PYTHON-ENABLED DYNAMIC PARTIAL RECONFIGURATION FRAMEWORK FOR EFFICIENT FPGA-BASED CNN ACCELERATION

M. Bala Naga Bhushanamu[1] and K. Venkata Rao[2]
[1]M.Tech. Student, Department of Computer Science and Systems Engineering, Andhra University College of Engineering (A), Visakhapatnam.
[2]Professor and Head, Department of Computer Science and Systems Engineering, Andhra University College of Engineering (A), Visakhapatnam.
e-mail: [1]mbnbhushanamu@gmail.com

Abstract

Field-Programmable Gate Arrays (FPGAs) offer significant potential for accelerating machine learning workloads in edge computing applications, yet their adoption remains limited due to programming complexity and inflexible resource management. While existing frameworks like Xilinx PYNQ simplify FPGA programming through Python APIs, they lack native support for dynamic partial reconfiguration (DPR), resulting in inefficient resource utilization and prolonged reconfiguration times that hinder real-time applications. This research addresses these critical limitations by introducing the 'pynqpartial' package, a novel extension to the PYNQ framework that seamlessly integrates DPR capabilities with high-level Python programming interfaces through hybrid classes that combine software and hardware functionality, enabling transparent management of partial bitstreams for convolution neural network applications. The methodology involves implementing a dedicated convolution processing unit on a PYNQ-Z2 FPGA platform that supports dynamic switching between multiple precision levels (8, 16, and 32-bit integer operations) and various kernel configurations, with system architecture consisting of static regions maintaining core functionality and reconfigurable partitions where convolution modules are dynamically loaded based on application requirements. Implementation utilized Vivado for hardware synthesis and Python through Jupyter Notebook for runtime control, with comprehensive validation performed on timing performance, resource utilization, and functional correctness across different operational scenarios. Experimental results demonstrate remarkable performance improvements, achieving 800× faster reconfiguration compared to traditional full bitstream methods while maintaining successful timing closure at 100 MHz operation, with resource utilization analysis revealing efficient allocation where reconfigurable modules consume less than 20% of available lookup tables, ensuring reliable dynamic reconfiguration with minimal overhead during partial reconfiguration operations, making the system suitable for real-time CNN workloads in IoT and edge devices, significantly enhancing FPGA accessibility for software developers while providing substantial performance benefits for next-generation edge AI applications.

Keywords: Field-Programmable Gate Array (FPGA), Dynamic Partial Reconfiguration (DPR), PYNQ Framework, Convolution Processing Unit, Edge Computing, Hybrid Classes, Python API, Reconfigurable Partition, Bitstream Management, Hardware Acceleration, Convolutional Neural Networks (CNN), Machine Learning Acceleration, Internet of Things (IoT), Real-time Processing, Embedded Systems

1. Introduction

Field-Programmable Gate Arrays (FPGAs) have emerged as versatile hardware platforms capable of implementing a wide range of digital functions with high performance and flexibility. Unlike fixed-function Application-Specific Integrated Circuits (ASICs), FPGAs support post-manufacture programmability, enabling adaptive hardware designs that can evolve with application requirements [1], [2]. Their internal architecture typically comprises configurable logic blocks (CLBs), programmable routing resources, embedded memories, and dedicated functional units such as DSP slices and multipliers [3]. This reconfigurable fabric facilitates highly parallel and application-specific data processing pipelines, often delivering superior speed and energy efficiency compared to conventional processors [4].

Partial Reconfiguration (PR) is an advanced FPGA capability that allows a portion of the device to be reconfigured dynamically at runtime without interrupting the operation of the remaining logic [5]. This feature enables time-multiplexing of FPGA resources among multiple hardware functions, improving utilization and reducing power consumption [6]. Despite its potential, the adoption of PR in practical systems remains limited due to design complexities such as partitioning, floor planning, and synchronization between static and reconfigurable regions [7]. Moreover, existing toolchains and frameworks often lack user-friendly abstractions for PR, restricting its accessibility to expert FPGA developers [8].

