Why are Graph Neural Networks Effective for EDA Problems? *(Invited Paper)*

Haoxing Ren, Siddhartha Nath, Yanqing Zhang, Hao Chen* and Mingjie Liu*
*NVIDIA Corporation, *University of Texas at Austin*
{haoxingr, siddharthan, yanqingz}@nvidia.com , * {haoc, jay_liu}@utexas.edu

***Abstract*—In this paper, we discuss the source of effectiveness of Graph Neural Networks (GNNs) in EDA, particularly in the VLSI design automation domain. We argue that the effectiveness comes from the fact that GNNs implicitly embed the prior knowledge and inductive biases associated with given VLSI tasks, which is one of the three approaches to make a learning algorithm physics-informed. These inductive biases are different to those common used in GNNs designed for other structured data, such as social networks and citation networks. We will illustrate this principle with several recent GNN examples in the VLSI domain, including predictive tasks such as switching activity prediction, timing prediction, parasitics prediction, layout symmetry prediction, as well as optimization tasks such as gate sizing and macro and cell transistor placement. We will also discuss the challenges of applications of GNN and the opportunity of applying self-supervised learning techniques with GNN for VLSI optimization.**

I. Introduction

Graph is a data structure represented by nodes and edges. It can be used to model the relationships between a set of objects. Recently, deep learning models on graphs, i.e. graph neural networks (GNNs) have been receiving a lot of attention in the research community. GNNs have been applied to various domains such as social networks, knowledge graphs, bio-medicine, etc. Many different GNN architectures [1] have been proposed to perform learning on graphs.

Although GNNs achieved great success in these domains, it is unclear why would GNNs achieve similar success in EDA, particularly in the VLSI design automation domain. Although VLSI design data structures can often be represented as graphs, e.g. a circuit netlist can be treated as a graph with devices as nodes and nets as edges, conventional GNNs developed for other problem domains do not necessarily solve the VLSI design automation problems directly. This is because most GNN methods are proposed to deal with data generated by natural phenomenons such as biochemical graphs, while the VLSI data are the results of deliberated engineering processes. However, we have seen a surge of successful GNN applications in the VLSI design automation domain recently [2]–[5], what makes GNN so effective for VLSI design automation problems?

To answer this question, we need to study a leading machine learning method for scientific d omain: p hysics i nformed machine learning. This method applies machine learning to model and forecasts the dynamics of multiphysics and multiscale systems, for example, solving partial differential equations (PDEs). In a recent survey paper on physics informed machine learning [6], the authors described three main principles to conduct physics informed machine learning: 1. introduce observational biases directly through data that embody the underlying physics; 2. incorporate inductive biases corresponding to prior assumptions by tailored interventions to an ML model architecture; and 3. introduce appropriate choice of loss functions, constraints and inference algorithms as learning biases.

We argue that similar principles can be applied to machine learning problems in the VLSI design automation domain. Out of these three principles, we believe the inductive bias principle is what makes GNN applications in the VLSI design automation domain successful. These inductive biases represent the assumptions about the VLSI design automation problems, which help the learning algorithms to prioritize one solution over another and improve the generalization beyond training data.

Therefore to utilize this principle for graph learning problems on the VLSI design automation domain, one should carefully analyze the inductive bias of the problem domain, and construct the graphs as well as GNN architecture according to that inductive bias. The graphs constructed do not necessarily represent exactly circuit graphs, and the architecture might be different to commonly used GNN architectures.

In this paper, we will first give a brief introduction on GNNs and the inductive bias. Then we will illustrate with examples how the inductive bias is used in various VLSI design analysis and optimization tasks. We will also discuss the challenges of using GNNs and give our thoughts on future directions of applying GNNs to VLSI design automation problems.

II. Background

A. Graph neural networks

In this section, we will briefly discuss the fundamental principles of GNN and its use in machine learning tasks. We will not attempt to define its taxonomy or give a broader coverage on GNN. Interested readers can find several comprehensive GNN surveys such as [1].

For structured data represented as a graph, traditionally data scientists need to hand encode features to solve prediction problems. However, hand encoding graph features is complex and involves expensive queries, they are more error prone, suboptimal and labor intensive. Deep learning can help learn features automatically, but conventional deep learning techniques only operate on sequence or grid. The main principle of GNN is to learn graph features, i.e. representation, to map nodes to d-dimensional embeddings such that similar nodes in the network are embedded close together.

Graph presentation can be trained directly with supervised learning, often the predicted targets are properties at node-level, link-level or the entire graph-level. It can also be trained with self-supervised learning to learn an embedding for downstream predictive tasks. It can further be used in a reinforcement learning framework to optimize some reward functions.

Key computational modules of GNN according to a recent GNN survey [1] include Propagation Module, which propagates information between nodes so that the aggregated information could capture both feature and topology information; Sampling Module, which samples neighboring nodes if there are a large number of neighbors for each node; and Pooling Module, which collects subgraph or entire graph level information from nodes.

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10.
https://doi.org/10.1145/3508352.3561093

Popular GNN architectures include GCN [7], GraphSage [8], GAT [9], etc. These models differ in their computational module and sampling module choices. For example, the propagation module of GAT is an attention-based spatial convolution operator, while that of GCN is a spectral convolution operator. GraphSage samples a fixed-size set of neighbors from each node, while GCN and GAT collect all neighbors from each node.

B. Inductive Bias

Neural Network is capable of fitting all possible functions, but the curse of dimensionality requires a large amount of data for training. Given a finite training set, we have to rely on some assumptions about the solution space to generalize to new input data. These assumptions are called inductive biases [10], which encourage the learning algorithm to prioritise solutions with certain proprieties. For example, convolutional neural networks (CNN) leverages two key inductive biases for computer vision tasks: spatial translation equivariance and composition. Spatial translation equivariance indicates shifting the image spatially will not change the function value. Composition means complicated functions can be composed with simpler ones.

