Implementation of MNIST Dataset using HLS4ML

Tselepi Eleni
Univerity of Thessaly(UTH)
Volos, Greece
etselepi@uth.gr

Georgios Kapakos
University of Thessaly (UTH)
Volos, Greece
gkapakos@uth.gr

***Abstract*—In this paper we use the open source tool HLS4ML, in an Ubuntu Linux System to optimize the area resources of a Neural Network, using the MNIST dataset. These optimizations include prunning and quantization of the model. All the changes are applied in the jupyter notebook created for the project. Experimental results have shown a significant 92% area reduction in DSP usage for optimized models over the unoptimized with 3% accuracy loss.**

***Index Terms*—HLS4ML, Machine Learning, MNIST, FPGA, Low-power**

I. Introduction

Deep Neural Networks are computationally intensive algorithms capable of obtaining outstanding results in applications. As new models keep growing in size and complexity, their number of operations and memory footprint pose significant challenges, especially under the strict latency and power consumption constraints typical of real-time edge systems. The main obstacle to broader adoption of FPGAs in the Deep Learning community is the wide gap between software design and RTL implementation: the translation of a Deep Neural Network model typically written and trained in a Python-based software framework into an efficient FPGA design requires hardware expertise that is not easily found among Machine Learning experts. Using the hls4ml library, an open source software tool for deploying machine learning models on FPGAs, we implemented a standard benchmark dataset like MNIST in order to examine area optimizations.

The rest of this paper is organized as follows. Section II briefly provides information about the HLS4ML and the dataset we use, while Section III describes the methodology we followed. In Section IV, we present the optimizations we have implemented on our model. Finally, Section V demonstrates our experimental results, while conclusions are drawn in Section VI.

II. Case Study

Below we present some information about the HLS4ML tool and the MNIST dataset we used.

A. HLS4ML

The hls4ml library [1] is an open source software designed to facilitate the deployment of machine learning (ML) models on field-programmable gate arrays (FPGAs), targeting low-latency and low-power edge applications. Taking as input a neural network model, hls4ml generates C/C++ code designed to be transpiled into FPGA firmware by processing it with a high-level synthesis (HLS) library. The hls4ml software is structured with a set of different back-ends, each supporting a different HLS library and targeting different FPGA vendors. So far, new development has been focused on the Vivado HLS back-end targeting Xilinx FPGAs. The workflow was demonstrated for fully-connected, or dense, neural networks (DNNs), binary and ternary networks, boosted decision trees, and graph neural networks. The hls4ml library accepts models from TENSORFLOW, KERAS, PYTORCH, and via the ONNX interface. It has been interfaced to QKERAS, in order to support quantization-aware training (QAT) allowing the user to better balance resource utilization and accuracy. The hls4ml design focuses on fully-on-chip deployment of neural network architectures. This avoids the latency overhead incurred by data transmission between the embedded processing elements and off-chip memory, reducing the overall inference latency. Conversely, this approach constrains the size and complexity of the models that the HLS conversion can easily support. Nevertheless, complex architectures can be supported, in the case of graph neural networks. Since HLS4ML [2] is still developing it supports only a few tools. It depends on a number of Python packages and external tools for synthesis and simulation, specifically, the dependencies are presented below:

- Linux Operating System
- TensorFlow (version 2.4 and newer) and QKeras are required by the Keras converter.
- Xilinx Vivado HLS 2018.2 to 2020.1 for synthesis for Xilinx FPGAs

Here we cite and explain some useful HLS4ML functions [3] that we also used in our project.

1) Useful HLS4ML functions:

- hls4ml.utils.config_from_keras_model: This function serves as the initial step in creating the custom conversion configuration. Users are advised to inspect the returned object to tweak the conversion configuration. The return object can be passed as hls_config parameter to convert_from_keras_model. This configuration is required by hls4ml to correctly interpret and convert the model into HLS code.
  Parameters are:
  - model: This parameter expects a Keras model. The Keras model built sequentially.

