Converting a large Verilog code effectively to the actual implementation on FPGA using GNN

Biprajit Suklabaidya, Tom Glint , Joycee Mekie
IIT Gandhinagar, Palaj , Gujarat , 382355

***Abstract*—This report presents the objectives, methodology, and benefits of a research project aimed at effectively converting a large Verilog code to the actual implementation on a Field-Programmable Gate Array (FPGA) using Graph Neural Networks (GNN). The motivation behind this research is to address the limitations of High-Level Synthesis (HLS) tools, which takes large amount of time for performance estimation and require intensive feature engineering. By leveraging the representation power of GNNs, the aim is to eliminate the Data Flow Graph (DFG) to Register Transfer Level (RTL) conversion and directly design the circuit based on the HLS front-end Intermediate Representation (IR) graph. This approach can lead to timely performance prediction and improve the efficiency and accuracy of FPGA implementations.**

Keywords:- ***Hardware, Register Transfer Language(RTL) , High Level Synthesis (HLS), Graph Neural Networks(GNN), autoencode, convolution, backpropagation , ResNet, Principal Neighbourhood Aggregation(PNA), Hierarchial GNN .***

INTRODUCTION

In agile hardware development, the focus is on rapid and accurate evaluation of circuit quality starting from the early design stages. High-Level Synthesis (HLS) plays a crucial role in this process by converting high-abstraction level descriptions of a design into Register-Transfer-Level (RTL) descriptions. This conversion involves several steps:

- **Translation to Data Flow Graphs (DFGs):** Programs are translated into DFGs, which serve as an Intermediate Representation (IR) of the design. DFGs represent the flow of data between various operations or modules in the circuit.
- **Resource Allocation and Binding:** Once the DFGs are obtained, resource allocation and binding techniques are applied. Resource allocation involves assigning hardware resources (such as registers, adders, or multiplexers) to different parts of the design. Binding refers to the process of determining the specific implementation of each operation or module in terms of available resources.
- **Transformation into RTL:** After resource allocation and binding, the DFGs are transformed into RTL descriptions. RTL represents the circuit design at a low level, including the specific behavior of individual registers, functional units, and interconnections.

High-Level Synthesis (HLS) can be performed using two main approaches: HLS tools and machine learning (ML)-based approaches.

HLS Tools: HLS tools, such as Xilinx Vivado HLS and Catapult HLS, are commonly used for converting high-level descriptions of a design into RTL descriptions. These tools automate the process by translating programs into Data Flow Graphs (DFGs) as Intermediate Representations (IR). However, one limitation of HLS tools is the significant time required to compute the quality of the actual hardware. This can lead to delays in evaluating the performance and efficiency of the circuit design.

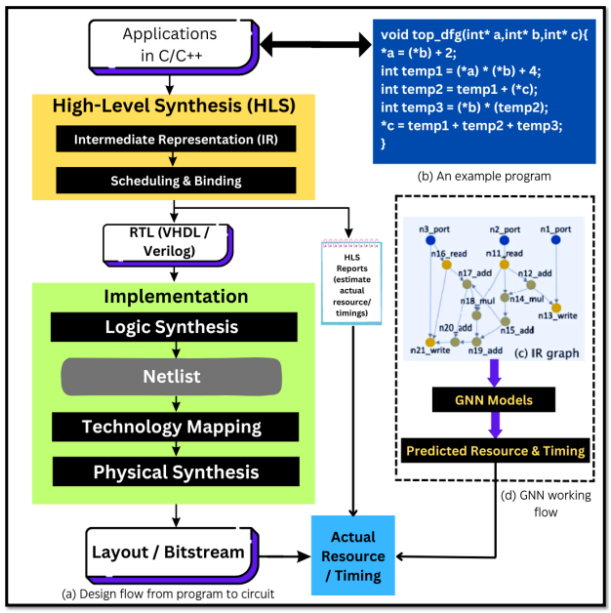

Figure 1: The overall performance prediction flow. (a) Design flow starting from behavioral programs to hardware circuits. (b) An example program written in C. (c) The intermediate representation (IR) graph extracted after front-end compilation. (d) The working flow of GNNs, predicting actual resource usage and timing merely based on raw IR graphs.

