A Survey of Graph Neural Networks for Electronic Design Automation

Daniela Sánchez Lopera *‡, Lorenzo Servadei *, Gamze Naz Kiprit *‡,
Souvik Hazra*, Robert Wille†, Wolfgang Ecker*‡
*Infineon Technologies AG, Germany, †Johannes Kepler University Linz, Austria,
‡Technical University of Munich, Germany

***Abstract*—Driven by Moore's law, the chip design complexity is steadily increasing. Electronic Design Automation (EDA) has been able to cope with the challenging very large-scale integration process, assuring scalability, reliability, and proper time-to-market. However, EDA approaches are time and resource-demanding, and they often do not guarantee optimal solutions. To alleviate these, Machine Learning (ML) has been incorporated into many stages of the design flow, such as in placement and routing. Many solutions employ Euclidean data and ML techniques without considering that many EDA objects are represented naturally as graphs. The trending Graph Neural Networks are an opportunity to solve EDA problems directly using graph structures for circuits, intermediate RTLs, and netlists. In this paper, we present a comprehensive review of the existing works linking the EDA flow for chip design and Graph Neural Networks.**

***Index Terms*—Electronic Design Automation, Very Large-scale Integration, Machine Learning, Register-Transfer Level, Graph Neural Networks**

I. Introduction

Over time, the chip design flow has incorporated multiple software tools to synthesize, simulate, test, and verify different electronic designs efficiently and reliably. The compendium of those tools is called Electronic Design Automation (EDA). Those tools automatize the chip design flow sketched in Figure 1. Nevertheless, the flow is sequential and time-demanding. Often, the design has to be verified and tested to ensure correctness, reliability, and objective closure. But only during physical verification and signoff, and testing, the quality of the design in terms of Power, Performance, and Area (PPA) can be measured. Corrective modifications in intermediate steps are often needed, and they result in multiple iterations of the design flow. Thus, estimations of PPA in earlier stages of the design would reduce the required iterations, increase the reliability of the design while going deeper on the flow, and finally, improve the Quality of Results (QoR).

In the last years, design complexity driven by Moore's law has increased. Chip capacity has been doubling around every two years, which translates into increasing efforts for the design and verification of even more diversified chips. EDA tools have aimed at coping with the new challenges and provided automated solutions for Very Large-Scale Integration (VLSI). EDA tools commonly face NP-complete problems, which Machine Learning (ML) methods could solve better and faster. Thus, ML has been integrated into EDA, specially to logic synthesis, placement, routing, testing and verification [1]. In [1], four main areas of action were recognized. First, ML is used to predict optimal configurations for traditional methods. Second, ML learns features of models and their performances to predict the behavior of unseen designs without running the costly step of synthesis. Moreover, design space exploration can be conducted by ML while optimizing PPA. Finally, Reinforcement Learning (RL) explores the design space, learns policies, and executes transformations to get optimal designs envisioning the future with an *"AI-assisted Design Flow"*.

978-1-6654-3166-8/21/$31.00 ©2021 IEEE

One enabling factor for using ML in EDA is the huge amount of data that is generated by the EDA tools along the design process. To apply ML over such data, these have to be pre-processed and labeled. Existing solutions use such data as Euclidean data, i.e. representing them in a 2-D Euclidean space, allowing the use of ML methods such as Convolutional Neural Networks (CNNs). However, the trending neural network framework for graphs called Graph Neural Networks (GNNs) has shown a significant improvement in dealing with data whose structure is intuitively a graph. Even though GNNs appeared already in 2005, their recent combination with Deep Learning (DL) operations like convolution and pooling has drawn significant attention in fields such as molecular graphs [2], recommendation systems [3], and traffic prediction [4].

In EDA, the most natural representation of circuits, intermediate RTL, netlists, and layouts are graphs. Thus, in the last two years, few studies have recognized this opportunity and have incorporated the usage of GNNs to solve EDA problems.

This survey gives a comprehensive review of some recent studies using GNNs in different stages of the EDA flow. It first provides background on both fields and, successively, a list of the seminal related works. The rest of this survey is organized as follows: In Section II, we briefly review the EDA flow and background concepts. In Section III, we provide a more detailed explanation of the different types of GNNs. In Section IV, we discuss different studies applied to EDA. Finally, Section V concludes by briefly mentioning some open challenges and future directions.