GNN leverages a key inductive bias: equivariance over nodes and edges. Similar edges between similar nodes should have similar function values. These functions often are permutation invariant, i.e. the order of neighboring nodes does not change the function value.

To encode inductive biases in the machine learning model, one often chooses to impose an architectural constraint on the models. For example, the translation invariance property is built in the convolution operator used in CNN, and the deep architecture of modern CNNs follows the composition property. For GNN, the computational modules described earlier should satisfy the permutation invariance property.

III. GNN for Design Analysis

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 Section III-A, GNN is used to predict switching activity given a netlist and its input stimulus; in Section III-B, GNN is used to predict layout parasites from its schematic design; and in Section III-C, GNN is used to create layout symmetry constraints for analogy circuits; and in Section III-D, GNN is used to predict post routing arrival and slack timing information from placement results.

A. Switching Activity Prediction

Given some stimuli, the resulting amount and frequency of switching activity that occurs in each gate instance in a design netlist is often valuable information that designers need to perform evaluation and analysis in applications such as power analysis, power and timing aware physical synthesis, IR-drop analysis, and signal integrity timing. Often, the industry standard SAIF (Switching Activity Interchange Format) file is used to store switching activity information. With large designs and/or numerous stimuli, designers may encounter a dilemma to attaining switching activity information in a fast and accurate manner. Gate-level simulation is widely considered the most accurate way to attain switching activity, where vector-based stimuli return exact switching counts that correspond to the input. However, gate-level simulations are very slow, typically ranging from 1-1000 cycles/s [3], [11].

An alternative solution to attaining switching activity is to model switching behavior so as to maneuver around the gate-level simulation step, thus being able to attain switching activity much faster. The drawback of traditional models, such as statistical activity propagation [3], [12], [13], is poor accuracy. Sometimes, these traditional models are referred to as switching activity estimators (SAE). [3] shows an experiment where accuracy in a traditional SAE can be poor, leading to an average of 31.4% error in average power across its benchmarks. Although SAEs are fast, they are inaccurate. SAEs run into issues such as difficulty in modeling the effects of signal correlation and reconvergence in the logic cone. It is very difficult to derive an expression or method that is comprehensive of all the possible relationships and interactions between input signal waveform sequences and Boolean logic functions of each gate over a variable amount of time (the simulation duration or window), and propagate those interactions through the entire gate-level netlist. In practice, SAEs are often used in *vectorless* cases where only the inputs' activity factors are known and propagated through the netlist.

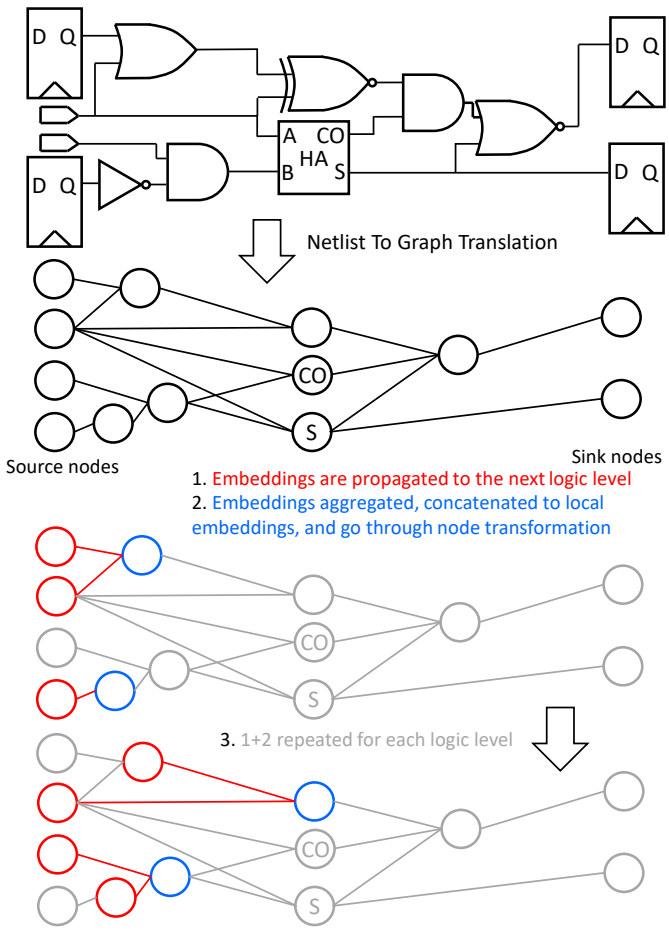

Fig. 1. In GRANNITE [3], the netlist is translated from Verilog to a graph representation. The graph then becomes part of a DAG-GNN model.

GRANNITE [3] aims to enable fast and accurate switching activity prediction through constructing a supervised learning-based SAE utilizing a DAG-GNN model architecture(Figure 1. The DAG-GNN model is a good choice, since many aspects of the model's design correlates well with how switching patterns are actually determined and propagated in ground truth. At the heart of the model, a DAG-GNN layer propagates node embeddings logic-stage-by-stage, in the same fashion signals are propagated from one logic stage to the next in reality. This tailored intervention of the model architecture helps it to understand the causal nature of switching activity within logic cones, while the propagated embeddings make use of the equivariance

over nodes and edges inductive bias–the switching conditions of each gate should partially be determined by previous states in the gate's fanin logic cone. To combat logic reconvergence issues, innate statistical properties of each combinational gate's *input pin as well as* Boolean function are annotated as graph node and edge features that are concatenated to the propagated embeddings in a fashion similar to GraphSAGE [8]. This type of Propagation Module, as well as incorporating prior knowledge of Boolean functions and their interaction with each unique input pin, which is vital in determining switching activity, further steers the inductive bias to understand different input pins' unique functionality. Lastly, the signal correlation issue is helped by using input features that reflect not only the input ports' switching activity, but also the order of their sequence. The input features are mapped to high dimension space and become the initial propagated embeddings in the DAG-GNN at the input port nodes. These model design decisions, centered around the key DAG-GNN layer, aid the model to accomplish its goal of learning a fast and accurate SAE. The function of the DAG-GNN model is an approximation of the complex, non-linear relationship between input switching activity, Boolean logic, netlist structure, and combinational gate switching activity. GRANNITE [3] achieves an average error of 5.3% when predicting average power (calculated from switching activity) and greater than 18.7× speedup across its benchmark experiments.