ML-Based approaches: Alternatively, ML-based approaches have been explored to enhance the HLS process. Examples include Pyramid and HLSPredict. These approaches leverage machine learning algorithms to predict circuit quality and performance metrics. However, one limitation of ML-based approaches is the need for intensive feature extraction and empirical feature engineering from the hardware design flow.

I. PROPOSED SOLUTION

For the compiled programs, which are often represented as IR (Intermediate Representation) graphs, the representation power of GNNs (Graph Neural Networks) can be exploited to eliminate the DFG-to-RTL conversion and make the quality prediction process quite faster.

II. METHODOLOGY

• Generating IR graphs, right after the HLS front-end compilation are extracted, which can directly predict post-implementation circuit performance metrics.

• The key idea is to generate node embeddings (a vector to represent node features) based on local network neighbours.

• Feeding the GNN model with actual circuit data like 'resource utilization' and 'power consumption' from Vivado HLx or Vitis HLS postimplementation reports (this data will act as the Training dataset).

• Training the GNN model to first learn how to identify the most optimized design among many, based on the provided dataset ; then refining the model, so that it can actually predict design quality performance of any hardware circuit .

III. GNN MODEL

• GNN stands for Graph Neural Network. It is a type of neural network specifically designed to operate on graph-structured data. Graphs consist of nodes (also called vertices) and edges (also called connections) that represent relationships between nodes.

• The key idea behind GNNs is to learn representations (also known as node embeddings) for each node in the graph by aggregating information from its neighboring nodes. This allows the model to capture the local and global dependencies and patterns present in the graph.

The inductive property of GNNs can be used to generalize a model to unseen nodes or even unseen graphs, as model parameters are shared with all nodes.

A. FUNCTIONING OF GNN

Graph Neural Networks (GNNs) operate on graph-structured data and involve several key steps in their process:

Graph Representation:Adjacency Matrix: The graph is represented using an adjacency matrix, which captures the connections and relationships between nodes in the graph.
Node Initialization: In the beginning, each node in the graph is assigned an initial feature vector. These initial features serve as the starting point for the GNN process.

Message Passing takes place in three steps:

1. For each node in the graph, gather all the neighboring node embeddings (or messages).

2. Aggregate all messages via an aggregate function (like sum).

3. All pooled messages are passed through an update function, usually a learned neural network.

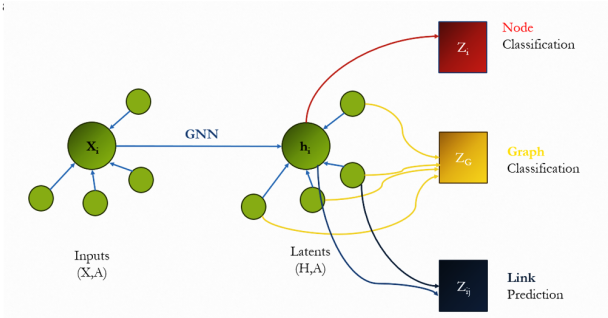

Figure 2: Node classification , Graph classification, Link prediction using latent representation of Graphs.

B. GRAPH AUTOENCODER

• A graph autoencoder is a type of neural network architecture that is specifically designed to encode and decode graph-structured data.
• The goal of the autoencoder is to learn a compressed representation of the input graphs, typically in the form of a lower-dimensional vector, and then reconstruct the original graphs from this compressed representation.

Encoder:The encoder takes an input graph and maps it to a lower-dimensional latent space representation. It applies graph convolutional layers or other graph-based operations to aggregate information from neighboring nodes and captures the structural and relational properties of the graph.
Decoder: The decoder takes the compressed latent representation from the encoder and reconstructs the original graph from it. It performs the inverse operations of the encoder, such as graph convolutional layers or other graph-based operations, to expand the latent code back into a graph structure.

```
#Define the graph autoencoder model

  input_dim = feat_matrices[0].shape[1]
  hidden_dim = 64
 latent_dim = 32

class GraphAutoencoder(nn.Module):
def __init__(self, input_dim, hidden_dim,
↪  latent_dim, num_layers, learning_rate):
    super(GraphAutoencoder, self).__init__()
    self.num_layers = num_layers

    self.layers = nn.ModuleList()
    self.layers.append(GCNConv(input_dim,
    ↪  hidden_dim))
    for _ in range(num_layers - 1):
        self.layers.append(GCNConv(hidden_dim,
        ↪  hidden_dim))
    self.layers.append(GCNConv(hidden_dim,
    ↪  latent_dim))

