Efficient Arithmetic Block Identification with Graph Learning and Network-flow

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

***Abstract*—Arithmetic block identification in gate-level netlists plays an essential role for various purposes, including malicious logic detection, functional verification, or macro-block optimization. However, current methods usually suffer from either low performance or poor scalability. To address the issue, we come up with a novel framework based on graph learning and network flow analysis, that extracts desired logic components from a complete circuit netlist. We design a novel asynchronous bidirectional graph neural network (ABGNN) dedicated to representation learning on directed acyclic graphs. In addition, we develop a convex cost network-flow-based datapath extraction approach to match the predicted block inputs with predicted block outputs. Experimental results on open-source RISC-V CPU designs demonstrate that our proposed solution significantly outperforms several state-of-the-art arithmetic block identification flows.**

I. Introduction

ARITHMETIC block identification in gate-level netlists has emerged as an essential procedure for numerous datapath optimization or functional verification methodologies. For example, in Symbolic Computer Algebra (SCA) based multiplier verification [2], [3], it is required to detect all the half adders from the multiplier netlist. Another use case is discussed by Wei *et al.* [4], where a detected arithmetic block can be replaced by more advanced intellectual property (IP) macros. Moreover, the demand for hardware Trojan detection has been pointed out in several papers [5]–[7] to ensure circuit security and authenticity, especially under the globalization of the semiconductor design and fabrication process. Aside from the applications mentioned above, there is an additional technical reason behind the need for such a 'reverse engineering' approach: after logic synthesis and technology mapping, most high-level components (e.g., function declaration, modules, etc.) are flattened into netlists of Boolean gates, as mentioned in [8]. Given all the facts stated above, arithmetic block identification is indeed worth exploring.

Traditional methods for arithmetic block identification are usually classified as either *functional methods* [9], [10] or *structural methods* [6], [11]–[14]. **Structural methods** focus on circuit topology while omitting the circuit functionality [15]. For instance, Li *et al.* [13] have introduced the *shape hashing* technique to generate candidate words by clustering wires with similar local topology. To be more specific, a k-hop depth-first search is executed on the graph, starting from each wire, where the serialization of a wire is constructed using the wire and cell information on the search path. Some studies consider a different scenario in which a reference (golden) library of circuits is provided, reducing the problem to matching subcircuits with pre-defined pattern circuits. Rubanov *et al.* [12] formulate the subcircuit matching problem as a regularized quadratic assignment problem (QAP) to simultaneously minimize both graph distance and vertex label distance, which is solved by a nonlinear version of the iterative Kaczmarz Method (KM). Structural methods can usually detect target components efficiently because of the customized algorithms. However, the heuristic methodology also indicates their mathematical incompleteness [1]. On the other hand, **functional methods** inspects the circuit functionally to look for target arithmetic blocks. Subramanyan *et al.* [9] have built a functional approach upon the above shape hashing method by using cut enumeration. They enumerate all 6-feasible cuts and then group equivalent cuts with permutation-independent Boolean matching. In this way, all the cuts within the same equivalence class are likely to match the same function from the given library. The authors [9] have further proposed to formulate the module matching problem as an equivalence checking problem, with the help of Quantified Boolean Formula (QBF). To conclude, functional methods are accurate and solver-ready, but usually at the cost of ultra-long runtime [1].

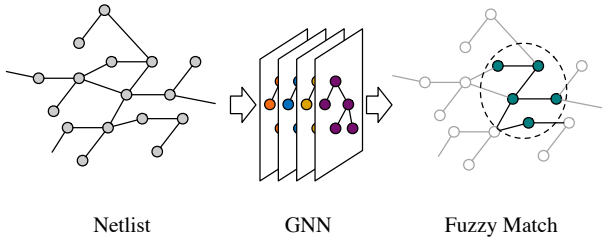


Fig. 1 Graph learning enables netlist fuzzy matching.

Recent advances in **machine learning**, especially deep learning, have offered new ideas for solving recognition problems. Silva *et al.*. [16] developed a flow that converts conjunctive normal form (CNF) clauses into images, which are later rescaled to the target size and fed into the deep neural network classifiers. Fayyazi *et al.*. [17] proposed a special data structure, termed level-dependent decaying sum (LDDS) existence vector, to compactly represent circuit topology. The

† Equal contributors

The preliminary version has been presented at the IEEE/ACM International Conference on Computer-Aided Design (ICCAD) in 2021 [1]. (*Corresponding authors: Bei Yu*)

Z. Wang, Z. He, C. Bai and B. Yu are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong SAR.

H. Yang is with nVIDIA.

existence vector encodes each circuit vertex using all its neighboring vertices. However, these solutions only work on small-scale circuits and suffer from poor scalability.

To address the above concerns, we propose a graph learning-based arithmetic block identification framework, as illustrated in Fig. 1. The framework can conduct efficient arithmetic block fuzzy matching. We choose graph neural networks (GNNs) as the preferable fuzzy matching technique since a netlist can be naturally converted into a directed acyclic graph (DAG). Intuitively, GNNs aggregate information from neighborhoods to automatically generate proper embeddings for each node, which are helpful in downstream tasks. However, most existing popular GNN models, such as GraphSAGE [18] and GIN [19], are designed to deal with general graphs or undirected graphs. To say in other words, they are not well-optimized for DAGs. Meanwhile, several works have been proposed that explored customized GNN for DAGs and have achieved state-of-the-art performance in many DAG tasks, e.g., testing point insertion, neural architecture exploration, high-level delay prediction [20]–[25]. Nevertheless, they are not targeted at or well-suited for arithmetic block identification. Therefore, we further design a new variant of GNN, *asynchronous bidirectional graph neural network* (**ABGNN**), that is customized for netlist embedding with excellent performance as well as high efficiency.

The paper makes the following contributions:

- For the first time, to the best of our knowledge, we present a graph learning-based framework that performs efficient fuzzy matching on arithmetic blocks;
- We design a novel GNN architecture customized for netlist representation learning;
- We analyze several practical challenges and develop a convex cost network-flow-based approach that matches the predicted inputs with outputs.
- We conduct experiments on open-source RISC-V CPU designs synthesized by industrial tools, which confirms the effectiveness and efficiency of our proposed framework compared with other state-of-the-art macro block detection solutions.
- We also carried out a comprehensive ablation study to analyze the effectiveness of the proposed techniques.

II. Preliminary

A. Related Works to DAG Embedding

Recently, there have emerged several works that aim to develop graph learning models for DAG. DAGNN [20] builds a multi-layer neural network that produces a representation for a DAG driven by the partial order. Ma et al. [21] first applied GNNs to DAGs in Electronic Design Automation (EDA). It takes a simple strategy to deal with the directional information, assigning different weights to predecessors and successors in the aggregation function. D-SAGE [25] treats the DAGs as heterogeneous graphs and applies separate aggregators to collect information from the two directions. Based on this unique architecture, D-SAGE manages to extract directional information and achieved state-of-the-art performance in operation delay prediction for FPGA HLS. However, the previous works do not account for direction when sampling neighborhoods, resulting in identical search depth in both directions, which is sub-optimal for arithmetic block identification since fanin and fanout directions are not equally important (details in Section IV-C). Besides, they follow a synchronous message passing scheme, which may not be compatible with netlists. In general, these works explored GNNs tailored for DAGs, but they are not aimed at or well-optimized for netlists and arithmetic block identification. This motivates us to design a GNN architecture customized for netlists.

B. Problem Definition

The *gate-level netlist* of an electric circuit consists of a list of gate-level circuit components (e.g., `AND` gates) and their interconnects. Gate-level netlists are generated by logic synthesis tools, which convert the behavior specification of a circuit into logic gate implementation. Mathematically, a gate-level netlist can be naturally represented as a directed acyclic graph, with vertices representing circuit components and edges representing wires between them. We say a gate-level netlist is *flattened* if only primitive gates are instanced, while the design hierarchy is unknown. Within a netlist, arithmetic blocks are the building blocks that perform simple arithmetic operations, such as integer addition or multiplication. The input boundary nodes of an arithmetic block are defined as the gates whose output wire is an input to the block. Similarly, we can define the output boundary nodes. In general, our goal is to identify the arithmetic blocks located in a flattened netlist.

Problem 1 (Arithmetic Block Identification)**.** *Given a flattened gate-level netlist, identify the target arithmetic blocks located in the netlist., e.g., adders, multipliers, etc. To be more specific, identify the boundary nodes of the target blocks.*

III. Flow Overview

Before diving into algorithmic details, we first provide a high-level overview of our proposed arithmetic block identification flow. Given a design netlist, we begin by converting it into a directed acyclic graph (DAG). The DAG is fed to our designed ABGNN (introduced in Section IV) to generate node embeddings. The node embeddings are further used to predict arithmetic block boundary (introduced in Section IV-A). Then, we run a network flow-based algorithm (introduced in Section V) to match the predicted input boundary nodes with the predicted outputs boundary nodes. We illustrate the overall flow in Fig. 2.

