Heterogeneous Graph Neural Network-based Imitation Learning for Gate Sizing Acceleration

Xinyi Zhou
The Chinese University of Hong Kong
Hong Kong, China
1155165857@link.cuhk.edu.hk

Junjie Ye*
Huawei Noah's Ark Lab
Shenzhen, China
yejunjie4@huawei.com

Chak-Wa Pui
Huawei Noah's Ark Lab
Shenzhen, China
puichakwa@huawei.com

Kun Shao
Huawei Noah's Ark Lab
Beijing, China
shaokun2@huawei.com

Guangliang Zhang
HiSilicon
Shenzhen, China
zhangguangliang@hisilicon.com

Bin Wang
Huawei Noah's Ark Lab
Beijing, China
wangbin158@huawei.com

Jianye Hao
Huawei Noah's Ark Lab
Beijing, China
Tianjin University
Tianjin, China
haojianye@huawei.com

Guangyong Chen*
Zhejiang Lab
Hangzhou, China
gychen@zhejianglab.com

Pheng Ann Heng
The Chinese University of Hong Kong
Hong Kong, China
pheng@cse.cuhk.edu.hk

ABSTRACT

Gate Sizing is an important step in logic synthesis, where the cells are resized to optimize metrics such as area, timing, power, leakage, etc. In this work, we consider the gate sizing problem for leakage power optimization with timing constraints. Lagrangian Relaxation is a widely employed optimization method for gate sizing problems. We accelerate Lagrangian Relaxation-based algorithms by narrowing down the range of cells to resize. In particular, we formulate a heterogeneous directed graph to represent the timing graph, propose a heterogeneous graph neural network as the encoder, and train in the way of imitation learning to mimic the selection behavior of each iteration in Lagrangian Relaxation. This network is used to predict the set of cells that need to be changed during the optimization process of Lagrangian Relaxation. Experiments show that our accelerated gate sizer could achieve comparable performance to the baseline with an average of 22.5% runtime reduction.

1 INTRODUCTION

As technologies evolve, constraints such as design rules, power, routability, heat disputation are becoming much more difficult to solve during the physical design stage. "Shift-left" suggests that circuit constraints and performance should be considered in earlier stages of circuit design [1]. Logic synthesis, which converts an abstract specification of desired circuit behavior into a netlist of logic gates, is proven to be an important step to address congestion [2], aging [3], power [4] in the "shift left" paradigm.

In ASIC design flow, the netlist generated by logic synthesis consists of logic gates such as combinatorial cells, registers, memories, IP cores, etc. Since each logic gate has several implementations corresponding to different characteristics (such as area, leakage power, timing, etc), the type selection of each logic gate will greatly affect the power, performance, and area (PPA) of the later stages. Besides the technology mapping step, gate sizing is another important step to optimize the type of each logic gate. Unlike technology mapping where the netlist is still under conversion, gate sizing operates in a completely converted netlist, which ensures that the optimized PPA has a higher correlation to the actual one.

Due to the discrete gate versions, the gate sizing problem is an NP-hard combinatorial optimization problem [5]. In the past few decades, methods such as approximate algorithms, heuristics have been studied extensively in academics to avoid exploring the exponential solution space. Existing traditional algorithms for gate sizing problems are based on various techniques, including geometric programming (GP) [6], dynamic programming (DP) [7], greedy heuristics [8] and Lagrangian relaxation (LR) [9–11]. Among these methods, LR-based algorithms show superior performance as they conduct global optimization and prune the search space by Karush-Kuhn-Tucker(KKT) conditions. In the classic and pioneer work [9], gate version and threshold voltage are assigned to each gate sequentially to minimize leakage power under the given timing closure. The problem is formulated as an optimization problem solved by LR. Due to the long runtime for global evaluation, local lambda delay is used as the cost for gate type selection. Various works based on the same framework are proposed later with improvement on converging speed [10], multi-thread enablement [11], asynchronous circuit support [12], etc. However, scanning the entire circuit in

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549361

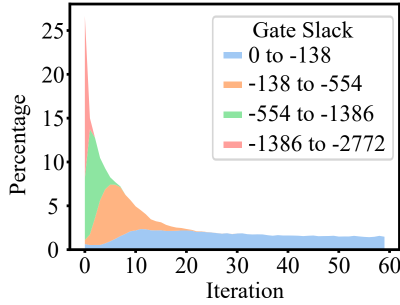


Figure 1: The percentage of gates changed by LR on the benchmark des_perf (fast) in ISPD 2013 Contest.

each LR iteration makes it slow for large designs, especially when more and more types are provided in the cell library for the same functionality, resulting in even longer time for LR-based methods to converge.

In recent years, machine learning (ML) has been applied to either improving or completely substituting the traditional methods in EDA, such as congestion estimation [13], detail routing [14], placement [15], testing [16], design for manufacture [17], etc. In particular, a recent work [18] proposed a novel end-to-end reinforcement learning (RL) framework to solve the gate sizing problem that focuses on the post-route stage. It uses a graph neural network (GNN) to encode the features of a gate as states and considers the total negative slack (TNS) gain as the reward of changing its type.

Our method aims to speed up LR-based algorithms and is motivated by an empirical observation that only a small percentage of gates are changed by LR in each iteration. See Figure 1 for an example. The total number of changed gates declines quickly and falls below 5% after 10 iterations, which implies considerable room for runtime improvement by filtering out gates that are unlikely to be changed when scanning. This motivates us to speed up LR-based methods by narrowing down the candidate set for resizing, which works in an orthogonal direction to the previous works [10, 11, 19]. In particular, we present a heterogeneous graph neural network (HGNN)-based imitation learning [20] method to accelerate LR-based gate sizing algorithms. We use the HGNN to encode the circuit state and predict the cells to be changed for each iteration by mimicking the LR-based method. We evaluate our method by comparing it with the open-source method proposed in [9]. Our contributions can be summarized as follows.

