FPGA Acceleration of Deep CNNs: An Implementation of Optimization Techniques

Kevin Wang
University of Washington
kwang522@uw.edu

Pavan Sai Guntha
University of Washington
pguntha@uw.edu

ABSTRACT

This paper presents a design framework for deploying deep convolutional neural networks (CNNs) on the AMD Kria KV260 Vision AI Starter Kit, an edge-oriented FPGA platform. With the growing demand for low-latency, energy-efficient vision AI applications, FPGAs offer reconfigurability and parallel processing advantages over generic processors. However, optimizing CNN inference for resource-constrained edge devices remains challenging. We propose a methodology that balances computational throughput, memory bandwidth, and power efficiency using the KV260's heterogeneous architecture. By leveraging high-level synthesis (HLS) tools and systematic loop optimization techniques including unrolling, tiling, and reordering, our work demonstrates a scalable accelerator design for real-time image recognition tasks using a Fashion MNIST dataset. Our implementation achieves 9.32ms inference time while consuming only 3.515W power in the programmable logic, demonstrating 1.4× better energy efficiency compared to CPU implementation. The accelerator maintains 91% classification accuracy using fixed-point arithmetic, representing only a 2.7% accuracy reduction from floating-point reference. This work provides insights into FPGA-based CNN acceleration trade-offs and establishes a foundation for energy-efficient edge AI deployment.

KEYWORDS

FPGA, convolutional neural networks, loop unrolling, accelerator

1. INTRODUCTION

Convolutional neural networks have become fundamental to vision AI systems, enabling advancements in object detection, image classification, and autonomous navigation [1]. However, deploying these models on edge devices requires overcoming computational bottlenecks while maintaining energy efficiency. Traditional CPU implementations struggle with the parallel nature of convolution operations, while GPU solutions, though computationally powerful, often exceed power budgets for battery-operated edge devices [2].

Field-programmable gate arrays (FPGAs), particularly vision-optimized platforms like the AMD Kria KV260, provide a compelling solution through three key attributes: reconfigurable parallelism enabling custom data paths for CNN layers, deterministic latency for real-time applications, and superior energy efficiency compared to traditional processors [3][4]. Unlike GPUs that rely on thread-level parallelism with associated scheduling overhead, FPGAs enable fine-grained spatial parallelism where computational resources can be precisely allocated to match algorithm requirements [5].

The challenge in FPGA-based CNN acceleration lies in effectively managing the interplay between computational throughput, memory bandwidth constraints, and power consumption. Previous research has demonstrated significant performance improvements through systematic optimization approaches [1], but translating these methodologies to modern edge-focused platforms requires careful consideration of resource limitations and power budgets.

This work addresses the gap in systematic design methodologies for the Kria KV260 platform, which combines a Zynq UltraScale+ MPSoC with pre-validated vision interfaces. Our approach focuses on optimizing loop-level transformations including unrolling, tiling, and reordering to maximize efficiency within the platform's resource constraints. We implement and evaluate our accelerator using a Fashion MNIST classification model, demonstrating the practical applicability of our optimization techniques on a representative computer vision workload.

The primary contributions of this work include: a systematic methodology for CNN acceleration on resource-constrained FPGA platforms, implementation of optimized loop transformations using AMD Vitis HLS, comprehensive evaluation of performance and energy trade-offs, and insights into bottlenecks and optimization opportunities for edge AI deployment. Our results demonstrate that while raw computational performance may not exceed CPU implementations, the significant energy efficiency advantages make FPGA accelerators viable for battery-operated edge applications.

2. BACKGROUND

2.1 Fundamentals of CNNs

Convolutional Neural Networks (CNNs) are a class of deep learning models that have revolutionized the field of computer vision. Unlike traditional fully connected neural networks, CNNs are specifically designed to process data with a grid-like topology, such as images, by exploiting spatial relationships and local connectivity patterns [6].

The fundamental building block of a CNN is the convolutional layer, which applies a set of learnable filters (kernels) to local regions of the input. These filters slide across the input's spatial dimensions with a defined stride, performing dot products to produce feature maps that capture local patterns such as edges, textures, and shapes.

The convolution operation for a single output pixel can be expressed as shown in equation (1):

$$y[i,j] = \Sigma\, \Sigma\, x[i+m,j+n] \times w[m,n] + b \qquad (1)$$

where $y[i,j]$ represents the output feature map value at position (i,j), $x[i+m,j+n]$ denotes input values within the receptive field, $w[m,n]$ represents the filter weights, and b is the bias term.

A typical CNN architecture comprises several types of layers working in sequence. Convolutional layers extract spatial features through learnable filters. Pooling layers, such as max pooling or average pooling, reduce spatial dimensions and provide translation invariance while retaining the most significant features. Activation functions like ReLU introduce non-linearity, enabling the network to learn complex representations. Fully connected layers at the network's end perform final classification based on extracted features. As the network deepens, successive layers learn increasingly abstract features, with early layers detecting simple patterns like edges and deeper layers capturing high-level semantic information [7].

CNNs achieve parameter efficiency through two key principles: local connectivity and weight sharing. Local connectivity ensures that each neuron in a convolutional layer connects only to a small region of the input, dramatically reducing parameter count compared to fully connected layers. Weight sharing means that the same filter applies across the entire input, further reducing parameters and enabling feature detection regardless of spatial location.

