ASP-DAC 2023
28th Asia and South Pacific Design Automation Conference
Date: January 16 – 19, 2023
ASIA SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE

Graph-Learning-Driven Path-Based Timing Analysis Results Predictor from Graph-Based Timing Analysis

Yuyang Ye[1], Tinghuan Chen[2], Yifan Gao[1], Hao Yan[1], Bei Yu[2], Longxing Shi[1]

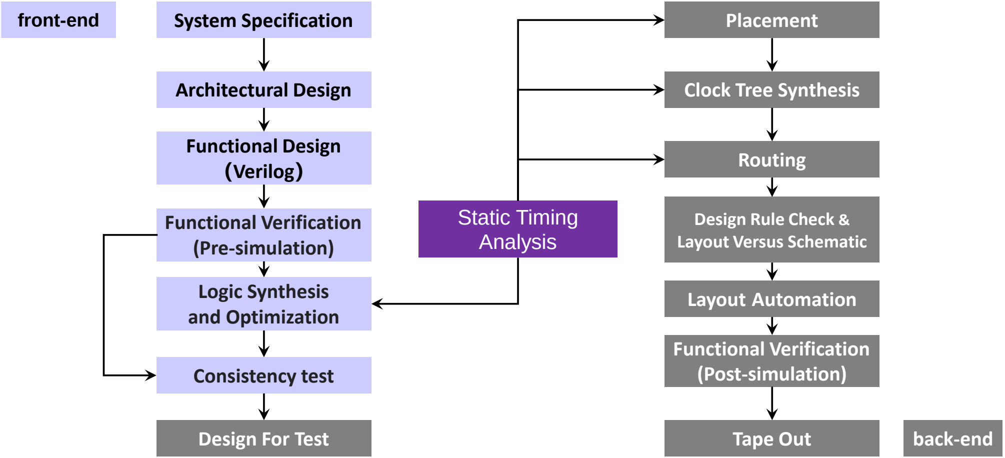
[1]Southeast University
[2]The Chinese University of Hong Kong

Jan. 18, 2023

Introduction

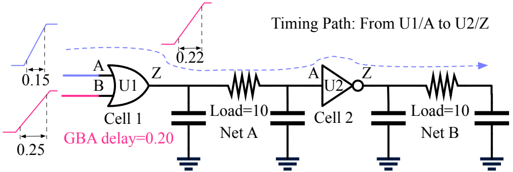
Static Timing Analysis (STA)

STA plays an important role in the design flow for timing closure.

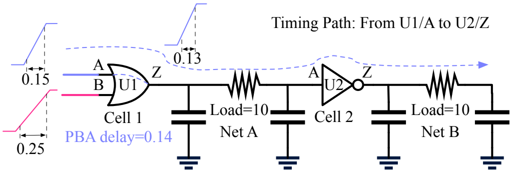


For achieving a tradeoff between efficiency and accuracy, STA is divided into two kinds:

- **Graph-based Analysis (GBA)**
  (fast but inaccurate)



- **Path-based Analysis (PBA)**
  (accurate but slow)



Motivation: PBA based on GBA

Molina [1] and Kahng [2] name **fast prediction of PBA results based on GBA results** as a solution to achieve runtime and accuracy tradeoff

Kahng et al. [3] develop **two tree-based classification and regression models** to capture divergence in cell slew/delay in PBA and GBA timing mode

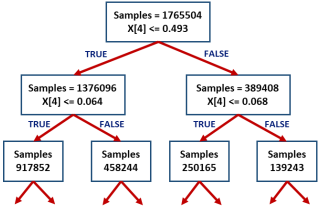
GBA Timing Results

Point	Trans	Incr	Path
U0_reg/CP	0.00000	0.00000	0.00000
U0_reg/Q	0.02015	0.05688	0.05688
U1/Z	0.03045	0.02473	0.08161
U2/Z	0.01066	0.02080	0.10241
Data arrival time			0.10241

PBA Timing Results

Point	Trans	Incr	Path
U0_reg/CP	0.00000	0.00000	0.00000
U0_reg/Q	0.01308	0.04382	0.04382
U1/Z	0.02745	0.01973	0.06355
U2/Z	0.00766	0.01808	0.08163
Data arrival time			0.08163

(a)



(b)

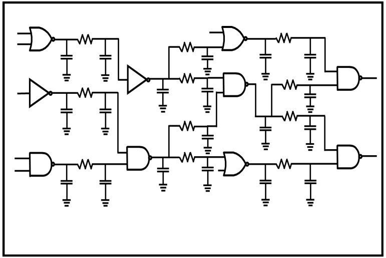
(a) From GBA to PBA; (b) Tree-based classification.

