Adaptive Layout Decomposition With Graph Embedding Neural Networks

Wei Li, Yuzhe Ma, *Member, IEEE*, Yibo Lin, *Member, IEEE*, and Bei Yu, *Member, IEEE*

***Abstract*— Multiple patterning layout decomposition (MPLD) has been widely investigated, but so far there is no decomposer that dominates others in terms of both result quality and efficiency. This observation motivates us to explore how to adaptively select the most suitable MPLD strategy for a given layout graph, which is nontrivial and still an open problem. In this article, we propose a layout decomposition framework based on graph convolutional networks to obtain the graph embeddings of the layout. The graph embeddings are used for graph library construction, decomposer selection, graph matching, stitch removal prediction, and graph coloring. In addition, we design a fast nonstitch layout decomposition algorithm that purely depends on the message passing graph neural network. The experimental results show that our graph embedding-based framework can achieve optimal decompositions in the widely used benchmark with a significant runtime drop even compared with fast but nonoptimal heuristics.**

***Index Terms*—Design methodology, layout decomposition, VLSI design.**

I. Introduction

THE SEMICONDUCTOR industry nowadays is greatly challenged by extreme scaling, which imposes severe issues on circuits manufacturing. Among various advanced lithography techniques, multiple patterning lithography (MPL) is one of the most practical solutions to enhance the manufacturability and has been widely adopted in industry [1].

The core problem of MPL is the layout decomposition, which assigns features on a layout to separate masks for printability improvement and is also called multiple patterning layout decomposition (MPLD). If two features located closer than minimum coloring distance are assigned to the same mask, a coloring conflict is introduced. Additionally, stitches can be inserted to assist conflict resolving, at a cost of potential yield loss though. Therefore, the objective of MPLD is to find a mask assignment for features such that the number of conflicts and stitches is minimized.

Due to the $\mathcal{NP}$-hardness of the general layout decomposition problem, a variety of decomposition approaches has been proposed to achieve high quality and efficiency. These approaches can be roughly categorized into three types: 1) mathematical programming; 2) graph-theoretical approaches; and 3) heuristic approaches. The mathematical programming approach formulates the problem into integer linear programming (ILP) [2]–[7], and its relaxations, such as semidefinite programming (SDP) [5], linear programming (LP) [8], and discrete relaxation method [9]. Besides mathematical programming, graph-theoretical approaches resolve the problem with graph theories, e.g., the maximal independent set (MIS) [10], shortest path [11], [12], and fixed-parameter tractable (FPT) [13] algorithms. Some heuristic approaches are also proposed in [5], [10], [14], and [15], which are generally efficient but may have low quality. A recent work formulated MPLD into an exact cover problem and achieved high quality and efficiency with algorithm *X* [15]. Another extremely fast solution is based on graph matching [14], in which a coloring solution library for small graphs is constructed, and then graphs are colored efficiently by graph matching.

Although many decomposition algorithms have been developed, there is no conclusion that one decomposer is always better than another. The ILP-based method ensures optimality but suffers from runtime overhead for large layouts. The exact-cover (EC)-based method demonstrates high efficiency for large layouts at a cost of marginal degradation on the solution quality. The graph matching-based method shows good performance in both efficiency and quality for small graphs. But the library size of this method cannot be too large and only nonstitch graphs are supported, which is not applicable to large layouts or layouts with stitches. This observation motivates that it is worth exploring how to adaptively select the most suitable MPLD strategies for a given layout, which is nontrivial and still an open problem so far.

With successful deep learning applications in various fields by learning from historical data, we can naturally cast the problem into a classification task and leverage learning-based approaches. We need to investigate as much information on the graphs as possible and let our framework learn to adaptively utilize proper decomposition algorithms. However, graphs usually vary in terms of scale, making them hard to digest for learning models. Therefore, we need to obtain graph embedding under a unified shape to represent the graph as shown in Fig. 1. Specifically, we use some techniques to generate the

Manuscript received 1 July 2021; revised 30 October 2021; accepted 16 December 2021. Date of publication 6 January 2022; date of current version 24 October 2022. This work was supported in part by the Cadence Design Systems; in part by NVIDIA; and in part by The Research Grants Council of Hong Kong, SAR under Project CUHK24209017. The preliminary version has been presented at the ACM/IEEE Design Automation Conference (DAC), 2020, doi: 10.1109/DAC18072.2020.9218706. This article was recommended by Associate Editor I. H.-R. Jiang. *(Corresponding author: Bei Yu.)*

Wei Li is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, SAR, and also with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA.

Yuzhe Ma is with the Microelectronics Thrust, Hong Kong University of Science and Technology (Guangzhou), Guangzhou 511453, China.

Yibo Lin is with the Center for Energy-Efficient Computing and Applications, Peking University, Beijing 100871, China.

Bei Yu is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: byu@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TCAD.2022.3140729

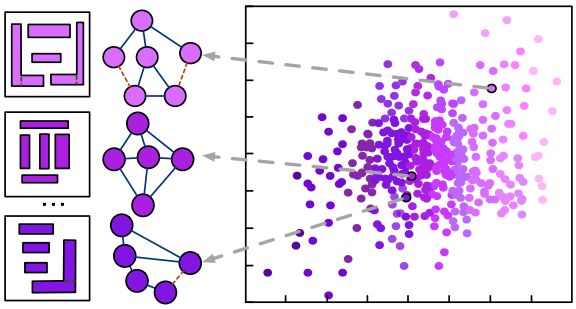


Fig. 1. Example of graph embeddings of layout graphs, where the graphs are transformed into vector space.

graph embedding such that the graph is transformed into a vector space in a lower but unified dimension with maximal representation capability and the powerful graph embedding helps us to adaptively select the best decomposer, where the best refers to the best solution quality at the lowest runtime.

Among different graph embedding methods, graph neural networks (GNNs) are widely used for irregular graph representations. In this article, we develop several GNN variations to obtain graph embeddings for different usages. First, we propose a nonstitch layout decomposer that purely depends on the graph embedding obtained by a specifically designed GNN. Second, the graph embeddings are used as representations to select the ILP-based decomposer (optimal but slow), EC-based decomposer (efficient but may not be optimal), or GNN-based decomposer (efficient and nearly optimal but does not support stitch). Besides decomposer selection, the graph embedding helps us to avoid isomorphic graphs during library construction. After that, it is used for matching graphs efficiently in the library and predicting whether the stitch edges in the layout graph are needed or not.

The main contributions are summarized as follows.

1) We point to the redundancy of stitch candidates in the layout graph, and develop a stitch redundancy prediction method based on graph embeddings.
2) We design a nonstitch layout decomposer that purely depends on GNN.
3) We design a graph library construction algorithm based on graph embeddings for small graphs excluding isomorphic ones.
4) We propose an adaptive workflow for efficient decomposer selection and graph matching using graph embeddings.
5) We conduct experiments on widely used benchmarks and experimental results demonstrate that our framework can reduce the runtime by 97.5% while still preserving the optimality compared with optimal but slow ILP-based decomposer.

The remainder of this article is organized as follows. Section II lists basic terminologies related to this work and gives the problem formulation. Section III introduces existing state-of-the-art decomposers and proposes a pure GNN-based decomposer, which is specifically designed for the nonstitch layout graphs. Section IV shows details of the GNN-based framework, including graph library construction and GNN model construction. Section V covers experimental results and finally, Section VI concludes this article.

II. Preliminaries

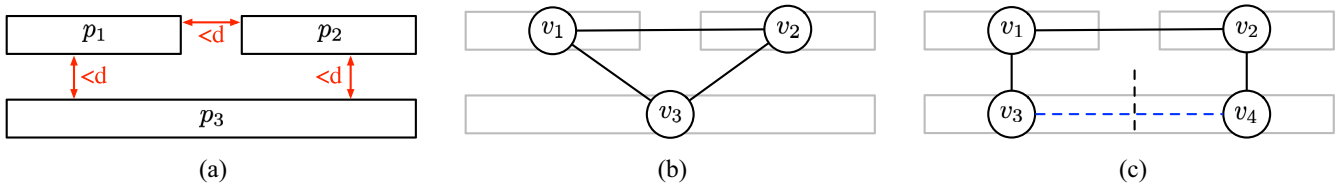
A. Multiple Patterning Lithography Decomposition (MPLD)

Given a routed layout represented by a set of polygonal features, $P = \{, \ldots, p_i, \ldots\}$, the minimal conflict space d, the number of masks k, and other constraints like precoloring constraints, the task of the MPLD problem is to assign masks to features or subfeatures divided by stitches so that the number of conflicts and stitches is minimized.

1) Conflict and Stitch: A *conflict* happens when two features whose relative distance is less than d are assigned the same mask. For example, we say one conflict happens when p_1 and p_3 in Fig. 2(a) are assigned the same mask. Sometimes, the conflict can be resolved by dividing the feature using two masks. Such a division is called a *stitch*. The polygonal feature p is split by stitch(es) into subfeatures, i.e., $p = \{, \ldots, r_i, \ldots\}$. To find effective stitches, many works [5], [16] generate a series of *stitch candidates* in the features before decomposition. These stitch candidates indicate possible locations of stitches to prevent the occurrence of conflicts. The p_1 and p_3 conflicts mentioned above can be resolved by inserting a stitch candidate [marked by a black-dotted line in Fig. 2(c)]. Previous works have shown that current stitch candidates are able to cover all possible stitches [3], [5].