B. Parasitics Prediction

Layout parasitics has significant impact on the performance of analog circuit. It often is the cause of slow convergences between schematic and layout designs. Circuit designers typically estimate parasitics from past experience when designing schematic, resulting in variability between designers and potential inaccuracy. Machine learning provides a viable approach to predict layout parasitics based on an existing layout dataset.

The inductive bias of this parasitics prediction problem is the invariance of layout parasitics of schematic structure, i.e. similar schematic structure produces similar layout parasitics. There are numerous circuit topologies and transistor configurations that commonly recur within various applications. These circuit topologies often share similar layout structures, which result in similar layout parasitics. Prior work [14] proposes manual schematic features such as maximal transistor series (MTS), a term indicating whether transistors are sharing source or drain diffusion, and estimate layout parasitics with linear regression over these manual features. GNN can directly learn from the raw features and the circuit topology without manual feature engineering.

To capture the structure invariance of the circuit topology, the GNN design needs to be able to distinguish different schematic circuit structures. In Paragraph [15], a heterogeneous graph is constructed from the schematic. As shown in Figure 2, devices (transistor, resistor, capacitor, etc.) as well as nets connecting these devices are mapped to graph nodes. Mapping nets into graph nodes allows the model to predict net parasitics, it also helps to reduce model complexity for high fanout nets. Since the electrical properties of each node and edge are different, e.g. device node is different to net node, diffusion-to-net edge is different to gate-to-net edge, there are different node types and edge types in the graph. The resulting graph is a heterogeneous graph.

To learn on this heterogeneous graph, [15] proposes to leverage the principles of three popular GNN models: GraphSage [8], Relational GCN (RGCN) [16] and Graph Attention Network (GAT) [9]. The propagation module works like this: on each node, it first applies

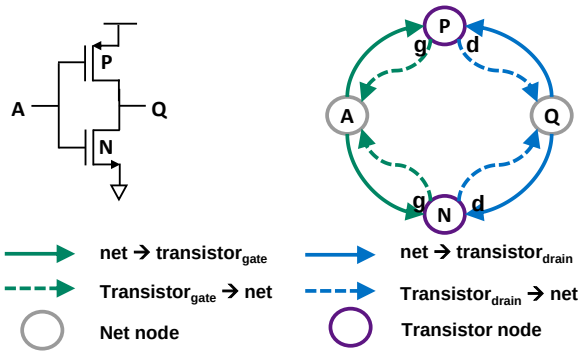

Fig. 2. Inverter Schematic and Heterogeneous Graph in [15]

GAT attention among edges of the same edge types to create an embedding for each particular edge type; then the embeddings of different edge types are combined together with a weighted sum to form the embedding of neighboring nodes, which is concatenated with its own embedding to compute the next embedding for this node.

Trained on a large dataset of industrial circuits, the model achieves an average prediction R^2 of 0.772 (110% better than XGBoost) and reduces average simulation errors from over 100% with designer's manual estimation to less than 10%.

C. Layout Symmetry Prediction

During the layout process for analog/mixed-signal (AMS) and custom circuits, various geometrical constraints (e.g., symmetry, common-centroid, array) often need to be applied for better post-layout circuit performance [17]. Among these constraints, the concept of symmetry and its variants are among the most essential and widely adopted. Given different constraints, place-and-route (P&R) algorithms can generate significantly diverse final layout solutions. Typically, the constraint set is carefully examined and manually annotated by layout experts, which leads to a tedious and time-consuming process. Therefore, automatic constraint annotation has been actively researched on automated custom layout synthesis.

Early automatic symmetry constraint annotation efforts leverage sensitivity analysis to find matching pairs [18], [19]. These methods are generalizable to various circuit types and performance metrics; however, they suffer from extensive simulations, thus inefficient. As circuit netlist can be described as graphs naturally, graph-based methods, including pattern matching [20], graph similarity [21], and also GNN-based approaches [4], [22], [23], have been investigated for better efficiency. These methods all rely on a inductive bias: circuit objects (e.g., devices, building blocks) that need matching consist of related circuit sub-structures and neighboring local connections.

In practice, symmetry constraints are often applied to a pair of circuit objects with homogeneous functional purposes. For example, symmetry constraints can be applied to building blocks with identical functionality, such as pairs of digital-to-analog converters (DACs). Symmetry constraints can also apply to transistors with similar properties, such as the differential pair in an operational transconductance amplifier (OTA).

In [4], circuit netlists are modeled as heterogeneous graphs. Different from Paragraph [15], the proposed heterogeneous graph model has one node type (i.e., device node) and four edge types, as shown in Figure 3. To cope with a large set of unlabeled circuits and make the GNN model generalizable, [4] performs unsupervised training on heterogeneous graphs. The propagation module includes gated graph

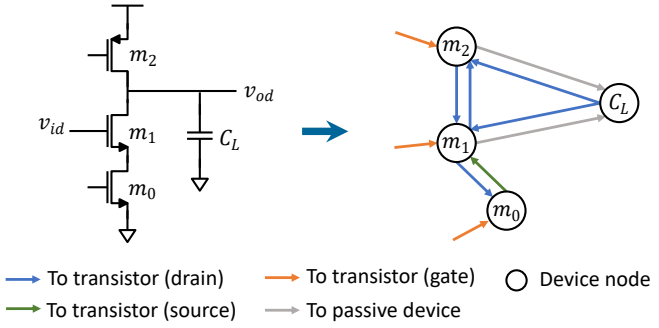

Fig. 3. Example of the heterogeneous graph circuit representation in [4].

convolution layers [24] to handle neighboring feature aggregation considering different edge types. A sampling module that samples a set of non-neighboring nodes for each node is also included. The unsupervised loss is calculated as the cross-entropy of the neighbor embedding similarity of non-neighbor embedding discrepancy. By iteratively minimizing the overall loss, the GNN model learns a strategy to improve neighboring nodes' embedding similarity and extend the discrepancies between each node and its non-neighboring samples. Intuitively, the GNN model will recognize the localized connection and the peripheral structure of each node.

