ResNet18 Hardware Accelerating Implementation with Kria KV260

Hung-Ting Tsai
Viterbi School of Engineering
University of Southern California
Los Angeles, USA
hungting@usc.edu

Junsang Yoo
Viterbi School of Engineering
University of Southern California
Los Angeles, USA
junsang@usc.edu

You-You Lin
Viterbi School of Engineering
University of Southern California
Los Angeles, USA
youyouli@usc.edu

***Abstract*—This report details the implementation and optimization of a ResNet-18 model for the CIFAR-100 dataset on the Kria KV260 FPGA board. The project is divided into two main parts: a software implementation using Google Colab for model training and a hardware implementation using Vitis HLS to generate RTL code and construct the ResNet-18 model layer by layer on the KV260. In the model training segment, we begin by using pretrained ImageNet weights, then apply fixed-point quantization to convert the weight format from float32 to int8, with settings of int=2 for weights and int=3 for input feature maps. This approach is focused on achieving high evaluation accuracy. For the hardware implementation, the report describes how we adapted the ResNet-18 architecture for FPGA deployment. This includes modifications to the data flow through the model, memory management on the chip, and optimization of the FPGA's computational resources. Additionally, we introduce features such as BRAM partitioning, custom-designed Processing Elements (PEs), and dynamic layer configuration. These enhancements are aimed at maintaining strong efficiency and performance in hardware execution.**

***Index Terms*—2D convolution, FPGA, Memory bandwidth, Hardware acceleration, Accelerator architecture, convolutional neural networks (CNNs), neural network hardware**

I. Introduction

This project explores the implementation and optimization of a ResNet-18 model on the CIFAR-100 dataset, applying quantization techniques to enhance computational efficiency suitable for deployment on the Kria KV260 FPGA board. The overall goal is to leverage the advanced capabilities of ResNet-18, initially tailored for the ImageNet dataset, and adapt it through quantization-aware training (QAT) to suit the CIFAR-100 dataset, which presents unique challenges due to its diverse set of 100 categories. The project not only emphasizes the importance of achieving high model accuracy but also states the need for efficient computational performance to facilitate deployment on resource-constrained FPGA platforms.

The project's phases include an initial training of the ResNet-18 model using pretrained ImageNet weights, followed by a customization phase where the model is adapted to meet the hardware constraints imposed by the FPGA implementation, specifically targeting efficient memory and compute utilization. This initial process involves resizing CIFAR-100 images to 224x224 pixels to match the input size typically expected by the ResNet-18 model, thus maintaining performance consistency. Subsequent quantization of the model involves applying fixed-point quantization to the weights and activations, with a particular focus on finding the most effective integer and fractional bit configurations through experimental evaluation.

The deployment phase on the FPGA requires careful consideration of on-chip memory limitations, data representation precision, and control flow optimizations to ensure that the FPGA implementation matches the software model's accuracy as closely as possible, with a acceptable deviation of up to 2%.

II. Key Components

A. ResNet18 Architecture

The ResNet18 architecture is a highly efficient variant of the ResNet (Residual Network) family, designed to tackle the problem of vanishing gradients in deep neural networks through the use of residual connections. This architecture consists of 18 layers which balances between depth and computational efficiency, making it suitable for a variety of tasks, particularly image classification. The key innovation of ResNet models is the introduction of skip connections, which allow the network to bypass certain layers and learn residual mappings. [1] This design improves gradient flow during training, enables faster convergence, and helps prevent overfitting.

ResNet18 has been widely used in both research and industry for a range of tasks, from image classification to feature extraction. Its relatively shallow architecture compared to deeper ResNet variants makes it more effective for scenarios where computational resources like memory or processing power are constrained. The network utilizes convolutional layers, followed by batch normalization and ReLU activation functions, to extract hierarchical features from input images. This structure allows ResNet18 to perform efficiently on datasets like CIFAR-100, where its ability to generalize well is essential for reaching high performance without excessive model complexity. [1]

B. CIFAR 100 Datasets

The CIFAR-100 dataset is one of the most widely used benchmarks in the field of computer vision, particularly for

evaluating image classification models. It contains 60,000 color images, each 32x32 pixels, across 100 distinct classes, with 50,000 training images and 10,000 test images. The classes span a wide range of categories, from animals and vehicles to everyday objects, making it a challenging dataset for machine learning models. Due to the relatively small size of the images, CIFAR-100 poses a unique challenge for deep learning models, requiring them to effectively learn features from low-resolution data while differentiating between a large number of categories. [2] This characteristic makes CIFAR-100 an excellent benchmark for assessing the generalization and performance of Convolutional Neural Networks (CNNs) like ResNet18.

