3D Point Clouds on Embedded Platforms

Final Year Project

Report by

Amur Saqib Pal (288334)

Muhammad Junaid Ali Asif Raja (290927)

Sannan Zia Abbasi (293786)

In Partial Fulfillment

Of the Requirements for the degree

Bachelor of Electrical Engineering

(BEE)

School of Electrical Engineering and Computer

ScienceNational University of Sciences and

Technology Islamabad, Pakistan

(2023)

DECLARATION

I hereby declare that this project report entitled “3D Point Clouds on Embedded Platforms” submitted to the “School of Electrical Engineering and Computer Sciences (SEECS)”, is a record of an original work done by us under the guidance of Supervisor “Dr. Faisal Shafait” and that no part has been plagiarized without citations. Also, this project work is submitted in the partial fulfillment of the requirements for the degree of Bachelor of Electrical Engineering

Team Member **Signature**

Amur Saqib Pal _____

Muhammad Junaid Ali Asif Raja _____

Sannan Zia Abbasi _____

Supervisor: **Signature**

Dr. Faisal Shafait _____

Dr. Adnan Ul Hassan _____

Date:_____

DEDICATION

I would like to dedicate this work to my teachers, and parents

who have supported me all along this journey to being a graduate of

Electrical Engineering.

ACKNOWLEDGEMENTS

I would like to wholeheartedly thank my advisor Dr. Faisal Shafait and the co-advisor Dr. Adnan ul Hassan for helping us throughout the final year project. I would like to especially mention Mr. Mohsin Ghaffar for his kind and able guidance throughout this project Without their keen guidance and support, it would not have been possible for me to meet the scope of the project in time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Abstract

The aim of this project is to develop a general framework for high-level synthesis (HLS) of deep learning models that deal with 3D point clouds. The framework will allow the efficient implementation of such models on FPGAs. The focus is on the design of the framework to be flexible and modular to support various deep learning models and architectures.

HLS is a method of converting high-level code, like C/C++, into hardware description language (HDL) that can be implemented on an FPGA. By developing HLS layers for deep learning models, it becomes easier to implement all of the models on FPGA, which is a popular platform for efficient deployment of deep learning models due to its parallel processing capabilities and low power consumption.

The development of a high-level synthesis (HLS) framework for deep learning models dealing with 3D point clouds involves creating a set of HLS layers that cater to the specific requirements of deep learning models that operate on point clouds.

Typically, deep learning models for point clouds involve operations like 3D/1D convolutions, pooling, and fully connected layers. These operations have different memory access patterns and arithmetic requirements compared to 2D convolutions used in image-based deep learning models. Therefore, the development of specific HLS layers for 3D point cloud processing is necessary for efficient implementation on FPGA.

The model we are going to be implementing is PointMLPElite and it is based on a multilayer perceptron (MLP) architecture and uses several advanced techniques such as skip connections, residual connections, and group normalization to improve performance. PointMLPElite has achieved state-of-the-art results on several benchmark datasets and is efficient in terms of both memory usage and inference speed.

Chapter 1

INTRODUCTION

Point Clouds

A point cloud is a collection of data points that represent a 3D space. These points can be generated from a variety of sources such as LiDAR sensors, photogrammetry, or structured light scanning. Typically, a point cloud is composed of XYZ coordinates, but can also include additional features such as surface normals, colors, and intensity values.

Point clouds have numerous applications in various fields. In robotics, point clouds can be used to create 3D models of environments for navigation and path planning. Autonomous vehicles use point clouds to create real-time maps for navigation and obstacle avoidance. The gaming industry also utilizes point clouds to create immersive augmented and virtual reality experiences. In the industrial sector, point clouds are used for quality control in manufacturing processes and for 3D printing. Additionally, point clouds can be used in the medical field for surgical planning and simulation.

In the architecture and construction industry, point clouds can be used for creating as-built models of structures and for monitoring construction progress. The use of point clouds can also aid in building maintenance and disaster management by providing detailed information about building structures.

Moreover, point clouds have several applications in environmental science and forestry. LiDAR data collected from airborne and ground-based platforms can provide detailed information about the vertical and horizontal structure of forests, as well as tree heights and canopy density. This information is useful for forest inventory and monitoring, forest management, and for studying the effects of climate change on forest ecosystems.

However, when we deal with 3D point clouds in deep learning, we face many problems due to the following reasons/properties of 3D point clouds.

1. **Irregularity:** The structure of point cloud data is non-uniform, resulting in unevenly distributed points across different regions of a scene or object, causing some regions to have denser points than others.

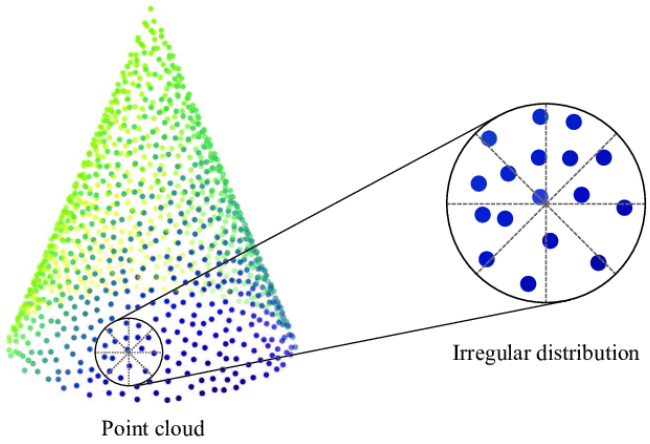

Figure 1 Irregularity of 3D Point Clouds

2. **Unstructured:** Point cloud data differs from regular grids as each point is independently scanned, and the distance to its neighboring points is not always fixed. This is unlike images, where pixels are represented on a 2-dimensional grid with fixed spacing between adjacent pixels.

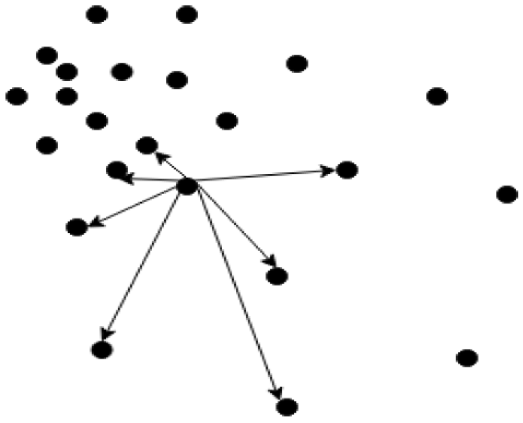

Figure 2 Unstructured 3D Point Clouds

3. **Unordered:** A point cloud refers to a collection of points that are usually represented by XYZ coordinates and are obtained from around the objects in a scene. These points are typically stored as a list in a file, and since they are considered a set, the order in which they are stored does not affect the scene that is being represented.

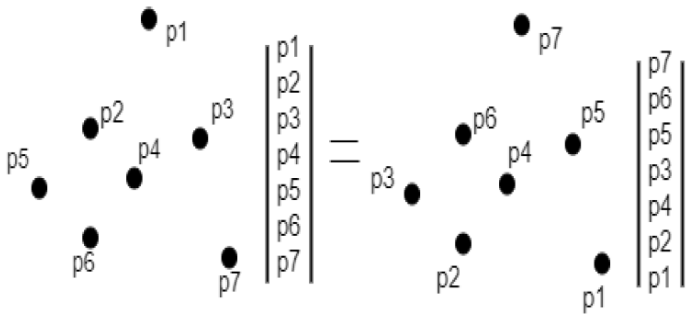

Figure 3 Unordered

The characteristics of point clouds, such as irregularity and unordered data, pose significant challenges for deep learning models, particularly convolutional neural networks (CNNs). CNNs rely on convolutions, which operate on structured, regular, and ordered data that can be represented on a grid. However, point clouds do not adhere to this format, making it difficult to apply CNNs to them.

To overcome these problems, early approaches were to convert the 3D point cloud into the form of a structured grid. However, recent deep learning-based approaches have made the job relatively easier by enabling us to use raw 3D point clouds without

converting them to a structured grid. These approaches can be divided into two different categories that are the following.

1. **Voxelization**: This method involves converting point cloud data into a 3D voxel structure with a specified size and convolving it with 3D kernels of appropriate size. The approach involves two essential operations: preprocessing and learning. Preprocessing involves converting the input into a 3D binary occupancy grid before applying 3D convolution operations to generate a feature vector. This feature vector is then passed through fully connected layers to obtain class scores. The deep convolutional neural network usually includes multiple 3D convolutional, pooling, and fully connected layers.

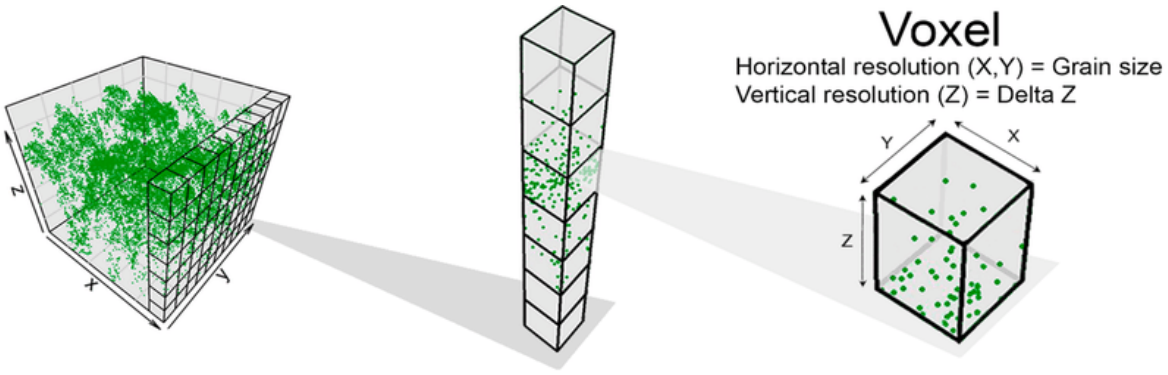

Figure 4 Voxelization

Several studies have proposed and tested various methods using voxelization. Despite showing promising results, voxel-based methods have some limitations. They suffer from high memory consumption due to the sparsity of the voxels, which results in wasted computation when convolving over non-occupied regions. Voxel resolution is also limited by memory consumption, usually between 32 cube to 64 cube. These limitations are in addition to the artifacts introduced by the voxelization operation.

2. **Multi-View Based Methods:** Multi-View based methods are an alternative approach to voxel-based methods in processing point cloud data using deep learning techniques. Multi-View based methods use existing 2D convolutional neural networks (CNNs) that have been well researched and optimized for image processing. Instead of directly processing point cloud data, Multi-View based methods convert the data into a collection of 2D images that represent different viewpoints of the 3D scene. The 2D images are then processed using existing 2D CNN techniques.

   Multi-View based methods have several advantages over voxel-based methods. Firstly, Multi-View based methods do not suffer from the sparsity and memory consumption issues associated with voxels. Secondly, Multi-View images contain richer information than 3D voxels, despite lacking depth information.