IV. Designing Graph Neural Network for DAGs

Graph neural networks (GNNs) have emerged as a promising approach for graph analysis. They follow an iterative neighborhood aggregation scheme to capture the structural information within nodes' neighborhoods. GNNs have achieved state-of-the-art performance on a variety of graph tasks, such

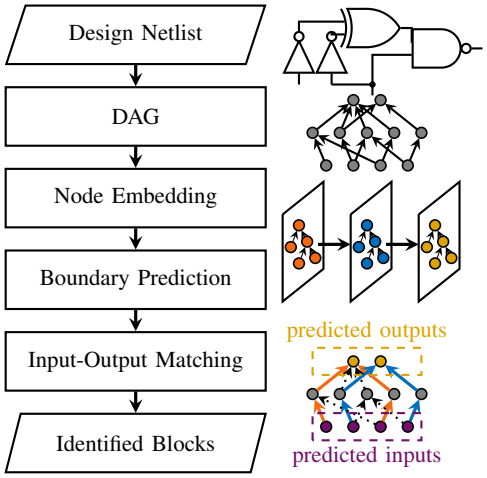


Fig. 2 Our arithmetic block identification flow.

as node classification, link prediction, and graph classification. Nonetheless, it is still critical to customize graph neural network architecture according to the actual task to earn the best result. This section discusses how we design a novel graph neural network architecture dedicated to DAG representation learning in our arithmetic block IO boundary prediction task.

A. Machine Learning Target Formulation

We begin with the discussion on target formulation. Essentially, the arithmetic block identification problem is to 'detect' instances of objects with target semantics in the graph, which sounds like a graph version of the *object detection* task in computer vision. A very related problem is subgraph matching, which looks for a subgraph in a large target graph that is isomorphic to the query graph. Despite the intuitive descriptions, solving such problems is indeed very challenging for the community due to **1)** the NP-complete nature of the problem and **2)** the requirement to consider graph topology, node features, and/or edge features at once. Although some techniques have been proposed to tackle subgraph matching, based on either combinatorial search (e.g., VF2 [26], RI [27]) or neural networks (e.g., GMNN [28], NeuroMatch [29]), we are not aware of any effective method to directly deal with the '*graph detection*' problem.

Given that, we propose to formulate a *node classification* problem to circumvent the hard-to-solve graph detection problem. Specifically, our neural model targets to **predict the boundary of arithmetic blocks**, namely, to predict input/output nodes of target blocks. Another alternative problem formulation is to predict the region covered by the arithmetic blocks, which is inferior, as will be demonstrated in Section VI-E. Note that a wire can be both an input to one arithmetic block and an output from another arithmetic block (consider the two consecutive expressions $c = a + b$ and $e = c + d$, where c is the output of the first adder and the input of the second adder). Therefore, we split the boundary prediction problem into two separate binary classification tasks, namely *input prediction* and *output prediction*. We train a feed-forward neural network to consume the representation vectors generated by GNN and carry out the prediction. In binary classification, we use binary cross entropy as the loss function:

$$\mathcal{L} = -y \log(p) - (1-y) \log(1-p), \tag{1}$$

where y is the ground truth of the prediction, and p is the prediction of the model.

B. General Graph Neural Network

Before diving into the technical details of our ABGNN model, we deliver a formal introduction to general graph neural networks, partly following the notations in GIN [19]. Let $G = \langle \mathcal{V}, \mathcal{E} \rangle$ denote a graph, where $\mathcal{V} = \{v_1, v_2, \cdots, v_n\}$ is the vertex set, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the edge set. Considering a K-layer GNN, the propagation of the k-th layer is represented as

$$\begin{aligned} \boldsymbol{a}_v^{(k)} &= \texttt{AGGREGATE}(\{\boldsymbol{h}_u^{(k-1)} : u \in \mathcal{N}(v)\}), \\ \boldsymbol{h}_v^{(k)} &= \texttt{COMBINE}(\boldsymbol{a}_v^{(k)}, \boldsymbol{h}_v^{(k-1)}), \end{aligned}$$

where $\boldsymbol{h}_v^{(k)}$ is the representation vector of vertex v after k iterations, $\boldsymbol{h}_v^{(0)} = \boldsymbol{x}_v$ (x_v is the initial node feature of v), and $\mathcal{N}(v)$ denotes the neighbouring nodes of v. Many GNN variants with different choices of `AGGREGATE` function (e.g., mean, sum, etc.) and `COMBINE` are proposed, which are crucial to the model performance. The expressive power of such GNNs is theoretically proved [19] to be upper-bounded by the Weisfeiler-Lehman graph isomorphism test, which is achieved when both `AGGREGATE` and `COMBINE` are injective functions over multisets.

C. Bidirectional Graph Neural Network

We first deal with the '*directed*' property of '*Directed Acyclic Graphs*'. Each edge in a directed graph is assigned a two-way direction, which naturally captures various real-life relations. In our netlist, the edge direction represents the current flow direction. In other words, it indicates the execution order of the circuit. Therefore, it is intrinsic to represent a netlist as a directed graph.

However, most existing GNN models are dedicated to undirected graphs. One historical reason is due to earlier spectral GNN models [30]–[32] built upon the analogy to Convolutional Neural Networks (CNNs). In spectral GNN models, a *graph convolution* is defined as the multiplication of a signal $\boldsymbol{x} \in \mathbb{R}^N$ with a filter $\boldsymbol{g}_\theta = \text{diag}(\boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta} \in \mathbb{R}^N$ in the Fourier domain, namely:

$$\boldsymbol{g}_\theta \star \boldsymbol{x} = \boldsymbol{U} \boldsymbol{g}_\theta \boldsymbol{U}^\top \boldsymbol{x},$$

where $\boldsymbol{U}$ is the matrix of eigenvectors of the normalized graph Laplacian $\boldsymbol{L} = \boldsymbol{I}_N - \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{A} \boldsymbol{D}^{-\frac{1}{2}} = \boldsymbol{U} \boldsymbol{\Lambda} \boldsymbol{U}^\top$. In this definition, $\boldsymbol{U}^\top \boldsymbol{x}$ is considered the *graph Fourier transform* of $\boldsymbol{x}$, which relies on the fact that the (real symmetric) normalized graph Laplacian $\boldsymbol{L}$ admits an eigendecomposition. Unfortunately, this property does not hold for a directed graph. One straightforward way is to relax the directed graph to an undirected graph by symmetrizing its adjacency matrix, which inevitably results in information loss.

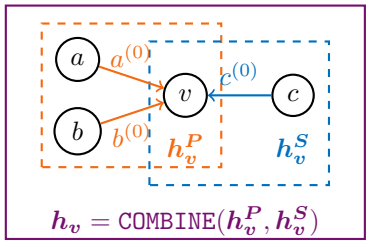


Fig. 3 Bidirectional information aggregation for the vertex v. Two separate GNNs are utilized to aggregate information from the fanin cone ($\boldsymbol{h}_v^P$, in orange) and the fanout cone ($\boldsymbol{h}_v^S$, in blue), respectively. The final embedding ($\boldsymbol{h}_v$, in purple) is given by the combination of the representation vectors from both directions.

Our designed bidirectional GNN is greatly motivated by the design of heterogeneous GNNs [33], [34]. As discussed in the previous work [33], one of the challenges in designing heterogeneous GNN is 'how to aggregate feature information of heterogeneous neighbors by considering the impacts of different node types'. In arithmetic block identification, the role of a gate depends on both its fanin and fanout cones. Therefore, combining information from both directions is required to generate representative node embeddings. Furthermore, the fanin and fanout cones are not of equal importance in arithmetic block identification. For instance, the fanin cone rooted at a node n, which may contain structural information of a target block, is unquestionably more useful in determining whether n is an output boundary node. On the other hand, input boundary detection might be dominated by fanout information. This puts the demand to decouple the neighborhood to achieve optimal search depth in both directions. In particular, information from the two directions should be collected and aggregated separately, with different search depths (number of GNN layers). Hereafter, we denote the *transpose graph* as $\mathcal{G}^\top$, which contains a directed edge (u, v) if and only if $\mathcal{G}$ contains the reversed edge (v, u).

To encode the edge directions and fully decouple the bidirectional neighborhood, each vertex **only aggregates information from its *predecessors***. In other words, information flows from a node x to a node y only if there is an edge (x, y). To realize bidirectional information aggregation, we apply two GNNs, one for $\mathcal{G}$ and one for the transpose graph $\mathcal{G}^\top$, to generate two embedding vectors $\boldsymbol{h}_v^P$ and $\boldsymbol{h}_v^S$ for each vertex. Ideally, the two embedding vectors collect information from the predecessors (i.e., fanin cone) and the successors (i.e., fanout cone), separately. Thus, the final embedding of each vertex is given by the combination of both $\boldsymbol{h}_v^P$ and $\boldsymbol{h}_v^S$:

$$\boldsymbol{h}_v = \texttt{COMBINE}(\boldsymbol{h}_v^P, \boldsymbol{h}_v^S) \tag{2}$$

The placeholder `COMBINE` can be any common reduction function such as `mean`, `max`, or `sum`. In our practice, we simply concatenate the two vectors for the final embedding. We illustrate the bidirectional information aggregation scheme in Fig. 3.

