HGNAS: Hardware-Aware Graph Neural Architecture Search for Edge Devices

Ao Zhou, Jianlei Yang, *Senior Member, IEEE*, Yingjie Qi, Tong Qiao, Yumeng Shi, Cenlin Duan, Weisheng Zhao, *Fellow, IEEE*, and Chunming Hu

***Abstract*—Graph Neural Networks (GNNs) are becoming increasingly popular for graph-based learning tasks such as point cloud processing due to their state-of-the-art (SOTA) performance. Nevertheless, the research community has primarily focused on improving model expressiveness, lacking consideration of how to design efficient GNN models for edge scenarios with real-time requirements and limited resources. Examining existing GNN models reveals varied execution across platforms and frequent Out-Of-Memory (OOM) problems, highlighting the need for hardware-aware GNN design. To address this challenge, this work proposes a novel hardware-aware graph neural architecture search framework tailored for resource constraint edge devices, namely HGNAS. To achieve hardware awareness, HGNAS integrates an efficient GNN hardware performance predictor that evaluates the latency and peak memory usage of GNNs in milliseconds. Meanwhile, we study GNN memory usage during inference and offer a peak memory estimation method, enhancing the robustness of architecture evaluations when combined with predictor outcomes. Furthermore, HGNAS constructs a fine-grained design space to enable the exploration of extreme performance architectures by decoupling the GNN paradigm. In addition, the multi-stage hierarchical search strategy is leveraged to facilitate the navigation of huge candidates, which can reduce the single search time to a few GPU hours. To the best of our knowledge, HGNAS is the first automated GNN design framework for edge devices, and also the first work to achieve hardware awareness of GNNs across different platforms. Extensive experiments across various applications and edge devices have proven the superiority of HGNAS. It can achieve up to a $10.6\times$ speedup and an 82.5% peak memory reduction with negligible accuracy loss compared to DGCNN on ModelNet40.**

***Index Terms*—Graph neural networks, hardware-aware neural architecture search, edge devices, hardware efficiency prediction.**

Manuscript received 14 September 2023; revised 8 August 2024; accepted 15 August 2024. Date of publication 23 August 2024; date of current version 8 November 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62072019 and in part by the Fundamental Research Funds for the Central Universities. Recommended for acceptance by E. Bini. *(Corresponding authors: Jianlei Yang; Chunming Hu.)*

Ao Zhou, Jianlei Yang, Yingjie Qi, Tong Qiao, Yumeng Shi, and Chunming Hu are with the School of Computer Science and Engineering, Beihang University, Beijing 100191, China (e-mail: jianlei@buaa.edu.cn; hucm@buaa.edu.cn).

Cenlin Duan and Weisheng Zhao are with the School of Integrated Circuits and Engineering, Beihang University, Beijing 100191, China.

Digital Object Identifier 10.1109/TC.2024.3449108

I. Introduction

GRAPH neural networks (GNNs) have been deemed as a promising engine of artificial intelligence (AI), achieving state-of-the-art (SOTA) performance in a wide range of real-world applications, such as node classification [1], link prediction [2], recommendation system [3] and 3D representation learning [4]. Due to the powerful feature extraction capabilities on topological structures, GNN has become a popular strategy for handling point cloud data [5], which reveals the prospect of GNN application in edge scenarios. Moreover, with the increasing popularity of 3D scanning sensors in edge devices, such as mobile phones and unmanned aerial vehicles, it is an inevitable trend to deploy GNNs on various edge devices to embrace hardware intelligence [6], [7], [8]. However, the inference process of GNNs involves both compute-intensive and memory-intensive stages, resulting in a huge computational gap between resource-limited edge devices and expensive GNNs. We deployed the popular Dynamic Graph Convolutional Neural Network (DGCNN) [9] used in point cloud processing on a Raspberry Pi, where it takes over 4 seconds to process a single frame and encounters Out-Of-Memory (OOM) problems when handling graphs with more than 1536 points [10]. Therefore, deploying GNNs on edge devices for real-time inference is extremely challenging due to resource constraints.

For the purpose of tackling the prohibitive inference cost, several handcrafted approaches have been dedicated to designing resource-efficient GNNs for point cloud processing [5], [11]. However, given the expansive design space and diverse hardware characteristics, manual optimization incurs significant computational overhead due to the extensive trial-and-error required to identify layers for optimization. In practice, manual optimization performance highly depends on human experience. To minimize manual labor and address unstable on-device performance, hardware-aware neural architecture search (NAS) has recently emerged as a promising automated technique for customizing optimal CNNs for various application scenarios [12], [13]. Inspired by these efforts, we seek to take the point cloud processing application as an opportunity to explore the prospect of hardware-aware NAS for designing efficient GNNs on edge devices. Although several studies have applied NAS techniques to GNN design [14], [15], [16], most focus solely on optimizing task accuracy while neglecting on-device efficiency. Both factors are crucial for real-world applications, especially

in resource-constrained edge scenarios. In this paper, we propose HGNAS, an efficient hardware-aware graph neural architecture search framework for edge devices to handle real-time inference requirements. Specifically, we focus on two critical efficiency metrics for edge applications: inference latency and peak memory usage.

In practice, efficient and high-quality GNN architecture exploration faces many challenges. **1) Tradeoff between efficiency and effectiveness of hardware-aware approaches**. The hardware efficiency of GNNs is influenced by various factors, including model structure, hardware sensitivity, and graph characteristics [17]. As such, hardware performance awareness strategies that estimate performance using approximate metrics, such as FLOPs, are often inaccurate [18]. While real-time on-device measurement offers more accurate results, the high overhead (communication time, on-device inference time, etc.) in evaluating numerous sub-architectures can severely impede exploration efficiency. **2) Redundancy in layer-wise GNN design space.** The design manner of stacking the same GNN layer results in operational redundancy, negatively impacting on-device inference efficiency [5]. **3) Poor search efficiency.** NAS is often criticized for its lengthy search times, particularly due to the lack of efficient exploration methods for fine-grained GNN search spaces tailored for edge applications.

