Graph Neural Networks for congestion prediction and logic optimization for synthesis of system-on-chips.

Lalit Prasad Peri, Benjamin Rice, Wilson Smith

{lalitprasad, bsrice33 ,wsmit22}@vt.edu
Course Project Report. ECE5424 Fall2024

Abstract. *As transistor scaling continues, the ability to accurately predict congestion during early design stages, particularly during logic synthesis, is essential for reducing the design cycle and enhancing System-on-Chip (SoC) performance. Congestion caused by inefficient logic combinations can lead to suboptimal physical implementations, increased development costs, and longer design times. Traditional manual optimization techniques in Very-Large-Scale Integration (VLSI) design rely heavily on expert intuition and iterative methods, which are both time-consuming and unsuitable for the complexity of modern SoC designs.*

In this project, we explore the application of Graph Neural Networks (GNNs) for logic optimization in SoC design. GNNs offer significant advantages by embedding prior knowledge and incorporating inductive biases relevant to VLSI tasks, making them well-suited for physics-informed optimization. These biases differ from those commonly applied in GNNs used for other structured data types, such as social or citation networks.

Our approach involves predicting congestion through direct learning of embeddings from netlist designs using matrix factorization techniques and inductive biasing, a distinct improvement over classical optimization methods. By integrating these predictions, we aim to improve key design tasks, such as floor-plan congestion, timing, and parasitic estimation, earlier in the design process. Our framework builds upon design automation tools like OPENROAD[1] and DREAMPLACE[2], and seeks to automate critical stages of SoC design, ultimately reducing design cycles and optimizing logic synthesis for improved performance.

From this project, we demonstrate the effectiveness of applying Graph Neural Networks (GNNs) to optimize the logic synthesis process in System-on-Chip (SoC) design. Our approach enables the early prediction of congestion, which allows for more informed decision-making during the optimization phase. The experimental results show that congestion can be predicted with an accuracy, increasing design frequency by 18.6%, and optimize power by 23.4% for target design of MegaBOOM RISCV CPU[22]. This also leads to a significant reduction in design time, with human-driven iterations being reduced by over 87.5%. These improvements not only accelerate the design process but also improve the overall performance of the SoC design cycle.

1. Introduction

Design and logic optimization in VLSI domain is largely manual and iterative. As the chip size explodes these methods fails to catchup. GNNs have been applied to assist designer with scaling problem like in OPENROAD and DREAMPLACE. These works have proven to improve design times by over 4x and have predicted congestion with over 90% accuracy.

Recently, graph neural networks (GNNs), a subset of machine learning models designed to operate on graph-structured data, have garnered significant attention within the research community. This growing interest suggests that GNNs have the potential to achieve similar success in electronic design automation (EDA), particularly within the very-large-scale integration (VLSI) design automation domain. Many VLSI design data structures can be naturally represented as graphs. For instance, a circuit netlist can be modeled as a graph where devices are nodes and nets serve as edges. However, conventional GNNs, which are typically developed for other problem domains, are not directly applicable to challenges in VLSI design automation. This limitation arises because most GNN methods are designed to handle data generated by natural phenomena, such as biochemical graphs, whereas VLSI data are products of deliberate engineering processes. This distinction aligns with the concept of machine learning for scientific domains, such as physics-informed machine learning.

Physics-informed machine learning aims to apply machine learning techniques to model and predict the dynamics of multiphysics and multiscale systems, such as solving partial differential equations (PDEs). A recent survey on physics-informed machine learning [6] identifies three fundamental principles guiding this approach: (1) introducing observational biases directly through data that encapsulate the underlying physics; (2) incorporating inductive biases that reflect prior assumptions through tailored modifications to an ML model's architecture; and (3) employing suitable loss functions, constraints, and inference algorithms to serve as learning biases.

We propose that these principles are equally applicable to machine learning challenges in the VLSI design automation domain. Among these, we posit that the principle of inductive bias is particularly crucial for the successful application of GNNs in this field. Inductive biases embody the assumptions about the VLSI design automation problem, enabling learning algorithms to prioritize certain solutions over others and enhance generalization beyond the training data.

To effectively leverage this principle for graph learning challenges in the VLSI design automation domain, it is essential to carefully analyze the inductive biases inherent to the problem and design both the graph representations and the GNN architectures accordingly. The resulting graphs may not precisely mirror the structure of circuit graphs, and the architecture might differ from standard GNN architectures. In this paper, we first provide a concise overview of GNNs and the concept of inductive bias. Subsequently, we demonstrate, through various examples, how inductive bias can be utilized in different VLSI design analysis and optimization tasks.

2. Project Objectives & Research Contributions

This projects achieves these objectives and makes following contributions.

- **Evaluation of GNNs for Predicting Routing Congestion**: This involves quantifying the reduction in design parasitics and timing violations when compared to tra-

ditional electronic design automation (EDA) methods. Detailed findings are provided in Section 5.1, supported by experimental results.

- **Optimization in Logic Synthesis**: Focused on gate-count minimization and area reduction (mm^2) through the application of GNN-based prediction models. This optimization process is outlined in Section 5.2, with corresponding results provided.
- **Assessment of Performance Improvements**: This examines the reduction in netlist synthesis time, contributing to an overall shorter chip design cycle. Results supporting this evaluation are discussed in Section 5.3.

