pooling layer comprises a single row with n columns, where n is the number of columns in the input.

As seen in Figure 7, the input data set is split up into sizable 2×2 squares for easy comparison and data processing in the max pooling layer. In terms of how the data input stream works, for instance, if we have input data that is 33 in column 3, the previous value in column 3, which is 23, will be placed in the window at row 1, column 2 before updating the value 33 in the buffer. Ultimately, the input value will be updated in the window's row 2, column 2, and buffer column 3 sequentially. However, the image illustrates that comparisons will be made here using a skip step of 2, and only comparisons will be made when the window and the data of those squares overlap. The data in column 2 of the window will shift left by one square and start updating the buffer and window in the subsequent data update state. A comparison will be performed, and the largest value will be the result once the window has been updated with the accurate numbers for the 2×2 squares.

Using this strategy, the function can output the highest values found in each window while drastically lowering the input size four times compared to the original size.

4) Upampling Layer: Our upsampling function performs this task to decode compressed images back to their original

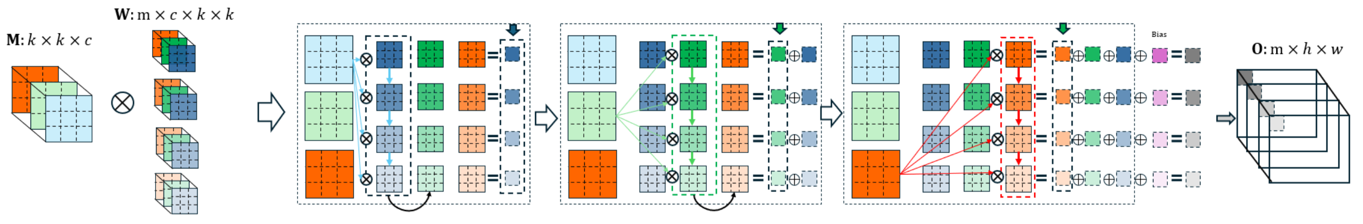


Fig. 6. Description of the process of 3D convolution with output saving

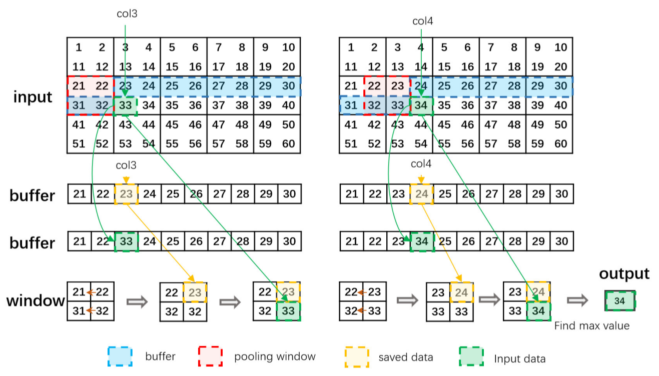


Fig. 7. Performing max pooling layer with high-level programming

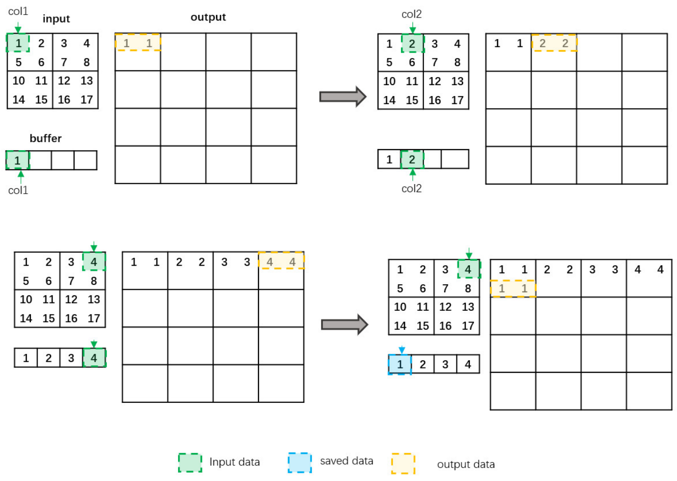


Fig. 8. Performing upsampling layer with high-level programming

size. Figure 8 illustrates how the function operates. A buffer with a size of $1 \times input\_width$ is utilized to support the upsampling process. The buffer stores the input values to facilitate reproducing the input values into the output. For example, with an input value at column 1, row 1 being 1, this value is stored in the buffer, and both row 1, column 1, and column 2 are set to 1. Subsequently, when the input is 2, the next two columns, 3 and 4, are updated with the input value, and similarly, as the buffer is updated to the end of row 1, row 1 of the output is also fully updated. In the next cycle, the input value is not read, and the output takes the values from the buffer to continue updating, as seen in the saved data cells marked in blue, aiding the output in updating two consecutive 1 values and then completing row 2 of the output.

After updating the first two rows, data begins to be updated in the buffer, repeating the process as above. In the algorithm, we need to pay attention to the oddness and evenness of rows and columns for updating. For instance, for every odd column in the output $(1, 3, 5, ...)$, we update the new values from the buffer, while for odd rows, the buffer updates new values from the input.

This augmentation of spatial dimensions is achieved by duplicating each value in the input feature map to occupy the enlarged space. With a stride of 2, the upsampling process skips every other pixel in each dimension, enlarging the feature map's size.

IV. Experimental setup and result

A. Environment setup

Using the Keras framework, we designed the network with two main components: Encode and Decode. The the model was trained on the MNIST handwritten digit dataset, where the input consists of grayscale images with dimensions of 28×28 pixels.