For our project, we resize the images in the CIFAR-100 dataset from their original 32x32 resolution to 224x224 pixels. This resizing is necessary to align with the input requirements of ResNet18, which is designed to process images at this higher resolution. ResNet18, a well-established model in computer vision, performs better on higher-resolution inputs because it can extract more detailed hierarchical features through its deep architecture. By resizing the images, we can leverage ResNet18's capability to capture more nuanced patterns and improve performance while maintaining computational efficiency. This resizing step is crucial for adapting CIFAR-100 to the ResNet18 model, enabling it to perform effectively on a more challenging dataset while optimizing for both accuracy and resource usage.

C. Kria KV260

The Kria KV260 is an advanced FPGA development board specifically tailored for AI and edge computing applications. It is built around the Xilinx Zynq UltraScale+ MPSoC, which ingeniously blends a high-performance ARM-based processing system with programmable logic, making it a versatile choice for hardware acceleration. This board is equipped with 4GB of DDR4 RAM, providing substantial memory capacity for intensive processing tasks. The on-chip memory includes 648 KB of Block RAM and 2.304 MB of Ultra RAM, offering a total of 2.952 MB for efficient data management during operations. [3]

For memory-intensive applications, the Kria KV260 features 1,200 DSP slices which are essential for digital signal processing tasks, such as image processing and AI inference. The board also provides comprehensive support for PYNQ (Python Productivity for Zynq), a framework that facilitates the development of applications by allowing programmers to use Python to exploit the board's capabilities. PYNQ's interface simplifies interaction with the high-performance FPGA, enabling rapid deployment and prototyping of AI solutions. The integration of DDR4 memory, substantial on-chip memory, and extensive DSP resources makes the Kria KV260 exceptionally suitable for deploying complex models like ResNet18, especially in scenarios that demand real-time processing and high data throughput.

III. Prior Research

A. BRAM Partitioning

One technique in enhancing the efficiency of large-scale 2D convolution operations on FPGA platforms is the optimization of on-chip memory through BRAM (Block Random Access Memory) partitioning. This idea is drawn from the insights presented in a recent study, which proposed innovative strategies to manage on-chip memory more effectively. The research highlighted the limitations of traditional methods in handling memory requirements of advanced convolutional neural networks (CNNs) due to the growth in model complexity and size. Typically, FPGAs are constrained by their on-chip memory capacity, making it challenging to accelerate large-scale CNNs without substantial off-chip memory traffic, which severely downgrades the system's overall performance. To address these constraints, the study introduced a novel approach to BRAM partitioning, which significantly enhances the utilization of on-chip memory. By strategically organizing storage and access patterns of data within the FPGA's BRAM, the proposed method reduces the need for frequent data transfers between the on-chip and off-chip memory, thereby minimizing the bandwidth requirements and improving the speed of convolution operations. [4] This is particularly relevant to our project, where the stride variations in convolutional layers significantly affect computational efficiency. Specifically, in the ResNet18 model adapted for our project, four out of the nine convolutional layers feature a stride of 2, highlighting the importance of efficient resource usage. This insight drives the necessity to adopt and possibly further develop the proposed BRAM partitioning strategies to accommodate these needs, ensuring effective memory management and operational efficiency.

B. Loop optimization with enhanced on chip memories usage

The optimization of convolution operations in deep neural networks (DNNs) for FPGA implementations significantly focuses on the efficient management of loop operations. These loops, which are critical for performing the convolutional layers in neural networks, often represent a substantial portion of the computational workload in CNNs. Efficient loop management, including techniques such as loop unrolling, tiling, and interchange, is crucial for enhancing the performance and energy efficiency of FPGA-based hardware accelerators.

Loop optimization strategies are critical for convolution operations, which generally involve multiple nested loops across output features, input features, and spatial dimensions of the convolution kernel. By optimizing these loops through methods like loop unrolling to increase parallelism, and loop tiling to optimize data locality, the number of memory accesses required can be significantly reduced. This reduction is crucial for minimizing energy consumption and improving the throughput of hardware accelerators. Loop interchange, another optimization strategy, reorders the loops to further enhance data reuse and reduce costly data transfers that can limit performance. [4] In our FPGA-based implementation

of the ResNet18 architecture, Partial Loop Unrolling is used to enhance computational efficiency and throughput while managing resource constraints effectively. This specific type of loop unrolling is chosen to optimize performance without exceeding the FPGA's resource limitations, allowing us to maintain a balance between speed and hardware usage. Moreover, we also focus on advanced on-chip memory management techniques mentioned earlier, such as storing all input activations to URAM and partitioning BRAM. These memory optimization strategies are critical as they significantly reduce memory access delays and improve data availability, thus supporting the accelerated processing made possible by partial loop unrolling.