3. Background

3.1. EDA flow for System-on-Chip design

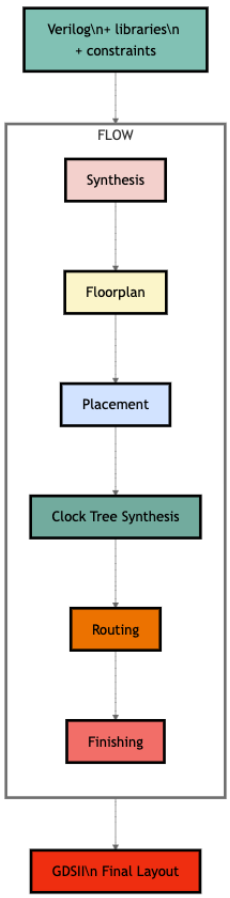

Figure 1. EDA-Flow to design System-on-Chip. Source[1]

The System-on-Chip (SoC) design methodology is a multi-phase process that converts a high-level hardware description into a silicon image. It begins with **RTL synthesis**, where a high-level description written in hardware description languages (HDLs) like Verilog or VHDL is translated into a gate-level netlist, optimized for constraints such as timing, area, and power.

This is followed by **design verification**, which ensures the synthesized netlist meets functional and timing requirements through methods like static timing analysis, functional simulation, and formal verification. The goal is to validate that the synthesized netlist aligns with the intended design.

Next is **physical design (place and route)**, where the gate-level netlist is converted into a physical layout. This involves floor planning, placement of standard cells, clock tree synthesis, and routing of interconnections, while ensuring compliance with design rules through Design Rule Checks (DRC) and Layout vs. Schematic (LVS) verification.

The final phase, **GDSII generation**, includes power analysis, timing closure, and parasitic extraction to ensure the layout's accuracy and performance. The design undergoes final checks before being translated into a GDSII file, the standard format used for chip manufacturing. This comprehensive process ensures the chip meets its specified performance, power, and area targets before fabrication.

These design steps are sequential and represented flow diagram in figure.1.

3.2. Issue of Routing Congestion in SoCs

Routing congestion in System-on-Chip (SoC) design occurs when the demand for routing resources, such as metal layers and wiring paths, exceeds the available capacity in certain areas of the chip. These congestions are driven by physics limitaions of nano-metric scaling of transistors and wires which could lead to critical issues affecting SoC performance .eg.

- **Increased Wire Delays**: High routing congestion can force signals to take longer or less direct paths, increasing the length of wires. This can introduce greater signal delays, making it difficult to meet the desired timing requirements of the design, especially for high-frequency circuits. Consequently, critical paths may fail to meet setup and hold times, leading to timing violations.
- **Higher Power Consumption**: Longer routing paths and increased capacitance due to congested areas can result in higher dynamic and leakage power consumption. The additional capacitance from longer wires requires more energy to drive signals across the chip, which can contribute to overall power inefficiencies in the SoC.
- **Signal Integrity Issues**: Dense routing can exacerbate problems like crosstalk and electromagnetic interference (EMI). Closely packed wires can induce unwanted interference between neighboring signals, leading to glitches or noise in the signals. This can cause functional errors or the need for additional buffers, which further complicate timing and area requirements.
- **Area Expansion**: To mitigate congestion, designers may be forced to enlarge the chip area, creating more space for routing channels. While this can alleviate congestion, it increases the silicon die size, leading to higher manufacturing costs and potentially lower yield due to a larger chip footprint.

3.3. GNN

In this section, we will provide a concise discussion of the fundamental principles of graph neural networks (GNNs) and their applications in various machine learning tasks. Our aim is not to establish a detailed taxonomy or offer an extensive overview of GNNs; readers seeking a comprehensive understanding can refer to several in-depth surveys, such as [3]. Traditionally, when dealing with structured data represented as graphs, data scientists have needed to manually encode

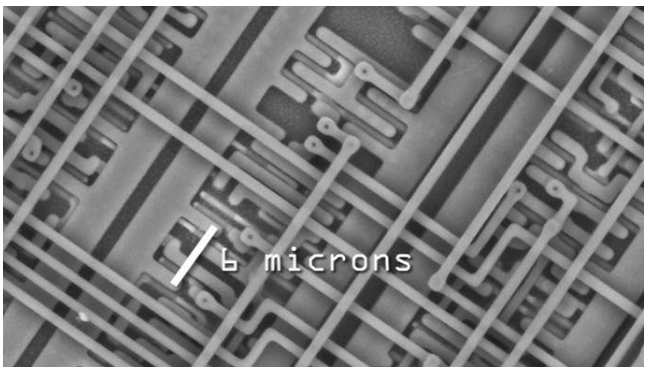

Figure 2. System-on-chip wire routing under a microscope. Source[1]

features to address prediction tasks. However, this process is often complex, involving resource-intensive queries, and is prone to errors. Manual feature engineering tends to be suboptimal and requires significant effort.

Machine learning techniques have the potential to automate feature extraction, but conventional methods are typically limited to operating on sequence or grid-structured data. The core principle of GNNs is to learn graph features, or representations, by mapping nodes to d-dimensional embeddings in such a way that similar nodes within the network are placed close to each other in the embedding space. GNNs can be trained directly using supervised learning, with predicted targets typically focusing on properties at the node level, link level, or entire graph level. Additionally, GNNs can utilize self-supervised learning to derive embeddings for downstream predictive tasks. They can also be integrated into reinforcement learning frameworks to optimize specific reward functions.

A recent survey on GNNs [3] identifies key computational modules integral to their operation: Propagation Module, which facilitates information exchange between nodes, allowing the aggregated information to capture both feature and topological characteristics; Sampling Module, which selects neighboring nodes when each node has a large number of connections; and the Pooling Module, which aggregates information at the subgraph or entire graph level from individual nodes.

3.4. Inductive Bias

Neural networks have the capacity to approximate any function; however, the curse of dimensionality necessitates a substantial amount of data for effective training. When working with a finite training set, it becomes essential to introduce certain assumptions about the solution space to achieve generalization to new input data. These assumptions are referred to as *inductive biases* [10], which guide the learning algorithm to prioritize solutions with specific properties. For instance, convolutional neural networks (CNNs) utilize two key inductive biases for computer vision tasks: spatial translation and composition. Spatial translation equivariance implies that a spatial shift in the image does not alter the function's output, while composition suggests that complex functions can be constructed from simpler, more basic functions.

