FPGA Based Convolution Accelerator For Image Recognition

Govarshini R B
M.E. VLSI Design (Pursing), Department of Electronics and Communication Engineering, College of Engineering, Guindy, AnnaUniversity, Chennai, India

Ewins Pon Pushpa S
Assistant Professor, Department of Electronics and Communication Engineering, College of Engineering, Guindy, AnnaUniversity, Chennai, India

Abstract:

The use of machine learning based applications has become widespread in today's era. Many CNN based implementation require high computational power and consumes time. CNNs are the base for any machine learning or neural network process. They are developed primarily using GPUs (Graphical Processing Units) that require very high-power consumption and are also expensive. The process is either sequential or pipelined and hence requires more computation time. With the usage of FPGAs, where the process is parallelized increases the efficiency of the process, thus reduces the power consumption compared to that of GPUs. Image recognition is the most commonly used machine learning application. Any machine learning application would involve an image recognition step. Hence developing of optimal energy consuming and low-cost models for image recognition is the need of the moment. FPGAs reconfigurability and its hardware parallelism provides flexibility and high performance with less processing power. These advantages of FPGAs can be utilized to generate an energy efficient and low-cost model of a CNN accelerator for image recognition. This helps to deploy the CNN models in low-end and battery-operated devices.

Keywords: *CNN, field-programmable gate arrays, DNN, Depthwise separable convolution.*

Introduction

Convolutional Neural Networks (CNNs) have revolutionized deep learning, serving as a foundation for numerous applications involving visual data, including image recognition, object detection, and video analysis. Their architecture, inspired from animal visual cortex, is designed to efficiently capture spatial hierarchies in images through a series of layered processing stages.

CNNs are widely used for processing visual data. They excel in tasks like image and video recognition and classification, medical image processing, and natural language processing. CNNs have different layers, including convolutional, pooling, and fully connected layers, which interact to extract and learn features from input images.

Initially, CNN algorithms were executed on CPUs (Central Processing Units). However, as the demand for more computational power grew, GPUs (Graphics Processing Units) became the preferred choice due to their parallel processing capabilities. GPUs enabled significant speedups in training and inference of CNNs by allowing many operations to be performed simultaneously.

Despite the advantages of GPUs, their high-power consumption and cost have spurred interest in alternative platforms, such as Field-Programmable Gate Arrays (FPGAs) that are designed to be customized by users or developers after production, earning the name "field-programmable." These devices feature an array of programmable logic blocks and a structured network of reconfigurable interconnections, allowing them to be configured for specific computational tasks.

 https://doi.org/10.5281/zenodo.15302810

While GPUs have played a significant role in the development of CNN technology, FPGAs present a viable substitute that combines flexibility, energy efficiency, and high performance. This change is expanding the effect of deep learning in daily life by creating new opportunities for the deployment of CNNs in a broader range of applications, from industrial automation to consumer electronics.

This paper proposes a methodology to help in deploying the CNN which has a DSC layer into a FPGA Board using tools and framework existing which can make them more easily applicable to low power devices

Related Work

The rapid progress in deep learning has been driven by enhanced computing power, yet deep neural networks (DNNs) remain resource-intensive in terms of computation and memory. To address this, energy-efficient DNN accelerators have been developed for GPUs, ASICs, and FPGAs. While GPU have high performance but are power consuming, ASICs are energy-efficient but do not have flexibility.[1]

FPGAs have features like performance, reconfigurability, and cost-effectiveness. Image recognition tasks mostly use CNNs and ResNet models, but their size poses challenges for deployment on low-end devices. MobileNetV2 tackles this issue with Depthwise Separable Convolution (DSC) layers, reducing both model size and computational complexity. An FPGA-based optimal energy accelerator for DSC layers optimizes power and memory usage by reducing multiplications and leveraging LUTs instead of DSPs, making it suitable for mobile devices.[5]

Utilizing binary weights and activations ranging from 3 to 6 bits helps the model to maintain high accuracy while drastically lowering its complexity and size. This reduces the dependency on DSPs by simplifying convolutional operations to basic addition. Additionally, it allows the entire model to fit in the chip, minimizing the use of slower external memory and enhancing it.[2]

MobileNetV2 is a neural network architecture specifically designed to enhance performance on portable embedded devices. It incorporates inverted residual blocks, which first expand the input, apply lightweight Depthwise convolutions, and then compress the output back to a lower dimension, effectively capturing intricate features. The architecture also utilizes linear bottlenecks to maintain a smooth information flow, avoiding the performance drops caused by non-linearities. This design achieves high accuracy while significantly reducing computational demands and memory consumption, making it well-suited for on-the-go applications on mobile devices like smartphones and IoT systems. Its balance of efficiency and performance makes it widely applicable to tasks such as image classification, object detection, and beyond.[6]