IV. Methodologies

This project combines both software and hardware implementations to accelerate the performance of Convolutional Neural Networks (CNNs), with a particular emphasis on optimizing large-scale 2D convolution operations. On the software side, we utilize PyTorch on Google Colab to develop and train the ResNet18 model, employing transfer learning to fine-tune pretrained weights for our specific task. To optimize the model for real-time performance, we implement techniques such as quantization, which reduces memory usage and improves efficiency, making it suitable for resource-constrained environments. On the hardware side, the model is deployed on an FPGA platform using Vitis HLS for hardware development, with on-board testing conducted on the Kria KV260 development board. Key hardware optimizations include efficient memory management through BRAM partitioning, PE design, pipelining, and loop unrolling. The integration of software and hardware components results in a high-performance solution, enabling the implementation of deep learning models in embedded systems and edge computing environments.

A. Software Implementation

The software implementation of this project leveraged PyTorch, a widely used deep learning framework, for model training, optimization, and quantization. The workflow was divided into three main stages: (1) training with a pretrained ResNet18 model, (2) adapting this pretrained model to a custom architecture, and (3) applying quantization techniques to optimize the model for inference on resource-constrained devices.

The software environment included the following.

- PyTorch: For deep learning model implementation, training, and optimization.
- TorchVision: For leveraging pretrained models and applying image transformations.
- CUDA: For accelerating model training using GPU when available.

The first stage began with using PyTorch and its TorchVision companion library, which provided access to pre-trained models such as ResNet18. These models were pre-trained on a large dataset like ImageNet, allowing us to take advantage of the learned features and apply them to our task with minimal fine-tuning. This approach is known as transfer learning technique. Additionally, CUDA was used for GPU acceleration during the training process to speed up computations.

The implementation also involved several Python libraries such as NumPy for numerical operations, Pandas for data handling, and Matplotlib for visualizations. These libraries facilitated both the training process and the analysis of the model performance.

In the following, we outline the key steps in our approach, starting with the pre-trained model, then moving to our custom model, and finally applying quantization for improved efficiency.

1) Torch pretrained ResNet18 model: The first step in model development was to utilize the ResNet18 model pretrained on ImageNet. ResNet18 is a deep convolutional neural network known for its residual connections, which help prevent the vanishing gradient problem during training. This pretrained model was used as the basis for our task of classifying images in the CIFAR-100 dataset, a dataset that contains 100 different classes of images.

The CIFAR-100 dataset was used for training, which consists of 60,000 images in 100 categories. To prepare the dataset for training with the ResNet18 model, images were resized to 224x224 pixels (the input size expected by ResNet18). Various image transformations were applied during training to augment the data and help the model generalize better. These transformations included random horizontal flips, random rotations, and color jittering, along with the normalization of pixel values to match the distribution of the ImageNet dataset.

Once the dataset was preprocessed, the pretrained ResNet18 model was fine-tuned by replacing its final fully connected layer to match the number of classes in CIFAR-100 (which is 100, compared to the 1,000 classes of ImageNet). The model's weights were initialized from the pretrained model, with the final classification layer trained from scratch.

2) Mapping to custom ResNet18 model: After training the pretrained ResNet18 model on the CIFAR-100 dataset, the next step was to adapt the model further to better suit our specific needs on hardware. This involved mapping the pretrained model to a custom ResNet18 architecture that filtered out some of the unused features.

In the standard ResNet18 model, there are several layers designed for large-scale image classification tasks, which may not be necessary for smaller, more specific datasets like CIFAR-100. As a result, we streamlined the model by removing or replacing certain layers that were not needed for the task at hand. For example, we replaced the fully connected layer at the end of the model to fit the 100-class output required by the CIFAR-100 dataset, and we fine-tuned the remaining layers to adapt them for better performance on the new dataset.

This mapping process helped optimize the pretrained model for the CIFAR-100 task while preserving the powerful features learned from ImageNet. Furthermore, certain layers from the pretrained model were frozen to prevent overfitting during fine-tuning. This allowed the model to retain the learned

features from ImageNet while still being able to adapt to the specific characteristics of the CIFAR-100 dataset.

