VLSI Placement Optimization using Graph Neural Networks

Yi-Chen Lu yclu@gatech.edu
Sai Pentapati sai.pentapati@gatech.edu
Sung Kyu Lim limsk@ece.gatech.edu

Department of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA, USA

Abstract

Placement is one of the most crucial problems in modern Electronic Design Automation (EDA) flows, where the solution quality is mainly dominated by on-chip interconnects. To achieve target closures, designers often perform multiple placement iterations to optimize key metrics such as wirelength and timing, which is highly time-consuming and computationally inefficient. To overcome this issue, in this paper, we present a graph learning-based framework named PL-GNN that provides placement guidance for commercial placers based on logical affinity among design instances. Experimental results on commercial multi-core CPU designs demonstrate that our framework improves the industry-standard placement flow by 3.9% in wirelength, 2.8% in power, and 85.7% in worst negative slack reduction.

1 Introduction

Placements of Application-Specific Integrated Circuits (ASICs) require designers to place millions or even billions of gate-level instances (VLSI netlists) on constrained physical layouts, which directly impacts the quality of the final full-chip design. With ever increasing design complexity driven by Moore's Law, commercial EDA tools are struggled with achieving high-quality placements without spending significant amount of time in optimization iterations, which severely bottlenecks the chip design process. Recently, "placement guidance" emerges as a promising approach to perform placement optimization in modern physical design flows. It is achieved by informing commercial placers about the instances that should be placed nearby in actual physical layouts in order to optimize key design metrics such as wirelength, congestion, and timing. With the given information, placers will spend effort in grouping those cells together during the placement process. However, performing placement guidance usually requires in-depth design-specific knowledge, which is only achievable by experienced designers who have deep understanding of the underlying data flow in Register-Transistor Level (RTL).

To overcome the above issue, in this paper, we present a universal framework named PL-GNN that provides automated and accurate placement guidance for any design. PL-GNN consists two stages. First, given a netlist, we perform unsupervised node representation learning using graph neural networks (GNNs), where the goal is to learn accurate node representations that are related to the netlist logical affinity. Then, we leverage the weighted K-means clustering algorithm [3] to group instances into different clusters based on the obtained representations from graph learning. Finally, the clustering results are utilized as placement guidance for a commercial placer. In this work, we target the renowned commercial physical design tool *Synopsys IC Compiler II (ICC2)* [13] as our baseline, and demonstrate that the proposed framework significantly improves the default placement flow of ICC2 on commercial multi-core CPU designs.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), ML for Systems Workshop.

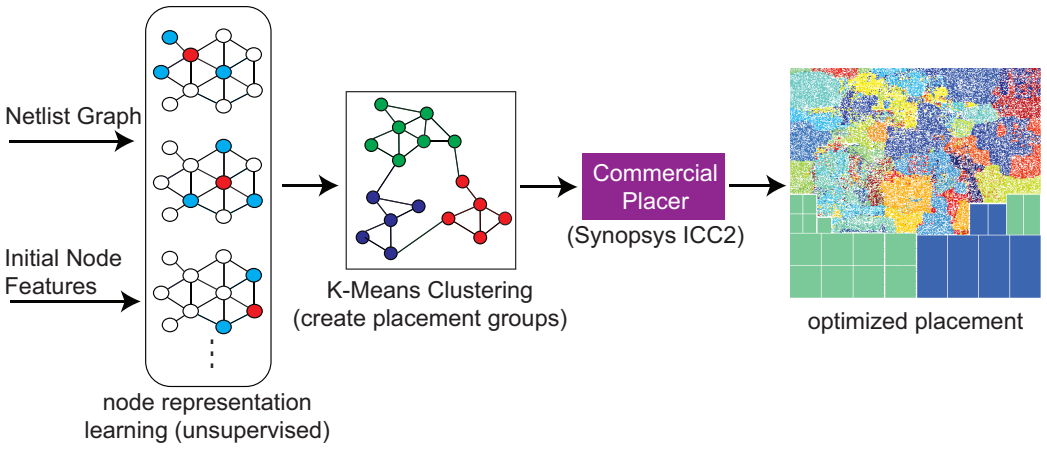

Figure 1: Overview of PL-GNN framework. Given a netlist graph and the initial node features, we first perform node representation learning to transform the initial features into better representations that accurately characterize the underlying design knowledge. Then, with the learned node embeddings, we perform weighted K-means clustering to determine the placement groups as placement guidance for a commercial placer. Based on the provided grouping information, the placer will spend effort in placing the instances in a common group together during global and detailed placements.

