Mô hình AI trên FPGA: Mô hình CNN gọn nhẹ thông lượng cao và công suất thấp cho bài toán nhận dạng chữ số

TÓM TẮT

Nghiên cứu này trình bày việc thiết kế và triển khai một mạng nơ-ron tích chập trên nền tảng SoC–FPGA để phân loại chữ số viết tay sử dụng bộ dữ liệu MNIST. Mục tiêu là xây dựng một bộ gia tốc CNN gọn nhẹ và hiệu quả, có dưới 1,000 tham số, hoạt động tương thích với bộ xử lý ARM trên bo mạch PYNQ-Z2 thông qua các giao tiếp DMA và AXI. Bộ gia tốc được hiện thực ở mức RTL, với các giai đoạn mô phỏng, tổng hợp và tối ưu hóa tài nguyên, đồng thời vẫn duy trì được độ chính xác của quá trình suy luận. Trên 10,000 ảnh kiểm thử MNIST, hệ thống đạt độ chính xác 91.28%—thấp hơn khoảng 5% so với mô hình chạy trên CPU hai nhân ARM Cortex-A9 (96.26%)—nhưng lại mang lại tốc độ xử lý nhanh hơn 7 lần và giảm 36% mức tiêu thụ điện năng. Thiết kế cho thấy hiệu quả của việc song song hóa và pipeline hóa các phép tích chập trực tiếp trên FPGA, giúp giảm đáng kể mức sử dụng tài nguyên và công suất tiêu thụ. Những kết quả này cung cấp một nền tảng thực tiễn cho các ứng dụng AI nhúng thời gian thực—chẳng hạn như nhận dạng ký tự, giám sát hình ảnh, hệ thống IoT thông minh và tính toán biên—trên các nền tảng SoC–FPGA.

Từ khóa: *Bộ tăng tốc Mạng Nơ-ron Tích chập (Convolutional Neural Networks Accelerator), FPGA, Hệ thống trên Chip (System on Chip), MNIST, Phân loại ảnh.*

Practical Embedded AI on FPGA: A Compact CNN Achieving High Throughput and Low Power for Digit Recognition

ASTRACT

This work presents the design and deployment of a Convolutional Neural Network on an SoC–FPGA platform for handwritten digit classification using the MNIST dataset. The goal is a compact, efficient FPGA-based CNN accelerator with fewer than 1,000 parameters that integrates seamlessly with the ARM processor on the PYNQ-Z2 board via DMA and AXI interfaces. The accelerator is realized at the register-transfer level and undergoes simulation, synthesis, and resource-focused optimization while preserving inference accuracy. On 10,000 MNIST test images, the system attains 91.28% accuracy—about 5 percentage points below a CPU implementation on dual ARM Cortex-A9 cores (96.26%)—but delivers a 7–8× speedup and a 36% reduction in power consumption. The design highlights effective parallelization and pipelining of convolution operations directly on the FPGA, achieving low resource usage and power draw. These results provide a practical foundation for real-time embedded AI applications—such as character recognition, image monitoring, intelligent IoT systems, and edge computing—on SoC–FPGA platforms.

Keywords: *Convolutional Neural Networks Accelerator, FPGA, System on Chip, MNIST, Image Classification.*

1. INTRODUCTION

In recent years, Convolutional Neural Networks (CNNs) have become the dominant approach for image recognition and classification owing to their efficient spatial feature extraction and high accuracy.[1] However, CNN models typically require substantial computation and memory, which makes deployment challenging on embedded systems with limited hardware resources.[2] Field-Programmable Gate Arrays (FPGAs)—with their flexibility, massive parallelism, and low power consumption—have proven to be effective platforms for accelerating CNNs in embedded applications.[1,3] Implementing CNNs on FPGAs can reduce inference latency relative to CPU- or GPU-based software while efficiently utilizing hardware resources such as DSP slices, Lookup Tables (LUTs), and block RAM (BRAM). To achieve high performance on FPGAs, many studies quantize weights and activations to replace floating-point operations with integer arithmetic, thereby reducing hardware complexity while maintaining accuracy.[2] In addition, techniques such as parallelization, pipelining, and data reuse are commonly applied to increase throughput and optimize memory bandwidth.[3] In SoC–FPGA architectures, the integration of FPGA Programmable Logic (PL) and the embedded ARM Processing System (PS) provides a balance between performance and flexibility.[4] The PL handles computationally intensive kernels, while the PS manages control and data movement via direct memory access (DMA) over the Advanced eXtensible Interface (AXI) interconnect.[5,6] On the PYNQ-Z2 platform, the Python Productivity for Zynq (PYNQ) framework enables direct control and testing of the CNN accelerator from Python, facilitating data transfer between Double Data Rate (DDR) memory and the FPGA and thereby simplifying system development and evaluation.[6,7]