  - granularity: This parameter determines the level of detail in the generated configuration. The default is 'model', which configures the model as a whole. Other values might allow for more detailed configuration, such as layer-level settings.
- converters.create_config: Generates a configuration dictionary that encompasses both the global settings for the HLS conversion process and the specific settings for each layer of the machine learning model. This configuration is essential for fine-tuning the HLS conversion to match the target FPGA's constraints and the desired performance characteristics.
  Parameters are:
  - backend: Specifies the backend framework (e.g., 'Vivado', 'Quartus', etc.).
- hls4ml.converters.keras_to_hls: This function automates the conversion of a Keras model to an HLS project. It takes the Keras model and the configuration settings, generating an HLS representation of the model that can be synthesized for FPGA deployment.
  Parameters are:
  - model: This is the Keras model to be converted.
- hls4ml.utils.plot_model: Generates a graphical representation of the HLS model, showing the different layers, their shapes, and optionally the precision of the data types used. This function helps users to quickly grasp the model architecture and ensure that it has been correctly translated from the original Keras model.
  Parameters are:
  - hls_model: The HLS model object to be visualized. This object is typically obtained after converting a Keras model using hls4ml.converters.keras_to_hls
  - show_shapes: A boolean flag indicating whether to display the shapes of the layers in the plot.
  - show_precision: A boolean flag indicating whether to display the precision of the data types used in each layer.
  - to_file: A string specifying the file path where the plot image should be saved. If None, the plot will be displayed but not saved to a file.
- hls4ml.model.profiling.numerical: Provides tools for profiling and evaluating the numerical behavior of HLS models. This module is particularly useful for verifying the accuracy and performance of the converted HLS model by comparing its outputs against the original Keras model. It's purpose is to compare the outputs of the HLS model with the original Keras model, evaluate the numerical precision and accuracy of the HLS model and track problems that affect the conversion process.
  Parameters are:
  - model: This is the Keras model to be converted.
  - hls_model: This is the HLS model created previously, by converting the Keras model.
- hls_model.compile(): This function compiles the HLS model, transforming it into a format ready for FPGA synthesis. It processes the high-level model configuration, translates the neural network layers into C++ code, and sets up the necessary configurations for the HLS tool. This step is crucial for preparing the model for hardware deployment, ensuring the generated code is optimized and compatible with the FPGA synthesis process.
- hls_model.build(): the build method automates the synthesis and implementation steps required to translate the HLS model into a format that can be deployed on an FPGA.
  This consists of:
  - Generating the HLS code from the model description.
  - Compiling the HLS code using the specified synthesis tool (e.g., Vivado HLS).
  - Creating the FPGA bitstream.

B. MNIST

The MNIST dataset [4]is from the National Institute of Standards and Technology (NIST). The training set consists of handwritten numbers from 250 different people, of which 50% are high school students and 50% are from the Census Bureau. The test set is also the same proportion of handwritten digital data. MNIST dataset totally contains 60,000 images in the training set and 10,000 patterns in the testing set, each of size 28x28 pixels with 256 gray levels. Some examples from the MNIST corpus are shown in Fig1.

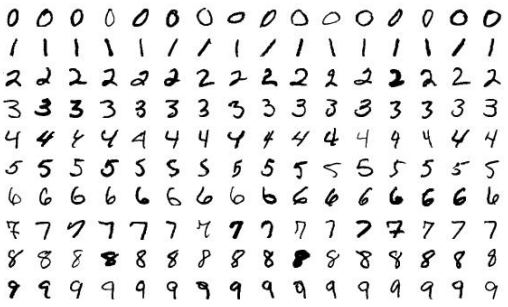

Fig. 1. Example images from the MNIST dataset.

III. Methodology

A. Set Dataset

At the beginning, we use TensorFlow Datasets (TFDS) to load the MNIST dataset, splitting it into training, validation, and test sets, and then extracting and displaying some basic information about the dataset. We use the first 90% of the training data for training and the last 10% for validation and for visualization purposes we display examples from the training dataset as we can see in Fig. 2

B. Model

Then we need to define a model. The model is composed by the following different types of layers:

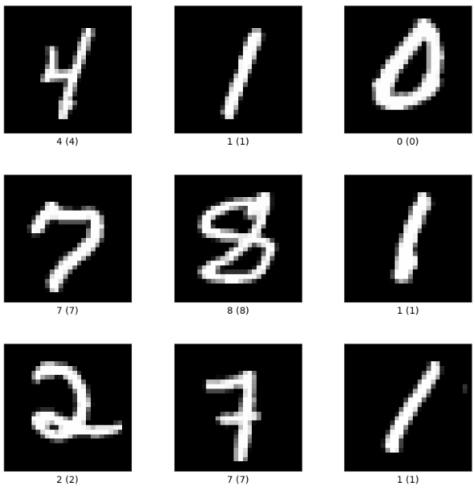

Fig. 2. Examples from the training dataset

- **Input Layer**: It is the first layer of our function having an input shape of (28, 28, 1), which is the size of an image of the MNIST Dataset. It receives the mnist dataset's data.
- **Conv2D Layer**: The first layer of a Convolutional Neural Network is always a Convolutional Layer. Convolutional layers apply a convolution operation to the input, passing the result to the next layer. A convolution converts all the pixels in its receptive field into a single value. For example, if you would apply a convolution to an image, you will be decreasing the image size as well as bringing all the information in the field together into a single pixel. The final output of the convolutional layer is a vector. Based on the type of problem we need to solve and on the kind of features we are looking to learn, we can use different kinds of convolutions. The most common type of convolution that is used is the 2D convolution layer and is usually abbreviated as conv2D. A filter or a kernel in a conv2D layer "slides" over the 2D input data, performing an element-wise multiplication. As a result, it will be summing up the results into a single output pixel. The kernel will perform the same operation for every location it slides over, transforming a 2D matrix of features into a different 2D matrix of features.
- **Max Pooling Layer**: Max pooling is a downsampling technique commonly used in convolutional neural networks (CNNs) to reduce the spatial dimensions of an input volume. It is a form of non-linear down-sampling that serves to make the representation smaller and more manageable, and to reduce the number of parameters and computation in the network. Max pooling operates independently on each depth slice of the input and resizes it spatially. The primary objective of max pooling is to reduce the amount of information in an image while maintaining the essential features necessary for accurate image recognition. This process helps to make the detection of features in input data invariant to scale and orientation changes and also aids in preventing overfitting.
- **Flatten Layer**: This layer is used to achieve the reshaping the output of the previous layer into an 1-D vector. Collapsing all the dimensions into an 1-D array. Flattening the output tensor makes it easier to process the data before their processing on the fully-connected layers, helping in the classification of the data.
- **Batch Normalization Layer**: it is a process to make neural networks faster and more stable through adding extra layers in a deep neural network. The new layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer. A typical neural network is trained using a collected set of input data called batch. Similarly, the normalizing process in batch normalization takes place in batches, not as a single input.
- **Dense Layer**: A dense layer also referred to as a fully connected layer is a layer that is used in the final stages of the neural network. This layer helps in changing the dimensionality of the output from the preceding layer so that the model can easily define the relationship between the values of the data in which the model is working. In any neural network, a dense layer is a layer that is deeply connected with its preceding layer which means the neurons of the layer are connected to every neuron of its preceding layer.
- **Activation Functions**: We use two different activation functions ReLU & Softmax:
  - **ReLU Function**: The rectified linear unit (ReLU) or rectifier activation function introduces the property of non-linearity to a deep learning model and solves the vanishing gradients issue. It interprets the positive part of its argument. It is one of the most popular activation functions in deep learning. The ReLU function is the most commonly used in the CNNs, that's why we use it on our Neural Network.

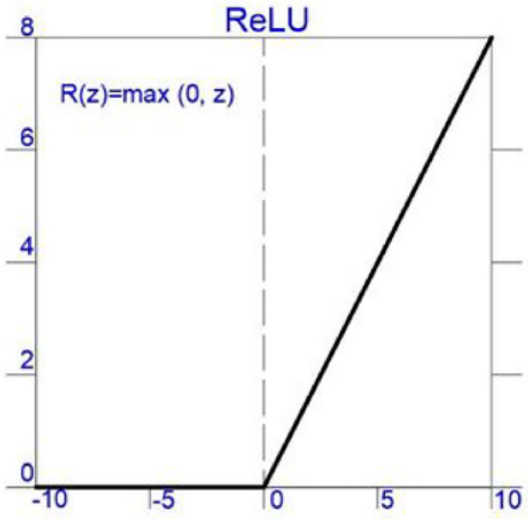

Fig. 3. ReLU Function.

