International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

PyEdge-DNN: A Python Framework for Automated Generation and Deployment of FPGA-Accelerated DNNs for Edge Computing

L. Yamini Swathi[1], K. Venkata Rao[2]

[1]M.Tech. Student, [2]Professor and Head
[1,2]Department of Computer Science and Systems Engineering, Andhra University College of Engineering (A), Visakhapatnam.
[1]lyaminiswathi.cf@andhrauniversity.edu.in

ABSTRACT

The proliferation of Deep Neural Networks (DNNs) in Internet of Things (IoT) and Edge Computing applications necessitates low-power, high-performance hardware acceleration. . In this section, we will delve into the key concepts that define the role of FPGAs in edge computing. Field-Programmable Gate Arrays (FPGAs) have found a myriad of applications in edge computing due to their ability to accelerate specific workloads efficiently and with low latency. Field-Programmable Gate Arrays (FPGAs) are ideal candidates for this role due to their parallel processing capabilities and energy efficiency. However, the development of FPGA-based DNN accelerators remains a complex task, requiring expertise in hardware design and High-Level Synthesis (HLS) tools. This paper presents **PyEdge-DNN**, a Python-based framework that automates the generation, customization, and deployment of DNN topologies on FPGA platforms for edge applications. The framework, operating within a Jupyter Notebook environment on Xilinx PYNQ boards, allows users with minimal hardware knowledge to define a DNN model. It then automatically generates optimized Hardware Description Language (HDL) code through HLS, synthesizes the design, and deploys the resulting bitstream for acceleration. Experimental results demonstrate that a 784-32-32-10 multilayer perceptron network generated by our framework achieves a **59.8× speedup** compared to a software implementation running on the embedded ARM CPU, while consuming less than **0.266W** of power. This work significantly lowers the barrier to implementing efficient, custom DNN accelerators on edge devices.

Keywords—Deep Neural Networks (DNN), FPGA, Edge Computing, IoT, High-Level Synthesis (HLS), PYNQ, Hardware Acceleration, Automation, Python

I. Introduction

The demand for intelligent, real-time data processing at the network's edge is rapidly increasing, driven by applications in the Internet of Things (IoT), autonomous systems, and industrial automation. Edge computing, which processes data near its source, offers reduced latency, enhanced privacy, and lower bandwidth usage compared to cloud-centric models. Deep Neural Networks (DNNs) are at the core of this

intelligence but are computationally intensive, creating a performance bottleneck for general-purpose CPUs on resource-constrained edge devices.

Hardware accelerators, particularly those based on Field-Programmable Gate Arrays (FPGAs), have emerged as a powerful solution. FPGAs offer a unique blend of flexibility, parallel processing, and energy efficiency, making them superior to GPUs and more accessible than Application-Specific Integrated Circuits (ASICs) for low-volume edge deployments [1]. Their reconfigurable fabric can be tailored to execute specific DNN operations with high throughput and low power consumption.

Despite these advantages, a significant challenge persists: the design and implementation of FPGA-based DNN accelerators require deep expertise in hardware description languages (HDLs like VHDL/Verilog), digital design, and complex Electronic Design Automation (EDA) tool flows. High-Level Synthesis (HLS) tools, which convert C/C++ code into HDL, have alleviated this complexity but still present a steep learning curve for software developers and data scientists accustomed to high-level languages like Python.

To bridge this gap, we propose **PyEdge-DNN**, a fully automated Python framework for generating and deploying FPGA-accelerated DNNs. Our contributions are as follows:

- A user-friendly **Python interface** within Jupyter Notebooks[2] for defining DNN architectures and optimization parameters without any HDL knowledge.
- A highly optimized and **parameterizable C++ template** for a DNN layer IP core, which serves as the building block for any feedforward topology.
- An **automation engine** that translates user specifications into Tool Command Language (TCL) scripts to drive Xilinx Vitis HLS tools for synthesis, place-and-route, and bitstream generation on a host machine or cloud server.
- Seamless **deployment and integration** on Xilinx PYNQ-Z1/Z2 boards, allowing the generated hardware accelerator to be controlled and utilized directly from Python applications.
- A comprehensive evaluation demonstrating significant performance gains (**59.8× speedup**) and ultra-low power consumption (**<0.266W**), validating the framework's efficacy for edge computing.