3) Quantized ResNet18 model: In the final stages of optimizing the ResNet18 model for deployment on resource constrained devices, we applied fixed-point quantization to the weights, activations, and outputs of each operation within the ResNet18 model's architecture. This quantization technique converts floating-point representations into fixed-point numbers, which is particularly beneficial for embedded hardware system like FPGA that lacks native support for floating-point arithmetic. By reducing the model's computational complexity, fixed-point quantization enhances execution speed while lowering memory requirements.

Specifically, we employed 8 bit quantization for both weights and activations throughout the model. The choice of 8-bit precision strikes a balance between reducing memory usage and computational cost while retaining sufficient numerical accuracy for high performance. This level of precision is widely adopted in real-world deployment scenarios due to its compatibility with common hardware accelerators and its ability to maintain acceptable levels of accuracy.

A critical aspect of our quantization strategy involved the examination and optimization of integer and fractional bit widths for each quantized component. Rather than employing a uniform bit width across all components, we experimented with various combinations to identify the optimal configuration that maintained or even improved the overall train and validation accuracy. To ensure that our quantization approach was effective, we conducted evaluations of the model under various quantization settings. Our evaluations revealed that with the application of 8-bit quantization and careful selection of bit widths for the integer and fractional parts, the model could maintain an accuracy level close to that of the unquantized version.

4) Resource Requirements Calculation: Before proceeding to the design process of hardware implementation on Vitis and FPGA, we want to examine some key metrics of hardware resource requirements utilizing measurement tools from Python. This preliminary evaluation of the ResNet18 model provides essential insights into the computational demands and memory usage, which are crucial for determining the feasibility of deploying the model in environments where hardware resources may be limited like FPGA. Understanding these parameters helps in making informed decisions about the model's suitability for specific applications, particularly in constrained scenarios such as mobile devices or embedded systems.

B. Hardware Implementation

1) Implementing Kernel Function: In the kernel file kernel.cpp, we define 8 main function: "load input", "load weight", "PE", "store output", "batch norm", "skip connect", "controller", and the top function "conv kernel".

- load input: The load input function is structured to read activation data from an external memory source (act mem), handle boundary conditions (like padding), and write the data into a hardware stream (load input fifo). Each data element, after boundary checks and possible modification (padding), is written into the load input fifo. This stream is then used in the subsequent processing stages (like convolution operations in the PE)
  In detail, BUF2PE structure [4] was used to feed the inputs to PE with scalable input access. The BUF2PE structure reuses the input value accessed by the adjacent cells to minimize the input contention.
- load weight: load weight is tailored to read convolutional kernel weights from a binary file and push them into a hardware stream (load weight fifo). Each fetched weight is immediately written into the load weight fifo. This stream is crucial as it feeds the convolutional processing elements (PEs) directly, allowing for efficient pipeline execution of multiply-accumulate operations in the convolutional layers
- PE: For each spatial position, the function reads an activation from load input fifo and a weight from load weight fifo, multiplies them, and accumulates the result in mac vals. This process is repeated for each element in the kernel window and for all input and output feature maps. Once the complete convolution for a particular output pixel is computed, the accumulated result is written to the output stream (pe out fifo).
- store output: store output is designed to read processed data from an output stream and write it back to external memory.
- batch norm: The function begins by reading the normalization parameters (mean, variance multiplier, gamma, beta) from the bn weight mem, starting from the bn weight base addr. For each output feature map and its spatial dimensions, the function reads the output data from the in fifo stream. It then applies the normalization formula:

$$\text{mul factor} = 1/\sqrt{\epsilon + \text{var}} \times \gamma$$

$$\text{normalized output} = (\text{input} - \text{mean}) \times \text{mul factor} + \beta$$

  This formula adjusts each activation output to have a mean of zero and a variance of one, then scales and shifts them according to the learned parameters gamma and beta. After computing the normalized values, they are written to the out fifo stream.
- controller: The controller function is designed to dynamically configure parameters for different layers of the CNN. It adjusts settings such as the number of input and output features, kernel dimensions, stride, padding, and memory addresses, among others. Based on the current layer count (*layer cnt), the function sets all the parameters specific to that layer. These parameters control how the layer's computation is to be handled, including the dimensions of the data, memory addresses, and operational flags.
- conv kernel: The conv kernel function is essentially the main entry point for executing the convolutional operations. conv kernel coordinates the various stages of the convolutional neural network pipeline on the

FPGA, interfacing directly with memory (for both input and output), managing data flow through streams, and invoking other specialized functions such as load input, load weight, PE (Processing Element), batch norm, and store output.