2) Graph Format: The MPLD problem can be modeled as a variation of a pure graph-based problem, since the input layout can be translated into an undirected graph $\mathcal{G} = (V, E)$ without any information loss. When we consider the stitch candidates, i.e., predefine possible stitch locations, $\mathcal{G}$ is a *heterogeneous* graph where the node v_i corresponds to the subfeature r_i, and the edge set E is composed of two subsets: 1) the conflict edge set CE and 2) the stitch edge set SE. In the heterogeneous layout graph, if one feature is split into multiple subfeatures by stitch candidate(s), it will be translated into multiple nodes in the graph. To be more specific, one node is either: 1) one polygonal feature if there is no stitch candidate in the feature or 2) one subfeature split by the stitch candidate(s) in the polygonal feature. That is, $v_i \rightarrow r_i \in p$ *or* $v \rightarrow p$ if p does not contain the stitch candidate. The edge set $E = \{CE, SE\}$ models the relations between nodes. Two nodes are connected by the conflict edge $e \in CE$ if their relative distance is less than d and they do not belong to the same feature. Two nodes are connected by the stitch edge $e \in SE$ if they belong to the same feature and are split by the stitch candidate [like node v_3 and v_4 in Fig. 2(c)].

Otherwise, when no stitch is introduced, the layout graph is a simple *homogeneous* graph, where the node corresponds to one polygonal feature, i.e., $v_i \rightarrow p_i$, and the edge only represents the conflict relation. We refer to such a homogeneous graph as $\mathcal{G}_p$ (parent graph). Clearly, any heterogeneous layout graph $\mathcal{G}$ can be transformed to one homogeneous graph $\mathcal{G}_p$ by merging nodes connected by stitch edges. One example of the two representations is shown in Fig. 2. When we merge the nodes v_3 and v_4 in Fig. 2(c), the graph becomes a homogeneous one as shown in Fig. 2(b).



(a) (b) (c)

Fig. 2. Example of the routed layout and its graph representations. (a) Input routed layout. (b) Homogeneous graph representation, where the black line represents the conflict relation. (c) Heterogeneous graph representation considering stitches, where the stitch candidate is marked by the black-dotted line, and the stitch edge is highlighted in blue. Here, the relationship between p and v is: $p_1 = \{v_1\}, p_2 = \{v_2\}$, and $p_3 = \{v_3, v_4\}$.

3) Objective Function: From the perspective of a graph coloring problem, the objective is to assign colors to each node so that the weighted sum of conflict cost and stitch cost is minimized. Let $f : v \to \{1, \ldots, k\}$ be the coloring (decomposition) function and $f(v)$ be the color assigned to v by f. Given two features $p_m = \{, \ldots, r_i, \ldots\}$ and $p_n = \{, \ldots, r_j, \ldots\}$, the conflict cost adds to one if at least two nodes in p_m and p_n: 1) are connected by the conflict edge and 2) are assigned the same color, i.e., $\exists r_i \in p_m, r_j \in p_n : \{v_i, v_j \in CE, f(v_i) = f(v_j)\}$. The stitch cost adds to one if the two nodes connected by one stitch edge are assigned different colors, i.e., $f(v_i) \neq f(v_j) : v_i, v_j \in SE$. Formally, the objective can be formulated in

$$\min_{f} \sum_{p_m, p_n \in P; m \neq n} C_{mn} + \alpha \sum_{\{v_i, v_j\} \in SE} s_{ij}, \tag{1a}$$

$$\text{s.t. } C_{mn} = \min\{\sum_{\substack{r_i \in p_m, \\ r_j \in p_n, \\ \{v_i, v_j\} \in \text{CE}}} c_{ij}, 1\} \tag{1b}$$

$$s_{ij} = \begin{cases} 1, \text{ if } f(v_i) \neq f(v_j) \\ 0, \text{ otherwise.} \end{cases} \tag{1c}$$

$$c_{ij} = \begin{cases} 1, \text{ if } f(v_i) = f(v_j) \\ 0, \text{ otherwise.} \end{cases} \tag{1d}$$

where α is a parameter indicating the relative importance between the conflict cost C and the stitch cost s, which is usually set as 0.1.

B. Graph Isomorphism and Graph Matching

The formal definition of graph isomorphism and graph matching is stated as follows [17]. Given two graphs $\mathcal{G}_1 = (V_1, E_1)$ and $\mathcal{G}_2 = (V_2, E_2)$ with $|V_1| = |V_2|$, where V_1, V_2 and E_1, E_2 are corresponding node sets and edge sets, respectively. The objective of graph matching is to find a node-to-node mapping $f : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$. This is called an isomorphism if such a mapping f exists, and $\mathcal{G}_1$ is said to be isomorphic to $\mathcal{G}_2$.

In the graph library construction, graph isomorphism is one of the most critical factors because there exists $n! - 1$ isomorphic graphs for any graph with size n. If not removing these isomorphic graphs, the library will be occupied by redundancy. Also, graph matching is inevitable when extracting the coloring results of the matched node stored in the graph library.

C. Graph Neural Networks

GNN takes the graph as input and returns the node embeddings or graph embedding. Nowadays, most widely used GNNs adopt an iterative manner composed of two steps: 1) aggregation and 2) combination, which exploit the neighborhood information and ego-information, respectively. Specifically, for each node v in graph $\mathcal{G}$, the aggregation step aggregates neighbor u's features $\boldsymbol{h}_u$ and obtains an intermediate representation $\hat{\boldsymbol{h}}_v$ such that the final graph embedding is able to contain graph structure information. During the combination, GNN combines the aggregated representation $\hat{\boldsymbol{h}}_v$ with the ego-feature $\boldsymbol{h}_v$, and the result feature becomes the input of the next layer. GNN can be also explained in a message-passing way where the intermediate representations can be viewed as messages. The aggregation is the actual message-passing phase and each node passes its message to its neighbors along the edge. The combination is served as the integration phase, in which each node integrates the received message and reduces it into its new message. Each message pass and integration phase formulate one GNN layer. A general GNN layer can be described as follows:

$$\boldsymbol{h}_v^{(i)} = \text{COM}^{(i)}\left(\boldsymbol{h}_v^{(i-1)}, \text{AGG}^{(i)}\left(\{\boldsymbol{h}_u^{(i-1)} : u \in \mathcal{N}(v)\}\right)\right) \tag{2}$$

where $\boldsymbol{h}_v^{(i)}$ is the feature of node v after the ith layer, COM is the combination function, and $\mathcal{N}(v)$ is the neighbors of node v. The feature after the final layer is called the node embedding of each node and the graph embedding by GNN is usually obtained by some node invariant operations on node embeddings such as summation or mean.

D. Problem Formulation

Given a set of layout graphs and two state-of-the-art decomposers, ILP-based decomposer and EC-based decomposer, our objective is to train several GNN variations to obtain the graph embeddings such that: 1) the embedding can help to directly color the homogeneous layout graph, i.e., a layout that does not contain any stitch candidate; 2) the embedding can be used to build a graph library for small graphs, recording the coloring solutions; 3) the embedding can predict whether the stitch candidates in a heterogeneous layout graph are all redundant; 4) any new graph can find the best decomposer using its embedding; and 5) any new small graph can find the coloring solution directly through graph matching with graphs in the library.

III. Layout Decomposition Algorithms

A. State-of-the-Art Decomposers

Over the past few years, lots of decomposers are developed to solve the MPLD problem. We compare all state-of-the-art

TABLE I
COMPARISON AMONG DIFFERENT DECOMPOSERS

Methods	Quality	Efficiency	Flexibility	Stitch
ILP	Optimal	Poor	Medium	Yes
SDP	Near optimal	Medium	Medium	Yes
EC	Near optimal	Fast	Strong	Yes
Graph Matching	Optimal	Fast	Poor	No
Our GNN decomposer	Near optimal	Very fast	Strong	No

decomposers in terms of four perspectives: 1) result quality; 2) efficiency; 3) flexibility in multithread, GPU-acceleration, larger layout, and more masks; and 4) whether the method supports the stitch insertion. A general comparison is shown in Table I. In the following paragraphs, we simply introduce these existing decomposers and discuss the performance from the four listed perspectives.

1) Integer Linear Programming: Given the objective function as shown in 1, the problem can be naturally solved by ILP [3], [5], where the node color $f(v_i)$ is represented by 1-bit 0-1 variable(s). The ILP model for triple patterning lithography decomposition (TPLD) is described in

$$\min \sum_{e_{ij}\in \mathrm{CE}, r_i\in p_m, r_j\in p_n} C_{mn} + \alpha \sum_{e_{ij}\in \mathrm{SE}} s_{ij} \tag{3a}$$

$$\text{s.t. } x_{i1} + x_{i2} \le 1 \tag{3b}$$

$$x_{i1} + x_{j1} \le 1 + C_{mn1} \quad \forall e_{ij} \in \mathrm{CE}, r_i \in p_m, r_j \in p_n \tag{3c}$$

$$(1 - x_{i1}) + (1 - x_{j1}) \le 1 + C_{mn1} \quad \forall e_{ij} \in \mathrm{CE}, r_i \in p_m, r_j \in p_n \tag{3d}$$

$$x_{i2} + x_{j2} \le 1 + C_{mn2} \quad \forall e_{ij} \in \mathrm{CE}, r_i \in p_m, r_j \in p_n \tag{3e}$$

$$(1 - x_{i2}) + (1 - x_{j2}) \le 1 + C_{mn2} \quad \forall e_{ij} \in \mathrm{CE}, r_i \in p_m, r_j \in p_n \tag{3f}$$

$$C_{mn1} + C_{mn2} \le 1 + C_{mn} \quad \forall e_{ij} \in \mathrm{CE}, r_i \in p_m, r_j \in p_n. \tag{3g}$$

In (3), C_{mn}, C_{mni}, s_{ij}, and x_{ij} are integer variables in $\{0, 1\}$. x_{ij} represents the jth bit to encode the color of v_i. If the mask number k goes beyond 4, the maximum of j should also be larger than 2. C_{mn} represents whether there exists a conflict between p_m and p_n. C_{mn} is 1 when both C_{mn1} and C_{mn2} are 1 by the constraint formulated in (3g). $C_{mn1}(C_{mn2})$ represents whether there exists $r_i \in p_m, r_j \in p_n$, s.t., $x_{i1}(x_{i2}) = x_{j1}(x_{j2})$ and is controlled by (3c), (3d) and [(3e), (3f)].

