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,ziyiwang21,byu}@cse.cuhk.edu.hk

***Abstract*—Arithmetic block identification in gate-level netlist is an essential procedure for malicious logic detection, functional verification, or macro-block optimization. We argue that existing methods suffer either scalability or performance issues. To address the problem, we propose a graph learning-based solution that promises to extract desired logic components from a complete design netlist. We further design a novel asynchronous bidirectional graph neural network (ABGNN) dedicated to representation learning on directed acyclic graphs. 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 the driving force for numerous datapath optimization or functional verification methodologies. For example, Symbolic Computer Algebra (SCA) based multiplier verification [1], [2] relies heavily on the detection of Half Adders in the multiplier netlist. It is also desired to replace a detected arithmetic block with pre-optimized logic or even new macro blocks built with more advanced technologies [3]. Additionally, the demand for malicious logic detection is widely pointed out [4]–[6] to ensure circuit security and functionality in the globalization of VLSI design, manufacturing, and distribution. Besides the applications mentioned above, a technical reason behind the need for such a 'reverse engineering' approach is that most high-level components, such as function declaration and modularization, are flattened into netlists of Boolean gates by logic synthesis and technology mapping [7]. Therefore, despite sounding like '*finding a needle in a haystack*' (or in a sea of bit-level gates [8]), arithmetic block identification is an essential procedure worthy of exploration.

Classic approaches to identify arithmetic components can be roughly categorized into either *structural methods* [5], [8]–[11] or *functional methods* [12], [13]. **Structural methods** concentrate on circuit topology while paying little attention to the functionality of the circuit [14]. For instance, *shape hashing* is introduced in [8] to group wires with the same local topology together to form candidate words. Specifically, a k-step-bounded depth-first traversal of the graph is performed starting from each wire to produce its serialization using the gate and wire types. Some other works consider the scenario where a library of reference circuits is given, and the problem becomes mapping subcircuits with reference circuits. In [10], the subcircuit matching problem is formulated as a regularized quadratic assignment problem to minimize both graph distance and vertex label distance. A nonlinear version of the iterative Kaczmarz Method (KM) is used to solve the obtained equations. Structural methods usually promise to identify target blocks with customized algorithms efficiently. However, they are often mathematically incomplete due to the heuristic methodology. On the other hand, **functional methods** functionally analyze the circuit for potential arithmetic components. A typical example as in [12] extends the above shape hashing method by considering the functions implemented by a set of gates using cut enumeration. They enumerate all 6-feasible cuts and group equivalent cuts using permutation-independent Boolean matching. In this way, each equivalence class of cuts may match a known library function. It is further proposed in [12] to use functional verification tools for module matching: suppose C is a potential arithmetic block with input word X and side inputs Y, C' is a reference circuit, and Φ is an inserted comparator miter between the outputs of C and C', the Quantified Boolean Formula (QBF) $\exists Y \forall X \Phi(X, Y)$ exactly models the equivalence checking problem. As can be seen, functional methods are accurate and solver-ready at the cost of ultra-long runtime.

† Equal contributors, listed alphabetically by last name.

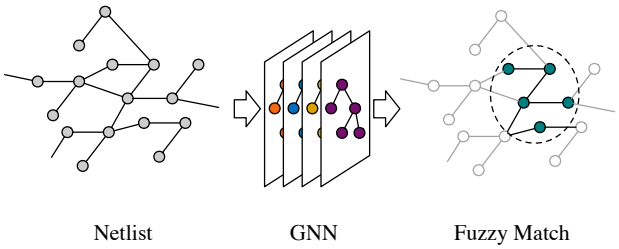


Fig. 1 Graph learning enables netlist fuzzy matching.

The development of machine learning and deep neural networks has provided alternate solutions to recognition and classification. To efficiently identify functional units, [15] and [16] recently propose deep learning-based solution to recognize arithmetic circuits. Silva *et al.* [15] develop a flow that converts conjunctive normal form (CNF) clauses into images, which are later rescaled to the desired size and fed into deep learning classifiers. Fayyazi *et al.* [16] present a compact representation called existence vector (EV) that encodes a circuit node with its all neighbors. A fixed number of EVs are selected to satisfy the fixed-input-size requirement of convolutional neural networks. However, these solutions are dedicated to one given unknown functional block. When dealing with large-scale netlist design, these solutions are facing significant challenges.