II. Background and Definition

In this section, we briefly review background concepts related to EDA flow, graphs and GNNs.

A. Electronic Design Automation

The progress in EDA tools and design methods, and the use of different levels of abstractions on the design flow have improved the hardware design productivity. Figure 1 sketches the stages of a modern chip design process. The flow starts with the chip specification modeling the desired application. The architecture analysis and prototype of the design represent the design as a collection of interacting modules such as processors, memories, and buses. In the functional and logical design, the behavioral description of those modules is mapped to Register Transfer Level (RTL) blocks using Hardware Description Languagess (HDLs) such

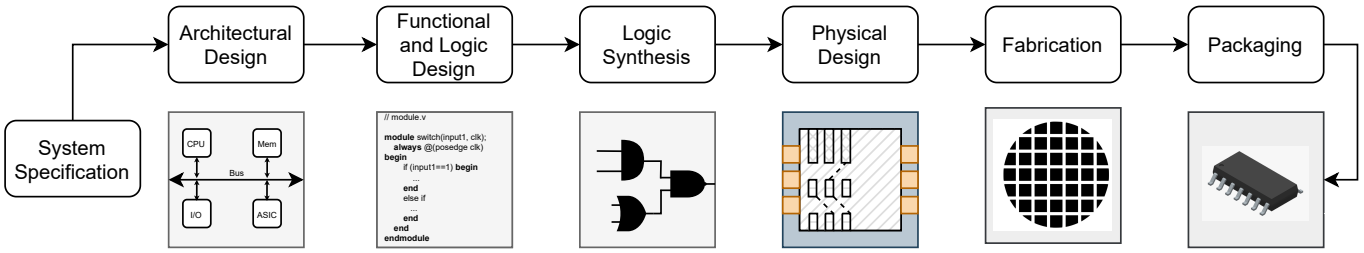

Fig. 1. Chip Design Flow

as Verilog. Nowadays, the transition from system specification to RTL can be done in different ways. For instance, using High-level Synthesis (HLS), which provides an automatic conversion from C/C++/System-C specifications to HDL, or using hardware design frameworks such as MetaRTL [5].

Logic synthesis maps the RTL blocks in HDL to a combination of gates selected from a given technology library while optimizing the design for different objectives. Normally, this optimization involves a trade-off between timing closure, and area and power consumption.

In physical synthesis, four main steps are executed: floorplanning, placement, clock insertion, and routing. First, main RTL blocks of the chip and ports are assigned to regions of the layout. Second, the gates of the resulting logic netlists are placed to specific locations of the chip. Finally, the wires for clock signals and for connecting the gates are added. These steps are executed targeting better area utilization, timing performance, and routability while considering design rules.

Since errors in the design cost time and resources, verification is a fundamental step of the flow and is executed after functional and physical design. After verification and signoff, the design goes through the manufacturing flow: fabrication, packaging, and final testing.

Even though the flow is highly automated, it encounters some drawbacks: (1) It relies on the hardware designer's expertise to select proper configurations, (2) Design space exploration is limited and time-demanding, (3) Corrections in the design would reinitialize the flow, (4) There is no early analysis or predictability of the results.

B. Graphs

a) Definition I (Graphs): A graph is a data structure for representing interactions between related objects. Mathematically, it is as a tuple $G = (V, E)$, where V is the set of nodes and E, the set of edges. The edges are defined as the connection between nodes, e.g. for the nodes $u, v \in V$, the edge is represented as $e_{u,v} \in E$. The neighborhood of a node $N(v)$ is defined as $N(v) = u \in V | (u, v) \in E$. A graph can be represented then as a list of nodes and edges. But a more convenient representation is through an *adjacency matrix* $\mathbf{A}$ defined as $\mathbf{A}_{u,v} = 1$ if $e_{u,v} \in E$, otherwise $\mathbf{A}_{u,v} = 0$. The degree of a node is the number of nodes D incident to a node u, mathematically it can be defined as $D_u = \sum_{v \in V} \mathbf{A}_{u,v}$ [6].

