PreRoutGNN for Timing Prediction with Order Preserving Partition: Global Circuit Pre-training, Local Delay Learning and Attentional Cell Modeling

Ruizhe Zhong[1], **Junjie Ye**[2], **Zhentao Tang**[2], **Shixiong Kai**[2], **Mingxuan Yuan**[2], **Jianye Hao**[2, 3], **Junchi Yan**[1*]
[1]Dept. of Computer Science and Engineering & MoE Key Lab of AI, Shanghai Jiao Tong University
[2]Huawei Noah's Ark Lab
[3]College of Intelligence and Computing, Tianjin University
zerzerzerz271828@sjtu.edu.cn, kourenmu@163.com, tangzhentao1@huawei.com, kaishixiong@huawei.com, Yuan.Mingxuan@huawei.com, haojianye@huawei.com, yanjunchi@sjtu.edu.cn

Abstract

Pre-routing timing prediction has been recently studied for evaluating the quality of a candidate cell placement in chip design. It involves directly estimating the timing metrics for both pin-level (slack, slew) and edge-level (net delay, cell delay), without time-consuming routing. However, it often suffers from signal decay and error accumulation due to the long timing paths in large-scale industrial circuits. To address these challenges, we propose a two-stage approach. First, we propose global circuit training to pre-train a graph auto-encoder that learns the global graph embedding from circuit netlist. Second, we use a novel node updating scheme for message passing on GCN, following the topological sorting sequence of the learned graph embedding and circuit graph. This scheme residually models the local time delay between two adjacent pins in the updating sequence, and extracts the lookup table information inside each cell via a new attention mechanism. To handle large-scale circuits efficiently, we introduce an order preserving partition scheme that reduces memory consumption while maintaining the topological dependencies. Experiments on 21 real world circuits achieve a new SOTA R^2 of 0.93 for slack prediction, which is significantly surpasses 0.59 by previous SOTA method. Code will be available at: https://github.com/Thinklab-SJTU/EDA-AI.

Introduction

The process of integrated circuit (IC) design can be thought of as a series of hierarchical decomposition steps, typically consisting of architectural design, logic synthesis, physical design (macro & cell placement, routing) and verification. As IC becomes more and more complex, timing constraints, such as delay and slack, are becoming more difficult to satisfy during the physical design stage. 'Shift-left' (Zhou et al. 2022) suggests circuit constraints and performance should be considered in earlier stages of design flow, for instance, taking timing into consideration during standard cell placement stage (Liao et al. 2022). Timing metrics are critical to judge the performance of a design, but accurate timing information is only available after routing, which is one of the most time-consuming steps. Since repetitive routing in cell placement stage is unacceptable, many analytical placers (Lu et al. 2015; Lin et al. 2019) use half-perimeter wirelength as a surrogate of design timing quality.

Machine learning has been widely applied in EDA design flow, such as logic synthesis (Yuan et al. 2023), placement (Mirhoseini et al. 2021; Cheng and Yan 2021; Cheng et al. 2022; Lai, Mu, and Luo 2022; Lai et al. 2023) and routing (Du et al. 2023). However, accurate timing information can be accessed only after routing stage, which is time-consuming. To estimate pre-routing timing metrics, machine learning has been introduced for timing prediction, such as delay (Yang, He, and Cao 2022), wirelength (Xie et al. 2021; Yang et al. 2022), to guide timing-driven cell placement.

Among various timing metrics, slack (Hu, Sinha, and Keller 2014) is one of the most important yet challenging metric for prediction. Slack is the difference between Required signal Arrival Time (RAT) and actual Arrival Time (AT), and it is calculated on each pin. Accurate slack prediction can be accessed only after routing stage, which is a time consuming process. To avoid the time consuming routing stage, directly estimating timing information is attractive, which can be defined formally as follows: given the intermediate results after standard cell placement stage and before routing stage, we want to predict slack for all pins after routing stage. Usually, RAT is provided in external files, so we mainly focus on the prediction of AT. However, in large designs, the timing path can be long with hundreds of circuit elements, where signal decay and error accumulation problems exist, making slack prediction challenging. Besides, directly applying deep learning into large circuits are always faced with high peak GPU memory cost problem, making training process hard or even intractable.

Typical GNNs (Graph Neural Networks) such as GCN (Kipf and Welling 2017) and GAT (Veličković et al. 2018) are designed to process graphs where local information is already sufficient. However, in circuits graphs, signals travel from primary inputs to timing endpoints, forming long timing paths (Hu, Sinha, and Keller 2014), where long range dependencies and global view play a critical role but they are hard for typical GNN to handle. Previous methods for other timing metric (delay, wirelength) prediction (Xie et al. 2018; Barboza et al. 2019; Ghose et al. 2021; Xie et al. 2021; Yang, He, and Cao 2022; Yang et al. 2022)

*Correspondence author.

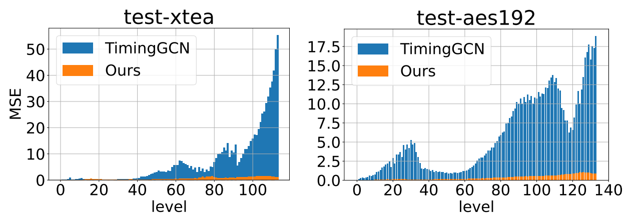


Figure 1: MSE of AT prediction v.s. topological level. In previous ML-based methods TimingGCN (Guo et al. 2022), MSE increases rapidly with level, showing the existence of signal decay and error accumulation.

do not take the global graph information into consideration. For example, CircuitGNN (Yang et al. 2022) converts each circuit to a heterogeneous graph, and design the topo-geom message passing paradigm for prediction of wirelength and congestion. Wirelength is estimated in each net using half-perimeter wirelength, and congestion is predicted in local area measured with pixel values. Both tasks are enclosed in local area, where local information is already sufficient. However, in pre-routing slack prediction, it relies all elements through timing paths, where long range dependencies and global view play a critical role, and they are absent in CircuitGNN. Consequently, we cannot directly apply these GNNs into the prediction of slack. Net^2 (Xie et al. 2021) for wirelength prediction applies graph partition and merges cluster embedding into node features, but cannot avoid the local receptive field enclosed in cluster. TimingGCN (Guo et al. 2022) utilizes a timing engine inspired GNN to predict slack, where node embedding is updated with predecessors asynchronously. However, it is prone to signal decay and error accumulation, demonstrated in Fig. 1. We first apply topological sorting (Lasser 1961) on circuit graph to get topological level (topological sorting order) of each pin. Then we calculate MSE (Mean Squared Error) between AT prediction and ground truth for pins belonging to the same sorting order. In other methods, MSE increases rapidly with level, demonstrating the existence of signal decay and error accumulation problems.

Motivated by these issues, we propose our two-stage approach for pre-routing slack prediction. In the first stage, we pre-train an auto-encoder to extract node and graph embeddings for following downstream tasks, providing a global receptive field and an informative representation. In the second stage, we design a new message passing and node updating scheme for long timing paths, and train our GNN for pre-routing slack prediction. We list features of different methods in Table 1. **The highlights of this work are:**

- **First time to introduce global circuit pre-training into timing (specifically slack) prediction.** Global view plays a critical role in addressing the signal decay and error accumulation issues. We capture this feature via graph embeddings, which can be effectively learned by global circuit pre-training technique. Besides, our global circuit pre-training can serve as a plug-and-play module for other timing prediction GNN.

Model	Update scheme	Global	Embed.	Level
GCN	Sync./Direct	✗	✗	✗
GAT	Sync./Direct	✗	✗	✗
GINE	Sync./Direct	✗	✗	✗
GCNII	Sync./Direct	✗	✗	✗
CircuitGNN	Sync./Direct	✗	✗	✗
TimingGCN	Async./Direct	✓	✗	✗
Net^2	Sync./Direct	✓	✓	✗
LHNN	Sync./Direct	✗	✗	✗
Ours	Async./Residual	✓	✓	✓

Table 1: Features comparison among different methods. Residual means ours node updating scheme in Eq. 3, which explicitly models net delay and cell delay. Embed. means graph embeddings. Level means we introduce topological sorting order into GNN.

- **Residual local learning of signal delay.** We design a new message passing and node updating scheme, with residual local learning to explicitly model local signal delay in long time paths.
- **Attention-based cell modeling.** We re-explore the modeling process of cell and explain it as a 'query-index-interpolation' procedure. Based on this, we propose multi-head joint attention mechanism to depict it.
- **Handling large-scale and strong performance.** For scalability and making training on large-scale circuits tractable, we design a order preserving graph partition algorithm to reduce memory cost while preserving the order dependencies. Experiments on public benchmark with 21 real world circuits demonstrate the effectiveness of our method. For R^2 of slack prediction, we achieve new SOTA 0.93, surpassing 0.59 of peer SOTA method.

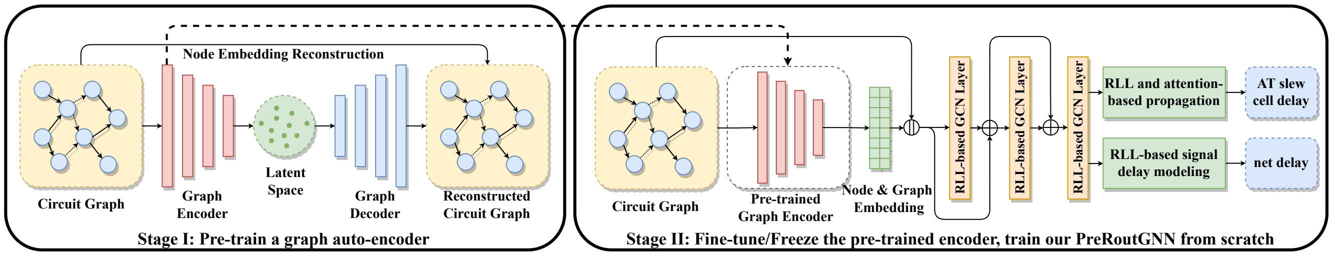
Preliminaries and Related Works

Static Timing Analysis Static timing analysis (STA) provides a profile of a design's performance by evaluating the timing propagation from inputs to outputs, forming multiple timing paths. A timing path (Hu, Sinha, and Keller 2014) is a set of directed connections through circuit elements and connections, and all signals travel through it. After routing, STA should be done under four corner conditions: early-late/rise- ? ?
fall (EL/RF) (Hu, Sinha, and Keller 2014).