This work presents the design and implementation of a lightweight CNN accelerator on an SoC–FPGA platform for handwritten-digit classification using the Modified National Institute of Standards and Technology (MNIST) dataset. The model is optimized to fewer than 1,000 parameters to balance accuracy, memory footprint, and hardware feasibility on the PYNQ-Z2. The system employs DMA/AXI for data exchange between the CPU and FPGA and integrates the PYNQ framework for control and real-time inference. Experimental results demonstrate that the proposed design delivers high performance, efficient resource utilization, and low power consumption, confirming the feasibility of FPGA-based real-time embedded AI systems.

2. DESIGN METHODOLOGY

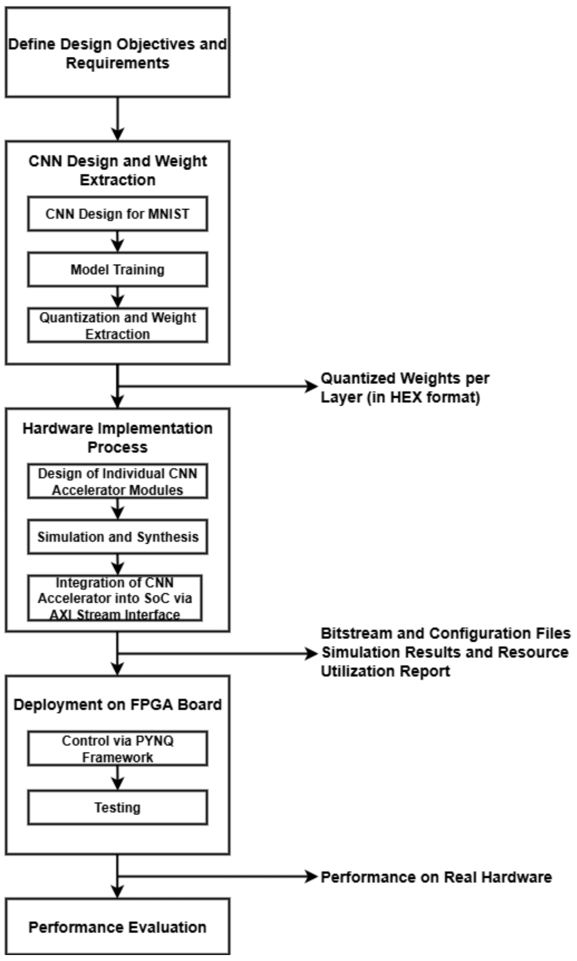
Define Design Objectives and Requirements
CNN Design and Weight Extraction
CNN Design for MNIST
Model Training
Quantization and Weight Extraction
Quantized Weights per Layer (in HEX format)
Hardware Implementation Process
Design of Individual CNN Accelerator Modules
Simulation and Synthesis
Integration of CNN Accelerator into SoC via AXI Stream Interface
Bitstream and Configuration Files
Simulation Results and Resource Utilization Report
Deployment on FPGA Board
Control via PYNQ Framework
Testing
Performance on Real Hardware
Performance Evaluation

Figure 1. Design Implementation Flow

As shown in Figure 1, the design process comprises four stages: first, we specify the MNIST classification task, target performance and accuracy, hardware resource constraints, and the PS–PL communication scheme within the Zynq SoC to guide subsequent decisions. Next, we design, train, and quantize the CNN in PyTorch, convert the quantized weights to 8-bit integers (int8), and export them as HEX files for the hardware stage. We then implement the CNN functional blocks at the Register-Transfer Level (RTL), perform simulation and synthesis to evaluate resource usage (LUTs, DSPs, BRAM), and integrate the accelerator into the Zynq SoC via the AXI4-Stream interface to enable high-throughput PS–PL data movement. Finally, we deploy the generated bitstream on the PYNQ-Z2, control execution through the PYNQ framework on the ARM Cortex-A9, and evaluate the system using 10,000 MNIST test images to measure performance, accuracy, and hardware resource utilization.