A graph with n nodes and m edges may have node and edge features of dimension d and c respectively, i.e. the node features are $\mathbf{X}$, where $\mathbf{X} \in \mathbb{R}^{n \times d}$ and the edge features $\mathbf{X}^e$, where $\mathbf{X}^e \in \mathbb{R}^{m \times c}$ [6].

b) Definition II (Types of Graphs): Graphs are classified into different classes. If the direction of the edges matter, the graph is *directed* and the adjacency matrix $\mathbf{A}$ will be symmetric. If the edges do not have a direction, the graph is *undirected*. In case the edges are represented with a cost or real-value, the graph is called *weighted*, and the matrix $\mathbf{A}$ will have real-values as entries. *Multiplex* graphs can be decomposed into layers, and the relation between each layer and the belonging nodes are additional *intra-layer* edges. Graphs can also be *homogeneous* or *heterogeneous*. In the former, all nodes and edges have the same type. In the latter, nodes have different types and their attributes may be of distinct types too (e.g. text or images) [6].

C. Shallow Embeddings Methods

The traditional approach for processing graph-structured data is to use shallow embedding methods. These aim to decompose the node information into low-dimensional embedding vectors that consider the position of the nodes in the graph and the structure of the neighborhood [6]. One of the best-known graph embedding techniques is Random Walk [7]. In this technique, given a starting point within a graph, a random neighbor point is selected. As a second step, a neighbor of the randomly selected point is chosen again. This, in a recursive fashion. This generates a random sequence of points, namely the random walk. DeepWalk [8] and Node2vec [9] are well-known graph embedding methods that are based on random walks. Although these methods have achieved groundbreaking success, they are *transductive*, i.e. they learn a unique embedding vector per node. Thus, they have two main limitations: They are computationally expensive in large graphs, and they cannot deal with unseen nodes. Moreover, they do not consider node features that could provide information during the encoding [6].

D. Graph Neural Networks

To overcome the limitations of the shallow methods, a novel Neural Network (NN) called GNN was introduced in [10]. GNNs are a framework for NNs, that operate directly on data structured as graphs, without losing structural and feature information [10]. Originally, GNNs were formulated as a type of Recurrent Neural Networks (RNNs) trained by a version of backpropagation through time [11].

Having an input graph, a GNN aims to learn the embedding vectors per node, defined as $\mathbf{h}_u \ \forall u \in V$, which encodes the neighborhood information of each node [12]. The message passing between nodes is assumed as the most generic GNN layer [11]. Through the message passing updates determined by the graph structure, the edge embeddings $\mathbf{h}^e_{(u,v)}$ are obtained using Equation 1.

$$\mathbf{h}^e_{(u,v)} = \phi(\mathbf{h}_u, \mathbf{h}_v, \mathbf{x}^e_{(u,v)}), \tag{1}$$

where $\phi(\cdot)$ is an arbitrary, non-linear, differentiable function that aggregates its inputs, and $\mathbf{x}^e_{(u,v)}$ the initial edge feature vector. After the edge representation is obtained, and defining $\mathbf{x}_{(u)}$ as the feature vector for the starting node, the node representation is updated as in Equation 2.

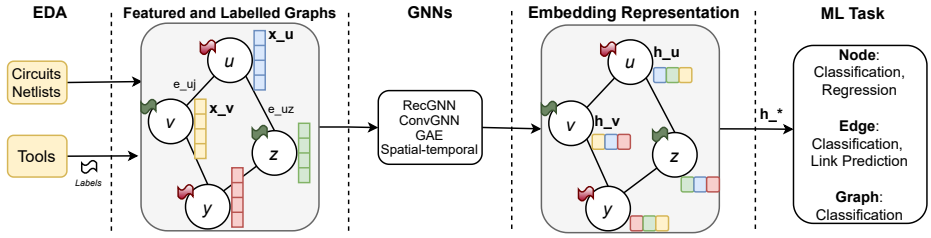

Fig. 2. End-to-End flow using EDA objects as graphs. Feature nodes x and ground truth labels (e.g. red and green flags) are collected using EDA tools. One of the GNNs flavors extracts node embeddings h, which are the inputs to other ML methods for classification or regression at node, graph, or edge level.

$$\mathbf{h}_u^{'} = \phi(\mathbf{h}_u, \sum_{v \in N(u)} \mathbf{h}^e_{(v,u)}, \mathbf{x}_u) \tag{2}$$

The graph embeddings learned by GNNs can be used as inputs to other ML models building an end-to-end framework depicted in Figure 2. There are three levels of tasks for such a framework: Node, edge, and graph [12]. In the node-level tasks, all nodes are labeled so that a regression or classification of the nodes is possible. In edge-level tasks, the goal is to classify edges or predict the link between two nodes. Finally, in graph-level tasks, the entire graph is labeled and a NN, combined with pooling and readout operations, can classify new unseen graphs. Moreover, the learning tasks of GNNs can be *transductive* or *inductive*. In the former, the GNN learns the embedding vectors per each node in the training graphs. Thus, during inference, it cannot generalize to new nodes. On the contrary, an inductive GNN learns the aggregation function that combines the node's neighborhood features to get the embedding vectors [12].