During STA, the circuit is represented as an Activity-On-Edge network, where each pin is a node and pin-to-pin connection is an edge. The activity cost on each edge is either net delay or cell delay. Starting from primary inputs (source nodes), the instant when a signal reaches a pin is quantified as the arrival time (AT). Similarly, starting from the primary outputs (sink nodes), the limits imposed for each AT to ensure proper circuit operation is defined as the required arrival time (RAT). The slack is defined as the difference between RAT and AT as follows:

$$slack^E = AT^E - RAT^E, slack^L = RAT^L - AT^L, \quad (1)$$

where E, L means early and late, and a positive slack means timing constraint is satisfied. Since RAT is usually provided



Node Embedding Reconstruction
Circuit Graph
Graph Encoder
Latent Space
Graph Decoder
Reconstructed Circuit Graph
Stage I: Pre-train a graph auto-encoder
Circuit Graph
Pre-trained Graph Encoder
Node & Graph Embedding
RLL-based GCN Layer
RLL-based GCN Layer
RLL-based GCN Layer
RLL and attention-based propagation
AT slew cell delay
RLL-based signal delay modeling
net delay
Stage II: Fine-tune/Freeze the pre-trained encoder, train our PreRoutGNN from scratch

Figure 2: Pipeline of our approach. Circuit graph is represented as a heterogeneous DAG. We first implement training an auto-encoder by global circuit reconstruction in a self-supervised way. After global circuit pre-training, we drop the decoder, and freeze/fine-tune the encoder to map each circuit graph to a low-dimensional latent space. The latent vector is treated as the global graph embedding and concatenates to original node features. Finally, heterogeneous circuit DAG is feed forward to the our PreRoutGNN for timing prediction.

in external files, so we mainly focus the estimation of AT, and use provided RAT and estimated AT to calculate slack.

Besides, the signal transition is characterized by its slew, which is defined as the required amount time of transition between electrical levels. Slew is also an important timing metric to evaluation the quality of circuit, and accurate slew can be accessed by STA after routing.

Pre-training in EDA As training from scratch requires much time and data, pre-training has been recently and widely applied in vision (Bao et al. 2022) and language (Devlin et al. 2018; Brown et al. 2020), and it is becoming a promising trend in EDA. Using pre-trained model not only saves these costs but provides more informative representation. DeepSeq (Khan et al. 2023), DeepGate (Li et al. 2022) and Bulls-Eye (Chowdhury et al. 2022) transform circuits into AND-Inverter-Graph format and learn corresponding representation for logic synthesis. DeepPlace (Cheng and Yan 2021) uses reinforcement learning agent pre-trained on macro connection to jointly solve placement and routing. However, few works about pre-training have been done in timing prediction, especially slack. We think pre-training can provide a graph embedding containing global view, which plays a critical role in addressing the signal decay and error accumulation in long timing paths.

GNNs for Timing Prediction GNNs are optimizable transformations for graph-structured data attributes. As they continue to evolve, new variants are emerging that target graph topological structure (Zhao et al. 2023) and large-scale characteristics (Wu et al. 2022, 2023a,b). Due to the graph structure of circuit netlist, it is natural to apply GNN in EDA field. GCN (Defferrard, Bresson, and Vandergheynst 2016; Kipf and Welling 2017) is proposed as a generalization from low-dimensional regular grids to high-dimensional irregular domains. GAT (Veličković et al. 2018) introduces attention mechanism to assign learnable weights for neighbours. GCNII (Chen et al. 2020) introduces initial residual and identity mapping to mitigate over-smoothing problem. GINE (Hu et al. 2020) is a variant of GIN (Xu et al. 2019) with additional edge features. At first, it is designed to address the graph isomorphism problem, but it also demonstrates strong ability in other graph related tasks, e.g. node classification. However, aforementioned GNNs are not specifically designed for circuits containing long timing paths, thus they are faced with local receptive field and over-smoothing problems. CircuitGNN (Yang et al. 2022) converts each circuit to a heterogeneous graph, and designs the topo-geom message passing paradigm for prediction of wirelength and congestion. TimingGCN (Guo et al. 2022) is a timing engine inspired GNN especially for pre-routing slack prediction. LHNN (Wang et al. 2022a) uses lattice hyper-graph neural network to prediction routing congestion. Net^2 (Xie et al. 2021) applies graph attention network into pre-placement net length estimation.

Methodology

Formulation. Given the intermediate design result after placement and before routing, we predict the AT and slack for all pins in the circuit after routing. Each circuit is represented as a heterogeneous and directed acyclic graph (DAG), containing one type of node and three types of edge. Primary inputs (input ports), primary outputs (output ports), and pins are treated as one type of node. Cell, net and net_inv are three types of edges, corresponding to fan-in to fan-out, net driver to net sink, net sink to net driver connection, respectively. Pipeline of our approach is shown in Fig. 2.

Global Circuit Pre-training

Due to the local receptive field of common message passing paradigm based GNNs, the global view is often absent. It plays an essential role in accurately modeling the timing feature along timing path. As a consequence, the lack of global view leads to the signal decay and error accumulation problems. Motivated by this issue, we capture this feature via graph embeddings, which can be effectively learned by global circuit pre-training.

The first stage of our approach is to train an auto-encoder for graph reconstruction in a self-supervised way. We use it to explicitly extract the global embedding for downstream tasks. It provides a global view for the whole graph, which is absent in previous methods. Since the pre-training is implemented in a self-supervised way, circuits without timing labels (AT, slew, delay) can also be used for graph reconstruction, saving the cost during complete design flow, e.g.

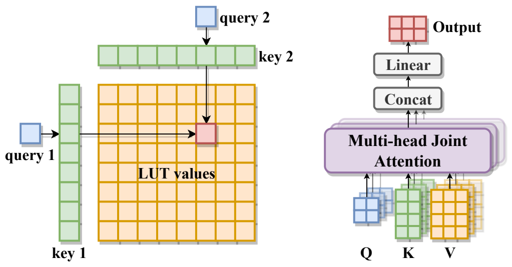


Figure 3: Multi-head Joint Attention (MJA) for cell modeling. To model this querying-indexing-interpolation process, we model it with a joint attention.

routing and STA, to access accurate labels.

The total loss of training auto-encoder is as follows:

$$\mathcal{L}_{AE} = \|\mathbf{X} - D_{\varphi}\left(E_{\theta}(g, \mathbf{X})\right)\|_2^2 + \boxed{\lambda \mathcal{L}_{KL},} \tag{2}$$

where g is the circuit graph, $\mathbf{X}$ is the node feature. To avoid arbitrarily high variance in latent space, we add a regularization term, a slight Kullback-Leibler divergence penalty towards a standard normal distribution, where λ is the KL-penalty weight. We train auto-encoder and our GNN with the same data, but timing labels do not involve this pre-training stage. After that, we drop the decoder and freeze/fine-tune the encoder to get the node embeddings in latent space. The global graph embedding is extracted through a global average pooling layer. We concatenate these embeddings to original node features and feed them to following GNN.

Residual Local Learning of Signal Delay

In line with TimingGCN (Guo et al. 2022), we define *topological level (level for short)* as a sub-set of whole graph's vertex set, and all vertices in a level have the same topological sorting order. Each vertex belongs to one and only one level. Given a level, we can set the sorting order of any its vertices as its level index, and this level index is unique.

From Fig. 1 we can see that, in previous estimation methods, MSE increases rapidly with topological level, suggesting the existence of signal decay and error accumulation. To mitigate these problems, we propose our message passing and node updating scheme for long timing paths.

Considering a circuit and its signal propagation, e.g. the computation of AT, the circuit can be represented as an Activity-On-Edge network (Hu, Sinha, and Keller 2014), where the activity cost on each edge is net delay or cell delay. The AT of current node i is computed as $AT_i = \max_{j \in \mathcal{N}(i)} (AT_j + d_{ji})$, where $\mathcal{N}(i)$ is the predecessors of i and d_{ji} is the delay from j to i.

To explicitly model this delay, we introduce residual local learning (RLL) in our node updating scheme:

$$\begin{aligned} \mathbf{m}_e^{(t+1)} &= \mathbf{x}_u^{(t)} + \phi\left(\mathbf{x}_u^{(t)}, \mathbf{x}_v^{(t)}, \mathbf{f}_e^{(t)}\right), e \in \mathcal{E} \\ \mathbf{x}_v^{(t+1)} &= \mathbf{x}_v^{(t)} + \rho\left(\left\{\mathbf{m}_e^{(t+1)} : e \in \mathcal{E}\right\}\right), \end{aligned} \tag{3}$$

where $\mathbf{x}_v \in \mathbb{R}^{d_1}$ is the feature for node v, $e = (u, v)$ is the directed edge from u to v, $\mathbf{f}_e \in \mathbb{R}^{d_2}$ is the feature for edge e, $\mathcal{E}$ is the edge set, t is the GNN layer index, ϕ is the message function, ρ is the reduce function, and ψ is the update function. This inductive bias better depicts the signal computation process and learning incremental value is the direct manifestation of learning cell delay and net delay.

Topological level encoding. Node topological sorting order is useful to address the signal decay and error accumulation, as it indicates the node location in timing path. We incorporate topological level encoding to map the level index to a multi-frequency vector. Inspired by NeRF (Mildenhall et al. 2021) and Transformer (Vaswani et al. 2017), we use following encoding:

$$\begin{aligned} \gamma(x) = [&x, \sin(2^0\pi x/L), \cos(2^0\pi x/L), \sin(2^1\pi x/L), \\ &\cos(2^1\pi x/L), \ldots, \sin(2^{N-1}\pi x/L), \cos(2^{N-1}\pi x/L)], \end{aligned} \tag{4}$$

where x is the level index of each pin, N is the number of different frequency components and L is the maximum level. This level encoding explicitly incorporate high-frequency components into node embedding, assisting the GNN to identify the level information. Besides, the multi-head joint attention we proposed in the following section can be explained as a low-pass filter (Wang et al. 2022b; Park and Kim 2022) The level encoding enhances and amplifies the high-frequency components, keeping them active during forward propagation. As a result, it is beneficial to model the topological order.

Multi-head Joint Attention for Cell Modeling

The output slew and cell delay for cell can be represented by the non-linear delay model (Hu, Schaeffer, and Garg 2015) stored in two-dimensional look-up tables (LUTs) in external files, e.g. Liberty files. The row and column indices of LUTs are pre-defined input slew and driving load. The values of LUTs are pre-defined cell delay or output slew. In STA, given an input slew and a driving load, the corresponding output slew or cell delay can be calculated by referencing the table with two input indices and applying bi-linear interpolation, which is a 'query-index-interpolation' procedure.