The ILP-based method gives the optimal solution and supports the stitch scheme. However, the poor efficiency impedes its deployment in a large layout, which becomes more and more important with the development of the semiconductor industry.

2) Semidefinite Programming: Solving (3) using ILP is $\mathcal{NP}$-hard. As an alternative solver, SDP can approximately solve (3) in linear time. The basic idea is to program the colors by vectors so that the inner product between two vectors gives different values based on whether the two vectors (colors) are the same or not. For example, in the TPLD problem [5], [18], [19], three colors are assigned to three two-dimensional vectors, $(1, 0)$, $(-1/2, \sqrt{3}/2)$, and $(-1/2, -\sqrt{3}/2)$, respectively. Then, given any two vectors $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$, which represent the colors of node i and node j, we have the following properties:

$$\boldsymbol{v}_i \cdot \boldsymbol{v}_j = \begin{cases} 1, & f(v_i) = f(v_j) \\ -1/2, & f(v_i) \neq f(v_j). \end{cases} \tag{4}$$

Therefore, the MPLD problem can be solved by semidefinite programming in polynomial time if we relax the discrete values of $\boldsymbol{v}$ to a continuous one. Given the solutions of SDP, a fast heuristic mapping process is used to map the continuous solutions to coloring results.

The SDP-based method makes a good balance on the efficiency and performance and can be applied to the stitch case by simply adjusting the cost function. Nevertheless, the vector programming process for the node color in [5] is specifically designed for the TPLD problem, when extended to the four masks (quadruple patterning problem) or even more masks, the dimension of the vector will also increase, which harms the efficiency.

3) EC-Based Method: The EC-based method [16] transforms the MPLD problem to an exact cover problem, and solves it by a customized and augmented combination of dancing links data structure and Algorithm X* (DLX). Here, the routed layout is translated into a 0-1 matrix. In the matrix, each row index represents one possible coloring solution of a single feature p. If there is no stitch candidate in p, there will be k rows, representing k different color assignments of p. Otherwise, there will be more than k rows to represent different color combinations of subfeatures split by stitch candidates. The column index models the conflict relation to assure that two nodes connected by the conflict edge are not assigned the same color. Finally, the EC-based method returns a set of rows, which can be translated back to the decomposition result of the MPLD problem.

The EC-based method demonstrates excellent efficiency, and also, it is applicable for the stitch scheme and multithread. Moreover, it shows a relatively fast execution for very large cases. However, for some cases, it cannot be optimal, and the result quality may vary largely depending on the structure of the layout graph and the node ordering in the matrix.

4) Graph Matching-Based Method: The basic idea of graph matching is to build a graph library that contains graphs and corresponding solutions. Each time when we try to decompose the layout, the system will match the target layout graph to graphs in the library. If a match is found, the corresponding decomposition solution will be returned. Generally, since the library can be constructed offline, i.e., before any decomposition, the decomposition runtime is only influenced by the efficiency of the matching algorithm and the size of the library. However, the graph library size explodes when increasing the graph size or considering the stitch. In [14], the graph library is not applicable to stitch and only contains graphs with a size less than seven. Therefore, the flexibility is poor compared with other decomposers.

B. Nonstitch Layout Decomposer by GNNs

Given the overwhelming success of GNNs, we may attempt to solve the MPLD problem by GNNs directly. However, as

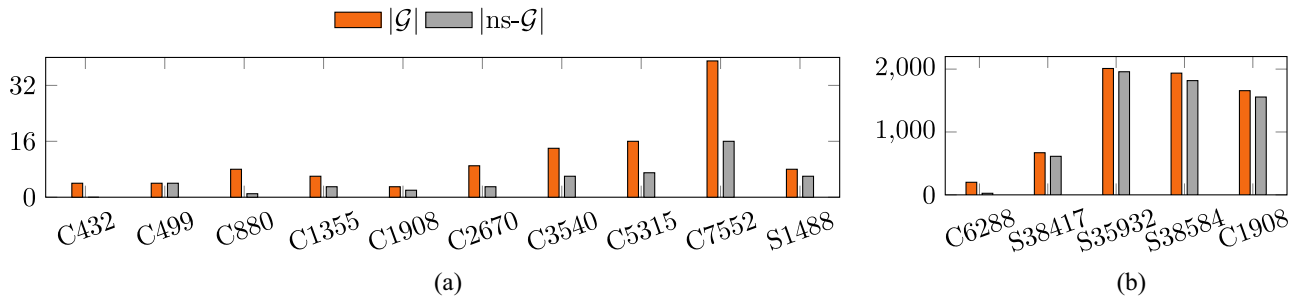


Fig. 3. Histogram of the number of graphs ($|\mathcal{G}|$, orange) and graphs that need not stitches ($|\text{ns-}\mathcal{G}|$, gray) in (a) small layouts and (b) large layouts.

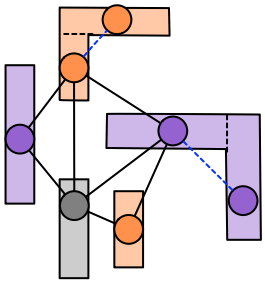


Fig. 4. Example of redundant stitch candidates. In this layout graph, both stitch candidates (highlighted in blue) finally do not generate any stitch.

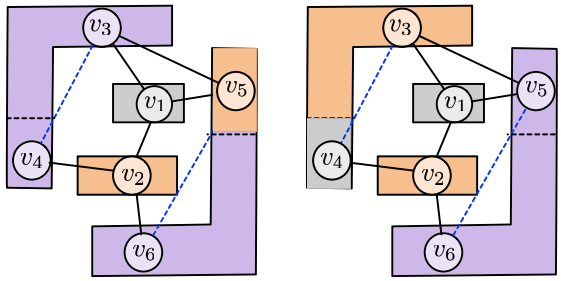


Fig. 5. Example of alterative solutions for stitches. The activated stitch is $\{v_5, v_6\}$ in (a) and $\{v_3, v_4\}$ in (b), respectively.

a heterogeneous variation of the coloring problem, the MPLD problem contains both stitch edge and conflict edge, making it more difficult than a pure coloring problem. On the other hand, there exist lots of redundant stitch edges (stitch candidates), which do not play a role in the final decomposition results. The histogram shown in Fig. 3 empirically states the phenomenon: over 80% layout graphs do not have stitches in the final results while most of them contain stitch edges. Although it is not easy to decompose the layout graph by GNNs directly, the stitch redundancy provides a vision of GNNs applying to the prediction of redundant stitch edges and then conducting a nonstitch layout decomposition. In the following section, we first introduce how graph embedding is used for the stitch redundancy prediction, and then propose a pure GNN-based method for the decomposition of nonstitch layouts.

1) Stitch Redundancy Prediction: Despite the fact that the state-of-the-art stitch candidate generation algorithm is able to enumerate all stitches, there are a huge number of stitch candidates that are not stitches in the final result, i.e., the two nodes split by the stitch candidates are assigned the same color in any optimal solutions. One example of the redundancy is given in Fig. 4, where each stitch candidate splits the corresponding feature into two subfeatures and generates two nodes connected by the stitch edge. Since both nodes are assigned the same color in the optimal solution, these redundant stitch candidates can be removed without any influence on the coloring quality.

On the other hand, useless candidates will increase the problem complexity largely and result in significant drops in efficiency performance. The layout statistics in Fig. 3 demonstrates that there exists a large portion of layout graphs that totally need no stitches. To avoid the waste of computation resources and further improve the efficiency of our decomposition framework, we propose a graph embedding-based method to remove these redundant and useless stitch candidates. The basic idea is that we can predict whether these stitch candidates are redundant or not. If redundant, these stitch candidates can be eliminated by merging those nodes split by stitch edges. Since the graph embedding can be obtained in parallel with other embeddings and the merge operation is processed in a constant time, the additional time cost can be ignored in light of the huge benefit from removing redundant stitches.

Although successful predictions bring about efficiency improvement, it is not easy to accurately predict which stitch candidate can be removed. The optimal solution is usually not unique: one stitch candidate can be redundant in one optimal solution while not in another one. The stitch candidate $\{v_3, v_4\}$ shown in Fig. 5 is a representative example. In Fig. 5(a), edge $\{v_3, v_4\}$ is redundant since node v_3 and v_4 are assigned the same color. On the contrary, the two nodes v_3 and v_4 have different colors in another optimal solution shown in Fig. 5(b), indicating that the stitch edge $\{v_3, v_4\}$ cannot be removed.

Considering the nonuniqueness of stitches as illustrated in Fig. 5, we regard the problem as a graph classification problem rather than an edge classification problem. That is, the algorithm predicts *whether the stitch candidates in a graph are all redundant (graph level)* instead of *whether stitch edge(s) in the graph are redundant (edge level)*. Therefore, the redundancy prediction can be implemented as a 2-class classifier on graph level and simply modeled by a multilayer perceptron (MLP) that uses the graph embedding as input. A detailed illustration can be found in Algorithm 1 (lines 1–7). After obtaining the corresponding graph embedding $\boldsymbol{h}_r$ (line 1), $\boldsymbol{h}_r$ is fed into the

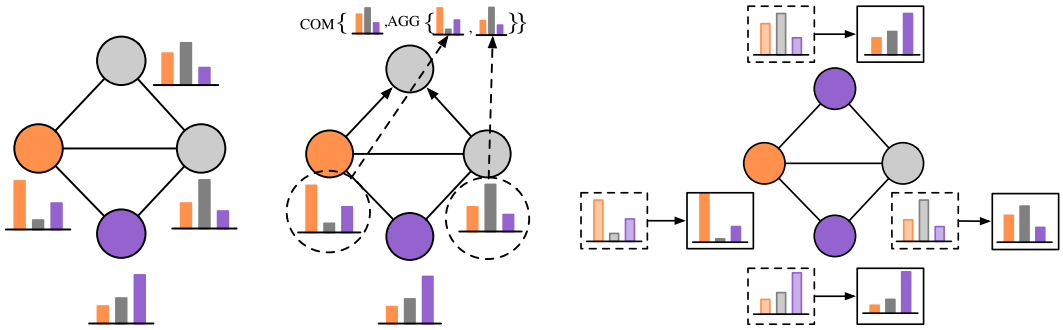


Fig. 6. Toy example on how *ColorGNN* gives the coloring results directly. (a) Randomly initialized color distribution. (b) Message passing procedure finished by trainable *ColorGNN*. (c) Final results (color distribution) of *ColorGNN*.