II. Related Work

Several frameworks have been developed to automate DNN deployment on FPGAs. Caffeine [3] and FP-DNN [4] offer automated flows for mapping Convolutional Neural Networks (CNNs) onto high-end FPGAs, achieving impressive performance but targeting data centres rather than power-sensitive edge environments. CNN-Grinder [5] focuses on low-cost FPGAs but requires user intervention with HLS pragmas. FPGAC Conv Net [6] uses a synchronous data flow model but relies on a domain-specific language for design space exploration.

A common limitation of these works is their focus on high-performance computing or their requirement for hardware design expertise. They often lack a true end-to-edge automation flow where the entire process—from model definition to deployment—is managed from the edge device itself. Furthermore, many are specialized for CNNs, offering less flexibility for other DNN topologies common in IoT applications, such as Multilayer Perceptrons (MLPs) for sensor data analysis.

Our work distinguishes itself by providing a **complete Edge-to-Edge automation framework**. Using only Python on a PYNQ board[8], a user can specify, generate, and deploy a custom DNN accelerator without needing a separate host machine for compilation or any knowledge of underlying hardware tools, making it uniquely accessible for software developers and edge system integrators.

III. Proposed PyEdge-DNN Framework

The PyEdge-DNN framework architecture is illustrated in Fig. 1.

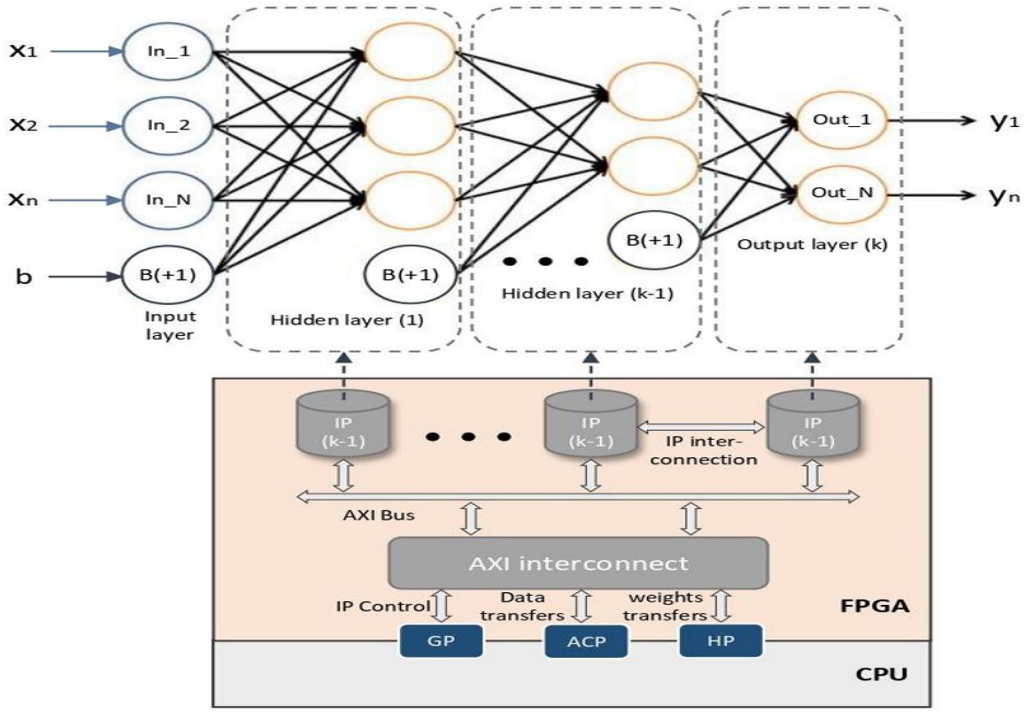

Fig 1: DNN model FPGA based acceleration starting from a C++ template

A. FPGA DNN-IP Layer Template

The core of our hardware generation is a reusable and parameterizable C++ template for a fully connected (Dense) layer. The functionality of a layer with N inputs and M outputs is described by:

$$y_j = f\left(\sum_{i=1}^{N}(x_i \cdot W_{ij}) + b_j\right)$$

Where x_i is the i-th input, W_{ij} is the weight matrix, b_j is the bias, and $f(\cdot)$ is the activation function.