- A novel heterogeneous directed graph is used to represent the timing graph, which includes timing arcs, nets, pins, and cells. A heterogeneous graph neural network is proposed to encode the state of the circuit and each gate.
- We treat the LR-based algorithm as the expert to demonstrate the desired behavior and train our model with imitation learning. The gates that are unlikely to be changed by the LR-based method are filtered accurately in each iteration.
- Experimental results verify that our method can reduce the runtime while preserving the effectiveness and generalization of the original algorithm. With the network only trained on 4 small benchmarks of the ISPD 2013 benchmarks [21], it is capable to transfer to unseen circuit graphs with different topologies and larger scales.

The rest of the paper is organized as follows. In Section 2, preliminaries about the baseline algorithm, HGNN, and imitation learning are introduced. Section 3 presents our proposed HGNN model and the integrated sizer. Section 4 reports our experimental results. We finally conclude the paper in Section 5.

2 PRELIMINARIES

2.1 Lagrangian Relaxation-based Gate Sizing

Given a netlist and the timing closure, the gate sizing problem is to determine the cell type of each logic gate such that the target metrics are optimized. As mentioned in Section 1, Lagrangian Relaxation is widely used in gate sizing. In this section, we will use the implementation in [9] to illustrate the LR-based gate sizing, which is also the baseline of our method. Given leakage power as the objective, the gate sizing problem is formulated as follows.

$$\begin{aligned} \mathcal{PP}: \quad & \text{minimize} \quad && leakage && \text{(1a)} \\ & \text{s.t.} && a_i + d_{i\to j} \le a_j \quad \forall arc_{i\to j} && \text{(1b)} \\ & && a_k \le q_k \quad \text{for each path endpoint } n_k && \text{(1c)} \end{aligned}$$

In Equation (1), a_i is the arrival time at node n_i, $d_{i\to j}$ is the delay of the timing arc $arc_{i\to j}$ connecting n_i and n_j, and q_k is the required time at n_k. Equations (1b) and (1c) represent how the arrival time is propagated in the timing graph and constrained by the timing closure set by users. Note that, the delay of a net is assumed to be zero in this formulation.

Following the LR procedure, we introduce a non-negative variable called Lagrange multiplier (LM) for each constraint and move them to the objective function. After such transformation, the original problem is converted to Lagrangian relaxation subproblems (LRS) associated with the LMs ($\boldsymbol{\lambda}$). By applying the KKT conditions to the LRS, it can be further simplified as shown in Equation (2).

$$\mathcal{LRS}/(\boldsymbol{\lambda}): \quad \text{minimize} \quad leakage + \sum \lambda_{i\to j} d_{i\to j}. \tag{2}$$

Let $Q(\boldsymbol{\lambda})$ denote the optimal value of the problem $\mathcal{LRS}/(\boldsymbol{\lambda})$. The Lagrangian dual problem (LDP) is as follows.

$$\mathcal{LDP}: \quad \text{maximize} \quad Q(\boldsymbol{\lambda}) \tag{3}$$

By iteratively updating $\boldsymbol{\lambda}$, the objective is gradually optimized while the constraint violations will be minimized. To model the timing violation induced by each arc, λs are updated as follow:

$$\lambda_{i\to j} = \lambda_{i\to j} \times \begin{cases} (1 + \frac{a_j - q_j}{T})^{\frac{1}{\alpha}} & a_j \ge q_j \\ (1 + \frac{q_j - a_j}{T})^{-\alpha} & a_j < q_j \end{cases}, \tag{4}$$

where T is the clock period and α is a positive hyper-parameter. As discussed above, only the LMs that satisfy the KKT conditions need to be considered, which is ensured by a procedure called projection after updating the LMs. Given the LMs at each iteration, a greedy algorithm is adopted to optimized $\mathcal{LRS}/(\boldsymbol{\lambda})$. To be specific, each logic gate in the circuit is scanned in topological order, where the best type of each gate is selected sequentially. When selecting the type of each gate, the following metrics are considered: (1) the change in load violations, slack violations, and slew violations, (2) the sum of lambda delay in the local graph. The cell type that has the least violations and biggest gain will be selected.

2.2 Heterogeneous Graph Neural Network

GNNs are neural networks that deal with graph-structured data and typically take graphs and possible node/edge features as input. In general, GNNs fall into two categories, namely spectral approaches and non-spectral approaches. Spectral approaches [22] work with a spectral representation of the graphs, and conduct the convolution operation in the Fourier domain by computing the eigen-decomposition of the graph Laplacian. While non-spectral approaches [23] operate on groups of spatially close neighbors, and aggregate features from a node's local neighborhoods. In each iteration, the representation of a node is updated by combining its own representation with the representation aggregated from its neighbors. After L iterations, each node v collects information from nodes with distance at most L from v.

Besides simple graphs whose nodes and edges are of the same type, tremendous real world graphs consist of nodes and edges of different types, which are known as heterogeneous graphs. Heterogeneous graph neural networks are special GNNs designed for heterogeneous graphs. To deal with heterogeneity, one approach is to use different weight matrices and encoding schema for different types of nodes and edges [24]. Another approach converts a heterogeneous graph to multiple homogeneous graphs by utilizing meta-path whose endpoints have the same type but are not directly linked in the original heterogeneous graph [25].

2.3 Imitation Learning

RL [26] is a powerful machine learning technique that trains an agent through its interaction with the environment, and has shown its ability to solve complex problems in recent years [27, 28]. Given a state of environment, the agent learns a policy to choose an action to maximize the cumulative rewards returned from the environment.

As an approach related to RL, imitation learning learns a policy by mimicking the demonstration of an expert instead of learning from scratch through interaction. In imitation learning, an expert gives examples of state and action pairs. The policy is then trained in supervised manner with the expert's action as the label [29]. The policy training could be further refined by various techniques, including querying an interactive expert for the action at present state during training [30] or learning the unknown reward function from the training samples, known as inverse reinforcement learning [31].