– **Softmax Function**: SoftMax activation function is commonly used in the final layer of neural networks to handle multi-class classification problems. It converts the logits (raw output scores) into probabilities, allowing the network to distribute probability across different classes.

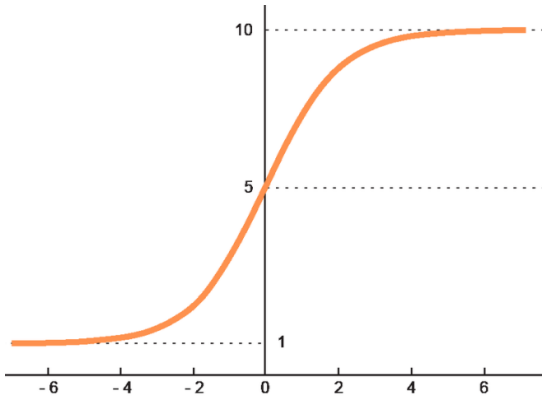

Fig. 4. Softmax Function.

The described model is now presented in the below Figure 5

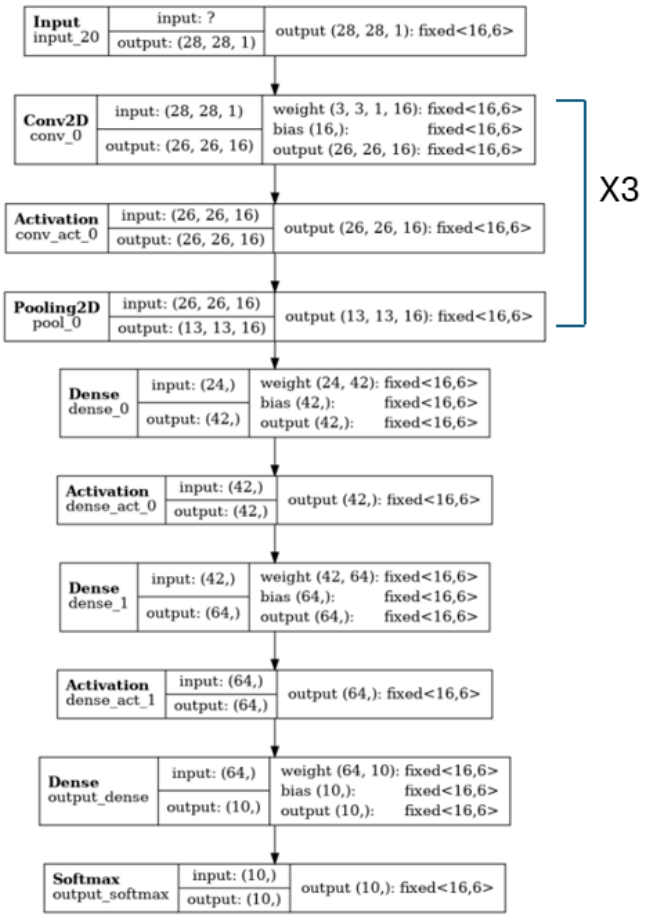

Fig. 5. Overview over the various shapes and precisions used for each layer.

The model has the following setup as seen in the shape above: The Input layer is followed by a convolutional layer(2-D), followed by an activation function(ReLU) and a Max-Pooling layer(2-D), this pattern is repeated for three times which is connected with Dense layer followed by an activation function(ReLU), repeated twice. Finally the later is connected to the output Dense layer, classifying the result in the 10 different classes, followed by a softmax function.

C. Training

After we set the model we proceed with the training. We defined the batch size to 1024 and set epochs to thirty. We specified the loss function, which is categorical crossentropy, commonly used for classification tasks with multiple classes and specified the Adam optimizer with learning_rate set to 0.003 rate. Then we train the model on train data from the MNIST dataset for 30 epochs.