From the perspective of attention mechanism, input slew and driving load can be treated as queries, LUT's row and column as keys, and table of LUT as values. Since the two kinds of keys are not independent, we use multi-head joint attention (MJA) to model cell as follows:

$$\begin{aligned} \mathbf{K} &= \mathbf{K}_1 \times \mathbf{K}_2 \in \mathbb{R}^{n \times n \times d_k} \xrightarrow{\text{reshape}} \mathbb{R}^{n^2 \times d_k} \\ \mathbf{O} &= \text{softmax}\left(\frac{(\mathbf{Q}\mathbf{W}_Q)(\mathbf{K}\mathbf{W}_K)^\top}{\sqrt{d}}\right)\mathbf{V}\mathbf{W}_V, \end{aligned} \tag{5}$$

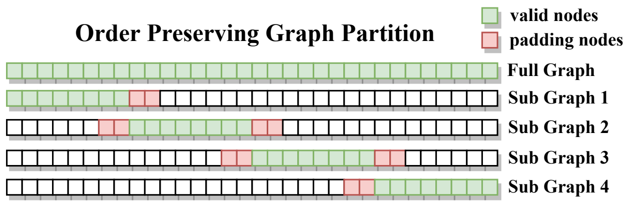


Figure 4: Illustration for order preserving graph partition. Padding nodes are not involved in loss computing and BP.

```
Algorithm 1: Order Preserving Graph Partition
Input: Circuit directed acyclic graph $G$, maximum size $m$
    for sub-graph, number of padding levels $k$.
Output: All partitioned sub-graphs $G_s$ of $G$.
 1: function TopologicalSort($G$)
 2:     $L$ = [ ];
 3:     # $l$ is a list of nodes belonging to the same level
 4:     for level $l$ in $G$ do
 5:         $L$.append($l$);
 6:     end for
 7:     return $L$
 8: end function
 9: $L$ = TopologicalSort($G$);
10: $i = 0, j = 0, g_s =$[ ]$, G_s =$[ ];
11: while $i < \text{len}(L)$ do
12:     if $i < \text{len}(L)$ and $\text{len}(g_s) + \text{len}(L[i]) < m$ then
13:         Add all nodes in $L[i]$ to $g_s$;    $i = i + 1$;
14:     else
15:         # Padding nodes do not involve training
16:         $g_s = L[\max(0, j-k) : j] \,||\, g_s \,||\, L[i : \min(i+k, \text{len}(L))]$;
17:         $j = i$;
18:         Construct sub-graph with nodes $g_s$ from $G$;
19:         Add the sub-graph to $G_s$;
20:         $g_s =$ [ ];
21:     end if
22: end while
```

where m, n^2 is the number of queries and keys/values respectively, $\mathbf{Q} \in \mathbb{R}^{m \times d_q}, \mathbf{K}_1, \mathbf{K}_2 \in \mathbb{R}^{n \times d_k}, \mathbf{V} \in \mathbb{R}^{n^2 \times d_v}$, $\mathbf{K}_{ijk} = (\mathbf{K}_1)_{ik}(\mathbf{K}_2)_{jk}$ is the joint key matrix. $\mathbf{K}_1, \mathbf{K}_2$ are key matrices from LUT rows and columns, $\mathbf{Q}$ is value matrix from LUT value, and $\mathbf{Q}$ is the query matrix. Our MJA is demonstrated in Fig. 3. We calculate AT for node i as follows if it is the end point of cell edge (edge type is 'cell'):

$$\begin{aligned}
\mathbf{Q}_{ji} &= \text{MLP}(\mathbf{f}_i || \mathbf{f}_j || \mathbf{AT}_j) + \mathbf{AT}_j \\
\mathbf{O}_{ji} &= \text{MJA}(\mathbf{Q}_{ji}, \mathbf{K}_{ji}, \mathbf{V}_{ji}) \\
\mathbf{Y}_{ji} &= \text{LayerNorm}(\mathbf{O}_{ji} + \mathbf{Q}_{ji}) \\
\mathbf{m}_{ji} &= \text{MLP}(\mathbf{f}_i || \mathbf{f}_j || \mathbf{AT}_j || \mathbf{Y}_{ji}) + \mathbf{AT}_j \\
\mathbf{a}_i &= \frac{\sum_{j \in \mathcal{N}(i)} \mathbf{m}_{ji}}{\text{size}(\mathcal{N}(i))}, \mathbf{b}_i = \max_{j \in \mathcal{N}(i)} \mathbf{m}_{ji} \\
\mathbf{AT}_i &= \text{MLP}(\mathbf{a}_i || \mathbf{b}_i || \mathbf{f}_i).
\end{aligned} \tag{6}$$

Order Preserving Graph Partition

Feeding the entire graph into GNN for training is faced with huge peak GPU memory cost. An appropriate graph partition method is necessary to reduce the memory cost while reserving topological order information.

To adapt to the signal propagation, we propose a order preserving graph partition algorithm. Given a circuit graph, we first apply topological sorting (Lasser 1961) to compute the sorting order of each pin. After sorting, we construct sub-graphs by selecting continuous levels of pins based on the given maximum sub-graph size. However, the marginal nodes of sub-graph lose their $1 \sim k$ hop (1 to k) neighbours, and the inner nodes also lose $2 \sim k, 3 \sim k, \ldots, k$-th hop neighbours, where k is the number of stacked GNN layers. To resolve this problem, we pad the sub-graph with k preceding and successive levels of pins. Note that these padding nodes are excluded in loss computation and backward propagation (BP), which ensures nodes involved gradient descent have full access to their k-hop neighbours like in the whole graph. Detailed graph partition algorithm is shown in Alg. 1 and Fig. 4. Our partition algorithm reduces the GPU memory cost from more than 48 GB (out of memory) to 22 GB, and preserves the consistency without losing neighbours.

Finally, we build our **PreRoutGNN** (**Pre-Rout**ing estimation **GNN**), which consists of stacked residual local learning-based GCN layers and a attention-based propagation layer. Extracted node embedding, graph embedding from pre-trained auto-encoder and multi-frequency level encoding are concatenated to the original node features. Graph is firstly fed into stacked GCN layers. For the forward process of attention-based propagation layer, we apply topological sorting on the circuit graph to calculate the order of each node and select nodes with the same order as a level. We feed this single level into the attention-based propagation layer, and calculate AT for its vertices. Note that we feed each level into attention-based propagation layer with ascending order, ensuring that when calculating AT of current level of nodes, AT of all their predecessors have already been calculated. Nodes belonging to the same level are updated first, followed by nodes of next order until to the last. This process realizes the asynchronous calculation of AT for each node, enabling the signal propagation along timing paths. Our model predicts AT as a main task, with slew, net delay and cell delay prediction as auxiliary tasks following (Guo et al. 2022). Complete pipeline is shown in Fig. 2.

The total loss for second stage is as follows:

$$\begin{aligned}
\mathcal{L}_{AS} &= \|M_{AS}^{\psi_2}(M^{\psi_1}(g, \mathbf{X}, E_\theta(g, \mathbf{X}))) - AS\|_2^2 \\
\mathcal{L}_{CD} &= \|M_{CD}^{\psi_3}(M^{\psi_1}(g, \mathbf{X}, E_\theta(g, \mathbf{X}))) - CD\|_2^2 \\
\mathcal{L}_{ND} &= \|M_{ND}^{\psi_4}(M^{\psi_1}(g, \mathbf{X}, E_\theta(g, \mathbf{X}))) - ND\|_2^2 \\
\mathcal{L}_{GNN} &= \mathcal{L}_{AS} + \lambda_{CD}\mathcal{L}_{CD} + \lambda_{ND}\mathcal{L}_{ND},
\end{aligned} \tag{7}$$

where g is the circuit graph, ψ_1 to ψ_4 are trainable parameters of GNN, $\mathbf{X}$ is node feature. CD, ND, AS are cell delay, net delay, AT concatenated with slew, $\lambda_{CD}, \lambda_{ND}$ are loss weights for cell delay and net delay.

Experiments

Evaluation Protocols

Benchmarks and compared methods. We evaluate our approach on public benchmark (Guo et al. 2022) with 21 real world circuits. Given the circuit after standard cell placement, the objective is to predict slack, slew, cell delay and net delay after routing. The compared methods include GCNII (Chen et al. 2020), GAT (Veličković et al. 2018), GINE (Hu et al. 2020) and TimingGCN (Guo et al. 2022).

Metrics. We evaluate the prediction of slack with the R^2 determination coefficient score as follows:

$$R^2 = 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2} = 1 - \frac{\text{MSE}(\mathbf{y}, \hat{\mathbf{y}})}{\text{VAR}(\mathbf{y})}, \tag{8}$$

	Dataset	R^2_{uf} (↑)				
		GCNII	GAT	GINE	TimingGCN	PreRoutGNN
train	BM64	0.763 ± 0.101	0.354 ± 0.223	0.623 ± 0.161	0.668 ± 0.252	**0.986 ± 0.006**
	aes128	0.458 ± 0.207	-0.245 ± 0.340	0.029 ± 0.352	-0.055 ± 0.312	**0.822 ± 0.008**
	aes256	0.581 ± 0.151	-0.097 ± 0.306	0.067 ± 0.311	0.499 ± 0.102	**0.937 ± 0.023**
	aes_cipher	0.020 ± 0.447	-0.438 ± 0.427	-0.162 ± 0.242	0.644 ± 0.123	**0.967 ± 0.000**
	blabla	0.889 ± 0.073	0.679 ± 0.131	0.737 ± 0.122	0.958 ± 0.013	**0.997 ± 0.001**
	cic_decimator	0.719 ± 0.112	0.597 ± 0.163	0.589 ± 0.107	0.963 ± 0.014	**0.996 ± 0.000**
	des	0.924 ± 0.056	0.566 ± 0.225	0.776 ± 0.220	0.905 ± 0.058	**0.993 ± 0.001**
	genericfir	0.710 ± 0.122	-0.436 ± 0.733	0.503 ± 0.219	0.222 ± 0.123	**0.871 ± 0.013**
	picorv32a	0.672 ± 0.096	0.239 ± 0.240	0.425 ± 0.169	0.839 ± 0.045	**0.974 ± 0.007**
	salsa20	0.814 ± 0.066	0.530 ± 0.170	0.717 ± 0.073	0.792 ± 0.118	**0.994 ± 0.002**
	usb	0.747 ± 0.088	0.678 ± 0.115	0.607 ± 0.118	0.934 ± 0.035	**0.996 ± 0.000**
	usb_cdc_core	0.793 ± 0.095	0.502 ± 0.212	0.563 ± 0.177	0.961 ± 0.008	**0.997 ± 0.001**
	wbqspiflash	0.742 ± 0.091	0.613 ± 0.131	0.522 ± 0.136	0.956 ± 0.013	**0.994 ± 0.002**
	zipdiv	0.822 ± 0.070	0.703 ± 0.129	0.593 ± 0.188	0.971 ± 0.011	**0.998 ± 0.001**
test	aes192	-0.017 ± 0.188	-0.727 ± 0.173	-0.048 ± 0.352	0.384 ± 0.205	**0.937 ± 0.009**
	jpeg_encoder	0.056 ± 0.080	-2.859 ± 0.667	-0.285 ± 0.198	0.478 ± 0.131	**0.764 ± 0.010**
	spm	-0.247 ± 0.309	-0.474 ± 0.134	0.040 ± 0.161	0.802 ± 0.083	**0.985 ± 0.006**
	synth_ram	-0.913 ± 0.031	-1.078 ± 0.025	-1.007 ± 0.162	0.735 ± 0.079	**0.977 ± 0.008**
	usbf_device	-0.215 ± 0.065	-0.946 ± 0.146	-0.133 ± 0.094	0.347 ± 0.119	**0.949 ± 0.002**
	xtea	0.210 ± 0.044	-0.055 ± 0.091	0.099 ± 0.181	0.888 ± 0.062	**0.985 ± 0.000**
	y_huff	-0.235 ± 0.450	-0.481 ± 0.114	-1.034 ± 0.999	0.490 ± 0.146	**0.931 ± 0.029**
	Avg. train	0.689	0.303	0.471	0.733	**0.966**
	Avg. test	-0.195	-0.946	-0.338	0.589	**0.933**