To address the above concerns, in this paper, we propose a graph learning-based arithmetic block identification framework, as briefly illustrated in Fig. 1, that can efficiently conduct fuzzy matching on arithmetic blocks. The framework takes a large-scale netlist as input, and outputs fuzzy-matched sub-graphs as target arithmetic components. Since a netlist is often represented as a directed acyclic graph (DAG), it is motivated to utilize graph neural networks (GNNs) as the preferable fuzzy matching solution. Intuitively, GNNs can aggregate information from neighbourhoods to generate meaningful low-dimensional embeddings for each vertex for downstream tasks. However, most existing popular GNN models, such as GraphSAGE [17] and GIN [18], are designed for general graphs or undirected graphs. In other words, they are not well-optimized for DAGs. Therefore, we come up with a variant of GNN, *asynchronous bidirectional graph neural network* (**ABGNN**), which is customized for DAG embedding with supreme performance and 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 DAG representation learning;
- 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 matching solutions.

The rest of the paper is organized as follows: Section II introduces the problem formulation. Section III overviews our proposed arithmetic block identification flow. Section IV describes the design details of our ABGNN model. Section V introduces the network flow approach for input output matching. Section VI discusses other implementation details of the proposed framework; Section VII presents experimental evaluations of our proposed framework and ablation studies of the proposed techniques. Section VIII concludes the paper.

II. Problem Formulation

We first introduce the problem formulation. 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 usually generated by logic synthesis tools, which converts an abstract specification of circuit behaviour (typically at *register transfer level* (RTL)) into design implementation in terms of logic gates. Mathematically, a gate-level netlist can be naturally formulated as a directed acyclic graph, whose vertices are the circuit components and edges represent wires between them. Sometimes we emphasize a *flattened* gate-level netlist, where 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 subtraction. In general, our target is to discover the arithmetic blocks located in a flattened netlist. For simplicity, in this paper, we focus on identifying adders, which are one of the major arithmetic components. However, our proposed graph learning-based framework can be easily extended to other desired arithmetic blocks.

Problem 1 (Adder Identification)**.** *Given the flattened gate-level netlist of a circuit design, identify adders located in the netlist.*

III. Flow Overview

Before introducing algorithmic details, we briefly overview our proposed arithmetic block identification flow. Given a design netlist, we first transform it into a directed acyclic graph (DAG) representation. 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 VI-A). Then, we run a network flow-based algorithm (introduced in Section V) to match the predicted input wires with the predicted outputs wires. We illustrate the overall flow in Fig. 2.

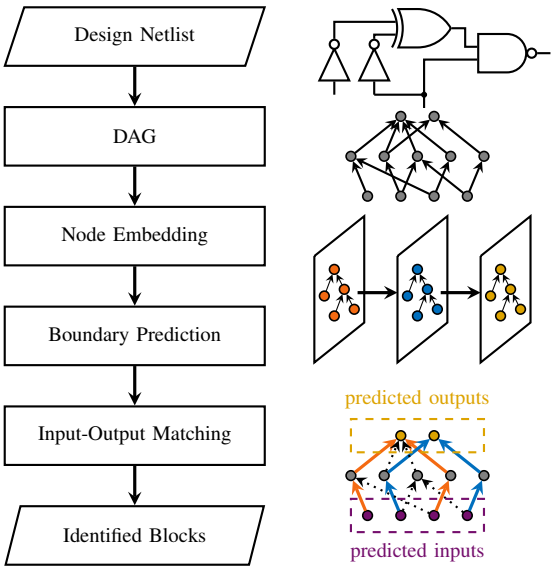


Fig. 2 Our arithmetic block identification flow.

IV. Designing Graph Neural Network for DAGs

Graph neural networks enable a powerful representation learning paradigm for graphs. In general, GNNs follow a neighbor aggregation (or equivalently, message passing) scheme [18]: the representation vector of a node is computed by recursively aggregating and transforming representation vectors of its neighboring nodes. The above message passing scheme has achieved state-of-the-art performance on various tasks on graphs, such as node classification, link prediction, and graph classification. Nevertheless, it is still critical to customize graph neural network architecture according to the actual task to earn the best result. In this section, we discuss how do we design a novel graph neural network architecture dedicated to DAG representation learning in our adder IO prediction task.