3 HGNN-BASED IMITATION LEARNING FOR GATE SIZING

3.1 Pipeline Overview

Our goal is to improve the efficiency of the traditional LR-based gate sizing without sacrificing its effectiveness. Algorithm 1 demonstrates our ML-accelerated LDP solver and Figure 2 compares the pipelines of the baseline and our proposed approach. To accelerate $\mathcal{LRS}/(\lambda)$, instead of scanning all gates in each iteration, we only try those that are likely to be changed. To be specific, the candidate gates are predicted by imitation learning given the circuit state encoded by an HGNN. Then the method mentioned in Section 2.1 will choose the best type for them.

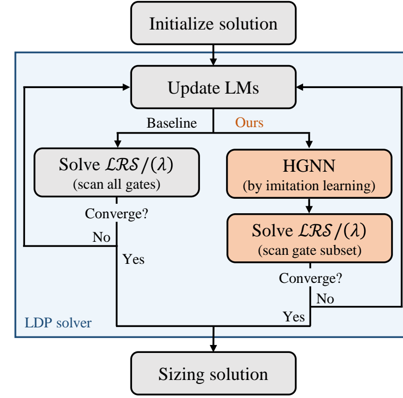


Figure 2: Pipelines of the baseline and our accelerated solver.

Algorithm 1 Our ML-accelerated LDP solver

Input: The input netlist and the inital values of λs.
Output: A netlist with optimized leakage power and timing.

1: update timing
2: $useHGNN = True$
3: **repeat**
4: update λs by (Equation (4))
5: **if** $useHGNN$ **then**
6: $S \longleftarrow$ gates predicted by HGNN model
7: **else**
8: $S \longleftarrow$ all gates
9: **end if**
10: $S_c \longleftarrow \emptyset$
11: **for each** gate $g \in S$ **do**
12: select the best type for g
13: add g to S_c if its type is changed
14: **end for**
15: **if** $\frac{|S_c|}{|S|} < th$ **then**
16: $useHGNN = False$
17: **end if**
18: **until** convergence
19: **return** the best solution of all iterations

As shown in Figure 1, the ratio of changed gates decreases very fast. After a few iterations, only a small number of gates are positive samples. The label distribution becomes very different from the beginning and extremely imbalanced. This leads to the Out-of-Distribution (OOD) problem which is known to be hard to address [32, 33]. To deal with the OOD problem, in practice, we calculate the ratio of gates in the candidate set that are indeed changed for each iteration. If the ratio drops below a threshold th set as 0.2 in our implementation, which implies that the label distribution has already shifted largely, the solver will scan every gate in the remaining iterations without using our model as filter.

In the following sections, we will give the details of circuit representation learning with HGNN model and imitation learning strategy.

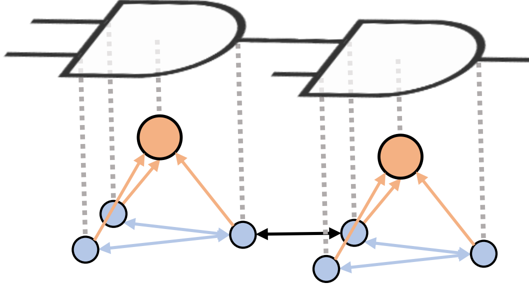


Figure 3: An illustration of representing a circuit in a heterogeneous graph. Above is a small circuit with two gates, where each gate has 2 input pins and 1 output pin. Below is its corresponding graph representation. The orange nodes are the gate nodes in V_g and the blue nodes are the pin nodes in V_p. The edges in E_g, E_p and E_n are presented in orange, blue, and black respectively.

3.2 Circuit Modeling

The circuit is represented by a heterogeneous directed graph $G = (V, E)$. Specially, $V = V_g \cup V_p$, where V_g contains the **g**ate nodes and V_p contains the **p**in nodes. And there are three types of edges, namely $E = E_g \cup E_p \cup E_n$, where E_g are edges from pins to their corresponding **g**ates; E_p are forward and backward edges between each input pin and each output **p**in inside a gate; E_n are forward and backward edges between input pins and their driver pins in a **n**et. See Figure 3 for an illustration.

The features of nodes and edges are summarized in Table 1. Note that we add backward edges to facilitate message passing on the graphs, but the data flow direction could still be preserved since we include the input/output pin type in node features. Considering the rise/fall timing constraints, the length of features is 6 for input pins and output pins in V_p, 4 for any $v \in V_g$, and 6 for any $e \in E$. We normalize the slack in pin features, delay and rcdelay in edge features by the clock period of the netlist. These features will further be embedded through embedding or linear layers depending on whether the feature is categorical or numerical, and concatenated into a vector of length 64. Specifically, the categorical and numerical features of each gate node are embedded into 18 and 46 dimensions respectively while those of each pin node are embedded into 8 and 56 dimensions respectively. In the following, we use $h(v) \in \mathbf{R}^d$ to denote the feature vector of node v and $H_v \in \mathbf{R}^{|V| \times d}$ to represent the node feature matrix formed by all nodes, where $d = 64$. Similarly definitions are used for $h(e) \in \mathbf{R}^d$ and $H_e \in \mathbf{R}^{|E| \times d}$.

3.3 Circuit State Encoding with HGNN

Based on the embedded features, we use our proposed HGNN model to encode the local circuit state for each gate. Figure 4 shows the architecture of our HGNN model. Node features H_v and edge features H_e are batch normalized and fed into the model, which consists of L HGNN layers. Each layer is built upon the graph neural (GN) block of [34] with additional *attention mechanism* and *mechanism for processing heterogeneous graphs*. Specifically, the edge features H_e^l and node features H_v^l are the inputs for the l-th layer. Batch normalization operation is first applied to the input features. Then

Table 1: The features for the nodes and edges in G.