To address the above issues, HGNAS integrates a novel hardware-aware technique to enable efficient GNN performance prediction. By abstracting the GNN architecture into graphs, the hardware-aware problems for GNNs can be transformed into graph-related problems, which GNNs are specialized in handling. This gives rise to the innovative concept of **Using GNN to perceive GNNs**. By leveraging a well-polished GNN-based predictor, HGNAS can effectively perceive the latency and peak memory usage of GNN candidates. In addition, we provide an efficient peak memory estimation method, leveraging GNN inference profiling to further promote the scalability and robustness of HGNAS. Furthermore, we develop a fine-grained design space composed of fundamental operations to unleash the potential of GNN computations. To improve search efficiency, HGNAS utilizes an efficient multi-stage hierarchical search strategy, mitigating the complexity inherent in exploring the fine-grained design space. As shown in Fig. 1, HGNAS has been proven superior in both latency and peak memory usage across various edge devices. The contributions of this paper can be summarized as follow:

- To the best of our knowledge, HGNAS is the first NAS framework to perform efficient graph neural architecture search for resource-constrained edge devices. HGNAS can automatically explore GNN models with multiple objectives (accuracy, latency, and peak memory usage) for targeted platforms.
- We propose an efficient GNN hardware performance predictor that perceives the latency and peak memory usage of GNNs on target devices in milliseconds. To our best understanding, HGNAS is also the first work to achieve hardware performance awareness for GNNs across platforms.

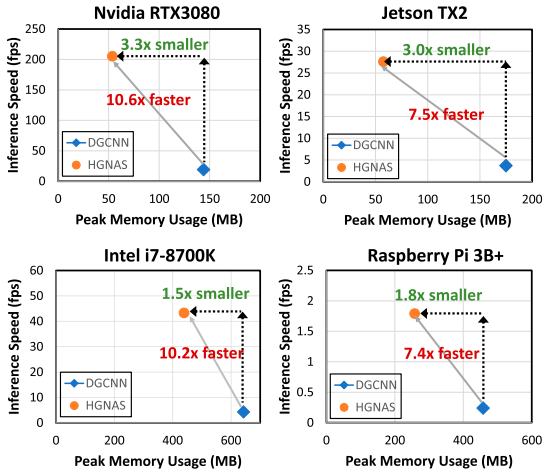

Fig. 1. Inference speed vs. peak memory usage. Our approach significantly improves hardware performance while maintaining similar accuracy to DGCNN on ModelNet40. Details are highlighted in bold in Table III.

- We provide a comprehensive analysis of memory usage during GNN inference and introduce a peak memory estimation method to improve the robustness and scalability of HGNAS.
- We propose an efficient multi-stage hierarchical search strategy to accelerate exploration in the expansive GNN fine-grained design space.
- We evaluate HGNAS on four edge devices. As expected, it can achieve up to $10.6\times$ inference speedup and 82.5% peak memory reduction with a negligible accuracy loss on point cloud classification tasks.

The rest of the paper is organized as follows. Section II introduces the background preliminaries and motivations of this paper. Section III elaborates on the proposed HGNAS framework and Section IV demonstrates the experimental results. After that, we discuss the related works in Section V. Finally, we conclude this paper in Section VI.

II. Preliminaries and Motivations

A. Graph Neural Networks

Generally, GNN architecture design follows the Message Passing (MP) paradigm. Fig. 2 illustrates a typical GNN pipeline using the example of DGCNN [9]. Each GNN layer consists of *sample*, *aggregate*, and *combine* operations. Specifically, the *sample* operation constructs the graph from the point cloud data for processing, the *aggregate* operation facilitates message propagation among nodes, and the *combine* operation updates all node features. Note that *sample* operations are also executed during the inference process to extract the graph structure from the point cloud data. The propagation rule of DGCNN at layer k is defined as follows:

$$x_i^k = \sum_{j \in \mathcal{N}(i)} h_\Theta^{(k)} \left(x_i^{(k-1)} \left\| \left(x_j^{(k-1)} - x_i^{(k-1)} \right) \right. \right), \quad (1)$$

where $h_\Theta^{(k)}$ denotes a neural network, i.e. a MLP. A GNN model is constructed by stacking GNN layers sequentially.

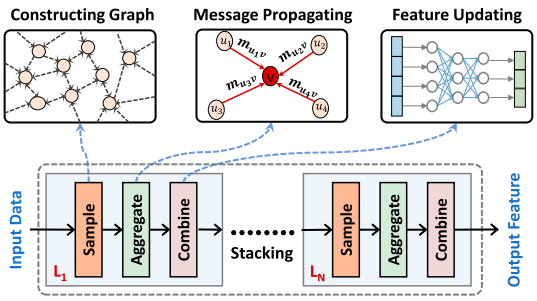

Fig. 2. Typical GNN pipeline with MP paradigm.

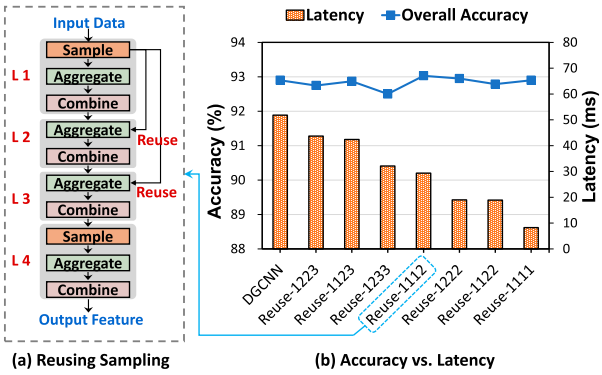

Fig. 3. (a) Example of reusing sampling results between layers (Reuse-1112). (b) Accuracy and latency comparison when performing sampled results reuse among different DGCNN layers on ModelNet40 [19] dataset.

While this design approach is straightforward, its rigid sequence of operations and repetitive layer stacking limit the scope for innovation, thereby constraining potential performance breakthroughs in GNN models.

B. Observations and Motivations

Drawing on prior research, we offer key observations that inspire efficient GNN exploration for edge devices.

Observation 1: Redundant operations introduced in GNN architecture design bring significant overhead. As previously described, GNN models designed following the MP paradigm are formed by stacking GNN layers which possess a fixed sequence of operations. This naturally raises the question: Do all the elements within each layer contribute to the final performance of the GNN model? To demonstrate the redundancy, we conduct inter-layer reuse experiments with DGCNN on Nvidia RTX3080. Specifically, we remove the *sample* operations within the latter GNN layers and reused the sampling results from the front layers. An example of this reuse is shown in Fig. 3(a). The experimental results, depicted in Fig. 3(b), demonstrate that reusing the sampling results has no significant impact on accuracy but remarkably enhances the computational efficiency. This observation is consistent with the phenomenon found in [5], fully demonstrating the considerable overhead introduced by redundant operations in GNN design, posing a major obstacle to GNN computational efficiency optimization.