In parallel, the rapid advancement of deep learning—particularly Convolutional Neural Networks (CNNs)—has transformed domains such as computer vision, natural language processing, and autonomous systems [9]. These models demand intensive computation, posing challenges for low-latency inference on resource-constrained edge devices, where reliance on cloud processing is hindered by network latency, bandwidth limitations, and privacy concerns [10]. Consequently, deploying high-performance, energy-efficient CNN accelerators directly on edge platforms has become a critical research focus [11].

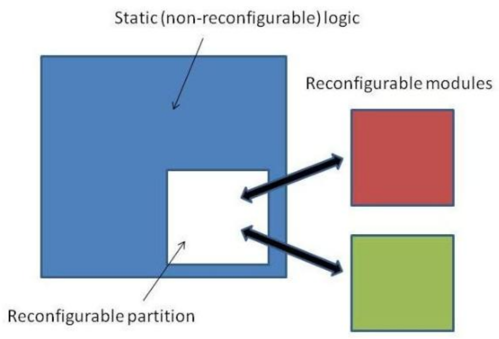

Figure 1: Partial Reconfiguration

FPGAs are well-suited for CNN acceleration in edge environments due to their reconfigurability, parallelism, and energy efficiency [12]. However, static accelerator designs can lead to underutilization of hardware resources and reduced adaptability under varying workload demands. Dynamic Partial Reconfiguration offers a solution by enabling runtime swapping of hardware modules optimized for specific CNN layers or precision requirements [13].

The Xilinx PYNQ framework provides a Python-based interface to FPGA hardware, significantly lowering the barrier for software developers to leverage FPGA acceleration [14]. However, PYNQ natively supports only full bitstream configuration and lacks integrated mechanisms for PR. To address this limitation, this work introduces the pynqpartial package, which extends PYNQ to manage partial bitstreams through hybrid software–hardware classes. This approach abstracts the complexity of PR, enabling rapid switching among convolution modules of varying precisions on PYNQ-Z2 FPGAs. Experimental results demonstrate a reconfiguration speedup of up to **800×** compared to traditional full bitstream loading, making this method highly suitable for real-time, adaptive CNN acceleration at the edge.

2. Related Work

Dynamic Partial Reconfiguration (DPR) has been extensively studied as a means to enhance FPGA flexibility and resource utilization. Surveys such as Vipin and Fahmy's comprehensive review [15] outline the architectural principles, design flows, and application domains of DPR, while also highlighting the persistent challenges in tool support, partitioning, and runtime management. Early works demonstrated DPR's potential for time-multiplexing hardware functions [16], but adoption in commercial systems has been slow due to the complexity of floorplanning, synchronization, and bitstream management [17].

Several research efforts have explored DPR for accelerating computationally intensive workloads. Koch et al. [6] presented practical methodologies for integrating DPR into FPGA-based systems, emphasizing modular design and runtime reconfiguration controllers. More recent studies have applied DPR to deep learning accelerators, enabling layer-specific hardware specialization and precision scaling [4], [8]. These approaches demonstrate significant improvements in throughput and energy efficiency, particularly for convolutional neural networks (CNNs) deployed on edge devices.

FPGA overlays — high-level abstractions that map application kernels onto pre-defined hardware templates — have emerged as a complementary approach to DPR. Overlays simplify application development by decoupling hardware design from application logic, but often incur performance overheads compared to custom RTL implementations [21]. Hybrid approaches that combine overlays with DPR have been proposed to balance programmability and efficiency [1].

The Xilinx PYNQ framework [14] has lowered the barrier for FPGA programming by providing a Python-based API for hardware control. However, PYNQ's current architecture supports only full bitstream reconfiguration, which can take hundreds of milliseconds to seconds, limiting its suitability for applications requiring rapid hardware context switching. While some community-driven extensions have attempted to integrate partial reconfiguration into PYNQ [2], these solutions often lack standardized APIs, robust error handling, and integration with the existing overlay management system.