[1] EDA vendors should improve the runtime performance of path-based timing analysis
[2] Machine learning applications in physical design: Recent results and directions
[3] Using machine learning to predict path-based slack from graph-based timing analysis

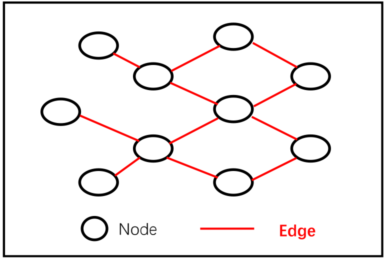
Motivation: Graph Learning for Timing Analysis

Graph learning methods are used to solve various EDA problems.

- **Cells $\rightarrow$ Nodes**
  Node features are researched in many problems
  Cell information is collected



- **Nets $\rightarrow$ Edges**
  Edge features are not fully considered
  Net information is ignored



Definition: Edge-featured Graph

An edge-featured graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \boldsymbol{X}, \boldsymbol{H}\}$ is defined as an undirected graph consisting of:

- a node set $\mathcal{V} = \{v^{(1)}, v^{(2)}, \ldots, v^{(n)}\}$, where $|\mathcal{V}| = N$. It denotes cell set on critical paths;
- an edge set $\mathcal{E}$, where $|\mathcal{E}| = M$. It denotes net set on critical paths;
- node features $\boldsymbol{X} \in \mathbb{R}^{n \times k_x}$, where i^{th} row vector $\boldsymbol{x}_i \in \mathbb{R}^{k_x}$ is the node features for the i^{th} node;
- edge features $\boldsymbol{H} \in \mathbb{R}^{m \times k_h}$, where the row vector $\boldsymbol{h}_p \in \mathbb{R}^{k_h}$ is the edge features for the p^{th} edge or the edge between i^{th} and j^{th} node.

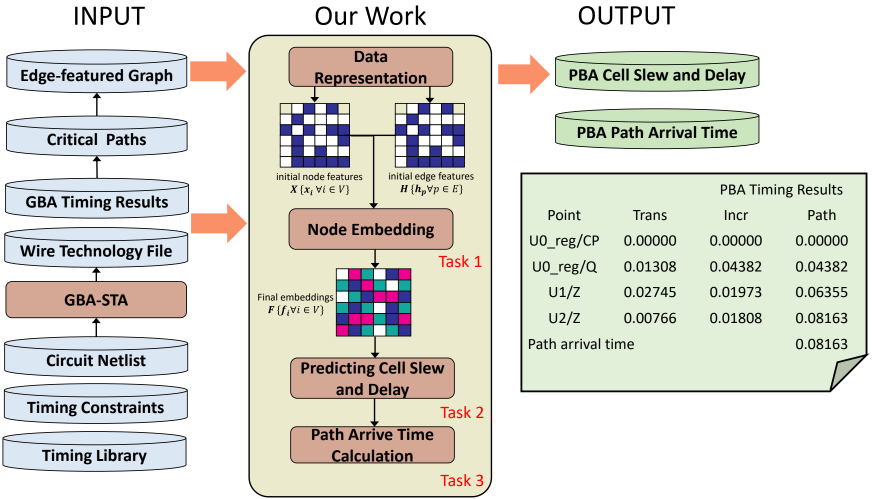
Problem 1:

- Given a training set P_{train} which includes edge-featured graphs representing critical paths with GBA and PBA timing results in training cases
- Train a graph-learning based model based on P_{train}
- Given a test set P_{test} (where $P_{\text{test}} \cap P_{\text{train}} = \emptyset$) which includes edge-featured graphs representing critical paths with GBA results in testing cases.
- Generate their PBA timing results in P_{test} using the trained model based on given GBA timing results and timing path structure information without additional STA runtime.

Algorithms

Overall Flow

In our work, Problem 1 is divided into three tasks based on delay calculation progress: node embedding, cell slew and delay prediction, path arrive time calculation.