To eliminate redundant operations in GNNs, [5], [11] propose identifying and manually simplifying the model structure through numerous ablation experiments and analyses, achieving notable speedup. However, this manual optimization requires extensive trial-and-errors and heavily relies on specialized expertise. On the other hand, some researchers decoupled the MP paradigm to construct a more flexible design space, achieving SOTA performance [20], [21]. Inspired by the GNN paradigm decoupling, we aim to decouple the GNN layer into operations and construct a fine-grained design space, allowing various configurations (e.g., aggregation range, operation order) to be generated through learning instead of manual effort.

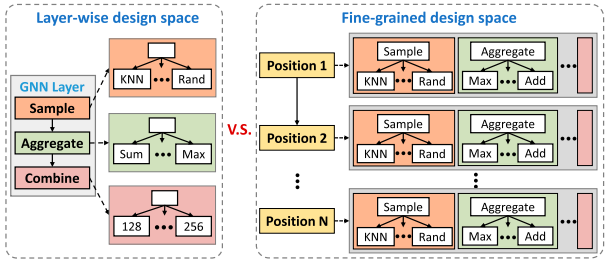

Fig. 4. Fine-grained design space greatly expands the scope of architectural exploration.

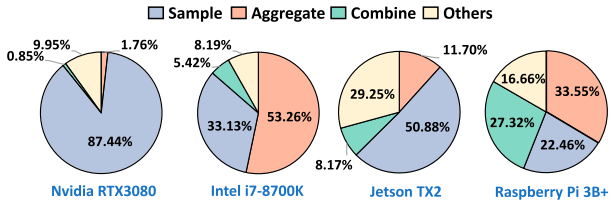

Fig. 5. Execution time breakdown of DGCNN across various edge devices on ModelNet40.

Observation 2: Exploring the GNN fine-grained design space is costly. Fig. 4 illustrates a comparison between the fine-grained design space and the layer-wise design space, highlighting the remarkable expansion of the architectural exploration scope. Specifically, the fine-grained design space comprises fundamental GNN operations, which covers sub-architectures that grow exponentially with the number of positions. For example, the backbone of DGCNN consists of four GNN layers, with each layer comprising three fundamental operations: *sample*, *aggregate*, and *combine*. Therefore, to cover most DGCNN variants, the fine-grained design space contains at least 12 positions, 3 candidate operations, and at least N functions for each operation (details in Sec. III-C). Consequently, the design space contains a staggering $(3N)^{12}$ possible configurations, thereby exacerbating the complexity of exploration.

The expansion of the GNN design space increases flexibility but also complicates exploration, making manual optimization challenging. As an AutoML technique, NAS can automatically locate the optimal design without navigating all candidates. However, NAS continues to grapple with significant efficiency challenges, exemplified by instances requiring up to 2,000 GPU-days for a single search [22]. As such, exploring the fine-grained design space requires a more efficient search strategy. In practice, the contribution of different layers towards the overall accuracy within a model may vary significantly [11], [23]. Leveraging this insight, we aim to improve exploration

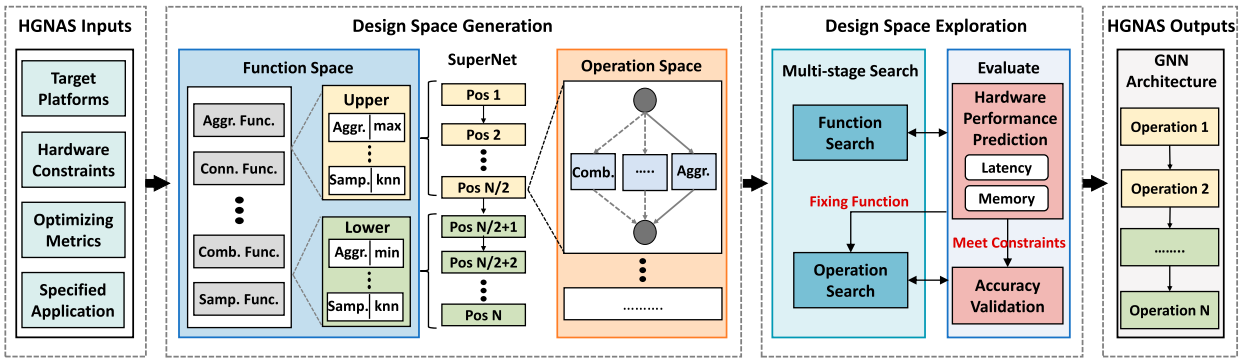

Fig. 6. Overview of the proposed HGNAS framework. HGNAS aims to search for top-performing GNN architectures that excel in both accuracy and efficiency.

efficiency by decoupling the design space and guiding search orientation at different positions.

Observation 3: The same GNN model may behave differently across various computing platforms. A detailed breakdown of DGCNN execution time across platforms is illustrated in Fig. 5, using data obtained by PyTorch Profiler. For Nvidia RTX3080 and Jetson TX2, the *sample* operation occupies the majority of execution time. This is because GPUs are better at handling compute-intensive matrix operations, and not so good at memory-intensive graph sampling operations. For Intel i7-8700K, *aggregate* and *sample* dominate the execution time, due to irregular memory accesses patterns. These observations confirm that DGCNN execution is largely `I/O-bound` on these platforms. Conversely, the resource constraints of the Raspberry Pi result in a `compute-bound` execution, as all three operations are time-consuming. Therefore, GNN models running on different devices exhibit varying hardware sensitivities that must be carefully considered during architecture design.

In practice, inference efficiency and accuracy are equally important for GNN design in edge scenarios [5]. Additionally, the hybrid execution mode of GNNs, which consists of both memory-intensive and compute-intensive operations, poses great challenges for effectively perceiving GNNs hardware performance [17]. Recent studies have explored analytical estimation to improve hardware efficiency for hardware/algorithm co-designs [24], [25], [26], [27], [28], [29]. However, these approaches are often tailored for specific hardware architectures, limiting their applicability across diverse edge computing platforms [30]. Therefore, an efficient and scalable hardware-aware approach is highly desirable. Inspired by [12], a GNN-based hardware performance predictor is integrated to efficiently and accurately perceive GNN hardware performance across various platforms.