D. Asynchronous Graph Neural Network

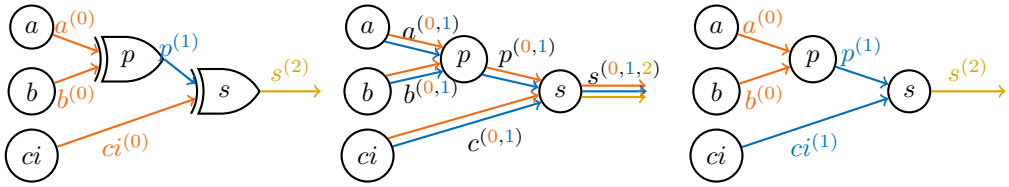
We move to the '*Acyclic*' property of '*Directed Acyclic Graphs*', which by definition contain no cycles. That is, if we start from any vertex v, walking through the graph along the edge directions, we will never come back to v. Although it sounds irrelevant to GNN design, we now demonstrate the possibility of improving GNN efficiency by utilizing the acyclic property.

We begin with an analogy to event-driven logic simulation, using the Chandy-Misra-Bryant (CMB) distributed-time algorithm [35] as an example. To enable parallel logic simulation with the CMB algorithm, circuit elements exchanged timestamped messages, and different elements consume events simultaneously at distinct simulation times. Conceptually, each element receives timestamped messages from its predecessors and consumes the messages at the earliest timestamp whenever all predecessors are ready. As a result of consuming the messages, the logic element updates its own local time and delivers one or more timestamped messages to its successors [36]. Fig. 4(a) illustrates the event message scheme assuming a unit delay for each gate. At timestamp 0, the primary inputs, a, b, and ci are ready, which triggers the execution of gate p since both its inputs are ready. After a unit delay, gate p sends out its message, which then triggers the execution of gate s together with ci. Similarly, gate s processes its inputs and sends out its output at timestamp 2. The original CMB algorithm is regarded as '*an approach to carry out asynchronous, distributed simulation on multiprocessor message-passing architectures*' [35].

On the contrary, typical GNNs work in a *synchronous* way. In a synchronous message passing scheme, all messages flow on edges simultaneously in each iteration, such that every vertex receives messages from its neighbors and updates its representation in every iteration. The message passing process in general synchronous GNNs is depicted in Fig. 4(b). We can see that all the nodes send out messages to their successors in both iterations 0 and 1, resulting in high computational costs.

Motivated by the CMB algorithm and the acyclic nature of the netlist, we propose an *asynchronous* GNN architecture, resembling the asynchronous message-passing scheme for logic simulation. To embed a target vertex v, consider its fanin cone rooted at v. The message passing process begins at the cone's leaf nodes and proceeds through the cone to v. At each 'timestamp', (i.e., each iteration of GNN message passing), only the vertices that received messages at the previous timestamp deliver messages to their direct successors. Fig. 4(c) shows an example to embed node s using such an asynchronous GNN. In iteration 0, only nodes a and b send out their messages to p, while in iteration 1, node p and node ci send out their messages to s. Obviously, asynchronous GNN executes as efficiently as logic simulation while being more efficient than synchronous message passing.

Formally, for a target vertex v, the aggregation scheme of the k-th iteration of a depth-Δ asynchronous GNN can be



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

Fig. 4 A comparison between (a) distributed logic simulation, (b) synchronous GNN message passing, and (c) asynchronous GNN message passing.

described as follows:

$$\begin{aligned} \boldsymbol{a}^{(k)}_{\{\mathbf{i}:\mathcal{D}(\mathbf{i},\mathbf{v})=\boldsymbol{\Delta}-\mathbf{k}\}} &= \texttt{AGGREGATE}(\{\boldsymbol{h}_u^{(k-1)} : u \in \mathcal{N}(i)\}), \\ \boldsymbol{h}^{(k)}_{\{\mathbf{i}:\mathcal{D}(\mathbf{i},\mathbf{v})=\boldsymbol{\Delta}-\mathbf{k}\}} &= \texttt{COMBINE}(\boldsymbol{a}_i^{(k)}, \boldsymbol{h}_i^{(\mathbf{0})}), \end{aligned} \tag{3}$$

where $\mathcal{D}(i, v)$ is the distance between vertices i and v in the graph, and $\boldsymbol{h}_i^{(0)}$ is the initial feature of vertex i. The boldface indices highlight the distinction between an asynchronous GNN and a general synchronous GNN. In other words, in the k-th iteration of a depth-Δ asynchronous GNN, only those vertices whose distance to the root v is $\Delta - k$ is *active* and aggregates information from its predecessors. Then, for each active node, the aggregated information is combined with its initial feature to update its own message, which will be sent to its predecessors in the next iteration. In this way, unlike synchronous GNNs, messages are passed through each edge exactly only once (in the embedding of each node), saving a significant amount of computational effort.

E. Dealing with Data Imbalance

The data imbalance issue refers to the phenomenon that the class distribution of a data set is biased. Imbalanced classifications pose a challenge for predictive modeling as most of the machine learning algorithms used for classification were designed around the assumption of an equal number of examples for each class [37]. For example, it is observed that the model would easily lean towards majority classes [38], invalidating some standard metrics like *accuracy* (since they may cause misinterpretation of data). We refer readers to [39] for a comprehensive review. In our dataset, the negative nodes account for over 99% of the total, revealing a severe imbalance.

Methods to address data imbalance can be divided into two categories, namely data-level methods and algorithm-level methods. Data-level methods seek to alter the distribution of the training dataset so that standard algorithms for balanced data can work well. Algorithm-level methods, on the other hand, keep the training dataset unchanged while adjusting the training/inference algorithm. We now introduce two techniques adopted in our training.

1) Oversampling: Oversampling is one of the most popular data-level methods used in machine learning. We adopt the basic version of it, called random minority oversampling, which supplements the training data with multiple copies of some of the minority classes [37]. Some more advanced oversampling methods (e.g., SMOTE [40]) have also been proposed, which we leave for possible future work. We do not favor the opposite method, undersampling of the majority class, because it discards a portion of available data.

2) Cost Sensitive Learning: Cost-sensitive learning [41] assigns different penalties to different types of misclassification errors. Mathematically, if C_{ij} refers to the cost for predicting class j when the actual class is i, the optimal prediction for an example x is given by

$$\operatorname*{argmin}_i \sum_j p(j|x) C_{ij},$$

where $p(j|x)$ is the estimated probability of example x being in class j.

We encode cost-sensitive learning into the loss function by decoupling the total loss $\mathcal{L}$ into two parts, namely the loss on the positive samples ($\mathcal{L}_{pos}$) and the loss on the negative samples ($\mathcal{L}_{neg}$). Since negative samples predominate, we **assign a penalty weight α ($\alpha < 1$) to the negative loss**, so that the contribution of negative nodes to the total loss function is reduced, which compensates for the imbalance between sample classes. The weighted loss function can be formulated explicitly as:

$$\mathcal{L} = (\mathcal{L}_{pos} + \alpha \mathcal{L}_{neg})/N, \tag{4}$$

where N is the total number of samples.

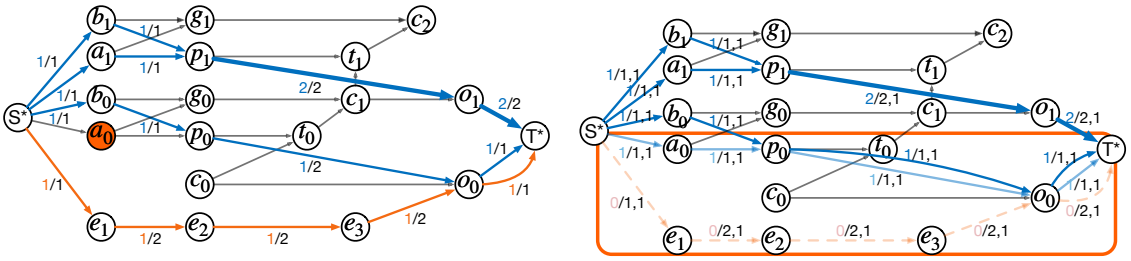
F. Putting It All Together

In previous subsections, we propose two unique GNN architectures, namely *bidirectional* and *asynchronous*, based on the directed and acyclic properties of the target graph (DAG), respectively. As the two structures are orthogonal, they can be combined in our final GNN architecture, asynchronous bidirectional graph neural network (ABGNN). We evaluate the performance of ABGNN in Section VI-E.

V. Input-output Matching

A. Network-flow-based Datapath Extraction

In the previous sections, we deal with the problem of block boundary detection, that is identifying the boundary nodes of target arithmetic blocks from a flattened netlist. Particularly, it predicts the input and output nodes of target blocks. In some applications, however, this is insufficient, and we need to further match the input bits with the corresponding output bits. In this section, we present a network-flow-based algorithm for extracting the datapaths within an arithmetic block that match



(a) Naive maximum flow might fail to handle the challenge of fake inputs. Here a fake input e_1 is involved.