A. General Graph Neural Network

We start with a formal introduction to general graph neural networks, partly following the notations in [18]. A graph can be represented as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, 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. Vertices are equipped with initial feature vectors $\mathcal{X} = \{\boldsymbol{x}_v | \forall v \in \mathcal{V}\}$. As introduced, GNNs follow a neighbor aggregation scheme. The k-th iteration of message passing, or say the k-th layer of a GNN, can be written as follows:

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

where $\boldsymbol{h}_v^{(k)}$ is the representation vector of vertex v after k iterations, $\boldsymbol{h}_v^{(0)} = \boldsymbol{x}_v$, and $\mathcal{N}(v)$ denotes the neighbouring nodes of v. Many GNN variants with different choices of the `AGGREGATE` function and the `COMBINE` are proposed, which are crucial to the model performance. In practice, common selection of the `AGGREGATE` function include `mean` aggregators, `max` aggregators, and `sum` aggregators, usually followed by a multi-layer perceptron (MLP). As a concrete example, *GraphSAGE* [17], one of the dominantly used architectures, aggregates neighborhood information in the following way:

$$\boldsymbol{h}_v^{(k)} = \sigma(\boldsymbol{W} \cdot \texttt{MEAN}(\{\boldsymbol{h}_v^{(k-1)}\} \cup \{\boldsymbol{h}_u^{k-1} : u \in \mathcal{N}(v)\})),$$

where σ is an activation function (e.g. `sigmoid`). This is also a rough, linear approximation of a localized spectral convolution.

B. Bidirectional Graph Neural Network

We now come to the first keyword, '*Directed*', of '*Directed Acyclic Graph*'. Directed graphs assign each edge a direction, which naturally captures various real-life relations. In our netlist, the edge direction indicates the current flow direction, or say the execution order of the circuit. Therefore, modeling a netlist with a directed graph is intrinsic and significant.

However, most existing GNN models assume to work for undirected graphs. One historical reason is that, earlier spectral GNN

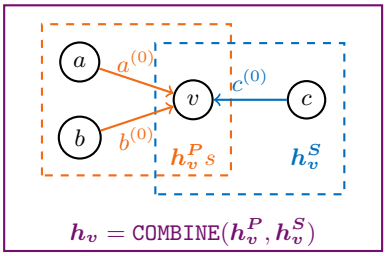


Fig. 3 Bidirectional information aggregation for the vertex v. We train two GNNs to aggregate information from the fanin cone ($\boldsymbol{h}_v^P$, in orange) and the fanout cone ($\boldsymbol{h}_v^S$, in blue) respectively. The final embedding ($\boldsymbol{h}_v$, in purple) is given by the combination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

models [19]–[21], built upon the analogy to Convolutional Neural Networks (CNNs), define the *graph convolution* 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, we do not directly have this property for a directed graph. One straightforward way is to relax the directed graph to an undirected graph by symmetrizing its adjacency matrix, but this inevitably results in information loss.

Our designed bidirectional GNN is greatly motivated by the design of heterogeneous GNNs [22], [23]. As discussed in [22], 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 cone and its fanout cone. It is therefore necessary to combine information from both directions to generate representative node embeddings. 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, each vertex **only aggregates information from its *predecessors***. In other words, information flows from x to y if there is an edge (x, y). We train 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 that aggregate information from the predecessors (i.e., fanin cone) and the successors (i.e., fanout cone) respectively. Thus, the final embedding of each vertex is given by the combination of both $\boldsymbol{h}_v^P$ and $\boldsymbol{h}_v^S$:

$$\boldsymbol{h}_v = \texttt{COMBINE}(\boldsymbol{h}_v^P, \boldsymbol{h}_v^S) \tag{1}$$

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

C. Asynchronous Graph Neural Network

We move to the second keyword, 'Acyclic', of 'Directed Acyclic Graph'. Acyclic graphs contain no cycles. That is, if we start from any vertex v, walking through the graph following the edge directions, we will never come back to v. Although this property may sound irrelevant to GNN design, we show that it is possible to improve the efficiency of GNN utilizing the acyclic property.

Let's make an analogy to event-driven logic simulation, taking the Chandy-Misra-Bryant (CMB) distributed-time algorithm [24] as an example. To enable parallel logic simulation with the CMB algorithm, circuit elements communicate with each other using timestamped messages, and different elements may consume events at distinct simulation times concurrently. Conceptually, each element consumes timestamped event messages on its inputs; whenever all inputs are ready, it advances its local time and possibly sends out new time stamped event messages on its output. Fig. 4(a) illustrates the event message scheme assuming a unit delay for each gate. The original CMB algorithm is regarded as '*an approach to carry out asynchronous, distributed simulation on multiprocessor message-passing architectures*' [24]. On the contrary, general GNNs work in a *synchronous* way. In synchronous message passing, all messages flow on edges simultaneously in each iteration, such that every vertex receives messages and updates its representation on every iteration, causing great computational demand, as shown in Fig. 4(b).