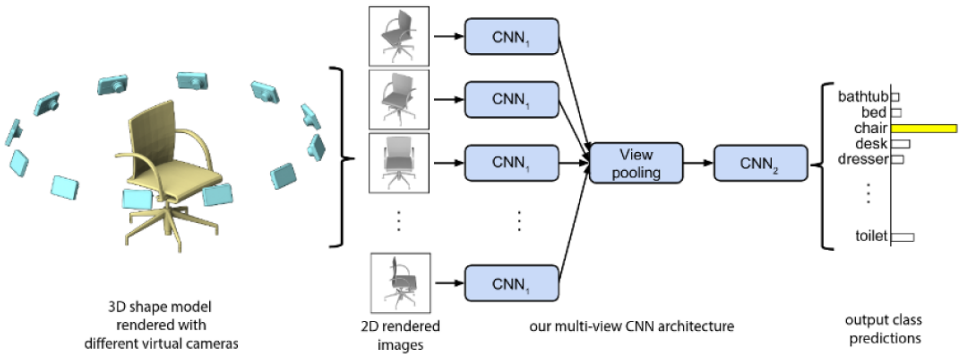

Figure 5 Example of Multi-View Based Processing of 3D Point Clouds

   This is because the Multi-View images contain more detailed texture and color information. Therefore, Multi-View based networks have shown better performance than voxel-based methods.

Chapter 2

PROBLEM STATEMENT

Running deep learning models on FPGAs has several advantages. Firstly, FPGAs offer high parallelism which can result in faster inference time compared to CPU or GPU-based systems. Secondly, FPGAs have low power consumption which can be beneficial for applications that are power-constrained. Thirdly, FPGAs can be reconfigured, which means that they can be adapted to specific applications and can be updated as new algorithms are developed. Lastly, FPGAs offer lower latency and higher bandwidth than traditional computing architectures, which can be useful for real-time applications.

Most modern point cloud DL models have in common layers like 1D convolutions, and algorithms such as furthest point sampling, k-nearest neighbors, and more. Our work provides a highly parallelized set of HLS layers and algorithms that can be used to develop new, or deploy existing deep learning models on FPGAs for point cloud inference. Furthermore, we show the results of our implementation using an ablated version of PointMLPElite.

Target FPGA

The FPGA on which our framework is implemented is AMD Zynq ZC706 developed by Xilinx. It has the following specifications.

Logic Cells	350
Block RAM	19.1 MB
DSP Blocks	900
I/0 Pins	362
Transceiver Count	16

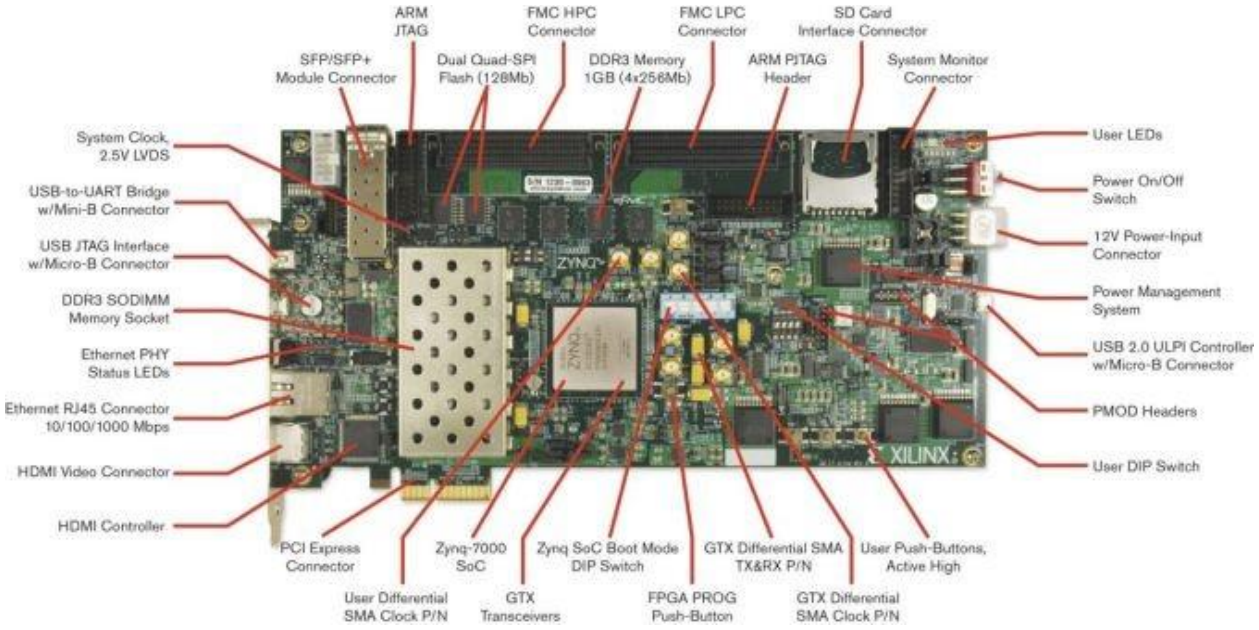

Figure 6 ZC706

Chapter 3

ARCHITECTURE

3.1 SOFTWARE ARCHITECTURE

In this project, we are using the deep learning model PointMLPElite and implementing it on an FPGA. PointMLPElite is an enhanced and advanced version of the deep learning model PointMLP. Both of them are considered state of the art for their ability to perform classification and segmentation on 3D point clouds. Compared to PointMLP, PointMLPElite boasts fewer parameters, a reduced number of layers, and a more compact overall size, making it an attractive option for users seeking improved efficiency and reduced computational cost.

Model	Accuracy (%)	Parameters (M)	Layers	Size (MBs)	Throughput (samples/sec)
PointMLP	94.1	12.6	40	50.49	112
	0.5%	18x		18.4x	
PointMLPElite	93.6	0.68	25	2.744	176

Table 1 Comparison of PointMLP and PointMLPElite

One of the key advantages of PointMLPElite is its ability to process large point clouds with high efficiency due to its fully connected architecture. Additionally, the model can be easily implemented on FPGA due to its efficient design and parallel processing capabilities.

PointNet (Qi et al., 2017a)	1k P	86.0	89.2			
PointNet++ (Qi et al., 2017b)	1k P	-	90.7	1.41M	**223.8**	**308.5**
PointNet++ (Qi et al., 2017b)	5k P+N	-	91.9	1.41M		
PointCNN (Li et al., 2018b)	1k P	88.1	92.5			
PointConv (Wu et al., 2019)	1k P+N	-	92.5	18.6M	17.9	10.2
KPConv (Thomas et al., 2019)	7k P	-	92.9	15.2M	31.0*	80.0*
DGCNN (Wang et al., 2019)	1k P	90.2	92.9			
RS-CNN (Liu et al., 2019b)	1k P	-	92.9			
DensePoint (Liu et al., 2019a)	1k P	-	93.2			
PointASNL (Yan et al., 2020)	1k P	-	92.9			
PosPool (Liu et al., 2020)	5k P	-	93.2			
Point Trans. (Engel et al., 2020)	1k P	-	92.8			
GBNet (Qiu et al., 2021b)	1k P	91.0	93.8	8.39M	16.3	112
GDANet (Xu et al., 2021b)	1k P	-	93.8	0.93M	26.3	14.0
PA-DGC (Xu et al., 2021a)	1k P	-	93.9			
MLMSPT (Han et al., 2021)	1k P	-	92.9			
PCT (Guo et al., 2021)	1k P	-	93.2			
Point Trans. (Zhao et al., 2021)	1k P	90.6	93.7			
CurveNet (Xiang et al., 2021)	1k P	-	94.2	2.04M	20.8	15.0
PointMLP **w/o vot.**	1k P	91.3	94.1	12.6M	47.1	112
PointMLP **w/ vot.**	1k P	**91.4**	**94.5**	12.6M	47.1	112
PointMLP-elite **w/o vot.**	1k P	90.9	93.6	**0.68M**	116	176
PointMLP-elite **w/ vot.**	1k P	90.7	94.0	**0.68M**	116	176

Table 2 Comparison of Different 3D point cloud Deep Learning Models

3.1.1 Architecture Design approach

The design approaches used in point-based methods for point cloud analysis are rooted in the development of PointNet and PointNet++. These techniques were motivated by the desire to process point clouds directly, without the need for preprocessing to transform them into a more structured form suitable for conventional techniques. By avoiding the need for unnecessary rendering processes, these models consume point clouds from the outset, providing a more efficient and effective way to analyze 3D data. PointNet was the first deep learning model to directly process point clouds, achieving state-of-the-art results in several tasks such as object classification and segmentation. PointNet++ improved on PointNet's architecture by introducing hierarchical feature learning and contextual modeling, which further enhanced its ability to capture complex patterns in point cloud data. These models paved the way for the development of other point-based methods and have significantly advanced the field of 3D point cloud analysis.

3.1.2 Architecture Design

PointMLPElite takes 3D points and their corresponding features as input and performs various tasks such as segmentation and classification. It is based on a multilayer perceptron (MLP) network. These points are first processed by a series of fully connected layers, which are responsible for learning and extracting useful features from the point cloud data.

The first MLP layer performs an affine transformation on each point, which helps in learning local features. The subsequent MLP layers perform non-linear transformations to extract high-level features from the input point cloud. The output of the last MLP layer is a feature vector that is passed through a softmax layer for classification or a sigmoid layer for segmentation.

3.1.3 Detailed Working

The PointMLPElite model extracts local features of a given point cloud in multiple steps. The process begins with the geometric affine module, which is applied to the local points of the input point cloud at the beginning of each stage. The geometric affine module rotates and translates the point cloud to allow the model to capture the rotational invariance and the geometric structure of the point cloud.

After the geometric affine transformation, the PointMLPElite model extracts the features of the point cloud in two stages. In the first stage, the model applies a multi-layer perceptron (MLP) to the input point cloud. This is called the pre-extraction stage. The output of the MLP layer is then passed to a max-pooling layer to extract the most relevant features of the point cloud.

To identify local structures in the point clouds, the model uses the farthest point sampling (FPS) algorithm to resample Ns points at each stage and employs K neighbors for each sampled point, which is then aggregated using max-pooling. This helps in capturing the important information of the point clouds while keeping the model size small.

In the second stage, called the post-extraction stage, the model applies another MLP layer to the output of the max-pooling layer. The output of this second MLP layer is then concatenated with the output of the pre-extraction stage. The concatenated output is then passed to another MLP layer to extract the final features of the point cloud.

During the feature extraction process, the model applies the rectified linear unit (ReLU) activation function after each layer to introduce non-linearity to the output. The ReLU activation function has been widely used in deep learning models for its simplicity and effectiveness in avoiding the vanishing gradient problem. By using ReLU, the PointMLPElite model is able to learn more complex representations of the input point cloud and improve its classification and segmentation performance.

By using both pre- and post-extraction stages, PointMLPElite is able to extract more robust features from the point cloud and achieve better classification and segmentation accuracy.