2.1 CNNs for Hand-written digit classification

Selecting an appropriate CNN model is pivotal to the overall system design because it directly influences accuracy, processing latency, and hardware resource utilization on the FPGA. On an SoC–FPGA platform constrained by DSP slices, LUTs, and BRAM, the model must balance computational complexity with hardware feasibility: an excessive parameter count can exceed on-chip storage, increase DDR access latency, and impede pipelining, whereas an overly simplified network may weaken feature extraction and reduce accuracy. Accordingly, the research team aims to develop a compact, efficient CNN architecture that enables high-throughput inference in a RTL implementation. As shown in Figure 2, the designed CNN model comprises two convolutional layers, two pooling layers with Rectified Linear Unit (ReLU) activation, and a single fully connected layer. The architecture follows the LeNet-5 paradigm but is simplified for FPGA deployment. A 5×5 kernel is employed to balance feature extraction quality with streamlined, pipelined Multiply–Accumulate (MAC) operations on DSP units. After the two convolution–pooling stages, the output is flattened into a 48×1 feature vector and passed to a fully connected layer that produces class scores over ten outputs corresponding to digits 0–9.

As shown in Figure 2, the proposed CNN contains a small number of parameters across all layers, reflecting its lightweight design for FPGA deployment. The CNN adopts a minimalist architecture with an optimized dataflow tailored to the FPGA's bandwidth and buffering constraints. The pooling layers progressively reduce the spatial dimensions of the feature maps, which facilitates deep pipelining and lowers the computational load of subsequent layers. With a total of 796 parameters, the model attains approximately 96% accuracy in single-precision (float32), providing a robust foundation for quantization and hardware implementation.

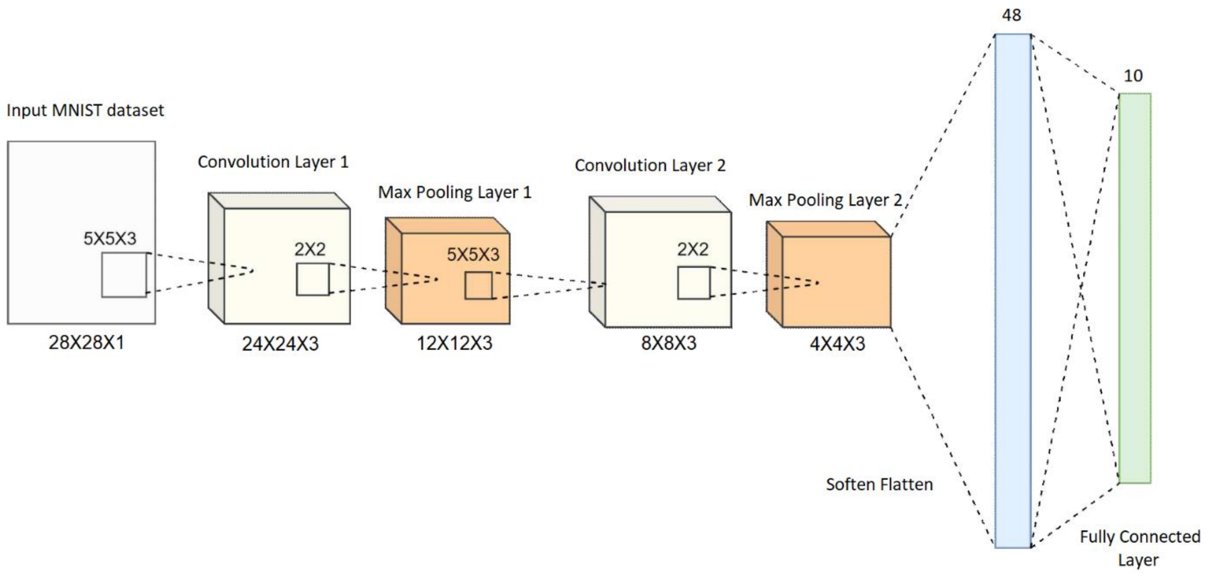

Figure 2. CNN Model Architecture for MNIST Classification

To reduce hardware cost and accelerate computation, we apply quantization-aware training (QAT) to convert the model to int8, reducing memory usage by approximately 4× while preserving accuracy close to the floating-point baseline. Quantized weights and biases are exported per layer to enable direct inference on the FPGA. The quantization pipeline consists of three steps: (1) normalize weights and biases to the range [−1, 1]; (2) scale by 128 to map values to the int8 range [−128, 127]; and (3) encode negative values in two's-complement form for FPGA storage. The resulting quantized parameters are written as .mem files (one per layer) and loaded directly into BRAM or register files within the RTL design. This workflow yields a CNN optimized for both accuracy and hardware deployability and is ready for accelerator construction and on-FPGA inference. As shown in Fig. 3, the quantization process follows a structured three-step pipeline that ensures numerical consistency between software simulation and hardware implementation.