III. Classification of GNNs

GNNs are divided into four types [12]. In this section, we review them and describe their core ideas.

A. Recurrent Graph Neural Networks

Recurrent Graph Neural Networks (RecGNNs) [13] process the node information recurrently by assuming that the nodes exchange information with their neighbors until a stable point is reached. RecGNNs define the node aggregation function as in Equation 3.

$$\mathbf{h}_u^t = \sum_{v \in N(u)} \phi(\mathbf{x}_u, \mathbf{x}^e_{(u,v)}, \mathbf{x}_v, \mathbf{h}_v^{(t-1)}), \tag{3}$$

where $\phi(\cdot)$ is a non-linear differentiable recurrent function. In [13], the proposed architecture is a RNN where the connections between the neurons are classified into internal and external connections. While the first refer to the internal connections within units of the network, the external ones refer to the edges of the processed graph.

B. Convolutional Graph Neural Networks

Convolutional Graph Neural Networks (ConvGNNs) are a generalization of CNNs to graph data and are divided into spectral, and spatial approaches. The spectral approaches are based on the Laplacian's eigenbasis, which differs depending on the graph structure. Spatial approaches, however, are based on the spatial structure of the graph, i.e. they work on spatially close neighbors [12]. In [14], a spectral approach for performing the convolution operations on graph structures has been presented. This is named Graph Convolutional Networks (GCNs). The main principle of this approach is to extract the high-level embeddings of the nodes by aggregating the features of the central node and the neighbor nodes. Mathematically, this is expressed as in Equation 4.

$$\mathbf{h}_u^{(l+1)} = \phi\Big(\sum_{v \in N(u)} \frac{1}{c_{(u,v)}} \mathbf{h}_v^{(l)} \mathbf{W}^{(l)} \Big), \tag{4}$$

where $c_{(u,v)} = \sqrt{|N(u)|}\sqrt{|N(v)|}$ is the normalization constant of the edge $e_{u,v}$, $\mathbf{W}$ is a learnable weight matrix, $l \in \{1, ..., L\}$ with L being the number of layers, and ϕ is the non-linear activation function [12]. Even though GCN are very powerful in generating low-dimensional embeddings of large graphs, they are *transductive*, i.e. all possible nodes have to be present during training. To alleviate this, GraphSAGE [15] proposes an inductive framework that generalizes to unseen nodes. It utilizes an update rule which is similar to Equation 4, by defining $c_{(u,v)} = |N(u)|$.

GraphSAGE uses a fixed-size neighborhood of the nodes, which can limit the network performance during inference. To solve this, Graph Attention Networks (GATs) [16] were introduced. A GAT computes each node embeddings by going through all the neighbors using the self-attention mechanism presented in [17]. Mathematically, this node embedding process can be expressed as in Equation 5.

$$\mathbf{h}_u^{(l+1)} = \phi\Big(\sum_{v \in N(u)} \alpha^{(l)}_{(u,v)} \mathbf{z}_v^{(l)} \Big), \tag{5}$$

where $\mathbf{z}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u$, and $\alpha^{(l)}_{(u,v)}$ is the normalized attention score of the node v to u calculated by the l-th attention mechanism. The main benefit of GAT is not only the consideration of the entire neighborhood of each node but also the increasing model's *expressiveness* that comes with the specification of different relevance scores of each edge for a given node.

C. Graph Autoenconders

Graph Autoencoders (GAEs) belong to the family of unsupervised frameworks and are used for graph-based representation learning and graph generation [12]. In both tasks, an encoder is employed to extract node embeddings of a graph, followed by the reconstruction of new graphs from corresponding latent or embedding vectors. For representation learning, graph structural information is reconstructed as an adjacency matrix. In the case of graph generation, the process might involve a step-wise generation of the nodes and edges, or output the entire graph at once.