```

```
        self.fc_hidden = nn.Linear(latent_dim,
        ↪  hidden_dim)
        self.fc_output = nn.Linear(hidden_dim,
        ↪  input_dim)

        self.learning_rate = learning_rate

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        for layer in self.layers:
            x = layer(x, edge_index)
            x = torch.relu(x)

        x = self.fc_hidden(x)
        x = torch.relu(x)
        optimized_features = self.fc_output(x)
        return optimized_features
```

Snippet 1: code for performing convolution using graph autoencoder.

IV. GNN CONVOLUTION

• GNN does not perform convolution in the traditional sense like in convolutional neural networks (CNNs) for image processing.

• The term "convolution" in the context of graph neural networks (GNNs) is used metaphorically to describe the information propagation and aggregation process.

A. *Polynomial filter on Graph*

:

Laplacian Matrix , L = D - A

$$P_w(L) = w_0 I_n + w_1 L + w_2 L^2 + \ldots + w_d L^d = \sum_{i=0}^{d} w_i L^i$$

These polynomials can be thought of as the equivalent of 'filters' in CNNs, and the coefficients w as the weights of the 'filters'. If a feature vector x is constructed , then the convolution with a filter pw can be defined as :

$$x' = p_w(L)x$$

When the degree of the polynomial d is increased and consider the case where instead w1=1 and all of other coefficients are 0., Then x' is just Lx, so :

$$\begin{aligned} x'_v &= (Lx)_v \\ &= \sum_{u \in G} L_{vu} X_u \\ &= \sum_{u \in G} (D_{uv} - A_{vu}) X_u \\ &= D_v X_u - \sum_{u \in \mathbb{N}(v)} x_u \end{aligned}$$

It can be seen the features of node v are combined with the features of its immediate neighbours $u \in \mathbb{N}(v)$.

Effectively , the convolution at node v occurs only with node u which are not more than d hops away. Thus, these polynomial filters are localized. The degree of the localization is governed completely by d.

Consider the graph below with dummy node features:

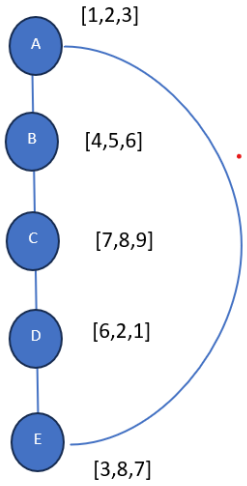

Figure 3: A dummy graph with dummy node features for demostration

The convolution can be performed as :

1. Message Passing: In this step, the target node gathers all the features of the nodes to which it is directly connected.

A : [4,5,6]+[3,8,7] = [7,13,13] (A connected to B and E)
B : [1,2,3]+[7,8,9] = [8,10,12] (B connected to A and C)
C : [4,5,6]+[6,2,1] = [10,7,7] (C connected to C and D)
D : [7,8,9]+[3,8,7] = [10,16,16] (D connected to C and E)
E : [6,2,1]+[1,2,3] = [7,4,4] (E connected to A and D)

Let the weigh matrix be :

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.01 & 0.2 & 0.7 \\ 0.04 & 0.3 & 0.11 \end{bmatrix}$$

2. Weighted sum:

$$\begin{aligned} &A : [7, 13, 13] \cdot w \\ &\Rightarrow [7, 13, 13] \cdot [0.1, 0.2, 0.3] + [7, 13, 13] \cdot [0.4, 0.5, 0.6] \\ &\quad + [7, 13, 13] \cdot [0.01, 0.2, 0.7] + [7, 13, 13] \cdot [0.04, 0.3, 0.11] \\ &\Rightarrow [0.7, 2.6, 3.9] + [2.8, 6.5, 7.8] + [0.07, 2.6, 9.1] + [0.28, 3.9, 1.43] \\ &\Rightarrow [3.85, 15.6, 22.23] \end{aligned}$$