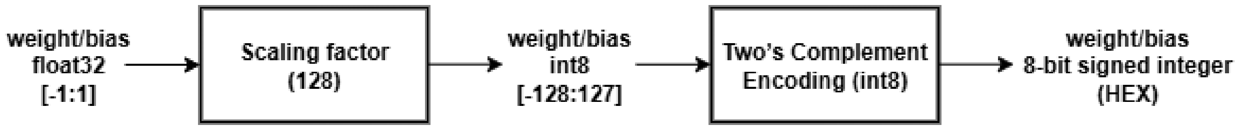
weight/bias float32 [-1:1]
Scaling factor (128)
weight/bias int8 [-128:127]
Two's Complement Encoding (int8)
weight/bias 8-bit signed integer (HEX)

Figure 3. Quantization Process of the CNN Model

2.2 Implementation of CNN Accelerator Core on FPGA

After completing the CNN model, the next step is the design of the RTL module. A CNN architecture can be implemented using various approaches, including Naive Convolution, Matrix Multiplication, or Winograd Convolution. In this work, the basic Naive Convolution method is adopted to construct the CNN hardware architecture.

Figure 4 illustrates the overall system architecture, in which the Buffer, Conv Calc, Maxpooling, ReLU, Fully Connected, and Comparator modules are independently designed and then integrated into a complete CNN block.

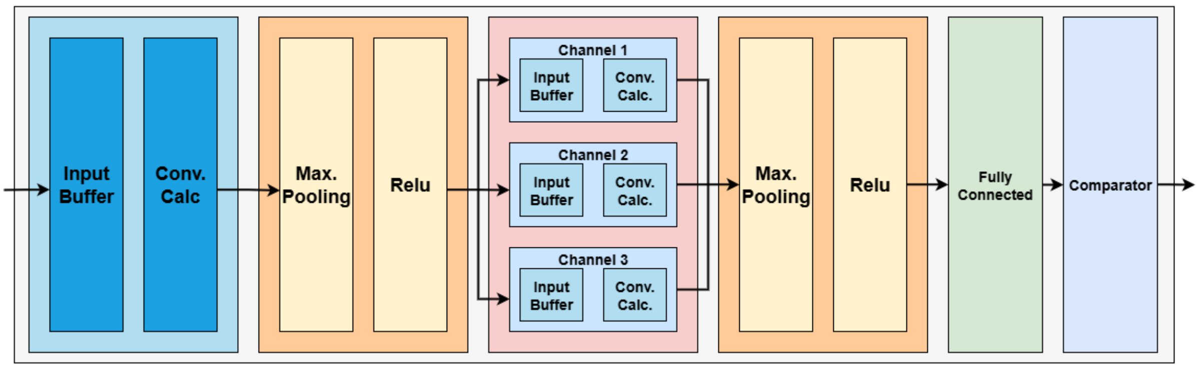

Figure 4. Block Diagram of the CNN Accelerator

The **Input Buffer Block** stores incoming image pixels. The accelerator ingests a 28×28 MNIST bitmap (784 pixels), with each pixel represented in 8 bits. Pixels arrive as a stream—one pixel per clock cycle—in raster-scan order from the top-left corner, proceeding left-to-right and top-to-bottom. This serial loading scheme avoids allocating on-chip memory for the entire image and allows computation to begin immediately, without waiting for full-frame capture. The line buffer holds 140 entries of 8 bits each (corresponding to 5 rows × 28 columns). After a row is processed, the buffer is overwritten with the next row until the entire image is consumed. For convolution, 5×5 pixel windows are extracted from the buffer and shifted by one pixel horizontally at each cycle, repeating until the end of the row. In every clock cycle, one 5×5 window is emitted and forwarded to the convolution stage. With a 5×5, stride-1, valid convolution on a 28×28 input, the first layer produces 24×24 = 576 windows (each containing 25 pixels), matching the output feature-map dimensions of the layer.