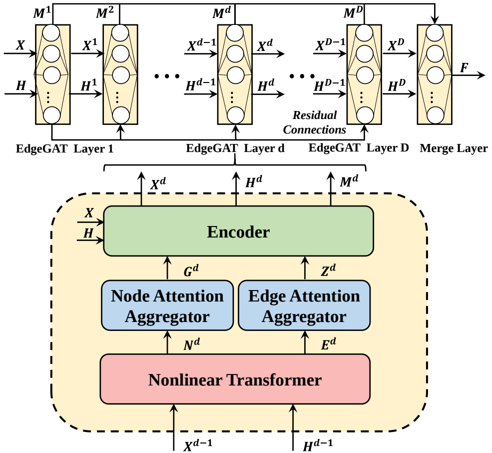


Point	Trans	Incr	Path
U0_reg/CP	0.00000	0.00000	0.00000
U0_reg/Q	0.01308	0.04382	0.04382
U1/Z	0.02745	0.01973	0.06355
U2/Z	0.00766	0.01808	0.08163
Path arrival time			0.08163

Cell Features and **Edge Features** are selected based on circuit knowledge and parameter-sweeping experiments, which can assist EdgeGAT.

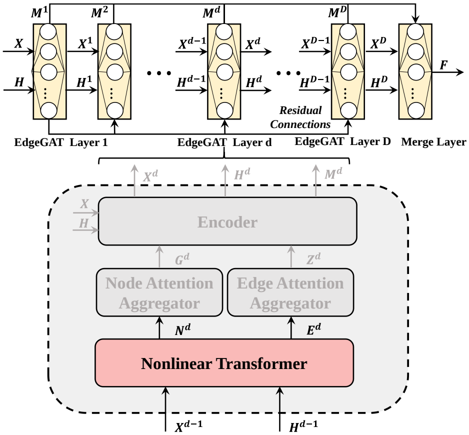
Type	Name	Description
Node	cell delay	delay of cell
	cell output slew	transition time of cell output pin
	cell input slew	transition time of cell input pin on path
	cell input slew type	rise or fall
	cell threshold voltage	threshold voltage of cell
	wst cell input slew	worst transition time of input pins
	cell drive strength	drive strength of cell
	cell functionality	functionality of cell
	tot cell input cap	sum of cell input pin cap
	tot cell load cap	total load capacitance of cell
Edge	net delay	delay of net
	net slew type	rise or fall
	net output slew	transition time of net output pin
	net input slew	transition time of net input pin
	tot net cap	sum of net capacitance
	tot net res	sum of net resistance
	net input cap	capacitance of driver cell for net
	tot net load cap	total capacitance of load cells

To predict the cell slew and delay accurately, **EdgeGAT layers** and **merge layer** in deep EdgeGAT are used to generate new node embedding $\boldsymbol{F}$: $\{f_i, \forall i \in \mathcal{V}\}$ for cells in circuit which is based on **node (cell) features** $\boldsymbol{X}$: $\{x_i, \forall i \in \mathcal{V}\}$, **edge (net) features** $\boldsymbol{H}$: $\{\boldsymbol{h}_p, \forall p \in \mathcal{E}\}$, and timing path structural information.



EdgeGAT Layer (Transformer)

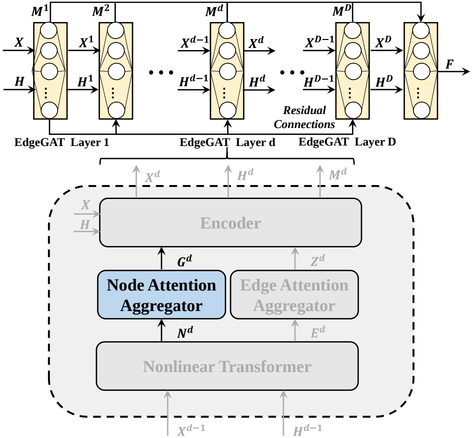
To achieve nonlinear transforming in the d-th EdgeGAT layer, two learnable matrices, $\boldsymbol{W}_X^d \in \mathbb{R}^{K_X^d \times K_X^{d-1}}$, $\boldsymbol{W}_H^d \in \mathbb{R}^{K_H^d \times K_H^{d-1}}$ and a hyper-parameter l^d, are used to transform the input node features $\{\boldsymbol{x}_i^{d-1} \in \mathbb{R}^{K_X^{d-1}}, \forall i \in \mathcal{V}\}$ and edge features $\{\boldsymbol{h}_p^{d-1} \in \mathbb{R}^{K_H^{d-1}}, \forall p \in \mathcal{E}\}$ into latent representations $\boldsymbol{n}_i^d$ and $\boldsymbol{e}_i^d$:



Nonlinear Transformer:

$$\boldsymbol{n}_i^d = ((1 - l^d)\boldsymbol{I} + l^d \mathbf{W}_X^d) \cdot \boldsymbol{x}_i^{d-1}$$
$$\boldsymbol{e}_p^d = ((1 - l^d)\boldsymbol{I} + l^d \mathbf{W}_H^d) \cdot \boldsymbol{h}_p^{d-1}$$

The node attention aggregator accepts the transformed node and edge representations generated as inputs, $\boldsymbol{n}_i^d$ and $\boldsymbol{e}_i^d$, and produces **aggregated node representations** $\boldsymbol{g}_i^d$ based on node attention coefficients α.



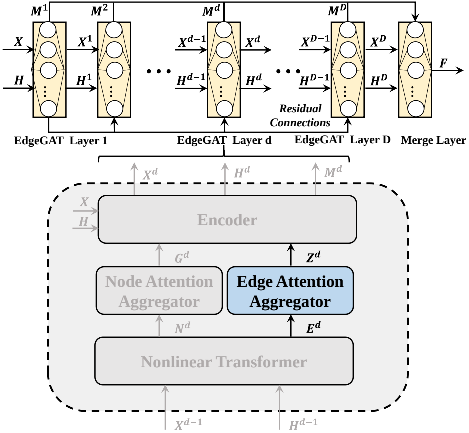
Node Attention Aggregator:

$$\alpha_{ij}^d = \frac{\exp\left(\mathrm{LeakyReLU}\left((\boldsymbol{a}^d)^\top \left[\boldsymbol{n}_i^d \| \boldsymbol{n}_j^d \| \boldsymbol{e}_{ij}^d\right]\right)\right)}{\sum_{k\in\mathcal{N}_i} \exp\left(\mathrm{LeakyReLU}\left((\boldsymbol{a}^d)^\top \left[\boldsymbol{n}_i^d \| \boldsymbol{n}_k^d \| \boldsymbol{e}_{ik}^d\right]\right)\right)}$$

$$\boldsymbol{g}_i^d = \sum_{j\in\mathcal{N}_i} \alpha_{ij} \boldsymbol{n}_j^d, \quad \forall i \in \mathcal{V}.$$

EdgeGAT Layer (Edge Attention Aggregator)

Different from node attention module, edge-attention module produces **aggregated edge representations** z_p^d based on edge attention coefficients β.

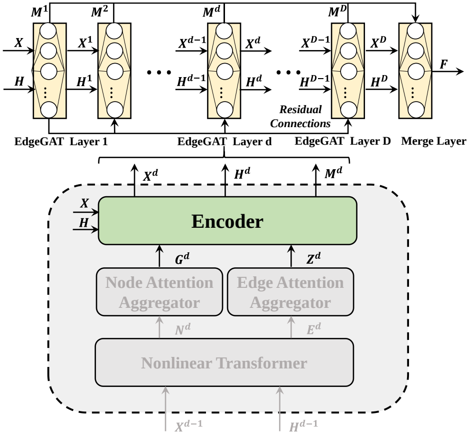


Edge Attention Aggregator:

$$\beta_{pq}^d = \frac{\exp\left(\mathrm{LeakyReLU}\left((\boldsymbol{b}^d)^\top \left[\boldsymbol{e}_p^d \| \boldsymbol{e}_q^d \| \boldsymbol{n}_{pq}^d\right]\right)\right)}{\sum_{k\in\mathcal{N}_p} \exp\left(\mathrm{LeakyReLU}\left((\boldsymbol{b}^d)^\top \left[\boldsymbol{e}_p^d \| \boldsymbol{e}_k^d \| \boldsymbol{n}_{pk}^d\right]\right)\right)}$$

$$\boldsymbol{z}_p^d = \sum_{q\in\mathcal{N}_p} \beta_{pq} \boldsymbol{e}_q^d, \quad \forall p \in \mathcal{E}.$$

A non-linear transformation σ is performed to **encode the aggregated representations**. After encoding, we can get new node feature matrix $\boldsymbol{X}^d$, edge feature matrix $\boldsymbol{H}^d$, and edge-integrated feature matrix $\boldsymbol{M}^d$.



