# The Law of Attraction: Affinity-Aware Placement Optimization using Graph Neural Networks

Yi-Chen Lu
yclu@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia USA

Sai Pentapati
sai.pentapati@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Sung Kyu Lim
limsk@ece.gatech.edu
Georgia Institute of Technology
Atlanta, Georgia 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 by generating cell clusters based on logical affinity and manually defined attributes of design instances. With the clustering information as a soft placement constraint, commercial tools will strive to place design instances in a common group together during global and detailed placements. Experimental results on commercial multi-core CPU designs demonstrate that our framework improves the default placement flow of *Synopsys IC Compiler II (ICC2)* by 3.9% in wirelength, 2.8% in power, and 85.7% in performance.

## CCS CONCEPTS

• **Hardware → Electronic design automation**; **Methodologies for EDA**; **Placement**.

**ACM Reference Format:**
Yi-Chen Lu, Sai Pentapati, and Sung Kyu Lim. 2021. The Law of Attraction: Affinity-Aware Placement Optimization using Graph Neural Networks. In *Proceedings of the 2021 International Symposium on Physical Design (ISPD '21), March 22–24, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3439706.3447045

## 1 INTRODUCTION

Placements of Application-Specific Integrated Circuits (ASICs) require designers to place millions or even billions of gate-level instances on constrained physical layouts, which are performed by sophisticated commercial tools in modern physical design (PD) flows. However, with the 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 performing placement iterations to achieve target closures.


*ISPD '21, March 22–24, 2021, Virtual Event, USA*

ACM ISBN 978-1-4503-8300-4/21/03…$15.00
https://doi.org/10.1145/3439706.3447045

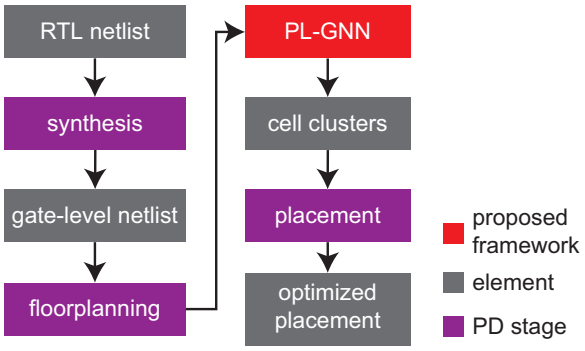


**Figure 1: PL-GNN powered design flow. The cell clusters determined by our framework PL-GNN are taken as placement guidance. During placement, *Synopsys ICC2* will spend effort in grouping the cells within a common effort together.**

It is well known that placement directly impacts the final quality of a full-chip design, and the logical affinity among design instances dominates the quality of the placement. To achieve a high-quality placement in terms of key quality of result (QoR) metrics, designers have to understand the underlying data flows in order to set instructions for commercial tools to place the design instances accordingly. In modern PD flows, this process is called "placement guidance", which heavily relies on the knowledge of experienced designers.

In recent years, placement guidance has become a must-use step to achieve high-quality placements in the semiconductor industry. It optimizes default placement flows in commercial tools by informing placers about the design instances that should better be placed nearby in actual physical layouts in order to optimize key design metrics. With the given grouping information, commercial placers will spend effort in grouping the cells in a common cluster together during the placement process. However, as mentioned, performing placement guidance requires in-depth design-specific knowledge, which is only achievable by experienced designers who knows the underlying data flows in Register-Transistor Level (RTL) well.

To overcome the above issue, in this paper, we present a universal placement optimization framework named PL-GNN that provides automated and accurate placement guidance for any design without requiring users to have profound design knowledge. Figure 1 shows the PL-GNN powered design flow, where our framework will determine the cell clusters in an unsupervised manner which serve as placement guidance in order to guide commercial placers to optimize the key metrics such as wirelength, power, and timing by placing cells with a common cluster together. PL-GNN is consisted of two stages. First, given a netlist, we perform unsupervised node

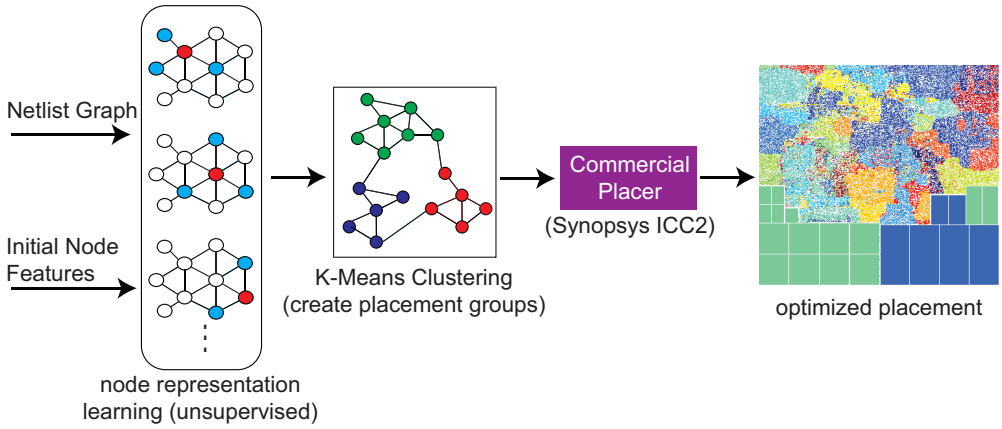


**Figure 2: Overview of PL-GNN framework. Given a netlist graph and the initial node features, we first perform node representation learning to transform the initial features into better representations that accurately characterize the underlying design knowledge. Then, with the learned node embeddings, 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.**