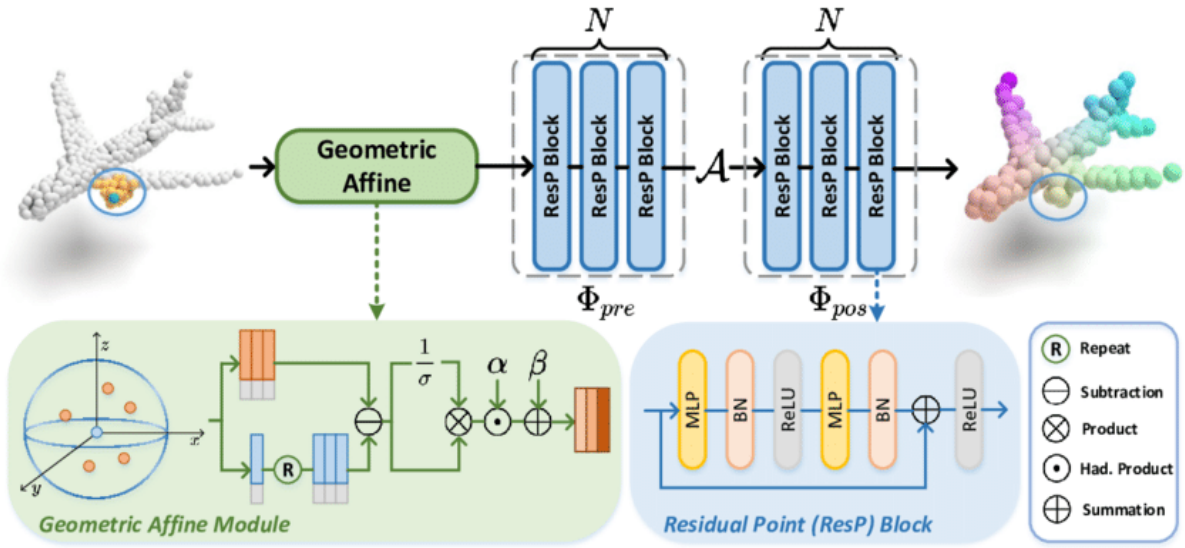

Figure 7 PointMLPElite Topology

The two main formulas in the PointMLPElite model are as follows.

$$g_i = \Phi_{pos}\left(\mathcal{A}\left(\Phi_{pre}\left(f_{i,j}\right), |j = 1, \cdots, K\right)\right)$$

Figure 8 Main Operation of PointMLPElite

Over here A is the aggregation also called max-pooling function. Phi is the function that extracts the important features from the 3D point cloud after the pre-extraction.

The following is the formula for geometric affine:

$$\{f_{i,j}\} = \alpha \odot \frac{\{f_{i,j}\} - f_i}{\sigma + \epsilon} + \beta, \quad \sigma = \sqrt{\frac{1}{k \times n \times d} \sum_{i=1}^{n} \sum_{j=1}^{k} (f_{i,j} - f_i)^2},$$

Figure 9 Mathematical Representation of Geometric Affine

3.1.4 Dataset

The PointMLPElite model was trained on the ModelNet40 dataset, which comprises point cloud models of various everyday objects classified into 40 classes. The dataset includes classes such as chair, plane, and cup, among others. Despite having only 1024 points per point cloud, the model attains state-of-the-art performance on both the class mean accuracy and overall accuracy metrics.

3.2 HARDWARE ARCHITECUTRE

3.2.1 Conv-ReLU and Local Grouper Blocks

The point cloud goes into a Sliding Window Unit, which is a method of breaking up the point cloud into smaller sections to be processed by the convolution operator.

The Sliding Window Unit combines the points with their weights, which essentially gives each point a level of importance in the processing. This combined data is then fed into a MAC (Multiply-Accumulate) operation block, where convolution takes place. Convolution is a mathematical operation that helps to extract features from the data, and it is a common operation used in deep learning algorithms.

The values obtained from the MAC operation then pass through a ReLU (Rectified Linear Unit), which is a common activation function used in deep learning. The ReLU helps to introduce non-linearity into the data, which can make the model more powerful.

Finally, the data is passed through a data width converter, which helps to convert the data into a suitable format for further processing. The output from this stage is an embedding, which is a mathematical representation of the original data that can be used for various purposes, such as classification or clustering. Overall, this pipeline of operations is commonly used in deep learning algorithms for processing point clouds.

After the embedding is created by the conv-relu block, it goes into a Local Grouper. The Local Grouper is a method of clustering points in the embedding space based on their proximity to one another. This is done using the k-NN (k-Nearest Neighbors) algorithm, which finds the k nearest neighbors to each point in the embedding.

The URS (Uniform Resampling System) is used to sample the point clouds to reduce the computation cost. The k-NN algorithm is repeated for the number of samples extracted through URS. For each sample, neighbors are extracted through the embedding, which helps to capture the local structure of the point cloud.

The embeddings are then sent to a normalization layer with two parameters, alpha and beta. This normalization layer helps to standardize the values of the embeddings so that they have zero mean and unit variance. The values of alpha and beta determine the scale and shift of the normalization, respectively.

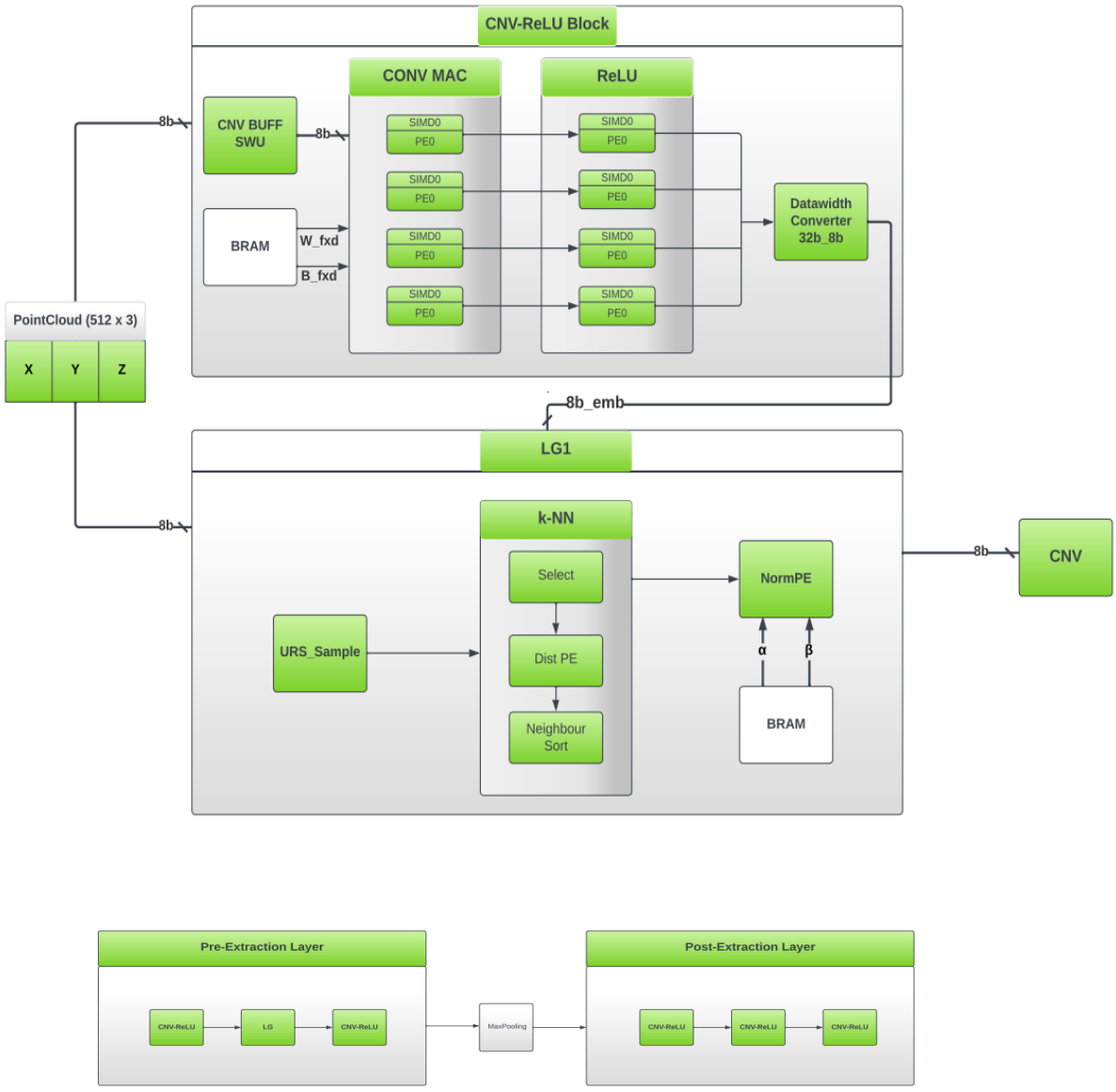

Figure 10 Hardware Implementation

This figure shows the inside of each pre/post extraction block. These blocks consist of varying number of Conv-ReLU blocks and local groupers (exclusive to pre-extraction). They are joined by an aggregation function, max pooling in this case, to create the effect of learning at different scales.

Chapter 4

LITERATURE REVIEW

This chapter gives insights to the research papers and models we explored and took inspiration from. Recently, there has been significant research activity in the field of 3D point clouds. Two main approaches are being used to extract information from these points. The first approach involves extracting useful information from the varying coordinate values of each point in 3D space, depending on the viewer's perspective. Researchers are exploring various techniques to extract practical and meaningful information from these points. Firstly, we would like to discuss PointNet and how it set the foundation for other 3D point cloud deep learning models.

The paper proposes a new deep learning architecture called PointNet for 3D classification and segmentation tasks. Unlike traditional deep learning models which operate on regular grids or images, PointNet takes point sets as input, which are unordered and unstructured data representations of 3D objects.

PointNet operates directly on point clouds without any pre-processing or feature extraction. It first applies a shared Multi-Layer Perceptron (MLP) network to each point independently to extract local features, and then aggregates the features of all points to obtain the global feature of the point cloud. Finally, a fully-connected network is used to make predictions based on the global feature.

Qi,C et al(2017)[1] also propose a novel transformation network that learns a transformation matrix to align the input point cloud to a canonical coordinate system before feeding it to PointNet. This allows PointNet to be invariant to input permutations and translations.

Experimental results on 3D object classification and segmentation datasets demonstrate that PointNet outperforms state-of-the-art methods that use hand-crafted features or voxelization-based methods. PointNet is also shown to be efficient in terms of memory usage and runtime, making it suitable for real-time applications.

Ma, Xu et al. (2022)[2] proposed a simple and efficient deep learning framework, PointMLP, for point cloud classification and segmentation tasks. They introduced

PointMLP as an extension of PointNet++ that uses a residual MLP block to extract local features of the point cloud. This framework overcomes some of the limitations of previous approaches that rely on convolutional neural networks and are not well-suited for unstructured point clouds.

The paper provides a detailed analysis of PointMLP's architecture, including its geometric affine, pre-extraction, and post-extraction phases. The authors show that PointMLP achieves state-of-the-art performance on several benchmark datasets, including ModelNet40 and ShapeNet, while using significantly fewer parameters than other deep learning models. The results demonstrate that PointMLP is highly effective in capturing the local geometric information of point clouds, leading to improved classification and segmentation accuracy.

Furthermore, the authors investigate the impact of different design choices on the performance of PointMLP, including the number of layers, the number of points per block, and the use of different activation functions. They show that a simple design with only two residual MLP blocks can achieve comparable performance to more complex models while significantly reducing the number of parameters.