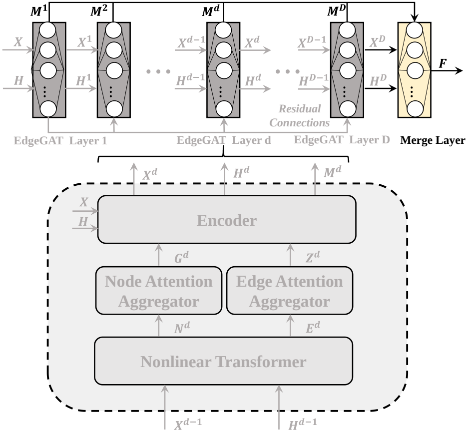
Encoder:

$$\boldsymbol{x}_i^d = \sigma(\boldsymbol{g}_i^d \| \boldsymbol{x}_i)$$

$$\boldsymbol{h}_p^d = \sigma(\boldsymbol{z}_i^d \| \boldsymbol{h}_i)$$

$$\boldsymbol{m}_i^d = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \left(\boldsymbol{n}_j \| \boldsymbol{e}_{ij}\right)\right)$$

We can get the final node embedding results $\boldsymbol{F}$: $\{\boldsymbol{f}_i, \forall i \in \mathcal{V}\}$ based on each edge-integrated feature matrix $\boldsymbol{M}^d$: $\{\boldsymbol{m}_i^d, \forall i \in \mathcal{V}\}$ in **merge layer**.



Merge Layer:

$$\boldsymbol{f}_i = \|_{d=1}^{D}(\boldsymbol{m}_i^d), \quad \forall i \in \mathcal{V}.$$

Then, a multilayer perceptron module (*MLP*) is used to predict the cell slew and delay in PBA mode. The minimizing Mean-Squared Error (MSE) between the predicted and the PBA result is taken as the loss function.

$$\mathcal{L}_{\text{slew}}(\boldsymbol{\theta} \mid \boldsymbol{F}, S_{\text{r}}^{\text{PBA}}) = \frac{1}{N} \sum_{i \in \mathcal{V}} (S_{\text{i}}^{\text{PBA}} - S_{\text{r\_i}}^{\text{PBA}})^2.$$

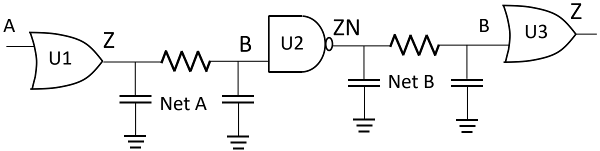
$$\mathcal{L}_{\text{delay}}(\boldsymbol{\phi} \mid \{\boldsymbol{F}, S_{\text{cell}}^{\text{PBA}}\}, D_{\text{r}}^{\text{PBA}}) = \frac{1}{N} \sum_{i \in \mathcal{V}} (D_{\text{i}}^{\text{PBA}} - D_{\text{r\_i}}^{\text{PBA}})^2.$$

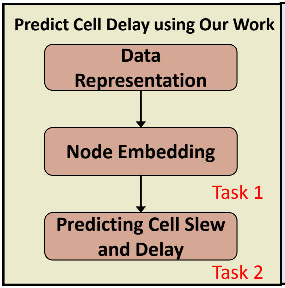
$$\mathcal{L}_{\text{tot}}(\boldsymbol{\theta}, \boldsymbol{\phi} \mid \{\boldsymbol{F}, S_{\text{cell}}^{\text{PBA}}\}, S_{\text{r}}^{\text{PBA}}, D_{\text{r}}^{\text{PBA}}) = \mathcal{L}_{\text{slew}} + \mathcal{L}_{\text{delay}}.$$

Task 3: Calculation

PBA arrival time of a critical path $AT_{\mathrm{CP}}^{\mathrm{PBA}}$ is estimated by the predicted PBA cell delay $D_{\mathrm{cell}}^{\mathrm{PBA}}$ and GBA wire delay $D_{\mathrm{wire}}^{\mathrm{GBA}}$.