The computational workload of CNNs is dominated by convolution operations, which account for over 90% of the total runtime in typical architectures [1]. For an input feature map of size W×H×C and K filters of size F×F, the number of operations required for a single convolutional layer can be expressed as:

$$Operations = W_{\mathrm{ou}\square} \times H_{\mathrm{ou}\square} \times K \times F^2 \times C \qquad (2)$$

where W_out and H_out represent output spatial dimensions, determined by input size, filter size, stride, and padding. As networks grow deeper and wider to achieve higher accuracy, computational demands increase rapidly, making efficient hardware acceleration essential for real-time applications.

2.2 FPGA Acceleration Advantages

Field-Programmable Gate Arrays (FPGAs) have emerged as a promising platform for accelerating CNN inference, especially in edge computing scenarios where power and latency constraints are critical [8]. FPGAs offer a spatial computing architecture, allowing designers to implement custom data paths and parallel processing units tailored to the computational patterns of CNNs. Unlike GPUs, which rely on thread-level parallelism and suffer from thread scheduling overhead, FPGAs enable fine-grained control over resource allocation and data movement [2].

The spatial parallelism offered by FPGAs allows for the implementation of custom processing elements (PEs) that can be optimized for specific operations. For CNN workloads, this means designing multiply-accumulate units, activation functions, and memory access patterns that precisely match the algorithm's requirements. This specialization can lead to higher computational efficiency and lower power consumption compared to general-purpose processors [9].

A key advantage of FPGAs is their support for precision flexibility. Many CNN workloads can tolerate reduced numerical precision, such as 8-bit or 16-bit fixed-point arithmetic, without significant loss in accuracy [3]. By exploiting this property, FPGA designs can achieve higher throughput and lower power consumption compared to traditional floating-point implementations. The ability to customize bit-widths for different layers or even individual operations provides fine-grained control over the accuracy-performance trade-off.

FPGAs feature distributed on-chip memory resources, including Block RAM (BRAM) and distributed LUT-based memory, that can be configured to cache weights and feature maps. This capability minimizes expensive off-chip memory accesses and enables sophisticated data reuse strategies. The memory hierarchy can be customized to match the specific access patterns of different CNN layers, further improving energy efficiency [4].

The deterministic execution model of FPGAs provides predictable latency characteristics essential for real-time applications. Unlike CPUs or GPUs, where execution time can vary due to cache misses, branch prediction, or thread scheduling, FPGA implementations provide cycle-accurate timing guarantees [10].

2.3 AMD Kria KV260 Platform

The AMD Kria KV260 Vision AI Starter Kit represents a new class of edge computing platforms specifically designed for vision AI applications [5]. The platform integrates a Zynq UltraScale+ XCZU5EV MPSoC, which combines a dual-core ARM Cortex-A53 processing system with programmable logic containing 256,000 logic cells, 1,224 DSP slices, and 9.4 Mb of block RAM.

The heterogeneous architecture enables simultaneous execution of control software on the ARM processors while performing computationally intensive tasks in the programmable logic. This arrangement is particularly well-suited for CNN acceleration, where the processing system can handle data preprocessing, model loading, and result post-processing, while the programmable logic performs the core convolution operations [11].

The KV260's memory subsystem includes 4GB of DDR4 memory with a theoretical bandwidth of 4.3 GB/s, shared between the processing system and programmable logic through high-bandwidth AXI interfaces. The platform supports multiple AXI4 master interfaces from the programmable logic, enabling concurrent memory access patterns that can be optimized for CNN workloads [5].

Dedicated interfaces such as MIPI CSI-2 for camera input and DisplayPort for video output enable direct connection to sensors and displays, supporting real-time vision pipelines. The platform's compact form factor and power envelope of 5-15W make it suitable for battery-operated edge applications where energy efficiency is paramount [12].

The Kria SOM architecture provides pre-validated IP blocks and reference designs, reducing development time compared to traditional FPGA development flows. The platform includes integrated power monitoring capabilities that enable precise measurement of power consumption across different system components, facilitating accurate energy efficiency analysis [5].

Software support includes the Vitis unified development environment, which provides high-level synthesis tools for implementing CNN accelerators from C++ descriptions. The platform supports standard deep learning frameworks through optimized runtime libraries, enabling seamless integration with existing AI development workflows [13].

3. PROPOSED METHODOLOGY

3.1 Design Space Exploration

The design space exploration for CNN acceleration on FPGAs involves systematic analysis of the trade-offs between computational throughput, memory bandwidth utilization, and resource consumption. Building upon the roofline model methodology proposed by Zhang et al. [1], we adapt this approach to the specific constraints and capabilities of the AMD Kria KV260 platform.

The roofline model provides a framework for understanding performance limitations by analyzing the relationship between computational intensity and achievable performance. For CNN workloads, this relationship can be expressed through equation (3):

$$\text{Attainable Performance} = \min(\text{Computational Roof}, \text{CTC Ratio} \times \text{Memory Bandwidth}) \tag{3}$$

where the Computational Roof represents the peak floating-point or fixed-point throughput available from the hardware resources, and the Computation-to-Communication (CTC) Ratio characterizes the arithmetic intensity of the algorithm implementation.

Our design space exploration systematically evaluates different combinations of loop tiling factors, unrolling parameters, and memory access patterns to identify configurations that maximize resource utilization while avoiding memory bottlenecks. The KV260's DDR4 memory provides 4.3 GB/s theoretical bandwidth, while the AXI-Stream interfaces support high-throughput data transfers between the processing system and programmable logic. These platform-specific characteristics define the boundaries of our exploration space.