Table 2: Slack prediction with R^2_{uf} as primary metric. Experiments are implemented on 5 different seeds.

	Dataset	R^2_{uf} (↑)					
		baseline	w/o AE	w/o RLL	w/o MJA	w/o FT	PreRoutGNN
test	aes192	0.649	0.603	0.825	0.878	**0.966**	0.946
	jpeg_encoder	0.271	**0.787**	-0.338	0.605	0.620	0.774
	spm	0.696	0.883	0.988	0.987	0.977	**0.992**
	synth_ram	0.763	0.423	0.975	0.958	0.949	**0.984**
	usbf_device	0.317	0.659	0.676	0.844	0.931	**0.951**
	xtea	0.781	0.921	0.976	0.955	0.980	**0.985**
	y_huff	0.552	0.766	0.883	0.885	**0.934**	0.901
	Avg. train	0.720	0.959	0.876	0.933	0.963	**0.964**
	Avg. test	0.576	0.720	0.772	0.873	0.908	**0.933**

Table 3: Ablation studies. We remove one and only one module each time. R^2_{uf} of slack prediction is reported. AE: global pre-trained auto-encoder, RLL: residual local learning, MJA: multi-head joint attention, FT: fine-tune the pre-trained encoder. Bold/underline indicates best/second.

where y_i is ground truth, $\hat{y}$ is prediction and $\bar{y}$ is the mean value of ground truth. Since slack has four channels (EL/RF), we can calculate R^2 in two ways: **1)** ***un-flatten*** R^2_{uf} and **2)** ***flatten*** R^2_f . In R^2_{uf}, we calculate R^2 on *un-flatten* data with shape $\mathbb{R}^{n\times 4}$, where n is the number of nodes. We calculate R^2 on each channel independently and average these four R^2 as the final R^2_{uf}. In R^2_f, we *flatten* the data to $\mathbb{R}^{4n}$, merging four different channels together and the number of nodes increases from n to $4n$. Under R^2_f, previous method achieves 0.866. However, under R^2_{uf}, it only achieves 0.576.

After reforming R^2 in Eq. 8, we calculate it with VAR (variance) of $\mathbf{y}$, MSE(Mean Squared Error) between $\mathbf{y}$ and
方差

$$R^2 = 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{\frac{1}{n}\sum_{i=1}^{n}(y_i-\bar{y})^2} = 1 - \frac{\text{MSE}(\mathbf{y},\hat{\mathbf{y}})}{\text{VAR}(\mathbf{y})},$$

$\hat{\mathbf{y}}$. After merging four different channels into a single channel, both MSE and VAR increase, but VAR increases more than MSE, resulting in the improvement of R^2. Let $k = \text{VAR}(\mathbf{y})/\text{MSE}(\mathbf{y},\hat{\mathbf{y}})$. Since slack has four channels, we compute k on each channel and average them to get the mean k_{uf}. Flatten ratio k_f is computed in the same way, except for flattening slack from $\mathbb{R}^{n\times 4}$ to $\mathbb{R}^{4n}$ with only one channel. With $k_f/k_{uf} > 1$ we can see that, after flattening the multi-channel slack, VAR increases faster and higher than MSE, leading to the improvement about 0.3 of R^2.

We think the 'distance', e.g. Kullback-Leibler divergence, between distributions under different corners is high. The seemingly high flatten R^2_f cannot reflect the real perfor-

	Dataset	R^2_{uf} slew (↑) TimingGCN	R^2_{uf} slew (↑) PreRoutGNN	R^2_{uf} net delay (↑) TimingGCN	R^2_{uf} net delay (↑) PreRoutGNN	R^2_{uf} cell delay (↑) TimingGCN	R^2_{uf} cell delay (↑) PreRoutGNN
test	aes192	0.816 ± 0.037	**0.967 ± 0.005**	0.954 ± 0.010	**0.979 ± 0.001**	0.702 ± 0.244	**0.975 ± 0.005**
	jpeg_encoder	0.783 ± 0.038	**0.949 ± 0.000**	0.966 ± 0.004	**0.981 ± 0.001**	0.636 ± 0.233	**0.947 ± 0.011**
	spm	0.904 ± 0.017	**0.981 ± 0.001**	0.920 ± 0.010	**0.940 ± 0.019**	0.831 ± 0.037	**0.981 ± 0.003**
	synth_ram	0.764 ± 0.042	**0.846 ± 0.073**	0.929 ± 0.014	**0.974 ± 0.001**	0.533 ± 0.332	**0.682 ± 0.249**
	usbf_device	0.859 ± 0.023	**0.924 ± 0.005**	0.968 ± 0.001	**0.970 ± 0.003**	0.824 ± 0.072	**0.976 ± 0.004**
	xtea	0.824 ± 0.072	**0.978 ± 0.002**	0.944 ± 0.011	**0.963 ± 0.000**	0.574 ± 0.303	**0.968 ± 0.007**
	y_huff	0.880 ± 0.026	**0.896 ± 0.002**	0.940 ± 0.014	**0.964 ± 0.005**	**0.844 ± 0.066**	0.813 ± 0.118
	Avg. train	0.883	**0.968**	0.982	**0.992**	0.832	**0.980**
	Avg. test	0.833	**0.934**	0.946	**0.967**	0.706	**0.906**

Table 4: Comparison of other tasks, including prediction of slew, net delay and cell delay.

	Dataset	R^2_{uf} (↑) TimingGCN	R^2_{uf} (↑) + Pre-training
test	aes192	0.384 ± 0.205	**0.654 ± 0.097**
	jpeg_encoder	0.478 ± 0.131	**0.505 ± 0.268**
	spm	0.802 ± 0.083	**0.944 ± 0.020**
	synth_ram	0.735 ± 0.079	**0.943 ± 0.023**
	usbf_device	0.347 ± 0.119	**0.525 ± 0.184**
	xtea	0.888 ± 0.062	**0.946 ± 0.026**
	y_huff	0.490 ± 0.146	**0.816 ± 0.046**
	Avg. train	0.733	**0.781**
	Avg. test	0.589	**0.762**

Table 5: Slack prediction with R^2_{uf}. After applying pre-trained graph encoder into TimingGCN, R^2_{uf} improves 29 %, showing that our pre-trained graph encoder can serve as a plug-and-play module for training.

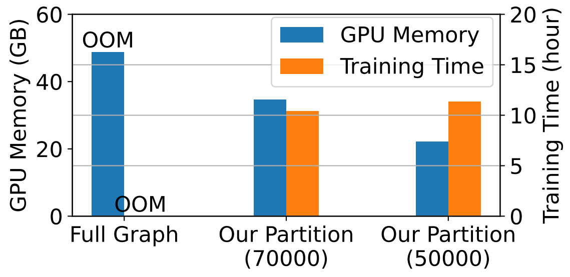


Figure 5: GPU memory cost and training time comparison between training on whole graphs and partitioned graphs. Number in brackets indicates the maximum sub-graph size.

mance of model, and it is an inappropriate evaluation metric. Thus, it is more reasonable to calculate R^2 on each channel independently and use R^2_{uf} as the final evaluation metric.

Results and Further Studies

Overall performance comparison. We use R^2_{uf} as primary metric. In Table 2, our approach achieves R^2_{uf} of 0.93, while the second best method only achieves 0.59.

Evaluation on other tasks. We also compare effectiveness on other tasks, including prediction of slew, net delay and cell delay. In Table 4, for R^2_{uf} on testing set, we achieve 0.93, 0.97, 0.91, surpassing the best baseline achieving 0.83, 0.95, 0.71 respectively, demonstrating that our model is also adaptive to multiple pre-routing timing prediction tasks.

Ablation studies. We ablate the effectiveness of each module of our method. In Table 3, we remove one and only one component each time. After removing global pre-training or RLL, R^2_{uf} drops dramatically, revealing the importance of global view and local modeling for signal delay. We also compare fine-tuning with freezing the pre-trained encoder. Even without fine-tuning, our model behaves well, only with a small drop 0.02 of R^2_{uf}, suggesting the importance of global circuit pre-training: providing a global view and a representation of circuit. We can also apply our global circuit pre-trained graph encoder to other baselines. Training other baselines along with fine-tuning graph encoder achieves better results with improvement of 29% in Table 5. The pre-trained graph encoder serves as a plug-and-play module for downstream tasks.

Runtime and GPU Memory cost. In Fig. 5, our method significantly reduces the peak GPU memory cost. Empirically, maximum sub-graph size 50,000 performs well. For large-scale circuits with about 300,000 pins, inference can be finished within one second.