Implementing a pipelined, model-specific architecture enhances resource utilization and is further optimized using task buffers and cache memories. Implementing a pipelined, model-

 https://doi.org/10.5281/zenodo.15302810

-specific architecture enhances resource utilization and is further optimized using task buffers and cache memories. Data flow blocks are used to organize and streamline the movement of data through the pipeline. These blocks operate on data streams and perform various functions, including: (i)dividing or duplicating data, (ii) concatenating or merging data, (iii) adding data, (iv) reordering data, (v) repeating data, (vi) adjusting the number of hardware channels involves utilizing buffers or internal memory within the data flow blocks to perform operations while minimizing dependency on external memory bandwidth or DSP blocks. [3]

Various software libraries and frameworks include Caffe2, PyTorch, TensorFlow, MXNet, CoreML, CNTK, TensorRT, that help CNN developers speedily achieve their goals by providing them competent APIs and optimizing model execution on CPUs, GPUs, another older generation of DSPs and specialized ASICs. This is in addition to tools such as Xilinx Vivado HLS, Intel FPGA OpenCL SDK, Maxeler MaxCompiler, and LegUp, providing simplified hardware design with C, C++, OpenCL, and Java. OpenCL and Java to support the creation of functionally accurate hardware designs. [7]

FINN framework, developed by AMD Research and Advanced Development (RAD), is an open-source initiative aimed at deploying DNNs on FPGAs efficiently. It specializes in QNNs, which utilize low-bit-width weights and activations to minimize memory usage and enhance computational efficiency. FINN generates custom, dataflow-style architectures for each network, emphasizing high throughput and low latency.[9]

Deep Neural Networks (DNNs) have several intermediate layers between input channel and output channel. Each layer is made up of neurons (nodes) that process the input data through various computations. DNNs are built to address complex problems by learning patterns from extensive datasets. Techniques such as overlapping pooling, dropout to mitigate overfitting, and data augmentation methods like image translations and reflections have enhanced the capacity of the model to generalize hidden data.[4]

FFT-based convolution provides substantial computational benefits, it also comes with notable drawbacks. Implementing FFT-based convolution involves intricate mathematical transformations that can be difficult to optimize for embedded hardware. Additionally, FFT-based approaches typically demand more memory than direct convolution, particularly with large input sizes. Although FFT can accelerate large-scale convolutions, it may introduce extra latency due to the overhead of executing both the FFT and its inverse (IFFT).[7]

Proposed Methodology

This section describes the concepts utilized and the methodological flow of this approach.

A. Standard 2D Convolution:

In standard 2D convolution, filters slide over the input image to extract spatial features across all channels, generating feature maps with spatial and channel-wise information. While effective, this method is computationally intensive and memory-heavy due to the huge no. of parameters, especially as filters, channels increase. This leads to longer training and inference times, higher energy consumption, and significant memory demands, posing challenges for resource-constrained devices like mobile and embedded systems. Fig. 1 shows the spatial representation of a standard 2D convolution.

 https://doi.org/10.5281/zenodo.15302810

ISSN: 0369-8963

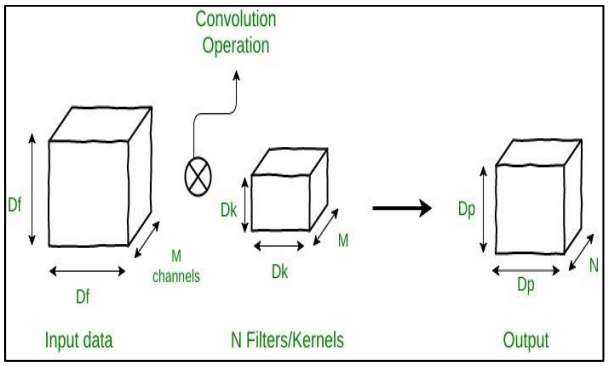

Fig. 1. Standard 2D Convolution [10]

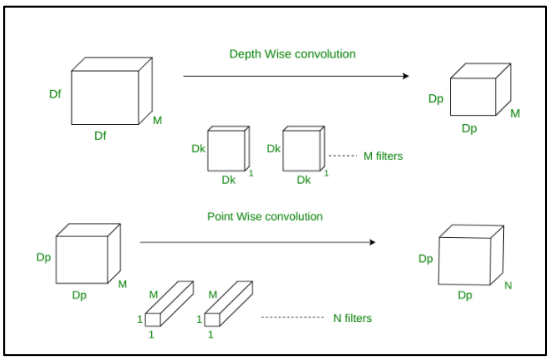

Fig. 2. Depthwise Separable Convolution [10]

$$Size\ of\ filter = Dk \times Dk \times M \quad (1)$$

$$Total\ no\ of\ multiplications = N \times Dp^2 \times Dk^2 \times M \quad (2)$$