The paper "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference"[3] introduces a framework for efficient deployment of binarized neural networks (BNNs) on FPGAs. BNNs use binary weights and activations, greatly reducing storage and computation requirements compared to traditional neural networks. However, deploying BNNs on FPGAs requires specialized hardware and software support.

The FINN framework provides a software stack for generating FPGA accelerators for BNNs using high-level synthesis (HLS). The framework includes a BNN-to-FPGA compiler that optimizes and maps the BNN to the FPGA fabric, as well as a software API for interfacing with the generated accelerator. The authors demonstrate that FINN

achieves state-of-the-art performance and energy efficiency compared to other FPGA-based BNN implementations.

The paper also introduces several innovations to improve the FPGAs, enabling high-performance and energy-efficient deployment of deep learning models in embedded and edge computing systems. The framework has been released as an open-source project, making it accessible to researchers and developers interested in deploying BNNs on efficiency and scalability of BNN inference on FPGAs. These include a novel sparse matrix multiplication (SpMM) accelerator for efficient dot-product computations, a custom quantization scheme that maximizes accuracy while minimizing FPGA resource utilization, and a scalable pipeline architecture for multi-layer neural networks.

Overall, the FINN framework provides a powerful tool for accelerating BNN inference on FPGAs.

In another paper, Bai, Lin et al (2020)[4] proposed an implementation of the PointNet architecture on FPGAs for real-time processing of LiDAR point clouds. The authors focus on optimizing the PointNet architecture for FPGA deployment, achieving significant speedup and reduced latency compared to traditional CPU and GPU implementations. This efficient implementation allows for real-time processing of LiDAR data, which is essential for safety-critical applications like autonomous driving. The paper demonstrates that the FPGA-based PointNet implementation can maintain high accuracy while offering reduced power consumption and cost, making it a promising solution for real-time LiDAR point cloud processing in various applications.

Wang, Zilun et al (2022)[5] focus on optimizing the hardware architecture to handle the unique characteristics of point cloud data, which include irregular data structures and complex computational requirements. By leveraging the flexibility and parallelism offered by FPGAs, the authors develop a highly efficient and scalable accelerator capable of outperforming traditional CPU and GPU implementations in terms of processing speed and power consumption.

The framework is not only capable of handling large-scale point cloud data, but also adaptable to various point cloud processing tasks, such as segmentation, classification, and registration. The results presented in the paper again demonstrate that the FPGA-based accelerator maintains high accuracy while offering significant improvements in processing time, energy efficiency, and cost.

B. Graham & L. Van Der Maatens (2017)[6] proposed a novel deep learning architecture designed for efficiently processing high-dimensional sparse data, such as point clouds and volumetric data. Traditional convolutional neural networks (CNNs) often struggle with such data due to their dense representation and the computational complexity of high-dimensional convolutions. To address these challenges, the authors introduce submanifold sparse convolutions, a technique that specifically targets sparse data by only applying convolutions to active sites in the input data.

Submanifold sparse convolutional networks (SSCNs) leverage this technique to create efficient and compact network architectures that can process high-dimensional sparse data with significantly reduced memory and computational requirements compared to traditional CNNs. The authors demonstrate the effectiveness of SSCNs on various tasks, such as 3D object recognition and semantic segmentation, showcasing their ability to maintain competitive performance with state-of-the-art methods while reducing computational complexity.

Wang Yue et al (2019)[7] introduced a novel deep learning architecture called Dynamic Graph Convolutional Neural Network (DGCNN) for processing and learning from point cloud data. As mentioned before, Point clouds are unordered sets of points in 3D space and they pose challenges for traditional convolutional neural networks (CNNs) due to their irregular structure and lack of grid-like topology.

The DGCNN addresses these challenges by constructing a dynamic graph representation of the point cloud data, which captures the local geometric structure of

the data in a flexible manner. The authors introduce a new edge convolution operation that operates on dynamically constructed graphs, enabling the network to learn local and global features from the point cloud data effectively.

Using the works of Wang Yu et al (2019), J. Golzar et al (2023)[8] presented an FPGA-based implementation of the Dynamic Graph Convolutional Neural Network (DGCNN). The authors focus on accelerating the DGCNN using Field-Programmable Gate Arrays (FPGAs) to achieve real-time processing of point cloud data while maintaining high accuracy. They propose a set of optimizations, including memory access optimization, pipelining, and resource sharing, to improve the efficiency and scalability of the FPGA-based DGCNN implementation.

Zheng Yitao et al (2019)[9] presented a novel parallel architecture implemented on Field-Programmable Gate Arrays (FPGAs) for processing point cloud data using neural networks. To address the computational challenges of the properties of 3D Point clouds, the authors propose a parallel architecture specifically designed for FPGAs, enabling efficient processing of point cloud data.

The proposed FPGA-based implementation focuses on exploiting the inherent parallelism of neural networks and optimizing the hardware resources to maximize throughput and minimize latency. The authors introduce a set of optimizations, including memory access optimization, pipelining, and resource sharing, to improve the efficiency and scalability of the FPGA-based neural network for point cloud processing.

Chapter 5

METHODOLOGY

This chapter of the report details the necessary software optimizations and changes made for efficient and successful hardware implementation. The chapter has explanations of the algorithms/working and the reasons why we had to optimize different parts of the model.

5.1 SOFTWARE OPTIMIZATION

In the context of modern machine learning models with large sizes and increasing complexity, various methods are employed to minimize their dimensions and computational requirements as much as possible. Quantization is one such approach that has gained traction in recent years. Essentially, quantization involves truncating a model's weights, biases, and activations, which leads to a more compact representation.

This process not only conserves space by decreasing the number of bits assigned to specific data but also by altering its data type. For example, if a model's weights are stored in a 32 or 64-bit floating-point format, they can be reduced to 8-bit or even lower precision integer values. Essentially, this means we are open to a trade-off between minor accuracy losses and a smaller model size, which in turn results in faster inference and reduced memory footprint.

5.1.1 Quantization-aware Training

There are multiple ways to quantize a neural network, each with its own set of advantages and limitations. One such method is quantization-aware training, which is utilized in this project. Quantization-aware training involves modifying the training process to account for the effects of quantization, such as introducing quantization error during forward and backward passes. This method helps the model to adapt to the reduced precision representation, resulting in a smaller yet accurate model.

Considering that the model is developed in PyTorch and the objective is to implement it on a Xilinx FPGA using the FINN high-level synthesis library, the most suitable choice for quantization-aware training (QAT) is to employ a library called Brevitas.

5.1.2 Brevitas

Brevitas is a PyTorch-based library specifically developed to enable researchers to rapidly investigate the performance of lower-precision alternatives to traditional models. Supported by FINN and Xilinx, it is an optimal choice for our specific implementation.

The brevitas.nn module mirrors the functionality of the torch.nn module in PyTorch. For example, if PyTorch features a convolutional layer called torch.nn.Conv2d, it can be substituted or used in combination with the brevitas.nn.QuantConv2d internal layer present in Brevitas.

A significant reason for Brevitas' adaptability is its extensive array of export options. The library can export quantized models to multiple formats, including FINN, ONNX, Q-ONNX, PyXIR, and TVM. Brevitas' open-source nature and comprehensive features have facilitated the creation, examination, and validation of a wide variety of custom AI accelerators.

5.1.3 Weights Only Optimization

Brevitas offers a variety of quantization schemes, among which is the weights-only quantization. This particular scheme is employed when the reduction of only the model's weights is needed, leaving biases and activations unchanged. For example, in the case of assessing the classification accuracy of a CIFAR-10 classifier using 3-bit integer weights, this quantization scheme proves to be the most effective.

CIFAR-10 is a widely used dataset in the machine learning community for benchmarking image classification algorithms. It consists of 60,000 32x32 color images, divided into 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images.

To apply this weights-only quantization scheme, some modifications are required in the original model implementation. Specifically, the convolutional and linear layers must be replaced with their Brevitas equivalents. The QuantConv2D and QuantLinear layers demand an extra parameter called "weight_bit_width," which can be used to specify the desired precision value (e.g., 3 in this case).

QuantConv2D is a specialized layer in Brevitas that allows for the quantization of the weights within the convolution operation, making it possible to reduce the model's complexity and memory footprint without significant loss in performance.

In contrast, activation, pooling, and batch-normalization layers remain unaltered from their original implementation. By leveraging Brevitas and its weights-only quantization scheme, it becomes feasible to efficiently explore the effects of reduced precision weights on the model's overall performance, ultimately enabling the development of efficient and resource-friendly neural networks.

5.1.4 Lower Precision Optimization

In practical scenarios, reducing a FP32 classifier to have 8-bit integer weights may not yield substantial improvements in speed or resource utilization. For more significant gains, it is necessary to further compress the model by quantizing not only its weights but also its biases and activations.

Brevitas provides multiple internal functions to address such requirements. One such function is QuantReLU, which quantizes the activation functions. Additionally, an 8-bit integer bias library can be imported to quantize the biases as well. The precision for biases is fixed at 8 bits by the developers, but a 16-bit integer (Int16Bias) option is also available, albeit with a less dramatic impact on the model's performance.

It is generally recommended to leave the input and output layers of the model un-quantized to ensure that the model can accurately process incoming data without initial

distortion. To achieve this, a QuantIdentity layer can be placed at the beginning and end of the model, with a configurable bit_width parameter available for customization.

A crucial detail to note is that each quantized layer should have an additional parameter called return_quant_tensor set to True. This setting allows for the proper propagation of the quantized tensor from the output of the previous quantized activation layer to the current quantized layer. Moreover, it provides greater user control and functionality, such as the bias quantization mentioned earlier.

By employing Brevitas and its various functions for quantization, it is possible to achieve more efficient models with improved performance, while maintaining the necessary accuracy for practical applications.

5.1.5 Convolution-Batch Normalization Fusion

Another technique used to optimize deep learning models for FPGA deployment is fusing the batch normalization layers with their preceding convolutional layers. There are two reasons for doing this, the first is to reduce the operations it would take on hardware to run a complete batch normalization layer. The second is that the parameters of the batch normalization layer such as the standard deviation, mean, variance, etc. would no longer have to be allocated space in the BRAM.

Instead, fusion of these two layers allows us to merge the weights and biases of the convolutional layer with the parameters of the batch normalization layer – thus ending up only with one set of parameters, which belong to the convolutional layer acting as the convolutional and batch normalization layer in one.