Type	Feature	Length			
		V_p		V_g	E
		input pin	output pin		
Categorical	footprint	-	-	1	-
	is_sizable	-	-	1	-
	is_output_pin	1	1	-	-
Numerical	#fan-ins	-	-	1	-
	area	-	-	1	-
	slack	2	2	-	-
	capacitance	1	-	-	-
	gain	0	1	-	-
	slew	2	2	-	-
	delay	-	-	-	2
	rc_delay	-	-	-	2
	lambda	-	-	-	2
Total		6	6	4	6

the edge features and node features are updated by an edge aggregation model, and a node aggregation model respectively. The updated features further go through a fully connected network and the normalization process, which results in the intermediate features $H_e^{l\prime}$ and $H_v^{l\prime}$. A residue connection is employed at last:

$$H_v^{l+1} = H_v^{l\prime} + H_v^l \tag{5a}$$

$$H_e^{l+1} = H_e^{l\prime} + H_e^l \tag{5b}$$

Finally, the output H_v^{l+1} and H_e^{l+1} will be the input features for the next layer.

After L layers of convolution, each gate is able to gather information from its L-hop neighborhood in its feature. Considering that a 4-hop neighborhood is already enough to approximate the global circuit environment well in the original LDP solver, we use $L = 4$ in our model. Finally, the output node features at the last layer, H_v^{L+1}, which encode the current gate state of the circuit, are passed to a policy network to predict the action.

In the following, we will introduce the details of node and edge aggregation models shown in Figure 4.

3.3.1 Edge Aggregation Model. For an edge e of type $t \in \{g, p, n\}$ from node u to node v, its edge feature $h(e)$ is updated to $h'(e)$ by Equation (6).

$$h'(e) = \text{ReLU}(W_e^t \cdot [h(u) \parallel h(e) \parallel h(v)]), \tag{6}$$

where $W_e^t \in \mathbf{R}^{d \times 3d}$ is the parameter matrix for edge type t and $\parallel$ means concatenation operation.

3.3.2 Node Aggregation Model. We leverage the EGAT layer [35] in the node aggregation model to update the node features. For a node v of type $t \in \{g, p\}$, that has m in-coming edges e_i for $i \in \{1 \dots m\}$ with corresponding start node v_i, the node aggregation model first aggregates $(h(v) \parallel h(e_i) \parallel h(v_i))$ to $r_i \in \mathbf{R}^d$ via a

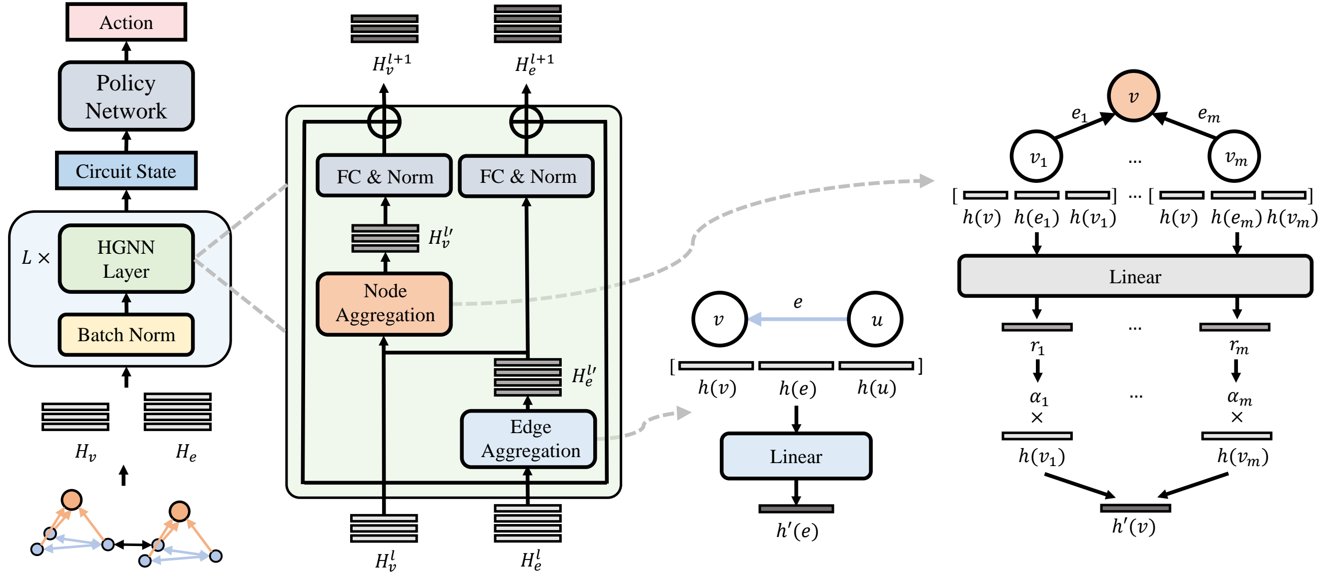


Figure 4: The architecture of our proposed HGNN model and policy network. The model overview is on the left while the details of our HGNN layer, the edge aggregation model, and the node aggregation model are shown on the right side. Each layer employs an edge aggregation model and a node aggregation model to update the edge and node features. The output features of the last layer encode the state of the circuit and are used to predict the action by the policy network.

linear layer, and compute an attention score α_i for v_i as follow:

$$r_i = \text{LeakyReLU}(W_r \cdot [h(v) \ \| \ h(e_i) \ \| \ h(v_i)]) \tag{7}$$

$$\alpha_i = \frac{\exp(a^T \cdot r_i)}{\sum_{i=1}^{m} \exp(a^T \cdot r_i)} \tag{8}$$

where $W_r \in \mathbf{R}^{d \times 3d}$ and $a \in \mathbf{R}^d$ are parameters shared by both node types. When v is a gate, α_i represents the importance of the pins on this gate. When v is a pin, α_i represents the importance of different pins and timing arcs connected to this pin. The node feature is then updated by

$$h'(v) = \sum_{i=1}^{m} \alpha_i h(v_i). \tag{9}$$

The attention mechanism allows the gates and pins to select the most important neighbors that carry critical timing or leakage information.