Where, N – No of Filters

Dp – Dimension of the output data

Dk – Dimension of the input data

M – No of input channels

B. Depthwise Separable Convolution:

Depthwise convolution, a more efficient variant, tackles this issue by splitting the process into Depthwise and pointwise convolution. In Depthwise step, each filter is applied independently to a single channel of the input, rather than spanning all channels. This means that if there are M input channels, M separate Depthwise convolutions are performed as represented in Fig. 2. Following this, the pointwise convolution (PWC) step combines the outputs from the Depthwise convolution (DWC) using 1x1 convolutions. This step aggregates the spatially filtered features across the channels, effectively reduces the complexity of the computation and the no. of multiplications as given in (3) – (5).

$$No\ of\ multiplications\ for\ DWC = M \times Dk2 \times Dp2 \quad (3)$$

$$No\ of\ multiplications\ for\ PWC = M \times Dp2 \times N \quad (4)$$

$$Total\ no.\ of\ multiplications = M \times Dp2 \times (Dk2 + N) \quad (5)$$

Where, N – No of Filters

Dp – Dimension of the output data

Dk – Dimension of the input data

M – No of input channels

C. Mobilenet Architecture

MobileNet is a family of very efficient CNN models specially designed for mobile and embedded devices. These models provide very good accuracy with low computational and memory power requirements. Depthwise Separable Convolutions achieve parameter and computation reduction. The key hyperparameters are the width multiplier and resolution multiplier, which control the number of channels and input size to find the balance among accuracy, model size, and computational cost involved.

D. Bottleneck Residual Bock:

Bottleneck Resididual Block performs the inverted residual structure. Herein, the first step includes reducing an input's dimensionality with a 1x1 convolution, then being subjected to a light Depthwise convolution (generally 3x3), finally followed by projection back to the input dimension using yet another 1x1 convolution.

 https://doi.org/10.5281/zenodo.15302810

E. Mechanics of Inverted Residual Block:

- Bottleneck Compression: The block starts with a 1x1convolution, known as the bottleneck layer, that decreases the number of input channels. This step compresses the input, reducing the computational load for the subsequent operations as represented in Fig. 3.
- Depthwise Convolution: The compressed representation is then processed using a Depthwise convolution, which incorporates a different filter to each input channel independently, unlike standard convolutions that operate across all channels. This approach significantly decreases both the amount of parameters and computation complexity.
- Expansion: Finally, a 1x1 convolution block is implemented to restore the original no. of channels, enabling that model to capture more complex representations.

Table 1. MobileNet Architecture

Layer Type	Expansion Factor	Output Channels	Repeats	Stride
Conv2D		32	1	2
Bottleneck	1	16	1	1
Bottleneck	6	24	2	2
Bottleneck	6	32	3	2
Bottleneck	6	64	4	2
Bottleneck	6	96	3	1
Bottleneck	6	160	3	2
Bottleneck	6	320	1	1
Conv2D (1X1)		1280	1	1
Avg Pool (7X7)			1	
Fully Connected		1000 (task - specific)	1	

- Residual Connection: A residual connection (or shortcut connection) is maintained between the input channel and the output channel. Which helps in preserving the original information and ensures that gradients flow smoothly during backpropagation, making the network easier to train.

Table 1. describes the entire MobileNet architecture with all the bottleneck layers and the input and output features.

F. ReLU Non-Linearity:

ReLU6 is used by MobileNetV2 in place of the conventional ReLU activation mechanism, capping the activation to 6. This change aids in increasing training stability, particularly with less precise equipment.

G. Methodology Flow

The flowchart that is shown in Fig. 4. outlines the process of implementing and deploying a Convolutional Neural Network (CNN) using the FINN framework for hardware acceleration. The CNN model was initially developed and trained using the PyTorch framework. The CNN model is quantized to a QNN using Brevitas. Quantization reduces the precision of weights and activations (e.g., to low-bit integers like INT8 or INT4).