representation learning using graph neural networks (GNNs) based on the initial features manually defined for each design instance. The goal of node representation learning is to learn accurate node representations that are related to the underlying logical affinity and attributes of a given netlist. In the second stage, based on the learned representations, we leverage the weighted K-means clustering algorithm [3] to group instances into different clusters. To find the optimal number of groups for clustering, we introduce the Silhouette score [19] and perform sweeping experiments to find the sweet spot. As aforementioned, the final clustering results are utilized as placement guidance for commercial placers. In this work, we target the renowned commercial physical design tool *Synopsys IC Compiler II (ICC2)* [20] as our baseline, and demonstrate that the proposed framework significantly improves the default placement flow of *ICC2* on commercial multi-core CPU designs.

The goal of this work is to provide designers a placement optimization framework that achieves high-quality placements for general designs by distilling underlying design knowledge. Note that since our framework learns the node representation for every design instance by optimizing an unsupervised loss function, it is generalizable to *any* design. In addition, PL-GNN does not assume any pre-defined netlist structure. Instead, it adapts to different netlists through novel graph embedding techniques. Finally, although we take *Synopsys ICC2* as the reference tool in this work, our framework can easily be integrated with other physical design tools to significantly improve the placement quality.

## 2 RELATED WORKS AND MOTIVATIONS

### 2.1 Learning-Based Placement Optimization

Recently, the authors of [15] 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 placement solution quality because 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 [17], 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 [17] 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. Another work [25] proposes a detailed placement optimization technique based on the prediction of pin assignment, where the goal is to fine-tune the placement with a pre-trained model for minimization of design rule violations (DRVs) after routing. However, since the pre-trained model is obtained through a online and supervised manner, the proposed method is subject to the underlying design flow. Furthermore, the improvement on the placement quality with the incremental update of the pre-trained model is minor.

### 2.2 Placement Prediction

As for placement prediction, previous work [6] proposes a new hypergraph to clique-based graph transformation model and leverages the Louvain modularity-based clustering method [1] to predict placement relevant cell clusters, where the goal is to predict the design 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 [12] (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. Another work [16] develops a method to encode placement features using transfer learning with layout images. However,

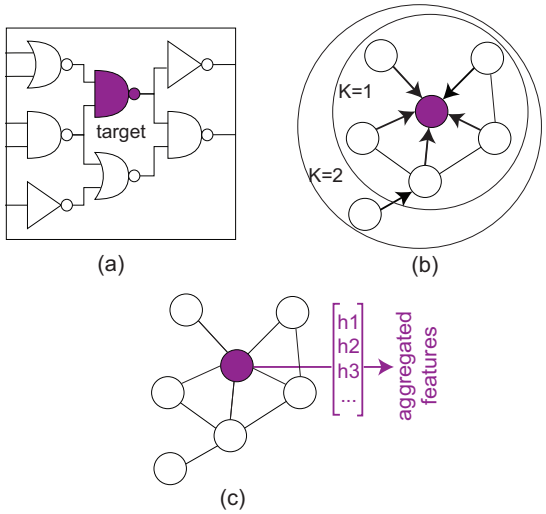


**Figure 3: Illustration of GNN aggregation process on a VLSI netlist. Given a netlist as shown in (a), we first transform the directed hypergraph (original representation of the netlist) into an undirected clique-based graph as shown in (b). Then, based on the clique-based graph, for each node, we perform feature aggregation on its neighborhood from $K = \{1, 2, ..., K\}$, and finally (c) obtain the final representations.**

the presented encoding method is at graph-level, where a single design is encoded into one single vector. Therefore, it cannot be utilized to cluster instances within a design.

### 2.3 Motivations

In this paper, we aim to overcome all the drawbacks presented above. We develop a graph learning-based placement optimization framework that significantly improves the standard industrial placement flow. Unlike [17] that uses RL to optimize macro placement, our framework focuses on optimizing the standard cell placement (global and detailed placements) by considering the netlist affinity and node-level hierarchy features. In addition, we present a detailed comparison with previous work [6] and demonstrate that the proposed graph learning-based technique better guide the commercial placer (ICC2) to optimize placement quality than the modularity-based clustering method [1].