$B : [8, 10, 12] \cdot w$
$\Rightarrow [8, 10, 12] \cdot [0.1, 0.2, 0.3] + [8, 10, 12] \cdot [0.4, 0.5, 0.6]$
$+ [8, 10, 12] \cdot [0.01, 0.2, 0.7] + [8, 10, 12] \cdot [0.04, 0.3, 0.11]$
$\Rightarrow [0.8, 2, 3.6] + [3.2, 5, 7.2] + [0.08, 2, 8.4] + [0.32, 3, 1.32]$
$\Rightarrow [4.4, 12, 20.82]$

And similarly for C,D,E.
3. Aggregation: In this step, the original node features are aggregated with the weighted sum to obtain the new features for each matrix.

A : [1,2,3]+[3.85, 15.6,22.23] = [4.85,17.6, 25.23]
B : [4,5,6]+[4.4,12,20.82] = [8.4,17,26.82]

And similarly for C,D,E.

The aggregation is performed at each layer (hidden layers) and after convolution at each layer activation function is used (ReLU , sigmoid, etc) to capture non-linearity.

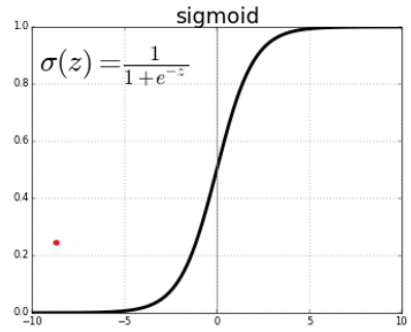

Figure 4: Sigmoid activation function

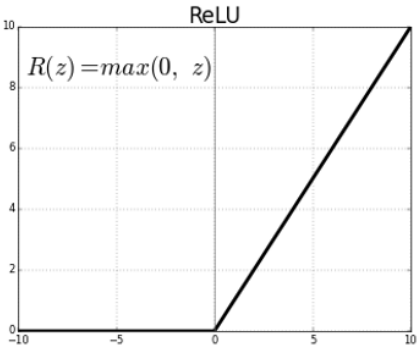

Figure 5: ReLU activation function

V. BACKPROPAGATION

It allows the neural network to adjust its parameters (weights and biases) based on the discrepancy between its predicted output and the desired output. This discrepancy is quantified by a loss function, which measures the error between the predicted output and the target output. The backpropagation takes place in three phases :
1. Forward pass: During the forward pass the input data is fed into the neural network and the activation of each layer are computed sequentially . Starting from the input layer, the activations are propagated through the hidden layers until the output layer is reached.
2. Loss calculation: Once the forward pass is complete , the predicted output is compared with target output using a suitable loss function.
3. Backward pass and optimization: In the backward pass , the gradients of the loss function with respect to the parameters of the network are calculated . The backward pass starts from the output layer and iteratively back propagates the gradients to the input layer , calculating gradients at each layer and then applying **Adam's optimization**.

If the neural network is too deep(the number of layers is too high) there can be a situation where the gradient of loss becomes almost neglegible during backpropagation which results in less or no learning.(**Vanishing graidient).** To overcome this ResNet architecture can be incorporated into GNN.

VI. RESNET

ResNet ses a technique called skip connections.Instead of directly learning the desired mapping, the network learns to approximate the residual mapping between the input and the desired output.

$$H(x) := F(x) + x$$

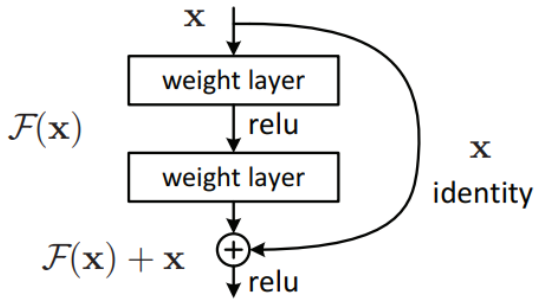

Figure 6: ResNet

VII. PRNINCIPAL NEIGHBOURHOOD AGGREGATION

Graph Neural Networks (GNNs) have been shown to be effective models for different predictive tasks on graph-structured data. The aggregation layers of current GNNs are unable to extract enough information from the nodes' neighbourhoods in a single layer, which limits their expressive power and learning

abilities. Most work in the literature uses only a single aggregation method, with mean, sum and max aggregators being the most used in the state-of-the-art models .This can be formalized in the form of a theorem as :
"In order to discriminate between multisets of size n whose underlying set is R, at least n aggregators are needed" To empirically demonstrate this, four aggregators, namely mean, maximum, minimum and standard deviation are leveraged.
1. Mean Aggregation ($\mu(X^l)$): The most common message aggregator in the literature, wherein each node computes a weighted average or sum of its incoming messages.