Conclusion

We propose a novel pre-routing timing prediction method, PreRoutGNN, to address the signal decay and error accumulation issues. We use global circuit pre-training to provide global view, and devise residual local learning, MJA for long timing path modeling. Besides, our global circuit pre-training can serve as a plug-and-play module for various timing prediction GNN. For tractability on large-scale circuits, we also devise order preserving partition scheme to reduce memory cost while preserving the topological dependencies. We believe our works demonstrate accurate modeling of signal propagation along timing paths.

Acknowledgements

This work was partly supported by China Key Research and Development Program (2020AAA0107600) and NSFC (92370132, 62222607).

References

Ajayi, T.; Blaauw, D.; Chan, T.; Cheng, C.; Chhabria, V.; Choo, D.; Coltella, M.; Dobre, S.; Dreslinski, R.; Fogaça, M.; et al. 2019. OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain. *Proc. GOMACTECH*.

Bao, H.; Dong, L.; Piao, S.; and Wei, F. 2022. Beit: Bert pre-training of image transformers. In *ICLR*.

Barboza, E. C.; Shukla, N.; Chen, Y.; and Hu, J. 2019. Machine learning-based pre-routing timing prediction with reduced pessimism. In *DAC*.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *NeurIPS*.

Chen, M.; Wei, Z.; Huang, Z.; Ding, B.; and Li, Y. 2020. Simple and deep graph convolutional networks. In *ICML*.

Cheng, R.; Lyu, X.; Li, Y.; Ye, J.; Hao, J.; and Yan, J. 2022. The Policy-gradient Placement and Generative Routing Neural Networks for Chip Design. In *NeurIPS*.

Cheng, R.; and Yan, J. 2021. On joint learning for solving placement and routing in chip design. In *NeurIPS*.

Chowdhury, A. B.; Tan, B.; Carey, R.; Jain, T.; Karri, R.; and Garg, S. 2022. Too Big to Fail? Active Few-Shot Learning Guided Logic Synthesis. *arXiv preprint arXiv:2204.02368*.

Defferrard, M.; Bresson, X.; and Vandergheynst, P. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *NeurIPS*.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Du, X.; Wang, C.; Zhong, R.; and Yan, J. 2023. Hubrouter: Learning global routing via hub generation and pin-hub connection. In *NeurIPS*.

Ghose, A.; Zhang, V.; Zhang, Y.; Li, D.; Liu, W.; and Coates, M. 2021. Generalizable Cross-Graph Embedding for GNN-based Congestion Prediction. In *ICCAD*.

Guo, Z.; Liu, M.; Gu, J.; Zhang, S.; Pan, D. Z.; and Lin, Y. 2022. A timing engine inspired graph neural network model for pre-routing slack prediction. In *DAC*.

Hartigan, J. A.; and Wong, M. A. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*.

Hu, J.; Schaeffer, G.; and Garg, V. 2015. TAU 2015 contest on incremental timing analysis. In *ICCAD*.

Hu, J.; Sinha, D.; and Keller, I. 2014. TAU 2014 contest on removing common path pessimism during timing analysis. In *ISPD*.

Hu, W.; Liu, B.; Gomes, J.; Zitnik, M.; Liang, P.; Pande, V.; and Leskovec, J. 2020. Strategies for pre-training graph neural networks. In *ICLR*.

Karypis, G.; and Kumar, V. 1998. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*.

Khan, S.; Shi, Z.; Li, M.; and Xu, Q. 2023. DeepSeq: Deep Sequential Circuit Learning. *arXiv preprint arXiv:2302.13608*.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kipf, T. N.; and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

Lai, Y.; Liu, J.; Tang, Z.; Wang, B.; Hao, J.; and Luo, P. 2023. Chipformer: Transferable chip placement via offline decision transformer. In *ICML*.

Lai, Y.; Mu, Y.; and Luo, P. 2022. Maskplace: Fast chip placement via reinforced visual representation learning. *NeurIPS*.

Lasser, D. J. 1961. Topological ordering of a list of randomly-numbered elements of a network. *Communications of the ACM*.

Li, M.; Khan, S.; Shi, Z.; Wang, N.; Yu, H.; and Xu, Q. 2022. Deepgate: Learning neural representations of logic gates. In *DAC*.

Liao, P.; Liu, S.; Chen, Z.; Lv, W.; Lin, Y.; and Yu, B. 2022. DREAMPlace 4.0: Timing-driven global placement with momentum-based net weighting. In *DATE*.

Lin, Y.; Dhar, S.; Li, W.; Ren, H.; Khailany, B.; and Pan, D. Z. 2019. Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement. In *DAC*.

Lu, J.; Chen, P.; Chang, C.-C.; Sha, L.; Huang, D. J.-H.; Teng, C.-C.; and Cheng, C.-K. 2015. ePlace: Electrostatics-based placement using fast fourier transform and Nesterov's method. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*.

Mildenhall, B.; Srinivasan, P. P.; Tancik, M.; Barron, J. T.; Ramamoorthi, R.; and Ng, R. 2021. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*.

Mirhoseini, A.; Goldie, A.; Yazgan, M.; Jiang, J. W.; Songhori, E.; Wang, S.; Lee, Y.-J.; Johnson, E.; Pathak, O.; Nazi, A.; et al. 2021. A graph placement methodology for fast chip design. *Nature*.

Park, N.; and Kim, S. 2022. How do vision transformers work? In *ICLR*.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*.

Sanders, P.; and Schulz, C. 2013. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings 12*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *NeurIPS*.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; and Bengio, Y. 2018. Graph attention networks. In *ICLR*.

Wang, B.; Shen, G.; Li, D.; Hao, J.; Liu, W.; Huang, Y.; Wu, H.; Lin, Y.; Chen, G.; and Heng, P. A. 2022a. LHNN: Lattice hypergraph neural network for VLSI congestion prediction. In *DAC*.

Wang, M.; Zheng, D.; Ye, Z.; Gan, Q.; Li, M.; Song, X.; Zhou, J.; Ma, C.; Yu, L.; Gai, Y.; et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*.

Wang, P.; Zheng, W.; Chen, T.; and Wang, Z. 2022b. Anti-oversmoothing in deep vision transformers via the fourier domain analysis: From theory to practice. In *ICLR*.

Wu, Q.; Yang, C.; Zhao, W.; He, Y.; Wipf, D.; and Yan, J. 2023a. Difformer: Scalable (graph) transformers induced by energy constrained diffusion. In *ICLR*.

Wu, Q.; Zhao, W.; Li, Z.; Wipf, D. P.; and Yan, J. 2022. Nodeformer: A scalable graph structure learning transformer for node classification. In *NeurIPS*.

Wu, Q.; Zhao, W.; Yang, C.; Zhang, H.; Nie, F.; Jiang, H.; Bian, Y.; and Yan, J. 2023b. Simplifying and Empowering Transformers for Large-Graph Representations. In *NeurIPS*.

Xie, Z.; Huang, Y.-H.; Fang, G.-Q.; Ren, H.; Fang, S.-Y.; Chen, Y.; and Hu, J. 2018. RouteNet: Routability prediction for mixed-size designs using convolutional neural network. In *ICCAD*.

Xie, Z.; Liang, R.; Xu, X.; Hu, J.; Duan, Y.; and Chen, Y. 2021. Net2: A graph attention network method customized for pre-placement net length estimation. In *ASP-DAC*.

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2019. How powerful are graph neural networks? In *ICLR*.

Yang, T.; He, G.; and Cao, P. 2022. Pre-routing path delay estimation based on transformer and residual framework. In *ASP-DAC*.

Yang, Z.; Li, D.; Zhang, Y.; Zhang, Z.; Song, G.; Hao, J.; et al. 2022. Versatile Multi-stage Graph Neural Network for Circuit Representation. In *NeurIPS*.

Yuan, J.; Wang, P.; Ye, J.; Yuan, M.; Hao, J.; and Yan, J. 2023. EasySO: Exploration-enhanced Reinforcement Learning for Logic Synthesis Sequence Optimization and a Comprehensive RL Environment. In *ICCAD*.

Zhao, W.; Wu, Q.; Yang, C.; and Yan, J. 2023. GraphGLOW: Universal and Generalizable Structure Learning for Graph Neural Networks. In *SIGKDD*.

Zhou, X.; Ye, J.; Pui, C.-W.; Shao, K.; Zhang, G.; Wang, B.; Hao, J.; Chen, G.; and Heng, P. A. 2022. Heterogeneous Graph Neural Network-based Imitation Learning for Gate Sizing Acceleration. In *ICCAD*.

A Details about Datasets

Datasets we use contain 21 real-world OpenROAD (Ajayi et al. 2019) circuits. In our experiments, the datasets are split into training and testing sets in line with the same settings in (Guo et al. 2022). Details of dataset are shown in Table 6.

B Implementation Details

We implement our model and algorithm with PyTorch (Paszke et al. 2019) and Deep Graph Library (DGL) (Wang et al. 2019) framework. We train and test our model on a Linux server with one NVIDIA RTX 3090 GPU with 24 GB CUDA memory, 64 AMD Ryzen Threadripper 3970X 32-Core Processor at 2.20 GHz and 128 GB RAM. We use Adam (Kingma and Ba 2014) optimizer with learning rate $lr = 5 \times 10^{-4}, \beta_1 = 0.9, \beta_2 = 0.999$. For all models, we train them with 20000 epochs.

C Computation of Slack

In circuit, electrical signal travels from primary inputs (input ports) to endpoints or primary outputs (output ports) along timing paths. Since signal cannot travel with an infinite speed, there are delay in circuit, and the instant when signal arrives at a pin is defined as *Arrival Time (AT)*. Considering the timing constraints in circuits, e.g. the instant when signal arrives a pin cannot be too late, there are also *Required Arrival Time (RAT)* for each pin, which means that the instant when signal arrives at this pin cannot be later than RAT.

Based on AT and RAT, the slack is defined as the difference between RAT and AT as follows:

$$\begin{aligned} slack^E &= AT^E - RAT^E \\ slack^L &= RAT^L - AT^L, \end{aligned} \tag{9}$$

where E, L means early and late respectively, and a positive slack means that the timing constraint is satisfied.