 https://doi.org/10.5281/zenodo.15302810

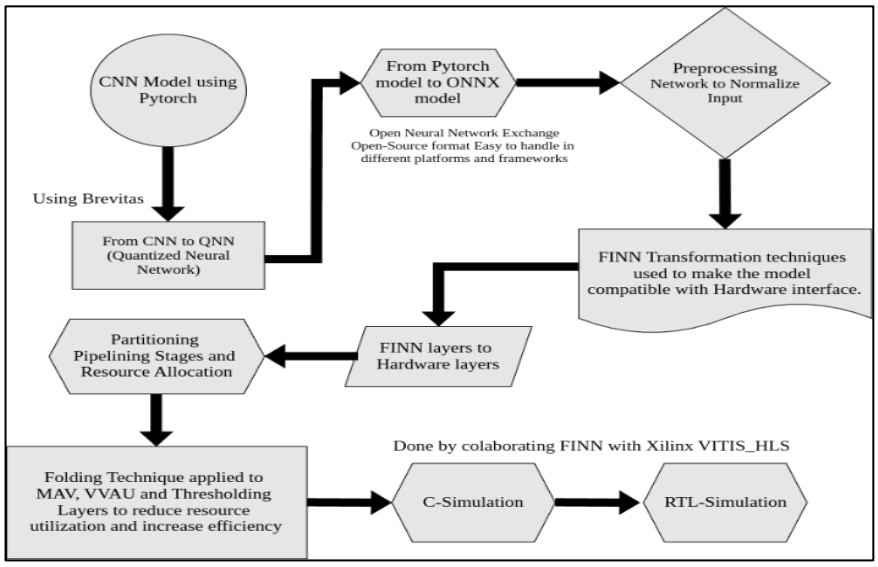

Fig. 4. Methodology Flow Chart

This significantly lowers memory usage, reduces computational complexity, and minimizes power consumption, making the model more efficient for hardware acceleration. By incorporating QAT, Brevitas ensures the network adapts to the non-linearity and rounding effects introduced during quantization, maintaining high accuracy even with reduced precision. From PyTorch model to ONNX model. The pre-trained PyTorch model is transformed into ONNX format. ONNX provides an intermediate representation that enables compatibility with various frameworks and tools, including FINN. Before deploying the model, input data preprocessing is implemented to normalize inputs. This step ensures that the model performs optimally when handling real-world input data.

The QNN undergoes FINN-specific transformations to make it compatible with hardware interfaces. These transformations may include simplifying computational graphs, Modifying layers to optimize resource utilization for FPGA, etc. The model is divided into stages to support pipelining, allowing different parts of the computation to execute concurrently. Resource Allocation ensures efficient distribution of hardware resources (e.g., logic blocks, memory).The QNN layers are mapped to corresponding hardware layers, such as MVAU, VVAU, and thresholding layers.

Folding Technique is applied to MVAU, VVAU, and thresholding layers to reduce hardware resource utilization and improve efficiency by sharing resources across computations. The model undergoes C-Simulation to validate its functionality and performance at a software level. This step simulates the hardware behavior using a C-based model. RTL Simulation is performed to ensure that the synthesized hardware meets the design requirements. It provides a bit-accurate representation of the hardware implementation.

Results and Discussion

A. Quantized Neural Network:

A Quantized Neural Network (QNN) for MobileNet using the PyTorch library Brevitas was modeled. Quantized MobileNet architecture is developed by replacing standard layers

 https://doi.org/10.5281/zenodo.15302810

with Brevitas quantized layers, such as QuantConv2d and QuantReLU. This approach helps create efficient models suited for resource-constrained environments, maintaining good performance while reducing computational and memory requirements. Parts of the quantized neural network is shown in Fig. 5.

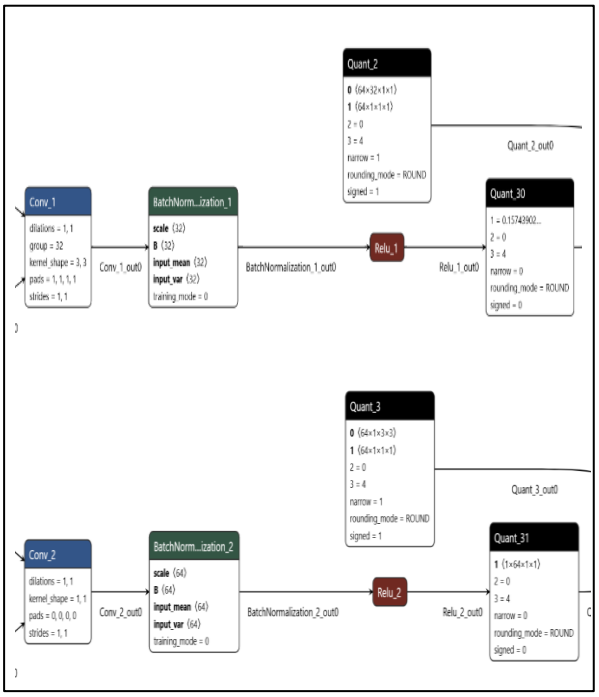

Fig. 5. Mobilenet Using QNN

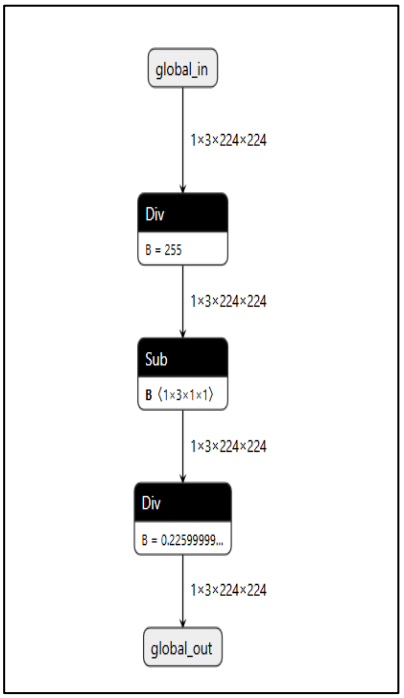

Fig. 6. Pre-Processing Unit

B. Pre-Processing Unit:

Preprocessing unit shown in Fig. 6 is added for the below key reasons: Normalization to make sure to have similar scale for all input features for numerical stability, Data Augmentation to improve model generalization and prevent overfitting, Feature Extraction to make critical information more accessible, and Input Standardization to maintain consistency across different data sources. These preprocessing steps collectively contribute to achieving higher accuracy, better generalization, and more robust performance in CNN models.