3.4 Candidates Selection by Imitation Learning

After the circuit state is encoded by the HGNN model, a policy network takes the encoded state to predict the action, i.e. selecting the set of candidate gates. Specifically, we employ a fully connected network as our policy network. We pass the features of the sizable gates to the policy network to produce the binary classification for each gate in parallel, indicating whether this gate is selected or not. Therefore, the total outputs will be of shape $|V_s| \times 2$, where V_s is the set of sizable gates.

We train our network in an imitation learning manner. The original LDP-solver is regarded as the expert to demonstrate actions for different states, which produces a sequence of state and action pairs. We treat these pairs as independent samples and train our network with supervision on these samples using the weighted Cross-Entropy loss:

$$Loss = -\frac{1}{|V_s|} \sum_{(v,l) \in D_s} w_l \cdot \log \frac{\exp(p_v^l)}{\exp(p_v^0) + \exp(p_v^1)}, \tag{10}$$

where $D_s = \{(v_i, l_i)\}_{i=1}^{|V_s|}$ is the set of pairs that represent whether the type of a sizable logic gate v_i is changed or not. Note that, $l_i = 1$ indicates gate v_i is changed and vice versa for $l_i = 0$. Here w_l with $l \in \{0, 1\}$ are reweighting parameters and (p_v^0, p_v^1) are the network's predicted value for gate v.

3.5 Inference on Large Benchmarks

During inference, for small benchmarks, we can take the whole heterogeneous directed graph and predict the cell selection by our HGNN model in one batch. However, due to memory limit, some of the benchmarks are too large to fit into memory after we convert them to graphs. To address this issue, instead of feeding the entire graph into the model, we divide the sizable gates into several batches. We take one batch at a time and sample the subgraph induced by the L-hop neighborhood of the target gates in the batch. The sampled subgraph is fed into the HGNN model and produce the predicted action for this batch. Since HGNN only gathers information from the L-hop neighborhood for each gate, this technique allows us to infer on large benchmarks without performance loss.

4 EXPERIMENTS

To evaluate our accelerated gate sizer, we take the benchmarks from ISPD 2013 Contest [21] as our dataset. Note that, we use (slow) and (fast) to denote a netlist with a slow and fast clock period respectively, and treat them as different benchmarks. As shown in

Table 2: The netlist size and number of sizable gates for each benchmark in our dataset.

	Benchmark	Netlist Size	Sizable
Training Set	usb_phy (slow)	622	510
	pci_bridge32 (slow/fast)	30,762	27,244
	fft (fast)	33,791	30,782
Validation Set	usb_phy (fast)	622	510
	fft (slow)	33,791	30,782
Test Set	cordic (slow/fast)	42,936	41,673
	des_perf (slow/fast)	113,345	104,310
	edit_dist (slow/fast)	129,226	121,004
	matrix_mult (slow/fast)	159,641	153,542
	netcard (slow/fast)	984,093	884,427

Table 2, we use 4, 2, and 10 benchmarks for training, validation, and testing respectively. In order to fully test the generalizability of our model, the netlists of training/validation and test sets are completely different such that the difference in graph structure is maximized. Moreover, to test if it can scale to larger instances, we only train the model on the smallest netlists.

During training, for each training/validation benchmark, we take results of the first 25 iterations of the baseline LDP solver as samples. Therefore, we have 100 and 50 samples for training and validation respectively. We use a batch size of 1 and learning rate 0.002. We set dropout ratio as 0.2 and the class weight for positive class as 0.95 to address the label imbalance issue. We train our network for 250 epochs and take the model with the lowest loss on validation set.

Since our method presents a general acceleration method to Lagrangian Relaxation, it could easily be applied to other LR-based gate sizing algorithms. For the ease of experiments, we compared our method with the open-source method proposed in [9].

4.1 PPA Results

Table 3 compares the baseline [9] and our ML-accelerated algorithm from the aspects of leakage power, timing violation, load violation and runtime. The runtime of our method is the summation of model inference time on GPU and LDP solving time on CPU. In one iteration, model inference only accounts for 7.6% of the runtime on average. The model is invoked for 19% of total iterations and reduces the number of scanned gates to 16% of the total sizable gates on average. The slew violation is omitted in the table since both methods achieve zero slew violation for all test benchmarks. On average, our method achieves 12.7% lower timing violation, 16.9% lower load violation and 22.5% reduction of runtime with only 1.8% overhead of leakage power. Our model also reveals good generalizability considering that it is trained on benchmarks of size at most 33K but tested on large unseen benchmarks of size up to 984K.

To further study the behavior of our ML-accelerated sizer, we demonstrate the trend of precision, recall and F1 score of our model on four benchmarks in Figure 5. Due to the use of high positive class weight, the recall stays relatively high. And due to the aforementioned label imbalance and OOD problems, the precision of our model becomes worse as the number of iterations increases.

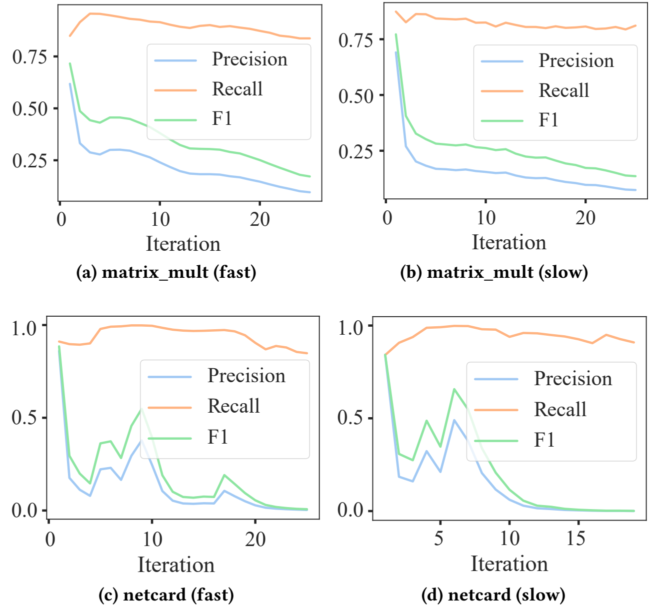


Figure 5: The precision, recall and F1 of selected benchmarks.