(b) Compared to naive maximum flow, min-cost flow manages to deal with fake inputs. The bounding boxes highlight the comparison between the wrong(**dotted shallow orange**) paths and the correct paths (**shallow blue**). It is obvious that the correct paths have lower costs and are thus chosen by our min-cost flow.

Fig. 5 Illustration of fake inputs, using a 2-bit ripple carry adder $o_i = a_i + b_i$ as an example. The numbers 'v/cp, ct' above each flow edge give the value of flow going through it, its total capacity, and the cost of sending a unit of flow through it respectively. The **bold** edges indicate a flow of value **two** passing through it while there is a unit flow on the rest colored edges. The flow paths are denoted by orange or blue colors, where orange represents wrong paths and blue (either dark or shallow) denotes correct paths. Particularly, there is **no actual flow** passing through the **dotted** paths in Fig. 5(b). Input/output nodes that fail to be matched are marked with orange colors.

the block inputs with outputs. The problem has gained great attention under the name **'datapath extraction'**, since it is believed that datapath-aware physical synthesis may achieve higher performance. Readers are referred to the study [42] for a survey on datapath extraction approaches and datapath-driven placement methodologies. For now, we illustrate the feasibility of the network-flow approach for arithmetic block IO matching, and leave the other possible solutions for future work, as it is beyond the main scope of this paper.

Problem 2 (Block Input-output Matching)**.** *Given an (unordered) block input set* $S = \{a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1}\}$ *and an (unordered) block output set* $T = \{o_0, o_1, \cdots, o_{m-1}\}$ *such that* $T[m-1:0] = A[n-1:0] \ \Delta \ B[n-1:0]$*, where* Δ *is an arithmetic operation, e.g., addition, multiplication, etc. Our task is to identify datapaths from* S *and* T *such that* ***1)*** *all nodes in* S *and* T *are covered, and* ***2)*** *each datapath starts from* a_i *or* b_i *and ends at* o_j*, where* $0 \leq i, j \leq n-1$.

Once the datapaths have been extracted, experts can easily group the input/output nodes, since datapaths within the same arithmetic block are much closer and more related to one another than datapaths from different blocks.

Inspired by networkflow-based datapath bit slicing [43], we formulate the problem as a *maximum flow problem*. We add a pseudo *source* node S^* and a pseudo *sink* node T^* in the graph and 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, while the rest edges are assigned a capacity of 2 (unit capacity for multipliers). Then we run a *maximum-flow algorithm* to find the routes between S and T. Ideally, since the total input capacity equals the total output capacity, all the inputs and outputs can be matched. However, this does not necessarily hold given fuzzy, imperfect predictions of target block boundaries. In the following part, we discuss some practical challenges and propose customized techniques to address them.

Before diving into the technical details, we first introduce a hypothesis that will be applied to simplify the problem.

Hypothesis 1. *Given a true output node* $o_i \in S$*, the closest wire to* o_i *in the predicted input set are probably* a_j *or* b_j *that belongs to the same block. This can also be analogous to true input nodes.*

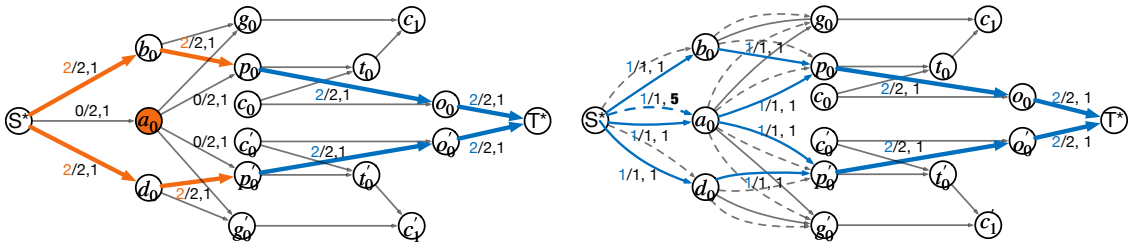
The hypothesis is based on the observation that the total target block regions occupy only a small portion of the whole circuit area. As a result, the fake input/output nodes are most likely not located within the target block regions. For any fake input wire e outside a target block b, the path from e to an output of b will first go through an input of b, implying that the outside fake input is farther to the block output than some block outputs. The same is true for fake output nodes.

To support the above hypothesis, we conduct some statistical analysis on our dataset. For each true block output $o_i \in T$, we calculate the distance (the length of the shortest path) from o_i to each predicted input wire. It is found that Hypothesis 1 holds for 97.6% of the block outputs, which confirms the universality of the above hypothesis.

B. Dealing with Practical Challenges

1) Fake Inputs/Outputs: Theoretically, we assume that all the block inputs/outputs are involved in S/T, and there are no fake inputs/outputs. However, due to the imperfection of GNN prediction, some true input/output nodes may be missed and some fake nodes may be included. We refer to the above challenge as **'fake inputs/outputs'**. Fig. 5 gives some examples, where a fake input e_1 is involved in Fig. 5(a). Under this situation, the path from the fake input e_1 to o_0 may occupy a unit of capacity on edge $(o_0, T*)$, causing a missing true input a_0 as well as a false datapath $e_1 \rightarrow o_0$.

To alleviate the potential performance degradation caused by fake inputs/outputs, we propose replacing the naive Maximum Flow algorithm with the Minimum Cost Flow algorithm. This is based on Hypothesis 1, which indicates that the wrong datapaths are longer than the correct datapaths. So



(a) Min-cost flow without convex cost plan fails to handle the challenge of shared inputs.

(b) The correct solution (min-cost) given by our convex cost scheme.

Fig. 6 The condition of shared inputs. For simplicity, we use two 1-bit ripple carry adders $o_0 = a_0 + b_0$ and $o_0^{'} = a_0 + d_0$ as an example. It can be seen that the input bit a_0 is matched to two different outputs o_0 and $o_0^{'}$. To distinguish between edges of different costs, we marked edges with **larger cost** with **dotted** lines (note that the meaning represented by 'dotted' here differs from that in Fig. 5(b)). The flow paths are colored orange or blue, with blue denoting correct (target) paths and orange denoting wrong paths. Input/output nodes that fail to be matched are marked with orange colors.

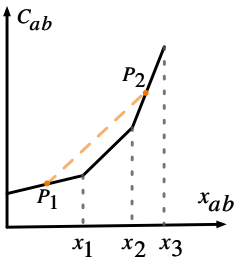


Fig. 7 A typical piecewise linear convex function.

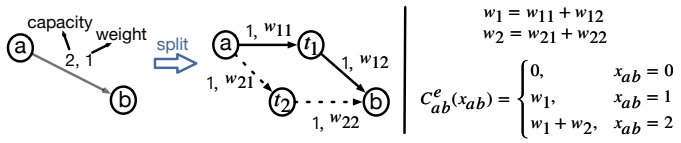


Fig. 8 An example of the split operation with $n = 2$.

by assigning each edge with unit weight, our new algorithm prefers the correct datapaths (which are shorter) over the false ones for a lower cost. For instance, in the above example shown in Fig. 5(b), the false datapath $e_1 \rightarrow o_0$ will not be picked since it is longer than the correct datapath $a_0 \rightarrow o_0$.

2) Shared Inputs: Another practical challenge is that sometimes the same inputs may be **shared** by multiple target blocks. In this scenario, a shared block input requires increased capacity to match multiple block outputs. A naive solution is to assign a larger capacity to each input edge. However, this might cause capacity preemption, as illustrated in Fig. 6(a), where a_0 is the matched input to both o_0 and $o_0^{'}$. In the case of Fig. 6(a), a_0 and b_0 are of the same distance from o_0, indicating that the two datapaths $a_0 \rightarrow o_0$ and $b_0 \rightarrow o_0$ are of the same cost and indistinguishable for o_0. Since b_0 is assigned with a larger capacity, the capacity on edge (o_0, T^*) might be only occupied by a flow of value two from b_0. Similarly, there might be a flow of value two from d_0, leaving no capacity balance a_0. From the above example, we can see that extra capacity may lead to over-allocation of flow on one datapath that **preempts** another.