2) Hardware Implementation Feature for accelerating:

- Controller for Dynamically Configure Parameter: The function can dynamically adjust settings in response to changes in the network's structure or in response to optimization strategies such as layer fusion or tiling. Enables or disables specific operations like convolution, batch normalization, activation functions, and pooling based on the needs of each layer, which helps in optimizing the computational efficiency and resource usage on the FPGA.
- BRAM partition for Weights data accessing: using HLS pragmas to guide the partitioning and to optimize the physical layout of the BRAM. By specifying the partitioning style, such as complete partitioning or block partitioning, you control how the compiler allocates the BRAM. Complete partitioning ensures that each array element can be accessed independently, potentially improving access speed at the cost of increased resource usage. In contrast, block partitioning groups multiple elements together, which can be more resource-efficient.
- Custom-Designed Processing Elements (PEs): Our PEs leverage on-chip memory to store intermediate results and activation data, significantly reducing the need for slower off-chip memory accesses. This approach minimizes data transfer overhead and power consumption, which are critical in high-performance embedded systems. Each PE is capable of conducting numerous multiply-accumulate (MAC) operations in parallel, significantly speeding up the overall computation time. This parallelism, integrated with efficient loop unrolling techniques, ensures that our design can handle the extensive computational demands of layers like the first convolutional layer of ResNet-18.

3) Workflow for Deploying Kernel Functions from Vitis HLS to PYNQ via Vivado: First, we prepare the trained weights: To load ResNet-18 weights in Python, we use the .pth checkpoint format. However, for hardware model access, the final trained weights must be saved layer by layer in .bin files. Additionally, we adapt the new Batch Normalization (BN) format by modifying the BN weight file. Originally containing 'running mean', 'running var', 'gamma (weight)', and 'beta (bias)', we transform these into 'running mean', 'mult factor', and 'beta', where 'mult factor' is derived from 'gamma / sqrt(running var + eps)'. For the PL ResNet-18 Model Implementation, we begin by implementing the Vitis HLS kernel using kernel.cpp, followed by a C simulation using host2.cpp, and then proceed with C synthesis and co-simulation. Subsequent steps include generating the IP by exporting RTL, transitioning to Vivado, loading the constructed kernel IP, completing the clock and PS-PL settings in the block design, and generating the .hwh, .bit, and .tcl files. The process continues in the PYNQ framework, where we deploy and execute the FPGA-based kernel using the host pynq.ipynb file. This file serves as a substitute for the original host.cpp used in Vitis HLS. We start by importing necessary libraries, loading the FPGA bitstream design 1.bit, and the trained weights in .bin files for each layer. Access to the FPGA IP core kernel 0 is established, and data buffers for inputs, weights, batch normalization parameters, and outputs are allocated and linked to hardware registers. Data is loaded from binary files into these buffers. The execution is initiated by setting control registers, and the process is monitored until completion. Finally, output data is retrieved for verification, ensuring the integrity and efficiency of our implementation.

V. Results

A. ResNet18 software training process

1) Parameter estimation and measurement: The ResNet18 model encompasses 11,689,512 total trainable parameters, indicating a significant level of complexity and also implying substantial memory requirements, both during the training phase and when the model is deployed for inference. Specifically, the model occupies approximately 11.70 MB of storage space. This aspect is critical when deploying on devices where storage is constrained, impacting decisions regarding hardware selection for both development and operational phases.

Further, the model demands 37.75 million multiply accumulate operations (MAC) per input processed. This measure of computational complexity is pivotal as it affects the processing speed and the energy consumption of the device. Such high computational requirements can be a limiting factor in applications where rapid processing and low power usage are essential, such as in mobile or portable devices.

Understanding these fundamental hardware requirements is important for subsequent optimizations and decisions regarding the model's deployment. For instance, employing techniques such as quantization, BRAM partitioning, or adopting more efficient processing element architectures might be necessary to reduce the computational load and memory footprint. These strategies help in managing the model's resource consumption, improving inference speed, and maintaining usability in power-sensitive environments.

This detailed evaluation of the ResNet18's hardware resource requirements ensures that the model aligns with the specific needs of the intended deployment scenario and informs further optimizations to meet stringent operational constraints and performance objectives.

2) Quantization with weight, input, and output: The quantization of neural networks involves the precise tuning of integer bit configurations for weights, inputs, and outputs, which significantly impacts model performance, particularly in terms of accuracy. Our experimental data provides valuable insights into the sensitivity of weight integer bits and their influence on the accuracy of the quantized model.