With the final embeddings of each device node, the pooling module of [4] uses the PageRank algorithm to decide the top representative nodes of each subcircuit as circuit embeddings. Lastly, by calculating the cosine similarity of the embeddings of each potential matching pair, a symmetry constraint will be added if the similarity of the two embeddings is greater than a pre-defined threshold.

Trained on a diverse circuit dataset, the unsupervised GNN learning-based framework in [4] achieves an average 15.8% F_1-score improvement on system-level constraint extraction and 218× speedup compared with the previous approach based on spectral methods. For device-level constraint extraction, [4] achieves an average 9.8% improvement in F_1-score with subsecond runtime.

D. Static Timing Prediction

Fast and accurate timing prediction is essential for timing-driven placement since accurate timing information is only available after routing. Placement solutions could significantly effect the downstream routing performance, but most state-of-the-art analytical placers optimize the half-perimeter wirelength as the surrogate for design quality. Thus a timing prediction model that could both effectively predict the routing behavior and replace the static timing analysis (STA) engine, would greatly reduce the runtime overhead, and further enable timing driven optimization early during placement.

Early works [25], [26] have proposed a variety of heuristics and wire load models to form methods of approximation to consider the routing-induced parasitic effects towards routing. Recent work [27] have also proposed a differentiable wire delay model using rectilinear steiner tree generated from pin connections as a fast predictor for routing results. The work of [28] develop a random forest tree model for predicting local net delay and slew, based on the extracted net features from placement results. Net^2 [29] presents a graph attention model for estimating the routed net length based on pre-placement results. Despite the above prior work, the work of [5] is the first work to present an end-to-end timing prediction model, effectively predicting the global post routing timing metrics from placement results without additional STA. In contrast with [28], the work of [5] improves the local net delay/slew prediction model generalization by using GNNs to replace hand-crafted features, and further leverage a DAG-GNN architecture (similar to [3] in Sec.III-A) on the underlying timing graph to replace STA.

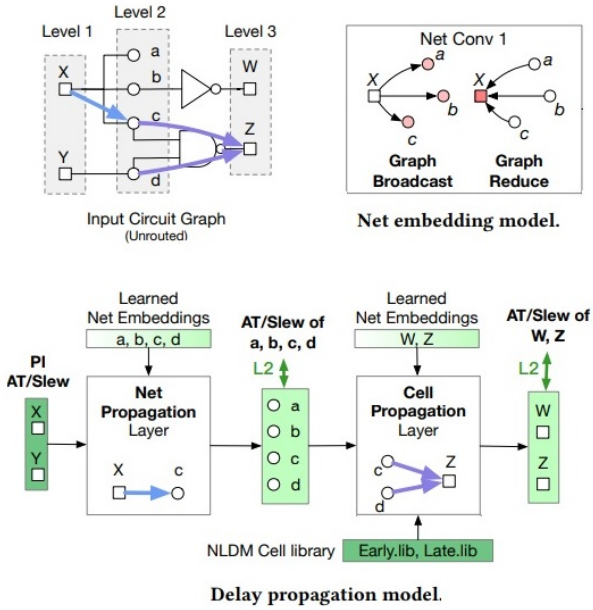

Fig. 4. Net embedding model and delay propagation model in [5].

The model in [5] consists of two key components, a net embedding model and a delay propagation model as shown in Figure 4. These two models are analogous to modern STA engines that splits the timing analysis into two steps, where local net delay and loads are calculated, and then the arrival time and slack are propagated on the timing graph. The circuit is represented as heterogeneous graphs, where the logic gates pins are represented as nodes in the graphs, with input (PI) and output (PO) being different node types. The graph edges are thus heterogeneous and directed acyclic (DAG), consisting of net edges (connecting PO with PI) and cell edges (connecting PI with PO). The graph is thus analogous to the timing graphs used in modern STA tools and effectively leverage the inductive bias of the problem. The underlying graph representation is also different with the work of [29] where nets are nodes in the graphs and might be less suitable for STA prediction. The net embedding model is a GNN operating on generated DAG to generate node embeddings on logic gate pins of the circuit, which is efficient in encoding neighboring circuit level information to each pin. The embedding GNN model removes additional feature engineering in [28] while predicting for local net delay and slew on the graph edges. The delay propagation model is a DAG-GNN that uses neural networks as function approximators to learn the delay calculation and propagation of STA. The net propagation layer operates on net edge types to propagate information to the PO pins information such as net delay and net slew. The cell propagation layer operates on the cell edge types and use neural networks to model the cell delay lookup and arrival/slack timing propagation calculations. Both the graph representation and GNN models are carefully designed to mimic the timing graph and delay computation in traditional STA engines, thus efficiently leveraging the inductive bias of static timing prediction.

The model is trained and tested on open-sourced circuit designs with timing reports generated from open sourced tools and PDK. The net embedding model when used in predicting local net delay

and slew shows improved model generalization when compared with feature-based random forest model [28]. This demonstrate that using GNNs on graph structure can extract useful circuit features in predicting local net delays. The global timing metrics such as arrival time and slack predicted at timing endpoints show strong correlation with the labeled data, with a testing R^2 score of 0.8957. The DAG-GNN propagation model effectively leverages the problem structure and inductive graph bias on timing graphs, where a deep GNN baseline show poor testing results since it does not effectively learn useful information when removing the inductive bias introduced in the graph representation and GNN model.