$$\mu(X) = \mathbb{E}[X]$$
$$\mu_i(X_l) = \frac{1}{d_i}\sum_{j\in N(i)} X_{lj}$$

2. Maximum and Minimum Aggregations: Also often used in the literature, they are very useful for discrete tasks, for domains where credit assignment is important, and when extrapolating to unseen distributions of graphs.

$$\max(X_l) = \max_{j\in N(i)} X_{lj}$$
$$\min(X_l) = \min_{j\in N(i)} X_{lj}$$

3. Standard Deviation Aggregation: The standard deviation (STD or σ) is used to quantify the spread of neighboring node features, allowing a node to assess the diversity of the signals it receives.

$$\sigma(X) = \sqrt{\mathbb{E}[X^2] - (\mathbb{E}[X])^2}$$

Degree-based scalers can be defined as functions of the number of messages being aggregated (usually the node degree), which are multiplied with the aggregated value to perform either an amplification or an attenuation of the incoming messages. Recent work shows that summation aggregation doesn't generalize well to unseen graphs , especially when larger. One reason is that a small change of the degree will cause the message and gradients to be amplified/attenuated exponentially (a linear amplification at each layer will cause an exponential amplification after multiple layers). A logarithmic amplification

$$S \propto log(d+1)$$

is used to reduce this effect.

$$S_{amp}(d) = \frac{log(d+1)}{\delta}, \delta = \frac{1}{|train|}\sum_{i\in train} log(d_i+1)$$

which can be further be generalized as :

$$S(d,\alpha) = \left(\frac{\log(d+1)}{\delta}\right)^{\alpha}, d>0, -1\le\alpha\le 1$$

α is a variable parameter where $\alpha < 0$ for attenuation, $\alpha > 0$ for amplification, and $\alpha = 0$ for no scaling.

The aggregators and scalers are combined to obtain Principal Neighbourhood Aggregation(PNA).

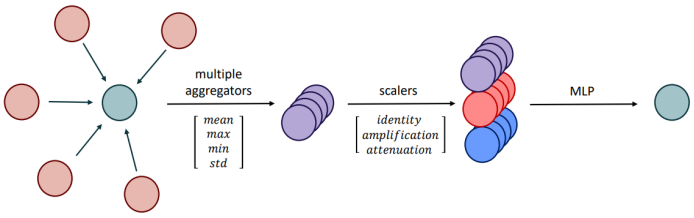

Figure 7: Principal Neignourhood Aggregation(PNA)

$$\oplus = \begin{bmatrix} I \\ S(D,\alpha=1) \\ S(D,\alpha=-1) \end{bmatrix} \otimes \begin{bmatrix} \min \\ \max \\ \sigma \\ \mu \end{bmatrix}$$

VIII. HIERARCHIAL GNN

Many existing GNN models follow a similar flat message-passing principle where information is iteratively passed between adjacent nodes along observed edges and they perform graph convolution, directly aggregate node features from neighbors in the given graph.

Flat message-passing GNNs struggle in capturing dependencies between distant node pairs and they were observed not to benefit from more than a few layers which led to a problem of over-smoothing node representations become indistinguishable when the number of GNN layers increases.

An attributed graph with n nodes can be represented as $G = (V, E, X)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is the node set, $E \subsetneq V \times V$ denotes the set of edges, and $X = \{x_1, x_2, \ldots, x_n\} \in \mathbb{R}^{n\times\pi}$ is the feature matrix. In the feature matrix X, each vector $x_i \in X$ is the feature vector associated with node v_i, and π is the dimension of the input feature vector of each node. Further, V and E are summarized into an adjacency matrix $A \in \{0,1\}^{n\times n}$.