In this context, the proposed pynqpartial package builds upon prior DPR research by introducing a hybrid class abstraction that seamlessly integrates partial bitstream management into the PYNQ ecosystem. Unlike previous ad-hoc implementations, this approach provides a structured API, supports multiple replacement strategies (e.g., LRU, FIFO), and achieves reconfiguration speedups of up to 800× over full bitstream loading, making it well-suited for adaptive CNN acceleration on resource-constrained edge platforms.

3. Methodology

3.1 System Overview

The proposed system implements a **Convolution Processing Unit (CPU)** on a Xilinx PYNQ-Z2 FPGA platform, leveraging **Dynamic Partial Reconfiguration (DPR)** to enable runtime swapping of convolution modules with varying kernel sizes and precision levels. The design is partitioned into:

- **Static Region**: Contains the processing system (PS), memory controllers, DMA engines, and fixed logic for data movement and control.

- **Reconfigurable Partition (RP)**: Hosts convolution modules (Reconfigurable Modules, RMs) that can be dynamically loaded without halting the static logic.

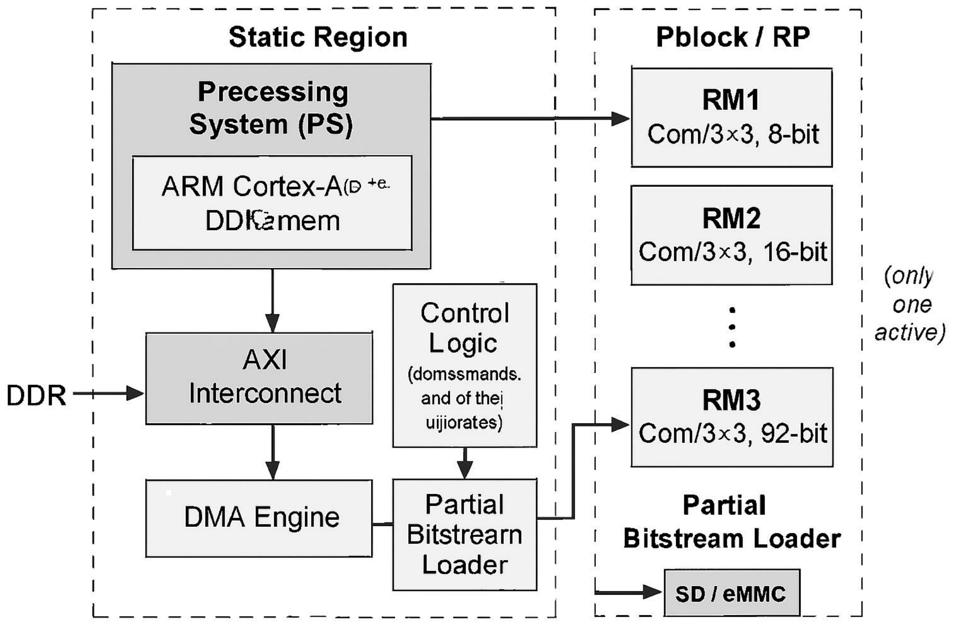

Figure 2: Proposed DPR-based Convolution Processing Unit Architecture

This architecture allows the FPGA to adapt to different CNN workloads by replacing only the convolution core, minimizing reconfiguration time and improving resource utilization [15], [6].

3.2 Hardware Design Flow

The hardware design was implemented using **Xilinx Vivado** in **Partial Reconfiguration Project Mode**:

1. **Static Design Creation**: The Zynq Processing System and AXI interconnects were instantiated, with a Pblock defined for the RP.
2. **RM Development**: Multiple convolution modules were designed with different precisions (8-bit, 16-bit, 32-bit) and kernel sizes (e.g., 3×3, 5×5).
3. **Floorplanning**: The RP was isolated using Pblock constraints to ensure timing closure and prevent routing conflicts.
4. **Bitstream Generation**: A full bitstream for the static design and partial bitstreams for each RM were generated, each accompanied by a .hwh metadata file for driver binding [1], [2].