2 Related Learning-Based Placement Optimization Works

Recently, the authors of [10] propose DREAMPlace, which utilizes GPUs with deep learning toolkits to significantly accelerate the runtime of analytical placers. However, the proposed method does not improve the solution quality since the underlying placement algorithms remains the same. To optimize placement quality, the authors of [7] map the traditional placement problem into a reinforcement learning (RL) problem and present the usage of applying GNNs to encode netlist features. In [11], a complete RL framework is proposed to perform floorplanning for memory macros of Google TPU designs, where a force-directed method is introduced to place standard cells. It is shown that the achieved final designs through RL agents outperform the ones built by designers in much shorter turn-around-time. Nonetheless, the proposed RL framework [11] only focuses on optimizing the locations of memory macros, where the logical affinity among standard cells are the most dominated factor to achieve high-quality placements. As for placement prediction, another work [6] leverages the Louvain modularity-based clustering method [1] to predict placement relevant cell clusters, where the goal is to predict the instances that will be placed nearby in the actual physical layouts. They demonstrate that the adopted clustering method better predicts the final placement results than the renowned k-way partitioning algorithm [9] (hMETIS) under evaluations of Davies–Bouldin index (DBI) [2]. However, the applications of such prediction are limited, because it is subject to a fixed placement flow, which means when the flow is changed, the prediction will be inaccurate.

3 PL-GNN Framework

3.1 Overview

Figure 1 presents a high-level overview of PL-GNN framework. Since VLSI netlists are originally represented as hypergraphs, given a netlist, we first transform the directed hypergraph into an undirected clique-based graph, where a net that originally contains k cells will form a k-clique. Then, based on the transformed graph and the initial node features defined for each instance, we leverage GraphSAGE [8], a variant of GNNs, to perform unsupervised node representation learning. After the learning is complete, we leverage the weighted K-means clustering algorithm [3] to determine the placement groups based on the learned representations, where the cell area is taken as the weight. Finally,the placement groups are taken as the placement guidance for the commercial placer (*Synopsys ICC2*) to optimize the final placement quality. Note that the placement of memory macros (floorplanning) is achieved based on design manuals. This work focuses on improving global and detailed placements of standard cells.

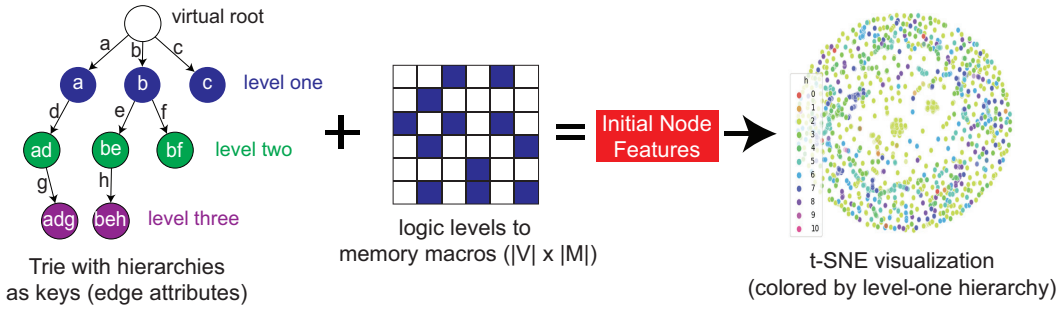

Figure 2: Construction and visualization of initial node features (colored in red), which are obtained from design hierarchy and logical affinity of memory macros. Alphabets on the edges of the trie structure denote hierarchies at different levels, where each node has a unique encoding obtained by concatenating the edge attributes on the path starting from the root to itself. Note that the initial features are further transformed to better representations through graph learning.