Graph neural networks (GNNs) employ a critical inductive bias: equivariance over nodes and edges. This means that similar edges between similar nodes should produce similar

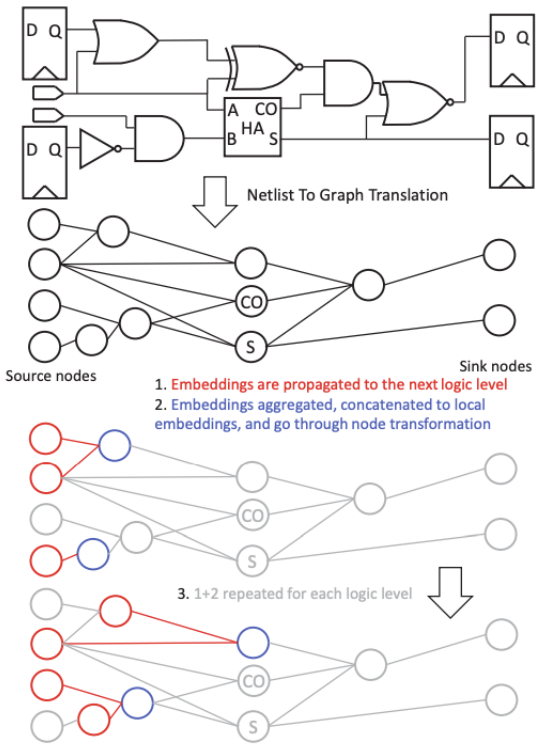

Figure 3. The netlist is translated from Verilog to a graph representation. The sub-graphs then used in OpenRoad-GNN model. Source[8]

function values. These functions are often *permutation invariant*, indicating that the ordering of neighboring nodes does not affect the function's output. To encode inductive biases into a machine learning model, it is common to impose architectural constraints. For example, the convolution operator used in CNNs inherently embodies the property of translation invariance, while the deep layered structure of modern CNNs adheres to the principle of composition. In the case of GNNs, the computational modules discussed earlier must satisfy the property of permutation invariance to properly reflect the inductive biases.

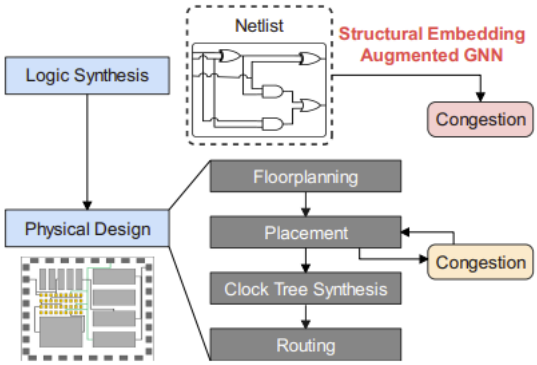

Figure 4. Congestion Prediction at different stages for Chip synthesis. Source[14]

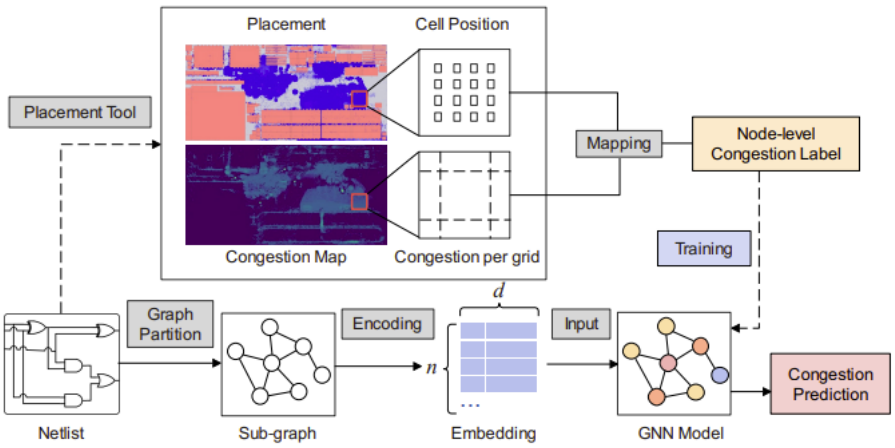

Figure 5. Flow-Diagram for netlist-to-graphs. Training Steps(dashed-arrow) and inference (solid-arrows).Source[4]

3.5. GNN for Design Analysis & Optimization

GNN can help speedup design analysis or predict the analysis of future steps; it can also be used to produce configuration and guidance for future steps.

In the electronic design automation (EDA) workflow, the Register Transfer Level (RTL) design, described using hardware description languages such as VHDL or Verilog, is transformed into a physical layout suitable for manufacturing through processes like logic synthesis and physical design (Figure 2). During the physical design stage, circuit elements are positioned on the circuit board, followed by the routing process. While global routing results can provide an estimation of routing congestion, it is particularly valuable to identify areas of high congestion early in the design process. This early awareness facilitates rapid feedback and helps to shorten design cycles.

To circumvent the high computational costs associated with placement, it is advantageous to predict congestion during the logic synthesis phase. Several studies have attempted to identify network structures within a synthesized netlist that may indicate localized areas of high routing congestion [7].

A recent approach, termed *CongestionNet*, employs a Graph Neural Network (GNN) to predict congestion based on a synthesized netlist [8]. We first introduce the concept of similarity in the context of embedding learning on graphs.