## 3 PL-GNN FRAMEWORK

### 3.1 Overview

GNNs are powerful to encode underlying graph knowledge into representative knowledge. They perform effective graph representation learning by aggregating the features from one node with its neighbors (not limited to direct 1-hop neighbors) in a message passing scheme. The initial features of a node are thus being transformed iteratively into better representations that are related to the objective of feature aggregation. These learned representations (transformed features) can be further utilized in downstream tasks such as node classification, clustering, or link prediction. In this paper, we devise an unsupervised loss function that serves as an objective for the feature aggregation process. With the aggregated features, we leverage the weighted K-means clustering algorithm [3] to determine the standard cells clusters that should better be placed nearby.

Figure 2 presents a high-level overview of our 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 clique-based graph and the initial node features we define for each instance, we leverage GraphSAGE [9], a variant of GNNs, to perform unsupervised node representation learning. GNN can be considered as a "graph filter" that iterates through every design instance to transform its initial features into better representations by aggregating its neighboring information. Figure 3 demonstrates the illustration of graph learning process on a VLSI netlist. First, as aforementioned, we transform the hypergraph (Figure 3 (a)) into a clique-based graph (Figure 3 (b)). Then, we leverage GNNs to perform node representation learning as shown in Figure 3 (b). In this work, our GNN has two layers, where each of them is dedicated to aggregate the neighboring features at a specific hop of neighborhood. Finally, after the graph 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.

The placement groups determined by the clustering algorithms are taken as the "placement guidance" for the commercial placer, where in this work we take *Synopsys ICC2* as the baseline flow. This clustering information is expected to help the commercial tool optimize placement quality by placing instances in a common cluster together in the actual physical layout. The key idea is that instances in a common cluster will have stronger affinity to those in different clusters, since they are being grouped by performing graph representation learning, which transforms the initial features by distilling the underlying design knowledge it terms of logical affinity. Therefore, by knowing which instances should better be placed together based on their affinity, the commercial tool will be able to insert less buffers to meet timing constraints, because cells with stronger affinity usually means that they have more connections. Note that in this work, the placement of memory macros is achieved manually based on design manuals provided by the design-house. This work focuses on improving global and detailed placements of standard cells. In the following sections, we illustrate the underlying algorithms in detailed.

### 3.2 Graph Model for Netlist Transformation

As mentioned above, VLSI netlists are originally represented as a directed hypergraph, which is not applicable for many graph optimization techniques. Therefore, throughout the years, extensive research has been conducted extensively to find appropriate graph models to transform a netlist from a directed hypergraph into a "normal" graph representation where one edge only contains two vertices. The clique-based model is one of the most popular graph transformation model [22] where the edge weight $w_e$ in the

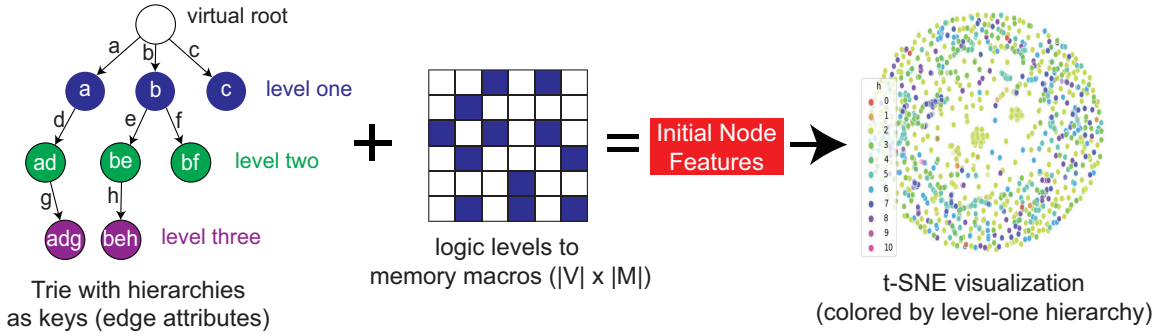


**Figure 4: Construction and visualization of initial node features (colored in red), which are obtained from design hierarchy and logical affinity of memory macros. Alphabets on the edges of the trie structure denote hierarchies at different levels, where each node has a unique encoding obtained by 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.**

transformed undirected graph has a weight as

$$w_e = \frac{1}{|n_e| - 1}, \tag{1}$$

where $|n_e|$ denotes the number of gates the edge (net) is connected to in the original hypergraph. The key rationale of this transformation is to keep the total weight of a net consistent between the two graphs. Many renowned algorithms [8, 13] adopt this clique-based model to solve the graph partitioning problem. Still, other improved transformation techniques [10, 21] are developed to tackle the ever-evolving crucial EDA problems. However, although the above approaches aim to find the best transformation model for general EDA problems, in [11], it is proven that there is no transformation that preserves the original information intact. In this paper, we adopt the transformation model based on Equation 1 to transform the original netlist into an undirected clique-based graph prior to the graph learning process.