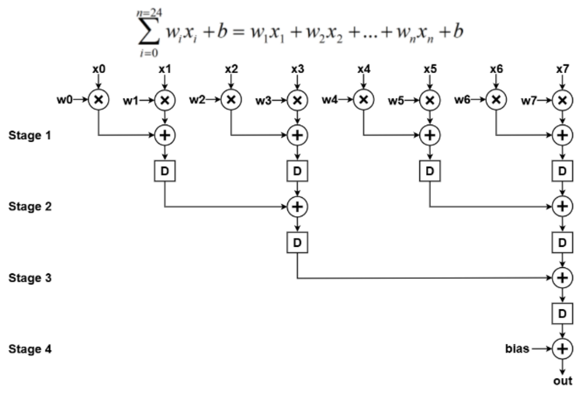
$\sum_{i=0}^{n=24} w_i x_i + b = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$
x0
x1
x2
x3
x4
x5
x6
x7
w0
w1
w2
w3
w4
w5
w6
w7
Stage 1
Stage 2
Stage 3
Stage 4
D
bias
out

Figure 5. Pipeline Stages of the Convolution Module

The **Convolution Calculation Module** performs the convolution between the input data stream from the buffer block and the 5×5 kernel weights. Its input is a stream of 576 windows, each containing 25 parallel pixels, supplied by the buffer. At each clock cycle, one 5×5 window is consumed to compute a dot product with the 5×5 kernel, followed by addition of the bias term. Because arithmetic operations on the FPGA incur propagation delay, the datapath is pipelined to sustain high throughput. Figure 5 illustrates the pipelined structure, realized by inserting registers to partition the computation into multiple stages, thereby shortening the critical path and increasing the achievable clock frequency. With a four-stage pipeline, the first valid output appears four cycles after the corresponding input window is received; thereafter, the module produces one output per cycle.

For **Max-Pooling** and **ReLU Modules**, the first convolution layer yields a 24×24 feature map, emitted as 576 sequential values in a continuous stream, which serves as the input to the MaxPooling and ReLU modules. The 2×2 MaxPooling unit processes pixels in pairs of rows (two from the first row and two from the second row), outputs the maximum among the four, and advances by one pooling stride. The result is then passed to the ReLU activation, which preserves non-negative values and sets negative values to zero. A line buffer stores 12 elements—one output row of the resulting 12×12 feature map from the MaxPooling–ReLU stage. For each 2×2 cell, the running maximum is updated in the buffer when the current value exceeds the stored value; otherwise, the stored value is retained. The buffered value is then compared with zero to apply ReLU. Pointers and control flags step through the buffer in sync with the 576-pixel input stream, producing a 12×12 feature map with 144 outputs.

The **Fully Connected** and **Comparator Module Block** receives input from the second convolution and MaxPooling layers. The 4×4 feature maps with three channels are flattened into a 48×1 vector, multiplied by the corresponding weights, and summed with bias terms to produce ten output neurons. Arithmetic operations in the Fully Connected module are pipelined similarly to the convolution module to optimize performance. After computing the ten neuron outputs, the Comparator identifies the neuron with the highest value (argmax). A ten-element line buffer temporarily stores the neuron values from the Fully Connected module, and the index of the maximum value is emitted as the predicted class (digits 0–9).

2.3 Integration of CNN Accelerator Core into Zynq SoC

After completing the CNN-accelerator hardware core, we integrated it with the ARM processor on the Zynq SoC. Image data (pixels) are stored in off-chip DDR memory and streamed to the accelerator for processing. A Xilinx intellectual property (IP) core provides the interface between the accelerator and the DDR memory system, as illustrated in Fig. 6. To enable standard communication, we implemented a high-level wrapper that exposes an AXI-Stream interface to the accelerator core. Because the accelerator already operates on streaming inputs, the corresponding AXI-Stream signals were defined and connected to the SoC interconnect to ensure correct operation. Fig. 6 illustrates the input–output operation flow of the CNN accelerator integrated within the SoC. First, the CPU in the processing system (PS) configures the AXI DMA via the S_AXI_LITE interface, specifying DDR memory addresses for both the input image and the output-result buffers. The DMA then reads the image data from DDR through the M_AXI_MM2S port and converts it to AXI4-Stream via M_AXIS_MM2S. This stream is temporarily buffered by axis_data_fifo_0 before being delivered to the CNN accelerator core (axis_cnn_mnist_0) for processing. The inference results produced by the accelerator are emitted through axis_data_fifo_1 and transferred back to the DMA via S_AXIS_S2MM. Finally, the DMA writes the output data to DDR through M_AXI_S2MM, and the PS CPU retrieves the results for display or further processing.