Algorithm 1 Pure GNN-Based Layout Decomposer

Require: $RGCN_r \leftarrow$ RGCN trained for generating graph embedding, where the embedding is used to predict the stitch redundancy;
Require: $ColorGNN \leftarrow$ GNN trained for graph coloring;
Require: $\mathcal{G} \leftarrow$ Target graph;
Require: $iter \leftarrow$ Number of repetitive executions;
Ensure: $\boldsymbol{x} \rightarrow$ The coloring results for each node in $\mathcal{G}$;
1: $\boldsymbol{h}_r \leftarrow RGCN_r(\mathcal{G})$;
2: confidence $\leftarrow MLP(\boldsymbol{h}_r)$;
3: **if** confidence $\leq b$ **then**
4: Decompose $\mathcal{G}$ by other decomposer;
5: **return** the decomposition solution;
6: **else**
7: $\mathcal{G}_p \leftarrow$ Remove all stitch edges in $\mathcal{G}$ and merge related nodes;
8: **end if**
9: **for** $i \in \{1, \ldots, iter\}$ **do**
10: $\boldsymbol{x} \leftarrow$ Randomly initialized probability distribution of colors for each node;
11: $\boldsymbol{c_i} \leftarrow ColorGNN(\mathcal{G}_p, \boldsymbol{x})$;
12: **end for**
13: **return** the best solution in $\{\boldsymbol{c_1}, \ldots, \boldsymbol{c_{iter}}\}$;

implemented MLP and predicts a confidence value (line 2). After prediction, if the confidence of the graph is larger than a specific bar, say b (lines 6 and 7), the graph will merge all stitch candidates in the graph, which results in a nonstitch graph $\mathcal{G}_p$.

2) Nonstitch GNN Decomposer: Although our redundancy prediction is only applicable for stitch-enabled cases, which may not be useful in some foundries, the following nonstitch GNN decomposer can directly help those foundries. We refer to our nonstitch GNN decomposer as *ColorGNN*, which uses the message passing GNN as a backbone, and gives a prior that the node embedding represents the probability (belief) of color assignments. The detail is described as follows. Given a nonstitch graph $\mathcal{G}_p = \{V, E\}$, where $E = \{CE\}$, we first randomly assign each node $v \in V$ a discriminative attribute $\boldsymbol{x}_v \in \mathbb{R}^k$ that represents the probability distribution of k masks.

In the aggregation step, we simply sum up the features from all neighbors. However, for each aggregation iteration, we randomly sample neighbors to do the summation. Such a random sampling scheme improves the efficiency and the inserted randomness helps to avoid local optimum in the coloring problem [20]. Formally, let $\boldsymbol{c}_v^{(i)} \in \mathbb{R}^k$ be the result returned by $\text{AGG}^{(i)}$ for the node v in the ith layer, and the aggregation layer can be represented by: $\boldsymbol{m}_v^{(i)} = \sum_{u \in \mathcal{N}'(v)} \boldsymbol{c}_u^{(i-1)}$, where $\mathcal{N}'(v)$ is defined as the subset of $\mathcal{N}(v)$ and is selected randomly.

In the combination function, we define $\text{COM}^{(i)}$ as a simple trainable weighted summation between ego feature and features from neighbors

$$\boldsymbol{c}_v^{(i)} = \boldsymbol{c}_v^{(i-1)} \lambda_C^{(i)} + \boldsymbol{m}_v^{(i)} \lambda_A^{(i)}. \tag{5}$$

Here, both $\lambda_C^{(i)}$ and $\lambda_A^{(i)}$ are trainable variables. Finally, the color of each node is assigned based on $\boldsymbol{c}_i$ that represents the color belief. A figure illustration of the whole process is shown in Fig. 6. In our implementation, we iteratively execute the GNN multiple times (five times in our experiments) by setting different initializations (lines 9–13 in Algorithm 1). Finally, we select the best solution among all iterations.

IV. Adaptive Decomposition Framework

In this section, we first briefly present the workflow of our proposed framework. Then, we describe the GNN used for graph embedding, and how the graph embedding is used for graph library construction, graph matching, and decomposer selection.

A. Overview

Combining with the pure GNN-based decomposer introduced in Section III, we propose a decomposition framework that selects the decomposition method adaptively and supports the GNN-based decomposer. The framework is divided into offline and online parts by whether the operation is needed in the decomposition.

The offline part is like a preprocessing step, and done before any decomposition. Generally, it includes GNN model training and graph library construction. We first train all GNN models required in the decomposition, including two relational graph convolutional networks (RGCNs) [22] and one proposed GNN decomposer for nonstitch decomposition. A summary of all GNNs used in our framework is shown in Table II. Overall, the first RGCN model RGCN is trained to select the better decomposer in EC and ILP. The generated graph embedding

TABLE II
GNNs Used in Our Framework

Steps	Models	Usage
Graph library construction	$\boldsymbol{h} = RGCN(\mathcal{G})$	Avoid isomorphism to reduce library size
Graph matching		Find the matched graph in the graph library
Algorithm selection		Select the better decomposition algorithm in EC/ILP
Stitch redundancy prediction	$\boldsymbol{h}_r = RGCN_r(\mathcal{G})$	Predict whether the stich edges in $\mathcal{G}$ are all redundant
GNN decomposer	$\boldsymbol{c} = ColorGNN(\mathcal{G}_p)$	Decompose non-stitch graph $\mathcal{G}_p$

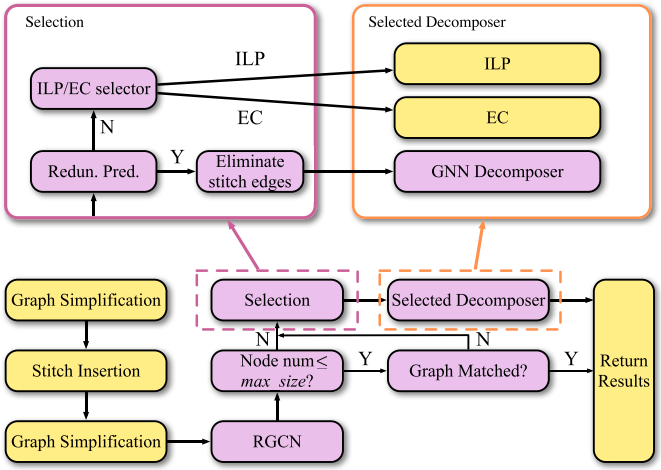


Fig. 7. Workflow of our framework. Purple blocks are executed in our framework while the yellow blocks are directly executed in OpenMPL [21].

$\boldsymbol{h}$ is also used to build the graph library and achieve efficient graph matching. The second RGCN model RGCN$_r$ is to predict whether the stitch edges in $\mathcal{G}$ are all redundant. RGCN and RGCN$_r$ have the same model architecture, but they are trained for different tasks so that their graph embeddings are expected to be different. Besides the training of all models, we use graph embeddings obtained by the trained RGCN to build the isomorphism-free graph library.

When the above offline part is done, we can execute layout decomposition following the workflow shown in Fig. 7. The layout graph transformed from the original layout is simplified by several simplification techniques, such as independent component computation (ICC) [5], hide small degrees [5], [14], and biconnected component analysis [3], [4]. Next, stitch candidates are inserted by pattern projection [5]. Stitch candidate insertion transforms the simplified homogeneous graphs into heterogeneous graphs, which contain both conflict and stitch edges. After another series of simplification steps for the heterogeneous graphs, these simplified heterogeneous graphs are fed into RGCN and RGCN$_r$ to obtain the graph embeddings $\boldsymbol{h}$ and $\boldsymbol{h}_r$. For a graph whose graph size is under the size constraint *max_size*, $\boldsymbol{h}$ is used to determine whether there is an isomorphism between the target graph and graphs in the library. If the isomorphic graph is found in the library, the corresponding node embeddings of two graphs are used to get the node-to-node mapping and directly return the final coloring result by the mapping in the library. If no isomorphic graph is found or the graph size is larger than *max_size*, the generated graph embeddings $\boldsymbol{h}$ and $\boldsymbol{h}_r$ help to select the best decomposers. During the selection, $\boldsymbol{h}_r$ is first used to predict whether the stitch edges are redundant. If predicted as yes, these stitch edges are eliminated by merging all stitch edges so that the heterogeneous graph $\mathcal{G}$ is simplified as a nonstitch homogeneous graph $\mathcal{G}_p$ (parent graph). Then, our proposed GNN decomposer *ColorGNN* is adopted to decompose $\mathcal{G}_p$. If predicted as no, an ILP/EC selector based on $\boldsymbol{h}$ is going to select which one is the better algorithm for the target graph. Here, a better algorithm means the one achieving lower cost or higher efficiency when the cost is the same. After all graphs are decomposed, a color recovery process is executed to get the final layout decomposition results.