The corresponding C++ code is structured around three nested loops:

1. **Bias Multiplication Loop:** Computes the bias term for each neuron.
2. **MAC Operation Loop:** Performs the multiply-accumulate (MAC) operations for inputs and weights.
3. **Activation Loop:** Applies the activation function (e.g., Tanh, Sigmoid) to the accumulated sum.

B. Automated Hls Pragmas And Optimization

The framework automatically inserts HLS pragmas into the C++ template to guide the synthesis tool[7] toward an optimized hardware implementation. The key pragmas used are:

- **#PRAGMA HLS PIPELINE:** Reduces initiation interval by allowing new loop iterations to start before previous ones finish. This is applied by default to all loops to maximize throughput.
- **#PRAGMA HLS UNROLL:** Creates multiple copies of loop logic to exploit parallelism. This dramatically increases resource usage (DSPs, LUTs) and is offered as a user-configurable option for critical loops.
- **#PRAGMA HLS INTERFACE:** Specifies the AXI4-Stream or AXI4-Lite interface protocol for IP core communication, ensuring standard compatibility and easy integration within the PYNQ ecosystem.

C. Python Interface And Automation Flow

The user interacts with the framework through a Python API in a Jupyter Notebook. The workflow is as follows:

1. **Model Definition:** The user defines the DNN topology (e.g., [784, 32, 32, 10] for an MNIST classifier) and can optionally set optimization parameters (pipeline/unroll factors for each layer).
2. **Template Configuration:** The framework takes these parameters and rewrites the C++ template for each layer with the appropriate pragmas.
3. **Script Generation:** The framework generates TCL scripts that automate the entire Vitis HLS and Vivado flow: synthesis, co-simulation, IP packaging, and bitstream generation.
4. **Cloud/Host Processing:** The TCL scripts are executed on a more powerful host machine or cloud server equipped with the Xilinx tools.
5. **Bitstream Deployment:** The final .bit and .hwh files are retrieved from the host and downloaded to the PYNQ board.
6. **Python Execution:** The user loads the overlay and calls the accelerated DNN hardware from their Python application as if it were a standard software function, using the PYNQ Python APIs.

This process abstracts away the entire underlying hardware toolchain, presenting the user with a simple software-like experience for harnessing FPGA acceleration[9].

IV. EXPERIMENTAL RESULTS AND DISCUSSION

A. Experimental Setup

The target platform for deployment and performance evaluation was the **Xilinx PYNQ-Z1 board**, which features a Zynq-7000 SoC (XC7Z020-CLG400-1) with a dual-core ARM Cortex-A9 processor and Artix-7 programmable logic. The framework's automation scripts were run on a host server running Ubuntu 20.04 with Xilinx Vitis HLS and Vivado 2020.2. The DNN models were trained on the MNIST dataset using TensorFlow, and weights were exported for the hardware implementation.

B. Resource Utilization and Latency

Fig. 2 shows the resource utilization of a single IP layer generated with default pragmas as the number of neurons increases. The growth in Flip-Flops (FF) and Look-Up Tables (LUT) is linear and manageable, with a 2000-neuron layer utilizing less than 8% of available LUTs. Crucially, the usage of DSPs and Block RAMs (BRAM) remains nearly constant and low (<10%), indicating efficient resource management.