	Dataset	# Node	# Edge			# Level
			cell	net	net_inv	
train	BM64	38458	25334	27843	27843	100
	aes128	211045	138457	148997	148997	126
	aes256	290955	189262	207414	207414	124
	aes_cipher	59777	41411	42671	42671	70
	blabla	55568	35689	39853	39853	154
	cic_decimator	3131	2102	2232	2232	40
	des	60541	41845	44478	44478	48
	genericfir	38827	25013	28845	28845	32
	picorv32a	58676	40208	43047	43047	168
	salsa20	78486	52895	57737	57737	160
	usb	3361	2189	2406	2406	34
	usb_cdc_core	7406	4869	5200	5200	74
	wbqspiflash	9672	6454	6798	6798	112
	zipdiv	4398	2913	3102	3102	72
test	aes192	234211	152910	165350	165350	134
	jpeg_encoder	238216	167960	176737	176737	124
	spm	1121	700	765	765	18
	synth_ram	25910	16782	19024	19024	24
	usbf_device	66345	42226	46241	46241	110
	xtea	10213	6882	7151	7151	114
	y_huff	48216	30612	33689	33689	58

Table 6: Dataset information: number of nodes, edges and levels for datasets.

	TimingGCN	k_{uf}	k_f	k_f/k_{uf}	R^2_{uf}	R^2_f
train	BM64	4.322	11.769	**2.723**	0.695	**0.915**
	aes128	3.867	23.190	**5.997**	-0.201	**0.957**
	aes256	6.753	29.780	**4.410**	0.649	**0.966**
	aes_cipher	5.877	36.984	**6.293**	0.614	**0.973**
	blabla	64.797	69.614	**1.074**	0.980	**0.986**
	cic_decimator	56.084	81.282	**1.449**	0.979	**0.988**
	des	28.218	98.766	**3.500**	0.894	**0.990**
	genericfir	1.814	7.154	**3.944**	0.142	**0.860**
	picorv32a	5.860	16.753	**2.859**	0.811	**0.940**
	salsa20	6.989	12.208	**1.747**	0.648	**0.918**
	usb	76.626	62.343	0.814	0.971	**0.984**
	usb_cdc_core	111.328	39.247	0.353	0.971	**0.975**
	wbqspiflash	52.909	50.267	0.950	0.972	**0.980**
	zipdiv	92.350	21.805	0.236	**0.958**	0.954
test	aes192	4.798	25.162	**5.244**	0.649	**0.960**
	jpeg_encoder	1.717	10.627	**6.190**	0.271	**0.906**
	spm	4.931	4.189	0.850	0.696	**0.761**
	synth_ram	7.898	5.301	0.671	0.763	**0.811**
	usbf_device	2.723	12.740	**4.678**	0.317	**0.922**
	xtea	11.444	4.211	0.368	**0.781**	0.763
	y_huff	6.311	17.172	**2.721**	0.552	**0.942**
	Avg. train	36.985	40.083	**2.596**	0.720	**0.956**
	Avg. test	5.689	11.343	**2.960**	0.576	**0.866**

Table 7: R^2 on both original and flatten data in SOTA ML-based method TimingGCN (Guo et al. 2022). R^2_f on flatten data ($\mathbb{R}^{4n}$) is much higher than R^2_{uf} on original data ($\mathbb{R}^{n\times 4}$). As a result, R^2_f cannot reflect the real performance.

D Experiments

D.1 Flatten R^2_f and Un-Flatten R^2_{uf}

Under R^2_f, previous method achieves 0.866. However, under R^2_{uf}, it only achieves 0.576 as shown in Table 7. With $k_f/k_{uf} > 1$ in Table 7 we can see that, after flattening the multi-channel slack, VAR increases faster and higher than MSE, leading to the improvement about 0.3 of R^2.

D.2 Experiments on Other Metrics

For fairness and integrity, we also evaluate baselines and our PreRoutGNN with flatten R^2_f, which shown in Table 8 and Table 9. Our approach still outperforms other baselines. However, we want to emphasize that compared to flatten R^2_f, un-flatten R^2_{uf} is a more appropriate evaluation metric.

We also evaluate different methods with Mean Squared Error (MSE) shown in Table 10.

D.3 Slack Prediction on Endpoints

We draw slack ground truth and prediction on all timing endpoints of testing set in Fig. 6, which shows a strong linear correlation between prediction and ground truth.

D.4 Time Cost for Graph Partition and Inference

We also evaluate the time cost for graph partition and inference shown in Fig. 7. For large-scale circuit with about 300,000 pins, the inference can be finished within one second.

	Dataset	flatten R_f^2 (↑) GCNII	GAT	GINE	TimingGCN	PreRoutGNN
train	BM64	0.941 ± 0.030	0.815 ± 0.080	0.898 ± 0.052	0.864 ± 0.131	**0.997** ± **0.001**
	aes128	0.923 ± 0.035	0.852 ± 0.055	0.895 ± 0.031	0.889 ± 0.060	**0.993** ± **0.001**
	aes256	0.936 ± 0.029	0.855 ± 0.057	0.905 ± 0.031	0.934 ± 0.021	**0.997** ± **0.002**
	aes_cipher	0.908 ± 0.038	0.840 ± 0.060	0.849 ± 0.064	0.971 ± 0.017	**0.998** ± **0.000**
	blabla	0.946 ± 0.032	0.851 ± 0.066	0.877 ± 0.058	0.970 ± 0.010	**0.998** ± **0.001**
	cic_decimator	0.861 ± 0.055	0.780 ± 0.102	0.770 ± 0.067	0.973 ± 0.012	**0.998** ± **0.000**
	des	0.978 ± 0.019	0.872 ± 0.071	0.941 ± 0.067	0.990 ± 0.004	**0.999** ± **0.000**
	genericfir	0.941 ± 0.028	0.700 ± 0.154	0.910 ± 0.042	0.846 ± 0.025	**0.979** ± **0.001**
	picorv32a	0.916 ± 0.037	0.822 ± 0.070	0.850 ± 0.052	0.950 ± 0.007	**0.996** ± **0.001**
	salsa20	0.941 ± 0.026	0.828 ± 0.076	0.896 ± 0.037	0.941 ± 0.031	**0.998** ± **0.001**
	usb	0.865 ± 0.045	0.832 ± 0.066	0.780 ± 0.062	0.954 ± 0.026	**0.998** ± **0.000**
	usb_cdc_core	0.850 ± 0.063	0.690 ± 0.135	0.698 ± 0.106	0.967 ± 0.007	**0.998** ± **0.001**
	wbqspiflash	0.872 ± 0.052	0.789 ± 0.091	0.711 ± 0.107	0.967 ± 0.016	**0.996** ± **0.002**
	zipdiv	0.898 ± 0.035	0.863 ± 0.057	0.715 ± 0.151	0.971 ± 0.015	**0.998** ± **0.001**
test	aes192	0.876 ± 0.027	0.771 ± 0.036	0.867 ± 0.021	0.923 ± 0.026	**0.996** ± **0.000**
	jpeg_encoder	0.829 ± 0.019	0.432 ± 0.132	0.778 ± 0.058	0.938 ± 0.019	**0.988** ± **0.003**
	spm	0.002 ± 0.266	-0.168 ± 0.110	0.259 ± 0.152	0.848 ± 0.066	**0.989** ± **0.005**
	synth_ram	-0.426 ± 0.025	-0.560 ± 0.018	-0.499 ± 0.122	0.798 ± 0.057	**0.984** ± **0.006**
	usbf_device	0.706 ± 0.019	0.549 ± 0.012	0.731 ± 0.039	0.934 ± 0.018	**0.994** ± **0.000**
	xtea	0.293 ± 0.044	0.199 ± 0.047	0.238 ± 0.100	0.886 ± 0.072	**0.991** ± **0.001**
	y_huff	0.518 ± 0.231	0.451 ± 0.045	0.294 ± 0.380	0.914 ± 0.028	**0.975** ± **0.011**
Avg. train		0.913	0.813	0.835	0.942	**0.996**
Avg. test		0.400	0.239	0.381	0.891	**0.988**

Table 8: Slack prediction with flatten R_f^2 as metrics. Experiments are implemented on 5 different seeds.