D. Compilation

After training the model we proceed with the compiling. Using the HLS4ML library we convert a trained Keras model into an HLS (High-Level Synthesis) project, which can then be synthesized for deployment on an FPGA. An explanation of what the code we used does follows. Starting, we used the command *hls4ml.utils.config_from_keras_model* we explained in Section II to generate a basic configuration dictionary from the provided Keras model. Then we set the numerical precision for the entire unoptimized model to fixed-point with 16 total bits and 6 integer bits and iterating over all layers in the model we set their strategy to Latency and their reuse factor to 1, aiming for a fully parallel implementation for minimal latency. Additionally, we set HLS4ML configurations like the type of input/output interface to io_stream, which is necessary for CNNs with the command *hls4ml.converters.create_config(backend='Vivado')*. Finally, we use the command *hls4ml.converters.keras_to_hls()* and *hls_model.compile()* to convert the Keras model into an HLS project based on the configurations we defined previous and to compile the HLS project, transforming the model into C++ code ready for FPGA synthesis. For more information about these commands see Section II.

After the compilation we use the command *hls4ml.model.profiling.numerical(model=model, hls_model=hls_model)* for profiling and comparing the numerical performance of the original Keras model and the corresponding HLS model. This method plots the distribution of the weights (and biases) as a box and whisker plot. As we can see in Fig III-D we observe the distribution of (non-zero) weights before compilation and after. The plot shows that there was no difference in the distribution of the non zero weights revealing the compilation was done right. Profiling helps us also with quantization as we can choose the right precision for our model.

E. Synthesis

Following the compilation we continue with the synthesis using the command *hls_model.build()* and the XILINX VI-

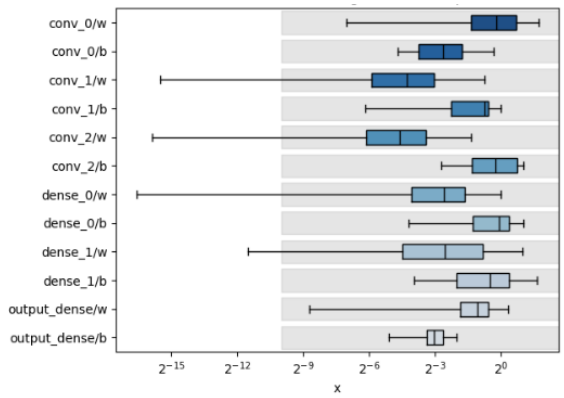

Fig. 6. Distribution of weights before compiling

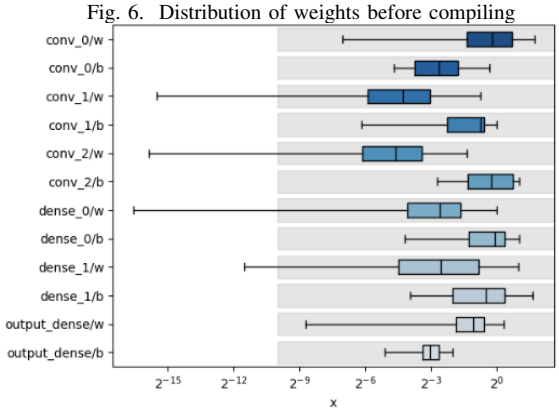

Fig. 7. Distribution of weights after compiling

VADO 2019.1 version. At the beginning, we set the path to the Xilinx Vivado installation and then we build the model with the following options:

- csim=False: Skips the C simulation step.
- synth=True: Performs high-level synthesis (HLS), converting the C++ model code into hardware description language (HDL) code.
- vsynth=True: Performs the Vivado synthesis step, which translates the HDL code into a gate-level netlist, ready for FPGA implementation.

IV. Optimizations

For optimizing our model we have tried pruning and then we also tried quantization using the functions hls4ml library provides. Firstly, we cite some information about pruning and quantization and then we will discuss the methodology we used to optimize our model.

- **Pruning**: Is a technique, that eliminates redundant components in our system, by removing the less important weights in the convolutional and dense layers of our Neural Network. By implementing pruning, low energy usage is achieved as well as lower chip area, because the utilization of the system's resources is reduced. On the other hand, pruning can reduce the accuracy of the neural network, depending on the percentage of the weights that are pruned.
- **Quantization**: Is a technique, which is used to reduce the precision of the numbers that are used to represent data. Quantization can significantly reduce the amount of memory required to store weights and activations, enabling more efficient use of storage elements. Also, lower-power consumption is achieved as less resources are used to implement the design, making it more portable for edge devices. Furthermore, computations can be executed faster as the new arithmetic type is smaller and more compact making the arithmetic operations simpler. Accuracy is expected to drop, as models have reduced bits for operations.