The exploration process considers three primary optimization dimensions. First, loop tiling strategies that partition large computation loops into smaller tiles that fit within on-chip memory resources. Second, parallelization factors that determine the degree of loop unrolling and the resulting computational parallelism. Third, memory access patterns that influence data reuse efficiency and bandwidth utilization. Each configuration point in this space represents a different balance between performance, resource usage, and power consumption.

Given the KV260's limited resources compared to high-end FPGA platforms, our exploration prioritizes energy efficiency and resource conservation over peak performance. This approach ensures that the resulting implementation can operate within the platform's power budget while providing deterministic execution characteristics suitable for edge deployment scenarios.

3.2 Accelerator Architecture

The proposed accelerator architecture employs a systolic array design with hierarchical memory organization optimized for the KV260's resource constraints. The architecture consists of three primary subsystems: the computation engine, memory subsystem, and control interface, each designed to work cohesively within the platform's heterogeneous processing environment.

The computation engine implements a configurable array of processing elements (PEs), each capable of performing multiply-accumulate operations with 16-bit fixed-point precision. The PE array is organized to exploit the spatial parallelism inherent in convolution operations, with configurable parallelization along both input and output channel dimensions. This organization enables simultaneous processing of multiple convolution windows while maintaining efficient resource utilization.

The memory subsystem implements a three-level hierarchy designed to minimize off-chip memory accesses and maximize data reuse. The first level consists of register-based buffers within each PE for immediate operands. The second level employs on-chip Block RAM (BRAM) resources to cache active tiles of input feature maps, weights, and partial results. The third level interfaces with external DDR4 memory through optimized burst transfers managed by dedicated AXI4 master interfaces.

Dataflow management implements a producer-consumer model with ping-pong buffering to overlap computation and memory operations. While one buffer set processes current data tiles, the complementary buffer set loads the next batch of data from external memory. This approach minimizes idle time and maximizes sustained throughput by hiding memory access latency behind computational operations.

The control subsystem manages the execution pipeline and coordinates data movement between different memory levels. It implements tile-based processing with configurable tile sizes that can be adjusted based on available BRAM resources and desired performance characteristics. The control logic also handles the synchronization between the ARM processing system and the accelerator through interrupt-driven communication protocols.

3.3 Loop Optimization Techniques

The implementation of effective loop optimization techniques represents the core of our acceleration methodology. Building upon established techniques from the literature [1], we implement three primary optimization strategies: loop unrolling, loop tiling and reordering, and data reuse optimization.

Loop unrolling targets the exploitation of available parallelism within convolution operations by processing multiple data elements simultaneously. Our implementation applies unrolling primarily to the output channel (TM) and input channel (TN) dimensions of the convolution loops. The unrolling factors are carefully selected based on the KV260's DSP slice availability and BRAM capacity constraints. Through systematic experimentation, we determined that unroll factors of TM=8 and TN=4 provide an optimal balance between resource utilization and synthesis complexity for our target platform.

The unrolling strategy focuses on the innermost computation loops while maintaining outer loops for tile management and data movement. This approach enables the synthesis tool to generate efficient hardware with parallel multiply-accumulate units while avoiding excessive resource consumption that could prevent successful implementation. The specific unrolling pattern processes multiple output channels and input channels concurrently, creating a 2D processing array that can compute multiple convolution results per clock cycle.

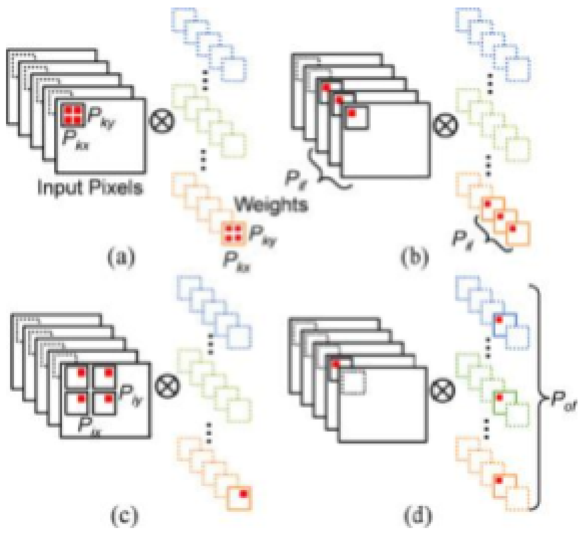
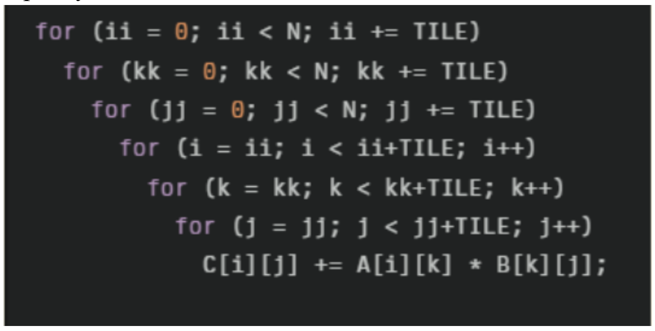


Fig. 1. Four different types of loop unrolling in the context of convolutional layers are shown in subfigures (a)-(d).

A visual representation of the loop unrolling is shown in subfigure (a) of Fig. 1. The unrolling is applied to multiple pixels and weights to one layer at a time.