IV. GNN in Design Optimization

In addition to VLSI design predictive tasks, GNN can also be used to VLSI design optimization tasks. The inductive bias of each design optimization task can be leveraged to design GNNs to generalize better. GNN can be used together with reinforcement learning or supervised learning frameworks for optimization. In Section IV-A and IV-B, GNN is used together with RL to optimize placement and gate sizing. In Section IV-B, GNN is directly used as the backbone to predict optimized gate sizes in a supervised learning setup.

A. Placement

Placement is a key physical design task. Placement happens at different scales, e.g. macro placement, cell placement, and transistor placement. The inductive bias of placement task is that similar circuits should have similar placements. This can be demonstrated by the popular force directed placement algorithm. This algorithm models the circuit connectivity as a graph: cells are modeled as vertices and nets are modeled as edges. The objective of placement is to minimize the total squared Euclidean distance between any two movable cells i and j as $(x_i - x_j)^2 + (y_i - y_j)^2$. This is equivalent to modeling nets as springs and calculating the state of equilibrium. Given the same circuit connectivity and IO boundary locations, the state of equilibrium should be the same. Therefore, if we make physical locations as attributes in the graph, the placement is invariant to its graph structure. Additional constraints such as non-overlapping constraints also influence placement significantly, but these additional constraints are a function of physical locations and cell sizes, which can also be represented in the graph.

Recent works have shown the power of GNN in performing placement at macro [30] and transistor [31] levels. Both works adopt reinforcement learning (RL) as the optimization framework to optimize placement. Although they work on different placement scale, both use GNN as feature extractors to learn information from their circuit graph. In [30], macro placement problem is cast as a Markov Decision Process (MDP) where macros are placed one at a time sequentially. It builds an Edge-GNN network to learn from the state of the existing macro placement and feed the output of this network to a CNN network to produce the RL policy for the next macro placement. The Edge-GNN is built on top of a clustered graph, where nodes represent either a cell cluster or a macro to be placed. The node features include macro size attributes width and height as well as location attributes x and y. The propagation module of the Edge-GNN is to combine the node features of connected nodes and edge features (net weights based on timing importance) to generate edge embeddings and update a node's embedding with the average of the edge embeddings on its connected edges.

GNN is also used in NVCell to place transistor within a standard cell. NVCell formulates the transistor placement problem as a MDP where each device (NMOS, PMOS or pin) is placed one at time from left to right. The state of the current placement step is represented in a graph, and a GNN is applied to compute its node embeddings. The GNN node embeddings are then used to compute the RL placement policy, which decides which PMOS, NMOS devices, pin, or dummies to be placed next. As shown in Figure 5, similar to the ParaGraph [15] conversion method, the nodes include device nodes and net nodes. Device nodes include NMOS transistors, PMOS transistors, IO pins (not shown in Figure 5 for illustration simplicity) and dummy transistors. The inclusion of dummy transistor nodes is to model the placement scenario where two adjacent transistors can not share diffusion together therefore a dummy transistor have to be placed. The circuit connectivity is presented as edges between nodes. Virtual edges are also created between dummy transistors and other transistors to make it possible for GNN to reason about when to insert dummy transistors. Edges between different types of nodes have different types. The features of each device nodes include the one-hot encoding of its node type, number of fins, its placement feature and whether the device is flipped or not. The placement feature is defined as following: $\{2\}$ if the device is placed in the immediate previous step, $\{1\}$ if it's placed previously, and $\{0\}$ if not yet placed. The inductive bias of the placement feature design is that devices placed subsequently have high correlation to each other. The propagation module of this GNN is based on the continuous kernel-based convolutional operator [32], which uses different edge weight metrics to combine node embeddings from different types of neighboring nodes. This propagation method can efficiently distinguish different subgraph patterns for the generated graph.

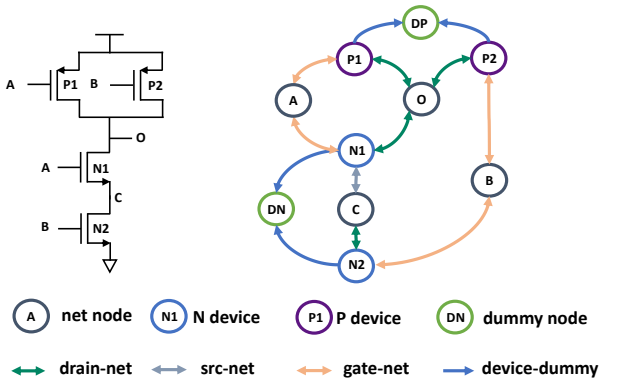

Fig. 5. NAND2 Cell Schematic and NVCell Graph in [31]. A, B are input nets, O is the output net, C is an internal net. For simplicity, IO pin nodes are not illustrated in this graph.

Both works [30], [31] demonstrated model generalization capability on unseen designs. This is mainly due to the fact that their GNN architectures applied the inductive bias of placement as their model constraints. For example, [30] demonstrated that Edge-GNN edge embeddings can be used to predict the total wirelength of the final placement, which can be used as the value function for the PPO [33] RL algorithm. This is not difficult to understand, since edge embedding can be treated as a wirelength predictor for each edge, and the total wirelength can be predicted by the mean prediction of all the edges.

B. Gate Sizing

Netlist optimization is an important and difficult predictive task that can leverage GNNs as feature extractors. The (local) neighborhood embedding or representation of instances learnt by

GNNs are the inductive biases for optimization problems. Here, we discuss two use-cases of GNNs applied for gate sizing optimizations applied to a netlist for timing closure and area/power recovery.