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 starts from the leaf nodes of the cone, through the cone, and all the way up to v. At each 'timestamp', only the vertices receive messages at the previous timestamp pass the message 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. The table in Fig. 4(d) lists are the messages sent out by nodes at each timestamp. We can see that asynchronous GNN executes as efficiently as logic simulation while being much 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 described as follows:

$$\begin{aligned} \boldsymbol{a}^{(k)}_{\{\mathbf{i}:\mathcal{D}(\mathbf{i},\mathbf{v})=\boldsymbol{\Delta}-\mathbf{k}\}} &= \texttt{AGGREGATE}(\{\boldsymbol{h}_u^{(k-1)} : u \in \mathcal{N}(i)\}), \\ \boldsymbol{h}^{(k)}_{\{\mathbf{i}:\mathcal{D}(\mathbf{i},\mathbf{v})=\boldsymbol{\Delta}-\mathbf{k}\}} &= \texttt{COMBINE}(\boldsymbol{a}_i^{(k)}, \boldsymbol{h}_i^{(\mathbf{0})}), \end{aligned} \tag{2}$$

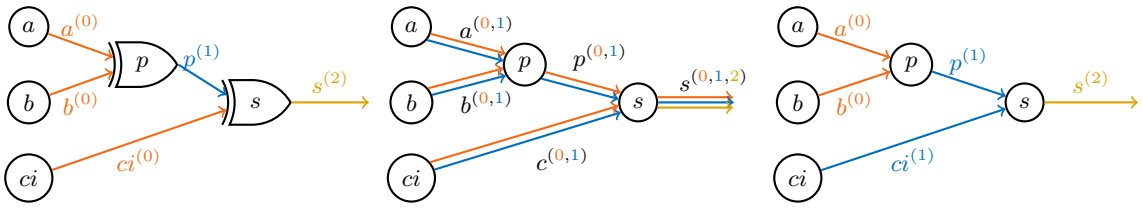
where $\mathcal{D}(i, v)$ is the distance between vertices i and v in the graph, and $\boldsymbol{h}_i^{(0)}$ is the initial feature of vertex i. The boldface indices emphasize the difference compared with a general GNN. In other words, in the k-th iteration of a depth-Δ asynchronous GNN, only those **vertices whose distance to the root v is $\Delta - k$ aggregates information** from its predecessors. Then, the aggregated embedding is **combined with their initial features** as the representation vector. In this way, unlike synchronous GNNs, messages are passed through each edge exactly only once (in the embedding of each node), which saves lots of computational efforts.

D. Putting It All Together

In previous subsections, we propose two special GNN architectural structures, namely *bidirectional* and *asynchronous*, according to the directed and acyclic properties of the target graph (DAG), respectively. The two structures are orthogonal, so that we can combine them in our final GNN architecture, asynchronous bidirectional graph neural network (ABGNN). We evaluate the performance of ABGNN in Section VII-E.

E. Related Works for DAG Embedding

There exist several works that aim to design graph learning models for DAGs. DAGNN [25] constructs a multi-layer network to generate an embedding for the whole DAG. The network is driven by the partial order induced by the DAG. However, their model is



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

	(a) Logic Simulation					(b) Synchronous					(c) Asynchronous				
	a	b	ci	p	s	a	b	ci	p	s	a	b	ci	p	s
T = 0	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓			
T = 1				✓		✓	✓	✓	✓	✓			✓	✓	
T = 2					✓					✓					✓

(d) Message passing comparison. A checkmark indicates that the node sends out message at the timestamp.

Fig. 4 A comparison between (a) distributed logic simulation, (b) synchronous GNN message passing, and (c) asynchronous GNN message passing. The table in (d) lists messages sent out by nodes at each timestamp. Asynchronous GNNs are more efficient than synchronous GNNs.

still computationally expensive since they use an iterative message passing scheme. Moreover, they use Gated Recurrent Units (GRUs) as the `combine` operator, further increasing the inference time. D-VAE [26] proposes an asynchronous message passing scheme to encode the computations on DAGs, which is the most similar work to ours. However, ABGNN differs from D-VAE since we focus on local structures and thus the generation of node-level embeddings, whereas D-VAE encodes information of the whole (computation) graph.

V. Input-output Matching

Our proposed graph learning framework identifies the boundary of arithmetic blocks. In particular, the model predicts the input wires and the output wires of arithmetic blocks. What if we want further to match the input bits with the corresponding output bits? In this section, we propose to use a network-flow-based algorithm to extract the datapaths within an arithmetic block. The problem of datapath extraction has gained great attention since it is believed datapath-aware physical synthesis may achieve higher performance. Readers are referred to [27] for a survey for datapath extraction approaches and datapath-driven placement methodologies. For now, we illustrate the feasibility of the network-flow approach for adder IO matching, and leave the other possible solutions for future work, since it is beyond the main scope of this paper.