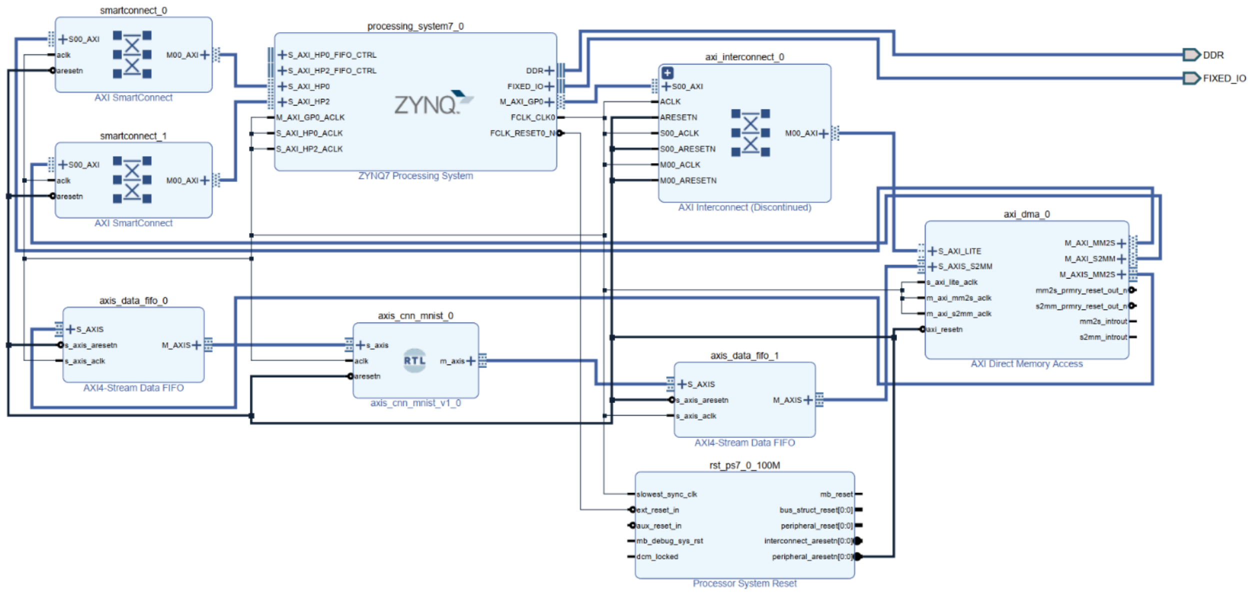

Figure 6. System Block Diagram for implementation in Xilinx Vivado

3. RESULTS

This work implements a CNN hardware accelerator on the PYNQ-Z2 board operating at 100 MHz. As summarized in Table 1, after simulating the accelerator with 10,000 MNIST test images, the classification accuracy reached approximately 91%. The implementation on the PYNQ-Z2 utilized 37.48% of LUTs, 100% of DSPs, and 2.5% of BRAM. Full utilization of DSP slices reflects the dominance of multiply–accumulate (MAC) operations in the convolutional layers, whereas the moderate LUT usage and minimal BRAM consumption indicate an efficient architecture with well-optimized data reuse and pipelining. To evaluate performance, we executed the classification function on both the PL-based accelerator and the dual-core ARM Cortex-A9 CPU (650 MHz) for comparison. Power consumption was measured as total board power during continuous inference on 10,000 MNIST test images over a 15-minute interval to ensure stable operating conditions. The baseline (idle) power was recorded with the board powered on and no inference running; the average processing power was then computed by subtracting this idle power from the total measured power.