Reinforcement Learning (RL)-driven gate sizing. The problem of combinational gate sizing in a placed netlist is formulated as a MDP by Lu et al. [34], and RL is used to find the best gate sizes to minimize design total negative slack (TNS). The authors propose *RL-Sizer* that is used to perform gate sizing optimizations on industrial-quality designs. There are two key aspects in *RL-Sizer* that leverage GNNs: (i) RL state of an "optimizable" instance[1], and (ii) the instance selection algorithm.

From every critical timing path, optimizable instances are selected for sizing by *RL-Sizer*. Figure 6 illustrates the selection of instance *u8* as the optimizable instance and the *subgraph* around it. The subgraph of *u8* consists of *u7* as the parent, *u10* and *u11* are the sink nodes and *u9* is the side-fanout node. Each of these nodes have initial timing features such as slack, transitions, arc delays and parasitics; library features such as nominal buffer delay and pin capacitances; logical features such as the numbers of fanins, fanouts and side-fanouts; and layout features such as net lengths and pin locations. Each optimizable instance has a learnt representation using a GNN that includes its neighboring nodes that are the instance's parent(s), sink(s) and side-fanouts, as well as initial features of these nodes. GraphSage [8] is used to obtain the learnt representations.

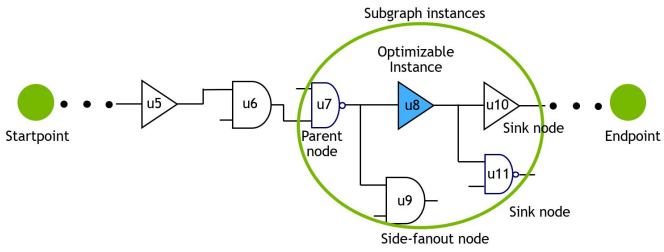

Fig. 6. Example of an instance subgraph in a timing path in [34].

The inductive bias in this sizing optimization problem are the learnt representation of each instance from its neighbors, and logical, physical and timing features. These representations are assumed to be invariant across instances across netlists. That is, another instance in the same or different netlist, that has similar representations would achieve a similar sizing solution by the optimizer. RL-Sizer leverages this bias for runtime speedup and as well as TNS convergence (and, overall convergence of the RL trajectories).

To achieve runtime speedup, *RL-Sizer* picks instances ordered by timing slack criticality and those that have non-overlapping subgraphs for simultaneous sizing. This ensures that transition changes induced by sizing of an instance does not affect its immediate neighbors (i.e., parents, fanouts and side-fanouts). Therefore, the sizer can predict sizes for a whole batch of non-overlapping instances during inference. To achieve TNS convergence, the authors use a proxy reward of TNS improvement in the subgraph instead of the design's TNS improvement, which would involve massive computation of timing arcs. These two reasons explain why it is useful to learn representations of an instance based on its subgraph (as shown in Figure 6). The authors use these subgraphs for training as opposed to the entire netlist graph, which enable large speedups in their GNN training. A subgraph has few tens of instances (or, nodes), whereas a full netlist graph contains greater than a million nodes.

GNN-derived instance representations also improve the prediction accuracy and design TNS in *RL-Sizer*. An experiment is conducted to compare efficacy of learnt instance representations based on local neighborhood information versus using the same set of features of an instance as RL state. As shown in Table IV of [34], which compares timing, power and runtime numbers across all blocks from $16nm$ to $5nm$, WNS can degrade by up to 12.5%, TNS can degrade by up to 24.7%, total power can degrade by up to 0.17%, and runtime can degrade by up to 20%. We, therefore, conclude that using GNN-learnt representations achieve better power, performance and runtime across all the designs used to validate *RL-Sizer*.

Supervised GNN gate sizing. In the above use-case, *RL-Sizer* uses GNN primarily as a feature extractor and derive learnt representation of optimizable instances. However, we can also use GNN as a supervised learning algorithm to predict electrical parameters of an instance. Another deep learning-based supervised model trained by using features from cell timing libraries then uses the predictions from the supervised GNN model and predict library-compatible sizes. We use such a modeling approach when we want to predict the postroute gate sizes in a pre-route stage.

Figure 7 shows the architecture of this use-case. The GNN model is trained on instance subgraphs with one-hop neighbors, and predicts input transition, load capacitance and worst cell arc delay at the postroute flow stage. The deep learning supervised model takes these predicted input transition and load capacitance along with other library features of this gate type (defined by its function ID), such as area, leakage, and predicts the worst cell arc delay for each library cell that are available for this gate type. Finally, we pick the library cell that has the closest match in the worst cell arc delay between the GNN-predicted model and the deep learning model. Such a use-case is a more conventional method to predict gate sizes by using a supervised model, and GNN model provides this supervision. We experimented with both GCN, GAT, GraphSage and GINConv models, and found that GraphSage to achieve the highest inference accuracy.

Similar to *RL-Sizer*, the inductive bias in this GNN-supervised gate sizing are the instance representations learnt by using instance-specific features, such as timing, physical and logical, and by transforming and aggregating representations from the instance's neighbors that consist of its parents, sinks and side-fanouts. The assumption here, again, is that similar representations would yield similar sizing solutions by the supervised model.

V. Challenges and Research Direction

GNN Challenges There are many challenges applying GNN to VLSI problems. Some challenges are common to any machine learning task. For example, features selections, dataset preparation, and model architecture tuning. There are also some specific challenges related to GNN.

Graph isomorphism [35] is one of the unique graph challenges. Two graphs are considered isomorphic if there is a mapping between the nodes of the graphs that preserves node adjacency. It is unclear whether graph isomorphic problem is NP-complete or not when the correspondence between two graphs are not provided. A necessary condition of graph isomorphism can be measured with the Weisfeiler-Lehman (WL) isomorphism test, which compares two graphs by converting both into their canonical forms. It has been shown [35]

[1]We refer to an instance as "optimizable" if it does not have user constraints such as "dont_touch" etc. applied on them, and can be optimized by an EDA tool.