To address this issue, a larger flow should be allowed to pass through each input node while keeping the flow paths from the same input node separable. Based on this, we propose introducing a convex cost scheme. The cost on edge (a, b) in the convex cost scheme is formulated as a piecewise linear convex function, which is no longer a constant value but varies with the flow value passing through. Let $C^e_{ab}(x_{ab})$ denote the cost of sending x_{ab} units of flow along edge (a, b), a typical piecewise linear convex cost function is illustrated in Fig. 7. Here the keyword 'convex' means that any line connecting two points (e.g., the dotted orange line connecting points P_1 and P_2 in Fig. 7) lay above the function. Formally, the cost function can be written as follows:

$$C^e_{ab}(x_{ab}) = \begin{cases} c_0 + k_1 \times x_{ab}, & x_0 \le x_{ab} < x_1 \\ c_0 + k_1 \times x_1 + k_2 \times (x_{ab} - x_1), & x_1 \le x_{ab} < x_2 \\ \cdots, & \cdots \\ c_0 + \sum_{i=1}^{n-1} k_i \times (x_i - x_{i-1}) \\ \quad + k_n \times (x_{ab} - x_{n-1}), & x_{n-1} \le x_{ab} \le x_n \end{cases} \tag{5}$$

We follow a standard way [44] to realize the Convex Cost Scheme. To begin, a split operation $S(n, w_1 \cdots w_n)$ on edge (a, b) is composed of two steps: (1) remove edge (a, b) and add n internal vertices $t_1, t_2 \cdots t_n$ and (2) for each internal vertex t_i, add two edges (a, t_i) and (t_i, b) with a unit capacity to connect a and b, whose total cost is w_i. The above operation $S(k, w_1 \cdots w_n)$ enables piecewise cost function since the split paths $(a \rightarrow t_i \rightarrow b)$ will be filled up in the order of cost to achieve lowest total cost. Suppose $w_0 = 0$ and $\forall i > 0$, $w_i \ge w_{i-1}$, the cost function can be written as follows:

$$C^e_{ab}(x_{ab}) = \sum_{i=0}^{x_{ab}} w_i \tag{6}$$

Fig. 8 gives an example to illustrate the above procedure.

We apply the Convex Cost Scheme to all the edges that connect to any input node. Specifically, for any input node $a_i \in S$, we apply $S(n, w_1^i \cdots w_n^i)$ to each in edge of a and $S(n, w_1^o \cdots w_n^o)$ to each out edge of a. In this way, any target

datapath $a_i \rightarrow o_i$ is divided into multiple paths with varying costs and thus different priorities.

Fig. 6(b) gives an example to illustrate the effectiveness of our algorithm. For simplicity, here we set $n = 2$, $w_1 = 1$, $w_2 = 5$. The dotted edges are of larger cost. Similarly, let $C^p_{ab}(x_{ab})$ denote the minimum cost of sending x_{ab} units of flow from a to b. For the output edges (o_0, T^*) and $(o_0^{'}, T^*)$, there are three ways to fill up their capacities: 1) $C^p_{b_0o_0}(2)$ + $C^p_{a_0o_0^{'}}(1)$ + $C^p_{d_0o_0^{'}}(1)$; 2) $C^p_{b_0o_0}(1)$ + $C^p_{a_0o_0}(1)$ + $C^p_{d_0o_0^{'}}(2)$, and 3) $C^p_{b_0o_0}(1)$ + $C^p_{a_0o_0}(1)$ + $C^p_{a_0o_0^{'}}(1)$ + $C^p_{d_0o_0^{'}}(1)$. It can be easily calculated that the minimum cost of the first and second solutions is 24 while the minimum cost of the third solution is 20. Then it is obvious that the third solution, which is also the correct solution, will be chosen. In practice, we set $n = 2$, $w_1 = 1$, $w_2 = 6$.

C. Discussion of Limitations

Although the proposed algorithm can handle several practical challenges, it is not complete. One of the main limitations is that it may perform badly when Hypothesis 1 does not hold. Given a target output o_j, suppose there is a fake input wire $e \in S$ that e is the same or less distant from o_j than the matched input a_i. In this case, our min-cost flow may choose the incorrect datapath $e \rightarrow o_j$ for a lower cost.

To make up for the above drawback, we propose running another naive maximum flow, the result of which is combined with that of the convex min-cost flow to form a final solution. However, this might lead to a higher false positive rate since more datapaths are extracted. Therefore, we propose the following filtering strategies:

1) **Remove intermediate datapaths.** In general, an extracted datapath is contained within a single arithmetic block. However, the inputs of a block B_1 might be matched to the outputs of another block B_2 at times. These datapaths are likely to include some additional inputs/outputs, such as $B1$ outputs or $B2$ inputs. Based on the observation, we remove datapaths containing multiple inputs or outputs.
2) **Filter overlong datapaths.** We remove datapaths whose length exceeds a pre-defined threshold based on the fact that fake datapaths are likely to be longer than correct ones. In practice, we set the threshold for adders to 10 and multipliers to 15.

VI. Experiments

A. Setup

Our arithmetic block detection framework is implemented using DGL [45] and PyTorch [46]. The input-output matching algorithms are developed upon networkx [47]. We also refer to the EPFL logic synthesis libraries [48] when we reimplement the baseline methods. We train and evaluate the models on a Linux machine with 1 GeForce RTX 2080 Ti GPU, 32 Intel Xeon CPUs at 2.20 GHz, and 32 GB memory. For fairness, we run each experiment 5 times and take the average performance. And for all the baseline models, we tune them and select the optimal hyper-parameters for each task.

TABLE I Statistics of the dataset.

Architecture		*BOOM*		*ROCKET*	
		#gates	#wires	#gates	#wires
Adder	Kogge-Stone	139005	361962	24540	57726
	Cond-sum	138358	360455	24737	57708
	Brent-Kung	139526	366280	24340	58124
	Hybrid	141319	369622	25491	60287
	Sklansky	141093	369774	25208	59567
	Ling	143903	378354	26179	62864
Accumulator	Radix8 Booth	140567	368155	24978	59132
	AND-based	139375	362124	24613	57891
	Mux-Based Booth	141518	369869	25732	60456
	NAND-based	139120	360374	24638	57594

B. Dataset

The dataset we use comes from open-source RISC-V CPU designs [49], including *Rocket* [50], a 5-stage in-order scalar core, and Berkeley Out-of-Order (*BOOM*) Core [50], an out-of-order superscalar RV64G core. We use *BOOM* as the training set and leave *Rocket* as the testing set. The netlists are synthesized with Synopsys Design Compiler using the SAED 32/28nm Digital Standard Cell Library. For each circuit, different designs are generated by DC through synthesizing with various design constraints. Since the design compiler will corrupt the design hierarchy by default to optimize circuit performance (i.e., across boundaries), labeling the boundary of blocks becomes difficult. We propose first using the commands '$set\ compile\_ultra\_ungroup\_dw\ false$' and '$compile\_ultra\ -no\_autoungroup$' to turn off the automatic ungrouping. This step generates a hierarchical netlist 'hier-netlist.v', where the boundary information of the target arithmetic blocks is preserved. The command 'ungroup -all -flatten' is then used to flatten the netlist and perform cross-module optimization, resulting in the final flatten netlist 'netlist.v'. Since the wire nomination keeps the same in 'hier-netlist.v' and 'netlist.v', we can label the flattened netlist with the help of the hierarchical one. It is worth noting that the above options are not required in the inference phase since prediction can be conducted without boundary information, as long as we have other ways to evaluate the prediction results, e.g., manually checked by experts.

We list the details about the generated netlists in TABLE I. In general, we evaluate the methods on two types of arithmetic blocks: adder and multiply accumulator. The Boolean functions of the two block types are $s = a + b$ and $s = a * b + c$, respectively. Compared to adders, multiply accumulators that combine addition and multiplication are more complicated. The inputs of an accumulator $s = a*b+c$ can be divided into two groups: 1) addition inputs, e.g., c, which we define as **input-1**, and 2) multiplication inputs, e.g., a and b, defined as **input-2**. Since the local structure of the two input types differs, we separate input-1 prediction from input-2 prediction.

C. Baselines

Some representative literature works [4], [14], [17] that cover functional methods, structural methods, and machine learning-driven methods are implemented as the baseline methods for comparison. In the functional approach [14],

TABLE II Performance comparison on adder boundary detection. The case 'Mixed' refers to a design with mixed adder architectures. We emphasize the best results with **boldface** and the second-best results with blue color. **Hit rate** is used for evaluation, which is defined as the number of correctly predicted (input/output) boundary nodes over the total number of target (input/output) boundary nodes.

Case	TETC'13 [14]			DATE'15 [4]			DATE'19 [17]			Ours		
	Input	Output	Runtime(s)	Input	Output	Runtime (s)	Input	Output	Runtime (s)	Input	Output	Runtime (s)
Cond-sum	0.825	0.598	305.6	0.770	0.787	13.4	0.808	0.744	13.0	**0.949**	**0.866**	**10.9**
Brent Kung	0.826	0.672	256.7	0.554	0.493	12.6	0.875	0.820	11.6	**0.950**	**0.954**	**10.2**
Hybrid	0.815	0.389	459.8	0.179	0.042	14.1	0.820	0.699	15.1	**0.947**	**0.957**	**12.0**
Kogge-Stone	0.823	0.648	436.7	0.755	0.783	14.3	0.763	0.810	13.2	**0.944**	**0.961**	**11.0**
Ling	0.803	0.456	262.5	0.249	0.022	14.8	0.874	0.653	16.3	**0.954**	**0.944**	**13.2**
Sklansky	0.823	0.626	376.5	0.484	0.483	13.3	0.864	0.845	14.1	**0.960**	**0.938**	**11.9**
Mixed	0.819	0.497	359.8	0.522	0.485	13.6	0.785	0.777	12.9	**0.940**	**0.928**	**11.1**
Average	0.819	0.555	350.7	0.502	0.442	13.7	0.827	0.763	13.8	**0.949**	**0.934**	**11.4**

TABLE III Performance comparison on multiply accumulator boundary detection. **Hit rate** is used for evaluation.