Suppose that we have a feature map called F ordered as CHANNELS x HEIGHT x WIDTH. This feature map can be normalized by applying simple matrix-vector operations on each of its spatial positions.

$$\begin{pmatrix} \hat{F}_{1,i,j} \\ \hat{F}_{2,i,j} \\ \vdots \\ \hat{F}_{C-1,i,j} \\ \hat{F}_{C,i,j} \end{pmatrix} = \begin{pmatrix} \frac{\gamma_1}{\sqrt{\hat{\sigma}_1^2+\epsilon}} & 0 & \cdots & & 0 \\ 0 & \frac{\gamma_2}{\sqrt{\hat{\sigma}_2^2+\epsilon}} & & & \\ \vdots & & \ddots & & \vdots \\ & & & \frac{\gamma_{C-1}}{\sqrt{\hat{\sigma}_{C-1}^2+\epsilon}} & 0 \\ 0 & & \cdots & 0 & \frac{\gamma_C}{\sqrt{\hat{\sigma}_C^2+\epsilon}} \end{pmatrix} \cdot \begin{pmatrix} F_{1,i,j} \\ F_{2,i,j} \\ \vdots \\ F_{C-1,i,j} \\ F_{C,i,j} \end{pmatrix} + \begin{pmatrix} \beta_1 - \gamma_1 \frac{\hat{\mu}_1}{\sqrt{\hat{\sigma}_1^2+\epsilon}} \\ \beta_2 - \gamma_2 \frac{\hat{\mu}_2}{\sqrt{\hat{\sigma}_2^2+\epsilon}} \\ \vdots \\ \beta_{C-1} - \gamma_{C-1} \frac{\hat{\mu}_{C-1}}{\sqrt{\hat{\sigma}_{C-1}^2+\epsilon}} \\ \beta_C - \gamma_C \frac{\hat{\mu}_C}{\sqrt{\hat{\sigma}_C^2+\epsilon}} \end{pmatrix}$$

Figure 11 Matrix-Vector Operations

In modern deep learning networks, the aforementioned layer can be performed as a simple 1x1 convolution. Moreover, since batch normalization layers in deep learning architecture are placed after convolutional layers, they can be fused together into one layer having the following parameters.

$$W^{new} = W^{BatchNorm} \circ W^{Convolution}$$

$$B^{new} = W^{BatchNorm} \circ B^{Convolution} + B^{BatchNorm}$$

This can be done in PyTorch code as follows.

```
conv_weights = conv.weight.clone().view(conv.out_channels, -1)

bn_weights = torch.diag(bn.weight.div(torch.sqrt(bn.eps+bn.running_var)))

combined_conv.weight.copy_(torch.mm(bn_weights,conv_weights).view(combined_
conv.weight.size()))
```

Thus, now we can add fusion to the list of software optimizations that we plan to study in order to determine the best hardware implementation.

5.1.6 Layer Significance & Alpha-Beta Distribution

Before applying quantization on our model, we can check the significance of each layer by determining the root-mean-square (RMS) values of the weights and biases of each layer. In case a layer-by-layer multi-precision quantization scheme is implemented, this

will allow us to know which layers should be quantized the hardest and which ones should use minimal quantization or be left untouched.

The layer significance of the PointMLP model is given below.

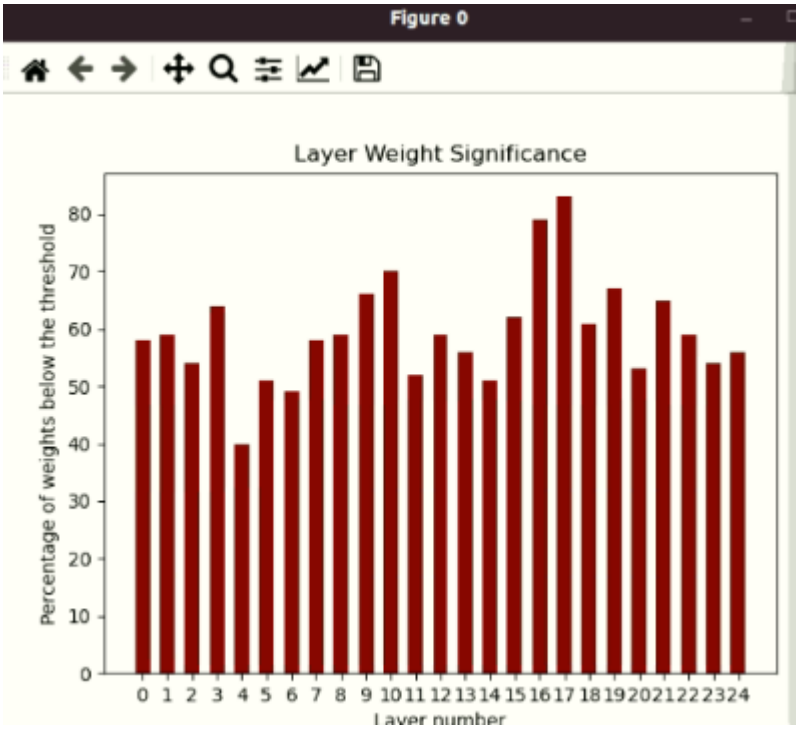

Figure 12 Layer Weight Significance

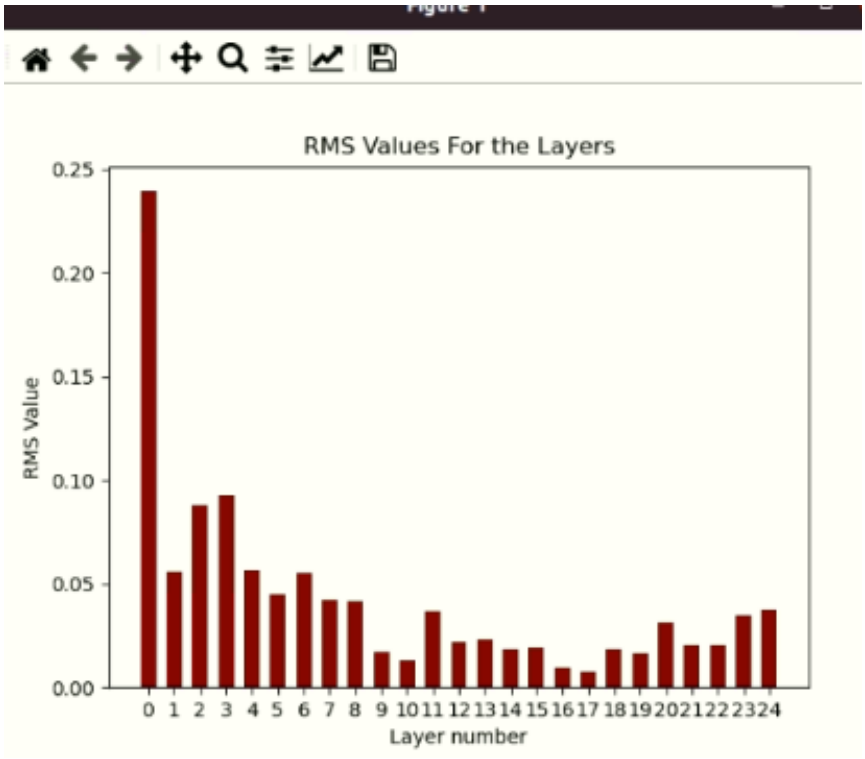

Figure 13 RMS Values of Layers

Furthermore, we can determine the "importance" of the parameters of the geometric affine module i.e. alpha and beta by plotting the distribution of their values. This can help us determine whether or not they can be optimized away in the hardware implementation to make the design smaller, and what kind of effect such an optimization may be expected to have.

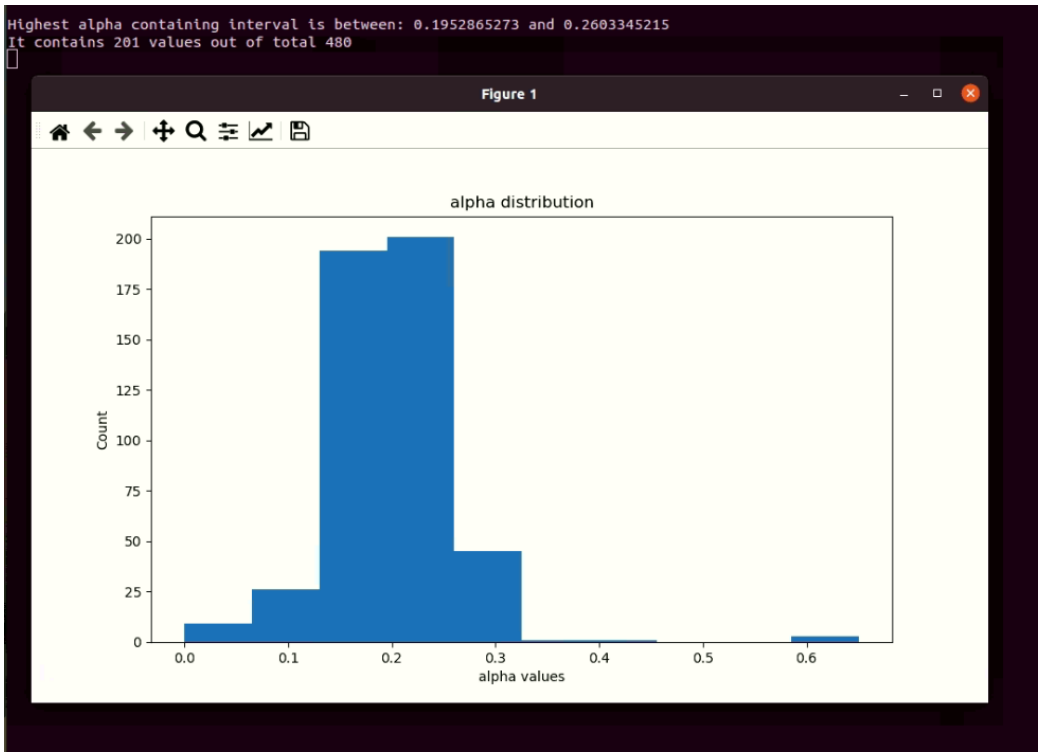

Figure 14 Alpha Distribution

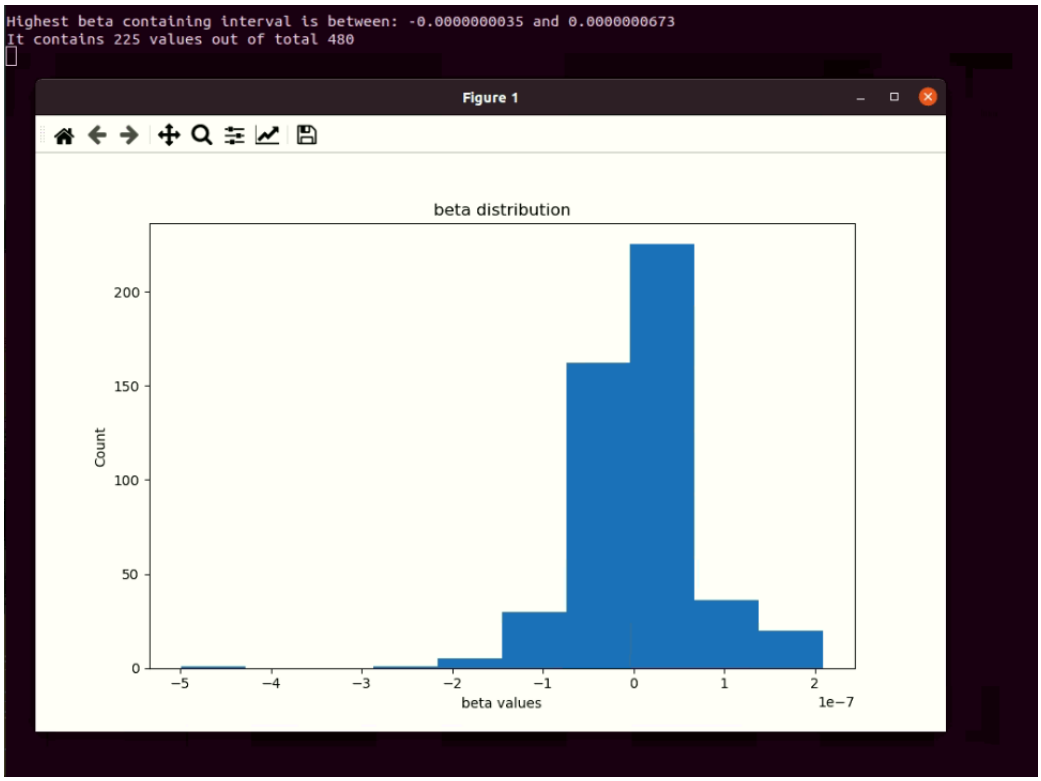

Figure 15 Beta Distribution

As we can see from the distributions, most of the values lie in such a small numeric interval, that they would be rounded to zero in a precision-limited fixed point representation in HLS. The experiments show results with a variety of alpha-beta optimizations, such as Q1.7 alpha, and no beta.