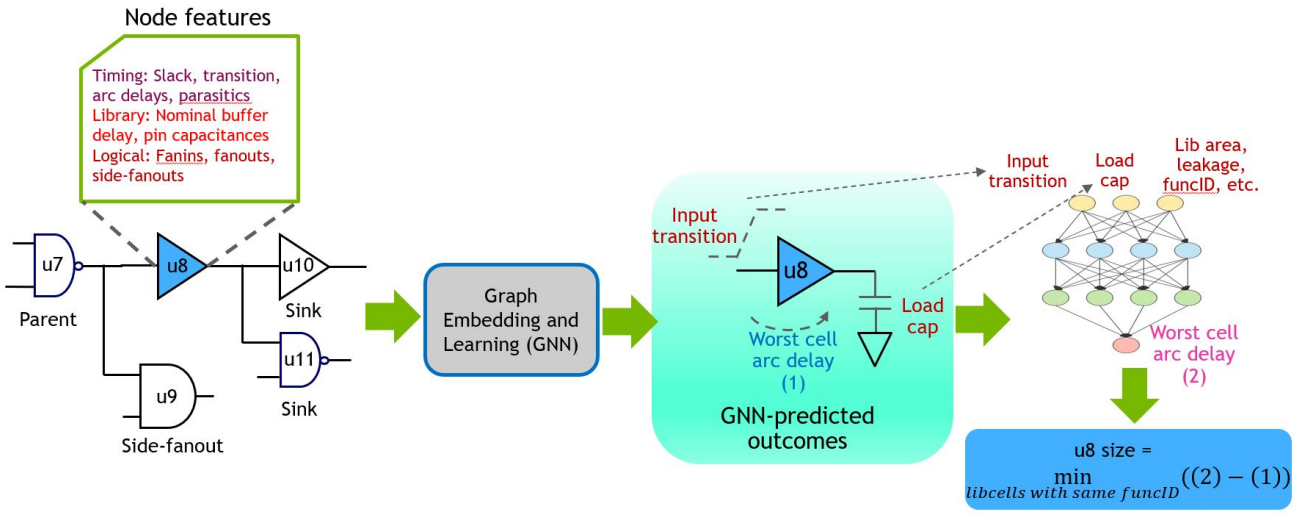

Fig. 7. GNN-supervised gate sizer architecture.

that common GNNs such as GCN [7] and GraphSage [8] can not distinguish different graph structures, while other carefully design architectures such as Graph Isomorphism Network (GIN) [35] can be as powerful as WL isomorphism test through injective aggregation scheme, e.g. adding noises to the scheme. It is still not clear, on whether WL isomorphism property alone is strong enough for distinguishing all isomophic graph structures, since it is only a necessary condition not a sufficient condition.

Unlike CNNs for computer vision tasks, which often have very deep frameworks with tens or hundreds layers, GNN often have shallow layers. This is due to message passing and then message reducing mechanism. The reducing process makes the neural network learn the average of several in-degree embeddings/representations, creating a 'smoothing' effect that contradicts the 'uniqueness' of each node/edge's true functionality in the graph. Because this limitation, the compositionality of GNN is limited.

The third challenge is GNN performance. Modern VLSI circuits can have billions of transistors, even on partition level, a design can have multiple million cells. This makes training and inference on GNN time consuming, especially when the message passing needs to be conducted in a sequential manner like the GRANNITE framework [3]. The GCN-based testability prediction work [2], an early applications of GNN in the VLSI design automation domain, implemented its own high performance GPU accelerated graph model to achieve efficient inference (1.5 sec) on designs with one million gates. Recently, open-source GPU accelerated GNN library such as DGL [36] and PyTorch Geometric [37] become available. These libraries include well-designed kernels and build-in common GNN architectures, which helps alleviate this challenge and improve the development productivity.

Graph Self-Supervised Learning Supervised learning method suffers from two major shortcomings: 1. there are not enough labeled data, 2. poor generalization due to the over-fitting problem. To overcome these shortcomings, self-supervised learning (SSL) [38] has been proposed to reduce the dependency on manual labels and provide better representation learning. SSL leverages supervision signals/loss functions from data itself without the need for manual label.

Recently there has been increasing interest in applying SSL to GNN. As illustrated in [39] , there are four types of Graph SSL: Generation-based methods form the pretext tasks as either feature or graph structure reconstruction; Auxiliary property-based methods added additional attributive and topological graph properties to the training loss; Contrast-based methods maximizes the similarity of augmented instances of same graph and minimizes that of different graphs; and Hybrid methods combines the previous three methods with more than one pretext tasks/training objectives.

Graph SSL can help improve the quality of VLSI predictive tasks. For example, [40] proposes to use the contrast-based method to learn a better functionality representation of digital circuits. Unlike commonly used graph augmentation methods such as node feature masking, node feature shuffling, edge modification, graph diffusion and subgraph sampling as described in [39], they understood the inductive bias of the functionality of the digital circuit is logic equivalence between different circuits implementing the same logic function, therefore designed a unique graph augmentation method based on equivalent logic circuit graphs.

VI. Conclusions

In this paper, we shown various applications of GNN in the VLSI design automation domain. We shown that GNN can not only perform VLSI predictive tasks, but also optimization tasks, which are the core of design automation. We argue that the key reason makes GNN successful for the VLSI design automation domain is that both the graph and the GNN architecture are carefully designed to incorporate inductive biases corresponding to the relevant tasks. We would encourage readers to use GNN in their VLSI design automation problems. Instead of naively converting a VLSI circuit into graph and blindly using GNN architectures designed for other domains, one should think deep about the inductive biases of the problem at hand to achieve better results.

References

[1] Z. Wu *et al.*, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019. [Online]. Available: http://arxiv.org/abs/1901.00596

[2] Y. Ma *et al.*, "High performance graph convolutional networks with applications in testability analysis," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317838

[3] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph Neural Network Inference for Transferable Power Estimation," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, 2020.