D. Spatial–Temporal Graph Neural Networks

Spatial-Temporal Graph Neural Networks (STGNNs) aim at capturing underlying spatial and temporal relation simultaneously [12]. The spatial relation is captured by using graph convolutions, and the temporal relation is modeled by employing RNN blocks.

IV. Application to EDA

These two seminal papers [18], [19] highlight the important link between EDA tasks and GNNs.

The first study to recognize the high potential of GNNs in EDA is [18]. They stated that graph structures are the most intuitive way to represent Boolean functions, netlists, and layouts, which are the main focus of the EDA flow. They see GNNs as an opportunity for EDA to improve the QoRs and to replace the used traditional shallow methods or mathematical optimization techniques. The paper lists related studies that have been applying analytical and heuristic approaches and shallow methods to EDA. Finally, they introduced *spectral-based* and *spatial-based* GNNs and presented only two cases of study: Test point insertion and timing model selection.

In [19], a review of CNNs as well as GNNs used in the EDA flow was presented. They stated that ML could improve the QoRs during the chip design flow by predicting important metrics in different phases such as design space exploration, power analysis, physical design, and analog design. Similar to [18], they envisioned the use of Deep Reinforcement Learning (DRL) to solve combinatorial optimization problems in EDA, similarly to what is done in [20].

In [18] and [19], the motivation of incorporating the trending GNNs into the EDA flow is clear. However, they did not exclusively center on both areas. [18] focuses also on existing applications using traditional shallow methods. On the other hand, [19] reviews the use of CNNs and GNNs. Moreover, [19] compares the applications from an EDA perspective, without revealing the details of the GNN concepts behind.

Considering the drawbacks of the above-mentioned work, this survey gives a background and a review of recent important studies applying GNNs to the EDA field. To clarify the link between both areas, the review of these studies is organized according to their corresponding stages in the design flow. Table I lists the studies considered in this review.

A. Logic Synthesis

During logic synthesis, the RTL blocks describing the hardware design are mapped to logic cells from a technology library. This mapping must meet the timing constraints to operate at the desired clock rate while considering area and power. Therefore, synthesis is a complex optimization problem where ML can be applied. For instance, providing earlier QoR predictions to avoid multiple runs of the time-demanding synthesis step.

To predict a more accurate delay for Field Programmable Gate Array (FPGA) blocks, [21] proposes to learn the mapping and clustering patterns of arithmetic operations in FPGAs, specially Digital Signal Processor (DSP) and carry blocks. They recognized that current HLS solutions only sum up the individual delays of each component along the paths. This does not consider the underlying optimizations done during the synthesis. As solution, they proposed a novel architecture D-SAGE [21], a GNN to predict the complex technology mapping done by logic synthesis. In [21], designs are mapped to Data Flow Graphs (DFGs). The nodes are the set of operations (i.e. additions or multiplications), and the edges, the data dependencies between the nodes. Node types and bit widths of the data are considered as node attributes. Nodes and edges are labeled according to the end-to-end task. For instance, if the nodes are mapped to DSP blocks or Lookup Tables (LUTs), they are labeled to one or zero, respectively. Similarly, edges are marked with one if their connected nodes are mapped to the same device. D-SAGE leverages GraphSAGE to support directed graphs and distinguish between successors $SU(u)$ and predecessors $PR(u)$ of a node u. To that end, Equation 4 is split over $SU(u)$ and $PR(u)$, and finally, the successors and predecessors embeddings are concatenated as in Equation 6.

$$\mathbf{h}_u^{(l+1)} = \phi\left(\mathbf{h}_{u,PR}^{(l+1)},\ \mathbf{h}_{u,SU}^{(l+1)}\right) \tag{6}$$

Using the graph embeddings, D-SAGE solves two end-to-end tasks: Binary node classification, to predict which nodes are mapped to which device, and binary edge classification, to cluster nodes mapped into the same device. D-SAGE outperforms HLS tools in node classification, edge prediction, and also in operation delay estimation across all data paths.

B. Verification and Signoff

Verification is done to check the functionality of the design after functional, logic, and physical design. Especially before fabrication, the correctness of the design has to be assured. In the signoff step, a set of verification steps is executed to formally verify the functionality but also design closure, signal integrity, lifetime checks, etc. The design closure is determined in terms of PPA. Negative results in this step translate into going backward on the flow and increasing the time-to-market of the chip. Therefore, early, accurate and fast estimations of those constraints could accelerate the design process.