Problem definition: Given a graph G and a pre-defined representation dimension d, the goal is to learn a mapping function $f : G \to Z$, where $Z \in \mathbb{R}^{n\times d}$ and each row $z_i \in Z$ corresponds to the representation of node v_i. The effectiveness of f is evaluated by applying Z to different tasks, including node classification, link prediction, and community detection.

The hierarchical GNN has 4 steps:

1. First, generate a hierarchical structure in which each level is formed as a supergraph.

2. Use the level t graph to update nodes of the level $t+1$ graph (bottom-up propagation: NN1).

3. Apply the typical neighbor aggregation on each level's graph (within-level propagation: NN2).

4. Use the generated node representations from level $2 \leq t \leq T$ to update node representations at level 1 (top-down propagation: NN3) and optimize the model via a task-specific loss.

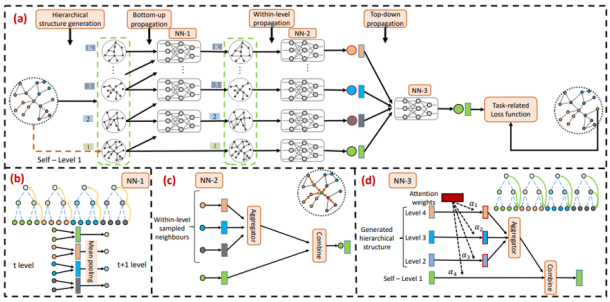

Figure 8: Hierarchial GNN

A. *Hierarchial message*

The hierarchical message-passing mechanism works as a supplementary process to enhance the node representations with long-range interactions and multi-grained semantics. Thus it does not change the flat node representation learning process .To ensure the local information is well maintained. And we adopt the classic GCN, as described, as our default flat GNN encoder throughout the paper. Particularly, the hierarchical message-passing mechanism consists of *l*-th layer consisting of 3 steps.

1. *Bottom-up propagation:* After obtaining node representations ($\mathbf{h}^{(l)}_{s^{t-1}}$) of $\mathcal{G}_{t-1}$, with *l*-th flat information aggregation, we perform bottom-up propagation, NN-1, using node representations in $\mathcal{G}_{t-1}$ to update node representations in $\mathcal{G}_t$ in the hierarchy $\mathcal{H}$ as follows:

$$\mathbf{a}^{(l)}_{s^t_i} = \frac{1}{|s^t_i|+1}\left(\sum_{s^{t-1}\in s^t_i} \mathbf{h}^{(l)}_{s^{t-1}} + \mathbf{h}^{(l-1)}_{s^{t-1}}\right)$$

2. *Within-level propagation*: We explore the typical flat GNN encoders to propagate information within each level's graph $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_T$, i.e., NN-2. The aim is to aggregate neighbors' information and update within-level node representations. Specifically, the information aggregation at level *t* is depicted as follows:

$$\begin{aligned}
\mathbf{m}^{(l)}_a &= \text{AGGREGATE}^N(\widehat{A}^t_{uv}, \mathbf{a}^{(l)}_u | u \in \mathcal{N}^t(v)),\\
\mathbf{m}^{(l)}_v &= \text{AGGREGATE}^N(\widehat{A}^t_{uv} | u \in \mathcal{N}^t(v))\mathbf{a}^{(l)}_v,\\
\mathbf{b}^{(l)}_v &= \text{COMBINE}(\mathbf{m}^{(l)}_v, \mathbf{m}^{(l)}_v)
\end{aligned}$$

3. *Top-down propagation:* The top-down propagation is illustrated by NN3.We use node representations in $\mathcal{G}_2, \ldots, \mathcal{G}_T$ to update the representations of original nodes in $\mathcal{G}$.

$$\mathbf{h}^{(l)}_v = \text{RELU}(W \cdot \text{MEAN}(\alpha_{uv})\mathbf{b}^{(l)}_u), \quad \forall u \in \mathcal{C}(v) \cup \{v\}$$

where α_{uv} is a trainable normalized attention coefficient between node v and super node u (or itself), and MEAN is the element-wise operation. The output node representations of the last layer (L) are obtained via:

$$\mathbf{z}_v = \sigma(W.MEAN\alpha_{uv}\mathbf{b}^{(l)}_u), \quad \forall u \in \mathcal{C}(v) \cup \{v\}$$