4.2 Comparison with Heuristics

Besides the baseline algorithm, which could be considered as a full selection strategy, we further compare our method with two more heuristics: slack-based selection and lambda-delay-based selection.

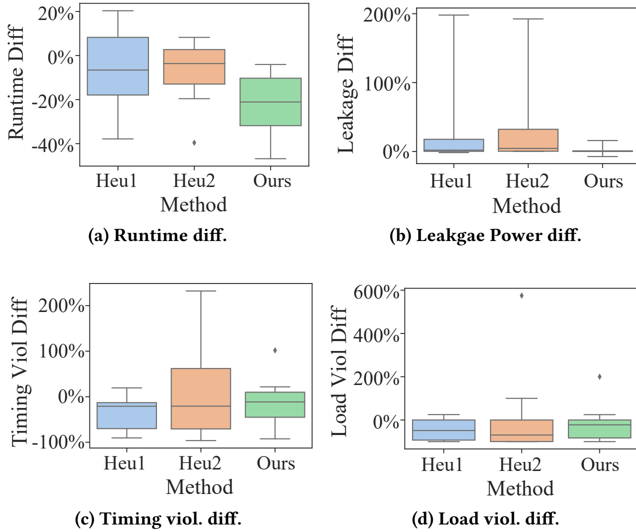
Since the LR-based method tends to change gates with large negative slack in the early stage as shown in Figure 1, it is natural to design the slack-based selection heuristic which selects gates with largest negative slack as the candidates for speedup. For each iteration, we sort the gates in descending order of their negative slack and select the top K gates as the candidates, where K is set to the same value as the number of gates selected by our ML-accelerated method.

Similarly, since the optimization objective includes minimizing the lambda delay, we implement another heuristic which selects the gates with largest local lambda delay as candidates. For each iteration, we sort the gates in descending order of their local lambda delay and select the top K gates likewise.

In Table 4 we show the performance difference of the three algorithms compared to the baseline. All results are averaged among 10 test benchmarks and displayed as percentage. In Figure 6, we use box plots to visualize the distribution of the performance difference on test benchmarks, where each distribution is summarized by a box with 5 lines that represent (1) the minimum, (2) the first quartile, (3) the median, (4) the third quartile, and (5) the maximum. For runtime reduction, our method runs 22.5% faster than the baseline on average while the two heuristics are only 7.0% and 7.1% faster. And according to Figure 6(a), our method achieves acceleration on all test benchmarks, i.e. no positive runtime difference, while the other two fail to obtain runtime reduction on some benchmarks. For leakage power optimization, our method is comparable to the baseline with only 1.8% increase on average, but the ones achieved

Table 3: Results on 10 test benchmarks. T is the clock period. Our method obtains an average 22.5% reduction of runtime with only slight overhead of leakage power and violations.

Benchmark	T (ps)	Leakage Power (W)		Timing Viol. (ps)		Load Viol. (fF)		Runtime (min)		
		Baseline	Ours	Baseline	Ours	Baseline	Ours	Baseline	Ours	Diff.
cordic (fast)	2626	1.6343	**1.5145**	**683.34**	773.82	0.04	**0.00**	17.3	**16.6**	**-4.0%**
cordic (slow)	3000	**0.3074**	0.3083	**133.58**	162.60	0.04	**0.00**	11.3	**10.2**	**-9.7%**
des_perf (fast)	1140	**0.7353**	0.8080	1829.22	**875.73**	0.00	**0.00**	38.0	**20.2**	**-46.8%**
des_perf (slow)	1300	**0.3373**	0.3408	813.15	**220.57**	0.00	**0.00**	28.8	**16.0**	**-44.4%**
edit_dist (fast)	3000	**0.5740**	0.5766	110.94	**89.48**	0.47	**0.46**	33.1	**22.2**	**-32.9%**
edit_dist (slow)	3600	**0.4291**	0.4297	**30.25**	61.06	**1.62**	2.02	28.6	**22.7**	**-20.6%**
matrix_mult (fast)	2200	**2.0527**	2.3794	485.32	**369.80**	0.12	**0.01**	55.7	**43.7**	**-21.5%**
matrix_mult (slow)	2800	0.4765	**0.4635**	71.63	**69.07**	**0.01**	0.03	41.8	**36.8**	**-12.0%**
netcard (fast)	2000	**5.1473**	5.1590	63.93	**4.84**	304.95	**130.02**	86.0	**61.5**	**-28.5%**
netcard (slow)	2400	**5.1168**	5.1219	0.95	**0.95**	3226.91	**1849.33**	50.9	**48.6**	**-4.5%**
Average			+1.8%		**-12.7%**		**-16.9%**			**-22.5%**



(a) Runtime diff.

(b) Leakgae Power diff.

(c) Timing viol. diff.

(d) Load viol. diff.

Figure 6: Distributions of metric difference on test set, where Heu1 represents the slack-based heuristic and Heu2 represents the lambda-delay-based heuristic.

Table 4: Comparison with two heuristics in terms of average difference in runtime, leakage power, timing violation and load violation.

Metric	Slack-based	Lambda-delay -based	Ours
Runtime	-7.0%	-7.1%	**-22.5%**
Leakage Power	+25.7%	+29.5%	**+1.8%**
Timing Viol.	**-35.0%**	+6.7%	-12.7%
Load Viol.	**-44.6%**	+13.7%	-16.9%

by the heuristics are unacceptable with more than 25% degradation. In terms of timing and load violation, lambda-delay-based heuristic has worse average timing and load violation compared to baseline and has large distribution variance as shown in Figure 6(c) and Figure 6(d), which indicates unstable performance. The slack-based heuristic improves the two violation and has similar distribution variance to our method. But as shown before, it has significantly worse leakage power and unstable runtime reduction.