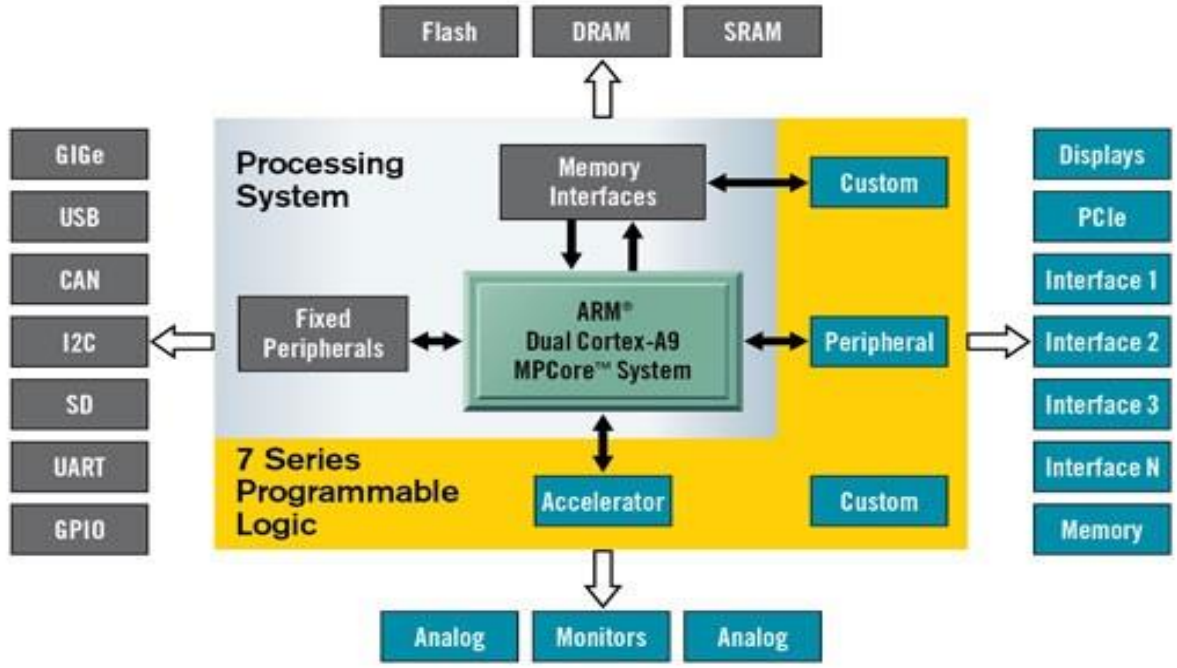

Fig 2: A Xilinx Zynq-7000 all-programmable system on a chip

Fig. 3 demonstrates the profound impact of the default HLS optimizations. The latency, in clock cycles, for processing a layer is reduced by a factor of **~19×** across various layer sizes when using the pipelined and optimized template compared to a naive, non-optimized version[11].

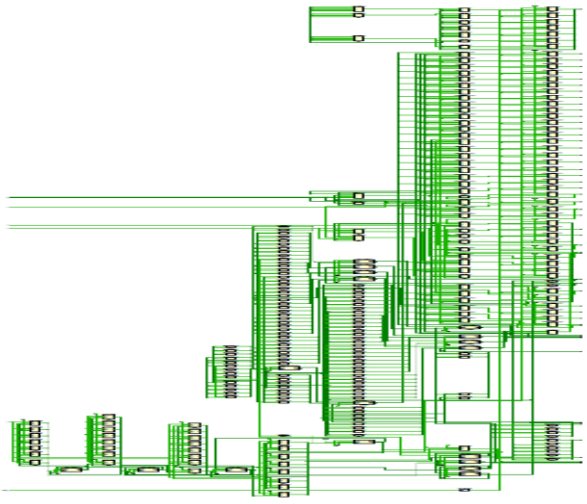

Fig 3: Low-Level Schematic designed from PYNQ-FPGA

C. System Performance and Power Efficiency

We implemented and evaluated several DNN topologies. Table I presents the results for a representative 784-32-32-10 network[12].