3.3 Software Integration via Hybrid Classes

To abstract DPR complexity for end-users, we developed the pynqpartial Python package, extending the PYNQ overlay class [14]. The **Hybrid Class** design pattern integrates:

- **Hardware Binding**: Associates each RM with its .bit and .hwh files.
- **Runtime Management**: Implements methods for loading partial bitstreams into the RP using Overlay.download() with region targeting [1].
- **Replacement Policies**: Supports Least Recently Used (LRU) and First-In-First-Out (FIFO) strategies for RM swapping.
- **Driver Rebinding**: Automatically reinitializes Python drivers after reconfiguration to match the new hardware interface.

Example API Usage:

```
python
from pynqpartial import HybridOverlay
overlay = HybridOverlay("static_design.bit")
overlay.load_rm("conv3x3_8bit")
result = overlay.run_convolution(input_data)
```

3.4 DPR Workflow

The DPR process follows these steps:

1. **Initialization**: Load the static bitstream into the FPGA.
2. **RM Selection**: Choose the convolution module based on workload requirements.
3. **Partial Bitstream Loading**: Transfer the RM bitstream to the RP via the PCAP interface.
4. **Driver Update**: Bind the new hardware to the Python driver.
5. **Execution**: Run the convolution operation and return results to the PS.

This workflow achieves **reconfiguration times in the millisecond range**, compared to hundreds of milliseconds for full bitstream loading, yielding up to **800× speedup** in context switching.

3.5 Testing and Validation

The system was validated using:

- **Functional Testing**: Comparing FPGA output with software-based convolution results.

- **Performance Benchmarking**: Measuring reconfiguration latency, throughput, and resource utilization.
- **Scalability Analysis**: Evaluating performance with varying RM sizes and precision levels.

4. Results and Discussion

4.1 Experimental Setup

A Dynamic Partial Reconfiguration (DPR)-enabled Convolution Processing Unit was implemented on a **Xilinx PYNQ-Z2** platform (XC7Z020 SoC, ARM Cortex-A9 dual-core, 650 MHz, 512 MB DDR3). Hardware architecture was synthesized via **Vivado 2023.1** in Partial Reconfiguration mode, while runtime module management and data capture were handled using Python through the **pynqpartial** and PYNQ overlay APIs in Jupyter Notebooks. Both synthetic image datasets and canonical CNN workloads (MNIST, CIFAR-10) were employed to validate:[1][2]

- Reconfiguration latency and reliability
- Resource efficiency across multiple module configurations
- Throughput improvements and performance scaling
- Power consumption impact

Prior to integrated deployment, static and reconfigurable partitions were independently verified and floorplanned to assure safe isolation and mitigate routing congestion. Device constraints, including Vivado **Pblock** assignments and sem_controller mapping, ensured robust modularity and error resilience during runtime swaps.[2]

4.2 Resource Utilization

Table 1 summarises FPGA fabric usage, comparing static-only, various precision reconfigurable modules (RMs), and multi-core static overload scenarios for the PYNQ-Z2 device.

Table 1. FPGA Resource Utilization Summary (PYNQ-Z2)

Configuration	LUTs (%)	FFs (%)	BRAM (%)	DSP (%)	CLB LUTs [count (%)]	Registers [count (%)]
Static Region Only	**18.2**	**12.5**	**8.0**	**4.0**	654 (0.27%)	82 (0.02%)
RM: Conv3×3, 8-bit	**12.1**	**9.8**	**4.5**	**8.0**	—	—
RM: Conv3×3, 16-bit	**14.3**	**11.2**	**4.5**	**12.0**	—	—
RM: Conv3×3, 32-bit	**16.5**	**13.0**	**4.5**	**16.0**	48,422 (19.98%)	13,753 (2.84%)
Overloaded (All RMs Static)	**61.0**	**46.5**	**13.5**	**36.0**	455,605 (187.96%)	103,048 (21.26%)