For instance, the power integrity signoff requires power analysis of the design. For this, vector-based methods are preferred because of their accuracy, but they require gate-level simulations, that are time-demanding and replaced in practice by Switching Activity Estimators (SAEs). SAEs are fast but not accurate. In [22], an alternative method based on GNN is proposed using toggle rates as inputs and improving the prediction accuracy. To that end, they built a graph based on the netlist, where single-output components are the nodes. The edges are defined as the connection between gates. From the RTL simulations, input and register toggle rates are taken as feature nodes, which are encoded in a 4-D vector. Finally, the predicted toggle rates per gate are evaluated against the ground truth labels obtained by the gate-level simulation. Intuitively, the toggle rates are expected to propagate from one level to the next one. Therefore, they proposed GRANNITE [22], a sequential and inductive version of a GCN, in which the node embeddings are not calculated in parallel but sequentially. Using the node embeddings, GRANNITE predicts average toggle rates from RTL simulations in few seconds with more accuracy than classical SAEs.

C. Floorplanning

In chip floorplanning, the main and large blocks of a netlist are placed onto 2-D grids aiming for optimal PPA, while obeying design rules. This can be represented as a Markov process, which can be solved using RL.

The most significant work in this area is presented in [23]. Google demonstrates the success of chip macro placement using a DRL framework for the floorplanning of Tensor Processing Unit (TPU) accelerators. In [23], a GNN is incorporated to the RL framework to encode the different states of the process, predict the reward labels for congestion, density, and wirelength, and generalize to unseen netlist. The proposed architecture is called Edge-Based Graph Neural Network (Edge-GNN) [23], which calculates the node and edge embeddings for the whole netlist. This RL agent gives

TABLE I
SUMMARY OF GNNS FOR THE EDA FLOW

Section	GNN Algorithm	Type of Task	End-to-End Task	Reference
Logic Synthesis	GraphSAGE	Node classification	Learning mapping patterns from HLS to FPGA blocks	D-SAGE [21]
Verification and Signoff	GCN	Node regression	Vector-based average power estimation	GRANNITE [22]
Floorplanning	GCN	GNN-RL	Floorplanning optimization as RL task	Edge-GNN [23]
Placement	GAT	Node regression	Net length estimation	Net^2 [24]
	GraphSAGE	Node clustering	Optimization of placement groups as guidance for tool	PL-GNN [25]
	GraphSAGE	GNN-RL	Generalization PPA optimization as RL task	- [26]
	GraphSAGE	GNN-RL	Optimization of placement parameters using RL	- [27]
Routing	GAT	Node regression	Estimation routing congestion	CongestionNet [28]
Testing	GCN	Node classification	Prediction of observation point candidates	- [29]
Analog Design	GraphSAGE, GAT	Node regression	Prediction of net parasitic capacitances	ParaGraph [30]
	GNN	Graph regression	Simulation electromagnetic properties of distributed circuits	CircuitGNN [31]
	GCN	GCN-RL	Transferring knowledge of transistor sizing	Circuit Designer [32]
	GAT	Node regression	Prediction analog circuit performance due to placement	PEA [33]

comparable or better results than a human designer but takes hours instead of months.

D. Placement

During placement, the design gates are mapped to the exact locations of the chip layout. The larger the design, the more complex this process is. A poor decision during placement can increase the chip area but also worsen the chip performance and, even, make it unsuitable for manufacturing if the wirelenght is higher than the available routing resources. Therefore, placement is seen as a constrained optimization problem. ML and specially, GNNs are being explored to ease this steps [24], [25], [26], [27].

A GAT called Net^2 is used to provide pre-placement net and path length estimations in [24]. To that end, they converted the netlists to directed graphs, where nets represent nodes, and edges connect nets in both directions. The number of cells, fan-in, fan-out sizes, and areas are used as feature nodes. The edge features are defined using clustering and partitioning results. The ground truth label for the nodes is the net length obtained as the half-perimeter wirelength of the bounding box after placement. During inference, Net^2 predicts the net length per node, outperforming existing solutions. For instance, Net^{2a}, a version targeting accuracy, is 15% more accurate in identifying long nets and paths, and Net^{2f} targeting runtime, is 1000 $\times$ faster.