Case	TETC'13 [14]				DATE'19 [17]				Ours			
	Input1	Input2	Output	Runtime (s)	Input1	Input2	Output	Runtime (s)	Input1	Input2	Output	Runtime (s)
Radix8 Booth	0.836	0.671	0.698	340.6	0.776	0.562	0.681	13.3	**0.986**	**0.956**	**0.973**	**12.1**
AND-based	0.831	0.720	0.432	359.3	0.781	0.673	0.690	12.5	**0.986**	**0.965**	**0.970**	**11.3**
Mux-Based Booth	0.825	0.636	0.664	461.8	0.775	0.584	0.699	14.1	**0.982**	**0.965**	**0.974**	**12.4**
NAND-based	0.857	0.738	0.371	359.8	0.772	0.621	0.862	12.9	**0.986**	**0.776**	**0.969**	**11.5**
Mixed	0.833	0.712	0.508	345.1	0.774	0.603	0.737	13.0	**0.981**	**0.945**	**0.971**	**11.7**
Average	0.836	0.695	0.535	373.3	0.776	0.609	0.734	13.2	**0.984**	**0.921**	**0.971**	**11.8**

all cuts are enumerated and grouped into permutation-independent equivalence classes. Candidate words are then aggregated within each class based on *signal propagation* or *common support signal*. The authors further propose a candidate word propagation procedure to form new words. An optimistic estimation of the algorithm's performance upper bound is made by simply including all the potential words, without considering the results of *symbolic simulation* or *equivalence checking*. Wei et al. [4] conducted structural identification by constructing `xor`-forests based on the connection hierarchy. To be more specific, they use cut enumeration to identify all `xor`-trees, and then build the`xor`-forests by identifying carry-out signals. These `xor`-forests are possible instances of adders. Fayyazi et al. [17] proposed to learn a special vector, *level-dependent decaying sum* (LDDS) *existence vector* (EV), to represent circuit topology. The proposed EV counts the number of gate types within the local neighborhood with distance-based penalty weights. For the sake of fairness, the oversampling technique is also applied during training.

D. Evaluation of Boundary Detection

Performance comparison on arithmetic block boundary detection is first made between our method and all baseline approaches [4], [14], [17]. Since the structural approach [14] is customized for adders, it was not applied to identify multiply accumulators. From Tables II and III, it can be concluded that our proposed method greatly outperforms prior works, achieving over 90% average hit rate in all the boundary detection tasks. The other machine learning approach [17] achieves the second-best performance in the case of adder detection, which confirms the excellent adaptability of deep learning methods. The method by Subramanyan et al. [14] are able to cover lots of words composed of replicated functional bitslices, and therefore achieves an acceptable hit rate, at the cost of much higher runtime. The approach by Wei et al. [4] is stable for the more regular architectures (Cond-sum, *Kogge-Stone*), but does not perform well given complicated or highly optimized structures (Hybrid, *Ling*), resulting in unsatisfactory average hit rate.

E. Ablation Study on ABGNN

We conducted comprehensive experiments to evaluate our proposed graph neural network architecture and demonstrate its outstanding capability in DAG representation learning. The models were trained for 100 epochs, taking around 2 hours.

Bidirectional Depths Selection. The depth of a Graph Neural Network model indicates the number of search hops. Larger search hops result in a broader search scope, allowing more structural information to be aggregated, which may benefit the recognition. On the other hand, the increased search scope is more likely to involve interference information and might suffer from the over-smoothing issue. Furthermore, the neighboring information from the two directions may not be of equal importance. For example, let node n be an input wire of block B; the fanout cone rooted at n, which contains the structural information of B, is unquestionably more useful in determining whether n is an input node. Therefore, it is natural to specify different network depths for the two directions, depending on the specific task. We carefully select proper fanin depth and fanout depth after comparing the performance of models with different depths. In practice, we set the (fanin depth, fanout depth) as (1, 4) for adder input prediction, (3, 0) for adder output, (2, 4) for accumulator input-1, (0, 5) for accumulator input-2, and (4,3) for accumulator output prediction.

TABLE IV Performance comparison between different GNN models on adder input boundary detection.

Case	GIN [19]		GAT [51]		GraphSage [18]		D-SAGE [25]		ABGNN (Ours)	
	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score
Cond-sum	0.875	0.890	0.882	0.885	0.935	0.931	0.940	0.935	**0.949**	**0.951**
Brent Kung	0.809	0.873	0.906	0.910	0.915	0.915	0.943	0.937	**0.950**	**0.964**
Hybrid	0.930	0.937	0.903	0.890	0.930	0.928	0.940	0.942	**0.947**	**0.949**
Kogge-Stone	0.925	0.917	0.918	0.887	0.940	0.920	0.951	0.931	**0.944**	**0.954**
Ling	0.930	0.925	0.915	0.881	0.950	0.941	0.940	0.942	**0.954**	**0.963**
Sklansky	0.935	0.938	0.901	0.895	0.928	0.930	0.947	0.932	**0.960**	**0.955**
Mixed	0.865	0.882	0.872	0.883	0.886	0.909	0.915	0.903	**0.940**	**0.928**
Average	0.896	0.908	0.899	0.890	0.926	0.921	0.940	0.932	**0.949**	**0.952**

TABLE V Performance comparison between different GNN models on adder output boundary detection.

Case	GIN [19]		GAT [51]		GraphSage [18]		D-SAGE [25]		ABGNN (Ours)	
	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score
Cond-sum	0.835	0.863	0.785	0.798	0.863	0.880	**0.888**	**0.905**	0.866	**0.905**
Brent-Kung	0.906	0.921	0.845	0.870	0.925	0.923	0.947	0.948	**0.953**	**0.950**
Hybrid	0.911	0.875	0.940	0.897	0.93	0.912	0.912	0.927	**0.955**	**0.939**
Kogge-Stone	0.941	0.915	0.878	0.889	0.945	0.942	0.963	0.952	**0.965**	**0.955**
Ling	0.916	0.912	0.935	0.909	0.912	0.911	0.895	0.905	**0.945**	**0.948**
Sklansky	0.898	0.894	0.865	0.855	0.932	0.919	0.879	0.912	**0.938**	**0.943**
Mixed	0.859	0.878	0.835	0.844	0.880	0.854	0.906	0.887	**0.917**	**0.921**
Average	0.895	0.894	0.869	0.866	0.913	0.905	0.913	0.920	**0.934**	**0.937**

TABLE VI Performance comparison between different GNN models on multiply accumulator input-1 boundary detection.

Case	GIN [19]		GAT [51]		GraphSage [18]		D-SAGE [25]		ABGNN (Ours)	
	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score
adix8 Booth	0.966	0.951	0.957	0.937	0.975	0.962	0.981	0.967	**0.986**	**0.975**
AND-based	0.971	0.949	0.980	0.966	0.980	0.962	0.981	0.967	**0.986**	**0.975**
Mux-Based Booth	0.962	0.944	0.957	0.953	0.971	0.951	0.949	0.951	**0.982**	**0.973**
NAND-based	0.966	0.955	0.975	0.966	0.971	0.957	0.981	**0.967**	**0.986**	0.954
Mixed	0.963	0.948	0.966	0.955	0.974	0.959	0.967	0.960	**0.981**	**0.964**
Average	0.966	0.949	0.967	0.955	0.974	0.958	0.972	0.962	**0.984**	**0.968**

TABLE VII Performance comparison between different GNN models on multiply accumulator input-2 boundary detection.

Case	GIN [19]		GAT [51]		GraphSage [18]		D-SAGE [25]		ABGNN (Ours)	
	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score
Radix8 Booth	0.781	0.836	0.526	0.656	0.829	0.809	0.772	0.807	**0.956**	**0.965**
AND-based	0.868	0.855	0.816	0.825	0.873	0.834	0.873	0.847	**0.965**	**0.878**
Mux-Based Booth	0.864	0.851	0.820	0.791	0.882	0.839	0.895	0.838	**0.965**	**0.967**
NAND-based	0.539	0.676	**0.785**	0.817	0.623	0.676	0.592	0.687	0.776	**0.818**
Mixed	0.832	0.844	0.805	0.811	0.832	0.825	0.853	0.819	**0.945**	**0.922**
Average	0.777	0.812	0.750	0.780	0.808	0.797	0.797	0.800	**0.921**	**0.910**

TABLE VIII Performance comparison between different GNN models on multiply accumulator output boundary detection.

Case	GIN [19]		GAT [51]		GraphSage [18]		D-SAGE [25]		ABGNN (Ours)	
	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score
Radix8 Booth	0.961	0.940	0.969	0.961	0.721	0.789	0.952	0.965	**0.973**	**0.982**
AND-based	0.952	0.932	0.952	0.952	0.956	0.916	0.961	0.969	**0.970**	**0.981**
Mux-Based Booth	0.965	0.940	0.952	0.950	0.956	0.911	**0.991**	**0.985**	0.974	0.975
NAND-based	0.956	0.924	0.974	0.961	0.712	0.762	0.948	0.962	**0.969**	**0.980**
Mixed	0.958	0.935	0.961	0.954	0.855	0.837	0.967	0.968	**0.971**	**0.979**
Average	0.958	0.934	0.962	0.956	0.840	0.843	0.964	0.970	**0.971**	**0.979**

TABLE IX Ablation Study of ABGNN. BGNN refers to a synchronous bi-directional model, and AGNN refers to an asynchronous uni-directional model.