The datapath extraction problem for an adder is defined as follows: given an (unordered) adder input set $S = A \cup B$ where $A = \{a_0, \cdots, a_{n-1}\}, B = \{b_0, \cdots, b_{n-1}\}$ and an (unordered) adder output set $T = \{t_0, t_1, \cdots, t_{n-1}\}$ such that $T[n-1:0] = A[n-1:0] + B[n-1:0]$, identify $2n$ datapaths from S and T such that **1)** all wires in S and T are covered, and **2)** each datapath starts from a_i or b_i ends at t_i. Inspired by [28], 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 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 capacity of 2. Then we run a *maximum-flow algorithm* to find the routes between S and T.

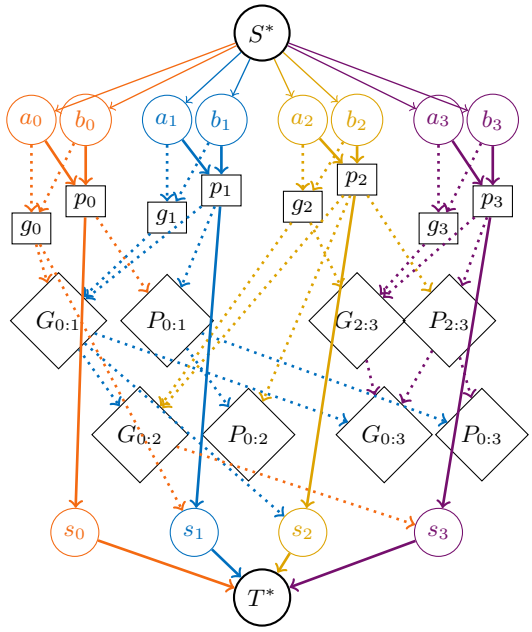


Fig. 5 A Brent-Kung adder example to demonstrate that the unique maximum flow matches inputs and outputs correctly. We analyze the flows in the order of orange, blue, yellow, to purple. Solid lines are charged with flows, and dotted lines are banned due to flow capacity constraints.

We illustrate the feasibility of the maximum-flow algorithm by taking the *Brent-Kung adder* [29] as an example. The maximal flow network is shown in Fig. 5. We analyze the flows in the order of orange, blue, yellow, to purple. The solid edges are charged with flows, while the dotted edges are banned due to flow capacity constraints. In fact, this is the **unique maximum flow solution** of the flow network. To charge the flow from s_0 to T^* (with flow value 2), the only possible route is to pass from p_0 to s_0, which occupies the edges from S^* to a_0, b_0 and up to p_0. These edges are marked

with solid orange lines, and meanwhile, the edges banned due to capacity constraints are marked with dotted orange lines, such as other edges starting from a_0 and b_0. Immediately, we observe that there is no route to charge the node g_0, which further indicates that there is no flow from g_0 to s_1, leaving (p_1, s_1) the only possible route to charge the flow from s_1 to T^*. Then we can do the same analysis for s_1 to T^* with blue lines, s_2 to T^* with golden lines, and s_3 to T^* with purple lines. Finally we will see the uniqueness of the maximum flow, and the matching is done.

How to better orchestra the network flow approach with a fuzzy matching framework deserves more discussions. Although our theoretical analysis finds the maximum flow solution unique, there is no such guarantee if we are given fuzzy, imperfect predictions of adder boundaries. However, we observe that besides input-output matching, the network flow approach also acts as a filter to remove some false alarms. In other words, if the maximum flow does not flow over some predicted input/output node, then the node is actually unlikely to be a boundary node of an adder. Inspired by so, we propose to run the maximum flow algorithm in both directions (inputs to outputs and outputs to inputs), so that the maximum forward flow (inputs to outputs) filters out false alarms of predicted outputs, and vice versa. To retain high sensitivity, we also add the siblings of predicted input (output) nodes in the forward (backward) runs, so that we are confident enough in the filter. This strategy indeed improves prediction precision with almost no sensitivity loss in our experiments.

VI. Other Algorithm Details

This section describes other important algorithm details of our arithmetic block identification flow, including discussions on the learning problem formulation, and the strategies to deal with the data imbalance issue.

A. Machine Learning Problem Formulation

As we introduced, we utilize a customized GNN for netlist representation learning. However, how to learn the parameters in the GNN model remains to be considered. Essentially, the arithmetic block identification problem is to 'detect' instances of target semantic objects in the graph, which sounds like a graph version of the *object detection* task in computer vision. Despite the intuitive descriptions, solving such problems is still 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.