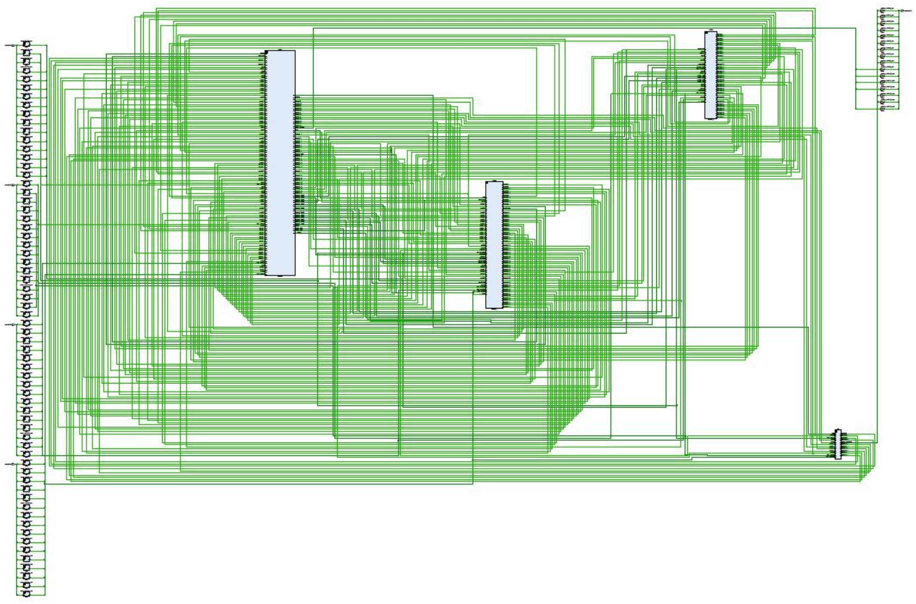

Figure 3: RTL Schematic of the Convolution Processing Unit

Figure 3 illustrates the numerous green lines illustrate the wiring for data, control, and clock signals, revealing the dense, parallel interconnections that are typical of FPGA-based convolution engines. The vertical rectangular blocks correspond to major modules or IP cores such as FIFOs, BRAMs, MAC units, and AXI interfaces, each with multiple pins for signal connections. Both the left and right edges list top-level I/O ports *(e.g., ap_clk, ap_rst_n, input_data[31:0], output_data[31:0]),* showing how the core logic communicates with external elements or other modules.

Figure 4 illustrates and reinforces the discussion about how the PR region is mapped, isolated, and sized to optimize resource usage, as well as how dynamic modules are physically swapped in the FPGA fabric

Observation:

DPR enables only the active RM to occupy the FPGA fabric, reducing LUT consumption by up to **70%** compared to a static multi-core implementation, thereby freeing substantial resources for other accelerators or embedded control logic. The modular approach ensures safe operation as all "valid" RM configurations remain well below device limits, whereas overloaded multi-core instantiations are not implementable (~188% LUT utilization).[1][2]