B. Graph Embedding Neural Network

Considering that the simplified graph contains both conflict edges and stitch edges, we apply RGCN [22], a GCN variation specifically for heterogeneous graphs, to obtain the graph embedding. The process for graph embedding is shown in Fig. 8. The original layout is transformed into multiple heterogeneous graphs by graph simplification and stitch insertion. Those simplified graphs are fed into the two-layer model. For each node v_i in a graph $\mathcal{G} = \{V, E\}, E = \{CE, SE\}$, the node representation $\boldsymbol{h}_i^{(l+1)} \in \mathbb{R}^{D^{(l+1)}}$ at the $(l+1)$th layer of the neural network can be calculated by the following formula:

$$\boldsymbol{h}_i^{(l+1)} = \text{ReLU}\left(\sum_{e\in E}\sum_{j\in N_i^e} \boldsymbol{W}_e^{(l)}\boldsymbol{h}_j^{(l)} + \boldsymbol{h}_i^{(l)}\right) \tag{6}$$

where $D^{(l)}$ is the dimension of node representation at the lth layer, $\boldsymbol{W}_e^{(l)} \in \mathbb{R}^{D^{(l+1)}\times D^{(l)}}$ is a learnable weight matrix of edge type $e \in E$, and N_i^e denotes the set of neighbor nodes of node v_i connected by e. Intuitively, RGCN specified in (6) works like the classical GCN, as both neural network layers contain two phases: 1) aggregation and 2) combination. The difference is that edges in GCN share the same learnable weight in each layer on the combination phase while only edges under the same edge type share the weight matrix for RGCN, which means that the message integration for different kinds of edges is independent. One central issue of the different weight matrixes strategy is the exploding number of parameters. Also, this strategy can easily lead to overfitting. These issues are solved by regularization of weight and we adopt a basis decomposition [22], in which each weight matrix $\boldsymbol{W}_e^{(l)}$ is a linear combination of basis transformations $\boldsymbol{V}^{(l)}$ and defined by

$$\boldsymbol{W}_e^{(l)} = \sum_{b=1}^{B} \delta_{rb}^{(l)} \boldsymbol{V}_b^{(l)} \tag{7}$$

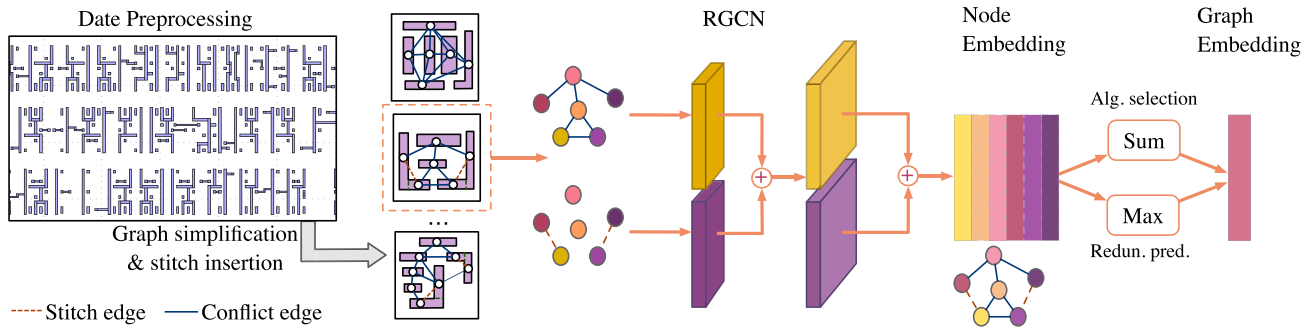


Fig. 8. Overview of the process for graph embedding.

where $\boldsymbol{V}_b^{(l)} \in \mathbb{R}^{D^{(l+1)} \times D^{(l)}}$ is one of the multiple basis transformations and $\delta_{rb}^{(l)}$ is the learnable coefficient.

The input feature of node v_i is defined as

$$\boldsymbol{h}_i^{(0)} = \sum_{j \in N_i} I_{\{e_{i,j} \in CE\}} + \alpha I_{\{e_{i,j} \in SE\}} \tag{8}$$

where $I_{\{\cdot\}}$ is an indicator function and $\alpha = -0.1$ is a user-defined parameter following the general stitch cost. After obtaining the node embeddings by the RGCN model, for the algorithm selection, we calculate the graph embedding by the summation of the node embeddings. A summation is used because the graph size may influence results of the two subtasks. Formally, we have $\boldsymbol{h} = \sum_{i \in V} \boldsymbol{h}_i^{(\text{out})}$, where $\boldsymbol{h}_i^{(\text{out})}$ is the node embedding of node v_i. As for the stitch redundancy prediction, we use max-pooling because it is some subgraph structures that determine whether there exist redundant stitch edges or not.

C. Graph Library Construction

Generally speaking, it is possible to enumerate all the valid graphs under a size constraint such that we can build up a graph library to accelerate decomposition by simply matching the graph with graphs in the library and collecting the coloring information stored in the library.

Previous work [14] follows the algorithm described in [23] and [24], and constructs a graph library that contains all homogeneous graphs (23 in total) with node numbers less than seven. However, the graph in that library does not contain stitch edge, which means that one heuristic stitch insertion and coloring method should be used if the nonstitch graph is not colorable. To store graphs containing stitch edges, we propose an isomorphism-free heterogeneous graph library construction algorithm that contains all possible graphs with both stitch edges and conflict edges.

Different from the general 2-connected graph described in [23], the graph transformed by circuit layout has some specific rules, especially after stitch insertion. The rules are stated as follows.

- The degree of each node in $\mathcal{G}_p$ is at least the mask number k.
- The degree of each node in $\mathcal{G}$ is at least two.
- One node pair $\{u, v\}$ cannot be connected if u and v are in the stitch relation. The stitch relation of two nodes means they belong to the same feature in the original layout.
- For any two nodes in the stitch relation, their neighbors connected by the conflict edges cannot be the same.

Algorithm 2 Graph Library Construction

Require: *max_size* → Maximal graph size.
Ensure: L → The isomorphism-free library of valid graphs;
1: $L \leftarrow \{\}$;
2: $S_p \leftarrow$ Generate graphs following method in [23];
3: $S_p \leftarrow$ Remove invalid Graphs in S_p;
4: $S \leftarrow$ Enumerate graphs containing stitches from graphs in S_p;
5: **for** $\mathcal{G} \in S$ **do**
6: **if** $\mathcal{G}$ satisfies layout graph rules **then**
7: $\boldsymbol{h} \leftarrow$ normalize(RGCN(G));
8: $\boldsymbol{\mathcal{L}}_h \leftarrow$ Extract graph embeddings stored in the library;
9: **if** $\max(\boldsymbol{\mathcal{L}}_h \times \boldsymbol{h}) < 1$ **then**
10: Decompose $\mathcal{G}$ with ILP-based decomposer;
11: Insert $\mathcal{G}$ and corresponding graph embedding, node embeddings, and decomposition result into L;
12: **end if**
13: **end if**
14: **end for**

The pseudocode of our library construction algorithm is illustrated in Algorithm 2. First, we enumerate $\mathcal{G}_p$ by the method in [23] (line 2), which generates an isomorphism-free 2-connected graph set and removes all invalid graphs (line 3). Then, for each $\mathcal{G}_p$, we enumerate valid $\mathcal{G}$, which satisfies the size constraint and all the rules above by splitting nodes in $\mathcal{G}_p$ and inserting stitch edges (lines 4–6). Because there may be multiple isomorphic graphs during the enumeration of $\mathcal{G}$, we use graph embedding to avoid isomorphism. Specifically, every time when the enumerated $\mathcal{G}$ is going to put into the library, $\mathcal{G}$ will be fed into the RGCN model (line 7) and the obtained normalized graph embedding $\boldsymbol{h}$ is compared with the embeddings stored in the library $\boldsymbol{\mathcal{L}}_h \in \mathbb{R}^{k \times D}$ (line 8), and a vector–matrix multiplication is performed, i.e., $\boldsymbol{m} \in \mathbb{R}^k = \boldsymbol{\mathcal{L}}_h \times \boldsymbol{h}$, where k is the number of graphs stored in the library temporarily. Whether there is an isomorphic graph in the library or not is determined by checking the maximum element in $\boldsymbol{m}$ (line 9) because two unit vectors are equal if and only if their product is 1. The idea is based on the fact that a GCN-based model is insensitive to the node order, which means that the graph embeddings of all isomorphic graphs by a GCN-based model are totally the same. After isomorphism checking, $\mathcal{G}$ will not be inserted into the library if there is an isomorphic graph. Otherwise, $\mathcal{G}$ will be decomposed by an ILP-based decomposer for optimal solution (line 10), and then graph $\mathcal{G}$ with its

optimal coloring result, corresponding graph embedding, and node embeddings will be stored in the library (line 11).

D. Graph Matching and Decomposer Selection

1) Graph Matching: In the decomposition process, when the graph embedding is obtained and the graph size is under the limitation, we directly match the graph with graphs in the library to obtain the decomposition result. We use the obtained graph embedding to find isomorphic graphs in the library. Then, we use the corresponding node embeddings to find the node-to-node mapping and return the solution directly.