In [25], GraphSAGE is leveraged to build PL-GNN, a framework helping placer tools to make beneficial decisions to accelerate and optimize the placement. PL-GNN converts netlists to hypergraphs, where nodes and edges features are based on the hierarchy and the affinity of the net with memory blocks, as this provides information about critical paths. A GNN is used to learn the node embeddings, which are clustered by the K-means algorithm to determine optimal placement groups based on the cell area. This guidance leads the placer tool to a placement that reduces wirelength, power, and worst slack.

A proof of concept framework mapping the PPA optimization task in EDA to a RL problem is presented in [26]. This RL framework uses GraphSAGE with unsupervised training to learn node and edges embeddings that can generalize to unseen netlist. The GCN is a key component because it extracts local and global information, which is needed by the RL agent. Wirelength optimization during 2-D placement is analyzed as a case of study.

In [27], an autonomous RL agent finds optimal placement parameters in an inductive manner. Netlists are mapped as directed graphs, and nodes and edges features are handcrafted placement-related attributes. GraphSAGE learns the netlist embeddings and helps to generalize to new designs.

E. Routing

In this step, the placed components, gates, and clock signals are wired while following design rules (e.g. type of permitted angles). These rules determine the complexity of the routing, which is mostly an NP-hard or NP-complete problem. Thus, routing tools are mostly based on heuristics, and they do not aim to find an optimal solution. ML methods could enhance the routing process by providing earlier estimations, which can be used to adjust the placement accordingly, and avoid high area and long wires.

In [28], a GAT is used to predict routing congestion values using only a technology-specific gate-level netlist obtained after physical design. To that end, the netlists are built as undirected graphs, where each gate is a node, and the edges are defined as the connections between gates that are connected by a net. The feature nodes are 50-D vectors containing information about cell types, sizes, pin counts, and logic descriptions. To get the ground truth labels for the nodes, congestion maps are split into grids, and the congestion value of each grid is taken as a label for the cells that were placed into that grid. The architecture presented in [28] called CongestionNet is not more than an eight-layer GAT following the Equation 5. The node embeddings are used to predict local congestion values. Using GATs improves the quality of the predictions, and the inference time is around 19 seconds for circuits with more than one million cells.

F. Testing

Testing takes place only after the packaging of the design. The larger the design, the higher the complexity and the execution time of the testing tools. Moreover, testing should

guarantee a high coverage, avoiding redundant test cases. Testing is not scalable, and it strongly relies on human expertise. To overcome those challenges, ML is being incorporated into the testing phase. For instance, to reduce test complexity by providing optimal test points in a design. [29] proposes a GCN to insert fewer optimal test points on the design while maximizing the fault coverage. To that end, a directed graph is built using the components of the netlist and primary ports as the nodes, and the wires between them as edges. The node features are 4-D vectors containing information about the logic level of each gate, controllability, and observability attributes. Using Design-for-Test tools, the ground truth labels are collected, and the nodes are labeled as "easy-to-observe" or "difficult-to-observe". The proposed GCN model generates the node embeddings using Equation 4, replacing the aggregation function with a weighted sum to distinguish between predecessor and successor nodes.

Having the node embeddings, a binary node classification is performed. During inference, the nodes of new netlists are classified into "difficult" or "easy-to-observe". This information is then used by testing tools to reduce the test complexity. Compared with commercial test tools, [29] reduces the observation points by around 11% while keeping the same coverage.

G. Analog Design

The high complexity of the analog design flow is due to the large design space and the signal susceptibility w.r.t. noise. Thus, the analog flow could strongly benefit from modern approaches like ML to modernize the methods and improve the QoRs. Specially, GNNs are being applied to this field. In [30], a GNN is used to predict net parasitic capacitance and device parameters. The design schematics are converted to hypergraphs and are the inputs to ParaGraph [30], a version of GNN that combines ideas from GraphSAGE and GAT. CircuitGNN [31] uses a GCN to predict magnetic properties per node using components as nodes and edges as magnetic and electrical relationships. Circuit Designer [32] uses a GCN to extract the node embeddings of a circuit, which are later used as inputs to an RL agent targeting technology-independent transistor sizing. In [33], a new architecture called Pooling with Edge Attention (PEA) is introduced to evaluate how different placement solutions would affect the analog circuit performance.