III. Methodology

A. HGNAS Overview

This section provides an introduction to the proposed HGNAS framework, as illustrated in Fig. 6. We begin with the problem definition, in which we integrate hardware performance metrics to guide the exploration process, ensuring that the designed GNNs are both efficient and accurate. Given the target edge devices, hardware constraints, specified dataset, and the optimizing metrics, HGNAS will first generate a *fine-grained hierarchical design space*, comprising the *Function Space* and *Operation Space*. HGNAS subsequently constructs a supernet to cover the GNN design space, allowing the search and training processes to be decoupled by one-shot exploration. Afterward, HGNAS navigates the hierarchical design spaces using the proposed *multi-stage hierarchical search strategy*. During exploration, each candidate architecture is evaluated based on both its accuracy on the validation dataset and its hardware performance on the target device. GNN hardware performance is assessed using our proposed *GNN hardware performance predictor*, obviating the need for laborious on-device measurements. In the following sections, we will detail the key components of the HGNAS framework.

B. Problem Definition

For simplicity, we first introduce the notations that will facilitate the subsequent exposition, as illustrated in Table I. In this paper, we aim to co-optimize the accuracy and hardware efficiency of GNNs deployed on edge devices. Hardware efficiency for GNNs is quantified by two key metrics: inference latency Lat and peak memory usage PM, both of which are critical considerations for edge applications. Given a target edge device $\mathcal{H}$, along with latency constraint C_{lat}, and the peak memory constraint C_{mem}, we can formulate the multi-objective optimization task for HGNAS as follows:

$$\arg \max_{\{\mathcal{A},\mathcal{H}\}} \left(\alpha * acc_{val}\left(\mathcal{W}^*, \mathcal{A}\right) - \beta * \mathcal{E}\left(\mathcal{A}, \mathcal{H}\right)\right), \tag{2}$$

$$s.t.\ \mathcal{W}^* = \arg\max_{\mathcal{W}} acc_{train}\left(\mathcal{W}, \mathcal{A}\right) \quad Lat < C_{lat} \quad and \quad PM < C_{mem}, \tag{3}$$

where $\mathcal{W}$ denotes the model weights, acc_{train} is the training accuracy, acc_{val} is the validation accuracy, $\mathcal{E}$ is the hardware efficiency on targeted platform $\mathcal{H}$, $\mathcal{A}$ is the GNN architecture candidate, α and β are scaling factors employed to balance the optimization objectives between accuracy and efficiency.

TABLE I
NOTATIONS OF THE INVOLVED MATHEMATICAL SYMBOLS AND CORRESPONDING DESCRIPTIONS

Term	Description
$\mathcal{A}$	GNN architectures
$\mathcal{H}$	target edge device
Lat	inference latency
PM	peak memory usage
C_{lat}	latency constraint
C_{mem}	peak memory constrain
acc_{val}	validation accuracy
$\mathcal{E}$	hardware efficiency
N	number of positions in GNN supernet
$\mathcal{F}_{\text{obj}}$	objective function during search
α	scaling factor for accuracy
β	scaling factor for efficiency

This ensures that the designed GNN aligns with the specific requirements of the target application. Note that both latency and peak memory usage are jointly influenced by the GNN architecture as well as the platforms where such networks are deployed.

C. Fine-Grained Hierarchical GNN Design Space

The traditional layer-wise search space aims to find an optimal block and stack multiple instances of it to construct the optimal architecture, which significantly constrains exploration within the design space. Additionally, the computational characteristics of GNNs make the sequence of operations significantly influence efficiency [17]. Notably, the sequence identified in the optimal block may not be universally applicable for optimizing all components of the model. To unleash the potential of efficient GNN design, we propose a fine-grained hierarchical design space, comprising all the fundamental operations involved in GNN computation. Furthermore, we decouple the operations by organizing the design space into two hierarchical layers: a *Function Space* and an *Operation Space*.

The GNN supernet. To lift the restrictions of the traditional GNN design space, HGNAS adopts a more flexible approach by building the design space based on positions for GNN operations, rather than presetting the number of GNN layers. Furthermore, as shown in Fig. 6, HGNAS organizes the GNN design space as a supernet based on positions, in order to minimize exploration overhead through the utilization of the single path one-shot NAS methodology [31]. Candidate architectures are generated by choosing and fixing an operation and its corresponding function at each supernet position. Specifically, for each position in the supernet, there are four fundamental operations including *connect*, *aggregate*, *combine*, and *sample*, each with distinct properties. Beyond the operations inherited from the MP paradigm, the *connect* operation, which includes both direct and skip-connections, further enhances the design freedom in constructing GNN models. Besides, the message type properties in aggregate operations dictate the construction method of messages to be aggregated for point cloud processing. In practice, supernet training demands uniform lengths for hidden dimensions across all operations situated at each position. For dimension alignment, HGNAS adds linear transformations to operations, such as *sample* and *aggregate*, that are otherwise incapable of modifying hidden dimensions. These linear transformations will be omitted in the finalized architecture to avoid introducing additional overhead.

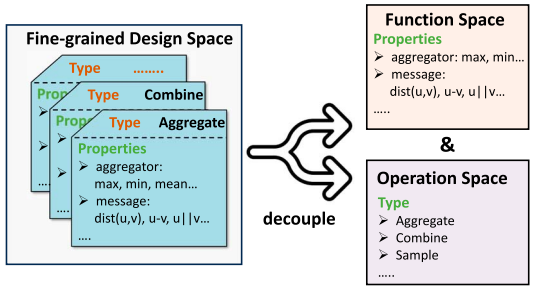

Fig. 7. Design space decoupling for GNN.

TABLE II
THE AVAILABLE CHOICES IN GNN'S SUPERNET

Operation	Function
Connect	Skip-connect, Identity
Aggregate	Aggregator type: sum, min, max, mean
	Message type: Source pos, Target pos, Relative pos, Source ‖ Relative pos, Target ‖ Relative pos, Euclidean distance, Full
Combine	8, 16, 32, 64, 128, 256
Sample	KNN, Random

The hierarchical design space. In practice, operations in the fine-grained design space can be further decoupled. For example, the *aggregate* operation encompasses various properties, including the aggregation operator and message construction type. Consequently, HGNAS decouple the fine-grained GNN design space into *Operation Space* and *Function Space*. As illustrated in Fig. 7, the *Operation Space* includes operation types, while the *Function Space* comprises specific operation properties. In addition, these two sub-spaces can be explored separately by leveraging the proposed multi-stage hierarchical search strategy (see Sec. III-D) to reduce exploration complexity. All candidate operations and functions are listed in Table II.