### 3.3 Initial Node Features

Prior to the graph learning process, given an undirected graph $G = (V, E)$, as shown in Figure 4, 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. For example, a cell may have a name as "a/d/g", where "a" is the first-level hierarchy, "d" is the sub-hierarchy of "a", and "g" is the instance name defined in "d". The combination of "a", "d", and "g" ("a/d/g") is unique, but each of them may be not (e.g. there might be two or more edges in the trie with the attribute "g"). 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.

### 3.4 Node Representation Learning

With the initial node features presented, we now illustrate the process of performing node representation learning using GraphSAGE [9]. The goal of graph learning is to obtain the node representations $h_v^K$ ($K$ denotes the aggregation level) that better characterize the underlying design knowledge than the initial features $h_v^0$ for each node $vi \in V$, where $G = (V, E)$ represents the transformed clique-based graph. Figure 3 demonstrates the illustration of the graph learning process on a target node colored in purple, where the key idea is to leverage GNNs to aggregate the information from its neighboring nodes based on the underlying logical affinity with consideration of the node attributes (features). This aggregation process is performed iteratively, where for each level (hop) $K = k$, there is a GNN layer (a one-layer neural network) dedicated to aggregate the features at the specific hop of neighborhood of the target node. These $K$ layers form the overall GNN module. Note that since the number of neighbors grow exponentially as the hop-count increases, to stabilize the training process and to prevent overfitting, we limit the number of neighbors $s_k$ to be aggregated at each level $k$.

For a node $v \in V$, the representations at level $k$ is obtained as:

$$\begin{aligned} h_{N_k(v)}^{k-1} &= \text{reduce\_mean}\left(\{\mathbf{W}_k^{agg} h_u^{k-1}, \forall u \in N_k(v)\}\right), \\ h_v^k &= \sigma\left(\mathbf{W}_k^{proj} \cdot \text{concat}[h_v^{k-1}, h_{N_v(v)}^{k-1}]\right), \end{aligned} \tag{2}$$

where $\sigma$ 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 which is subject to the sampling size $s_k$, $W_k^{agg}$ and $W_k^{proj}$ denote the aggregation and projection matrices respectively, which together form the neural layer at level $k$. Note that the concept of "level" is corresponding to the concept of "hop", where $h_v^0$ is the

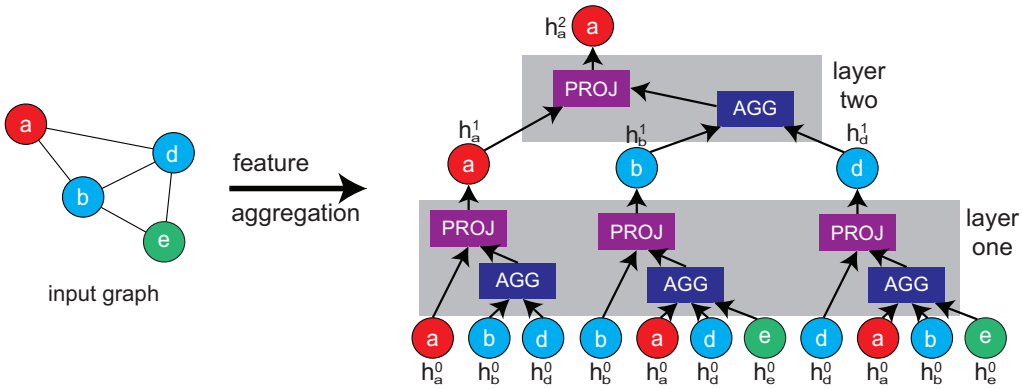


**Figure 5: Illustration of feature aggregation process in node representation learning. We leverage a two-layer GNN to determine the final representation of node "a" in the input graph by considering information within its 2-hop neighborhood. AGG denotes the aggregation matrix and PROJ denotes the projection matrix in Equation 2. Gradient descent is utilized to update the parameters of these two matrices by minimizing Equation 3.**

initial features of node $v$, and $h_v^{k=K}$ is the final representation after aggregation the information within the $K$-hop neighborhood of $v$. Figure 5 demonstrates the illustration of the feature aggregation process of the target node "a" (colored in red) in the input graph, which is shown that GNN is not a traditional fully-connected network, since all the node features of the previous level have to be processed through Equation 2 before performing the aggregation of the current level. In the implementation, our GNN module has two layers, which means for each design instance we would perform feature aggregation within its two-hop neighborhood to obtain better representations. These transformed features are further utilized to cluster design instances into placement groups through the weighted K-means clustering algorithm [3].