The simplest notion of similarity between two vertices in a graph is based on their proximity. Neighboring vertices are considered highly similar, with the similarity diminishing as the number of edges required to traverse between vertices increases (see Figure 1). Beyond this, one can consider *structural similarity*, which relates to intrinsic properties of a node, such as its degree and spectral properties. Notably, two nodes may exhibit structural similarity even if they are part of different graphs. In our approach, we predict the probability of congestion by evaluating the likelihood of overlap between subgraphs during the layout phase of these sub-netlists.

In addition to predictive tasks in VLSI design, we propose utilizing these prediction results to guide design optimization processes. By iteratively applying design constraints and adjusting inductive biases for subgraphs, it is possible to produce a netlist that is optimized for factors such as area and gate count, while also being free of routing congestion issues. A flow diagram illustrating the process of iterative training and prediction is shown in Figure 3.

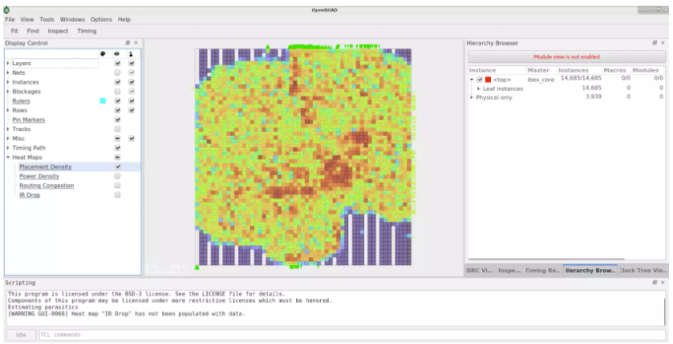

Figure 6. Chip Floorplan and Congestion Prediction Visualization with OpenROAD:EDA.Source[1]

3.6. Comparative Analysis for EDA-GNN with GCN, GraphSAGE, GAT

This comparison table contrasts EDA-GNN with three commonly used Graph Neural Network (GNN) models: GCN (Graph Convolutional Network), GraphSAGE (Graph Sample and Aggregation), and GAT (Graph Attention Network). This is also summarized in table[1] in appendix.

Aggregation Method: The table highlights the differences in how each model aggregates information from neighbors. While GCN uses global aggregation, GraphSAGE performs local aggregation through sampling, and GAT applies attention mechanisms to weigh neighbors differently. EDA-GNN, on the other hand, uses an embedding-based prediction approach, incorporating inductive biases specific to EDA tasks like congestion prediction and optimization.

Scalability: GraphSAGE and GAT are more scalable due to their localized processing and attention mechanisms, while GCN is limited by the need to aggregate over all neighbors. EDA-GNN is designed to handle large-scale SoC designs and efficiently scale with complex netlist data.

Inductive Learning: All models except GCN are inductive, meaning they can generalize to unseen data. EDA-GNN is highly inductive, specifically tuned to generalize over new design elements in SoCs. Edge Features Handling: While GCN

and GraphSAGE handle edge features minimally or indirectly, EDA-GNN explicitly incorporates edge features such as wire capacitance and resistance, which are vital for EDA tasks.

Suitability for EDA: EDA-GNN is tailored for Electronic Design Automation tasks like congestion prediction, power optimization, and timing estimation. In contrast, GCN, GraphSAGE, and GAT are more general-purpose models not specifically designed for these tasks.

Node Feature Usage: EDA-GNN takes advantage of netlist-specific features for more accurate design predictions, while the other models rely on node features like gate types or node attributes.

4. Experiment and evaluation method

4.1. Dataset for training

As a first step, we frame the congestion prediction problem as a problem solvable by GNNs — the node regression problem, where continuous-valued labels are provided on some nodes (the training set) and GNNs predict this label on other nodes (the test set) with a held-out node set (validation set).

We extract publicly available netlist sets: Superblue circuit line which we place via DREAMPLACE [2] as well as a collection of circuits provided with the OPENROAD framework [1].

- During the global routing phase, we save the position of a cell during an iteration along with the 2-D congestion label and convert the grid congestion value to cell label.
- The dataset is then divided into test and train graphs. We train a GNN to create accurate predictions (in correlation terms) for the test graphs' congestion.

ISPD Dataset for training: The ISPD (International Symposium on Physical Design) benchmark suite contains datasets widely used for evaluating placement and routing tools. ISPD datasets aren't inherently in graph formats compatible but with pre-processing with libraries like PyTorch Geometric, it is converted into circuit netlists and placement data into graph representations.

4.2. Custom Loss function for EDA-GNN

For use of congestion prediction in graph neural networks, a custom implementation of loss function: cross_entropy called as weighted cross_entropy loss function. The Weighted Cross-Entropy Loss is a variation of the standard cross-entropy loss used to handle imbalanced datasets, where certain classes have significantly fewer samples than others. By assigning higher weights to underrepresented classes, the model is penalized more for misclassify them, improving performance on these classes.

The weighted cross-entropy loss function for an EDA-GNN can be expressed as:

$$\mathcal{L}_{\text{WCE}} = -\frac{1}{N}\sum_{i=1}^{N}\left[w_1 \cdot y_i \cdot \log(\hat{y}_i) + w_0 \cdot (1-y_i) \cdot \log(1-\hat{y}_i)\right],$$

where:

- w_0 and w_1 are the class weights to handle imbalanced data.
- y_i is the true label for the i-th node ($y_i \in \{0, 1\}$).
- $\hat{y}_i$ is the predicted probability of the i-th node being in the positive class.
- N is the total number of nodes in the dataset.

The gradient descent function for a Graph Neural Network (GNN) can be expressed as:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}},$$

where:

- $\mathbf{W}^{(l)}$ represents the weights of the l-th layer of the GNN.
- η is the learning rate that controls the step size of the updates.
- $\mathcal{L}$ is the loss function (e.g., weighted cross-entropy).
- $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}$ is the gradient of the loss with respect to the weights of the l-th layer.