V. Conclusion

To the best of our knowledge, our work is the first that collects seminal papers on the crossing between EDA and modern GNNs. In the presented works, GNNs outperformed the baseline methods. However, as the complexity of circuits in EDA continues growing, scalability and heterogeneity are still open challenges. We expect that the usage of GPUs helps to alleviate this bottleneck. We believe that the high potential of ML combined with GNNs will open the door to many more solutions targeting the EDA flow. We hope that using netlist, layouts, and intermediate RTL directly as graph structures can accelerate the earlier prediction of hardware metrics, and the usage of RL to solve combinatorial tasks in the EDA flow. Finally, we also expect that future work on GNNs targets some open-graph-related challenges such as heterogeneity, scalability, and diversity of graphs.

References

[1] G. Huang *et al.*, "Machine Learning for Electronic Design Automation: A Survey," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2021.

[2] J. You *et al.*, "Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation," *ArXiv*, 2018.

[3] R. Ying *et al.*, "Graph Convolutional Neural Networks for Web-scale Recommender Systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[4] B. Yu *et al.*, "Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting," *ArXiv*, 2017.

[5] J. Schreiner *et al.*, "Design Centric Modeling of Digital Hardware," in *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2016.

[6] W. L. Hamilton, *Graph Representation Learning*.

[7] L. Lovász *et al.*, "Random Walks on Graphs: A Survey," *Combinatorics, Paul erdos is eighty*, 1993.

[8] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online Learning of Social Representations," ser. KDD '14, Aug. 2014.

[9] A. Grover and J. Leskovec, "Node2vec: Scalable Feature Learning for Networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, Aug. 2016.

[10] M. Gori *et al.*, "A New Model for Learning in Graph Domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks*, 2005.

[11] T. N. Kipf, "Deep Learning with Graph-Structured Representations," Ph.D. dissertation, University of Amsterdam, 2020.

[12] Z. Wu *et al.*, "A Comprehensive Survey on Graph Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[13] F. Scarselli, M. Gori *et al.*, "The Graph Neural Network Model," *IEEE Transactions on Neural Networks*, 2009.

[14] T. N. Kipf *et al.*, "Semi-Supervised Classification with Graph Convolutional Networks," *ArXiv*, 2017.

[15] W. L. Hamilton *et al.*, "Inductive Representation Learning on Large Graphs," *ArXiv*, 2018.

[16] P. Veličković *et al.*, "Graph Attention Networks," *ArXiv*, 2018.

[17] Z. Lin *et al.*, "A Structured Self-attentive Sentence Embedding," *ArXiv*, 2017.

[18] Y. Ma *et al.*, "Understanding Graphs in EDA: From Shallow to Deep Learning," in *ACM Proceedings of the 2020 International Symposium on Physical Design*, 2020.

[19] B. Khailany *et al.*, "Accelerating Chip Design With Machine Learning," *IEEE Micro*, 2020.

[20] L. Servadei *et al.*, "Cost Optimization at Early Stages of Design Using Deep Reinforcement Learning," ser. MLCAD '20, 2020.

[21] E. Ustun *et al.*, "Accurate Operation Delay Prediction for FPGA HLS Using Graph Neural Networks," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20, 2020.

[22] Y. Zhang *et al.*, "GRANNITE: Graph Neural Network Inference for Transferable Power Estimation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[23] A. Mirhoseini *et al.*, "A Graph Placement Methodology for Fast Chip Design," *Nature*, 2021.

[24] Z. Xie *et al.*, "Net2: A Graph Attention Network Method Customized for Pre-Placement Net Length Estimation," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021.

[25] Y.-C. Lu *et al.*, "VLSI Placement Optimization using Graph Neural Networks," 2020.

[26] A. Agnesina *et al.*, "A General Framework for VLSI Tool Parameter Optimization with Deep Reinforcement Learning."

[27] ——, "VLSI Placement Parameter Optimization Using Deep Reinforcement Learning," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20, 2020.

[28] R. Kirby *et al.*, "CongestionNet: Routing Congestion Prediction Using Deep Graph Neural Networks," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2019.

[29] 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, 2019.

[30] 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.

[31] G. Zhang *et al.*, "Circuit-GNN: Graph Neural Networks for Distributed Circuit Design," in *International Conference on Machine Learning*, 2019.

[32] H. Wang *et al.*, "GCN-RL Circuit Designer: Transferable Transistor Sizing with Graph Neural Networks and Reinforcement Learning," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[33] Y. Li *et al.*, "A Customized Graph Neural Network Model for Guiding Analog IC Placement," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020.