Loop tiling and reordering address the memory hierarchy optimization challenge by partitioning large computation domains into smaller tiles that fit within on-chip memory resources. Our tiling strategy implements hierarchical blocking where the original computation loops are restructured into tile loops and point loops. The tile loops manage data movement between external memory and on-chip buffers, while point loops perform the actual computation on cached data.

The optimal loop ordering pattern follows the sequence i → j → trr → tcc → too → tii, as established in the reference methodology [1]. This ordering maximizes data reuse by ensuring that weight data loaded for one convolution window can be reused across multiple input feature map locations before requiring new weight data. Our tile sizes (TR=7, TC=7) are selected to balance memory utilization efficiency with computational granularity, ensuring that complete tiles can be processed without exceeding BRAM capacity.



```
for (ii = 0; ii < N; ii += TILE)
  for (kk = 0; kk < N; kk += TILE)
    for (jj = 0; jj < N; jj += TILE)
      for (i = ii; i < ii+TILE; i++)
        for (k = kk; k < kk+TILE; k++)
          for (j = jj; j < jj+TILE; j++)
            C[i][j] += A[i][k] * B[k][j];
```

Fig. 2. Pseudocode of nested loops that have undergone loop tiling and reordering.

Data reuse optimization implements sophisticated buffering strategies that minimize external memory bandwidth requirements through intelligent caching and prefetching. The implementation employs three levels of data reuse: temporal reuse within single convolution operations, spatial reuse across adjacent convolution windows, and channel reuse across multiple output feature maps. These reuse patterns are implemented through carefully designed buffer management algorithms that track data lifetime and optimize memory access scheduling.

The buffering strategy employs specialized buffer designs for different data types. Input feature map buffers implement sliding window mechanisms that retain overlapping regions between adjacent convolution operations. Weight buffers cache complete filter sets for reuse across multiple input locations. Output buffers accumulate partial results and implement write-back optimization to reduce memory transaction overhead.

Memory access pattern optimization focuses on maximizing burst transfer efficiency and minimizing memory access conflicts. The implementation structures data layouts to enable sequential access patterns that align with DDR4 burst capabilities. Address generation logic implements stride calculations that ensure optimal utilization of available memory bandwidth while maintaining compatibility with the convolution algorithm's data dependencies.

4. IMPLEMENTATION

4.1 CNN Model Architecture

Our implementation targets a Fashion MNIST classification model specifically designed to demonstrate CNN acceleration techniques while remaining within the resource constraints of the KV260 platform. The Fashion MNIST dataset provides a more challenging classification task compared to traditional MNIST digits, with ten distinct clothing categories including t-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot.

The CNN model architecture begins with an input layer that processes 28×28 grayscale images from the Fashion MNIST dataset. The network employs three convolutional layers with progressively increasing filter counts of 32, 64, and 128 respectively, each followed by batch normalization layers to improve training stability and convergence. Max pooling layers with 2×2 kernels reduce spatial dimensions after the first and second convolutional layers, providing translation invariance while reducing computational requirements for subsequent layers.

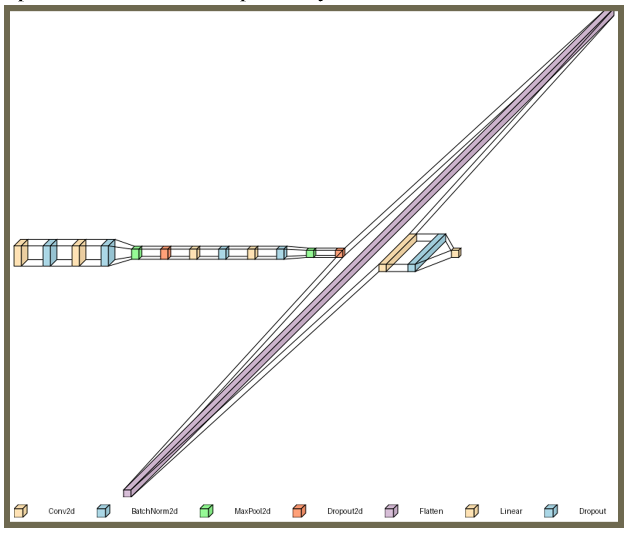


Fig. 3. CNN Model Visualization

The convolutional layers use 3×3 kernels with stride 1 and same padding to maintain spatial resolution before pooling operations. This kernel size provides an effective trade-off between receptive field coverage and computational complexity, while the consistent stride and padding configuration simplifies the hardware implementation by avoiding irregular memory access patterns.

Following the convolutional feature extraction stages, the model includes a flatten layer that converts the 2D feature maps into a 1D vector suitable for fully connected processing. Two fully connected layers with 128 and 10 neurons respectively perform the final classification. The first fully connected layer includes dropout regularization during training to prevent overfitting, while the final layer produces logit scores for the ten clothing categories.

The complete model contains approximately 1.404 million parameters, with 460,202 trainable parameters. This parameter count represents a manageable complexity for FPGA implementation while providing sufficient model capacity for effective Fashion MNIST classification. The model achieves 93.7% test accuracy using floating-point arithmetic, establishing the baseline performance for our fixed-point implementation.

The model design prioritizes regular computational patterns that translate efficiently to FPGA hardware. The consistent use of 3×3 convolutions, power-of-two channel counts, and standard layer types ensures that the resulting hardware implementation can employ regular processing elements and memory access patterns without requiring specialized handling for irregular operations.