Now let's proceed with the methodology we used to optimize our model. At the beginning, we tried pruning all dense and convolutional layers skipping the output layer because the output layer is the final layer in the neural network where desired predictions are obtained. We pruned the layers gradually from 0 to 50% sparsity every 2 epochs ending by the 10th epoch using the TensorFlow Model Optimization Toolkit. Tensorflow model optimization (TF-MOT) is a library for optimizing machine learning models. We defined a custom function *pruneFunction* that takes as input a layer. The function checks if a layer is a Conv2D or a Dense layer (excluding the output layer) and applies pruning using the *sparsity.keras.prune_low_magnitude* function from TensorFlow Model Optimization Toolkit. Specifically, it prunes the weights of the layer by setting the low-magnitude (i.e., small absolute value) weights to zero. Using the class *PolynomialDecay* we define a schedule to gradually increase the sparsity of a model's weights over a specified period. This technique is used during the pruning process to ensure that the model's performance does not degrade suddenly, but rather adapts to the increasing sparsity gradually. The pruning schedule that we used starts with no pruning and ends with 50% sparsity and finally it starts pruning after 2 epochs and ends pruning after 10 epochs. After pruning the model we examine the accuracy by training the model. As we will discuss further in Section V, the accuracy didn't decrease dramatically.

Additionally, mentioned at the beginning, we also tried the quantization technique. For better results, we created a pruned **and** quantized model using QKeras. We build the model sequentially, adding at the beginning the convolutional layers using the *QConv2DBatchnorm*. QConv2DBatchnorm is used to add a quantized convolutional layer with batch normalization (Batch normalization is a technique used in training deep neural networks to make the training faster and more stable by normalizing the inputs to each layer). With the command *quantized_bits(6,0,alpha=1)* we quantized the weights and biases to **6-bit** precision from 16-bit precision. Then we added a quantized ReLU activation function using QActivation and a max-pooling layer to reduce the spatial dimensions. The same procedure is done for the dense layers that again we quantized the weights to 6-bit precision. Finally, we added to our model an output layer and applied softmax activation to produce class probabilities. After setting the

model we move with the training. As we examined the accuracy loss is negligible for 30 epochs.

V. Results

In this section, we are going to present the results of the optimizations we proposed. We conducted a comparison between the unoptimized model and the two methods, pruning and quantization. To start with, the unoptimized model utilized a total of 64 BRAMs, 2850 DSPs, 31,765 Flip Flops, and 141,367 LUTs. The Utilization percentage (%) is 1 for the BRAMs 23 for the DSPs and 8 for the LUTs. After the unoptimized method we examined the pruning technique. To begin with, the pruned model utilized a total of 64 BRAMs, **1840** DSPs, 27,260 Flip Flops and 88,507 LUTs. The Utilization percentage (%) is now 1 for the BRAMs **14** for the DSPs and 5 for the LUTs. Lastly, we examined the quantized model. The previous model utilized a total of 64 BRAMs, **222** DSPs, 24,617 Flip Flops and 110,807 LUTs. The Utilization percentage (%) is 1 for the BRAMs **1** for the DSPs and 6 for the LUTs.

TABLE I
Resources comparison between the three models

Model\Resources	**BRAM**	**DSP**	**FF**	**LUT**
No Optimization	64	2850	31765	141367
Pruned	64	1840	27260	**88507**
Quantized	64	**222**	**24617**	110807

As we can see from the above table I the quantized method had less DSPs and Flip Flops than the unoptimized and the pruned method. However, the pruned model uses the least LUTs. The optimized models show a 35% reduction in DSP usage for the pruned model and an impressive **92%** reduction for the quantized model compared to the unoptimized model. Additionally, there is a reduction in Flip Flops by 14 % for the pruned model and 22.5% for the quantized model. Last but not least, the LUT usage is reduced by **37%** for the pruned model and 21% for the quantized model. These reductions are depicted in the graph below 8.