## 3.5 Unsupervised Loss Function

Previous sub-section introduces the forward process of graph learning. To update the parameters $\{W_k\}$ during back propagation, we introduce an unsupervised loss function $\mathcal{L}$ as the objective function, where $\mathcal{L}$ takes the form of

$$\mathcal{L}(h_v) = -\sum_{u \in N(v)} \log(\sigma(h_v^\top h_u)) - \sum_{i=1}^{M} \mathbb{E}_{n_i \sim Neg(v)} \log(\sigma(-h_v^\top h_{n_i})), \tag{3}$$

where $Neg(v)$ represents the negative sampled nodes in the perspective node $v$, and $M$ represents the negative sampling size. The negative sampled nodes are the nodes that are distant (not within 2-hop neighborhood) from the target node $v$ in the clique-based graph, and at each iteration, these nodes are re-sampled. The reason we introduce this negative sampling technique is because we not only want to enhance the similarity between the target node and its neighbors, but also want to maximize the dissimilarity of the target node with the nodes that are distant from it. This negative sampling technique is known to help improve the efficiency of graph learning by providing faster loss convergence. Essentially, Equation 3 encourages nodes that share common neighborhoods to have similar representations, and penalizes similarity to the ones that are distant. By minimizing Equation 3 using gradient descent, we can update the parameters in the GNN module. Note that since our objective function is defined in an unsupervised manner, our framework has the ability to adapt to various netlists since it does not require any pre-train process.

## 3.6 Training Methodology

Algorithm 1 summarizes the graph learning process of our PL-GNN framework. Lines 3–10 illustrate the forwarding process of graph learning (feature aggregation), where for each node $v \in V$, we transform its initial features $h_v^0$ to $h_v^K$ by aggregating its neighboring features at each level (hop) through Equation 2. Note that prior to the aggregation at each level, we normalize the node representations at previous level in Line 2 and Line 9. This normalization helps improve the convergence of the overall training process by reducing the oscillation of gradient descent. Finally, based on the learned representation vectors, in Lines 12–19 we calculate the unsupervised loss based on the aggregated features by Equation 3, where a negative sampling technique (Lines 14–15) is leveraged to improve the overall training process. Finally, to update the parameters in the the framework, we leverage *Adam* [14], a renowned gradient descent optimizer to minimize the loss function. The overall training process takes about an hour on each CPU design utilized in this work based on a machine with a 2.40$GHZ$ CPU and a NVIDIA RTX 2070 graphic cards with 16$GB$ memory.

## 3.7 Complexity Analysis of Graph Learning

**Time Complexity.** The time complexity of the proposed framework is linear with respect to the netlist size. Since the sampling size $s_k$ at each aggregation level is fixed, our GNN module which acts as a "graph filter" only spends constant amount of time in visiting every design instance and collecting features from its neighbors.
**Space Complexity.** Instead of storing the graph connectivity in a $|V|x|V|$ matrix which requires $O(n^2)$ space complexity, we store the connectivity information in the compressed sparse row (CSR) format [23] thanks to the high sparsity of the netlist adjacency matrix. Therefore, the space complexity is far less than $O(n^2)$, which can be considered as pseudo-linear.

**Algorithm 1** Graph learning in PL-GNN.
We use default values of $\alpha = 0.001, K = 2, M = 30, s_1 = 10, s_2 = 5, \beta_1 = 0.9, \beta_2 = 0.999$.

**Input:** $G(V, E)$: clique-based graph. $\{h^0\}$: initial node features. $\alpha$: learning rate, $K$: maximum aggregation level, $M$: negative sampling size, $\{s_k, \forall k \in \{1, ..., K\}\}$: k-hop neighborhood sampling size, $\sigma$: sigmoid function, $\{W_k, \forall k \in \{1, ..., K\}\}$: parameters of NN at hop (level) k, $\{\beta_1, \beta_2\}$: Adam parameters.
**Output:** $\{y\}$: learned node representations.

**while** $\{W_k\}$ do not converge **do**
  $h_v^0 \leftarrow \frac{h_v^0}{\|h_v^0\|_2}, \forall v \in V$
  **for** $k \leftarrow 1$ to $K$ **do** ▷ feature aggregation
    **for** $v \in V$ **do**
      $N_k(v) \leftarrow$ Sample $s_k$ neighbors at $k$-hop
      $h_{N_k(v)}^k = \text{reduce\_mean}\left(\{\mathbf{W}_k^{agg} h_u^{k-1}, \forall u \in N_k(v)\}\right)$
      $h_v^k = sigmoid\left(\mathbf{W}_k^{proj} \cdot \text{concat}[h_v^{k-1}, h_{N_v(v)}^k]\right)$
    $h_v^k \leftarrow \frac{h_v^k}{\|h_v^k\|_2}, \forall v \in V$
  $y_v \leftarrow h_v^K, \forall v \in V$
  **for** $v \in V$ **do** ▷ minimize unsupervised loss
    **for** $u \in N(v)$ **do**
      $N_k(v) \leftarrow$ Sample $M$ samples from $\{V - N(v)\} \setminus v$
      $neg\_loss \leftarrow \sum_{n_i \in N_k(v)} \log(\sigma(-y_v^\top y_{n_i}))$
      $g_v \leftarrow \nabla_W [\log(\sigma(y_v^\top y_u)) + neg\_loss]$
      $\{W_k\} \leftarrow Adam(\alpha, \{W_k\}, g_v, \beta_1, \beta_2)$

**Algorithm 2** Weighted K-means Clustering.
Learned representations $\{h_v\}$ from Algorithm 1 are taken as $\{y\}$. Cell areas are taken as node weights $\{w\}$.