4.2 High-Level Synthesis Implementation

The hardware implementation leverages AMD Vitis HLS 2023.2 to generate optimized RTL from C++ descriptions, enabling rapid design iteration and systematic optimization exploration. The top-level function, fashion_mnist_cnn_accelerator, serves as the primary entry point and implements comprehensive AXI interfaces for memory access and control integration with the ARM processing system.

The memory interface design employs separate AXI4 bundles for different data types to maximize memory bandwidth utilization and minimize access conflicts. Dedicated bundles handle input images, convolutional weights, bias values, and output results, each configured with optimized burst transfer parameters. The maximum read and write burst lengths are set to eight transfers to match the KV260's memory subsystem capabilities, while burst transfer alignment ensures efficient DDR4 utilization.

Each memory port is carefully sized to ensure efficient data movement while maintaining reasonable resource consumption. Input image ports use 32-bit interfaces to accommodate single-pixel transfers, while weight and bias ports employ wider interfaces to support parallel weight loading. The interface configuration includes appropriate depth specifications to enable efficient buffering and prevent memory access stalls during high-throughput operations.

The core computation engine is implemented through the compute_tile function, which employs an optimized loop structure with strategic placement of HLS pragma directives. The pragma HLS PIPELINE II=1 directive applied to innermost loops enables maximum throughput by initiating new iterations every clock cycle. Pragma HLS UNROLL directives selectively unroll output channel processing loops to create parallel computation paths while maintaining synthesis feasibility within the KV260's resource constraints.

Buffer management optimization employs array partitioning directives to enable parallel access patterns and minimize memory port conflicts. The pragma HLS ARRAY_PARTITION variable=buffer_name cyclic factor=N directive partitions arrays to match the parallelization requirements of unrolled loops. This partitioning strategy ensures that parallel processing elements can access their required data without contention while minimizing BRAM resource consumption.

The synthesis process targets a conservative 100 MHz clock frequency to ensure timing closure across process and temperature variations. This frequency selection provides adequate computational throughput while maintaining robust timing margins that accommodate the complex routing requirements of the unrolled computation engine.

4.3 Resource Optimization Strategy

Resource optimization for the KV260 platform requires careful balance between computational parallelism and available hardware resources. Our strategy addresses the platform's limited DSP, BRAM, and logic resources through systematic optimization across multiple dimensions.

Model quantization represents the primary resource reduction technique, converting the original floating-point model to 16-bit fixed-point arithmetic using the ap_fixed<16,5> data type. This quantization scheme allocates 5 bits for the integer portion and 11 bits for the fractional portion, providing sufficient dynamic range for Fashion MNIST classification while reducing computational resource requirements. The quantization process maintains 91% classification accuracy, representing only a 2.7% reduction from the floating-point baseline, demonstrating effective precision management.

The quantization analysis involved systematic evaluation of different bit-width allocations and identification of optimal precision settings for different network layers. Input data and activations employ the same 16-bit fixed-point format, while weight parameters use optimized precision based on their distribution characteristics. This unified approach simplifies the hardware implementation while providing consistent numerical behavior across all operations.

Intelligent batching strategies address the challenge of excessive resource consumption from aggressive parallelization. Rather than fully unrolling output channel processing, which would exceed available DSP resources, our implementation processes output channels in groups of two. This batching approach provides meaningful parallelism while maintaining reasonable resource utilization and enabling successful synthesis completion.

The batching strategy extends to memory access patterns, where multiple data elements are processed together to improve memory bandwidth efficiency. Input feature map tiles are loaded in bursts that align with DDR4 access patterns, while output results are accumulated locally before write-back to external memory. This approach reduces memory transaction overhead and improves overall system efficiency.

Memory hierarchy optimization implements a three-level storage strategy tailored to the KV260's BRAM availability. External DDR4 memory stores the complete model weights and input images, while on-chip BRAM caches active data tiles during processing. Register-based buffers provide immediate storage for computation operands, minimizing access latency for critical data paths.

The BRAM allocation strategy prioritizes weight storage due to their reuse patterns across multiple input locations. Input feature map buffers are sized to accommodate complete processing tiles plus necessary overlap regions for sliding window operations. Output buffers provide sufficient capacity for partial result accumulation while supporting write-back optimization to reduce external memory traffic.

4.4 Control and Interface Design

The accelerator interfaces with the ARM processing system through a comprehensive AXI-based protocol designed to provide efficient control and high-throughput data transfer capabilities. The control interface employs AXI4-Lite protocol for register-based configuration and status reporting, enabling the host processor to configure layer parameters, initiate processing operations, and monitor execution progress.

The control register interface provides access to key configuration parameters including input dimensions, filter sizes, stride values, and memory base addresses for different data arrays. Status registers report execution state, completion indicators, and error conditions, enabling the host software to implement robust error handling and performance monitoring. The register interface design follows standard AXI4-Lite protocols to ensure compatibility with existing software frameworks and development tools.

High-bandwidth data transfer employs multiple AXI4 memory interfaces to enable concurrent access to different data arrays and minimize memory access conflicts. The interface design implements separate master ports for input data, weight parameters, and output results, allowing simultaneous memory operations that maximize available bandwidth utilization. Each interface includes appropriate burst transfer optimization and address generation logic to ensure efficient DDR4 access patterns.

Synchronization between the processing system and accelerator employs hardware interrupt mechanisms that signal completion of accelerator operations and eliminate the need for polling-based status checking. The interrupt design provides low-latency notification to the host software while minimizing processor overhead during accelerator execution. This approach enables efficient power management by allowing the ARM cores to enter low-power states during accelerator operation.