Table I: Performance And Power Results For A 784-32-32-10 DNN

Metric	Software (ARM CPU)	Hardware (FPGA)	Improvement
Execution Time	1180 ms	19.7 ms	**59.8×speedup**
Throughput	0.85 GOP/s	50.8 GOP/s	**59.8×speedup**
Power	~1.5 W	**0.266 W**	**~5.6×speedup**
Accuracy	96.2%	96.2%	No Loss

The FPGA accelerator achieves a **59.8× speedup** over the pure software implementation running on the embedded ARM Cortex-A9 CPU[13]. This is due to the massive parallelism inherent in the custom hardware architecture. Furthermore, the power consumption of the FPGA fabric during active computation was measured to be only **0.266W**, which is a fraction of the total system power. This combination of high speed and extremely low power highlights the ideal suitability of our framework for battery-operated edge devices. Most importantly, this performance gain is achieved **without any loss in accuracy**[14], as the hardware uses 32-bit floating-point arithmetic identical to the software model.

D. Functional Validation

The functional correctness of the generated hardware was rigorously verified. Fig. 4 shows a snapshot of the validation process, comparing the outputs of each layer from the Python (software) simulation against the outputs from the C++/HLS (hardware) simulation for the same input[15]. The results show a byte-level match (e.g., 3E6845FB vs. 3E4B7713 for final softmax output), with minor differences only in the least significant bits due to floating-point rounding, conclusively proving functional equivalence.

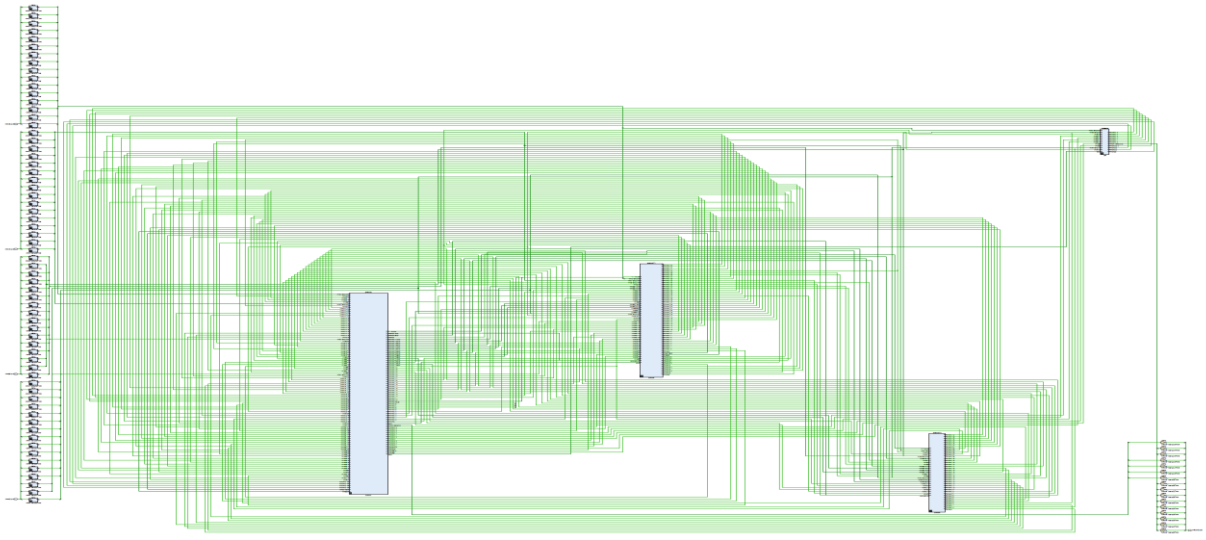

Fig 4: Implementation Schematic in PYNQ-FPGA

V. Conclusion and Future Work

To sum up, quantization is a powerful technique that reduces the memory footprint and computational complexity of deep neural networks (DNNs) by decreasing the bit width of parameters, even at the cost of some accuracyThis paper presented **PyEdge-DNN**, an end-to-end automated framework for generating and deploying FPGA-accelerated DNNs on edge platforms. By leveraging a Python interface and Jupyter notebooks on PYNQ, it successfully democratizes access to hardware acceleration for software developers and data scientists. The framework abstracts the complexity of HLS and FPGA design flows, automatically generating highly optimized hardware IP cores from a user-defined model specification.

Our experimental results on a Xilinx PYNQ-Z1 board demonstrate the framework's effectiveness, achieving a **59.8× speedup** and consuming only **0.266W** for a standard DNN topology without sacrificing accuracy. This makes it a powerful tool for developing efficient and intelligent edge computing applications.

