Graph Learning-Based Arithmetic Block Identification

Zhuolun He[1], Ziyi Wang[1], Chen Bai[1], Haoyu Yang[2], Bei Yu[1]

[1]The Chinese University of Hong Kong
[2]NVIDIA
{zlhe,byu}@cse.cuhk.edu.hk

Nov. 1, 2021

Outline

1. Introduction
2. Designing Graph Neural Networks for DAG
3. Netflow for Input-Output Matching
4. Experimental Results

Introduction

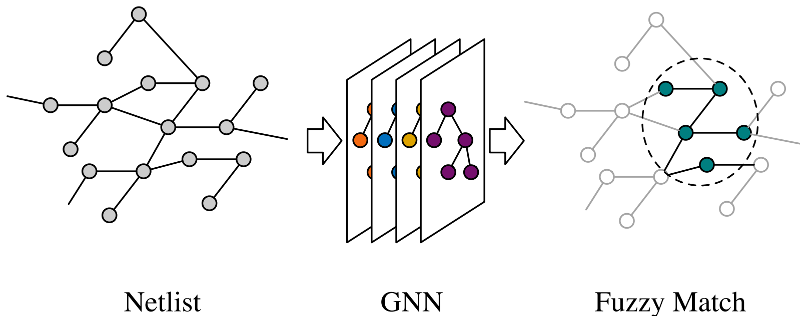
Arithmetic Block Identification

- Identify arithmetic blocks in gate-level netlists
- Lots of applications
  - Functional verification [ICCAD'18, DAC'19]
  - Logic optimization [DATE'15]
  - Malicious logic detection [IDTC'10, ISTFA'16, DAC'19]
- In this work, we focus on *adder* identification

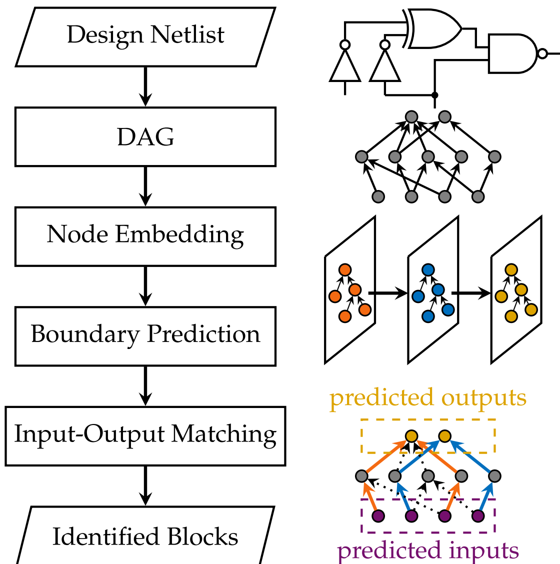
Related Approaches

- Structural methods
  - Concentrate on circuit topology
  - ☺ Efficient with customized algorithms
  - ☹ Often mathematically incomplete
- Functional methods
  - Functionally analyze the circuit for potential arithmetic components
  - ☺ Accurate and solver-ready
  - ☹ Ultra-long runtime
- Machine learning methods
  - Alternate solutions to recognition and classification
  - ☺ Dedicated to one given unknown functional block
  - ☹ Facing significant challenges dealing with large-scale netlists

We propose a graph learning-based arithmetic block identification framework



Overall Flow



Designing Graph Neural Networks for DAG

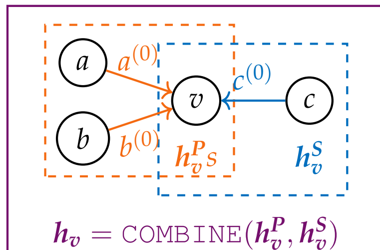
- Enable powerful representation learning on graphs
- Follow a neighbor aggregation scheme: node embeddings are computed by recursively aggregating and transforming embeddings of neighboring nodes

$$a_v^{(k)} = \texttt{AGGREGATE}(\{h_u^{(k-1)} : u \in \mathcal{N}(v)\}),$$
$$h_v^{(k)} = \texttt{COMBINE}(a_v^{(k)}, h_v^{(k-1)})$$

- Not customized for any specific task

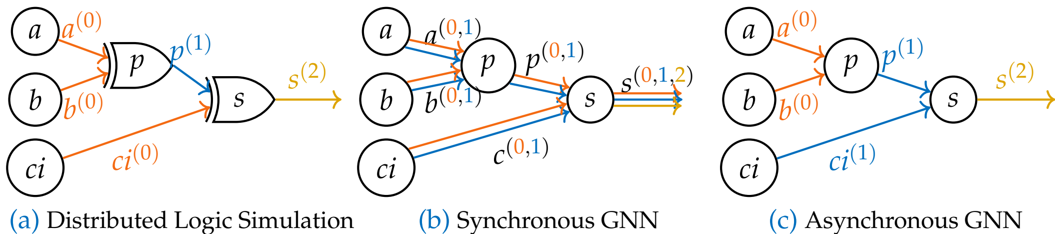
Question: How to design a better GNN architecture to encode DAGs?

- DAGs are directed
- Motivation: encode information from both directions
- Train two GNNs, one for $\mathcal{G}$ and one for $\mathcal{G}^\top$. In other words, one aggregates information from predecessors and the other from successors.
- Combine the two embeddings as the final embedding.