3.2 Initial Node Features

Prior to the graph learning process, given an undirected graph $G = (V, E)$, as shown in Figure 2, we determine an initial feature vector for each instance $v \in V$ based on its hierarchy information and the logical affinity with memory macros M in the design. To encode the hierarchy information, we implement a trie [4] (suffix graph) data structure, where the keys are the hierarchies in different levels. Since in a gate-level netlist, the name of a design instance takes a combination of multiple hierarchies as its prefix, there is a unique mapping from an instance in the design to a node in the trie. Note that since the length of the node attributes varies in the trie, we perform zero-padding to ensure every instance to have a common length of features. The reason we take hierarchy information as features is because instances with a common hierarchy tend to have more connections compared with those in different hierarchies, and these interconnects dominate the placement quality. Apart from the hierarchy information, for each design instance v, we also take its logical levels to memory macros M as features, which results in a vector in $R^{|M|}$. The reasons we introduce the memory related features is because the logic to memory paths are often the critical timing paths. Finally, we concatenate the hierarchy features with the memory features to form the initial node representations.

3.3 Node Representation Learning

With the initial node features presented, we perform node representation learning using GraphSAGE [8]. The goal of graph learning is to obtain the node representations that better characterize the underlying design knowledge than the initial features as shown in Figure 2. To learn a better representation for each design instance v, we leverage GNNs to sample and aggregate the features within its local neighborhood $N(v)$ through:

$$h_v^k = \sigma\left(h_v^{k-1} + \theta_k \cdot \frac{1}{s_k} \sum_{u \in N_k(v)} h_u^{k-1}\right), \tag{1}$$

where σ is the sigmoid function, h_v^k denotes the representation vector of node v at level k, $N_k(v)$ denotes the neighbors sampled at k-hop, s_k denotes the corresponding sampling size, and θ_k represents the parameters of the neural network (NN) at k-hop (each hop has its own NN). Note that the concept of "level" is corresponding to the concept of "hop", where h_v^0 is the initial features of node v, and $h_v^{k=K}$ is the final representation after aggregation the information within the K-hop neighborhood of v. Note that the feature aggregation process is performed iteratively, where for each level (hop) k, a dedicated NN layer (parameterized by θ_k) aggregates the neighboring features within $N(v)$ at the $k-1$ level. These NN layers together form the GNN module in our framework. In the implementation, our GNN module has two layers and each with 128 neurons. To update the parameters $\{\theta_k\}$, we introduce an unsupervised loss function $\mathcal{L}$ as

$$\mathcal{L}(h_v) = -\sum_{u \in N(v)} \log(\sigma(h_v^\top h_u)) - \sum_{i=1}^{M} \mathbb{E}_{n_i \sim Neg(v)} \log(\sigma(-h_v^\top h_{n_i})), \tag{2}$$

Table 1: Placement optimization impact on commercial CPU designs. The baseline flow is denoted as "ICC2 default", where we compare our graph learning-based method with the modularity-based method [1] to perform placement guidance on the baseline flow.

Design Name	Method	# of clusters	Wirelength (m)	WNS (ns)	TNS (ns)	Total Power (mW)
CPU-Design-A (# cells: 202k, # macros: 21)	ICC2 default	-	4.37	-0.07	-0.22	142
	modularity [1]	82	4.34	-0.10	-0.62	141
	PL-GNN (ours)	22	4.20	-0.01	-0.03	138
CPU-Design-B (# cells: 537k, # macros: 29)	ICC2 default	-	11.66	-0.24	-240.39	582
	modularity [1]	58	11.65	-0.38	-296.54	578
	PL-GNN (ours)	32	11.55	-0.18	-62.21	574

where $Neg(v)$ represents the negative sampling distribution of node v, and M represents the negative sampling size. This negative sampling technique is known to improve the efficiency of graph learning with faster loss convergence. Essentially, Equation 2 encourages nodes that share common neighborhoods to have similar representations, and penalizes similarity to the ones that are distant.