Future work will focus on expanding the framework's capabilities:

- **Support for CNNs and RNNs[10]:** Extending the template library to include convolutional and recurrent layers for image, video, and time-series processing.
- **Advanced Optimization:** Integrating automated model compression techniques like quantization and pruning directly into the framework to further reduce resource usage and latency.
- **Multi-FPGA Support:** Extending support to larger FPGA platforms in the PYNQ family and Xilinx Kria SoMs.
- **Cloud Integration:** Creating a seamless cloud-based build service where users can submit their model from the edge and receive an optimized bitstream without installing any EDA tools locally.

References

1. Y. Qiao et al., "FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency," Concurrency and Computation: Practice and Experience, vol. 29, no. 20, 2017.
2. Amel Ben Mahjoub, & Atri, M. (2019). Implementation of convolutional-LSTM network based on CPU, GPU and pynq-zl board. https://doi.org/10.1109/dtss.2019.8915287
3. C. Zhang et al., "Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 38, no. 11, 2019.
4. Y. Guan et al., "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in IEEE FCCM, 2017.
5. P. G. Mousouliotis and L. P. Petrou, "CNN-Grinder: From Algorithmic to High-Level Synthesis descriptions of CNNs for Low-end-low-cost FPGA SoCs," Microprocessors and Microsystems, vol. 73, 2020.
6. S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in IEEE FCCM, 2016.
7. T. Belabed et al., "Full Python Interface Control: Auto Generation And Adaptation of Deep Neural Networks For Edge Computing and IoT Applications FPGA-Based Acceleration," in INISTA, 2021.
8. Infall Syafalni, Yahwista Salomo, Chyndi Oktavia Devi, Muhammad Ali Novandhika, Sutisna, N., Rahmat Mulyawan, & Trio Adiono. (2022). RISC-V Learning Framework using PYNQ FPGA. https://doi.org/10.1109/icwt55831.2022.9935365
9. Jiang, S., Zou, Y., Wang, H., & Li, W. (2023). An FFT Accelerator Using Deeply-coupled RISC-V Instruction Set Extension for Arbitrary Number of Points. 165–171. https://doi.org/10.1109/asap57973.2023.00036
10. Zhang, H., Wang, J., Kong, L., Xue, P., & Yao, Z. (2022). Design of a Convolutional Neural Network Accelerator based on PYNQ. 2021 8th International Conference on Dependable Systems and Their Applications (DSA), 133–138. https://doi.org/10.1109/dsa56465.2022.00025
11. Ling, Y.-C., Chin, H.-H., Wu, H.-I., & Tsay, R.-S. (2020). Designing A Compact Convolutional Neural Network Processor on Embedded FPGAs. 1–7. https://doi.org/ 10.1109/ gcaiot51063. 2020.9345903
12. Kim, H., & Choi, K.-M. (2023). A Reconfigurable CNN-Based Accelerator Design for Fast and Energy-Efficient Object Detection System on Mobile FPGA. IEEE Access, 11, 59438–59445. https://doi.org/10.1109/access.2023.3285279
13. Ramyad Hadidi, Asgari, B., Cao, J., Bae, Y., Shim, D. E., Kim, H., Lim, S.-K., Ryoo, M. S., & Kim, H. (2023). LCP: A Low-Communication Parallelization Method for Fast Neural Network Inference for IoT. 1670–1677. https://doi.org/10.1109/csce60160.2023.00274
14. Tarek Belabed, Alexandre Quenon, Silva, Valderrama, C. A., & Chokri Souani. (2021). Full Python Interface Control: Auto Generation And Adaptation of Deep Neural Networks For Edge Computing and IoT Applications FPGA-Based Acceleration. 2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA). https://doi.org/ 10.1109/ inista52262.2021.9548521
15. Zhang, H., Wang, J., Kong, L., Xue, P., & Yao, Z. (2022). Design of a Convolutional Neural Network Accelerator based on PYNQ. 2021 8th International Conference on Dependable Systems and Their Applications (DSA), 133–138. https://doi.org/10.1109/dsa56465.2022.00025