[4] H. Chen *et al.*, "Universal symmetry constraint extraction for analog and mixed-signal circuits with graph neural networks," in *ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1243–1248.

[5] Z. Guo *et al.*, "A timing engine inspired graph neural network model for pre-routing slack prediction," in *Proceedings of the 59th Annual Design Automation Conference 2022*. ACM, 2022.

[6] G. Karniadakis *et al.*, "Physics-informed machine learning," in *Nature Review Physics 3*, 2021, pp. 422–440.

[7] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016.

[8] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," 2017. [Online]. Available: https://arxiv.org/abs/1706.02216

[9] P. Velickovic *et al.*, "Graph attention networks," *ArXiv*, vol. abs/1710.10903, 2017.

[10] A. Goyal and Y. Bengio, "Inductive biases for deep learning of higher-level cognition," *CoRR*, vol. abs/2011.15091, 2020. [Online]. Available: https://arxiv.org/abs/2011.15091

[11] Y. Zhang *et al.*, "GATSPI: GPU Accelerated Gate-Level Simulation for Power Improvement," in *Proceedings of the 59th ACM/EDAC/IEEE Design Automation Conference*, 2022.

[12] M. Nourani, S. Nazarian, and A. Afzali-Kusha, "A Parallel Algorithm for Power Estimation at Gate Level," in *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002.*, vol. 1, 2002, pp. I–511.

[13] H. Mehta, "Accurate Estimation of Combinational Circuit Activity," in *32nd Design Automation Conference*, 1995, pp. 618–622.

[14] H. Yoshida, K. De, and V. Boppana, "Accurate pre-layout estimation of standard cell characteristics," in *DAC*, 2004, pp. 208–211.

[15] H. Ren *et al.*, "Paragraph: Layout parasitics and device parameter prediction using graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[16] M. Schlichtkrull *et al.*, "Modeling relational data with graph convolutional networks," *Lecture Notes in Computer Science*, p. 593–607, 2018.

[17] A. Hastings and R. A. Hastings, *The Art of Analog Layout*, 1st ed. USA: Prentice Hall, 2001.

[18] U. Choudhury and A. Sangiovanni-Vincentelli, "Constraint generation for routing analog circuits," in *ACM/IEEE Design Automation Conference (DAC)*, 1990, pp. 561–566.

[19] E. Charbon, E. Malavasi, and A. Sangiovanni-Vincentelli, "Generalized constraint generation for analog circuit design," in *IEEE/ACM International Conference on Computer-Aided Design*, 1993, pp. 408–414.

[20] H. Chen *et al.*, "MAGICAL 1.0: An open-source fully-automated ams layout synthesis framework verified with a 40-nm 1gs/s $\delta\sigma$ adc," in *IEEE Custom Integrated Circuits Conference (CICC)*, 2021, pp. 1–2.

[21] M. Liu *et al.*, "S^3DET: Detecting system symmetry constraints for analog circuits with graph similarity," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2020, pp. 193–198.

[22] X. Gao *et al.*, "Layout symmetry annotation for analog circuits with graph neural networks," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2021, pp. 152–157.

[23] K. Kunal *et al.*, "A general approach for identifying hierarchical symmetry constraints for analog circuit layout," in *IEEE/ACM International Conference on Computer-Aided Design*, 2020, pp. 1–8.

[24] Y. Li *et al.*, "Gated graph sequence neural networks," in *International Conference on Learning Representations (ICLR)*, 2016, pp. 1–20.

[25] C. Alpert *et al.*, "Accurate estimation of global buffer delay within a floorplan," in *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, 2004, pp. 706–711.

[26] M. Vujkovic *et al.*, "Efficient timing closure without timing driven placement and routing," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 268–273. [Online]. Available: https://doi.org/10.1145/996566.996646

[27] Z. Guo and Y. Lin, "Differentiable-timing-driven global placement," in *Proceedings of the 59th Annual Design Automation Conference 2022*. ACM, 2022.

[28] E. C. Barboza *et al.*, "Machine learning-based pre-routing timing prediction with reduced pessimism," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[29] Z. Xie *et al.*, "Net2: A graph attention network method customized for pre-placement net length estimation," in *ASPDAC '21: 26th Asia and South Pacific Design Automation Conference, Tokyo, Japan, January 18-21, 2021*. ACM, 2021, pp. 671–677. [Online]. Available: https://doi.org/10.1145/3394885.3431562

[30] A. Mirhoseini *et al.*, "Chip placement with deep reinforcement learning," *CoRR*, vol. abs/2004.10746, 2020. [Online]. Available: https://arxiv.org/abs/2004.10746

[31] H. Ren and M. Fojtik, "Invited- nvcell: Standard cell layout in advanced technology nodes with reinforcement learning," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1291–1294.

[32] J. Gilmer *et al.*, "Neural message passing for quantum chemistry," *CoRR*, vol. abs/1704.01212, 2017. [Online]. Available: http://arxiv.org/abs/1704.01212

[33] J. Schulman *et al.*, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[34] Y.-C. Lu *et al.*, "RL-sizer: VLSI gate sizing for timing optimization using deep reinforcement learning," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 733–738.

[35] K. Xu *et al.*, "How powerful are graph neural networks?" *CoRR*, vol. abs/1810.00826, 2018. [Online]. Available: http://arxiv.org/abs/1810.00826

[36] M. Wang *et al.*, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[37] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[38] X. Liu *et al.*, "Self-supervised learning: Generative or contrastive," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2021.

[39] Y. Liu *et al.*, "Graph self-supervised learning: A survey," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2022.

[40] Z. Wang *et al.*, "Functionality matters in netlist representation learning," in *2022 58th ACM/IEEE Design Automation Conference (DAC)*, 2022.