3.4 Clustering for Placement Guidance

Finally, after obtaining the learned representations, we leverage the weighted K-means clustering algorithm [3] to cluster design instances into placement groups, where the cell area is taken as the weight. To determine the optimal number of clusters (the optimal K), we perform sweeping experiments from $k = 8$ to $k = 32$ based on the Silhouette metric [12]. In addition, to perform fair comparisons with the previous work [6] that predicts placement relevant clusters, we also implement the modularity-based clustering algorithm [1] to create placement groups for placement guidance.

4 Experimental Results

We validate the proposed framework, PL-GNN, on two commercial multi-core CPU designs in the *TSMC 28nm* technology node. Due to the confidentiality, we name the two designs as "CPU-Design-A" and "CPU-Design-B", respectively. In the experiments, we take the default placement flow in *Synopsys IC Compiler II (ICC2)* as our baseline, and demonstrate the placement guidance results achieved by PL-GNN and a modularity-based clustering algorithm [1]. In ICC2, the placement groups are created by the command "*create_placement_attraction {instance_list}*". PL-GNN and the modularity-based method [1] is implemented in *Python3* with the *PyTorch Geometric* [5] library.

The experimental results are shown in Table 1. Compared with the default placement flow in ICC2, PL-GNN achieves up to 3.9% wirelength, 2.8% power, and 85.7% performance improvements. Furthermore, we demonstrate that PL-GNN outperforms the modularity-based method [1] adopted by previous work [6], which is mainly because [1] simply focuses on grouping the design instances based on connectivity, where PL-GNN not only considers the underlying logical affinity when performing placement guidance, but also takes the features (presented in Figure 2) that are crucial to the final placement quality into account. Note that the number of clusters in [1] is determined by maximizing the modularity heuristic. Finally, we want to emphasize that PL-GNN is able to generalize to *any design*, since it performs the placement guidance is by optimizing an unsupervised loss function.

5 Discussions

The superior results achieved by our framework PL-GNN can be accounted in two-fold. First, the initial node features accurately capture the underlying instance characteristics that are highly related to achieving high-quality placements. Specifically, the macro-related features for each design instance denoted by logic levels are particularly critical to the final timing results. Second, GNNs are highly effective in encoding graph structures with node attributes. Since the final physical location of a design instance highly depends on the local neighborhood structure, it is especially suitable to leverage GNNs to encode such information. In conclusion, we have presented a graph learning-based placement optimization framework and demonstrate that the framework significantly improves the standard placement flow in an industrial-leading tool *Synopsys ICC2* on commercial CPU designs.

References

[1] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment, 2008(10):P10008, 2008.

[2] D. L. Davies and D. W. Bouldin. A cluster separation measure. IEEE transactions on pattern analysis and machine intelligence, pages 224–227, 1979.

[3] R. C. De Amorim and B. Mirkin. Minkowski metric, feature weighting and anomalous cluster initializing in k-means clustering. Pattern Recognition, 2012.

[4] R. De La Briandais. File searching using variable length keys. In Papers presented at the the March 3-5, 1959, western joint computer conference, pages 295–298, 1959.

[5] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. arXiv preprint arXiv:1903.02428, 2019.

[6] M. Fogaça, A. B. Kahng, R. Reis, and L. Wang. Finding placement-relevant clusters with fast modularity-based clustering. In Proceedings of the 24th Asia and South Pacific Design Automation Conference, pages 569–576, 2019.

[7] A. Goldie and A. Mirhoseini. Placement optimization with deep reinforcement learning. In Proceedings of the 2020 International Symposium on Physical Design, pages 3–7, 2020.

[8] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In Advances in Neural Information Processing Systems, 2017.

[9] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. VLSI design, 11(3):285–300, 2000.

[10] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Z. Pan. Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020.

[11] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae, et al. Chip placement with deep reinforcement learning. arXiv preprint arXiv:2004.10746, 2020.

[12] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of computational and applied mathematics, 20:53–65, 1987.

[13] I. Synopsys. Compiler user guide, 2019.