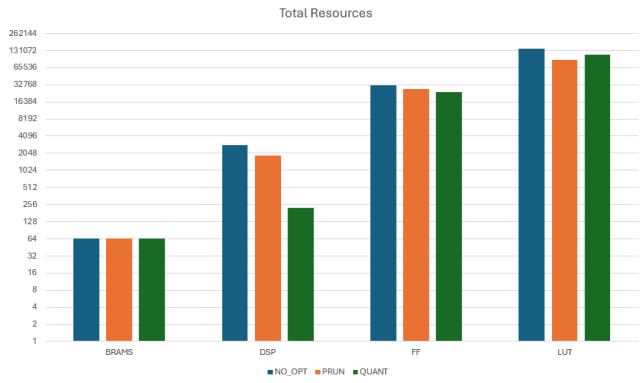

Fig. 8. Total resources for each optimization.

TABLE II
Model accuracy over the epochs

Model\Epochs	**10**	**20**	**30**
No Optimazation	0,9852	0,9876	0,9919
Pruned	0,9797	0,9855	0,9879
Quantized	0,9547	0,9468	0,9603

The graph 8 illustrates the number of resources (BRAM, DSP, FF, LUT) used by each model. The unoptimized model is represented in blue, the pruned model in orange, and the quantized model in green. Significant gains in DSPs can be observed for the quantized model, along with improvements in FFs and LUTs for both optimized models compared to the unoptimized model. However, we should also include the accuracy alongside these improvements. Table II presents the presicion for every model over the number of epochs. As shown in the graph, the accuracy improves as the number of epochs increases. The unoptimized model achieves a notable accuracy of 0.99, while the pruned model has a slightly lower precision at 98.79. Finally, the quantized model achieves an accuracy of 0.96. This means there is a **0.4%** loss in accuracy between the unoptimized and pruned models, and a **3%** loss in accuracy between the unoptimized and quantized models. All these are shown in Graph 9. Again the unoptimized model is represented in blue, the pruned model in orange, and the quantized model in green. For 10 epochs, there is a noticeable difference between the unoptimized and quantized models, but this difference diminishes as the number of epochs increases. The accuracy difference between the unoptimized and pruned models remains minimal. While the unoptimized model achieves a high precision, the pruned and quantized models demonstrate notable reductions in resource usage.

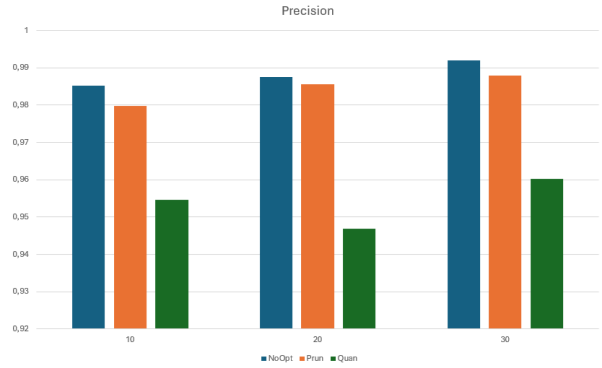

Fig. 9. The precision of the models for 10, 20 and 30 epochs.

VI. Conclusion

In this paper, we tried optimizing the standard benchmark dataset MNIST using the open source software HLS4ML. By applying methods such as pruning and quantization using the functions hls4ml library provides we gain area resources with

insignificant accuracy loss. Experimental results have shown up to a **92%** reduction with negligible accuracy loss.

REFERENCES

[1] T. Aarrestad, V. Loncar, N. Ghielmetti, M. Pierini, S. Summers, J. Ngadiuba, C. Petersson, H. Linander, Y. Iiyama, G. Di Guglielmo, J. Duarte, P. Harris, D. Rankin, S. Jindariani, K. Pedro, N. Tran, M. Liu, E. Kreinar, Z. Wu, and D. Hoang, "Fast convolutional neural networks on fpgas with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 4, p. 045015, Jul. 2021. [Online]. Available: http://dx.doi.org/10.1088/2632-2153/ac0ea1

[2] F. Team, "hls4ml." [Online]. Available: https://github.com/fastmachinelearning/hls4ml

[3] "Welcome to hls4ml's documentation! - hls4ml 0.8.1 documentation," 2024, https://fastmachinelearning.org/hls4ml/.

[4] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.