	Dataset	MSE slew (↓)		MSE net delay (↓)		MSE cell delay (↓)		flatten R_f^2 slew (↑)		flatten R_f^2 net delay (↑)		flatten R_f^2 cell delay (↑)	
		TimingGCN	PreRoutGNN	TimingGCN	PreRoutGNN	TimingGCN	PreRoutGNN	TimingGCN	PreRoutGNN	TimingGCN	PreRoutGNN	TimingGCN	PreRoutGNN
train	BM64	0.032 ± 0.013	**0.006** ± **0.002**	0.045 ± 0.022	**0.022** ± **0.003**	0.002 ± 0.001	**0.000** ± **0.000**	0.960 ± 0.016	**0.993** ± **0.002**	0.984 ± 0.008	**0.992** ± **0.001**	0.955 ± 0.030	**0.995** ± **0.002**
	aes128	0.052 ± 0.018	**0.005** ± **0.001**	0.045 ± 0.009	**0.021** ± **0.001**	0.002 ± 0.001	**0.000** ± **0.000**	0.909 ± 0.032	**0.991** ± **0.002**	0.972 ± 0.006	**0.987** ± **0.001**	0.949 ± 0.022	**0.996** ± **0.001**
	aes256	0.050 ± 0.015	**0.006** ± **0.001**	0.047 ± 0.011	**0.023** ± **0.002**	0.003 ± 0.002	**0.000** ± **0.000**	0.916 ± 0.025	**0.990** ± **0.002**	0.974 ± 0.006	**0.988** ± **0.001**	0.945 ± 0.029	**0.996** ± **0.001**
	aes_cipher	0.048 ± 0.014	**0.008** ± **0.000**	0.061 ± 0.010	**0.036** ± **0.003**	0.002 ± 0.001	**0.000** ± **0.000**	0.922 ± 0.023	**0.987** ± **0.000**	0.971 ± 0.005	**0.983** ± **0.002**	0.970 ± 0.011	**0.994** ± **0.000**
	blabla	0.059 ± 0.024	**0.011** ± **0.003**	0.064 ± 0.029	**0.029** ± **0.003**	0.005 ± 0.003	**0.001** ± **0.000**	0.935 ± 0.027	**0.988** ± **0.003**	0.987 ± 0.006	**0.994** ± **0.001**	0.907 ± 0.057	**0.987** ± **0.003**
	cic_decimator	0.028 ± 0.012	**0.002** ± **0.001**	0.016 ± 0.011	**0.004** ± **0.000**	0.001 ± 0.000	**0.000** ± **0.000**	0.955 ± 0.019	**0.997** ± **0.002**	0.987 ± 0.009	**0.997** ± **0.000**	0.976 ± 0.010	**0.996** ± **0.001**
	des	0.058 ± 0.024	**0.005** ± **0.001**	0.051 ± 0.011	**0.023** ± **0.001**	0.003 ± 0.002	**0.000** ± **0.000**	0.919 ± 0.034	**0.993** ± **0.002**	0.976 ± 0.005	**0.989** ± **0.001**	0.978 ± 0.015	**0.998** ± **0.001**
	genericfir	0.018 ± 0.006	**0.004** ± **0.000**	0.029 ± 0.007	**0.018** ± **0.002**	0.001 ± 0.001	**0.000** ± **0.000**	0.966 ± 0.012	**0.991** ± **0.000**	0.976 ± 0.006	**0.986** ± **0.002**	0.960 ± 0.022	**0.998** ± **0.001**
	picorv32a	0.093 ± 0.014	**0.062** ± **0.002**	0.051 ± 0.015	**0.025** ± **0.003**	0.002 ± 0.001	**0.000** ± **0.000**	0.879 ± 0.019	**0.920** ± **0.003**	0.982 ± 0.005	**0.991** ± **0.001**	0.964 ± 0.014	**0.993** ± **0.002**
	salsa20	0.043 ± 0.012	**0.007** ± **0.001**	0.054 ± 0.013	**0.030** ± **0.002**	0.003 ± 0.002	**0.000** ± **0.000**	0.936 ± 0.018	**0.989** ± **0.002**	0.983 ± 0.004	**0.991** ± **0.001**	0.938 ± 0.028	**0.992** ± **0.002**
	usb	0.033 ± 0.016	**0.002** ± **0.001**	0.020 ± 0.014	**0.004** ± **0.000**	0.001 ± 0.001	**0.000** ± **0.000**	0.956 ± 0.022	**0.997** ± **0.001**	0.991 ± 0.007	**0.998** ± **0.000**	0.979 ± 0.013	**0.997** ± **0.001**
	usb_cdc_core	0.070 ± 0.011	**0.045** ± **0.001**	0.022 ± 0.010	**0.008** ± **0.001**	0.001 ± 0.001	**0.000** ± **0.000**	0.882 ± 0.018	**0.925** ± **0.001**	0.982 ± 0.009	**0.993** ± **0.000**	0.977 ± 0.011	**0.996** ± **0.001**
	wbqspiflash	0.043 ± 0.014	**0.012** ± **0.002**	0.030 ± 0.018	**0.009** ± **0.001**	0.002 ± 0.001	**0.000** ± **0.000**	0.937 ± 0.020	**0.982** ± **0.002**	0.990 ± 0.006	**0.997** ± **0.000**	0.970 ± 0.009	**0.994** ± **0.001**
	zipdiv	0.045 ± 0.012	**0.020** ± **0.001**	0.024 ± 0.019	**0.004** ± **0.001**	0.001 ± 0.000	**0.000** ± **0.000**	0.950 ± 0.013	**0.978** ± **0.001**	0.989 ± 0.009	**0.998** ± **0.000**	0.978 ± 0.008	**0.997** ± **0.001**
test	aes192	0.062 ± 0.012	**0.011** ± **0.002**	0.081 ± 0.018	**0.037** ± **0.002**	0.003 ± 0.002	**0.000** ± **0.000**	0.896 ± 0.021	**0.981** ± **0.003**	0.954 ± 0.010	**0.979** ± **0.001**	0.936 ± 0.031	**0.992** ± **0.001**
	jpeg_encoder	0.071 ± 0.013	**0.017** ± **0.000**	0.097 ± 0.011	**0.053** ± **0.002**	0.004 ± 0.001	**0.001** ± **0.000**	0.882 ± 0.022	**0.972** ± **0.000**	0.966 ± 0.004	**0.981** ± **0.001**	0.924 ± 0.023	**0.987** ± **0.001**
	spm	0.036 ± 0.007	**0.007** ± **0.000**	0.066 ± 0.008	**0.049** ± **0.016**	0.002 ± 0.000	**0.000** ± **0.000**	0.945 ± 0.010	**0.989** ± **0.001**	0.921 ± 0.010	**0.941** ± **0.019**	0.946 ± 0.008	**0.992** ± **0.000**
	synth_ram	0.155 ± 0.028	**0.102** ± **0.047**	0.268 ± 0.052	**0.099** ± **0.004**	0.012 ± 0.008	**0.010** ± **0.005**	0.827 ± 0.031	**0.887** ± **0.052**	0.929 ± 0.014	**0.974** ± **0.001**	0.881 ± 0.076	**0.899** ± **0.047**
	usbf_device	0.059 ± 0.010	**0.031** ± **0.002**	0.055 ± 0.002	**0.052** ± **0.006**	0.004 ± 0.002	**0.001** ± **0.000**	0.910 ± 0.015	**0.952** ± **0.003**	0.968 ± 0.001	**0.970** ± **0.003**	0.927 ± 0.031	**0.990** ± **0.002**
	xtea	0.085 ± 0.035	**0.011** ± **0.001**	0.095 ± 0.019	**0.064** ± **0.000**	0.004 ± 0.002	**0.001** ± **0.000**	0.887 ± 0.046	**0.986** ± **0.001**	0.944 ± 0.011	**0.963** ± **0.000**	0.922 ± 0.043	**0.989** ± **0.003**
	y_huff	0.076 ± 0.016	**0.066** ± **0.000**	0.211 ± 0.049	**0.124** ± **0.016**	**0.003** ± **0.001**	0.005 ± 0.003	0.915 ± 0.018	**0.926** ± **0.000**	0.940 ± 0.014	**0.964** ± **0.005**	**0.953** ± **0.017**	0.923 ± 0.042
Avg. train		0.048	**0.014**	0.040	**0.018**	0.002	**0.000**	0.930	**0.980**	0.982	**0.992**	0.960	**0.995**
Avg. test		0.078	**0.035**	0.125	**0.068**	0.005	**0.003**	0.894	**0.956**	0.946	**0.967**	0.927	**0.968**

Table 9: Comparison of other tasks, including prediction of slew, net delay and cell delay. We evaluate baseline and our PreRoutGNN with metric flatten R_f^2 and MSE. Experiments are implemented on five different random seeds.

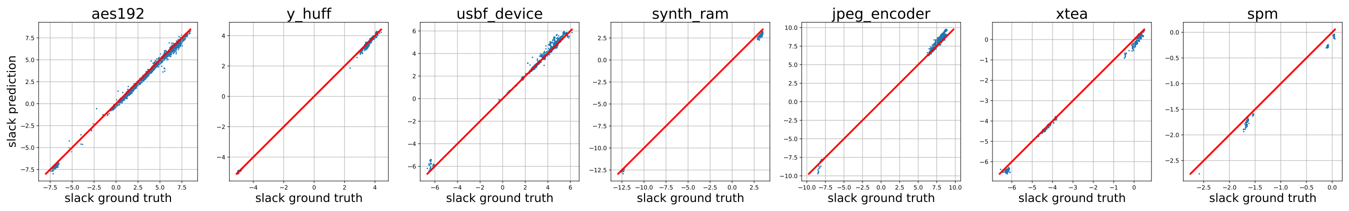


Figure 6: Slack ground truth and prediction on timing path endpoints, showing a strong linear correlation (red line is $y = x$).

E Why Other Graph Partition Algorithms Inappropriate?

As we have discussed, the AT of current node relies on *all* its predecessors. We can and only can calculate its AT after *all* ATs of its predecessors have been calculated. We apply topological sorting to depict this dependency.