5.1.7 Reduced Density of 3D Point Clouds

The final optimization to PointMLPElite was reducing the number of points in the input point cloud fed into the model. Originally, the ModelNet40 contains point clouds consisting of 1024 distinct points. However, in hardware, this has severe consequences as the arrays and buffers used to contain these values would have to be very large.

Thus, we experimented with reducing the density of the point cloud from 1024 to 512, 512 to 256, and 256 to 128. We found 512 to be the optimal choice, as the accuracy drop was negligible, and it would result in 2x reduction in array sizes.

The presented tables provide a discussion of the accuracies achieved by our model across various optimizations.

Precision	Geometric Parameters	Input Points	Accuracy (%)
FP32	α = FP32, β = FP32	1024	93.07
Fused W8A8	α = Q1.7, β = None	1024	92.75
Fused W4A4	α = Q1.7, β = None	1024	92.34
Fused W8A4	α = Q1.7, β = None	1024	92.18
Fused W6A2	α = Q1.7, β = None	1024	92.02
Fused W8A2	α = Q1.7, β = None	1024	88.57
Fused W8A8	α = Q1.7, β = None	512	91.98
Fused W8A8	α = Q1.7, β = None	256	91.69
Fused W8A8	α = Q1.7, β = None	128	74.07

Table 3 Tradeoff b/w quantization, input points, and accuracy

Precision	Geometric Parameters	Accuracy (%)	Size (KBs)
FP32	α = FP32, β = FP32	93.07	2810
W8A8	α = FP32, β = FP32	93.00	702.5
W4A4	α = FP32, β = FP32	91.04	351.2
Fused FP32	α = Q1.7, β = None	93.07	2,806.75
Fused W8A8	α = Q1.7, β = None	93.23	702.0
Fused W4A4	α = Q1.7, β = None	91.49	351.5
Fused W8A4	α = Q1.7, β = None	92.79	702.0
Fused W6A2	α = Q1.7, β = None	90.70	527
Fused W8A2	α = Q1.7, β = None	90.64	702.0

Table 4 Tradeoff b/w quantization, size, and accuracy

5.2 HARWARE IMPLEMENTATION

In this section, we implemented the software component of the PointMLPElite model on our FPGA by translating the entire model into C++ and then utilizing Vivado HLS to convert the C++ code into Verilog. In order to accommodate the model to the limited resources of the FPGA, we made several optimizations. These optimizations include reducing the number of neurons in each layer of the model to minimize the memory requirements for storing the weights and biases, as well as optimizing the hardware architecture of the model to reduce data movement and improve resource utilization.

5.2.1 Local Grouper

Local Grouper is an algorithm used in point cloud processing to cluster or group points that are spatially close to one another. The main goal of Local Grouper is to identify and extract local structures within a point cloud dataset, which can be helpful in understanding the underlying geometry and relationships between different points.

The Local Grouper algorithm can be implemented using various techniques, such as k-means clustering, DBSCAN, or octrees, depending on the specific requirements and characteristics of the dataset. Here's a general overview of the Local Grouper algorithm:

1. Determine the neighborhood size or distance threshold: Choose an appropriate neighborhood size or distance threshold, which defines the maximum distance between two points for them to be considered part of the same local group. This parameter should be chosen based on the desired granularity of the local groups and the specific characteristics of the dataset.

2. Define the search method: Select a search method for finding neighboring points, such as k-nearest neighbors, radius search, or spatial partitioning (e.g., octrees). The choice of search method depends on the dataset's size, density, and the required computational efficiency.

3. Compute local groups: For each point in the dataset, find all neighboring points that fall within the specified neighborhood size or distance threshold using the chosen search method. Group the neighboring points together to form a local group centered around the current point.

4. Output the local groups: Store the resulting local groups as separate clusters or label each point in the dataset with the corresponding group identifier. This information can be used as input for further processing, such as segmentation, feature extraction, or classification.

5.2.2 Farthest Point Sampling

Farthest Point Sampling (FPS) is an algorithm used to select a subset of points from a larger dataset, which aims to reduce the number of points while maintaining the essential structure and features of the original dataset. It is commonly used in computer graphics,

point cloud processing, machine learning, and computational geometry. FPS can be especially useful in dealing with large point clouds, where processing the entire dataset can be computationally expensive or even infeasible.

The basic idea of the FPS algorithm is to iteratively select points that are as far away as possible from the previously selected points. This helps in maximizing the coverage of the underlying structure and maintaining the essential features of the original dataset with fewer points.

1. Initialization: Choose an arbitrary starting point from the input dataset as the first point in the sampled subset. This can be a random point, the centroid of the dataset, or any other criterion.

2. Distance computation: Calculate the Euclidean distance (or any other appropriate distance metric) between the first sampled point and all other points in the dataset.

3. Point selection: Select the point with the maximum distance from the first sampled point and add it to the sampled subset.

4. Update distances: For each point in the dataset, calculate the distance to the newly selected point. Update the distance value for each point by taking the minimum of the current distance and the distance to the new point. This ensures that the distance value for each point is the minimum distance to any of the points in the sampled subset.

5. Iteration: Repeat steps 3 and 4 until the desired number of points are sampled, or a stopping criterion is met (e.g., the minimum distance between points in the dataset falls below a certain threshold).

5.2.3 Uniform Random Sampling

Uniform Random Sampling (URS) is a widely used statistical technique for selecting a subset of data points from a larger dataset, where each point has an equal probability of being included in the sample. The main goal of this method is to obtain a representative sample of the entire dataset without any bias or distortion. URS is widely used in various fields, such as survey design, machine learning, and statistical analysis.

1. Define the sample size: Determine the desired number of points (n) that you want to select from the dataset.

2. Assign unique identifiers: For each data point in the dataset, assign a unique identifier, such as an index or a key. This helps in tracking the points and avoiding duplicate selections.

3. Generate random numbers: Generate n random numbers without replacement from the range of the unique identifiers assigned in step 2. The random numbers should be generated using a well-defined probability distribution, typically a uniform distribution, to ensure equal probability for each data point. Many programming languages and libraries provide built-in functions to generate random numbers from a uniform distribution.

4. Select data points: For each generated random number, select the corresponding data point from the dataset using its unique identifier. Add the selected points to the sampled subset.

5. (Optional) Repeat sampling: If necessary, repeat steps 3 and 4 to obtain multiple independent samples from the dataset.

PointMLPElite uses the Farthest Point Sampling algorithm on the isolated regions produced by the local grouper. Although effective in generating well-distributed samples from large datasets, it tends to consume more FPGA resources, such as Block RAM (BRAM) and Digital Signal Processing (DSP) blocks, compared to Uniform

Random Sampling (URS). The resource consumption in FPS is primarily due to the iterative nature of the algorithm, which involves distance computation between sampled points and the remaining data points at each step, followed by selection and updating of distances. This process leads to increased computational complexity, resulting in higher utilization of BRAM and DSP blocks on FPGAs. Furthermore, the iterative nature of FPS contributes to poor latency performance compared to URS, which relies on generating random numbers to directly sample points from the dataset. URS's simplicity and non-iterative nature make it more resource-efficient and faster in terms of latency, giving it an advantage when dealing with resource-constrained FPGA platforms or applications that demand low-latency solutions.

By using URS instead of FPS we were able to implement our design with virtually no BRAM, DSP, FF and LUT consumption.

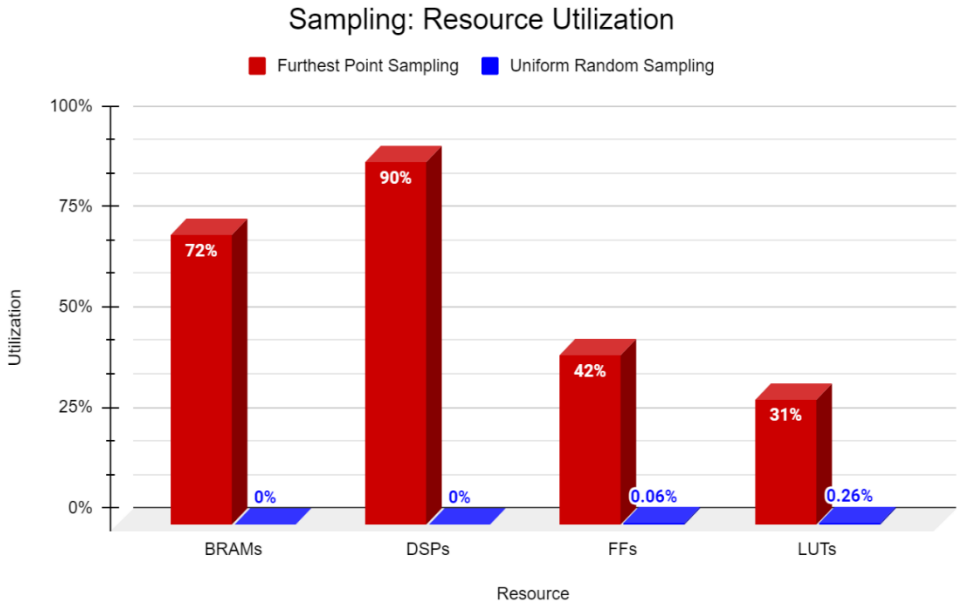

Figure 16 Sampling Resource Utilization

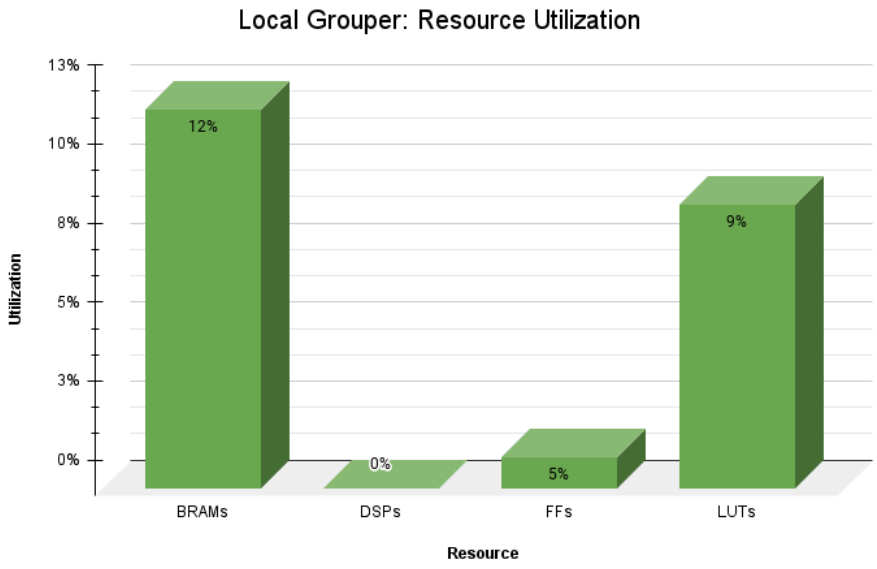

Figure 17 Local Grouper Resource Utilization

5.2.4 k-NN

PointMLPElite uses the simple k-NN algorithm while processing the point cloud data. The problem faced by doing this was that it consumed a lot of Block RAM of our target FPGA device. This was in part due to using arrays to pass information regarding point clouds to the k-NN and storing the nearest neighbors. The workaround was to use streaming k-NN. Streaming k-Nearest Neighbors is a variation of the k-Nearest Neighbors algorithm designed to handle data streams, where data points arrive sequentially over time. The streaming k-NN algorithm is based on FIFOs and thus significantly reduces BRAM consumption