Table 1. CNN Accelerator Hardware Synthesis Results

	LUTs	DSPs	BRAM
CNN core	17052 (32.05%)	220 (100%)	0
PL	19942 (37.48%)	220 (100%)	3.5 (2.5%)

Fig. 7a shows the instantaneous power profile of the PYNQ-Z2 board during inference on a single input image, highlighting the transient transition between idle and active states. Besides, Fig. 7 further compares the average power consumption under three conditions: (b) idle (baseline), (c) inference on the ARM CPU, and (d) inference on the PL/SoC accelerator.

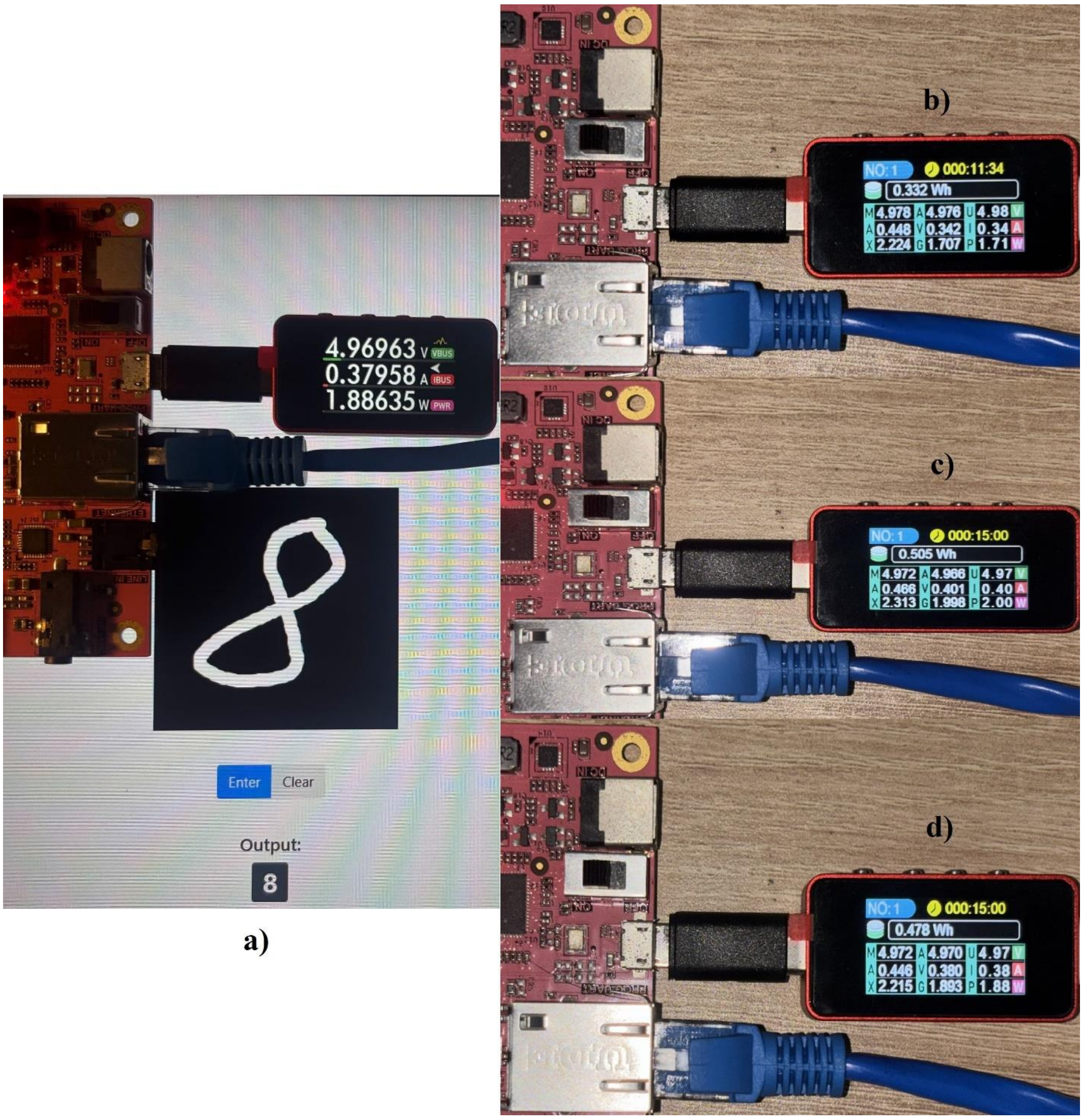

Figure 7. Measurement of instantaneous and average power consumption during CNN inference on the PYNQ-Z2 platform. (a) Instantaneous power profile for a single input image, illustrating the transient from idle to active operation. (b), (c), (d) Average power consumption comparison for idle, CPU-based inference, and PL/SoC accelerator inference.