The two heuristics neglect leakage optimization and only consider the timing constraint related part in the optimization objective by focusing on gates with large negative slack or lambda delay. Therefore, they have either worse optimization results or unstable performance. This validates that our HGNN model can not only select gates that may violate timing or load constraints, but also those with small negative slack, and thus ensure a balance between leakage and timing optimization.

Table 5: The performance of models with different feature removed.

Model	Precision	Recall	F1	Accu.
-footprint	0.42	0.89	0.54	0.46
-is_sizable	0.42	0.88	0.54	0.48
-is_output_pin	0.43	**0.94**	0.56	0.50
-#fan-ins	0.42	0.88	0.53	0.46
-area	0.43	0.91	0.56	0.51
-slack	0.64	0.47	0.54	0.73
-capacitance	0.44	0.85	0.55	0.52
-gain	0.44	0.88	0.56	0.53
-slew	0.40	0.63	0.48	0.52
-delay	0.46	0.69	0.51	0.60
-rc_delay	0.46	0.84	0.56	0.55
-lambda	0.43	0.93	0.56	0.48
All	**0.69**	0.90	**0.78**	**0.83**

4.3 Feature Importance

We conduct ablation study to evaluate the importance of different node and edge features. For each feature used in our model, we remove it and retrain our model. Table 5 shows, for the first iteration, the average precision, recall, F1 and accuracy of these models on test benchmarks.

According to Table 5, using all features obtains the highest F1 score and accuracy. Removing one of the features will lead to at

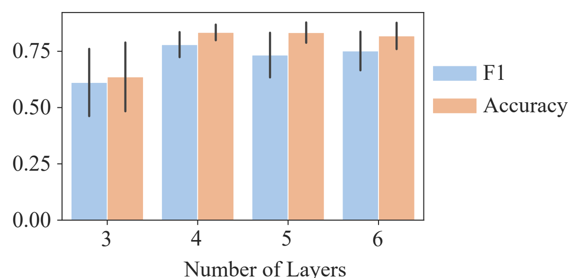


Figure 7: Comparison between models with different number of HGNN layers on F1 and accuracy. Results are averaged on test set.

least 20% decrease in F1 score. Specifically, removing slack will make model recall decrease significantly. Removing the following features will result in drastic loss of model precision: cell footprint, if a cell is sizable, if a pin is output pin, number of cell fan-ins, cell area, pin capacitance, gain, edge rc_delay and edge lambda. And removing edge slew and delay will influence both precision and recall.

In our scenario, high model recall ensures that most positive gates could be included in the candidate set and therefore is crucial to PPA results. High model precision, on the other hand, makes sure the candidate set contains less redundant gates and thus mostly contributes to reducing runtime.

4.4 Number of HGNN Layers

For models with different number of HGNN layers, Figure 7 shows their average F1 score and accuracy for the first iteration on test benchmarks. Model with 3 layers has the worst F1 score and accuracy, since the original LR employs a 4-hop neighbourhood to represent the local environment of a gate and the model needs at least 4 layers to model that environment. Models with more than 4 layers have slightly worse F1 and accuracy than the one with 4 layers, possibly because of the over-smoothing issue caused by the increasing model depth, which has been shown to be especially harmful to node classification tasks [36].

5 CONCLUSION

In this paper, we propose an HGNN model trained with imitation learning to accelerate LR-based gate sizing algorithms. The HGNN model is able to effectively encode timing and power information and transfer the learned knowledge to larger unseen graphs. The resulting algorithm achieves comparable performance to the baseline sizer in a 22.5% shorter runtime on the ISPD 2013 Contest benchmarks. There is still large potential to improve our model to achieve better speedup by easing the OOD problem.

Applications to Other LR-based Algorithms. Notably, our approach is general and applicable to LR-based algorithms, where similar iterative procedure is performed and in each iteration a Lagrangian relaxation subproblem is optimized by searching a solution space [37–41]. Besides its application in the gate sizing problem demonstrated in this work, our method has the potential to narrow the search space for LR-based algorithms in other tasks, e.g. sparsing the graph in network design [40] and reducing candidate warehouses in supply chain management [41].

ACKNOWLEDGEMENT

This work was supported by Hong Kong Innovation and Technology Fund Project No. ITS/170/20.