To illustrate the process clearly, we provide a simple example and explain the details step by step. The graph library $\boldsymbol{\mathcal{L}}$ in this example is composed of three graphs, in which each graph has four nodes and the dimension of graph embedding is two. The library stores all information of graphs needed by our framework, including its node embeddings $\boldsymbol{\mathcal{L}}_u \in \mathbb{R}^{3\times4\times2}$, graph embeddings $\boldsymbol{\mathcal{L}}_h \in \mathbb{R}^{3\times2}$, and optimal solutions $\boldsymbol{\mathcal{L}}_s \in \mathbb{R}^{4\times3}$.

$$\boldsymbol{\mathcal{L}}_u = \begin{bmatrix} 0.4 & 0.4 \\ 0.3 & -1.0 \\ 0.1 & 0.6 \\ -0.2 & 0.8 \end{bmatrix} \quad \boldsymbol{\mathcal{L}}_h = \begin{bmatrix} 0.6 & 0.8 \\ 0.6 & -0.8 \\ 1 & 0 \end{bmatrix} \quad \boldsymbol{\mathcal{L}}_s = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 2 \end{bmatrix}$$

Different colors represent different graphs in the library. Taking a target graph $\mathcal{G}$ with four nodes as an example, we use the RGCN model to obtain the corresponding node embedding $\boldsymbol{u} \in \mathbb{R}^{4\times2}$ and graph embedding $\boldsymbol{h} \in \mathbb{R}^2$, where $\boldsymbol{h} = \sum_i \boldsymbol{u}_i$

$$\boldsymbol{u} = \begin{bmatrix} 0.3 & -1.0 \\ -0.2 & 0.8 \\ 0.4 & 0.4 \\ 0.1 & 0.6 \end{bmatrix}, \boldsymbol{h} = \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix}. \tag{9}$$

We first multiply the graph embedding $\boldsymbol{h}$ with graph embeddings $\boldsymbol{\mathcal{L}}_h$ in the library, i.e., $\boldsymbol{m} \in \mathbb{R}^3 = \boldsymbol{\mathcal{L}}_h \times \boldsymbol{h}$.

$$\boldsymbol{m} = \underbrace{\begin{bmatrix} 0.6 & 0.8 \\ 0.6 & -0.8 \\ 1 & 0 \end{bmatrix}}_{\boldsymbol{\mathcal{L}}_h} \times \underbrace{\begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix}}_{\boldsymbol{h}} = \begin{bmatrix} 1 \\ -0.28 \\ 0.6 \end{bmatrix}$$

Then, the matched graph index i in the library is defined by

$$i = \begin{cases} \arg\max(\boldsymbol{m}), & \text{if } \max(\boldsymbol{m}) = 1 \\ -1, & \text{otherwise} \end{cases} \tag{10}$$

where -1 means there is no isomorphic graph matched in the library such that the graph matching process is terminated and redirected to decomposer selection. Otherwise, $\boldsymbol{\mathcal{L}}_u[i]$ is extracted and compared with the target graph's node embedding to get the final node-to-node mapping. The matching method is also based on the node order insensitivity of the GCN-based model: if the input feature does not contain any node order information such as a one-hot vector of the node order, the final graph embedding will be order-invariant. In this example, $\boldsymbol{m}[0] = 1$; therefore, $i = 0$, representing that the first node embedding $\boldsymbol{\mathcal{L}}_u[0]$ is used to compare with $\boldsymbol{u}$.

The node-to-node mapping f is executed by comparing two node embeddings and formulated by

$$f(j) = k, \text{ if } \boldsymbol{u}[j] = \boldsymbol{\mathcal{L}}_u[i][k] \text{ for } j, k \text{ in } \{0, \ldots, |G| - 1\} \tag{11}$$

where $|G|$ means the number of nodes in the graph. In this example, $|G|$ is 4 and f is then defined by: $f(\{0, 1, 2, 3\}) = \{1, 3, 0, 2\}$.

After f is found, the solution $\boldsymbol{s}$ can be matched quickly by

$$\boldsymbol{s}[j] = \boldsymbol{\mathcal{L}}_s[f(j)][i], \text{ for } j \text{ in } \{0, \ldots, |G| - 1\} \tag{12}$$

so the final solution of $\mathcal{G}$ in this example is mapped as $[2, 1, 1, 0]$.

2) MPL Decomposer Selection: When the size is larger than the size limitation or no mapping is found, the graph embedding $\boldsymbol{h}$ is used to select the decomposer (ILP/EC). Therefore, the decomposer selector can be regarded as a 2-class classifier and simply modeled by a summation of one trainable weight matrix $\boldsymbol{W}_s \in \mathbb{R}^{2\times D}$ and a bias vector $\boldsymbol{b}_s \in \mathbb{R}^2$ combined with the arg max function, which can be formulated as

$$y = \arg\max(\boldsymbol{W}_s\boldsymbol{h} + \boldsymbol{b}_s) \tag{13}$$

where $\boldsymbol{h} \in \mathbb{R}^D$ is the graph embedding obtained by the RGCN model with dimension D. The final decomposition result is then generated by the selected decomposer.

V. Experimental Results

A. Benchmark and Experimental Settings

The experiments are performed on the scaled-down and modified ISCAS benchmarks, which are widely used in previous works [5], [14], [15]. The framework is mainly implemented in Python with PyTorch [25] and DGL [26] and integrated into the open-source layout decomposition framework OpenMPL [21]. Fig. 7 specifies the detailed task execution platform of the workflow. We follow the same settings in [5], [14], and [15] on the minimum color space, where the first ten cases are set to 120 nm and the last five cases are set to 100 nm. The cost of stitch is set to 0.1 such that the decomposition cost is calculated by $cn\# + 0.1st\#$, mask number is set to 3, and the graph simplification level in OpenMPL is 3. It should be noted that our graph embedding, as well as the whole framework, is flexible to be extended to other decomposition tasks under different lithography constraints.

B. Model and Training Settings

Generally, we prepare and train three independent GNN models for: 1) graph matching and decomposer selection (RGCN); 2) stitch redundancy prediction (RGCN$_r$); and 3) nonstitch GNN decomposer (*ColorGNN*). The two RGCN models contain two layers whose output dimensions are 32 and 64, respectively, such that the dimension of graph embedding is 64. *ColorGNN* contains ten layers. The training strategy of RGCNs follows the idea of K-fold cross-validation; specifically, each time two of the 15 layouts in the benchmark are used as the test/validation set separately, and the other 13 layouts are put together to form a training set. Therefore, there are 15 trained models for 15 layouts following the same model configurations. In the algorithm selection, the label of each simplified graph is

TABLE III
F1 SCORE COMPARISON OF (a) PROPOSED RGCN AND (b) CONVENTIONAL GCN

(a)

		Label	
		ILP	EC
Predicted	ILP	13	682
	EC	0	5900
Recall		**100.0%**	
F1-score		**0.0367**	

(b)

		Label	
		ILP	EC
Predicted	ILP	2	244
	EC	11	6338
Recall		15.4%	
F1-score		0.0154	

set as 0 (ILP) if the cost by the ILP-based decomposer is smaller than the EC-based decomposer and 1 (EC) for other cases. In the stitch redundancy prediction, each layout graph is labeled as "not redundant" if there exists at least one stitch in the optimal solution obtained by the ILP method. In the training phase of RGCN(RGCN$_r$), we concatenate the graph embedding network with the MPL for decomposer selector (stitch redundancy prediction) such that the cross-entropy loss function can be adopted. The training of *ColorGNN* relies on an unsupervised margin loss, formulated by

$$\min \sum_{\{u,v\}\in CE} \max\{m - d(\boldsymbol{c}_u, \boldsymbol{c}_v), 0\} \tag{14}$$

where m is a predefined margin and set as 1. The loss function is motivated by the truth that two connected nodes should be assigned different features. In all GNN-related operations, the simplified graphs are batched together for efficient inference. All the experiments are conducted on an Intel Core 2.9-GHz Linux machine with one NVIDIA TITAN Xp GPU.

C. Effectiveness of Model Selection

In the first experiment, we compare the effectiveness of our proposed RGCN model with the conventional GCN model. The classical GCN model only supports homogeneous graphs while is not compatible with this task. Therefore, we slightly modify the message passing function by multiplying the edge weight α_e for different edge types

$$\boldsymbol{u}_i^{(l+1)} = \mathrm{ReLU}\left(\sum_{e\in E}\sum_{j\in N_i^e} \alpha_e \boldsymbol{W}^{(l)} \boldsymbol{u}_j^{(l)} + \boldsymbol{u}_i^{(l)}\right). \tag{15}$$

Here, α_e follows the weighted cost setting and is set as 1 for conflict edge and -0.1 for stitch edge. The result is illustrated by the confusion matrix shown in Table III, where each row contains the number of graphs selected to be decomposed by the corresponding decomposer while each column contains the number of graphs labeled by the corresponding decomposer. For example, the element (0, 0) in the confusion matrix represents the number of graphs that are labeled as positive (ILP) and also selected to be decomposed by an ILP-based decomposer. In the experiment, we use two more metrics: 1) recall and 2) F1 score. Recall is used to measure the proportion of ILP-labeled graphs that are correctly identified, and therefore, influences the decomposition quality directly. $F1$-score is a general metric for the model's accuracy. According to Table III, we can see that the $F1$-score of our model is more than $2\times$ of that in the conventional GCN, which demonstrates the powerful representation capability of our model compared with conventional GCN. Another important point is that our model classifies all the graphs labeled as positive correctly such that our recall achieves 100% while conventional GCN only classifies 15.4% correctly.

D. Comparison With Other State-of-the-Art Methods

In the second experiment, we compare our results with state-of-the-art decomposers under one thread. All the decomposers are implemented and measured in OpenMPL such that we can keep the preprocess procedure the same and compare the results without potential bias due to different simplification methods or stitch insertion techniques. Table IV lists the decomposition cost of all decomposers. Table V lists all the decomposition runtime excluding the time for graph simplification and stitch insertion for better comparison. As expected, there is no one existing decomposer that can dominate others among existing decomposers. The EC-based decomposer outperforms others on runtime while causing some additional costs. An ILP-based decomposer obtains the optimal results while the runtime is significantly worse than others. The SDP-based decomposer shows a runtime improvement compared with the ILP-based decomposer but cannot compete with the EC-based decomposer on both runtime and quality. Our framework [27] obtains the optimal results in all cases no matter whether we integrate the proposed nonstitch GNN decomposer. The average runtime is reduced to 12.3% compared with the ILP-based decomposer because of the efficient graph matching technique and EC-based decomposer, which is selected as the decomposer in most cases. Moreover, when we integrate the GNN decomposer into our framework, the runtime can be further reduced to 4.2% with the optimality still preserved. The main reasons for the large improvement are: 1) the existence of considerable graphs that need not stitches and 2) the efficiency of our proposed purely GNN decomposer under GPU acceleration.