IX. EXPERIMENTAL SETUP

During testing, the GNN model was evaluated on 1,113 protein graphs from the TU Dataset. It exhibited a strong ability to capture structural characteristics and accurately predict properties such as secondary structure and protein function. However, the main intention was to apply the model to hardware circuits. To explore this, it was implemented on the MachSuite dataset, consisting of 45 C codes representing diverse hardware circuits. The GNN model showcased promising results, accurately predicting properties like power consumption and timing characteristics. Nevertheless, challenges arose with complex circuit designs and limited data availability, indicating a need for further refinement to enhance performance in such scenarios.

X. RESULTS AND OBSERVATION

The following results and observations can be drawn from the experiment performed:

1. (Layers = 4, Hidden layers = 32, Error = MSE, $\alpha = 0.01$**)**

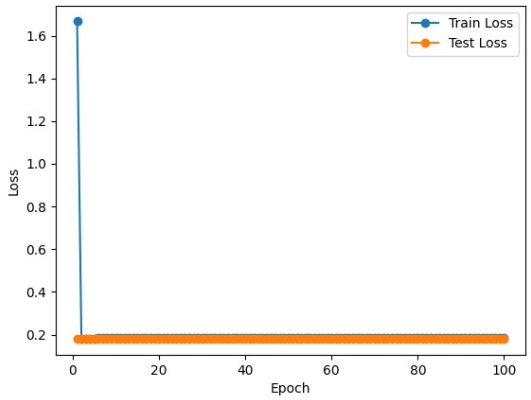

Figure 9: Output - 1

2. (Layers = 4, Hidden layers = 64, Error = MSE, $\alpha = 0.001$**)**

3. (Layers = 7, Hidden layers = 64, Error = MAE, $\alpha = 0.01$**)**

4. (Layers = 7, Hidden layers = 64, Error = RE, $\alpha = 0.01$**)**

5. (Layers = 7, Hidden layers = 64, Error = MAE, $\alpha = 0.001$**, ResNet)**

1. With hidden layers = 64, layers = 4/10/7, learning rate ($\alpha = 0.1/0.01$), and error = MSE, the optimization loss was approximately 18 %.