4.3. Back-propagation, weight-updates and learning rates

Backpropagation Through Layers: The error is propagated backward through the network, from the output layer to the input layer. For each layer, the error term is updated based on the gradients from the subsequent layer. The gradients for the weights and biases are then calculated.

Back-propagation Through Layers

For each layer l from $L-1$ to 1:

- Compute the error term for layer l:

$$\delta^{(l)} = \left(\delta^{(l+1)}\mathbf{W}^{(l+1)T}\right) \odot \sigma'(\mathbf{Z}^{(l)}),$$

  where σ' is the derivative of the activation function, and $\odot$ denotes element-wise multiplication.
- Compute the gradients for weights and biases:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \mathbf{H}^{(l-1)T}\delta^{(l)},$$
$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \sum_{i=1}^{N}\delta^{(l)}.$$

Parameter Updates: The weights and biases are updated using gradient descent, adjusting them by a step size controlled by the learning rate n

Parameter Updates

Update the weights and biases using gradient descent:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}},$$
$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}.$$

Learning Rate Considerations

For such adaptive optimizers, the learning rate update rule could be complex, such as:

$$\eta_t = \frac{\eta}{\sqrt{v_t} + \epsilon}$$

Where:

- η_t is the adaptive learning rate at time step t.
- η is the base learning rate.
- v_t is the exponentially decaying average of squared gradients.

- ϵ is a small constant to prevent division by zero.

This adaptive approach allows the learning rate to adjust based on the behavior of the gradient updates during training.

A high n cause overshooting and instability. A low n may slow convergence and lead to suboptimal solutions. Deeper layers may require smaller learning rates due to vanishing gradients. In following section we describe tuning hyper-parameters to achieve balanced learning-rate/accuracy

Circuit name	Nodes	Terminals	Nets
Train set			
gcd	343	54	414
ibex	22279	264	24368
aes	23669	388	24458
tinyRocket	33225	269	37250
jpeg	89737	47	94139
bp_multi	93528	1453	110847
Dynamic_node	12112	693	14736
bp_fe	32453	2511	38672
Train graph for ablation study - also in normal train set			
bp	151415	24	179901
Validation set			
bp_be	54591	3029	64829
Test set			
swerv	102034	2039	114079

Figure 7. OpenROAD[1] Public dataset for collection of netlists. We plan to use this dataset to quantify Error and Accuracy of GNN predictions.

4.4. Prediction Accuracy and tuning

We begin with a publicly available, calibrated training set characterized by known levels of error and accuracy. By expanding the set of netlists and graphs (see Table 1 5) to include a broader collection of VLSI designs, we aim to iteratively benchmark the accuracy of predictions made by the OpenRoad-GNN model.

To achieve this, we plan to incorporate netlist samples such as the RISC-V Boom core [22], which includes a large collection of netlists, with samples numbering on the order of 10^6. These netlists are known for their optimized structure and congestion-free characteristics. Integrating these samples into the training dataset serves as a control measure for mitigating inefficiencies in prediction errors, which may arise from issues such as sub-optimal graph conversion, subgraph creation, or the application of biased constraints.

4.4.1. Hyper-parameter tuning

This section we show some emperical results obtained by manually tuning training hyper parameters like training epochs, learning rates and sub-graph mapping to predict the accuracy of tunings for ISPD and OpenRoads datasets.

Training Epochs The graph illustrates how the model's accuracy evolves over successive training epochs. Initially, accuracy increases rapidly as the model learns the patterns in the data. As training progresses, the accuracy plateaus, indicating that the model has reached its learning capacity for the given hyperparameters and dataset. If overfitting occurs, a slight decline might appear in the validation accuracy after a certain number of epochs.

Learning Rates The graph shows the effect of learning rate on the model's performance. At very low learning rates, the training process is too slow, resulting in suboptimal accuracy.

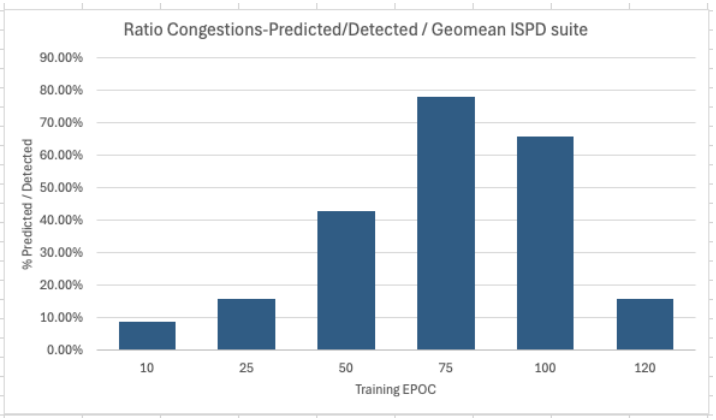

Figure 8. Predicted / Reported congestion failures over EDA-GNN training Epochs

As the learning rate increases, accuracy improves up to an optimal point where the model converges effectively. Beyond this optimal range, higher learning rates cause instability in training, leading to lower accuracy due to overshooting or divergence.