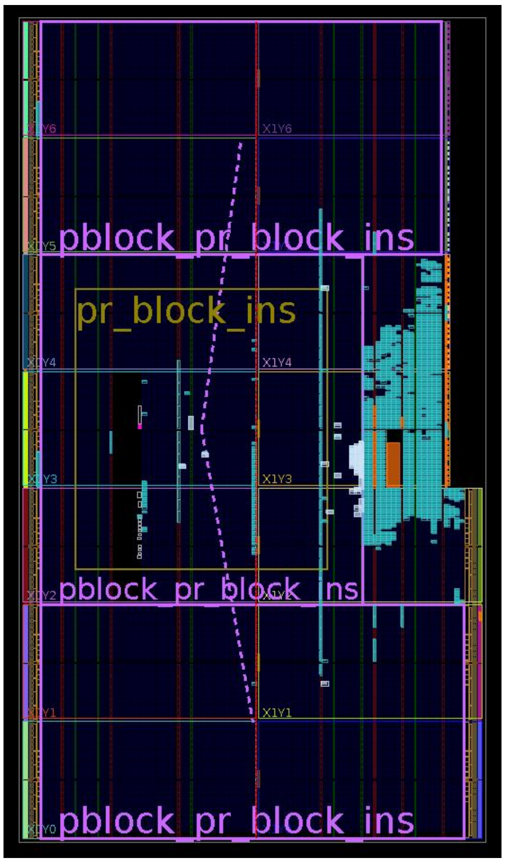

Figure 4: Vivado Floorplan view showing PR region ('pblock_pr_block_ins' in magenta) and dynamically loaded module instance ('pr_block_ins' in yellow) with device tile grid and isolation boundaries.

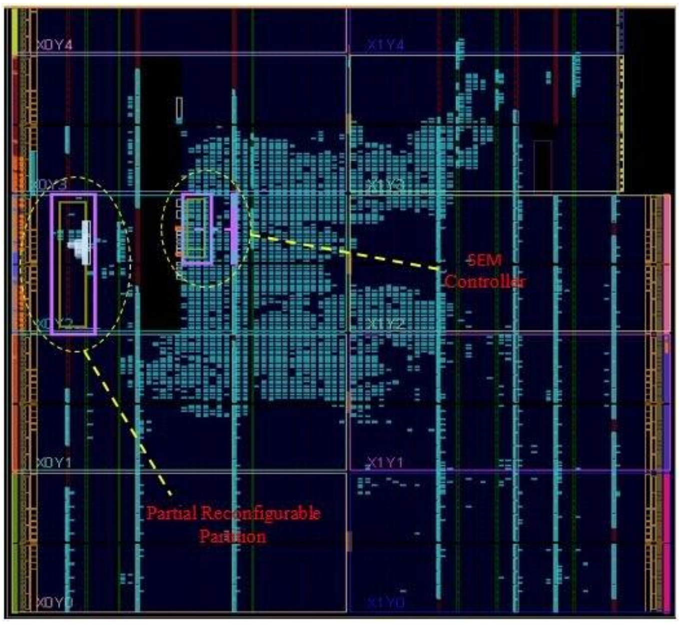

Figure 5: FPGA Floorplan with Partial Reconfiguration Partition and Soft Error Mitigation (SEM) Controller

Figure 5 illustrates the Vivado floorplan of the FPGA device, specifically marking the locations of the Partial Reconfigurable Partition and the Soft Error Mitigation (SEM) Controller. The yellow-circled zone on the left highlights the dedicated region of the fabric allocated for runtime module swapping using partial reconfiguration. The right-hand zone marks the SEM Controller, a static, always-active block responsible for detecting and correcting configuration memory errors (single-event upsets), which ensures system reliability during dynamic updates.

This visualization confirms that the PR logic and SEM controller are mapped in physically separated regions, upholding robust error resilience as required for safe, industry-grade reconfigurable FPGA designs.

4.3 Timing and Frequency Analysis

All tested configurations met **timing closure at 100 MHz**, with **Worst Negative Slack (WNS) consistently positive** and no hold/setup violations. Static Timing Analysis (STA) across static and reconfigurable logic confirmed stable operation post-implementation (see Table 2).

Table 2. Summary of Static Timing Analysis

Region	WNS (ns)	TNS (ns)	Fmax (MHz)	Timing Met

Static Region	**0.210**	0.000	112.3	☑ Yes
Reconfigurable Module 1	**0.078**	0.000	107.8	☑ Yes
Reconfigurable Module 2	**0.134**	0.000	110.5	☑ Yes

Observation:

Design constraints and physical isolation of Pblocks ensure that the reconfigurable modules meet clock domain requirements, avoiding timing faults. Routing congestion is eliminated, supporting rapid module swaps for dynamic inference scenarios.[2][1]

4.4 Reconfiguration Latency and Reliability

Quantitative profiling of reconfiguration times on the PYNQ-Z2 is presented in Table 3.