Asynchronous Graph Neural Network

- DAGs are acyclic
- Motivation: improve GNN efficiency utilizing the acyclic property
- Resembling distributed logic simulation, asynchronous message passing starts from the leaf nodes of the fanin cone and all the way up to the target node



(a) Distributed Logic Simulation (b) Synchronous GNN (c) Asynchronous GNN

We propose two architectural structures

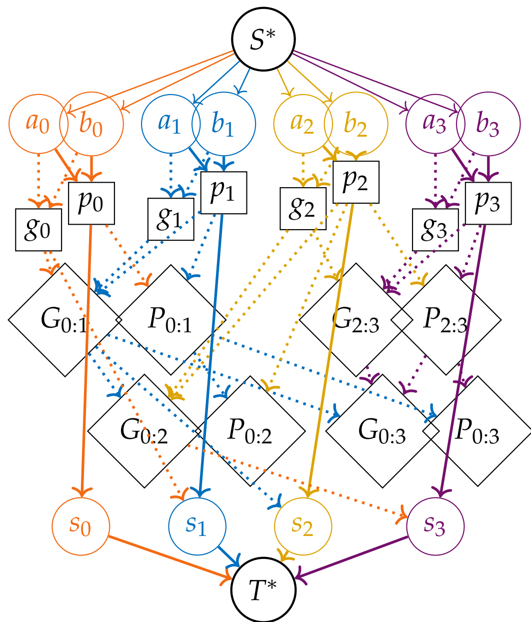
- bidirectional GNN for directed graphs
- asynchronous GNN for acyclic graphs

We combine them in our final architecture, asynchronous bidirectional graph neural network (ABGNN), which is customized for DAG embedding.

Netflow for Input-Output Matching

- After identifying adder boundaries, we further want to match the input bits S with the corresponding output bits T.
- We formulate a maximum flow problem to find the routes between inputs and outputs
  - Add a pseudo source node $S*$ and a pseudo sink node $T*$ in the graph
  - Add edges from $S*$ to every node in S, as well as every node in T to $T*$
  - The newly added edges from $S*$ to nodes in S are assigned unit capacity
  - The rest edges are assigned capacity of 2

A Brent-Kung Adder Example



Experimental Results

Setup

- Developed the graph object detection framework in Python
  - Libraries: DGL, PyTorch, networkx
- Refer to EPFL logic synthesis libraries when implementing baseline methods
- Dataset: open-source RISC-V CPU designs
  - *BOOM* for training
  - *Rocket* for testing
  - Netlists generated by Chisel, synthesized with Synopsys Design Compiler
  - Synthesize various adder architectures for each design

Comparison with Baseline Methods

Case	TETC'13			DATE'15			DATE'19			Ours		
	Input	Output	Time(s)	Input	Output	Time(s)	Input	Output	Time(s)	Input	Output	Time(s)
Brent Kung	0.826	0.672	302.0	0.554	0.493	13.4	0.875±0.022	0.820±0.013	11.6±3.9	**0.950±0.000**	**0.954±0.020**	**10.2±1.8**
Cond-sum	0.825	0.598	380.6	0.770	0.787	14.6	0.808±0.013	0.744±0.020	13.0±3.7	**0.949±0.000**	**0.866±0.014**	**10.9±0.6**
Hybrid	0.815	0.389	597.2	0.179	0.042	15.4	0.820±0.032	0.699±0.026	15.1±5.1	**0.947±0.000**	**0.957±0.018**	**12.0±0.7**
Kogge-Stone	0.823	0.648	525.2	0.755	0.783	15.8	0.763±0.015	0.810±0.011	13.2±3.5	**0.944±0.000**	**0.961±0.010**	**11.0±0.9**
Ling	0.803	0.456	315.6	0.249	0.022	16.5	0.874±0.013	0.653±0.074	16.3±5.5	**0.954±0.000**	**0.944±0.015**	**13.2±0.9**
Sklansky	0.823	0.626	467.4	0.484	0.483	14.7	0.864±0.017	0.845±0.017	14.1±3.7	**0.960±0.000**	**0.938±0.010**	**11.9±0.5**
Average	0.819	0.565	431.3	0.499	0.435	15.1	0.834±0.019	0.761±0.027	13.9±4.2	**0.951±0.000**	**0.937±0.015**	**11.5±0.9**

- Our proposed method greatly outperforms prior works on all the testcases

Comparison with State-of-the-Art GNNs

- We evaluate our proposed ABGNN with several state-of-the-art Graph Neural Networks, including GAT, GIN, and GraphSAGE
- Our model achieves the best performance on all the cases with much higher recall and F_1 scores, showing its superiority on DAG representation learning
- Up to 6.2% Recall gain
- Up to 9.5% F_1 score gain

Ablation Study

- Asynchronous GNN reduces runtime with no accuracy degradation:

Task	Model	Recall	F_1-score	Runtime (ms)
Input	asynchronous	**0.951±0.000**	**0.956±0.000**	**122.1**
	synchronous	0.943±0.003	0.951±0.002	152.2
Output	asynchronous	**0.937±0.015**	**0.940±0.012**	**77.6**
	synchronous	0.933±0.012	0.937±0.009	94.6

- Bidirectional GNN improves performance:

Task	Model	Recall	F_1-score
Input	bidirectional	**0.951±0.000**	**0.956±0.000**
	unidirectional	0.933±0.002	0.935±0.002
Output	bidirectional	**0.937±0.015**	**0.940±0.012**
	unidirectional	0.891±0.001	0.829±0.011

Conclusion

- Identifying arithmetic blocks is a vital procedure for various tasks
- In this paper, we proposed:
  - a graph learning-based framework for efficient arithmetic block recognition
  - a specialized GNN for DAG representation learning
  - a network flow approach to match input and output wires predicted by the GNN model
- We conducted comprehensive experiments on open-source RISC-V CPU designs to evaluate our methods

THANK YOU!