C. Streamlining Transformations:

The streamlining transformations used from FINN are designed to optimize neural network models by eliminating floating-point operations and simplifying the network structure. Some of the transformations used: MoveMulPastDWConv, AbsorbMulIntoMultiThreshold, MoveTransposePastScalarMul, AbsorbTransposeIntoFlatten, MoveScalarMulPastMatMul, CollapseRepeatedMul, RoundAndClipThresholds. These transformations as shown in Fig. 7 help in creating more efficient and compact neural network models.

 https://doi.org/10.5281/zenodo.15302810

ISSN: 0369-8963

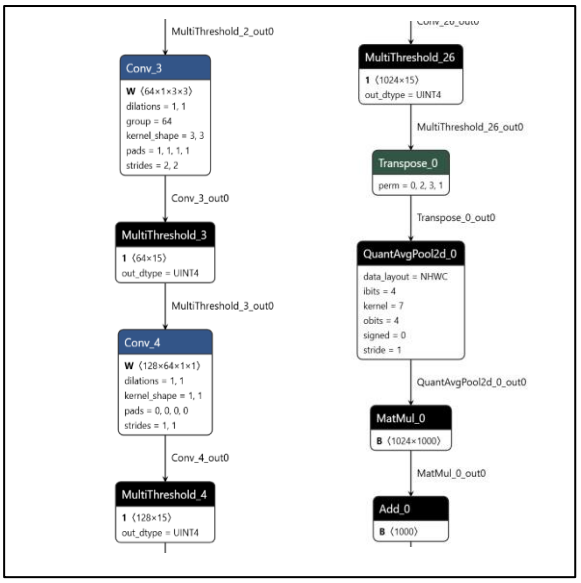

Fig. 7. Mobilenet model after Streamlining Transformations

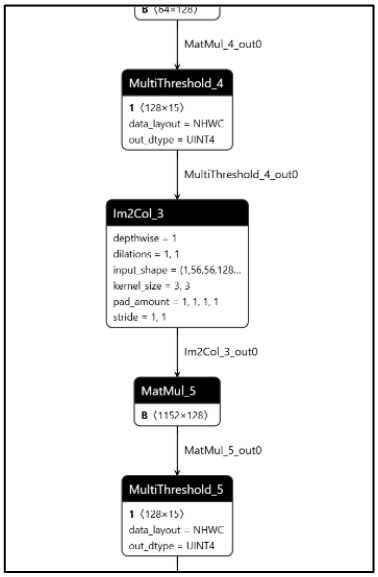

Fig. 8. IM2COL Layer

D. IM2COL Layer:

Convolution layers are then transformed to IM2COL layers which perform the same convolution operation in an optimized manner as shown in Fig. 8. The Im2Col operation is used in optimizing convolution operations, especially for hardware implementations like FPGAs. It transforms the input image into a column matrix, making it easier to perform matrix multiplications, which are more efficient on hardware. Transformation Process of the IM2COL layer. Extract Patches: The Im2Col operation extracts overlapping patches from the input image based on the kernel size, stride, and padding. Flatten Patches: Each patch is flattened into a column. Concatenate Columns: All column vectors are concatenated to form a large matrix, where each column represents a patch from the input image.