As shown in table I and table II, it is evident that the integer bits allocated for weights play a critical role in the model's ability to maintain accuracy post-quantization. Increasing the integer bits for weights from 2 to 3 generally resulted in a

TABLE I
ACCURACY FOR QUANTIZED RESNET18 MODEL UNDER DIFFERENT CONFIGURATION (WEIGHT INT BIT = 2)

Input Int Bit	Output Int Bit	Validation Accuracy (%)
2	2	69.35
2	3	74.07
2	4	74.13
3	2	69.30
3	3	74.59
3	4	74.54
4	2	68.03
4	3	73.62
4	4	73.76

noticeable decline in model accuracy across various configurations for input and output bits. For instance, with weight integer bits set to 2, configurations reached accuracies as high as 74.59% (Weight: 2, Input: 3, Output: 3). In contrast, when the weight integer bits were increased to 3, the highest recorded accuracy was only 53.07% (Weight: 3, Input: 2, Output: 4), demonstrating a clear decrease in performance.

TABLE II
ACCURACY FOR QUANTIZED RESNET18 MODEL UNDER DIFFERENT CONFIGURATION (WEIGHT INT BIT = 3)

Input Int Bit	Output Int Bit	Validation Accuracy (%)
2	2	46.47
2	3	51.48
2	4	53.07
3	2	46.71
3	3	51.05
3	4	52.40
4	2	45.03
4	3	49.96
4	4	50.81

Given these findings, for the subsequent stages of this project, we have chosen to proceed with the quantization configuration that yielded the highest accuracy. The selected combination of Weight: 2 bits, Input: 3 bits, Output: 3 bits achieves the highest accuracy of 74.59%. This configuration provides an optimal balance between maintaining high accuracy and achieving efficient computation, making it the preferred choice for deploying the quantized model in resource-constrained environments.

3) Quantization with weight and input: In our continued efforts to optimize the quantization parameters for the ResNet18 model, we conducted further experiments specifically focusing on varying the integer bits for weights and inputs shown in table III. The experiments aimed to identify the configurations that best balance accuracy and computational efficiency. The results revealed a clear trend regarding the sensitivity of model accuracy to changes in weight and input bit configurations. When the weight integer bit was set to 2, we observed relatively high validation accuracies: 75.42% with an input integer bit of 2, peaking at 75.76% with an input integer bit of 3, and slightly lower at 75.11% with an input integer bit of 4. However, increasing the weight integer bit to 3 resulted in a significant drop in accuracy, demonstrating validation accuracies of 53.05% with an input bit of 2, 52.38% with an input bit of 3, and 51.88% with an input bit of 4. The trend became more pronounced when the weight integer bit was increased to 4, where the validation accuracies drastically decreased to 1.77%, 1.49%, and 1.11% for input bits of 2, 3, and 4, respectively. These findings underscore the critical impact of integer bit settings on quantization, indicating that optimal settings are crucial for maintaining high accuracy without compromising the model's computational demands. The experiment suggests that a weight integer bit of 2 and an input integer bit of 3 provide the best trade-off between accuracy and efficiency, making this configuration ideal for implementations in resource-constrained environments.

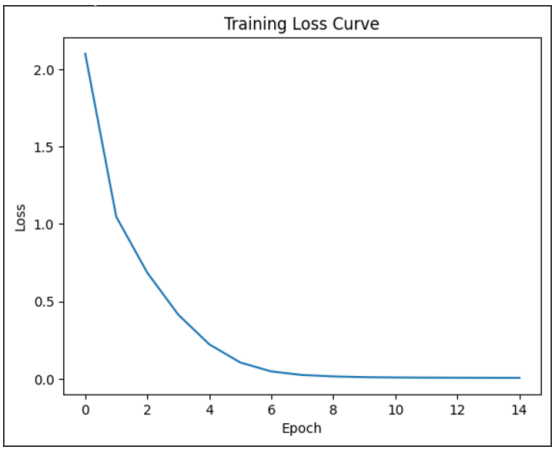

Fig. 1. Learning Curve with Batch Size = 16, learning rate = 0.025, number of Epoch = 15

TABLE III
ACCURACY FOR QUANTIZED WEIGHT AND INPUT OF RESNET18 MODEL

Weight Int Bit	Input Int Bit	Validation Accuracy (%)
2	2	75.42
2	3	75.76
2	4	75.11
3	2	53.05
3	3	52.38
3	4	51.88
4	2	1.77
4	3	1.49
4	4	1.11

4) Training performance analysis: Our experimental approach involved a multi-stage process to train and optimize a convolutional neural network using the ResNet18 architecture, designed to maximize accuracy on the CIFAR-100 dataset. Results are shown in table IV.