Given that, we propose to formulate a *node classification* problem to circumvent the hard-to-solve graph detection problem. Specifically, the target of our neural model is to **predict boundary (input wires and output wires) of arithmetic blocks**. Another possible problem formulation is to predict the region of arithmetic blocks, which is abandoned after comparison. Note that a wire can be both an input to one arithmetic block and an output from another arithmetic block (consider the two 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 use *input prediction* and *output prediction* to refer to the two independent binary classification tasks for boundary prediction. We use an MLP with binary cross-entropy loss to consume the representation vectors generated by GNN and carry out the prediction.

B. Dealing with Data Imbalance

The data imbalance issue refers to the phenomenon that some classes (*majority*) have a significantly higher number of examples in the training set than other classes (*minority*). It is a common problem in real-life applications from various domains, which has been established to have a significant detrimental effect on training classifiers in terms of both training convergence and generalization ability [30]. For example, it is observed that the model would easily lean towards majority classes [31], making some standard metrics like *accuracy* invalid (since they may cause misinterpretation of data). We refer readers to [32] for a comprehensive review. In our dataset, the ratio of negative nodes to positive nodes is around $100:1$, which is indeed highly imbalanced.

Methods to address data imbalance can be divided into two categories, namely data-level methods and algorithm-level methods. Data-level methods aim to alter the distribution of the training dataset so that standard algorithms for balanced data can work well. On the other hand, algorithm-level methods keep the training dataset unchanged and adjust the training/inference algorithm. We now introduce three techniques we adopt 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 simply **replicates randomly selected samples from minority classes** [30]. Some more advanced oversampling methods (e.g., SMOTE [33]) have also been proposed, which we leave for possible future work.

2) Cost Sensitive Learning

Cost sensitive learning [34] assigns different penalties to different 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. Let the total loss $\mathcal{L}$ be decoupled 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 are the majority, 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 the imbalance between sample classes. The weighted loss function can be formulated explicitly as:

$$\mathcal{L} = (\mathcal{L}_{pos} + \alpha \mathcal{L}_{neg})/N, \tag{3}$$

where N is total number of samples.

VII. Experiments

A. Setup

We develop the graph object detection framework with DGL [35], a graph learning library, which is based on PyTorch [36] for tensor manipulations. The network flow algorithm (viz. *Edmonds–Karp*) is implemented with networkx [37]. We also refer to the EPFL logic synthesis libraries [38] when we reimplement the baseline methods. Graph neural networks are trained on a Linux machine with 48 Intel Xeon Silver 4212 cores (2.20GHz), 1 GeForce RTX 2080 Ti graphics card, and 32 GB main memory. Training details are discussed in subsequent sections.

B. Dataset

The dataset we use comes from open-source RISC-V CPU designs [39], including *Rocket* [40], a 5-stage in-order scalar core, and Berkeley Out-of-Order (*BOOM*) Core [40], an out-of-order

TABLE I Overall performance comparison on the test set (the *Rocket* core). Best results are emphasized with **boldface**, and second-best results are colored in blue. Our proposed arithmetic block identification method greatly improves boundary recognition performance compared with previous works. It also runs the fastest among all the methods.

Case	TETC'13 [11]			DATE'15 [3]			DATE'19 [16]			Ours		
	Input	Ouput	Runtime(s)	Input	Ouput	Runtime(s)	Input	Ouput	Runtime(s)	Input	Ouput	Runtime(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**

superscalar RV64G core. Since *BOOM* is more complicated (around 5x larger than *Rocket*), we use it as the training set, while leaving *Rocket* as the testing set.

The netlists are automatically generated from Chisel, which is further synthesized with Synopsys Design Compiler targeting the SAED 32/28nm Digital Standard Cell Library. For each design, we synthesize a set of netlists using various design constraints, so that different adder designs could be generated by DC. Statistics of the generated netlists are listed in TABLE II. In fact, there are other related constraints that could be specified, such as the radix of the prefix structure in adders or some timing constraints. In our experiments, we observe very similar outcomes as we adjust this set of constraints, so we simply omit them for simplicity.

Architecture	*Rocket*		*BOOM*	
	#gates	#wires	#gates	#wires
Brent-Kung	24340	58124	139526	366280
Cond-sum	24737	57708	138358	360455
Hybrid	25491	60287	141319	369622
Kogge-Stone	24540	57726	139005	361962
Ling	26179	62864	143903	378354
Sklansky	25208	59567	141093	369774

TABLE II Statistics of the dataset. We use *BOOM* as the training set as it is more complicated, leaving *Rocket* as the testing set. We synthesize a set of netlists for each design by specifying different adder architectures in Design Compiler.