$$AT_{\mathrm{CP}}^{\mathrm{PBA}} = \sum_{\mathrm{i} \in \mathcal{V}_{\mathrm{CP}}} D_{\mathrm{i}}^{\mathrm{PBA}} + \sum_{\mathrm{p} \in \mathcal{E}_{\mathrm{CP}}} D_{\mathrm{p}}^{\mathrm{GBA}}.$$





Collect Net Delay From GBA Results

Point	Trans	Incr	Path
U0_reg/CP	0.00000	0.00000	0.00000
netA	0.00042	0.00012	0.00012
U0_reg/Q	0.02015	0.05688	0.05688
U1/Z	0.03045	0.02473	0.08161
U2/Z	0.01066	0.02080	0.10241
Data arrival time			0.10241

Algorithm 1 summarizes the overall training process of PBA cell slew/delay predictor. We leverage a parallel training scheme by partitioning critical paths over multi-GPUs.

Algorithm 1 Training Methodology.

Input: Edge-featured graph: $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \boldsymbol{X}, \boldsymbol{H}\}$; Node feature matrix: $\boldsymbol{X}$: $\{\boldsymbol{x}_i, \forall i \in \mathcal{V}\}$; Edge feature matrix: $\boldsymbol{H}$: $\{\boldsymbol{h}_p, \forall p \in \mathcal{E}\}$; Real PBA cell slew $S_{\mathrm{r}}^{\mathrm{PBA}}$ and delay $D_{\mathrm{r}}^{\mathrm{PBA}}$; Search depth D=100; Parameters in the LeakyReLU nonlinear function.

Output: Trainable parameters $\boldsymbol{W}$: $\{\boldsymbol{W}_X^d$ and $\boldsymbol{W}_H^d, \forall d \in \{1, ..., D\}\}$ in EdgeGAT layers; θ and ϕ in MLP

1: **for** $i \in \mathcal{V}$ **do**
2: $\quad \boldsymbol{f}_i \leftarrow \|_{d=1}^{D} \left(\boldsymbol{m}_i^d\right)$; ▷ Node embedding
3: $\quad S_{\mathrm{i}}^{\mathrm{PBA}} \leftarrow MLP(\boldsymbol{\theta} \mid \boldsymbol{F})$; ▷ Predicting cell slew
4: $\quad D_{\mathrm{i}}^{\mathrm{PBA}} \leftarrow MLP(\boldsymbol{\phi} \mid \boldsymbol{F}, S_{\mathrm{i}}^{\mathrm{PBA}})$; ▷ Predicting cell delay
5: **end for**
6: Compute $\mathcal{L}_{\mathrm{tot}}$;
7: Minimize $\mathcal{L}_{\mathrm{tot}}$ via Adam and update all parameters $\boldsymbol{W}$

Experimental Results

Experimental Settings

- Training Device: a Linux machine with 32 cores and 4 NVIDIA Tesla V100 GPUs in parallel with 128GB memory.
- PBAGBA Device: a 72-core 2.6GHz Linux machine with 1024 GB memory
- Benchmarks: 18 open-source circuits with TSMC28nm

	Benchmark	#Cells	#Nets	#FFs	#CPs
Train	PCI_BRIDGE	1234	1598	310	456
	DMA	10215	10898	1956	1475
	B19	33785	34399	3420	5093
	SALSA	52895	57737	7836	9648
	RocketCore	90859	93812	16784	12475
	VGA_LCD	56194	56279	17054	8761
	ECG	84127	85058	14,018	13189
	TATE	184601	185379	31,409	27931
	JPEG	219064	231934	37,642	36489
	NETCARD	316137	317974	87,317	46713
	LEON3MP	341000	341263	108,724	50716
	Total	1390111	1075068	326470	212766
Test	WB_DMA	40962	40664	718	9619
	LDPC	39377	42018	2048	7613
	DES_PERT	48289	48523	2983	10976
	AES-128	113168	90905	10686	24973
	TV_CORE	207414	189262	40681	33706
	NOVA	141990	139224	30494	39341
	OPENGFX	219064	231934	37,642	47831
	Total	810264	782530	125252	221890