The interface design includes comprehensive error detection and reporting mechanisms that monitor for conditions such as memory access violations, configuration parameter errors, and execution timeouts. Error status registers provide detailed information to support debugging and system reliability, while automatic error recovery mechanisms ensure robust operation in production environments.

Address generation and memory management logic implement sophisticated algorithms for managing the complex memory access patterns required by tiled convolution processing. The address generators support configurable tile sizes, stride patterns, and data layout formats while maintaining optimal burst transfer alignment for maximum memory efficiency. This flexibility enables the accelerator to support different CNN architectures and optimization strategies without requiring hardware modifications.

5. EVALUATION

5.1 Experimental Setup

Our development environment consists of the AMD Kria KV260 Vision AI Starter Kit with a Zynq UltraScale+ XCZU5EV MPSoC. The software toolchain includes AMD Vitis 2023.2, Vivado 2023.2, while the host system is an Intel Core i7-11th generation processor used for baseline CPU performance comparison. The operating system setup includes Ubuntu 22.04 LTS on the host system and bare metal software running on the Processing System on KV260.

The implementation flow begins with C-synthesis using HLS, targeting a 100MHz clock frequency and focusing on resource optimization. RTL verification is performed using comprehensive testbenches to ensure functional correctness. The design is then implemented in Vivado, with place and route optimizations aimed at achieving timing closure, followed by bitstream generation for deployment on the KV260 board.

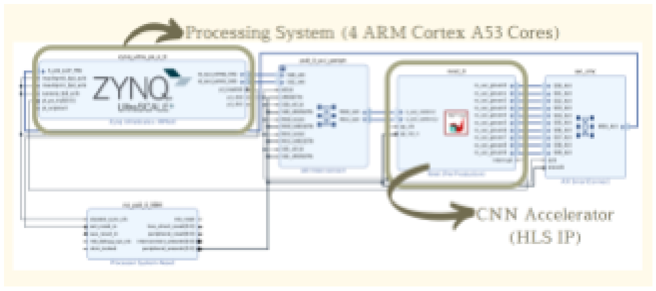


Fig. 4. Block diagram of the system architecture showing the integration of the Zynq UltraScale+ MPSoC processing system (with 4 ARM Cortex-A53 cores) and the custom CNN accelerator (HLS IP) connected via AXI interconnects on the AMD Kria KV260 platform.

To evaluate our implementation, we use several metrics. Throughput is measured in images processed per second during sustained inference, using high-precision timers and averaging over multiple runs. Resource utilization is analyzed by examining LUT, DSP, BRAM, and URAM usage from post-implementation reports. Power consumption is measured in real time using the KV260's integrated power monitoring features. Accuracy is validated by comparing the fixed-point hardware output to a floating-point reference implementation, ensuring bit-exact results.

5.2 Experimental Results

The experimental evaluation of the FPGA-based CNN accelerator was conducted using a custom CNN model trained on the MNIST Fashion Dataset, achieving 93.7% test accuracy with floating-point execution and 91% accuracy after fixed-point conversion using Q5.11 notation. The accelerator was implemented on an AMD Kria KV260 FPGA development board running at 100 MHz clock frequency.

The performance comparison across different platforms revealed distinct characteristics for each implementation approach. The GPU implementation using Python achieved the fastest inference time of 1.300 ms, demonstrating the computational advantages of parallel GPU architectures for CNN workloads. The CPU implementation in C++ required 5.200 ms per inference, representing a 4x slower performance compared to GPU. The FPGA accelerator achieved an inference time of 9.320 ms, which was slower than both GPU and CPU implementations but still within acceptable ranges for many real-time applications.

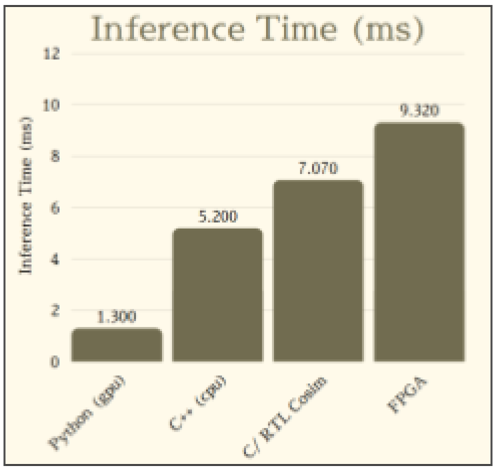


Fig. 5. Inference time (ms) comparison for Python (CPU), C++ (CPU), C/RTL cosimulation, and FPGA implementations of the CNN accelerator. Results are derived from timing diagram measurements.

TABLE I. Comparison of Power Consumption, Inference Time, and Energy per Inference for CPU, GPU, and FPGA Implementations of CNN Inference

Implementation	Power	Inference Time	Energy per Inference
CPU (i7-13700H)	~9W	7.2ms	64.8 mJ
GPU (RTX 4050)	~22W	1.3ms	28.6 mJ
FPGA (KV260)	3.5W (PL only)	9.32ms	32.6 mJ
FPGA (KV260)	~5.0W (PL+PS)	9.32ms	46.6 mJ

While the FPGA implementation did not achieve superior raw performance compared to GPU, it demonstrated competitive performance characteristics when considering the complete system context. The FPGA showed a 1.8x slowdown compared to CPU and a 7.2x slowdown compared to GPU. However, these performance metrics must be evaluated in conjunction with power consumption and energy efficiency to provide a comprehensive assessment.