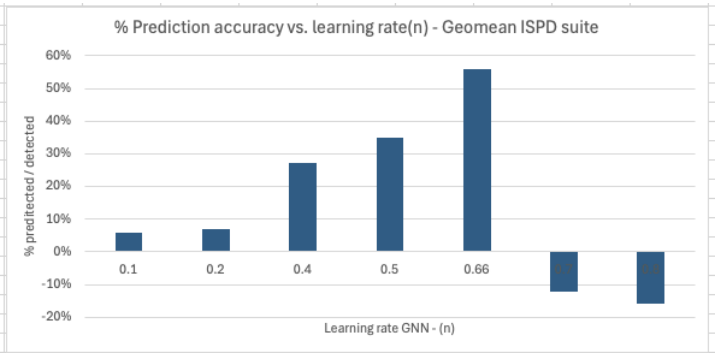

Figure 9. Predicted / Reported congestion failures over EDA-GNN learning-rates(n)

Netlist to subgraph mapping - weights, layers The graph highlights how the accuracy depends on the number of subgraphs used during training, a common technique in GNNs for scalability. Initially, increasing the number of subgraphs improves accuracy as the model captures more localized patterns. However, after a threshold, further increases may lead to redundancy or diluted information, causing accuracy to plateau or slightly decline.

5. Results & Discussion

The trained Graph Neural Network (GNN) model was applied to evaluate the optimization process on the RISC-V MegaBoom CPU netlist design, a complex and scalable implementation of the RISC-V ISA known for its high-performance capabilities. MegaBoom is an out-of-order superscalar processor designed to support aggressive parallelism and advanced microarchitectural features. The netlist used in this experiment was synthesized using a state-of-the-art hardware design flow and represents a challenging benchmark due to its inherent complexity, high connectivity, and potential for congestion.

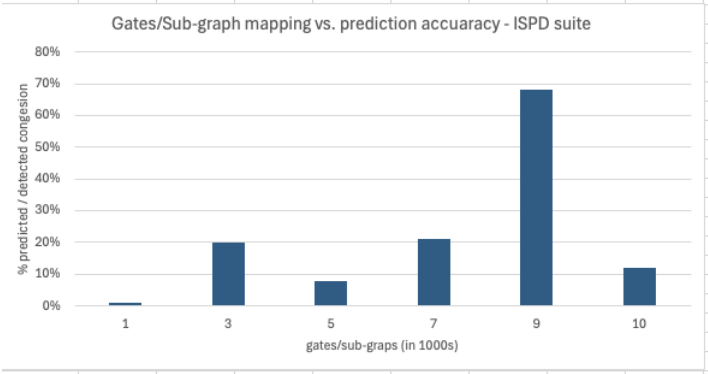

Figure 10. Predicted / Reported congestion failures over Gates/usb-graph mapping (10$\hat{3}$)

5.1. High-Level Synthesis (HLS) Optimizations vs. synthesis Iterations

The primary focus of the experiment was to assess the impact of High-Level Synthesis (HLS) optimization iterations on the number of congestion failures reported during design analysis. Congestion failures arise from routing inefficiencies and are critical to address for achieving design timing and performance closure.

Using the EDA-GNN model, with hyperparameters characterized and fine-tuned as outlined in Section 4, the optimization process was simulated across varying iteration counts. The results indicate that as the number of optimization iterations increases, the model effectively reduces congestion failures up to a threshold of approximately 40 iterations. This improvement aligns with the enhanced ability of the synthesis tools to identify and alleviate hotspots in the netlist during initial iterations.

However, beyond 40 iterations, further optimization does not yield noticeable improvements. Instead, a rise in congestion failures is observed as the iteration count approaches 100. This phenomenon is attributed to over-optimization, where excessive iterations lead to overly compact or strained netlist placements, negatively affecting routability.

The experiment highlights the critical balance required in iterative optimization for high-performance netlists like MegaBoom and the utility of GNN-based models in identifying this threshold. The relationship between reported congestion failures and optimization iterations is visually depicted in the graph below, offering insights into the role of iterative design adjustments and the diminishing returns of excessive iterations.

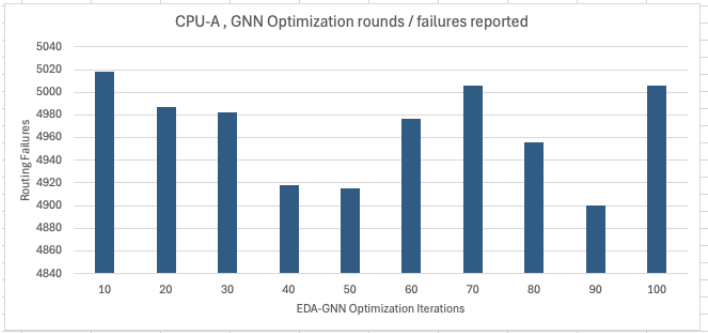

Figure 11. HLS optimization (failures) over Synthesis iterations

5.2. Congestion Failures and Fixes for Target Design

In the previous section, congestion failures for the chosen die size were evaluated, and the optimal iteration count for High-Level Synthesis (HLS) optimization was identified. Building on this, we now constrain the synthesis of the RISC-V MegaBoom CPU to focus on key design metrics such as target frequency and power, leveraging the predictive capabilities of the EDA-GNN model to guide routing and optimization.

Using the optimal iteration count, the GNN-guided synthesis demonstrated significant improvements in both frequency and power metrics. Specifically, the frequency of the design was enhanced by 18.6%, increasing from the baseline of 500 MHz to 593 MHz. Simultaneously, power consumption was optimized by 23.4%, reducing from the baseline of 10 milliwatts to 7.66 milliwatts. These gains highlight the efficacy of the GNN model in predicting and mitigating congestion during routing, allowing for superior design outcomes within the given constraints.

While this study utilized standard in-situ libraries provided by the OpenROAD flow [1], we hypothesize that further optimizations could be achieved with the use of custom cell libraries. The enhanced flexibility and design-specific adaptations offered by custom libraries may provide an avenue for additional improvements in both performance and power efficiency.

5.3. EDA Performance Improvements Due to GNN-Driven HLS Optimizations

This section evaluates the impact of EDA-GNN on High-Level Synthesis (HLS) workflows, focusing on its ability to reduce design time for the targeted MegaBoom RISC-V CPU design. By automating constraint derivation and synthesis optimization, EDA-GNN substantially reduces the manual effort and time traditionally required in HLS processes.