E. Runtime and Algorithm Selection Analysis

In the third experiment, we analyze the runtime and the algorithm selection results in our framework. The decomposition runtime of our framework is mainly composed of five parts: 1) decomposition runtime by our GNN decomposer; 2) decomposition runtime by the ILP-based decomposer; 3) decomposition runtime by the EC-based decomposer; 4) algorithm selection time; and 5) the runtime for the stitch redundancy prediction. The runtime for graph matching and graph embedding is counted in the decomposer selection since the 2-class classifier is integrated into the graph embedding network for fast inference. Fig. 9 shows the result, where the metric is the total decomposition runtime of 15 layouts as before. From the figure, we can clearly see that the decomposition runtime by the selected decomposer (ILP and DL) is the major bottleneck and occupies 84.31% of the total runtime. The result indicates that the runtime of GNN-related operations in our framework is trivial, meaning that our method has strong scalability and can be applied to select other more efficient decomposers in the future. The inference runtime of

TABLE IV
DECOMPOSITION COST COMPARISON

Circuit	ILP			SDP			EC			[27]			[27] + GNN decomposer		
	st#	cn#	cost	st#	cn#	cost	st#	cn#	cost	st#	cn#	cost	st#	cn#	cost
C432	4	0	0.4	4	0	0.4	4	0	0.4	4	0	0.4	4	0	0.4
C499	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C880	7	0	0.7	7	0	0.7	7	0	0.7	7	0	0.7	7	0	0.7
C1355	3	0	0.3	3	0	0.3	3	0	0.3	3	0	0.3	3	0	0.3
C1908	1	0	0.1	1	0	0.1	1	0	0.1	1	0	0.1	1	0	0.1
C2670	6	0	0.6	6	0	0.6	6	0	0.6	6	0	0.6	6	0	0.6
C3540	8	1	1.8	8	1	1.8	8	1	1.8	8	1	1.8	8	1	1.8
C5315	9	0	0.9	9	0	0.9	9	0	0.9	9	0	0.9	9	0	0.9
C6288	205	1	21.5	203	4	24.3	203	5	25.3	205	1	21.5	205	1	21.5
C7552	21	1	3.1	21	1	3.1	21	1	3.1	21	1	3.1	21	1	3.1
S1488	2	0	0.2	2	0	0.2	2	0	0.2	2	0	0.2	2	0	0.2
S38417	54	19	24.4	48	25	29.8	54	19	24.4	54	19	24.4	54	19	24.4
S35932	40	44	48	24	60	62.4	46	44	48.6	40	44	48	40	44	48
S38584	117	36	47.7	108	46	56.8	116	37	48.6	117	36	47.7	117	36	47.7
S15850	97	34	43.7	85	46	54.5	100	34	44	97	34	43.7	97	34	43.7
average			12.893			15.727			13.267			12.893			12.893
ratio			**1.000**			1.220			1.029			**1.000**			**1.000**

TABLE V
DECOMPOSITION RUNTIME COMPARISON

Circuit	ILP	SDP	EC	[27]	[27] + GNN decomposer
C432	0.486	0.016	0.005	0.007	0.024
C499	0.063	0.018	0.011	0.015	0.023
C880	0.135	0.021	0.010	0.014	0.032
C1355	0.121	0.024	0.011	0.015	0.025
C1908	0.129	0.024	0.017	0.031	0.023
C2670	0.158	0.044	0.035	0.046	0.040
C3540	0.248	0.086	0.032	0.038	0.043
C5315	0.226	0.106	0.039	0.049	0.027
C6288	5.569	0.648	0.151	0.154	0.775
C7552	0.872	0.157	0.071	0.111	0.108
S1488	0.147	0.031	0.013	0.016	0.023
S38417	7.883	1.686	0.329	0.729	0.140
S35932	13.692	5.130	0.868	1.856	0.373
S38584	13.494	4.804	0.923	1.840	0.310
S15850	11.380	4.320	0.864	1.792	0.328
average	3.640	1.141	0.225	0.448	0.153
ratio	1.000	0.313	0.062	0.123	0.042

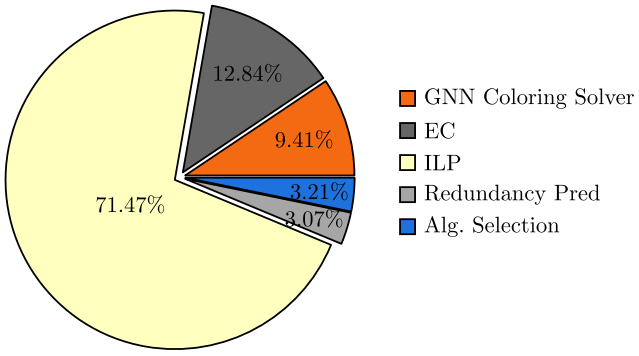


Fig. 9. Runtime breakdown of our framework.

algorithm selection and redundancy prediction is very close because both of them use RGCN as the backbone with the same parameters.

For each graph, our adaptive framework tries to select the most suitable decomposition algorithm. Here, we compare the ratio of graphs assigned to different decomposers. The result is shown in Fig. 10, more than 86.11% graphs are predicted as graphs with stitch redundancy, and therefore, decomposed by our nonstitch GNN decomposer. Although

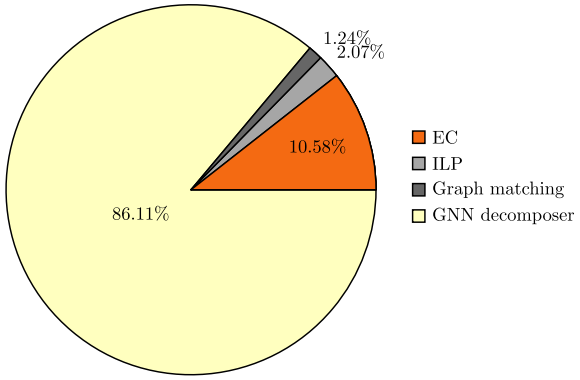


Fig. 10. Decomposer usage breakdown, i.e., the percentage of decompositions by using ILP/DL/*ColorGNN*/Matching.

ILP only decomposes 2.07% graphs, it still occupies most of the decomposition time of our framework due to its low efficiency.

F. Effectiveness of Redundant Stitch Prediction

In the fourth section, we demonstrate the effectiveness of our GNN-based stitch redundancy predictor empirically. The results are presented in Table VI in the form of a confusion matrix. Table VI(a) counts all instances, and Table VI(b) only counts instances whose prediction score is larger than 0.99 are selected. According to the results, our GNN-based predictor successfully predicts most redundancy, which largely improves the efficiency after eliminating these redundancies. More importantly, benefitting from the bar constraint, the prediction avoids any false prediction that predicts a nonredundant graph as redundant. A more detailed result for each circuit is shown in Table VII, where $|pred.\ ns\text{-}\mathcal{G}|$ is the number of successful predictions among all instances.

G. Effectiveness of Nonstitch GNN Decomposer

In the final experiment, we separately study the effectiveness of our proposed GNN-based decomposer, which is specifically for nonstitch graphs. The results are shown in Table VII, where

TABLE VI
F1 SCORE OF STITCH REDUNDANCY PREDICTION. THE RESULTS INCLUDE (a) ALL INSTANCES AND (b) INSTANCES WHOSE PREDICTION CONFIDENCE ARE ABOVE THE BAR. "REDUN." AND "NOT REDUN." REPRESENT WHETHER THE STITCH CANDIDATES IN A GRAPH ARE ALL REDUNDANT OR NOT

(a)

		Label	
		Redun.	Not Redun.
Pred.	Redun.	5962	46
	Not Redun.	55	498
Recall		99.23%	
F1-score		0.9916	

(b)

		Label	
		Redun.	Not Redun.
Pred.	Redun.	5730	0
	Not Redun.	2	185
Recall		**100.0%**	
F1-score		**0.9998**	

TABLE VII
LAYOUT STATISTICS AND RESULTS BY GNN DECOMPOSER

Circuit	$\|G\|$	$\|nsc\text{-}\mathcal{G}\|$	$\|ns\text{-}\mathcal{G}\|$	$\|pred.\ ns\text{-}\mathcal{G}\|$	ILP cost	GNN cost	ILP time	GNN time
C432	4	0	0	0	0	0	0	0.0135
C499	4	0	4	1	0	0	0.0041	0.0134
C880	8	0	1	0	0	0	0	0.0135
C1355	6	0	3	2	0	0	0.0045	0.0136
C1908	3	0	2	0	0	0	0	0.0135
C2670	9	0	3	0	0	0	0	0.0136
C3540	14	0	6	0	0	0	0	0.0140
C5315	16	0	7	2	0	0	0.0619	0.0135
C6288	200	0	25	17	0	0	0.0051	0.0143
C7552	39	1	16	3	0	0	0.0045	0.0135
S1488	8	0	6	6	0	0	0.0339	0.0135
S38417	670	3	613	584	16	16	2.8480	0.0169
S35932	2010	12	1958	1869	22	22	8.6960	0.0164
S38584	1936	3	1817	1735	22	22	8.1550	0.0171
S15850	1657	4	1556	1510	22	22	6.8680	0.0153
average	440.27	1.533	401.13	381.93	5.467	5.467	1.778	0.0152
ratio	1.000	0.003	0.911	0.867	**1.000**	**1.000**	1.000	**0.008**

$\mathcal{G}$ is the graph set after simplification and stitch insertion. *nsc*-$\mathcal{G}$ (no stitch candidate graph) is a subset of $\mathcal{G}$ in which graphs do not contain stitch edges. *ns*-$\mathcal{G}$ (nonstitch graph) is a subset of $\mathcal{G}$ in which the optimal decomposition results contain no stitches. *pred. ns*-$\mathcal{G}$ (predicted nonstitch graph) is a subset of $\mathcal{G}$ in which our proposed stitch redundancy predictor predicts that these graphs do not need stitch edges. ILP(GNN) cost represents the total cost decomposed by the ILP method (our proposed GNN decomposer) for graphs in *pred. ns*-$\mathcal{G}$, and ILP time is the total decomposition time by the ILP method for graphs in *pred. ns*-$\mathcal{G}$. GNN time is the total execution time by our GNN decomposer. Since we implement the decomposer in a batch-process manner, we use the GNN decomposer to decompose all graphs even before the stitch redundancy prediction rather than waiting for the prediction result of each case (note that the additional runtime is trivial for the fast inference). Therefore, in some layouts, such as C432, our GNN decomposer still decomposes some graphs though there is no redundant graphs according to the prediction, i.e., $|pred.\ ns\text{-}\mathcal{G}| = 0$.