**Input:** $G(V, E)$: clique-based graph, $\{w\}$: node weights, $\{y\}$: node representations, $k$: number of clusters.
**Output:** $\{C_1, ..., C_k\}$: $k$ clusters.

Select $k$ initial centroids $\{c_1, ..., c_k\}$ randomly
**repeat**
$\{C_1, ..., C_k\} = \text{argmin}_C \sum_{i=1}^{k=2} \sum_{v \in C_i} w(v) \|y_v - c_i\|^2$
$c_i = \frac{\sum_{v \in C_i} y_v w(v)}{\sum_{v \in C_i} w(v)}, \forall i = 1, ..., k$
**until** $\{C_1, ..., C_k\}$ no longer change

## 3.8 Clustering for Placement Guidance

After obtaining the learned representations, we leverage the weighted K-means clustering algorithm [3] to cluster design instances into placement groups to perform placement guidance in *Synopsys ICC2*. Algorithm 2 summarizes our clustering algorithm, where the cell areas of design instances are taken as the node weights. Given the learned node representations $\{y\} = \{h^{k=3}\}$ from Algorithm 1 and the weights $\{w\}$ as cell areas, the algorithm will strive to cluster all the nodes $V$ into $k$ weight-balanced groups by minimizing the Euclidean distance of every node to its assigned centroid. The objective function of clustering is derived as

$$\mathcal{L}_{kmean} = \sum_{i=1}^{k} \sum_{v \in C_i} w(v) \cdot \|y_v - c_i\|^2, \tag{4}$$

where $c_i = \frac{\sum_{v \in C_i} y_v w(v)}{\sum_{v \in C_i} w(v)}$ denotes the weighted centroid of cluster $C_i$. Equation 4 is updated in an iterative manner. The key idea is that in each iteration, we assign each node belongs to the cluster that it has the minimum distance with, where the procedure works as follows:

- Starting from an initial centroids $\{c_1, ..., c_k\}$, for each iteration, we determine the clusters $\{C_1, ..., C_k\}$ by assigning each node to the centroid that has the minimum weighted distance (Line 3).
- After the assignments, we update the centroids based on the newly obtained clusters (Line 4).
- Repeat previous two steps until the locations of the centroids no longer change.

To determine the optimal number of clusters (the optimal $K$), we perform sweeping experiments from $k = 8$ to $k = 32$ based on the Silhouette score [19]. Here, we explain the calculation of the Silhouette score given a clustering result, which is consisted of two parts. In the first part, for a node $v$, we calculate the average distance $a(v)$ between it and other nodes in a common cluster, which is derived as:

$$a(v) = \frac{1}{|C_v| - 1} \sum_{i \in C_v, v \neq i} \|y_v - y_i\|^2, \tag{5}$$

where $C_v$ represents that cluster that node $v$ belongs to. The average distance $a(v)$ (Equation 5) essentially represents how well the node $v$ is assigned to its current cluster. Note that the reason the sum of distance is weighted by $\frac{1}{|C_v|-1}$ is because the distance from $v$ to itself is not included in the summation. Besides $a(v)$, we calculate another metric $b(v)$ which represents the smallest distance of $v$ to every other node in a different cluster $C_v^{'}$, which is derived as:

$$b(v) = \min_{k \neq i} \frac{1}{|C_k|} \sum_{i \in C_k} \|y_v - y_i\|^2. \tag{6}$$

In Equation 6, the cluster $C_k$ that results in the minimum $b(v)$ represents the second-best cluster for the target node $v$. Therefore, the greater $b(v)$ is, the better the target node $v$ is assigned to its current cluster. Finally, the Silhouette score for a node $v$ is defined as

$$s(v) = \frac{b(v) - a(v)}{max\{a(v), b(v)\}}. \tag{7}$$

Note that Equation 7 is defined in node-level. We take the average of $s(v)$ over every node (design instance) in the clique-based graph to evaluate how well the current clustering results is (the higher the average of $s(v)$, the better the result). To find the best number of clusters (optimal $K$ of K-Means), we perform sweeping experiments over $K$ and take the one that produces the highest Silhouette score as the final clustering solution for placement guidance.