D. Multi-Stage Hierarchical Search Strategy

To tackle the excessive exploration complexity introduced by fine-grained design space (see **Observation 2**), we propose an efficient multi-stage hierarchical search strategy that divides the search process into two stages, corresponding to *Function Space* and *Operation Space*. Inspired by [31], we employ a one-shot approach during the two-stage search process to decouple the supernet training and architecture search, avoiding the exorbitant cost of sub-architecture retraining. In particular, full supernet training is only performed once after determining the optimal function settings. During the search phase, candidates meeting hardware constraints are evaluated for accuracy by inference on the validation set using the pre-trained weights of the supernet. As illustrated in Alg. 1, the algorithm inputs

Algorithm 1: Multi-stage hierarchical search strategy.

Inputs: population size P, hardware constraints $\mathcal{C}$, target device $\mathcal{H}$, *Operation Space* $\mathbb{S}_{op}$, *Function Space* $\mathbb{S}_f$, max iteration T, number of positions N.
Outputs: the best found GNN design $\mathcal{A}^*$ for target device $\mathcal{H}$.
Initialize GNN supernet $\mathcal{N}_{super}$ with N positions and two function sets $upper \leftarrow \emptyset$, $lower \leftarrow \emptyset$
/* Stage 1: Function search */
Assign function set: $\mathcal{N}_{super}[0, N/2] \leftarrow upper$, $\mathcal{N}_{super}[N/2+1, N] \leftarrow lower$
for $1 \leq t \leq T$ **do**
  $\{upper, lower\} \leftarrow EA(P, \mathcal{N}_{super}, \mathbb{S}_f, obj = max(acc_{val}))$
end
Fix function set $\mathbb{F} \leftarrow \{upper, lower\}$ for $\mathcal{N}_{super}$
Re-initialize and pre-train $\mathcal{N}_{super}(\mathbb{S}_{op}, \mathbb{F})$
/* Stage 2: Operation search */
Initialize operation set $\mathcal{O} \leftarrow \emptyset$
for $1 \leq t \leq T$ **do**
  $\mathcal{O} \leftarrow EA(\mathcal{P}, \mathcal{N}_{super}, \mathbb{F}, S_{op}, obj = max(\mathcal{F}_{obj}(\mathcal{C})))$
end
return optimal architecture $\mathcal{A}^* \leftarrow \{\mathcal{O}, \mathbb{F}\}$

include: (1) population size P in evolutionary search, (2) hardware constraints $\mathcal{C}$ which will determine the upper bounds of latency and peak memory usage in the finalized GNNs, (3) the target edge device $\mathcal{H}$, (4) *Operation Space* $\mathbb{S}_{op}$, *Function Space* $\mathbb{S}_f$, (5) maximum iterations T which determines the terminating conditions, and (6) the number of positions N which affects the exploration scope. During exploration, HGNAS first searches for an optimal function setting in the *Function Space*. After that, the GNN supernet is pre-trained based on this optimal setting. Subsequently, a multi-objective search is conducted across all positions in the supernet for optimal operations. Finally, the algorithm outputs the top-performing model $\mathcal{A}^*$ tailored for the target edge device $\mathcal{H}$. The details of the multi-stage search strategy are outlined below.

Stage 1: Function search. In this stage, HGNAS seeks to identify a function setting that maximizes supernet accuracy. During the search, HGNAS utilizes an evolutionary algorithm (EA) to iteratively select sub-functions from the GNN supernet. The score of each sub-function is determined by the corresponding supernet accuracy, requiring only a few training epochs. To further improve the exploration efficiency, HGNAS partitions the N positions of the GNN supernet into two halves, sharing one set of functions among the *Upper* half $(0, ..., N/2)$ and another set among the *Lower* half $(N/2+1, ..., N)$. This sharing scheme is inspired by the differing contributions of the front and latter GNN layers to accuracy [11]. Although this approach risks overlooking some promising architectures, the considerable gain in exploration efficiency justifies its use. For a supernet with 12 positions, through sharing functions among positions in the decoupled design space, HGNAS can reduce the number of exploration candidates from $\mathbf{4.2 \times 10^{12}}$ to $\mathbf{1.7 \times 10^{7}}$. Finally, an optimal function set $\mathbb{F}$ is determined for initializing the supernet $\mathcal{N}_{super}$. Note that fixing $\mathbb{F}$ will significantly reduce the complexity of subsequent exploration.

Stage 2: Operation search. Upon fixing $\mathbb{F}$, we pre-train the GNN supernet to obtain shared weights for all sub-architectures, thus avoiding retraining. During this stage, HGNAS explores the remaining *Operation Space* with the aim of locating a set of operations to maximize the accuracy and efficiency of the GNN candidate on the target device. Specifically, the objective function during the operation search is formulated as:

$$\mathcal{F}_{\text{obj}}(\mathcal{C}) = \begin{cases} 0, & \text{if } \mathcal{E} \geq \mathcal{C} \\ \alpha * acc_{val} - \beta * \mathcal{E}, & \text{if } \mathcal{E} < \mathcal{C} \end{cases} \tag{4}$$

where $\mathcal{E}$ represents the hardware efficiency, including latency and peak memory usage. For GNN architectures that fail to meet the hardware constraints $\mathcal{C}$, we will not further evaluate the accuracy and directly mark zero to avoid the unqualified GNNs. The evaluation of hardware efficiency is based on the proposed GNN hardware performance predictor (see Sec. III-E), which can perceive the latency and peak memory usage of candidate GNNs on the target device in milliseconds. By adjusting α and β, we can easily direct the search trend (towards more accurate or more efficient) to serve the requirements of different application scenarios.

E. GNN Hardware Performance Prediction

By abstracting the GNN architecture into graphs, the GNN hardware-awareness problem can be reformulated as a graph representation learning problem, an area where GNNs excel. As such, we propose an efficient GNN hardware performance predictor to learn the relationship between GNN architectures and hardware efficiency. In addition, we perform an intensive analysis of the GNN computation process and propose a peak memory usage estimation method to assist in predicting GNN's run-time peak memory usage. As shown in Fig. 8, the prediction process consists of the following phases: *graph construction*, *node feature generation*, *latency prediction*, *peak memory usage prediction*. Note that the proposed GNN hardware performance predictor is only applied during the search process, whereas the experiment results in Sec. IV are directly measured on target edge device for fair comparisons.