In the streaming k-NN algorithm, the k nearest neighbor points are typically sorted in ascending order according to their distances from the query point. However, in some applications, it might be necessary to sort the nearest neighbor points based on their data type or size, rather than their distance. For example, if the data points represent images,

the nearest neighbor points may need to be sorted based on their image size or resolution, rather than their distance. This sorting can be done by extracting the required features or metadata from the nearest neighbor points and using a comparison function to sort them accordingly. This approach can be particularly useful in scenarios where the data points have complex or multi-dimensional attributes, and the distance metric alone is not sufficient to capture the desired similarities or differences. Sorting the nearest neighbor points based on data type or size can also help in reducing the computation cost of subsequent analysis tasks by prioritizing the relevant points or features. This helped us greatly reduce the BRAM consumption when we implemented the model on the FPGA.

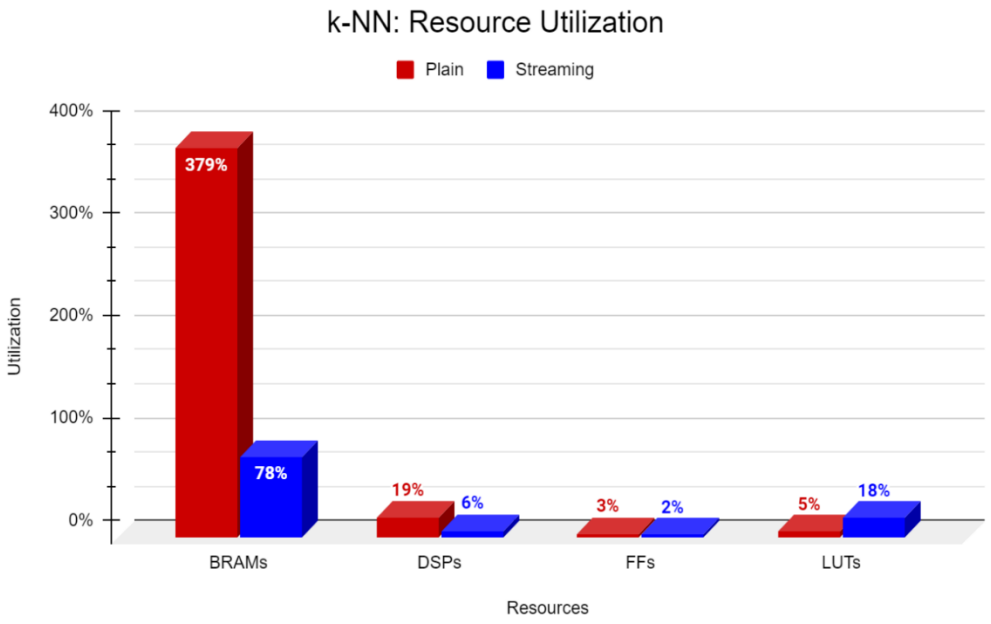

Figure 18 k-NN Resource Utilization

After further optimization of the algorithms, when put together in the local grouper, we get the following overall hardware resource utilization estimates in Vivado HLS.

5.2.5 Linear Feedback Shift Registers

A linear feedback shift register (LFSR) is a digital circuit used for generating pseudo-random sequences of binary numbers. It is based on a shift register, which is a series of flip-flops connected in a chain, where each flip-flop stores a single bit of data. The LFSR adds a feedback mechanism, which combines the outputs of selected flip-flops using an XOR gate, to produce a new input to the first flip-flop.

The LFSR generates pseudo-random sequences by repeatedly shifting the contents of the shift register and applying the feedback function to the output bit. The sequence of binary numbers generated by the LFSR depends on the initial state of the shift register, which is known as the seed value.

To mimic Uniform Random Sampling in hardware using LFSRs, we can use the output of the LFSR as a binary representation of a number. By selecting an appropriate LFSR length, we can generate a sequence of binary numbers that appear to be uniformly distributed. To obtain a random number in a specific range, we can use modular arithmetic to reduce the generated number to the desired range. For example, if we want to generate a random number between 0 and 255, we can use an 8-bit LFSR and compute the remainder of the generated number divided by 256. By repeating this process, we can generate a sequence of random numbers that appear to be uniformly distributed in the desired range.

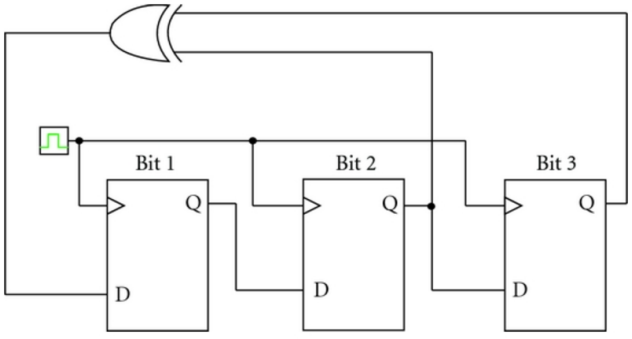

Figure 19 Linear Feedback Shift Register

Chapter 6

RESULTS AND DISCUSSIONS

6.1 Vivado: Power, Timing & Resources

We have discussed the experiments we conducted in software (with Brevitas and other optimizations), and the hardware simulations in Vivado HLS. Both of these steps of the ablation study have helped set up a firm design foundation that we will be running on the hardware.

This section describes the Vivado workflow and results obtained after passing synthesis and hardware-software co-simulation.

6.1.1 Block Design

Once the RTL is exported from Vivado HLS, it is repackaged as an IP and used in Vivado to construct the block design of the entire SoC.

The block design consists of seven IPs, namely S1, S2, S3, S4 (further divided into S4.1, S4.2, S4.3, and S4.4). These IPs are the 4 different stages of PointMLPElite, each with their own constituent layers and structure of pre/post-extraction.

As evident from the figure, the ZYNQ7 processing system is used as the PS on our SoC, and it directs all the control signals and initiates transactions from DRAM to DMA, from DMA to BRAM, and back.

The DMA takes the input point cloud using a High Performance AXI slave port (HP0 on the Zynq) from the DRAM, which is then transferred from the DMA to the BRAM. This is done by connecting the stream master MM2S to the input of the first IP – S1. Next, the output of this IP is fed directly as the input to S2, then the output of S2 goes to S3, and so on until S4.4. When S4.4 is done processing, DMA transfers the processed results using the S2MM pin and writes it back to the DRAM, which we can read through a device-specific hardware driver later.

The control signals of each IP are connected to AXI master ports of the AXI interconnect. Moreover, the interrupts of each IP and the DMA (mm2sintr_out & s2mmintr_out) and concatenated and joined to the IRQ_F2P port of the Zynq PS.

6.1.2 Synthesis & Implementation

Having verified and generated the block design, the next step is to synthesize it. We use out-of-context module-level synthesis and implementation as it is faster and suitable for our design (considering we have multiple IPs).

After running implementation, we must make sure that all the logical IO ports in the implemented design have user-assigned location constraints and a user-specified IO standard. This can be done by using Vivado's internal IO planning feature which autoplaces the logical ports at the correct locations for Xilinx ZC706. The ports are then fixed in their places.

Finally, a short TCL script can be ran in the terminal to assign the LVCMOS18 standard to all the pins. This eliminates all UCIO and NSTD critical warnings, and we can then see the placed circuit on the board.

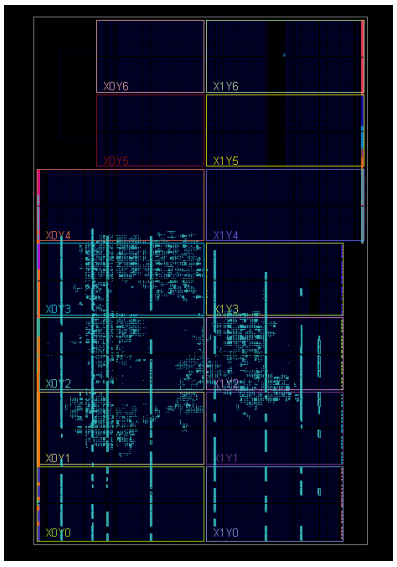

Figure 20 Routing & Placement

6.1.3 Power Analysis

After the bitstream has been generated, Vivado provides power information of the design we have implemented. As it can be seen, 63% of the total on-chip power (0.575W) was used by the PL i.e. our design. This comes out to be 0.36W and the PS consumes 0.214W. Such power consumption is excellent in terms of performance, and tens of times better than what the GPU consumes to run the same model in SW.

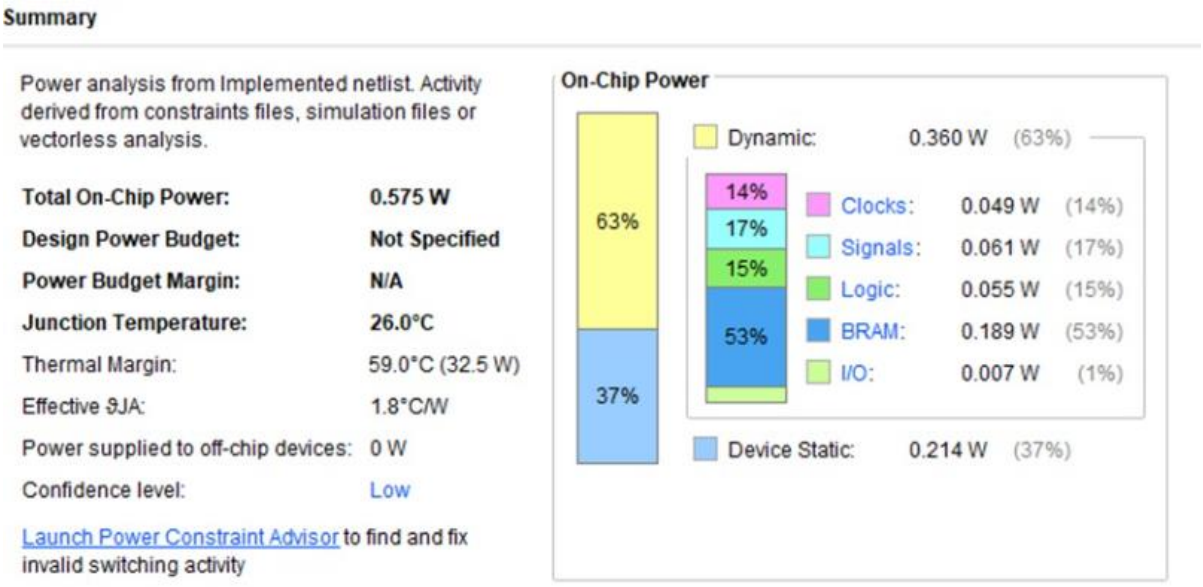

Figure 21 FPGA Power Consumption

6.1.4 Timing Analysis

Similarly, we can also use Vivado to investigate timing details of our design. These are given below and show good performance as all user-specified timing constraints are met.