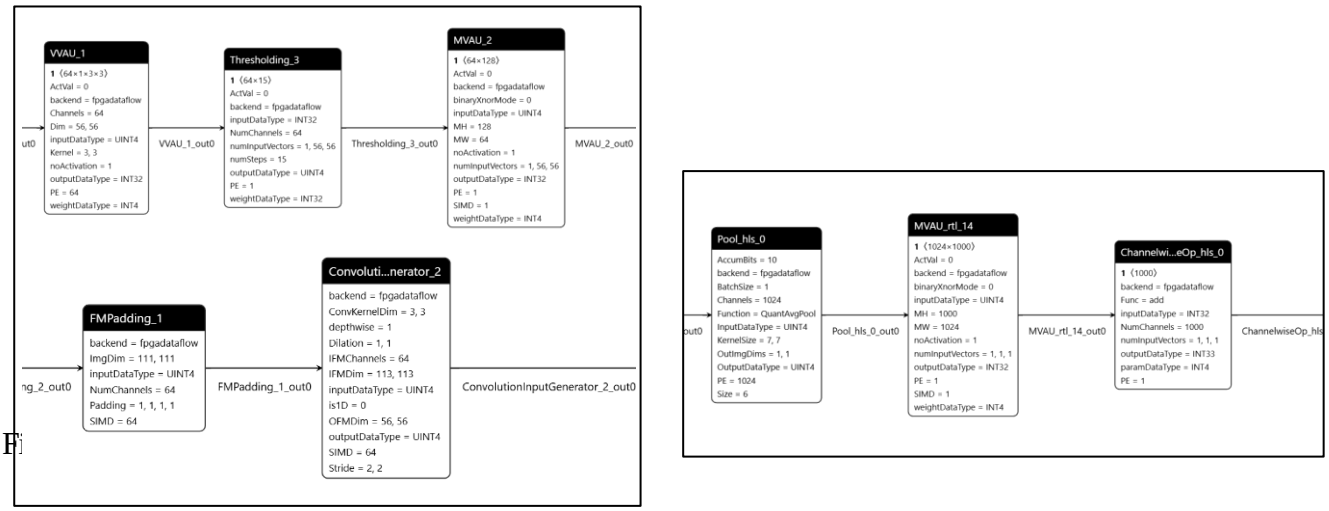

Fig. 9. Hardware Layers

https://doi.org/10.5281/zenodo.15302810

E. Hardware Layers:

The QNN layers are then transformed to their corresponding hardware layers to make them compatible for C-simulation and RTL-simulation as represented in Fig. 9.

- Convolution Input Generator: Identified convolution layers are replaced with the appropriate convolution input generator nodes, which are optimized for FPGA implementation.
- Thresholding: Transformation in FINN is used to convert multi-threshold layers into hardware-friendly thresholding layers.
- Vector-Vector Activation Unit (VVAU): Designed to convert matrix multiplication (MatMul) layers with quantized inputs and weights into VVAU layers. Useful when the sparsity annotation of the weight matrix indicates that the MatMul layer belongs to a depth wise convolution.
- Matrix-Vector Activation Unit(MVAU): Designed to convert matrix multiplication (MatMul) layers with quantized inputs and weights into VVAU layers. Here it is used to convert the pointwise convolution.
- FM padding layer (Fixed-Point Multiplier padding layer): Design to help in efficiently handling the padding required for convolution operations by performing fixed-point multiplications, which are more hardware-friendly and faster compared to floating-point operations.

F. Specialize Layers:

This process involves converting standard or custom layers into hardware (HW) abstraction layers, which are then further specialized into either Register-Transfer Level (RTL) or High-Level Synthesis (HLS) variants.

- Dataflow Partitioning: The network graph is split into hardware and non-hardware parts. The hardware part is further processed, while the non-hardware part remains for additional transformations.
- Convert to HW Layers: Standard or custom layers are converted to hardware abstraction layers, which are placeholders that can be implemented in HLS or RTL as shown in Fig. 10.
- Specialize Layers: The network is converted to hardware abstraction layers, and non-hardware layers are excluded to continue processing the model to make sure that the model is optimized for hardware implementation.

G. Optimization:

Optimization techniques are used to reduce the throughput and resource utilization of the model. The techniques used are listed below.

- Folding: This technique involves combining multiple operations into fewer steps, which reduces the computational load and resource usage. It is particularly useful for FPGA implementations where hardware resources are limited.
- Processing Elements (PE): These are the basic units of computation in an FPGA. By adjusting the number of PEs, the function optimizes the parallelism and resource usage of the model.
- SIMD (Single Instruction, Multiple Data): This concept allows a single instruction to be executed on multiple data points simultaneously, setting SIMD allows us to set the degree of parallelism a layer should use.
- RAM Style: Type of RAM style such as block RAM, or distributed RAM is determined based on the layers used.

 https://doi.org/10.5281/zenodo.15302810

ISSN: 0369-8963

H. Resource Utilization ,Timing Distribution and Simulation Report:

The graph in Fig. 11. illustrates the number of LUTs required for each layer of a neural network, with the settings PE=1 and SIMD=1.

Some layers (e.g., T5, T3) have significantly higher LUT usage, likely due to more computationally intensive operations such as large convolutional kernels or fully connected layers. The variations in LUT consumption reflects differences in layer configurations (e.g., number of channels, kernel sizes) or optimizations. Layers like C0 and M7 show minimal LUT usage, since they use simpler operations and lower computational complexity.

The graphs show the clock cycles required for each layer before and after optimization. Before performing optimization as shown in Fig. 12. techniques, the throughput is very large for the MAVU layers making them very complex and time consuming and after optimization in Fig. 13. the load of the MAVU layers is reduced the overall clock cycles usage is distributed evenly. This is done by applying the folding technique and specifying the RAM style of the different layers based on the type of the layer.