The Encode portion comprises three Convolutional layers with a 3×3 kernel size each and three Maxpooling layers with a 2×2 size. The encoded image is obtained through the third Maxpooling layer, which has a size of $4 \times 4 \times 8$. Subsequently, this encoded image undergoes four Convolutional layers and three Upsampling layers in the Decode section to reconstruct the compressed image.

The Adam optimizer is used in which loss function is binary cross entropy. Evaluation metrics included Structural Similarity Index (SSIM) and Peak Signal-to-Noise Ratio (PSNR). The training process involved 500 epochs with a batch size of 120. Throughput, or frames per second (FPS), is a measurement used for batch inputs with a designed autoencoder IP block, excluding the time delay through additional blocks such as camera and display, which is calculated by dividing the total number of inputs processed by the processing time, which is established from the Cosimulation process' waveforms. This provides an understanding of number of frames can be processed within a second, facilitating performance evaluation and system optimization. This is the formula we apply to the architectural throughput calculation:

$$FPS = \frac{n}{T}(frames/second)$$

where n is the number of inputs, T and FPS are respectively the processing time and frame per second of the system.

B. High level implementation

The proposed CNN inference accelerator was implemented on the Alveo U250 Data Center Accelerator Card. Resource usage included 69407 flip-flops, 104467 LUTs, 96 IOs, 1 BUFG and 3716 DSPs. The optimal PSNR value achieved was 62.08dB, with a corresponding Frame Per Second (FPS) of 508. The synthesis results are presented in Table I.

TABLE I
RESOURCE UTILIZATION RESULT FOR POST-IMPLEMENTATION IN FPGA

Resource	***Available***	***Used***	***Percentage(%)***
FF	1728000	69407	6.05
LUT	3456000	104467	2.01
DSP	12288	3716	93.23
IO	676	96	14.20
BUFG	344	1	0.07

C. Comparasion

The experimental findings illustrate that the suggested accelerator achieves a power consumption of 5.572W on-chip, operating at 100 MHz. Contrasted with commercially available devices and the latest cutting-edge implementations, the proposed accelerator offers significant advantages in resource utilization and design adaptability. The trade-off is the lower precision than 32-bit floating point used in [12] and [14].

In the related studies mentioned in this paper, our design has advantages and disadvantages compared to the design proposed here. As shown in Table II, the design achieves a frequency of 100 MHz and utilizes fewer resources than other research. Fixed-point arithmetic helps expedite computation and is less complex than floating-point arithmetic. This explains why the design requires fewer resources compared to other methods. Both our design and [13] utilize fixed-point representation for data.

TABLE II
COMPARISION WITH RELATED WORK

		[12]	**[13]**	**[14]**	**Proposed Design**
Full - Pipeline		Yes	Yes	Yes	Yes
Hardware Implement		All	All	Conv Only	All
High Level Synthesis		No	No	Yes	Yes
Frequency (MHz)		166.07	100	100	100
FPGA Capacity	**LUT**	232643	115036	186251	104467
	FF	15327	174412	205704	69407
	DSP	0	1436	2240	3716
Precision		Float (32b)	Fixed (8-16b)	Float (32b)	Fixed (24-32b)
CNN Model		2 Layers	AlexNet	Image Net	3 Layers
Power (W)		1.949	24.8	18.61	5.572

V. Conclusion

The Autoencoder project for image compression and decompression has yielded positive results, showcasing the power of CNN technology in optimizing the representation and restoration of visual information. The Autoencoder model not only effectively compresses image data but also retains crucial features, reducing storage space and enhancing the efficiency of image transmission and processing. In advancing the project, we focus on optimizing the model, expanding its applications into various domains, integrating with other models such as GANs, incorporating it into real-time applications, handling large datasets, and enhancing security measures. These progressive steps aim to transform the Autoencoder model into a versatile and powerful tool applicable across various real-time scenarios.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[3] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua, "A convolutional neural network cascade for face detection," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 5325–5334.

[4] A. Graves, A. rahman Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," 2013.

[5] R. Sarikaya, G. E. Hinton, and A. Deoras, "Application of deep belief networks for natural language understanding," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 4, pp. 778–784, 2014.

[6] S. Li, W. Dai, Z. Zheng, C. Li, J. Zou, and H. Xiong, "Reversible autoencoder: A cnn-based nonlinear lifting scheme for image reconstruction," *IEEE Transactions on Signal Processing*, vol. 69, pp. 3117–3131, 2021.

[7] E. Wang and D. Qiu, "Acceleration and implementation of convolutional neural network based on fpga," in *2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT)*, 2019, pp. 321–325.

[8] D. Rongshi and T. Yongming, "Accelerator implementation of lenet-5 convolution neural network based on fpga with hls," in *2019 3rd International Conference on Circuits, System and Simulation (ICCSS)*, 2019, pp. 64–67.

[9] S. Qiao and J. Ma, "Fpga implementation of face recognition system based on convolution neural network," in *2018 Chinese Automation Congress (CAC)*, 2018, pp. 2430–2434.

[10] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," 2021.

[11] D. Baptista, F. Morgado-Dias, and L. Sousa, "A platform based on hls to implement a generic cnn on an fpga," in *2019 International Conference in Engineering Applications (ICEA)*, 2019, pp. 1–7.

[12] H. V. Phu, T. Minh Tan, P. Van Men, N. Van Hieu, and T. Van Cuong, "Design and implementation of configurable convolutional neural network on fpga," in *2019 6th NAFOSTED Conference on Information and Computer Science (NICS)*, 2019, pp. 298–302.

[13] Z. Liu, Y. Dou, J. Jiang, and J. Xu, "Automatic code generation of convolutional neural networks in fpga implementation," *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 61–68, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:1466627

[14] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060