C. Baselines

We reimplemented several representative literature works [3], [11], [16] as the baseline methods for comparison. These works have covered structural methods, functional methods, as well as machine learning methods in their proposed solutions. [11] first enumerates all cuts[1] and groups them into permutation-independent equivalence classes, which are then aggregated into candidate *words* based on *common support* or *signal propagation*. The candidate words are further propagated in the graph to form new words based on neighboring gate types. We optimistically estimate the performance upper bound of the algorithm without running *symbolic simulation* and *equivalence checking*, but simply include all the potential words instead. [3] builds `xor` trees, identifies carry-out signals, and constructs `xor`-forests based on the connection hierarchy. [16] proposes to represent circuit topology using *level-dependent decaying sum* (LDDS) *existence vector* (EV), which basically marks the gate types that appeared in a local subgraph and assigns distance-based penalty weights. We follow the LDDS-EV construction, expect that we clip all large values in the EV to 64, and add a batch normalization layer in the neural network to stabilize training. We also apply the oversampling technique by using a weighted random sampler during training. Since this method was originally evaluated for circuit classification, we adapt the method to our problem formulation and our proposed flow.

[1]The authors [11] suggested enumerating 6-feasible cuts, but our reported results are based on 5-feasible cut enumeration because it yields almost the same performance with much shorter (0.01×) runtime.

D. Overall Comparison

We first compare the performance between our proposed method and all baseline approaches [3], [11], [16] as introduced in Section VII-C. The results are listed in TABLE I. Our proposed arithmetic block identification method greatly outperforms prior works on all the testcases, averaged 95.1% and 93.7% sensitivity in input and output boundary identification, respectively. It is also the fastest method even though we run a maximum flow algorithm for input-output matching. The other machine learning approach [16] achieves the second-best performance (83.4% and 76.1% sensitivity), but its precision (around 0.35 on average) is in fact much lower than ours (over 0.94 on average). Nevertheless, it still confirms the good adaptability of deep learning methods and the effectiveness of the oversampling strategy for imbalanced datasets. [11] is able to cover lots of words composed of replicated functional bitslices, and therefore achieves acceptable sensitivity (81.9% and 56.5%), at the cost of much higher runtime (37.5× over ours). [3] 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 sensitivity (49.9% and 43.5%).

E. Evaluation of ABGNN

We conducted comprehensive experiments to evaluate our proposed graph neural network architecture and demonstrate its outstanding capability in DAG representation learning. We set the fanin depth and fanout depth of ABGNN to 1 and 5 respectively for input boundary prediction, and $(2, 2)$ for output boundary prediction.

Comparison with State-of-the-Art GNNs. We evaluate our proposed ABGNN with several state-of-the-art Graph Neural Networks, including GAT [41], GIN [18], and GraphSAGE [17], on the *Rocket* dataset. Our model achieves the best performance on all the cases with much higher recall and F1 scores, showing its superiority on DAG representation learning. In some complex cases (e.g., input prediction in the Brent-Kung case), our model outperforms other models by 5%–9% for the F1 score. On average, our model achieves 2.8%-5.0% recall gain and 3.3%-9.5% F1 score gain in input identification (TABLE III), as well as 1.9%–6.2% recall gain, and 2.6%–7.0% F1 score gain in output identification (TABLE IV).

Effect of asynchronous message passing. We conducted exper-

TABLE III Performance of different models recognizing input boundaries of adders on the test dataset (the *Rocket* core). Best results are emphasized with **boldface**, and second-best results are colored in blue. Our proposed ABGNN outperforms other models in all the test cases.

Case	GAT [41]		GIN [18]		GraphSage [17]		ABGNN (Ours)	
	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score
Brent Kung	0.906±0.009	0.910±0.015	0.809±0.019	0.873±0.018	0.915±0.021	0.915±0.017	**0.950±0.000**	**0.964±0.000**
Cond-sum	0.882±0.022	0.885±0.018	0.875±0.024	0.890±0.021	0.935±0.015	0.931±0.016	**0.949±0.000**	**0.951±0.000**
Hybrid	0.903±0.021	0.890±0.022	0.930±0.017	0.937±0.016	0.930±0.008	0.928±0.005	**0.947±0.000**	**0.949±0.000**
Kogge-Stone	0.918±0.016	0.887±0.019	0.925±0.015	0.917±0.015	0.940±0.005	0.920±0.008	**0.944±0.000**	**0.954±0.000**
Ling	0.915±0.016	0.881±0.019	0.930±0.009	0.925±0.005	0.950±0.004	0.941±0.011	**0.954±0.000**	**0.963±0.000**
Sklansky	0.901±0.021	0.895±0.022	0.935±0.009	0.938±0.011	0.928±0.012	0.930±0.006	**0.960±0.000**	**0.955±0.000**
Average	0.904±0.017	0.891±0.018	0.901±0.016	0.913±0.013	0.933±0.011	0.923±0.010	**0.951±0.000**	**0.956±0.000**