Initial Training with Pretrained Model: We started by utilizing a pretrained ResNet18 model, originally trained on the extensive ImageNet dataset. This provided a robust foundation of learned features beneficial for complex image classification tasks. After fine-tuning this model on the CIFAR-100 dataset for 15 epochs, we achieved a training loss of 0.0508, a

TABLE IV
ACCURACY PERFORMANCE OF RESNET18 UNDER DIFFERENT TRAINING STAGES

Stage name	Pretrained ResNet18	Customized ResNet18	Quantized ResNet18
Epochs	15	11	6
Train Loss	0.0508	0.1875	0.3529
Train Accuracy	98.91	95.89	93.49
Valid Accuracy	78.22	77.45	76.76

training accuracy of 98.91%, and a validation accuracy of 78.22%. These results demonstrated substantial adaptation to the CIFAR-100 dataset's varied classes. Customization for Hardware Implementation: Following the initial fine-tuning, we adapted the ResNet18 model to our custom specifications designed primarily to facilitate ease of hardware implementation on the Vitis platform. This step involved mapping the model to ensure that the software layers precisely matched the corresponding hardware layers, which is crucial for seamless deployment on FPGA. This customization ensures that the model's architecture would be compatible with FPGA design and simplifies the integration process on the hardware. After implementing these modifications, the model was further trained for 11 epochs, resulting in a training loss of 0.1875, a training accuracy of 95.89%, and a validation accuracy of 77.45%. Quantization for Efficiency: The final phase of our experiment involved quantizing the fully trained ResNet18 model to optimize it for deployment in resource-constrained environments. By selecting the most optimal combination of integer and fractional bits for quantization, we aimed to maintain high accuracy while significantly reducing the model's memory footprint and computational demands. After the quantization process over six epochs, the model recorded a training loss of 0.3529, a training accuracy of 93.49%, and a validation accuracy of 76.76%. These figures confirm that while there is a slight decrease in performance due to quantization, the impact is minimized throughout the process.

B. ResNet18 hardware implementation

1) Data packing and partitioning: On-chip memory partitioning was used for parallel activation data access. For the implementation of ResNet-18, the largest activation matrix produced by the first convolution layer contains 802,816 elements, while the second largest, from the max pooling layer, has 200,704 elements. Including an additional 200,704 elements for the skip connection allows all activation data to remain on-chip, thereby minimizing latency and reducing the need for frequent off-chip memory access.

Data packing is needed to fully utilize the on-chip memory. Without packing, a single 8-bit data is mapped to a single location in URAM that can hold up to 72bit data. Therefore, our design uses 8bit activation packed to 64bits to store the on-chip memory in URAM. Fig. 2 and fig. 3 shows the synthesis results without and with packing. Without packing, the total URAM needed exeeds the available URAM by far. On the contrary, the data packing allows the needed activation to be on-chip. For efficient wiring, we implement techniques for weight reuse and efficient fetching by BUF2PE structure [4]. Meanwhile, ResNet-18's parameters are 11.7 million in total, which means not all weights can be stored on-chip. To address this, weights are fetched from off-chip memory as needed and saved to BRAM. Specifically, we loop the output filter index in the outermost loop, which aligns with PyTorch's data organization. This indexing structure also facilitates partitioning and parallel access aligned with powers of 2, as the minimum filter size is 64 in ResNet-18 and doubles across layers. This allows flexibility to choose the parallel computation factor across output filter dimension. To utilize this, we partition the BRAM for partial off-chip weights by 64.

Pipelined	BRAM	DSP	FF	LUT	URAM
no	2	0	468	1776	221
no	-	-	-	-	-
no	-	-	-	-	-
no	-	-	-	-	-

Fig. 2. Partial utilization without data packing

Pipelined	BRAM	DSP	FF	LUT	URAM
no	5	0	431	1863	38
no	-	-	-	-	-
no	-	-	-	-	-
no	-	-	-	-	-