Task	Model	Recall	F1-score	Runtime (ms)
Region	ABGNN	0.733	0.705	143.8
Input	ABGNN	0.951	0.943	171.5
	BGNN	0.941	0.939	214.4
	AGNN	0.929	0.92	156.3
Output	ABGNN	0.952	0.958	110.6
	BGNN	0.948	0.952	139.4
	AGNN	0.878	0.845	75.2

Comparison with State-of-the-Art GNNs. To fully demonstrate the effectiveness of our proposed techniques, our ABGNN is compared with several state-of-the-art Graph Neural Networks, including GAT [51], GIN [19], GraphSAGE [18], and D-SAGE [25] on the *Rocket* dataset. As listed in Tables IV to VIII, ABGNN shows its superiority on DAG representation learning with a higher average recall and F1 score than the baseline methods. Among all the baselines, D-SAGE is the most similar to ours, which also considers bidirectional information combination. However, the identical search depth for both directions (fanin/fanout) in D-SAGE leads to performance degradation. As previously stated, information from the two directions plays different roles in the boundary identification tasks, implying distinct optimal search depth for fanin and fanout. Consequently, the completely-separate bidirectional information collection in our model fits the scenario better, resulting in superior performance. Furthermore, the decoupling operation reduces the neighborhood scale significantly so that the over-smoothing issue is alleviated. As a result, it allows deeper models and thus better performance. Particularly, ABGNN greatly outperforms other models by more than 10% on the task of accumulator input-2 boundary prediction. The performance gain benefits from the deeper architecture of ABGNN. Since the scale of a single-direction neighborhood is much smaller than that of a full neighborhood, ABGNN suffers less from the over-smoothing issue and allows for larger search depth. As a result, ABGNN shows its superiority in complex cases that requires a deeper view. To sum up, our proposed ABGNN is shown to be superior to state-of-the-art GNNs on netlist representation learning.

Ablation Studies. In the following part, we conduct some ablation studies to demonstrate the efficacy of the proposed techniques. Due to limited space, we only show the average performance on the two types of arithmetic blocks. We first make a comparison between the asynchronous and synchronous message passing scheme to find out its effect. A synchronous model (**BGNN**) is built by replacing the message passing scheme in ABGNN with a synchronous one while remaining the model architecture. To be fair, we leave the other hyper-parameters the same, e.g., the number of layers. The results are listed in TABLE IX. It can be seen that the asynchronous scheme reduces the model's inference time by 20.0% and 20.7% for input and output boundary identification, respectively, with no performance degradation.

We also conducted experiments to demonstrate the effects of bidirectional information aggregation, as listed in TABLE IX. A unidirectional model (**AGNN**) is built with no fanout layers for output boundary detection and no fanin layers for input identification. It is found that bidirectional information aggregation improves recall by 7.4% and F_1-score by 11.3% for the output task, while for the input task, the improvement is 2.2% and 2.3%, respectively. The results indicate that information from a single direction is insufficient in the boundary prediction problem. Therefore, combining representations learned from both directions is necessary.

Furthermore, we compare the two different problem formulations mentioned in Section IV-A, namely region detection and boundary detection. It is worth noting that we adopt the latter one in our final solution. For region detection, the idea is to assign a positive label to all the nodes within the target arithmetic block (including I/O nodes). TABLE IX shows the performance of ABGNN in the two tasks. As can be seen, ABGNN performs far better in boundary detection than in region detection. The result indicates that internal nodes are not as distinct as boundary nodes.

F. Evaluation of Input-output Matching

In this part, we conducted experiments to measure the performance of our proposed convex cost network-flow-based input-output matching approach. For accumulators, we run two separate flows f_1 and f_2 to match input-1 nodes and input-2 nodes with the output nodes, respectively. Since the input-1 nodes serve as operands for addition, f_1 resembles adder input-output matching. On the other hand, f_2 is close to multiplier input-output matching. Due to limited space, we only discuss the results of the matching for input-2 because it deserves more attention.

In the following part, two metrics are used for evaluation: 1) Hit Rate (HR): the most important metric defined as the proportion of matched true input/output nodes to total target input/output nodes. 2) False Positive Rate (FPR): a metric that depicts the proportion of fake input/output nodes involved in the results, defined as the number of fake input/output nodes divided by the total number of matched input/output nodes.

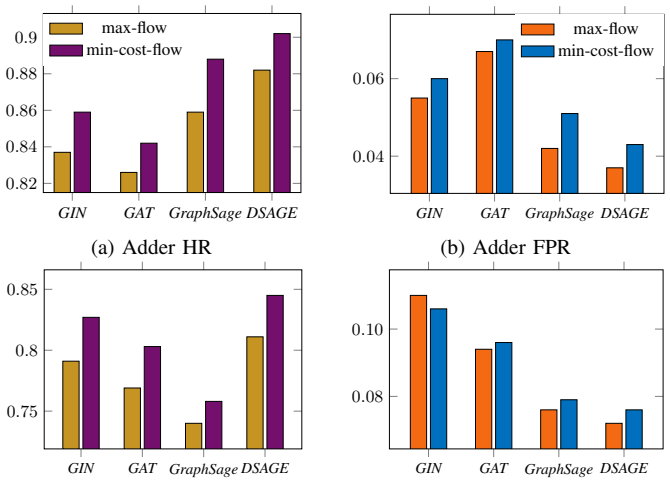
We compare the performance of our convex cost network-flow-based algorithm with the naive maximum-flow-based one proposed in the previous work [1]. All the algorithms have been combined with the filtering techniques proposed in Section V-C to reduce the false-positive rate. The results are listed in Tables X and XI. It can be seen that our newly proposed approach outperforms the previous approach. On average, it achieves a hit rate gain of 2.1%–4.7% at the cost of 0.0%–1.4% increase in false positive rate. Overall, our proposed algorithm only misses 3.8% inputs and 0.4% outputs while involving less than 5% fake nodes. On the other hand, the performance is degraded for accumulators. We can see a miss rate of 0.5% for inputs and 4.9% for outputs, as well as around 10% fake nodes. More specifically, both methods perform badly in the case of NAND-based multiply accumulators. Combined with the results from TABLE VII, we speculate that the degradation might be caused by lost input/output nodes due to the fuzzy prediction of GNN models. We also conducted experiments to evaluate the efficacy

TABLE X Performance comparison on adder input-output matching. HR and FPR are hit rate and false positive rate, respectively.

Case	Input				Output			
	Naive max-flow		Min-cost-flow (ours)		Naive max-flow		Min-cost-flow (ours)	
	HR	FPR	HR	FPR	HR	FPR	HR	FPR
Brent Kung	0.939	0.020	**0.950**	**0.018**	0.912	0.021	**0.954**	**0.019**
Kogge-Stone	0.916	0.025	**0.942**	**0.023**	0.896	**0.027**	**0.957**	**0.027**
Cond-sum	0.769	**0.071**	**0.784**	0.098	0.813	0.052	**0.866**	**0.051**
Ling	0.924	**0.016**	**0.948**	**0.016**	0.901	0.017	**0.941**	**0.015**
Hybrid	0.889	0.051	**0.914**	**0.049**	0.883	0.046	**0.954**	**0.042**
Sklansky	0.917	**0.046**	**0.950**	0.051	0.936	**0.043**	**0.938**	0.050
Mixed	0.878	**0.042**	**0.892**	0.056	0.857	0.035	**0.912**	**0.031**
Average	0.890	**0.039**	**0.911**	0.044	0.885	**0.034**	**0.932**	**0.034**

TABLE XI Performance comparison on multiply accumulator input-output matching (input-2 with output).

Case	Input-2				Output			
	Naive max-flow		Min-cost-flow (ours)		Naive max-flow		Min-cost-flow (ours)	
	HR	FPR	HR	FPR	HR	FPR	HR	FPR
Radix8 Booth	0.930	**0.009**	**0.955**	0.018	0.934	**0.009**	**0.965**	0.013
AND-based	0.882	**0.152**	**0.943**	0.192	0.865	**0.150**	**0.913**	0.194
Mux-Based Booth	0.947	**0.014**	**0.964**	0.018	0.961	0.027	**0.973**	**0.022**
NAND-based	0.774	**0.124**	**0.775**	0.128	0.777	**0.136**	**0.803**	0.168
Mixed	0.914	**0.027**	**0.944**	0.035	0.938	0.042	**0.958**	**0.037**
Average	0.889	**0.065**	**0.916**	0.078	0.895	**0.073**	**0.922**	0.087



(a) Adder HR (b) Adder FPR

(c) Accumulator HR (d) Accumulator FPR

Fig. 9 Performance comparison on input-output matching algorithms when combined with baseline GNN models. Here we take the average performance on inputs and outputs.

of our approach combined with the baseline models. As can be seen from Fig. 9, our matching algorithm outperforms the previous maximum-flow-based method in all the cases. Specifically, our approach achieves an average hit rate gain of 2.2% for adders and 3.0% for multiply accumulators, at the cost of less than 0.5% increase in false positive rate. The results fully demonstrate the efficacy of our proposed algorithm and its superiority over the previous work.