Results: Cell Slew/Delay Prediction Accuracy

Benchmark	Cell Slew/Delay Prediction Accuracy (R^2 score)					
	MLP	GCNII[1]	GraphSage[2]	GAT[3]	EGNN[4]	Deep EdgeGAT
WB_DMA	0.795/0.761	0.875/0.861	0.881/0.846	0.883/0.876	0.915/0.907	**0.996/0.971**
LDPC	0.762/0.732	0.842/0.832	0.865/0.814	0.877/0.871	0.921/0.916	**0.991/0.987**
DES_PERT	0.766/0.727	0.896/0.887	0.847/0.826	0.906/0.900	0.963/0.960	**0.989/0.987**
AES-128	0.731/0.712	0.801/0.792	0.821/0.810	0.856/0.816	0.938/0.921	**0.977/0.970**
TV_CORE	0.756/0.717	0.838/0.817	0.847/0.837	0.856/0.844	0.957/0.944	**0.982/0.979**
NOVA	0.725/0.718	0.826/0.812	0.824/0.818	0.864/0.855	0.905/0.871	**0.974/0.971**
OPENGFX	0.699/0.681	0.819/0.802	0.809/0.798	0.834/0.816	0.862/0.840	**0.982/0.974**
Average	0.748/0.721	0.843/0.829	0.842/0.821	0.868/0.854	0.923/0.909	**0.984/0.977**

- Ours outperforms GCNII by 0.142/0.147, GraphSage by 0.141/0.156, GAT by 0.116/0.123 and EGNN by 0.062/0.069.

[1]Simple and deep graph convolutional networks [2]Inductive representation learning on large graphs
[3]Graph attention networks [4]Exploiting edge features for graph neural networks

Results: Path Delay Prediction Accuracy

Benchmark	Path Delay Prediction Accuracy: R^2 score / MAE(ps)						Runtime(s)				
	STA Tool (PrimeTime)		Prior Work	Ours			PBA		Ours		Comparison
	PBA	GBA	CART[1]	*D*=25	*D*=50	*D*=100	Full	GBA	Predictor	Total	Speedup
WB_DMA	1.000/0.00	0.549/64.91	0.732/21.34	0.881/10.74	0.928/3.23	**0.998/0.89**	276.7	12.1	**1.197**	13.297	**20.81**×
PCI_BRIDGE	1.000/0.00	0.471/89.23	0.694/41.01	0.896/14.65	0.901/9.51	**0.993/1.46**	365.9	15.3	**0.798**	16.098	**22.73**×
DES_PERT	1.000/0.00	0.452/50.84	0.702/37.86	0.891/25.17	0.931/10.92	**0.997/1.02**	386.3	16.4	**1.614**	18.014	**21.44**×
AES-256	1.000/0.00	0.393/130.92	0.511/80.75	0.702/22.94	0.822/9.37	**0.977/3.94**	593.7	31.2	**2.731**	33.931	**17.50**×
TV_CORE	1.000/0.00	0.424/91.27	0.651/57.93	0.825/29.36	0.897/19.34	**0.984/6.81**	614.6	22.1	**2.410**	24.51	**25.08**×
NOVA	1.000/0.00	0.419/88.64	0.673/36.59	0.839/23.83	0.904/14.37	**0.983/4.11**	1133.8	30.5	**4.276**	34.776	**32.60**×
OPENGFX	1.000/0.00	0.378/267.91	0.571/147.03	0.793/53.74	0.851/27.89	**0.987/5.84**	1185.4	36.3	**4.432**	40.732	**29.10**×
Average	1.000/0.00	0.441/111.96	0.647/60.36	0.832/25.78	0.891/13.52	**0.988/3.44**	642.3	23.4	**2.494**	25.894	**24.80**×

- According to the R^2 scores, the accuracy of our work reaches 0.832, 0.891, and 0.988 on average when *D*=25,50 and 100. And the average maximum absolute error of our results is just 3.44ps.
- the average runtime of our workflow to get accurate PBA timing results costs 25.894s, which achieves 24.80× speedup compared with PrimeTime.

[1] Using machine learning to predict path-based slack from graph-based timing analysis

Conclusion

Conclusion

- Using GBA results to predict PBA makes a tradeoff between accuracy and runtime.
- Our predictor has the potential to substantially predict PBA timing results accurately. According to the R^2 scores, the accuracy of our work reaches 0.988 on average with maximum error reaching 6.81 ps.
- Our work accelerates PBA timing results which achieves an average 24.80× speedup faster than PBA using the commercial STA tool.

THANK YOU!