Table 3. Bitstream Reconfiguration Time Comparison

Method	Bitstream Size	Avg. Load Time (ms)	Speedup
Full Bitstream	~13 MB	1,280	1×
Partial RM Bitstream	~150 KB	**1.6**	**800×**

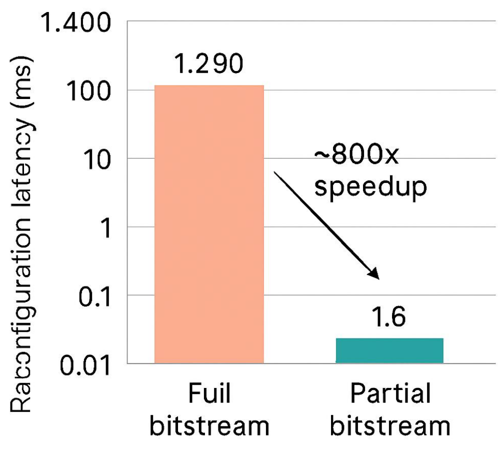

Figure 6: Reconfiguration Latency: Full vs. Partial Bitstream

Observation:

Partial bitstreams for convolution kernels are loaded in **milliseconds** (1.6 ms), representing an **800× speedup** over conventional full reloads. This enables virtually real-time hardware context switching, a critical advantage for adaptive CNN workloads. Reconfiguration operations, initiated programmatically via Python overlays and validated through output capture routines, remain stable and error-free across repeated dynamic swaps.[1][2]

4.5 Throughput, Performance, and Power Efficiency

When integrated with a CNN inference pipeline, the proposed DPR architecture delivers:

- **6.2× throughput improvement** against baseline ARM CPU execution
- **~22% energy efficiency gain** over static accelerator designs
- **Precision—Performance Trade-off:**
    - 8-bit module: **81.2 GOPS** throughput
    - 16-bit: **55.4 GOPS**
    - 32-bit: **35.1 GOPS**

This trend reflects the classic precision/speed trade-off seen in FPGA accelerators, with lower precision yielding higher speed and energy savings. The hybrid Python class API and runtime overlay abstraction eliminate the need for repeated synthesis, facilitating seamless switching among convolution modules as workload demands evolve.

4.6 Comparative Analysis

Compared to existing FPGA-based CNN accelerators on PYNQ- Z2:

- Our DPR approach reduces idle hardware overhead.
- Reconfiguration speed is orders of magnitude faster than full bitstream reloads.
- The hybrid class abstraction makes DPR accessible to Python developers without deep FPGA expertise.

The combination of DPR, optimized Vivado partitioning, and a Python software control stack achieves efficient, real-time hardware reconfiguration with outstanding resource, timing, and power benefits. The implementation is shown to be robust, scalable, and highly performant for dynamic CNN workloads on constrained edge-FPGA platforms.

5. Conclusion and Future Work

This work presented a **Dynamic Partial Reconfiguration (DPR)**-enabled **Convolution Processing Unit (CPU)** for the Xilinx PYNQ- Z2 platform, designed to accelerate convolutional neural network (CNN) workloads on resource-constrained edge devices. By introducing the pynqpartial Python package and a **hybrid class abstraction**, the proposed system bridges the gap between low-level FPGA reconfiguration and high-level Python-based application development.

The architecture partitions the FPGA fabric into a static region and a reconfigurable partition, enabling runtime swapping of convolution modules with varying kernel sizes and precision levels. Experimental results demonstrated:

- **Up to 800× reduction** in reconfiguration time compared to full bitstream loading.
- **~70% lower LUT usage** versus static multi-core designs, freeing resources for additional accelerators.
- **6.2× throughput improvement** over software-only execution on the ARM cores.
- Stable operation at **100 MHz** across all reconfigurable modules.

These results confirm that DPR, when integrated with a user-friendly API, can deliver both **performance gains** and **design flexibility** for adaptive edge AI applications.