Graph construction. During this phase, HGNAS abstracts GNN architectures into directed graphs, which serve as the input for the GNN predictor. While GNNs typically use undirected graphs for better connectivity, we choose directed graphs because their unique dataflow properties significantly impact prediction accuracy. Specifically, the nodes in these architecture graphs represent inputs, outputs, and operations, while the edges depict the dataflow within the GNN architecture. In practice, accurate prediction of hardware efficiency requires both candidate model architecture and the graph property of the input dataset, which GNN execution highly depends on. However, this plain abstraction of the original GNN architectures is too sparse for the predictor to obtain sufficient structural features, and lacks the necessary information on input data. To address

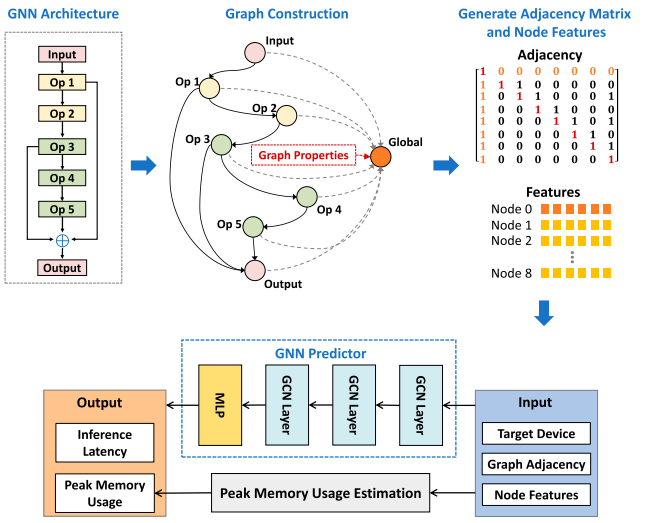

Fig. 8. Latency and peak memory usage prediction of a candidate GNN model for the target device.

this limitation, HGNAS introduces a global node that connects with all the other nodes in the graph to improve the graph connectivity. In this way, the propagation of operational information throughout the entire architecture graph is significantly improved, which benefits the learning of the GNN predictor. Finally, HGNAS will output an architecture graph in adjacency matrix format $G \in \mathbb{R}^{N \times N}$, where N denotes the number of nodes.

Node feature generation. For the node feature initialization, HGNAS employs the one-hot strategy commonly used in GNNs [1] to encode the possible candidates at each position. For an operation node, the node feature comprises the operation type and its corresponding function. Specifically, HGNAS encodes these two components into 7-dimensional and 9-dimensional one-hot vectors, respectively, and subsequently concatenates these vectors to form the node feature. For input and output nodes, HGNAS assigns them with zero vectors, indicating that they do not have any specific operation associated with them. Regarding the global node, HGNAS encodes the input graph data properties (such as the number of nodes, density, etc.) into a 16-dimensional vector as the global node feature. Afterwards, a node feature matrix $X \in \mathbb{R}^{N \times L}$ will be generated as input to the GNN predictor, where L is the feature length.

Latency prediction. To avoid the over-smoothing problem often induced by deeper GNNs on small-scale graphs (i.e., the abstracted architecture graph), the latency predictor consists of only three GCN layers [1] and a multi-layer perceptron (MLP). Specifically, the GCN layers utilize the sum aggregator with hidden dimensions of $256 \times 512 \times 512$. For the MLP part, we employ three fully-connected layers with hidden dimensions of $256 \times 128 \times 1$ respectively, followed by a LeakyReLU activation function σ for generating a scalar prediction of latency. The predictor takes target edge device $\mathcal{H}$, adjacency matrix $\mathcal{G}$, and node feature matrix $\mathcal{X}$ as inputs, and outputs the predicted latency. A one-hot encoding method is used for the target device input. When integrating new devices, only a small amount of data collection is required, followed by incremental training of the predictor. The GNN-based latency prediction can be formulated as follows:

$$Lat\left(\mathcal{A}\right) = \sigma\left(MLP\left(GCN\left(\mathcal{A}, \mathcal{G}, \mathcal{X}, \mathcal{H}\right)\right)\right) \tag{5}$$

The predictor is trained for 250 epochs on 30K randomly sampled candidate architectures (21K for training and 9K for validation) in our fine-grained GNN design space, with labels obtained from measurement results on various edge devices. During the predictor training, we utilize the *mean absolute percentage error* (MAPE) as the loss function to mitigate the impact of potential outliers, such as system disruptions during data collection. Additionally, we utilize `AdamW` as the training optimizer, employing a batch size of 32 and an initial learning rate of 0.0008, which is dynamically adjusted using the `ReduceLROnPlateau` scheduler. As architecture graphs typically contain only a few dozen nodes, the prediction overhead is mostly negligible, requiring only about 10 ms on GPU platforms.

Peak memory usage prediction. In the design of efficient GNNs for edge applications, peak memory usage serves as a critical metric, substantially influencing the feasibility of such networks on the target platform. Therefore, we extend the GNN hardware performance predictor to include peak memory usage prediction during the search process, thereby mitigating the risk of unsuitable design choices. For data preparation, we collect 30K random GNN architectures from our fine-grained GNN design space, employing the same dataset splitting strategy for latency prediction. For GPU devices, we utilize the `max_memory_allocated` function in PyTorch [32] to obtain peak memory usage data. For CPU devices, we perform data collection using the `Resource` tool [33]. As we observed, data collection accuracy directly affects prediction accuracy. Hence, we independently sample the peak memory data of each architecture to minimize interference. Furthermore, we employ the same predictor structure and training scheme as for latency prediction, while adjusting the initial learning rate to 0.0003 and batch size to 16. In practice, initializing the peak memory predictor using the weights of the latency predictor can expedite the model convergence, thereby suggesting an interrelation between the two performance metrics (see Sec. IV-E). As such, we utilize the well-trained latency predictor's weights to initialize the peak memory predictor. This approach enables the peak memory predictor to reach training completion within tens of epochs while attaining performance on par with the latency predictor.