In our graph partition algorithm, we add nodes belong-

	Dataset	MSE (↓)					R_{uf}^2 (↑)				
		GCNII	GAT	GINE	TimingGCN	PreRoutGNN	GCNII	GAT	GINE	TimingGCN	PreRoutGNN
train	BM64	1.728 ± 0.881	5.442 ± 2.342	3.002 ± 1.544	3.997 ± 3.848	**0.076 ± 0.038**	0.763 ± 0.101	0.354 ± 0.223	0.623 ± 0.161	0.668 ± 0.252	**0.986 ± 0.006**
	aes128	3.195 ± 1.426	6.096 ± 2.282	4.346 ± 1.279	4.581 ± 2.470	**0.307 ± 0.024**	0.458 ± 0.207	-0.245 ± 0.340	0.029 ± 0.352	-0.055 ± 0.312	**0.822 ± 0.008**
	aes256	3.119 ± 1.408	7.078 ± 2.778	4.617 ± 1.489	3.238 ± 1.037	**0.169 ± 0.079**	0.581 ± 0.151	-0.097 ± 0.306	0.067 ± 0.311	0.499 ± 0.102	**0.937 ± 0.023**
	aes_cipher	1.391 ± 0.569	2.408 ± 0.899	2.269 ± 0.958	0.429 ± 0.255	**0.034 ± 0.007**	0.020 ± 0.447	-0.438 ± 0.427	-0.162 ± 0.242	0.644 ± 0.123	**0.967 ± 0.000**
	blabla	6.123 ± 3.651	17.083 ± 7.588	14.044 ± 6.629	3.475 ± 1.125	**0.233 ± 0.076**	0.889 ± 0.073	0.679 ± 0.131	0.737 ± 0.122	0.958 ± 0.013	**0.997 ± 0.001**
	cic_decimator	0.468 ± 0.184	0.738 ± 0.342	0.770 ± 0.225	0.091 ± 0.040	**0.006 ± 0.000**	0.719 ± 0.112	0.597 ± 0.163	0.589 ± 0.107	0.963 ± 0.014	**0.996 ± 0.000**
	des	1.866 ± 1.662	11.066 ± 6.096	5.129 ± 5.805	0.856 ± 0.355	**0.055 ± 0.004**	0.924 ± 0.056	0.566 ± 0.225	0.776 ± 0.220	0.905 ± 0.058	**0.993 ± 0.001**
	genericfir	0.234 ± 0.112	1.190 ± 0.613	0.359 ± 0.168	0.612 ± 0.099	**0.085 ± 0.005**	0.710 ± 0.122	-0.436 ± 0.733	0.503 ± 0.219	0.222 ± 0.123	**0.871 ± 0.013**
	picorv32a	2.346 ± 1.033	4.952 ± 1.957	4.167 ± 1.440	1.386 ± 0.189	**0.109 ± 0.028**	0.672 ± 0.096	0.239 ± 0.240	0.425 ± 0.169	0.839 ± 0.045	**0.974 ± 0.007**
	salsa20	3.991 ± 1.751	11.635 ± 5.117	7.024 ± 2.475	4.005 ± 2.107	**0.160 ± 0.075**	0.814 ± 0.066	0.530 ± 0.170	0.717 ± 0.073	0.792 ± 0.118	**0.994 ± 0.002**
	usb	0.580 ± 0.192	0.725 ± 0.285	0.947 ± 0.267	0.197 ± 0.112	**0.008 ± 0.001**	0.747 ± 0.088	0.678 ± 0.115	0.607 ± 0.118	0.934 ± 0.035	**0.996 ± 0.000**
	usb_cdc_core	0.830 ± 0.348	1.713 ± 0.745	1.673 ± 0.588	0.185 ± 0.041	**0.013 ± 0.003**	0.793 ± 0.095	0.502 ± 0.212	0.563 ± 0.177	0.961 ± 0.008	**0.997 ± 0.001**
	wbqspiflash	1.321 ± 0.532	2.181 ± 0.936	2.988 ± 1.106	0.343 ± 0.169	**0.039 ± 0.021**	0.742 ± 0.091	0.613 ± 0.131	0.522 ± 0.136	0.956 ± 0.013	**0.994 ± 0.002**
	zipdiv	1.307 ± 0.442	1.750 ± 0.725	3.645 ± 1.939	0.370 ± 0.187	**0.027 ± 0.014**	0.822 ± 0.070	0.703 ± 0.129	0.593 ± 0.188	0.971 ± 0.011	**0.998 ± 0.001**
test	aes192	6.429 ± 1.407	11.918 ± 1.889	6.934 ± 1.105	4.000 ± 1.328	**0.228 ± 0.015**	-0.017 ± 0.188	-0.727 ± 0.173	-0.048 ± 0.352	0.384 ± 0.205	**0.937 ± 0.009**
	jpeg_encoder	10.893 ± 1.237	36.278 ± 8.409	14.163 ± 3.730	3.950 ± 1.208	**0.747 ± 0.209**	0.056 ± 0.080	-2.859 ± 0.667	-0.285 ± 0.198	0.478 ± 0.131	**0.764 ± 0.010**
	spm	1.444 ± 0.384	1.690 ± 0.159	1.071 ± 0.220	0.220 ± 0.096	**0.016 ± 0.007**	-0.247 ± 0.309	-0.474 ± 0.134	0.040 ± 0.161	0.802 ± 0.083	**0.985 ± 0.006**
	synth_ram	65.672 ± 1.137	71.823 ± 0.828	69.009 ± 5.632	9.302 ± 2.617	**0.753 ± 0.260**	-0.913 ± 0.031	-1.078 ± 0.025	-1.007 ± 0.162	0.735 ± 0.079	**0.977 ± 0.008**
	usbf_device	5.545 ± 0.365	8.497 ± 0.226	5.062 ± 0.743	1.249 ± 0.339	**0.117 ± 0.001**	-0.215 ± 0.065	-0.946 ± 0.146	-0.133 ± 0.094	0.347 ± 0.119	**0.949 ± 0.002**
	xtea	22.387 ± 1.400	25.368 ± 1.489	24.132 ± 3.173	3.613 ± 2.270	**0.285 ± 0.041**	0.210 ± 0.044	-0.055 ± 0.091	0.099 ± 0.181	0.888 ± 0.062	**0.985 ± 0.000**
	y_huff	4.259 ± 2.036	4.849 ± 0.394	6.235 ± 3.354	0.764 ± 0.245	**0.225 ± 0.099**	-0.235 ± 0.450	-0.481 ± 0.114	-1.034 ± 0.999	0.490 ± 0.146	**0.931 ± 0.029**
	Avg. train	2.036	5.290	3.927	1.698	**0.095**	0.689	0.303	0.471	0.733	**0.966**
	Avg. test	16.661	22.917	18.087	3.300	**0.339**	-0.195	-0.946	-0.338	0.589	**0.933**

Table 10: Slack prediction with MSE and R_{uf}^2 as primary metrics. Experiments are implemented on 5 different seeds.

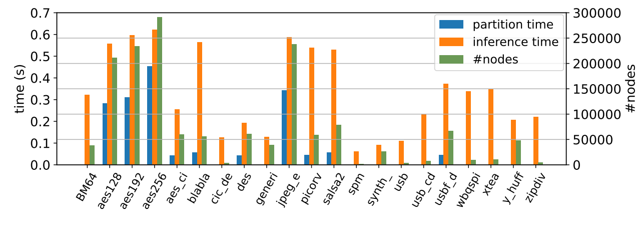


Figure 7: Partition time with maximum sub-graph size 50000. If the size of original graph is less than maximum sub-graph size, we will not partition this circuit.

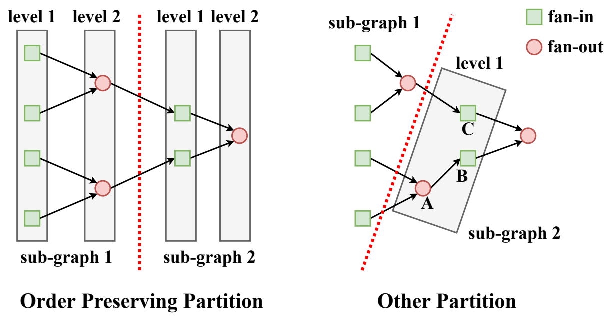


Figure 8: Our order preserving partition algorithm, ensuring that after partitioning, nodes belonging to the same topological sorting order serve the same logic role (all of them are fan-in or fan-out), and they are still in the same level. As a result, our graph partition algorithm can preserve the sorting order.

ing to the same topological sorting order into a level, and construct sub-graph based on continuous levels. If we apply topological sorting in the sub-graph, the relative sorting orders will be the same in the original whole graph, which preserve the topological dependency in timing path.

Traditional graph clustering methods (Hartigan and Wong 1979) merge nodes into a cluster, leading to the loss of fine-grained connections. In this task, we need to calculate AT of each node. As a result, it is inappropriate in this settings. Other partitioning methods, such as KaHIP (Sanders and Schulz 2013) and METIS (Karypis and Kumar 1998) are based on the optimizing for cut- or connectivity-metric, mainly focus on neighbours structure, while in circuit, deep timing paths play a critical role. Furthermore, in topological propagation layer, the relative topological sorting orders will be different from that in the original whole graph, which disrupts the topological sorting level information in circuit graph, even makes propagation unable to implement. For example, in original whole graph and our partitioned sub-graphs, nodes belonging to the same order serve as the same logic role, which means they are all fan-in or fan-out. However, if we apply topological sorting on sub-graphs partitioned by other algorithms, nodes belonging to the same order possibly do not serve the same logic role.

We also illustrate it in Fig 8. Green rectangle means fan-in, which is input pin of cell. Red circle means fan-out, which is output pin of cell. Under usual circumstances, a cell consists with two input pins and a single output pin. The dotted line splits the whole graph into two sub-graphs. Left is our order preserving partition algorithm, ensuring that after partitioning, nodes belonging to the same topological sorting order still serve as the same logic role (all of them are fan-in or fan-out), and they are still in the same level. However, in the right part, results through other partition algorithms, possibly are faced with such situation. After applying topological sorting on the sub-graph, nodes belonging to the same topological order do not always serve as the same logic role. For example, in sub-graph 2 of other graph partition, A has the same topological sorting order with B and C, but A serves as a fan-out, which is the output pin of a cell, while B and C serve as fan-in, which are the input pin of another cell. This makes topological propagation unable to implement. As a result, our graph partition algorithm can preserve the sorting order.

F Model Structure

F.1 Global Pre-training Circuit Graph Auto-Encoder

The forward propagation of our GNN layer is shown as follows:

$$\begin{aligned}
\mathbf{m}_{ji} &= \mathrm{MLP}(\mathbf{f}_j || \mathbf{f}_i || \mathbf{e}_{ji}) + \mathbf{f}_j \\
\mathbf{a}_i &= \frac{\sum_{j\in\mathcal{N}(i)} \mathbf{m}_{ji}}{\mathrm{size}(\mathcal{N}(i))} \\
\mathbf{b}_i &= \max_{j\in\mathcal{N}(i)} \mathbf{m}_{ji} \\
\mathbf{c}_i &= \min_{j\in\mathcal{N}(i)} \mathbf{m}_{ji} \\
\mathbf{d}_i &= \sum_{j\in\mathcal{N}(i)} \mathbf{m}_{ji} \\
\mathbf{f}_i &= \mathrm{MLP}(\mathbf{f}_i || \mathbf{a}_i || \mathbf{b}_i || \mathbf{c}_i || \mathbf{d}_i) + \mathbf{f}_i,
\end{aligned} \tag{10}$$

where $\mathbf{f}_i$ is the node feature for node $i, \mathcal{N}(i)$ is the predecessors of i, $\mathbf{e}_{ji}$ is the edge feature for edge (j, i). We select 4 as number of encoder layer, 4 as number of decoder layer, 32 as hidden dimension and 4 as dimension of latent space. Both MLPs are 2 layers with 32 as hidden dimension. In MLP, we use leaky ReLU with negative slope 0.2 as activation.

F.2 RLL-based GCN Layer

We stack 4 layers of RLL-based GCN layers. Each MLP in it is 2 layers with 64 as hidden dimension. In MLP, we use leaky ReLU with negative slope 0.2 as activation.

We update node features in RLL-based GCN layer for net connection (edge type is 'net' or 'net_inv') as follows:

$$\begin{aligned}
\mathbf{m}_{ji} &= \mathrm{MLP}(\mathbf{f}_j || \mathbf{f}_i || \mathbf{e}_{ji}) + \mathbf{f}_j \\
\mathbf{a}_i &= \frac{\sum_{j\in\mathcal{N}(i)} \mathbf{m}_{ji}}{\mathrm{size}(\mathcal{N}(i))}, \mathbf{b}_i = \max_{j\in\mathcal{N}(i)} \mathbf{m}_{ji} \\
\mathbf{f}_i &= \mathrm{MLP}(\mathbf{f}_i || \mathbf{a}_i || \mathbf{b}_i) + \mathbf{f}_i,
\end{aligned} \tag{11}$$

where $\mathbf{f}_i$ is the node feature for node $i, \mathcal{N}(i)$ is the predecessors of i, $\mathbf{e}_{ji}$ is the edge feature for edge (j, i).

F.3 Baselines

For GCNII (Chen et al. 2020), we stack 8 layers with hidden dimension 64. We use leaky ReLU with negative slope 0.2 as activation.

For GAT (Vaswani et al. 2017), we stack 8 layers with hidden dimension 64. We use leaky ReLU with negative slope 0.2 as activation.

For GINE (Hu et al. 2020), we stack 8 layers with node feature hidden dimension 64, edge feature hidden dimension 64. We use leaky ReLU with negative slope 0.2 as activation.

For TimingGCN, we use the same model structure in line with (Guo et al. 2022).