2. However, for the same parameters, when the Reconstruction Error method or MAE (Mean Absolute Error) was used, the

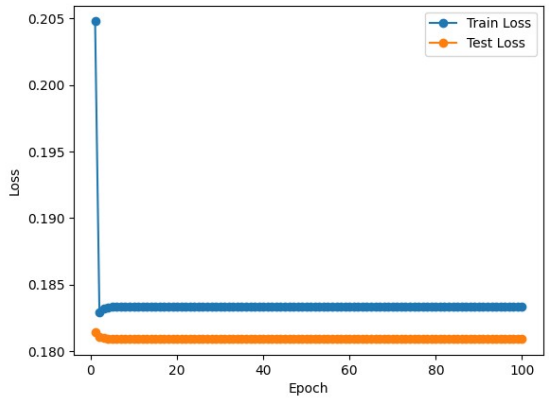

Figure 10: Output - 2

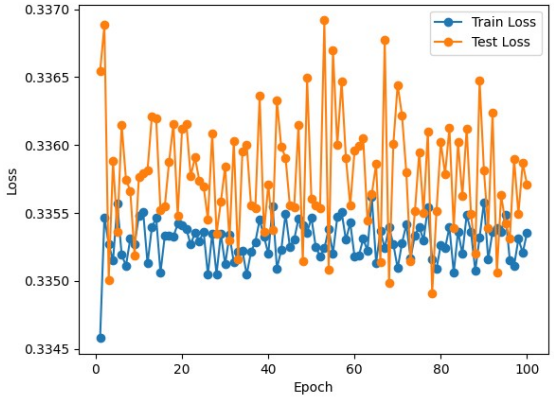

Figure 11: Output - 3

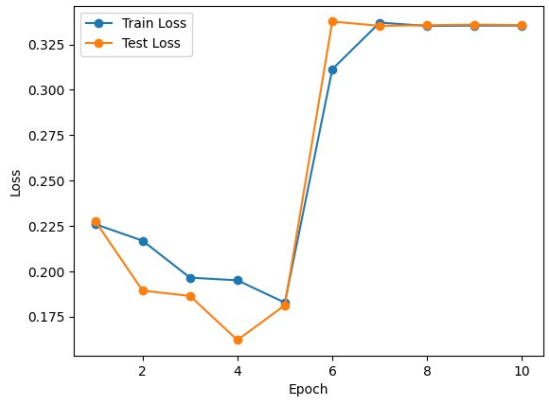

Figure 12: Output - 4

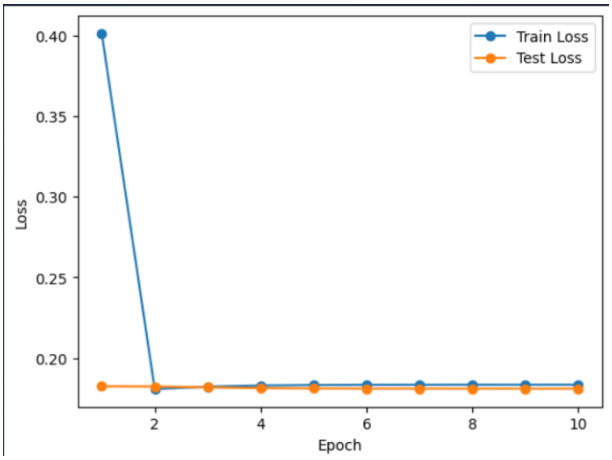

Figure 13: Output - 5: ResNet

optimization loss was approximately 33 %.

3. On implementing the ResNet method for all tested parameters, the optimization loss was approximately 20 % or 18 %.

4. The model was not able to capture all the node features if the number of nodes was larger than approximately 40 %.

5. In some cases, a disconnected graph was generated for connected graphs and vice versa.

XI. FUTURE PROSPECTS

1. We aim to be able to generate a DFG which can give the exact number of DSPs, LUTs, FFs, and other components being used in each node of the Intermediate Representation.
2. Based on this DFG, we aim to modify the current GNN model so that it can predict the performance and quality of any given Hardware Accelerator circuit (the ultimate goal).

XII. CONCLUSION

In conclusion, this project report discusses the application of Graph Neural Networks (GNNs) for effectively converting a large Verilog code into the actual implementation on a Field-Programmable Gate Array (FPGA). The motivation behind this research is to overcome the limitations of High-Level Synthesis (HLS) tools, such as time-consuming performance estimation and intensive feature engineering requirements. By leveraging GNNs, we aim to streamline the process of circuit quality evaluation and accelerate the development of hardware designs. The proposed methodology involves generating Intermediate Representations (IR) graphs from HLS front-end compilation, which are then fed into the GNN model to predict post-implementation circuit performance metrics. This approach eliminates the need for DFG-to-RTL conversion and enables faster quality prediction. The GNN model utilizes message passing and aggregation techniques to capture the structural dependencies within the circuit graph. Furthermore, the use of Principal Neighborhood Aggregations (PNA) enhances the expressive power of the GNN model by combining multiple aggregators and scalers. Through

this research, we have demonstrated the potential of GNNs and machine learning-based approaches in advancing agile hardware development, offering a more efficient and accurate process for circuit evaluation and optimization. Future work could involve exploring additional GNN architectures and further improving the performance prediction accuracy of the model. Overall, this project opens new avenues for leveraging GNNs in the field of hardware design and contributes to the ongoing efforts to enhance the efficiency and productivity of FPGA-based implementations.

XIII. REFERENCES

[1] High-Level Synthesis Performance Prediction using GNNs: Benchmarking, Modelling, and Advancing (Nan Wu, Hang Yang, Yuan Xie, Pan Li, and Cong Hao)

[2] Automated Accelerator Optimization Aided by Graph Neural Networks (Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong)

[3] Principal Neighbourhood Aggregation for Graph Nets (Gabriele Corso, Luca Cavalleri, Dominique Beain, Pietro Liò, Petar Velicković)

[4] Hierarchical Message-Passing Graph Neural Networks (Zhiqiang Zhong, Cheng-Te Li, and Jun Pang)