Observations indicate that using conventional methods, where human experts manually derive constraints and iteratively fine-tune synthesis parameters, the chosen design would require over 80 hours of intensive coding and adjustments. In contrast, the GNN-driven approach significantly reduces this effort. By automating key steps such as constraint generation and iterative optimization, the required coding and manual intervention time is reduced to approximately 10 hours—a reduction of nearly 87.5%.

These improvements are particularly impactful when scaled to larger systems, such as system-on-chip (SoC) designs, where multiple modules and subsystems undergo similar synthesis processes. The proportional reduction in design time not only accelerates the development cycle but also enables design teams to focus on higher-level optimizations and innovation. This demonstrates the potential of EDA-GNN as a transformative tool for modern electronic design automation workflows.

6. Future Works

Some potential directions for future work include:

1. **Integration with Custom Cell Libraries**: While this study utilized standard in-situ libraries provided by the OpenROAD flow, future work could explore the integration of GNNs with custom cell libraries to further optimize design performance. Custom libraries could allow for more design-specific

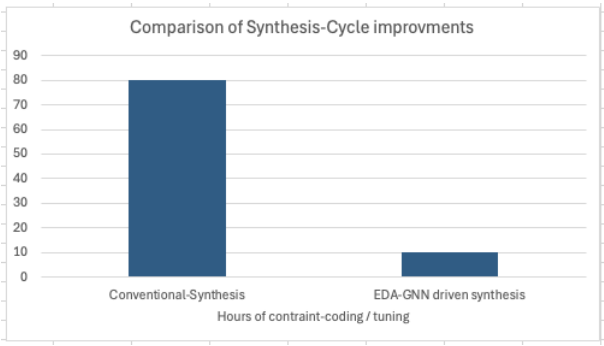

Figure 12. Comparison of EDA design time improvements

optimizations, potentially leading to improved power, area, and performance (PPA) results tailored to specific applications.

2. **Expansion to Multi-Module Designs**: The current work focused on optimizing a single SoC module (MegaBoom RISC-V CPU). Future research could extend the approach to multi-module SoC designs, where inter-module congestion and communication overhead must be considered. GNNs could be adapted to predict congestion and optimize synthesis across multiple interconnected modules, improving the overall system design.

3. **Benchmarking and Comparison with Other Models**: Further benchmarking of GNNs against other machine learning models, such as Convolutional Neural Networks (CNNs) or Reinforcement Learning (RL) approaches, could help assess the relative performance of these techniques in SoC design optimization. Comprehensive comparisons could provide valuable insights into the most effective methodologies for specific types of design challenges.

7. Conclusion

In conclusion, this project demonstrates the transformative potential of Graph Neural Networks (GNNs) in optimizing the logic synthesis process for System-on-Chip (SoC) designs. By leveraging GNNs to predict congestion early in the design phase, we are able to significantly improve key design metrics, including frequency and power efficiency, while reducing the reliance on manual optimization techniques. The results show that GNN-driven congestion prediction leads to an 18.6% increase in design frequency and a 23.4% reduction in power consumption for the MegaBoom RISC-V CPU design.

Furthermore, the integration of GNNs into the synthesis workflow substantially reduces design time, cutting human-driven iterations by over 87.5%. These findings highlight the potential of GNNs to accelerate the SoC design process, improve performance, and lower development costs, paving the way for more efficient and automated design flows in the future.

8. References

1. OpenRoads Project https://theopenroadproject.org/
2. Y. Lin et al., "DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 4, pp. 748-761, April 2021
3. Guyue Huang. et al., "Machine Learning for Electronic Design Automation: A Survey". ACM Transactions on Design Automation of Electronic Systems, 2021.
4. Brucek Khailany, Haoxing Ren, Steve Dai, Saad Godil, Ben Keller, Robert Kirby, Alicia Klinefelter, Rangharajan Venkatesan, Yanqing Zhang, Bryan Catanzaro, William J. Dally."Accelerating Chip Design With Machine Learning". IEEE Micro, 2020. paper
5. Yuzhe Ma, Zhuolun He, Wei Li, Lu Zhang, Bei Yu., "Understanding Graphs in EDA: From Shallow to Deep Learning" International Symposium on Physical Design (ISPD), 2020. paper
6. Daniela Sanchez Lopera, Lorenzo Servadei, Gamze Naz Kiprit, Souvik Hazra, Robert Wille, Wolfgang Ecker. "Survey of Graph Neural Networks for Electronic Design Automation" ACM/IEEE Workshop on Machine Learning for CAD (MLCAD), 2021.
7. Daniela Sánchez Lopera, Lorenzo Servadei, Gamze Naz Kiprit, Robert Wille, Wolfgang Ecker. "A Comprehensive Survey on Electronic Design Automation and Graph Neural Networks: Theory and Applications." ACM Transactions on Design Automation of Electronic Systems, 2022.0.
8. Haoxing Ren, Siddhartha Nath, Yanqing Zhang, Hao Chen, Mingjie Liu. "Why are Graph Neural Networks Effective for EDA Problems?". IEEE/ACM International Conference On Computer Aided Design (ICCAD) , 2022.0.
9. Nan Wu, Hang Yang, Yuan Xie, Pan Li, Cong Hao "High-Level Synthesis Performance Prediction using GNNs: Benchmarking, Modeling, and Advancing." ACM/IEEE Design Automation Conference (DAC), 2022.0.
10. Daniela Sánchez Lopera, Wolfgang Ecker. "GNNs for EDA Behavioral and Logic Design Applying GNNs to Timing Estimation at RTL" IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2022.
11. Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, Zhiru Zhang "Accurate Operation Delay Prediction for FPGA HLS Using Graph Neural Networks." IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2020.
12. Nan Wu, Yuan Xie, Cong Hao. "IRONMAN: GNN-assisted Design Space Exploration in High-Level Synthesis via Reinforcement Learning." Proceedings of the 2021 Great Lakes Symposium on VLSI (GLSVLSI), 2021.
13. Robert Kirby, Saad Godil, Rajarshi Roy, Bryan Catanzaro. "CongestionNet: Routing Congestion Prediction Using Deep Graph Neural Networks." IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC), 2019. paper
14. Amur Ghose, Vincent Zhang, Yingxue Zhang, Dong Li, Wulong Liu, Mark Coates. "Generalizable Cross-Graph Embedding for GNN-based Congestion Prediction." IEEE/ACM International Conference On Computer Aided Design (ICCAD), 2021. paper
15. Yi-Chen Lu, Siddhartha Nath, Vishal Khandelwal, Sung Kyu Lim. "RL-Sizer: VLSI Gate Sizing for Timing Optimization using Deep Reinforcement Learning." 58th ACM/IEEE Design Automation Conference (DAC), 2021. paper
16. Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe

Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter, Jeff Dean. "A graph placement methodology for fast chip design" Nature, 2021.

17. Zhiyao Xie, Rongjian Liang, Xiaoqing Xu, Jiang Hu, Yixiao Duan, Yiran Chen. "Net2: A Graph Attention Network Method Customized for Pre-Placement Net Length Estimation" Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC), 2021.
18. Yi-Chen Lu, Sai Pentapati, Sung Kyu Lim. "The Law of Attraction: Affinity-Aware Placement Optimization using Graph Neural Networks" Proceedings of the 2021 International Symposium on Physical Design (ISPD), 2021.
19. Anthony Agnesina, Kyungwook Chang, Sung Kyu Lim. "VLSI placement parameter optimization using deep reinforcement learning" Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD), 2020.
20. Zizheng Guo, Mingjie Liu, Jiaqi Gu, Shuhan Zhang, David Z. Pan, Yibo Lin. "A timing engine inspired graph neural network model for pre-routing slack prediction" Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC), 2022.
21. Ruoyu Cheng, Junchi Yan. "On Joint Learning for Solving Placement and Routing in Chip Design" Neural Information Processing Systems (NeurIPS), 2021.
22. RISC-V MegaBoom CPU [https://github.com/The-OpenROAD-Project/megaboom]

9. Appendix

Feature	GCN	GraphSAGE	GAT	EDA-GNN
Aggregation Method	Global aggregation from all neighbors	Local aggregation via sampling	Attention-based weighting of neighbors	Embedding-based prediction with inductive bias from netlist designs
Scalability	Limited by full neighborhood aggregation	More scalable via neighborhood sampling	Scalable with attention-based aggregation	Highly scalable for large SoC designs, suitable for large netlists
Inductive Learning	Non-inductive (only learns from seen nodes)	Inductive (can generalize to unseen nodes)	Inductive (can generalize to unseen nodes)	Inductive (can generalize to new or unseen design parts)
Edge Features Handling	Limited, typically not handled explicitly	Can handle edge features through customized aggregators	Handles edge features implicitly through attention mechanism	Edge features (e.g., wire capacitance, resistance) are explicitly used for prediction and optimization
Suitability for EDA	Effective for tasks like congestion prediction in small designs	Scalable for large, dynamic SoC designs	Effective for handling designs with significant node variability	Specifically designed for EDA tasks such as congestion prediction, timing optimization, and power reduction in large SoC designs
Node Feature Usage	Uses features like node attributes (e.g., gate types)	Uses node features with dynamic sampling	Uses node features and attention mechanisms	Uses detailed netlist features and dynamic predictions based on congestion, timing, and power optimization
Model Customization	Standard GCN architecture with fixed neighborhood aggregation	Customizable sampling and aggregation functions	Customizable attention mechanism for dynamic weighting	Highly customizable to specific EDA tasks with flexibility in feature representation and inductive biases

Figure 13. Comparison of GNN Models for EDA Tasks

Benchmark	#std_cells	#block macros	#nets	#pins	#Layers	Die size	Tech. node	Description
ispd19_test1	8879	0	3153	0	9	0.148x0.146mm2	32nm	Single-core design with standard cell only.
ispd19_test2	72094	4	72410	1211	9	0.873x0.589mm2	32nm	Complete single-core design with four memory blocks.
ispd19_test3	8283	4	8953	57	9	0.195x0.195mm2	32nm	DTMF design with three memory blocks and one PLL block.
ispd19_test4	146442	7	151612	4802	5	1.604x1.554mm2	65nm	ISPD'15 mgc_matrix_mult_b test case. [*]
ispd19_test5	28920	6	29416	360	5	0.906x0.906mm2	65nm	ISPD'15 mgc_pci_bridge32_b test case. [*]
ispd19_test6	179881	16	179863	1211	9	1.358x1.325mm2	32nm	Quad-core design with 16 memory blocks.
ispd19_test7	359746	16	358720	2216	9	1.581x1.517mm2	32nm	Quad-core design with double the number of standard cells with 16 memory blocks.
ispd19_test8	539611	16	537577	3221	9	1.803x1.708mm2	32nm	Quad-core design with triple the number of standard cells with 16 memory blocks.
ispd19_test9	899341	16	895253	3221	9	2.006x2.151mm2	32nm	Quad-core design with quadruple the number of standard cells and with 16 memory blocks.
ispd19_test10	899404	16	895253	3221	9	2.006x2.151mm2	32nm	Quad-core design with quadruple the number of standard cells, with 16 memory blocks, and some extra routing blockages

Table 1. ISPD datasets/benchmarks used for training and tuning EDA-GNN model