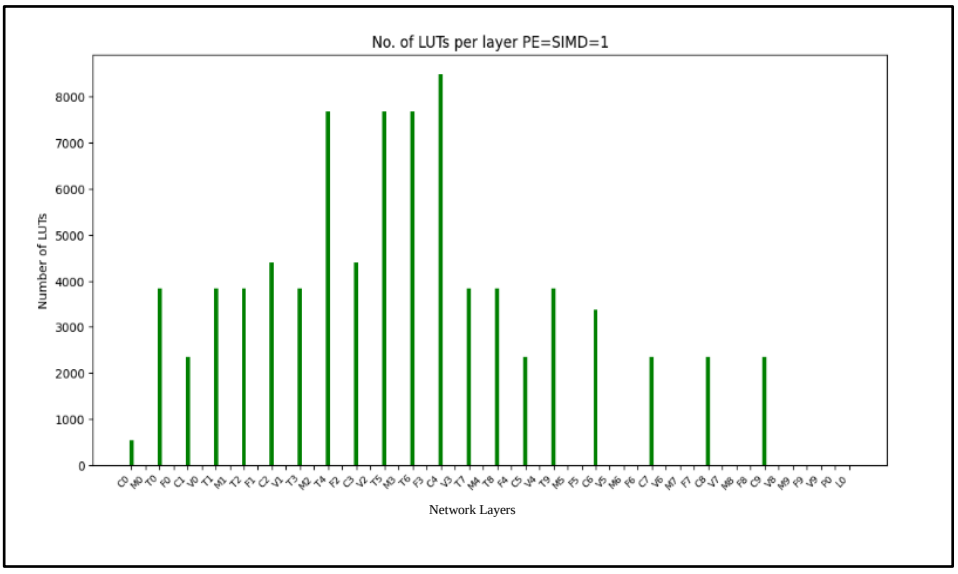

Fig. 11. Number of LUT's Required

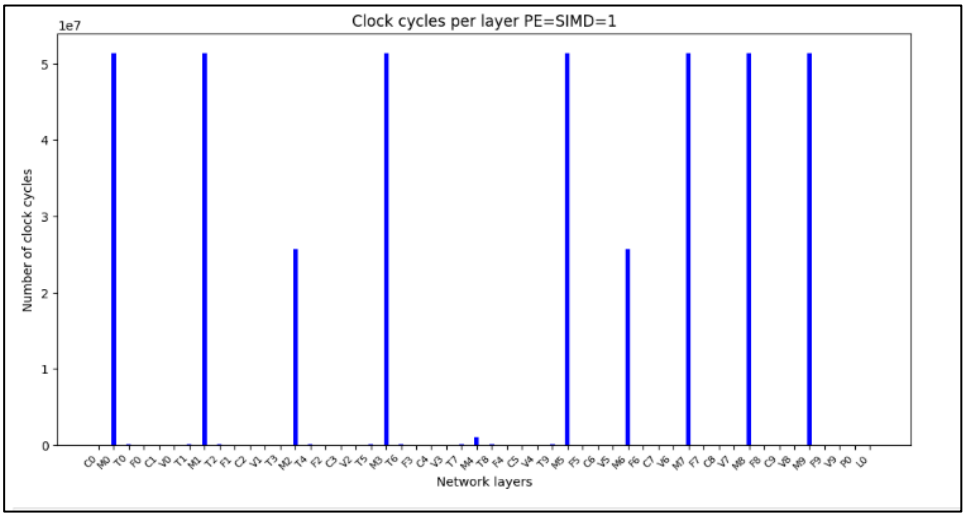

Fig. 12. Timing Before Optimization

 https://doi.org/10.5281/zenodo.15302810

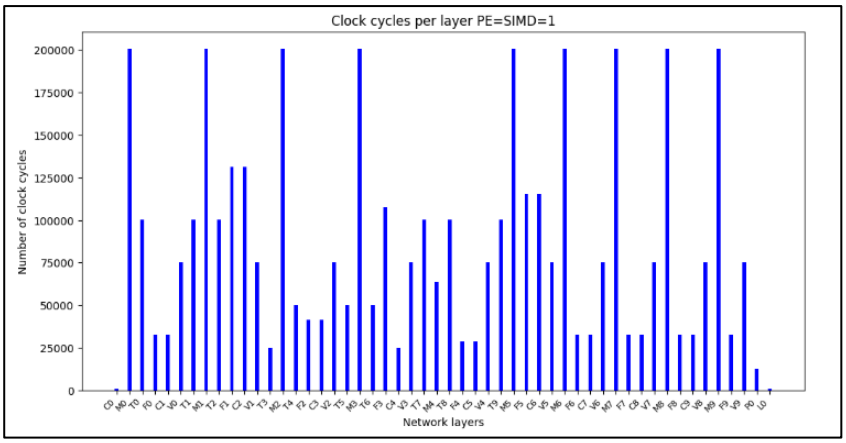

Fig. 13. Timing After Optimization

IP Generation is done. C-Simulation and RTL Simulation is done and the weights generated with the original model is compared with the C- simulation and RTL model simulated weights and they are found to be the same as shown in Fig. 14. The RTL simulation report is generated which provides the various parameters of the simulated model such as latency, throughput , clock frequency, etc. The various parameters generated are shown in Table 2.

```
[33]: # check result with golden values
      golden = np.load("end2end_mobilenet_golden_top5.npy")
      # golden_prob = np.load(build_dir + "/end2end_mobilenet_golden_top5_prob.npy")
      print(golden)
      print(res_rtlsim_ip)
      assert (golden == res_rtlsim_ip).all()
      # assert np.isclose(golden_prob, res_rtlsim_ip_prob[0, 0, 0, :5]).all()

      [152 156 220 219 155]
      [[[[152. 156. 220. 219. 155.]]]]

]:    # check result with golden values
      golden = np.load("end2end_mobilenet_golden_top5.npy")
      golden_prob = np.load("end2end_mobilenet_golden_top5_prob.npy")
      assert (golden == res_cppsim).all()
      print(golden)
      print(res_cppsim)

      [152 156 220 219 155]
      [[[[152 156 220 219 155]]]]
```

Fig. 14. C- Simulation and RTL Simulation Output

Table 2. Simulation Report

Parameter	Value
N_IN_TXNS	50176
N_OUT_TXNS	5
cycles	459942
N	1
latency_cycles	459942
runtime[ms]	1.379826
throughput[images/s]	724.7290600409037
fclk[mhz]	333.3333333333333
stable_throughput[images/s]	724.7290600409037

https://doi.org/10.5281/zenodo.15302810

Conclusion

The LUT usage is highly variable across layers, with certain layers consuming significantly more LUTs (e.g., layers like T5 and T3). This variability suggests that computationally intensive operations (e.g., convolutions with large kernels or layers with high channel counts) are driving resource consumption. In future work to focus optimizing the high-LUT layers by reducing kernel sizes or number of channels and applying techniques like pruning. The clock cycle analysis shows a few dominant layers (e.g., T0, M1, M6) requiring an exceptionally high number of cycles. These layers are bottlenecks in terms of timing, as they take significantly longer to compute compared to other layers.