REFERENCES

[1] V. Bhardwaj, "Shift left trends for design convergence in SOC: An EDA perspective," *International Journal of Computer Applications*, vol. 174, no. 16, pp. 22–27, 2021.
[2] T. Kutzschebauch and L. Stok, "Congestion aware layout driven logic synthesis," in *IEEE/ACM International Conference on Computer Aided Design*, pp. 216–223, IEEE, 2001.
[3] M. Ebrahimi, F. Oboril, S. Kiamehr, and M. B. Tahoori, "Aging-aware logic synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design*, pp. 61–68, IEEE, 2013.
[4] K. O. Tinmaung, D. Howland, and R. Tessier, "Power-aware FPGA logic synthesis using binary decision diagrams," in *Proceedings of the 15th International Symposium on Field Programmable Gate Arrays*, pp. 148–155, 2007.
[5] W. Ning, "Strongly NP-hard discrete gate-sizing problems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 8, pp. 1045–1051, 1994.
[6] S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S.-M. Kang, "An exact solution to the transistor sizing problem for CMOS circuits using convex optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 11, pp. 1621–1634, 1993.
[7] S. Hu, M. Ketkar, and J. Hu, "Gate sizing for cell-library-based designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 6, pp. 818–825, 2009.
[8] C. C. Chu and M. D. Wong, "Greedy wire-sizing is linear time," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 4, pp. 398–405, 1999.
[9] G. Flach, T. Reimann, G. Posser, M. Johann, and R. Reis, "Effective method for simultaneous gate sizing and vth assignment using lagrangian relaxation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 4, pp. 546–557, 2014.
[10] A. Sharma, D. Chinnery, S. Dhamdhere, and C. Chu, "Rapid gate sizing with fewer iterations of lagrangian relaxation," in *2017 IEEE/ACM International Conference on Computer-Aided Design*, pp. 337–343, IEEE, 2017.
[11] A. Sharma, D. Chinnery, T. Reimann, S. Bhardwaj, and C. Chu, "Fast lagrangian relaxation-based multithreaded gate sizing using simple timing calibrations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 7, pp. 1456–1469, 2019.
[12] G. Wu and C. Chu, "Simultaneous slack matching, gate sizing and repeater insertion for asynchronous circuits," in *2016 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1042–1047, IEEE, 2016.
[13] J. Chen, J. Kuang, G. Zhao, D. J.-H. Huang, and E. F. Young, "Pros: A plug-in for routability optimization applied in the state-of-the-art commercial EDA tool using deep learning," in *2020 IEEE/ACM International Conference On Computer Aided Design*, pp. 1–8, IEEE, 2020.
[14] T. Qu, Y. Lin, Z. Lu, Y. Su, and Y. Wei, "Asynchronous reinforcement learning framework for net order exploration in detailed routing," in *2021 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1815–1820, IEEE, 2021.
[15] C.-W. Pui, G. Chen, Y. Ma, E. F. Young, and B. Yu, "Clock-aware ultrascale fpga placement with machine learning routability prediction," in *2017 IEEE/ACM International Conference on Computer-Aided Design*, pp. 929–936, IEEE, 2017.
[16] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu, "High performance graph convolutional networks with applications in testability analysis," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.
[17] B. Jiang, L. Liu, Y. Ma, H. Zhang, B. Yu, and E. F. Young, "Neural-ILT: migrating ilt to neural networks for mask printability and complexity co-optimization," in *2020 IEEE/ACM International Conference On Computer Aided Design*, pp. 1–9, IEEE, 2020.
[18] Y.-C. Lu, S. Nath, V. Khandelwal, and S. K. Lim, "RL-Sizer: VLSI gate sizing for timing optimization using deep reinforcement learning," in *58th ACM/IEEE Design Automation Conference*, pp. 733–738, IEEE, 2021.
[19] H. Tennakoon and C. Sechen, "Gate sizing using lagrangian relaxation combined with a fast gradient-based pre-processing step," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pp. 395–402, 2002.
[20] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Computing Surveys*, vol. 50, no. 2, pp. 1–35, 2017.

[21] M. M. Ozdal, C. Amin, A. Ayupov, S. Burns, G. Wilke, and C. Zhuo, "An improved benchmark suite for the ISPD-2013 discrete cell sizing contest," in *Proc of ACM International Symposium on Physical Design*, pp. 168–170, 2013.

[22] J. Bruna, W. Zaremba, A. Szlam, and Y. Lecun, "Spectral networks and locally connected networks on graphs," in *ICLR*, 2014.

[23] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *KDD*, 2016.

[24] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*, 2018.

[25] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The World Wide Web Conference*, pp. 2022–2032, 2019.

[26] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[28] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[29] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, 2009.

[30] K. Judah, A. P. Fern, and T. G. Dietterich, "Active imitation learning via reduction to i.i.d. active learning," in *2012 AAAI Fall Symposium Series*, 2012.

[31] B. D. Ziebart, A. L. Maas, J. A. Bagnell, A. K. Dey, *et al.*, "Maximum entropy inverse reinforcement learning.," in *AAAI*, vol. 8, pp. 1433–1438, 2008.

[32] Z. Shen, J. Liu, Y. He, X. Zhang, R. Xu, H. Yu, and P. Cui, "Towards out-of-distribution generalization: A survey," *arXiv preprint arXiv:2108.13624*, 2021.

[33] X. Zhang, P. Cui, R. Xu, L. Zhou, Y. He, and Z. Shen, "Deep stable learning for out-of-distribution generalization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5372–5382, 2021.

[34] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[35] K. Kamiński, J. Ludwiczak, M. Jasiński, A. Bukala, R. Madaj, K. Szczepaniak, and S. Dunin-Horkawicz, "Rossmann-toolbox: a deep learning-based protocol for the prediction and design of cofactor specificity in rossmann fold proteins," *Briefings in Bioinformatics*, vol. 23, no. 1, p. bbab371, 2022.

[36] W. Huang, Y. Rong, T. Xu, F. Sun, and J. Huang, "Tackling over-smoothing for general graph convolutional networks," *arXiv preprint arXiv:2008.09864*, 2020.

[37] L. Yang and X. Zhou, "Constraint reformulation and a lagrangian relaxation-based solution algorithm for a least expected time path problem," *Transportation Research Part B: Methodological*, vol. 59, pp. 22–44, 2014.

[38] M. El-Kebir, J. Heringa, and G. W. Klau, "Lagrangian relaxation applied to sparse global network alignment," in *IAPR International Conference on Pattern Recognition in Bioinformatics*, pp. 225–236, Springer, 2011.

[39] A. A. Butt and R. T. Collins, "Multi-target tracking by lagrangian relaxation to min-cost network flow," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1846–1853, 2013.

[40] C. Xie and M. A. Turnquist, "Lane-based evacuation network optimization: An integrated lagrangian relaxation and tabu search approach," *Transportation Research Part C: Emerging Technologies*, vol. 19, no. 1, pp. 40–63, 2011.

[41] A. Diabat, J.-P. Richard, and C. W. Codrington, "A lagrangian relaxation approach to simultaneous strategic and tactical planning in supply chain design," *Annals of Operations Research*, vol. 203, no. 1, pp. 55–80, 2013.