VII. Conclusion

Identifying arithmetic blocks is a vital procedure for various tasks like malicious logic detection and logic optimization. In this work, we propose a graph learning-based arithmetic block identification framework that efficiently recognizes the boundary of arithmetic blocks. To boost the performance of the whole framework, we propose a specialized graph neural network architecture for netlist representation learning, which outperforms existing dominantly used GNNs. We further develop a convex cost network-flow approach to match the input and output wires predicted by the GNN models. Experimental results have confirmed the superior our framework: compared with state-of-the-art structural, functional, and machine learning-based block mapping schemes, our framework achieves the highest sensitivity with the fastest runtime in adder and multiply accumulator identification from an open-source RISC-V CPU design (the Rocket core). Moreover, our proposed input-output matching algorithm handles several practical challenges and outperforms previous works.

References

[1] Z. He, Z. Wang, C. Bai, H. Yang, and B. Yu, "Graph learning-based arithmetic block identification," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021, pp. 1–8.

[2] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[3] A. Mahzoon, D. Grose, and R. Drechsler, "Revsca: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[4] X. Wei, Y. Diao, T.-K. Lam, and Y.-L. Wu, "A universal macro block mapping scheme for arithmetic circuits," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2015, pp. 1629–1634.

[5] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[6] T. Meade, S. Zhang, Z. Zhao, D. Pan, and Y. Jin, "Gate-level netlist reverse engineering tool set for functionality recovery and malicious logic detection," in *Proc. ISTFA*, 2016.

[7] H. Li, S. Patnaik, A. Sengupta, H. Yang, J. Knechtel, B. Yu, E. F. Young, and O. Sinanoglu, "Attacking split manufacturing from a deep learning perspective," in *ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[8] C. Yu and M. Ciesielski, "Automatic word-level abstraction of datapath," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 1718–1721.

[9] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2013, pp. 1277–1280.

[10] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik, "Template-based circuit understanding," 2014, pp. 83–90.

[11] T. Doom, J. White, A. Wojcik, and G. Chisholm, "Identifying high-level components in combinational circuits," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 1998, pp. 313–318.

[12] N. Rubanov, "A high-performance subcircuit recognition method based on the nonlinear graph optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 25, no. 11, pp. 2353–2363, 2006.

[13] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "Wordrev: Finding word-level structures in a sea of bit-level gates," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 67–74.

[14] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascón, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, "Reverse engineering digital circuits using structural and functional analyses," *IEEE Transactions on Emerging Topics in Computing (TETC)*, vol. 2, no. 1, pp. 63–80, 2013.

[15] L. Azriel, R. Ginosar, and A. Mendelson, "SoK: An overview of algorithmic methods in IC reverse engineering," in *ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, 2019, pp. 65–74.

[16] L. M. Silva, F. V. Andrade, A. O. Fernandes, and L. F. M. Vieira, "Arithmetic circuit classification using convolutional neural networks," in *International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–7.

[17] A. Fayyazi, S. Shababi, P. Nuzzo, S. Nazarian, and M. Pedram, "Deep learning-based circuit recognition using sparse mapping and level-dependent decaying sum circuit representations," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2019, pp. 638–641.

[18] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Annual Conference on Neural Information Processing Systems (NIPS)*, 2017, pp. 1024–1034.

[19] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[20] V. Thost and J. Chen, "Directed acyclic graph neural networks," *arXiv preprint arXiv:2101.07965*, 01 2021.

[21] 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 *ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[22] T. Chen, Q. Sun, C. Zhan, C. Liu, H. Yu, and B. Yu, "Analog ic aging-induced degradation estimation via heterogeneous graph convolutional networks," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. IEEE, 2021, pp. 898–903.

[23] ——, "Deep h-gcn: Fast analog ic aging-induced degradation estimation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.

[24] M. Zhang, S. Jiang, Z. Cui, R. Garnett, and Y. Chen, "D-vae: A variational autoencoder for directed acyclic graphs," *arXiv preprint arXiv:1904.11088*, 01 2019.

[25] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate operation delay prediction for fpga hls using graph neural networks," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020, pp. 1–9.

[26] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 26, no. 10, pp. 1367–1372, 2004.

[27] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BMC bioinformatics*, vol. 14, no. 7, pp. 1–13, 2013.

[28] K. Xu, L. Wang, M. Yu, Y. Feng, Y. Song, Z. Wang, and D. Yu, "Cross-lingual knowledge graph alignment via graph matching neural network," *arXiv preprint arXiv:1905.11605*, 2019.

[29] Z. Lou, J. You, C. Wen, A. Canedo, J. Leskovec *et al.*, "Neural subgraph matching," *arXiv preprint arXiv:2007.03092*, 2020.

[30] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[31] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *arXiv preprint arXiv:1606.09375*, 2016.

[32] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[33] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2019, pp. 793–803.

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

[35] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, no. 4, pp. 198–206, 1981.

[36] L. Soule and A. Gupta, "An evaluation of the chandy-misra-bryant algorithm for digital logic simulation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 1, no. 4, pp. 308–347, 1991.

[37] M. Buda, A. Maki, and M. A. Mazurowski, "A systematic study of the class imbalance problem in convolutional neural networks," *Neural Networks*, vol. 106, pp. 249–259, 2018.

[38] B. Kang, S. Xie, M. Rohrbach, Z. Yan, A. Gordo, J. Feng, and Y. Kalantidis, "Decoupling representation and classifier for long-tailed recognition," *arXiv preprint arXiv:1910.09217*, 2019.

[39] A. Ali, S. M. Shamsuddin, and A. L. Ralescu, "Classification with class imbalance problem," *Int. J. Advance Soft Compu. Appl*, vol. 5, no. 3, 2013.

[40] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[41] C. Elkan, "The foundations of cost-sensitive learning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, vol. 17, no. 1, 2001, pp. 973–978.

[42] Z. He, P. Liao, S. Liu, Y. Ma, Y. Lin, and B. Yu, "Physical synthesis for advanced neural network processors," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2021, pp. 833–840.

[43] H. Xiang, M. Cho, H. Ren, M. Ziegler, and R. Puri, "Network flow based datapath bit slicing," in *ACM International Symposium on Physical Design (ISPD)*, 2013, pp. 139–146.

[44] A. Ravindra, M. Thomas, and O. James, "Network Flows," MIT Press, Cambridge, Massachusetts, United States, Tech. Rep., 1993.

[45] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[46] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

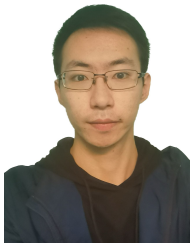
[47] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using NetworkX," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[48] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The EPFL logic synthesis libraries," *arXiv preprint arXiv:1805.05121*, 2018.

[49] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[50] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[51] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.



Ziyi Wang received his B.S. degree from the Department of Computer Science and Technology, Fudan University in 2021. He is currently pursuing his Ph.D. degree at the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include graph learning applications in electronic design automation (EDA) and timing analysis.

Zhuolun He received his B.S. degree in computer science and engineering from Peking University in 2017. He is currently a Ph.D. candidate at the Chinese University of Hong Kong. His research interests are to improve EDA tool efficiency via programming and algorithmic approaches.

Chen Bai received the B.E. degree in software engineering from the University of Electronic Science and Technology of China in 2020. He is currently pursuing the Ph.D. degree at the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include design space exploration and agile VLSI design methodologies. He received William J. McCalla Best Paper Award from ICCAD 2021.

Haoyu Yang is currently a Research Scientist at NVIDIA Corp., where he actively conducts research on computational lithography and machine learning. Prior to that, he was a postdoctoral fellow in the Department of Computer Science and Engineering, the Chinese University of Hong Kong. His research interests include (1) Machine Learning in VLSI Design for Manufacturability (2) High Performance VLSI Physical Design with Parallel Computing and (3) Machine Learning Security. Haoyu Yang is the recipient of SPIE Nick Cobb scholarship and best paper candidate of ASPDAC2019.

Bei Yu (Senior Member, IEEE) received the Ph.D. degree from The University of Texas at Austin in 2014. He is currently an Associate Professor in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. He has served as TPC Chair of ACM/IEEE Workshop on Machine Learning for CAD, and in many journal editorial boards and conference committees. He is Editor of IEEE TCCPS Newsletter. He received nine Best Paper Awards from DATE 2022, ICCAD 2021 & 2013, ASPDAC 2021 & 2012, ICTAI 2019, Integration, the VLSI Journal in 2018, ISPD 2017, SPIE Advanced Lithography Conference 2016, and six ICCAD/ISPD contest awards.