According to Table VII, we can observe some statistical properties in the layout dataset. First, the existing stitch candidate generation algorithm will insert stitch candidates into most graphs. Among over 6000 graphs, only 23 graphs are free of stitch edges. However, we observe that most of these inserted stitch edges are not useful: in the final optimal results, 91.1% graphs contain no stitches, meaning that considerable generated stitch candidates are redundant. Our predictor can predict redundancy with high accuracy (381.93 over 401.13). Then, for graphs whose stitch edges are predicted as redundant, we can employ our GNN-based decomposer, which is specifically for the homogeneous graphs, i.e., graphs only containing conflict edges. As shown in the table, our GNN decomposer achieves the same result quality with the optimal ILP solver, with a large improvement in the efficiency (reduce to 0.8%). Although our GNN decomposer does not guarantee an optimal result for any layout graph due to the inserted randomness, such as random node attributes and random neighbor sampling, we experimentally show that our GNN decomposer achieves "optimal"İresults in the ISCAS benchmark. These results demonstrate that our GNN-based decomposer can be a practical option for the nonstitch decomposition problem.

VI. CONCLUSION

In this article, we used RGCNs to obtain graph embeddings, which are used to build the isomorphism-free graph library, match graphs in the library, adaptively select decomposer, and predict the stitch redundancy. We also proposed a pure GNN-based decomposer, which is applicable for nonstitch graphs. The results show that the obtained graph embeddings have powerful representation capability and demonstrate an excellent balance between decomposition quality and efficiency. Also, our GNN-based decomposer achieves an optimal results for nonstitch cases in the experimental benchmark with a huge runtime improvement.

REFERENCES

[1] D. Z. Pan, B. Yu, and J.-R. Gao, "Design for manufacturing with emerging nanolithography," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 10, pp. 1453–1472, Oct. 2013.

[2] Y. Xu and C. Chu, "GREMA: Graph reduction based efficient mask assignment for double patterning technology," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2009, pp. 601–606.

[3] A. B. Kahng, C.-H. Park, X. Xu, and H. Yao, "Layout decomposition approaches for double patterning lithography," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 6, pp. 939–952, Jun. 2010.

[4] K. Yuan, J.-S. Yang, and D. Z. Pan, "Double patterning layout decomposition for simultaneous conflict and stitch minimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 2, pp. 185–196, Feb. 2010.

[5] B. Yu, K. Yuan, D. Ding, and D. Z. Pan, "Layout decomposition for triple patterning lithography," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 3, pp. 433–446, Mar. 2015.

[6] B. Yu, Y.-H. Lin, G. Luk-Pat, D. Ding, K. Lucas, and D. Z. Pan, "A high-performance triple patterning layout decomposer with balanced density," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2013, pp. 163–169.

[7] B. Yu, S. Roy, J.-R. Gao, and D. Z. Pan, "Triple patterning lithography layout decomposition using end-cutting," *J. Micro/Nanolithogr. MEMS MOEMS*, vol. 14, no. 1, 2015, Art. no. 011002.

[8] Y. Lin, X. Xu, B. Yu, R. Baldick, and D. Z. Pan, "Triple/quadruple patterning layout decomposition via linear programming and iterative rounding," *J. Micro/Nanolithogr. MEMS MOEMS*, vol. 16, no. 2, 2017, Art. no. 023507.

[9] X. Li, Z. Zhu, and W. Zhu, "Discrete relaxation method for triple patterning lithography layout decomposition," *IEEE Trans. Comput.*, vol. 66, no. 2, pp. 285–298, Feb. 2017.

[10] S.-Y. Fang, Y.-W. Chang, and W.-Y. Chen, "A novel layout decomposition algorithm for triple patterning lithography," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 33, no. 3, pp. 397–408, Mar. 2014.

[11] H.-A. Chien, S.-Y. Han, Y.-H. Chen, and T.-C. Wang, "A cell-based row-structure layout decomposer for triple patterning lithography," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2015, pp. 67–74.

[12] H. Tian, H. Zhang, Q. Ma, Z. Xiao, and M. D. F. Wong, "A polynomial time triple patterning algorithm for cell based row-structure layout," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2012, pp. 57–64.

[13] J. Kuang and E. F. Y. Young, "Fixed-parameter tractable algorithms for optimal layout decomposition and beyond," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2017, p. 61.

[14] J. Kuang and E. F. Y. Young, "An efficient layout decomposition approach for triple patterning lithography," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2013, p. 69.

[15] I. H.-R. Jiang and H.-Y. Chang, "Multiple patterning layout decomposition considering complex coloring rules and density balancing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 12, pp. 2080–2092, Dec. 2017.

[16] H.-Y. Chang and I. H.-R. Jiang, "Multiple patterning layout decomposition considering complex coloring rules," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2016, p. 40.

[17] E. Bengoetxea, "Inexact graph matching using estimation of distribution algorithms," Ph.D. dissertation, Dept. Traitement du Signal et des Images, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec. 2002.

[18] T. Matsui, Y. Kohira, C. Kodama, and A. Takahashi, "Positive semidefinite relaxation and approximation algorithm for triple patterning lithography," in *Proc. Int. Symp. Algorithms Comput.*, 2014, pp. 365–375.

[19] B. Yu and D. Z. Pan, "Layout decomposition for quadruple patterning lithography and beyond," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2014, p. 53.

[20] D. Gamarnik and M. Sudan, "Limits of local algorithms over sparse random graphs," in *Proc. 5th Conf. Innov. Theor. Comput. Sci.*, 2014, pp. 369–376.

[21] W. Li *et al.*, "OpenMPL: An open source layout decomposer," in *Proc. IEEE Int. Conf. ASIC (ASICON)*, 2019, pp. 1–4.

[22] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *Proc. Eur. Semantic Web Conf.*, 2018, pp. 593–607.

[23] D. Stolee, "Isomorph-free generation of 2-connected graphs with applications," 2011, *arXiv:1104.5261*.

[24] B. D. McKay, "Isomorph-free exhaustive generation," *J. Algorithms*, vol. 26, no. 2, pp. 306–324, 1998.

[25] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *Proc. NeurIPS Autodiff Workshop*, 2017, pp. 1–4.

[26] M. Wang *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *Proc. ICLR Workshop Represent. Learn. Graphs Manifolds*, 2019, pp. 1–18.

[27] W. Li, J. Xia, Y. Ma, J. Li, Y. Lin, and B. Yu, "Adaptive layout decomposition with graph embedding neural networks," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

Wei Li received the B.Sc. and M.Phil. degrees in computer science from the Chinese University of Hong Kong, Hong Kong, in 2018 and 2021, respectively. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA.

His research interests include geometric deep learning and its applications on VLSI design and testing.

Mr. Li received Best Student Paper Award from ICTAI 2019, the Distinguished Paper Award from ISSTA 2019, and the Best Paper Award from ASPDAC 2021.

Yuzhe Ma (Member, IEEE) received the B.E. degree from the Department of Microelectronics, Sun Yat-sen University, Guangzhou, China, in 2016, and the Ph.D. degree from the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong, in 2020.

He is currently an Assistant Professor of Microelectronics Thrust with The Hong Kong University of Science and Technology (Guangzhou), Guangzhou. His research interests include agile VLSI design methodologies, machine learning-aided VLSI design, and hardware-friendly machine learning.

Dr. Ma received Best Paper Awards from ICCAD 2021, ASPDAC 2021, ICTAI 2019, and Best Paper Award Nomination from ASPDAC 2019.

Yibo Lin (Member, IEEE) received the B.S. degree in microelectronics from Shanghai Jiao Tong University, Shanghai, China, in 2013, and the Ph.D. degree from the Electrical and Computer Engineering Department, University of Texas at Austin, Austin, TX, USA, in 2018.

He is currently an Assistant Professor with the Computer Science Department associated with the Center for Energy-Efficient Computing and Applications, Peking University, Beijing, China. His research interests include physical design, machine learning applications, GPU acceleration, and hardware security.

Bei Yu (Member, IEEE) received the Ph.D. degree from The University of Texas at Austin, Austin, TX, USA, in 2014.

He is currently an Associate Professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

Dr. Yu received seven Best Paper Awards from ICCAD 2021, ASPDAC 2021, ICTAI 2019, *Integration, the VLSI Journal* in 2018, ISPD 2017, SPIE Advanced Lithography Conference 2016, ICCAD 2013, ASPDAC 2012, and six ICCAD/ISPD contest awards. He has served as a TPC Chair of ACM/IEEE Workshop on Machine Learning for CAD, and in many journal editorial boards and conference committees. He is an Editor of IEEE TCCPS Newsletter.