TABLE IV Performance of different models recognizing output boundaries of adders in the test dataset (the *Rocket* core). Best results are emphasized with **boldface**, and second-best results are colored in blue. Our proposed ABGNN outperforms other models in all the test cases.

Case	GAT [41]		GIN [18]		GraphSage [17]		ABGNN (Ours)	
	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score
Brent-Kung	0.845±0.019	0.870±0.020	0.906±0.011	0.921±0.012	0.925±0.025	0.923±0.016	**0.953±0.019**	**0.950±0.016**
Cond-sum	0.785±0.015	0.798±0.013	0.835±0.017	0.863±0.007	0.863±0.022	0.880±0.011	**0.866±0.015**	**0.905±0.009**
Hybrid	0.940±0.024	0.897±0.021	0.911±0.021	0.875±0.008	0.93±0.016	0.912±0.016	**0.955±0.019**	**0.939±0.016**
Kogge-Stone	0.878±0.019	0.889±0.021	0.941±0.015	0.915±0.014	0.945±0.014	0.942±0.016	**0.965±0.011**	**0.955±0.015**
Ling	0.935±0.009	0.909±0.013	0.916±0.004	0.912±0.011	0.912±0.016	0.911±0.01	**0.945±0.015**	**0.948±0.007**
Sklansky	0.865±0.016	0.855±0.016	0.898±0.019	0.894±0.011	0.932±0.013	0.919±0.018	**0.938±0.012**	**0.943±0.008**
Average	0.875±0.017	0.870±0.017	0.901±0.016	0.897±0.01	0.918±0.017	0.914±0.015	**0.937±0.015**	**0.940±0.012**

TABLE V Comparison between asynchronous and synchronous GNNs. Asynchronous GNNs reduce inference time without performance 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

iments to verify the effect of the asynchronous message passing scheme by comparing it with synchronous GNNs, while leaving other hyper-parameters the same, including the number of layers, oversampling rate, etc. TABLE V shows that compared with synchronous GNNs, asynchronous GNNs reduce inference time by 19.8% and 18.0% respectively for input and output boundary identification, without any performance degradation. Here the runtime refers to the inference time of GNN, namely the time the model takes to generate node representations. We want to emphasize that the efficiency will likely improve as the model depth increases (confirmed by our preliminary experiments), and thus the asynchronous GNN might work even better for more complicated tasks.

Effect of bidirectional information aggregation. We also carry out experiments to see the effects of bidirectional information aggregation. We build unidirectional models by reducing fanin depth to 0 for input boundary identification and fanout depth to 0 for output identification. As shown in TABLE VI, bidirectional information aggregation improves 4.6% recall and 11.1% F_1-score for the output model, as well as 1.8% recall and 2.1% F_1-score for the input model. The performance gain indicates that information from a single direction is not sufficient to identify thhe input/output boundary of an adder, and therefore combining representations learned from both directions is indeed necessary.

TABLE VI Comparison between bidirectional and unidirectional GNNs. Bidirectional GNNs outperform unidirectional GNNs, confirming the effectiveness of bidirectional informatrion aggregation.

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

VIII. 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 DAG representation learning, which outperforms existing dominantly used GNNs. We further come up with a network flow approach to match input and output wires predicted by the GNN model. Experimental results have confirmed the excellent performance of our framework: compared with state-of-the-art functional, structural and machine learning-based block mapping schemes, our framework achieves the highest sensitivity with the fastest runtime in adder identification from an open-source RISC-V CPU design (the Rocket core). We also carried out a comprehensive ablation study to analyze the effectiveness of the proposed techniques.

Acknowledgment

This work is partially supported by HiSilicon and The Research Grants Council of Hong Kong SAR CUHK14209420, CUHK14208021.

REFERENCES

[1] 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.

[2] ——, "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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik, "Template-based circuit understanding," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2014, pp. 83–90.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Computer Architecture Letters (CAL)*, vol. 31, no. 03, pp. 260–264, 1982.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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