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	2.377 ns	Worst Hold Slack (WHS):	0.056 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	25867	Total Number of Endpoints:	25867

All user specified timing constraints are met.

Figure 22 Timing Analysis

6.1.5 Resource Utilization

In the final step, we can see final resource utilization of PointMLPElite on the board. As shown, only 40% of BRAM is used due to our 8-bit fixed point implementation and hardware optimizations that we used earlier.

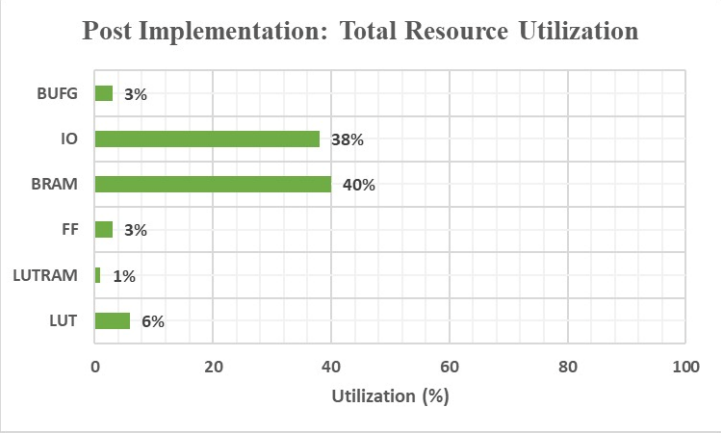

Figure 23 Resource Utilization

6.2 FPGA: Latency, Throughput & Speedup

In this section, we show and discuss the results after running our PointMLPElite implementation on the Xilinx Zynq-7000 SoC ZC706 evaluation kit. The exact FPGA chip used is XC7z045-FFG900.

6.2.1 Board Setup

A hardware driver written for our Xilinx board is used to set the DRAM addresses for the input point cloud and the output results. This driver object is instantiated at the following address.

```
auto *driver = new HardwareDriver(0x00000000, true);
```

The design takes in 1536 unsigned integers which are reinterpreted inside the blackbox IP as ap_fixed<8,1> datatypes. These values are the input point cloud and are written to the DRAM using the hardware driver at the following addresses.

```
inputAddress = 0x2000000
outputAddress = 0x3000000
```

Having done that, we simply set the inputs and send a control signal to the IP from the Zynq PS to start processing the point cloud from the specified address. Once it's finished, it writes the results back to the output address, and they can be read.

6.2.2 Results

As shown below, we get the correct prediction for the input point cloud. Furthermore, we also record the latency and throughput values.

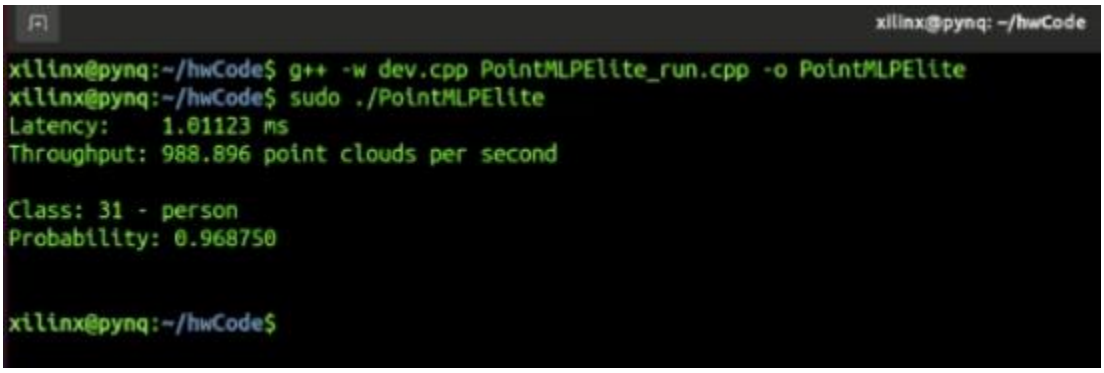

Figure 24 FPGA Latency and Throughput

6.2.3 Speedup

The following image shows PointMLPElite inference and records the time it takes for the NVIDIA RTX A5000 GPU to process it. The PointMLP paper reports the model's software testing speed to be 176 samples per second.

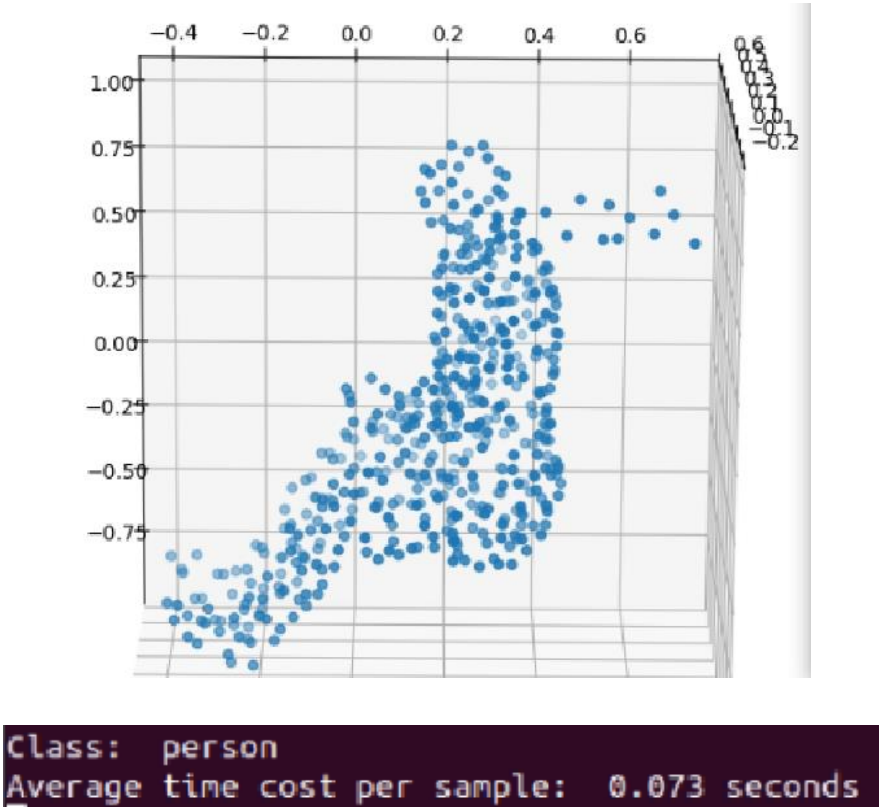

Figure 25 FPGA Samples/sec

We can see that there is a significant increase in throughput and decrease in latency when comparing the results with our FPGA point cloud accelerator. The latency is improved by 73x, and the throughput is 5.61x better.

6.3 Discussion

In this project, we have shown FPGAs to be a prime target device for acceleration of 3D point cloud deep learning models. Additionally, we have designed an HLS framework which makes it possible to not only deploy existing deep learning point cloud models on FPGAs, but also build and train new ones.

The significant performance boost also shows that high-level synthesis provides a practical approach to hardware implementations of large deep learning models, and delivers great real-time performance in terms of latency, throughput, and low power consumption.

Chapter 7

CONCLUSION

In conclusion, this final year project focused on the development of a general framework for high-level synthesis (HLS) of deep learning models dealing with 3D point clouds. The aim was to enable efficient implementation of these models on FPGA platforms, taking advantage of their parallel processing capabilities and low power consumption.

Specific HLS layers were designed to cater to the unique requirements of deep learning models operating on 3D point clouds. Unlike image-based deep learning models that utilize 2D convolutions, point cloud models involve operations such as 3D/1D convolutions, pooling, and fully connected layers. These operations possess distinct memory access patterns and arithmetic requirements, necessitating the development of specialized HLS layers for efficient FPGA implementation.

The chosen model for implementation, PointMLPElite, employs a multilayer perceptron (MLP) architecture and incorporates advanced techniques like skip connections, residual connections, and group normalization. This model has demonstrated state-of-the-art performance on benchmark datasets while being memory-efficient and achieving fast inference speeds.

The successful development of this HLS framework opens up opportunities for accelerated processing of 3D point clouds on embedded platforms, specifically FPGAs. This has significant implications for applications such as autonomous driving, robotics, virtual reality, and augmented reality, where real-time processing of 3D data is crucial.

Future work could involve expanding the framework to support additional deep learning models and architectures specifically designed for 3D point clouds. Furthermore, optimizations can be explored to improve the overall performance and efficiency of the HLS framework, enabling even faster inference speeds and reduced power consumption.

References

[1] Qi, C. R., Su, H., Mo, K., & Guibas, L. J. (2017). Pointnet: Deep learning on point sets for 3d classification and segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 652-660).

[2] Ma, X., Qin, C., You, H., Ran, H., Fu, Y. (2022). Rethinking network design and local geometry in point cloud: A simple residual MLP framework. arXiv preprint arXiv:2202.07123.

[3] Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M., Vissers, K. (2017, February). Finn: A framework for fast, scalable binarized neural network inference. In Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays (pp. 65-74).

[4] Bai, L., Lyu, Y., Xu, X., Huang, X. (2020, October). Pointnet on fpga for real-time lidar point cloud processing. In 2020 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-5). IEEE.

[5] Wang, Z., Mao, W., Yang, P., Wang, Z., Lin, J. (2022, September). An Efficient FPGA Accelerator for Point Cloud. In 2022 IEEE 35th International System-on-Chip Conference (SOCC) (pp. 1-6). IEEE

[6] Graham, B., Van der Maaten, L. (2017). Submanifold sparse convolutional networks. arXiv preprint arXiv:1706.01307.

[7] Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., Solomon, J. M. (2019). Dynamic graph cnn for learning on point clouds. Acm Transactions On Graphics (tog), 38(5), 1-12.

[8] Jamali Golzar, S., Karimian, G., Shoaran, M., Fattahi Sani, M. (2023). DGCNN on FPGA: Acceleration of the Point Cloud Classifier Using FPGAs. Circuits, Systems, and Signal Processing, 42(2), 748-779.

[9] X. Zheng, M. Zhu, Y. Xu, Y. Li, An FPGA based parallel implementation for point cloud neural network, in 2019 IEEE 13th International Conference on ASIC (ASICON) (2019), pp. 1–4