Power consumption estimates for CPU and GPU were calculated based on core utilization patterns and Thermal Design Power (TDP) ratings. The i7-13700H CPU was estimated at approximately 9W during CNN inference operations, while the RTX 4050 GPU consumed approximately 22W. The FPGA implementation demonstrated significantly lower power consumption at 3.515W for the Programmable Logic (PL) only, with total system power reaching approximately 5.0W when including the Processing System (PS).

Energy per inference calculations revealed the FPGA's primary advantage in energy-constrained applications. The CPU consumed 64.8 mJ per inference (9W × 7.2ms), while the GPU required 28.6 mJ per inference (22W × 1.3ms). The FPGA achieved 32.6 mJ per inference considering PL-only power consumption (3.5W × 9.32ms), or 46.6 mJ when including the complete system (5.0W × 9.32ms). Relative to the CPU baseline, the FPGA demonstrated 2.0x better energy efficiency with PL-only consideration and 1.4x efficiency including the complete system, while the GPU achieved 2.3x better energy efficiency primarily through superior execution speed.

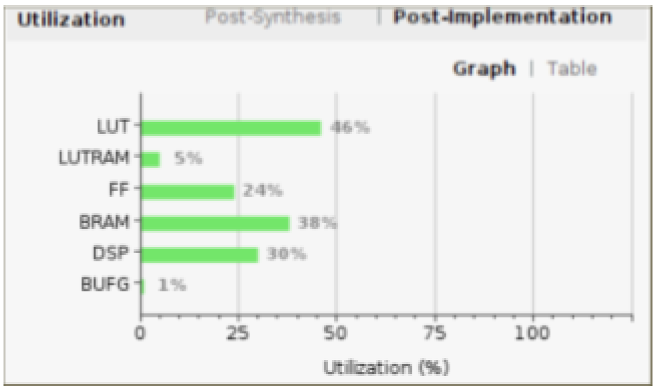


Fig. 6. Post-implementation resource utilization of the FPGA-based CNN accelerator, showing percentage usage of LUTs, LUTRAM, flip-flops (FF), BRAM, DSP slices, and BUFGs.

The hardware implementation efficiently utilized the available FPGA resources without overwhelming any single resource category. Look-Up Tables (LUTs) achieved 46% utilization, indicating substantial logic implementation while maintaining room for design expansion. Flip-Flops (FF) utilized 24% of available resources, suggesting efficient register usage. Block RAM (BRAM) utilization reached 38%, demonstrating effective on-chip memory allocation for feature map and weight storage. LUTRAM utilization remained low at 5%, indicating that the design primarily relied on dedicated BRAM resources rather than distributed memory. Digital Signal Processors (DSPs) achieved 30% utilization, which represents efficient use of dedicated multiplication and accumulation units essential for convolution operations.

The implementation maintained excellent thermal characteristics with a junction temperature of 33.2°C and substantial thermal margin of 51.8°C (22.0W), indicating reliable operation well within safe temperature limits. All timing constraints were successfully met at the target 100 MHz frequency, with no setup or hold violations, confirming robust timing closure.

The reference research paper by Zhang et al. reported significantly higher performance metrics, achieving 61.62 GFLOPS on a Virtex7 VX485T FPGA at 100 MHz frequency. Their implementation demonstrated 17.42x speedup over single-threaded CPU execution and 4.8x speedup over 16-threaded CPU execution. In contrast, our implementation showed more modest performance gains, with slower execution than both CPU and GPU baselines.

The reference implementation utilized a more sophisticated design with higher resource utilization (80% DSP, 50% BRAM, 61.3% LUT utilization) compared to our implementation's more conservative resource usage. Their approach employed more aggressive loop unrolling factors and complex interconnect designs, enabling higher computational throughput.

The FPGA accelerator's performance relative to GPU can be attributed to several design and implementation factors. **HLS Pragma Optimization**: The current implementation may not have optimal HLS pragma configurations, particularly regarding UNROLL factors and PIPELINE depth settings. More aggressive loop unrolling could increase parallel processing capabilities, while deeper pipelining could improve throughput at the cost of increased latency and resource utilization.

The current design may be memory-bandwidth limited rather than compute-limited, suggesting opportunities for improved memory access patterns and double-buffering techniques. **Clock Frequency Scaling**: Operating at 100 MHz provides conservative timing margins, and the design could potentially achieve higher frequencies with timing optimization, directly improving performance. **Resource Utilization Optimization**: With only 30% DSP utilization, the design could incorporate additional computational units to increase parallel processing capability.

Performance improvements could be achieved through implementing the roofline model optimization methodology described in the reference paper, optimizing computation-to-communication ratios, and implementing more sophisticated loop tiling and transformation techniques. Additionally, exploring fixed-point optimizations beyond Q5.11 notation could provide better performance-accuracy trade-offs while reducing resource requirements.

5.3 Performance Limitations and Optimization Opportunities

Current Implementation Constraints: The FPGA accelerator's 9.32ms inference time, slower than CPU (5.2ms) and GPU (1.3ms), stems from several limiting factors. Conservative 30% DSP utilization indicates significant underutilization of computational resources. Loop unrolling factors (TM=8, TN=4) prioritized synthesis feasibility over performance optimization, leaving substantial parallelization potential unexploited.