Peak memory usage estimation. Unlike the runtime latency of GNNs, influenced by multiple factors, peak memory usage is closely associated with the size of the tensor generated during inference. This correlation allows us to develop an accurate method for estimating peak memory usage for a wider range of platforms with enhanced evaluation robustness. To this end, we profile several randomly sampled architectures from the fine-grained design space and summarize the rules governing the patterns of their peak memory usage. During GNN inference, memory usage can be broadly divided into three categories:

those related to the model, the dataset, and intermediate variables. The model-related memory usage M_p is approximately equal to $U_k \times N_p$, where U_k and N_p are the precision and number of the model parameters. And dataset-related Memory usage M_d is determined by the volume of data required for the inference of a single batch. Upon loading both the model and the data, the total memory usage M is given by $M = M_p + M_d$. In addition, the memory usage of intermediate variables are generated during each operation in the forward execution. For clarity, we henceforth use M to denote post-execution memory usage and M' for pre-execution memory usage of the current operation. Specifically, the memory consumption brought by some typical GNN operations are delineated below.

(1) ***Sample.*** For this operation, the intermediate result takes the form of a graph that is represented as an edge list. Assume that there are N nodes in the graph and K neighbors are sampled for each node, the memory usage of *sample* operation can be formulated as follows:

$$M_{sample} = N_e \times 2 \times U_{index}, \tag{6}$$

where N_e, which equals to $N \times K$, is the number of edges in the graph and U_{index} is the precision of the edge index. After executing *sample* operation, the memory usage M is equal to $M' + M_{sample}$.

(2) ***Aggregate.*** The *aggregate* operation is executed in two phases: message construction and message broadcasting. The former generates messages on the edges, while the latter updates the node features through message passing. By assuming the node feature length as L, the introduced memory usage can be calculated as follows:

$$M_{msg} = N_e \times 2 \times L \times U_k, \tag{7}$$

$$M_{broad} = N \times L \times U_k. \tag{8}$$

Upon completing the message construction phase, the updated memory usage M is equal to $M' + M_{msg}$. Additionally, the memory usage after the completion of the message construction phase is denoted as M_{mc} for subsequent peak memory calculations. In practice, memory allocated for messages is automatically recycled after message broadcasting. Consequently, the memory usage M is updated after the *aggregate* operation as follows: $M = M' + M_{broad} - M_{msg}$.

(3) ***Combine.*** This operation is typically implemented based on MLP, introducing memory usage that can be straightforwardly calculated as follows:

$$M_{com} = N \times L_{out} \times U_k, \tag{9}$$

where L_{out} is the output feature dimension of MLP. Upon execution of this operation, the updated memory usage M becomes $M' + M_{com}$.

The total memory usage of a candidate GNN architecture can be determined by accumulating the memory consumed in each successive operation during forward computation. Nevertheless, the peak memory usage during GNN inference may not necessarily align with M. Explicit message construction in graphs with numerous edges and high-dimensional node features can consume considerable memory, not accounted for in M. In practice, this phase is often where peak memory usage is observed. Thus, GNN's final peak memory usage can be calculated as follows:

$$PM = Max(M, M_{mc}^{(1)}, ..., M_{mc}^{(n)}), \tag{10}$$

where n is the number of *aggregate* operations in GNNs.

In summary, this estimation method allows for direct calculation of peak memory during GNN inference across different GPU devices. Note that this approach does not directly assess peak memory on CPU devices, as memory usage on these devices encompasses both program execution data and Tensor data. Nevertheless, the method remains useful for assessing the relative peak memory consumption of candidates during exploration, irrespective of the device targeted. Practically, HGNAS integrates both the predictor output and estimation metrics to improve evaluation robustness throughout the search process. When the predictor output is lower than the estimated value, indicating a biased prediction, the estimated value is used to measure the peak memory usage of the candidate architecture.

IV. Experiment

A. Experimental Settings

Baselines and datasets. To evaluate HGNAS, we consider two different application datasets for the graph classification task: the point cloud processing benchmark ModelNet40 [19] and the text analysis dataset MR [34]. Evaluation and hyperparameter settings are based on [11] and [16], respectively. Our comparison included several baselines: (1) the popular point cloud processing model DGCNN [9], (2) two manually optimized variants of DGCNN [5], [11], (3) two SOTA methods for optimizing point cloud classification accuracy PointGS [35] and LGFNet [36], and (4) the GNN NAS framework PAS [16]. For a fairer comparison with manual optimizations in [5], [11], we adopt their reported accuracy, inference speedup, and memory reduction as the baseline on the GPU platform. For other edge platforms, we reproduce these baselines based on PyTorch Geometric (PyG) framework [37], due to the lack of pre-trained models and evaluation results. To compare with PAS, we use the PAS_G model, which is the relatively lightweight optimal searched GNNs, for our testing. In addition, all experimental and profiling results are obtained using the PyG framework, taking the average results of 10 runs.

HGNAS settings. We assign 12 positions for the GNN supernet to cover DGCNN architectures. For design space exploration, the maximum number of iterations is set to 1000, and the population size for the evolutionary algorithm (EA) is fixed at 20. Both the search and training phases of HGNAS are carried out on an Nvidia V100 GPU. During function search and operation search, the number of GNN supernet training epochs is set as 50 and 500, respectively.

Predictor settings. The predictor is trained for 250 epochs using 30K randomly sampled architectures from our fine-grained design space, where 21K of these are adopted for training and 9K for validation. Labels for these architectures are obtained from performance measurements on various edge

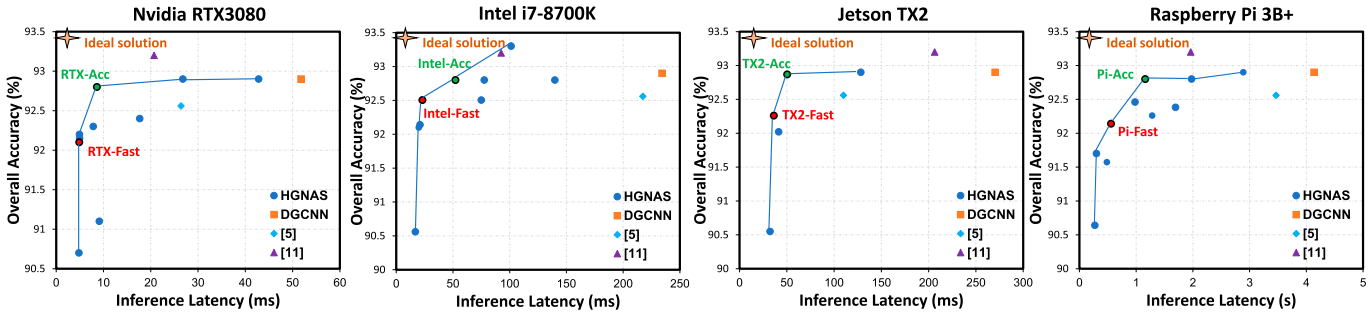

Fig. 9. Comparison between existing networks and HGNAS across various devices.

devices. The performance of the predictor is evaluated using MAPE and validation accuracy as metrics.