To reduce clock cycles, we can split large operations into smaller, more parallelizable tasks. Explore layer-wise optimizations, such as reducing bit width or simplifying operations.

FINN supports low-bit width operations, but excessive bit reduction can lead to loss of accuracy. Evaluate the accuracy of the transformed model. If accuracy degradation is significant, consider using QAT to regain precision. If the current design complexity is a challenge, consider simplifying the MobileNet architecture. Use a reduced version of MobileNet (e.g., MobileNetV2-Tiny). Replace layers with more hardware-friendly counterparts, such as using Depthwise separable convolutions or skipping certain blocks.

As future work to implement the Mobilenet model developed using FINN on an FPGA board in real time and test its real time capabilities by using images capture in real-time.

References

T. Nguyen et al., "A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection," in IEEE Transactions on Very Large-Scale Integration (VLSI) Systems, vol. 27, no. 8, pp. 1861-1873, Aug. 2019 .

Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 2017, pp. 1800-1807.

G. Zacchigna, "Methodology for CNN Implementation in FPGA-Based Embedded Systems," in IEEE Embedded Systems Letters, vol. 15, no. 2, pp. 85-88, June 2023.

Krizhevsky, A. et al., “ImageNet classification with deep convolutional neural networks.” Communications of the ACM, vol. 60(6), pp. 84–90 , May. 2017.

L. Xuan et al., "An FPGA-Based Energy-Efficient Reconfigurable Depthwise Separable Convolution Accelerator for Image Recognition," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 69, no. 10, pp. 4003-4007, Oct. 2022.

M. Sandler, et al., "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 2018, pp. 4510-4520.

T. Abtahi, C. Shea, A. Kulkarni and T. Mohsenin, "Accelerating Convolutional Neural Network With FFT on Embedded Hardware," in IEEE Transactions on Very Large-Scale Integration (VLSI) Systems, vol. 26, no. 9, pp. 1737-1749, Sept. 2018.

Venieris, S. I., Kouris, A., & Bouganis, C. “Toolflows for mapping convolutional neural networks on FPGAs.” ACM Computing Surveys, vol. 51(3), pp. 1–39, June. 2018.

Yaman Umuroglu, et al., “FINN: A Framework for Fast, Scalable Binarized Neural Network Inference,” (2017) In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Association for Computing Machinery, New York, NY, USA, pp 65–74.

GeeksforGeeks. (2022, September 29). Depth wise Separable Convolutional Neural Networks. GeeksforGeeks. https://www.geeksforgeeks.org/depth-wise-separable-convolutional-neural-networks/

 https://doi.org/10.5281/zenodo.15302810