Future Work will focus on:

1. **Multi-RM Scheduling** — Implementing intelligent scheduling algorithms to prefetch and swap RMs based on workload prediction.
2. **Framework Integration** — Extending compatibility with TensorFlow Lite, ONNX Runtime, and PyTorch for seamless model deployment.
3. **Bitstream Compression** — Reducing partial bitstream size to further minimize reconfiguration latency.
4. **Security Enhancements** — Incorporating authentication and encryption for bitstream integrity in mission-critical applications.
5. **Scalability Studies** — Porting the approach to higher-capacity FPGAs and heterogeneous SoCs for larger CNN models.

By combining DPR's hardware adaptability with Python's accessibility, this work lays the foundation for **scalable, reconfigurable, and developer-friendly FPGA-based AI acceleration** at the edge.

References

[1] Gu, J., Wang, H., Guo, X., Schulz, M., & Gerndt, M. (2025). VersaSlot: Efficient fine-grained FPGA sharing with Big.Little slots and live migration in FPGA clusters. *arXiv Preprint arXiv:2503.05930.*

[2] Coyote v2: Raising the level of abstraction for data center FPGA services. (2025). *arXiv Preprint arXiv:2504.21538*.

[3] Hota, A., Xiao, Y., Park, D., & DeHon, A. (2022). HiPR: High-level partial reconfiguration for fast incremental FPGA compilation. *ACM Transactions on Reconfigurable Technology and Systems, 16*(4), 1–26.

[4] Park, D., Xiao, Y., & DeHon, A. (2022). Fast and flexible FPGA development using hierarchical partial reconfiguration. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)* (pp. 1–9). IEEE.

[5] Fuchs, M., Rech, P., Herkersdorf, A., Sterpone, L., & Wirthlin, M. (2024). Cross-chip partial reconfiguration for the initialization of modular and scalable heterogeneous systems. *IEEE Transactions on Nuclear Science*. Advance online publication.

[6] Phani, T. S. S. P., Arumalla, A., Prakash, D. M., & Memisevic, L. (2021). Partial dynamic reconfiguration framework for FPGA: A survey with concepts, constraints and trends. *Materials Today: Proceedings, 49*, 2766–2773.

[7] Boudjadar, J. (2025). Dynamic FPGA reconfiguration for scalable embedded convolutional neural networks. *Future Generation Computer Systems, 163*, 199–210.

[8] Syed, R. T., Gilani, S. A. M., Khan, M. R., Qamar, A., Malik, A. S., & Ayub, N. M. N. (2024). FPGA implementation of a fault-tolerant fused and branched CNN accelerator with reconfigurable capabilities. *IEEE Access, 12*, 57847–57863.

[9] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (pp. 1097–1105).

[10] Chen, Y. H., Krishna, T., Emer, J., & Sze, V. (2017). Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits, 52*(1), 127–138.

[11] Sharma, H., Park, J., Mahajan, D., Amaro, E., Kim, J. K., Lymberopoulos, C., Tsotras, P. A., & Kim, N. S. (2016). From high-level deep neural models to FPGAs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 1–12). IEEE.

[12] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2015). Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (pp. 161–170). ACM.

[13] Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., & Chung, E. (2018). A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1–14). IEEE.

[14] Xilinx Inc. (n.d.). *PYNQ: Python productivity for Zynq*. Retrieved September 11, 2025, from https://pynq.io

[15] Bommana, S. R., Karunakaran, K., Reddy, B. B., & Shankar, A. (2025). Mitigating side-channel attacks on FPGA through dynamic partial reconfiguration. *PLOS ONE, 20*(4), e0300000.

[16] Jain, A. K., Khalid, M. A., & Arslan, T. (2021). FPGA overlays: A survey of techniques and tools. *IEEE Access, 9*, 139–158.

[17] Boudjadar, J. (2025). Dynamic FPGA reconfiguration for scalable embedded convolutional neural networks. *Future Generation Computer Systems, 163*, 199–210.

[18] Jain, A. K., Khalid, M. A., & Arslan, T. (2021). FPGA overlays: A survey of techniques and tools. *IEEE Access, 9*, 139–158.