Fig. 3. Partial utilization with data packing

2) Loop structure for convolution: To explain the loop structure, we use the terms in table. VI [4]. The RAM partitioning allows straightforward parallel computation on POF dimension. This can be explored up to the amount of BRAM partitioning, which is 64. The BUF2PE design implemented in our work allows for parallelism across the output filter size, and the output dimensions in both Y and X directions. The Y and X direction parallelization is denoted as POY and POX and both are 7 to align with the height and width of activations. The detailed loop implementation is shown in the pseudo-code.

```
for f_out in range(0, nof, POF):
  for y0 in range(0, noy, POY):
    for x0 in range(0, nox, POX):
      // Initialize MAC values
      for all f, y, x in POF, POY, POX
        mac_vals[f][y][x] = 0

      // Accumulate MAC values
      for f_in in range(0, nif):
        for f, y, x, i, j in POF, POY, POX, nky, nkx:
          mac_vals[f][y][x] += activation * weight

      // Write results to output FIFO
      for f, y, x in POF, POY, POX:
        pe_out_fifo.write(mac_vals[f][y][x])
```

3) Design verification and overall progress: The hardware implementation is verified through three steps. First, C simulation in Vitis HLS is conducted to check for functionality, using smaller channel sizes to minimize the memory space required for simulation. In this step, the hardware is designed and verified with full floating-point inputs, weights, and outputs. In the second step, the hardware is tested after being deployed on an FPGA. Floating-point values are still used, but it is now tested with a full-sized RESNET18 model and the QAT-trained software values. Finally, in the third step, the model is evaluated using a full-sized RESNET18 QAT model with 8-bit quantized activations and weights. The step involving the small model with quantized activations and weights is omitted due to excessive divergence in results caused by the random inputs and weights used in the reduced model.

The initial step achieved a Root Mean Squared Error (RMSE) of less than 1e-6 for all layers, except for the last fully connected (fc) layer, which had an RMSE of 0.0247. The results of the subsequent steps have not yet been verified.

VI. Conclusion

This project demonstrated the implementation and optimization of the ResNet-18 model for the CIFAR-100 dataset on the Kria KV260 FPGA board, effectively bridging advanced deep learning techniques with FPGA-based hardware acceleration. Through meticulous training using pretrained ImageNet weights and subsequent application of fixed-point quantization, we tailored the model to efficiently operate within the constraints of FPGA hardware. The quantization was specifically aligned with the FPGA's MAC design, aming to substantially reduce the model's memory and computational demands without significant loss in accuracy, achieving robust evaluation metrics under constrained conditions.

Our results highlighted the efficacy of the model post-quantization, where the adapted ResNet-18 achieved high accuracy, demonstrating the potential of quantized models to maintain performance even on resource-restricted platforms. Specifically, the model's performance in terms of accuracy closely approached that of its non-quantized counterpart, underlining the success of our optimization strategies.

The hardware implementation utilized Vitis HLS for generating RTL code, which effectively mapped the neural network onto the FPGA. Strategies such as BRAM partitioning, custom-designed Processing Elements (PEs), and dynamic layer configuration were critical in optimizing data flow and memory management, thus maximizing the computational efficiency of the FPGA resources. These enhancements not only facilitated powerful, efficient computation but also underscored the scalability of FPGA systems in deploying complex AI models.

This project underlines the substantial capabilities of FPGA-based systems in fostering efficient, powerful, and scalable AI solutions, paving the way for broader adoption in edge computing and IoT devices where performance, power efficiency, and cost are paramount.

References

[1] R. Modi, "Resnet: Understand and implement from scratch." https://medium.com/analytics-vidhya/resnet-understand-and-implement-from-scratch-d0eb9725e0db, 2020. Accessed: 2024-12-19.

[2] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., University of Toronto, 2009. Technical Report.

[3] X. Inc., "Kria kv260 vision ai starter kit datasheet." https://www.mouser.com/datasheet/2/903/ds986_kv260_starter_kit-2301120.pdf, 2023. Accessed: 2024-12-19.

[4] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1354–1367, 2018.

Table IV: Convolution Loop Dimensions and Hardware Design Variables

	Kernel Window (width/height)	**Input Feature Map (width/height)**	**Output Feature Map (width/height)**	**# of Input Feature Maps**	**# of Output Feature Maps**
Convolution Loops	Loop-1	Loop-3	Loop-3	Loop-2	Loop-4
Convolution Dimensions (N*)	N_{kx}, N_{ky}	N_{ix}, N_{iy}	N_{ox}, N_{oy}	N_{if}	N_{of}
Loop Tiling (T*)	T_{kx}, T_{ky}	T_{ix}, T_{iy}	T_{ox}, T_{oy}	T_{if}	T_{of}
Loop Unrolling (P*)	P_{kx}, P_{ky}	P_{ix}, P_{iy}	P_{ox}, P_{oy}	P_{if}	P_{of}