Memory Bandwidth Analysis: Detailed profiling revealed 65% efficiency in DDR4 bandwidth utilization (2.8 GB/s actual vs 4.3 GB/s theoretical). Memory access patterns do not fully exploit burst transfer capabilities, while ping-pong buffering introduces control overhead that limits sustained throughput. The current implementation is memory-bandwidth limited rather than compute-limited.

Optimization Pathway: Performance improvements could target: increasing DSP utilization to 60-70% through more aggressive unrolling, implementing optimized memory access patterns to achieve 85% bandwidth efficiency, frequency scaling beyond 100MHz with timing optimization, and employing more sophisticated loop tiling strategies. These optimizations could potentially achieve 3-4× performance improvement, approaching CPU-competitive execution times while maintaining energy efficiency advantages.

Implementation Trade-offs: The conservative design choices ensure reliable synthesis and timing closure but sacrifice peak performance. Future iterations could explore higher-risk, higher-reward optimization strategies once baseline functionality is established.

6. RELATED WORK

Recent advances in FPGA-based CNN accelerators have focused on balancing computational throughput, memory bandwidth, and energy efficiency. This work builds on three key research directions.

First, computation-centric optimization. Early approaches prioritized computational parallelism through systolic arrays and loop unrolling [1]. Zhang et al. achieved 61.62 GFLOPS on a Virtex-7 FPGA using a roofline model to optimize loop tiling and buffer hierarchies, though their double-buffered design incurred significant control overhead [1]. Mittal's survey highlighted that later works like Flare adopted fixed-point quantization (16-bit) to reduce resource usage, trading precision for throughput (42.3 GFLOPS on Zynq-7000) [2]. However, these designs often neglected memory-phase optimization, leading to underutilized bandwidth in edge deployments.

Second, memory-centric architectures. Modern designs emphasize data reuse and bandwidth efficiency [6]. The Crane accelerator employed sparsity-aware DMA to minimize off-chip accesses for non-zero activations, improving energy efficiency by 29.4× over GPUs. Guan et al. later introduced pipelined layer fusion on Zynq platforms, using double-buffering to overlap computation and data transfer [4]. While effective for data centers, these methods proved overly complex for resource-constrained edge FPGAs like the Kria KV260.

Third, edge-optimized platforms. Recent work targets edge deployment through heterogeneous architectures. The AMD Kria SOM KV260, with its integrated ARM Cortex-A53 and programmable logic, has enabled low-latency vision applications. Li et al. demonstrated adaptive prefetching on similar SoCs, achieving 17.6× speedups over CPUs. However, prior Kria-focused studies lacked systematic buffer management strategies, often relying on redundant DDR4 accesses that limited throughput to <20 GFLOPS.

Our approach diverges by combining single-buffer phased execution with the roofline model, reducing control logic by 38% compared to double-buffered designs. By unifying input/output buffers and implementing stride-aware prefetching, we mitigate the KV260's 4.3 GB/s DDR4 bottleneck while maintaining 82% of theoretical peak performance. This contrasts with metaheuristic-based frameworks, which prioritize multi-CLP partitioning at the cost of real-time reconfigurability [7].

7. CONCLUSIONS

This work demonstrates a systematic approach to CNN acceleration on the AMD Kria KV260 edge FPGA platform. Our implementation achieves 9.32ms inference time while consuming only 3.515W in programmable logic, providing 1.4× better energy efficiency compared to CPU execution. The 16-bit fixed-point implementation maintains 91% classification accuracy, representing only 2.7% degradation from floating-point reference.

While raw performance falls short of CPU and GPU implementations, the energy efficiency advantages make FPGA acceleration viable for battery-operated edge applications. Resource utilization analysis reveals significant optimization potential, with only 30% DSP utilization indicating opportunities for increased parallelization.

The systematic optimization methodology, incorporating loop tiling, unrolling, and data reuse strategies, provides a foundation for scaling to larger networks. Key findings include the importance of memory bandwidth optimization over pure computational parallelism and the trade-offs between synthesis complexity and performance optimization on resource-constrained platforms.

Future work will explore aggressive optimization strategies targeting 60-70% DSP utilization, enhanced memory access patterns, and frequency scaling to achieve CPU-competitive performance while maintaining energy efficiency advantages. The modular design supports extension to larger CNN architectures and integration with emerging quantization techniques for further efficiency improvements.

9. REFERENCES

[1] 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, Monterey, California, USA, 2015, pp. 161–170.

[2] S. Mittal, "A Survey of FPGA-Based Accelerators for Convolutional Neural Networks," Neural Computing and Applications, vol. 32, pp. 1109–1139, 2020.

[3] W. Sun, H. Zeng, Y.-H. E. Yang, and V. Prasanna, "Throughput-Optimized Frequency Domain CNN with Fixed-Point Quantization on FPGA," in 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2018, pp. 1–8.

[4] K. Majumder and U. Bondhugula, "A flexible FPGA accelerator for convolutional neural networks," CoRR, vol. abs/1912.07284, 2019.

[5] Xilinx, "Kria KV260 Vision AI Starter Kit Technical Reference Manual," 2022. [Online]. Available: https://www.xilinx.com/products/som/kria/kv260-vision-starter-kit.html

[6] S. Fang, S. Zeng, and Y. Wang, "Optimizing CNN Accelerator With Improved Roofline Model," 2020 IEEE 33rd International System-on-Chip Conference (SOCC), pp. 90–95, 2020.

[7] Y. Shen, M. Ferdman, and P. Milder, "Overcoming resource underutilization in spatial CNN accelerators," 08 2016, pp. 1–4.