# 4 MODULARITY-BASED CLUSTERING

In this work, besides leveraging the presented framework PL-GNN to determine cell clusters for placement guidance, we also implement the Louvain modularity-based clustering method [1] adopted by previous work [6] to perform placement guidance to perform comparison with our framework. The modularity metric is first

**Table 1: Our commercial benchmarks and their attributes in *TSMC 28nm*.**

| Design Name | # Cells | # Flip-Flops | # Nets | # Macros |
|---|---|---|---|---|
| CPU-Design-A | 202791 | 22366 | 206224 | 21 |
| CPU-Design-B | 537085 | 47552 | 542391 | 29 |

proposed in [18], which is expressed as

$$Q = \frac{1}{2m} \sum_{i,j} \left[ E_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j), \tag{8}$$

where $m$ denotes the total edge weights in the clique-based graph, $E_{ij}$ denotes the edge weight between node $i$ and node $j$, $k_i$ and $k_j$ denote the sum of edge weights for nodes $i$ and $j$, $C_i$ and $C_j$ denote the clusters nodes $i$ and $j$ belong to, and finally $\delta(\cdot)$ denotes the indicator cluster which returns 1 when the two clusters are connected, and 0 otherwise. Based on the modularity metric defined above, the Louvain clustering method [1] can efficiently group millions of instances into clusters with $O(n \cdot log^2 n)$ time complexity. The number of clusters are determined automatically by maximizing Equation 8. Therefore, no sweeping experiment is needed as in the clustering approach (Section 3.8) above. However, Louvain performs the optimization in a greedy manner, where small clusters are first found by performing optimization on local subgraphs, and then being merged greedily to determine the final solutions. Furthermore, this clustering method does not take cell attributes into account, which is purely achieved by minimizing connectives among clusters. Hence, it cannot comprehend the underlying design knowledge. In the experiments, we perform head-to-head comparison between the proposed framework PL-GNN and the modularity-based clustering algorithm Louvain on optimizing placement quality through placement guidance in *Synopsys ICC2*.

# 5 EXPERIMENTAL RESULTS

In this work, we validate the proposed framework PL-GNN on two commercial multi-core CPU designs in the *TSMC 28nm* technology node. Their attributes after performing synthesis from *Synopsys Design Compiler* are shown in Table 1. Due to the confidentiality, in this paper, we name the two designs as "CPU-Design-A" and "CPU-Design-B", respectively. As aforementioned, the memory macros of the two designs are placed by experienced designers based on design manuals from the design-house. In the experiments, we take the default placement flow in *Synopsys IC Compiler II (ICC2)* as our baseline, and demonstrate the placement optimization results achieved by our framework. Note that in ICC2, the placement groups are created by using the command "*create_placement_attraction {instance_list}*", which are taken as soft placement constraints by the commercial placer. With this clustering information, ICC2 will spend effort in grouping the cells in a common group together during global and detailed placements. Finally, the PL-GNN framework and the modularity-based method [1] is implemented in *Python3* with the *PyTorch Geometric* [5] library.

## 5.1 Optimization Results on CPU Designs

The detailed optimization results are shown in Table 2. Compared with the default placement flow in ICC2, the proposed framework PL-GNN achieves up to 3.9% wirelength, 2.8% power, and 85.7% performance improvements. Furthermore, it is demonstrated that the proposed method outperforms the modularity-based method Louvain [1] adopted by the previous work [6]. The main reason is that Louvain simply determines the clustering groups based on graph connectivity in a greedy manner as mentioned in Section 4, where our framework not only considers underlying logical affinity when determining the cell clusters for placement guidance, but also takes the node attributes (presented in Figure 4) that are crucial to the final placement quality into account.

## 5.2 Discussions

**Timing Perspective.** Timing is a highly critical objective during the placement stage in modern PD flows, which is as important as the minimization of wirelength. Timing quality of a placement directly impacts the final quality of a full-chip design. In the PD flow of *Synopsys ICC2*, prior to the placement stage, all the buffers inserted in the synthesis stage are removed in the gate-level netlist in order to give placement engine more freedom to achieve timing closures (same situation also happens in *Cadence Innovus*). The main rationale is that with the actual physical information of design instances, the commercial tool will have more accurate information regarding the strength or the number of buffers needed to meet timing closures compared with the synthesis stage. In Table 2, we observe that with the placement guidance provided by our framework, the placement engine in *ICC2* inserts significantly less buffers compared with the default placement flow. The reasons are two-fold. First, the features (defined in Figure 4) take the impact of memory macros into account for determine the placement groups, and the logic-to-memory paths are usually the critical timing paths. Therefore, with this information, the commercial placer is able to place the cells that could potentially lead to bad timing results together. Second, the hierarchy features also impact the timing results, because instances within a common hierarchy tend to have more connections that those in different hierarchies. Therefore, if instances in a common hierarchy are physically placed in distant, more buffers will need to be inserted to meet timing.

**Wirelength Perspective.** In the table, we also observe the wirelegnth gets improved as well. One of the reasons is that because fewer buffers are inserted as the reasons mentioned in the previous discussion, the net count of the optimized placement achieved by our framework will be smaller as well, which results in smaller wirelength as a by-product from the reduction of buffers. Another reason is that because the placement guidance provided by our framework PL-GNN is achieved by considering the underlying logical affinity of a given netlist, cells that have more connections will tend to be placed together, which reduces detours of routing.