Table 2 presents the summarized results—including classification accuracy, frame rate, and average power. The proposed CNN accelerator achieves a classification accuracy of 91.28%, which is approximately 5% lower than the CPU implementation (96.26%). However, processing latency is significantly reduced—from 4.25 ms on the CPU to 0.54 ms on the FPGA—representing a 7–8× speedup, despite the FPGA operating at a much lower frequency (100 MHz vs. 650 MHz). This improvement highlights the benefits of parallel computation and deep pipelining inherent in FPGA-based architectures. In terms of power efficiency, the FPGA implementation consumes 186 mW on average, compared with 291 mW for the CPU, resulting in an overall 36% reduction in power consumption. The corresponding energy per inference decreases from 1.234 mJ per image on the CPU to 0.102 mJ per image on the FPGA, demonstrating a substantial improvement in energy efficiency. Overall, these results indicate that the proposed CNN accelerator offers a favorable balance between performance and energy consumption, making it well suited for real-time embedded AI applications on resource-constrained edge devices.

Table 2. Experimental results and performance comparison

Platform	Latency	Accuracy	FPS	Power	Efficiency
FPGA (100 MHz)	0.54 ms	91.28%	1852	186 mW	0.102 mJ/pic
CPU (ARM) Cortex A9 (650 MHz)	4.25 ms	96.26%	235	291 mW	1.234 mJ/pic

4. CONCLUSION

This study completes an end-to-end workflow—from CNN construction and weight quantization to RTL design, SoC–FPGA integration, and deployment on the PYNQ-Z2 board. Experiments on 10,000 MNIST test images show that the CNN implemented in the FPGA's programmable logic (PL) achieves 91.28% accuracy—approximately 5 percentage points lower than the dual ARM Cortex-A9 CPU base-line—while delivering a 7–8× speedup and a substantial reduction in power consumption. These results demonstrate the effectiveness of FPGA-based acceleration for CNNs in both performance and energy efficiency, making the approach well suited to real-time embedded AI applications such as character recognition, image monitoring, and intelligent IoT systems. The implementation process provided comprehensive experience in CNN training and quantization, RTL development, pipeline optimization, resource management, and system integration using the PYNQ framework. Despite limitations imposed by the PYNQ-Z2's constrained resources (few DSP slices and limited BRAM) and the modest accuracy drop due to int8 quantization, this work lays the groundwork for further optimizations—such as layer folding, deeper pipelining, additional parallel processing elements, and scaling to more capable FPGAs to support larger models. Overall, the results are academically meaningful and indicate strong potential for practical deployment in recognition, autonomous robotics, and edge data processing, underscoring FPGAs as promising platforms for intelligent, energy-efficient, and scalable AI systems.

Acknowledgments

REFERENCES

1. A. Shawahna, S. Sait, A. El-Maleh. "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, **2019**, *7*, 7823-7859.

2. V. K. Pham, N. Q. Tran, N. L. Nguyen. "Optimizing the convolutional neural networks for resource-constrained hardwares," *Science & Technology Development Journal – Engineering and Technology*, **2022**, *4*(4), 906.

3. S. Bouguezzi, H. B. Fredj, T. Belabed, C. Valderrama, H. Faiedh, C. Souani. "An efficient FPGA-based convolutional neural network for classification: Ad-MobileNet," *Electronics*, **2021**, *10*(18), 2272.

4. M. Faizan, I. Intzes, I. Cretu, H. Meng. Implementation of deep learning models on an SoC-FPGA device for real-time music genre classification, *Technologies*, **2023**, *11*(4), 91.

5. Anjali, J. P. Anita. AXI based DMA Memory System Testbench Architecture Using UVM Harness Technique, *The 9th International Conference on Advances in Computing and Communication*, ICACC 2019, Kochi, India, 2019.

6. A. Sharma. Evaluation of AXI-Interfaces for Hardware Software Communication,Master's thesis, Technische Universität Chemnitz, Chemnitz, 2019.

7. S. Sun, J. Zou, Z. Zou, S. Wang (eds.). Experience of PYNQ: Tutorials for PYNQ-Z2, *Springer Nature Singapore*, Singapore, 2023.