# 6 WHY PL-GNN WORKS?

The superior achievements of our framework can mainly be accounted in two perspectives:

**Well-Defined Initial Features.** First, the initial node features presented in Figure 4 accurately capture the underlying characteristics of each design instance that are related to achieving high-quality placements. Although these features are not good enough to perform placement guidance as aforementioned, they provide precious

**Table 2: Placement optimization impact on commercial CPU designs, where "ICC2 default" represents the tool's default placement flow, and "Louvain" [1] denotes the modularity-based clustering method.**

| Design Name | Method | # of clusters | Wirelength (m) | WNS (ns) | TNS (ns) | Total Power (mW) | # inserted buffers |
|---|---|---|---|---|---|---|---|
| CPU-Design-A | ICC2 default | - | 4.37 | -0.07 | -0.22 | 142 | 5942 |
| | Louvain [1] | 82 | 4.34 | -0.10 | -0.62 | 141 | 5826 |
| | ours | 22 | 4.20 | -0.01 | -0.03 | 138 | 5371 |
| CPU-Design-B | ICC2 default | - | 11.66 | -0.24 | -240.39 | 582 | 2728 |
| | Louvain [1] | 58 | 11.65 | -0.38 | -296.54 | 578 | 2689 |
| | ours | 32 | 11.55 | -0.18 | -62.21 | 574 | 2274 |

information for the graph learning process. Based on these features, GNNs will know which are the nodes (instances) that are inherently similar to each other (e.g. hierarchy features), and will further transform them into better representations. Furthermore, by taking the logic levels to memory macros as features, GNNs will learn to balance the critical paths when performing feature aggregation, which results in fewer inserted buffers during placement.

**Superiority of Graph Learning.** Second, GNNs are highly effective in encoding graph structures with consideration of node attributes, and particularly in highly sparse graphs [24]. They capture invaluable latent knowledge on netlists that are hard to be observed even by experienced designers, but crucial to achieve high-quality placements. Since the final physical location of a design instance highly depends on the local neighborhood structure and its design attributes, GNNs become particularly suitable to encode such information by performing representation learning. Unlike modularity-based clustering algorithm [1] that groups design instances into clusters simply based on connectivity, our framework PL-GNN leverages GNNs to carefully distill the underlying design knowledge in order to accurately determine the cell clusters that achieve high-quality placements. In summary, the logical affinity among design instances directly dominates the quality of placement, and one should better encode, rather than fighting "the law of attraction" when performing placement.

## 7 CONCLUSION AND FUTURE WORK

In this paper, we have proposed PL-GNN, a graph learning-based framework that performs placement guidance for commercial tools. We demonstrate the proposed framework significantly improves the default placement flow in *Synopsys ICC2* on commercial CPU designs. We believe this work demonstrates promising directions of leveraging ML algorithms to solve crucial EDA problems. In the future, we plan to study the impact of leveraging different node features, and apply our framework in other advanced technologies.

## 8 ACKNOWLEDGMENT

This work is supported by the National Science Foundation under Grant No. CNS 16-24731 and the industry members of the Center for Advanced Electronics in Machine Learning.

## 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, (2):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, 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] L. Hagen and A. B. Kahng. A new approach to effective circuit clustering. In ICCAD, volume 92, pages 422–427, 1992.
[9] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In Advances in Neural Information Processing Systems, 2017.
[10] D.-H. Huang and A. B. Kahng. When clusters meet partitions: new density-based methods for circuit decomposition. In Proceedings the European Design and Test Conference. ED&TC 1995, pages 60–64. IEEE, 1995.
[11] E. Ihler, D. Wagner, and F. Wagner. Modeling hypergraphs by graphs with the same mincut properties. Information Processing Letters, 45(4):171–175, 1993.
[12] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. VLSI design, 11(3):285–300, 2000.
[13] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. The Bell system technical journal, 49(2):291–307, 1970.
[14] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.
[15] 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.
[16] Y.-C. Lu, J. Lee, A. Agnesina, K. Samadi, and S. K. Lim. Gan-cts: A generative adversarial framework for clock tree prediction and optimization. In 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2019.
[17] 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.
[18] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. Physical review E, 69(2):026113, 2004.
[19] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of computational and applied mathematics, 20, 1987.
[20] I. Synopsys. Compiler user guide, 2019.
[21] R.-S. Tsay and E. Kuh. A unified approach to partitioning and placement (vlsi layout). IEEE Transactions on Circuits and Systems, 38(5):521–533, 1991.
[22] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng. Electronic design automation: synthesis, verification, and test. Morgan Kaufmann, 2009.
[23] J. B. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In Proceedings Fourth International Conference on High-Performance Computing, pages 66–71. IEEE, 1997.
[24] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? arXiv preprint arXiv:1810.00826, 2018.
[25] T.-C. Yu, S.-Y. Fang, H.-S. Chiu, K.-S. Hu, P. H.-Y. Tai, C. C.-F. Shen, and H. Sheng. Lookahead placement optimization with cell library-based pin accessibility prediction via active learning. In Proceedings of the 2020 International Symposium on Physical Design, pages 65–72, 2020.