Edge devices. We employ four edge devices for comparing HGNAS and competitors: (1) Nvidia RTX3080 with 10GB memory, (2) Intel i7-8700K, (3) Jetson TX2 with 8GB memory, (4) Raspberry Pi 3B+ with a Cortex-A5 processor and 1GB memory. Note that the hardware performance of both competitors and HGNAS is derived from real GNN inference executions on the specified devices.

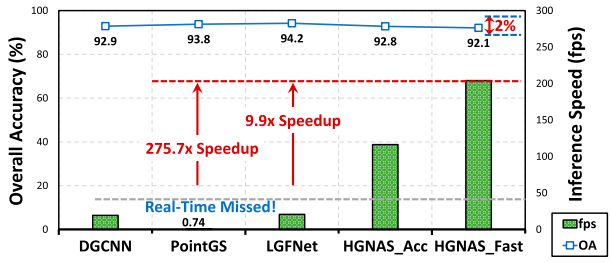

Fig. 10. Accuracy and efficiency trade-offs in GNN designs, highlighting substantial gains with minimal accuracy loss.

B. Evaluation on ModelNet40

1) Accuracy vs. Efficiency: Fig. 9 depicts the outcome of HGNAS exploration, aiming for reduced latency and enhanced accuracy. The ideal solution is designated by a star and positioned in the top-left corner of the figure. The green markers labeled Device_Acc (e.g., RTX_Acc) represent architectures optimized for specific devices by HGNAS without compromising accuracy. In contrast, the red markers labeled Device_Fast permit a 1% drop in accuracy. Note that the original intention of HGNAS is to design efficient GNN models for edge devices. As such, we have applied a larger scaling factor β to the hardware efficiency metrics during search, with the objective of identifying ultra-efficient architectures that still adhering to accuracy requirements. The results demonstrate that HGNAS consistently maintains a better performance frontier on various devices, which is guaranteed by the accurate hardware performance prediction of the candidate GNNs during the search. By carefully selecting scaling factors, HGNAS can easily balance hardware efficiency and task accuracy, as detailed in Sec. IV-D.

In real-time applications, efficiency is as crucial as accuracy. Significant progress in enhancing GNN accuracy does not mitigate their primary deployment obstacle on edge devices: inefficient inference. As shown in Fig. 10, PointGS and LGFNet prioritize optimizing accuracy, significantly improving point cloud classification. However, this emphasis on accuracy compromises inference efficiency, leading to higher on-device latency and failing to meet real-time requirements. Conversely, HGNAS incorporates efficiency metrics into the NAS objective, enabling the creation of a GNN model that achieves inference efficiency gains with minimal accuracy loss, meeting real-time constraints. Specifically, HGNAS demonstrates a balance between accuracy and efficiency, with less than 2% accuracy loss compared to PointGS and LGFNet, while achieving speedups of $275.7\times$ and $9.9\times$ on the RTX3080 platform, respectively.

2) HGNAS Over Existing Graph Neural Architectures: In this set of experiments, we benchmark HGNAS against competitors across four edge devices. For HGNAS, we conduct separate explorations prioritizing accuracy, latency, and peak memory usage by assigning them larger scaling factors. The results are summarized in Table III. It can be observed that GNN models designed by HGNAS show improved hardware efficiency in terms of reduced latency and peak memory usage across various edge computing platforms while maintaining similar accuracy levels. Specifically, HGNAS achieves speedups of $6.0\times$, $4.5\times$, $5.3\times$, and $3.6\times$, all while maintaining the same accuracy as DGCNN on the four edge devices. With a permissible 1% accuracy loss, the Device_Fast GNNs designed by HGNAS achieve speedups of up to $10.6\times$, $10.2\times$, $7.5\times$, and $7.4\times$. This superior performance is attributed to the accurate latency predictions during the model search process. In practice, only achieving latency awareness is insufficient to meet edge application requirements, and peak memory awareness is important for memory-constrained scenarios. Therefore, based on our previously published work [10], we propose two new approaches in this paper: one for peak memory prediction and another for peak memory estimation, to guide the exploration. By incorporating peak memory metrics into the multi-objective optimization process, the Device_Small architecture is explored to achieve lower peak memory usage on each device. As shown in Table III, the Device_Small architectures achieve peak memory reductions of 82.5%, 38.0%, 81.4%, 43.7% across the four edge devices, outperforms other competitors and the Device_Fast architectures. This substantially enhances the viability of deploying GNNs on edge devices with limited resources. Furthermore,

TABLE III
COMPARISON OF HGNAS AND EXISTING METHODS, WHERE OA AND MACC DENOTE OVERALL ACCURACY AND BALANCED ACCURACY, RESPECTIVELY

Device	Network	Size [MB]	OA	mAcc	Latency [ms]	PM [MB]
RTX	DGCNN	1.81	92.9	88.9	51.8	174.9
	[5]	-	92.6	89.6	(2.0×↑)	(51.9%↓)
	[11]	-	93.2	90.6	(2.5×↑)	-
	RTX_Acc	1.61	92.8	90.1	8.6 (6.0×↑)	60.5 (65.4%↓)
	RTX-Fast	**1.46**	**92.1**	**88.5**	**4.9 (10.6×↑)**	**53.8 (69.2%↓)**
	RTX_Small	1.49	92.4	89.3	4.9 (10.6×↑)	30.7 (82.5%↓)
Intel	DGCNN	1.81	92.9	88.9	234.2	643.0
	[5]	2.33	92.6	89.6	217.4 (1.1×↑)	581.3 (9.6%↓)
	[11]	1.80	93.2	90.6	92.4 (2.5×↑)	454.6 (29.3%↓)
	Intel_Acc	1.61	92.8	90.1	52.2 (4.5×↑)	426.8(33.6%↓)
	Intel_Fast	**1.47**	**92.5**	**88.8**	**23.1 (10.2×↑)**	**439.2 (31.6%↓)**
	Intel_Small	1.37	92.2	89.6	41.6 (5.6×↑)	398.8 (38.0%↓)
TX2	DGCNN	1.81	92.9	88.9	270.4	174.9
	[5]	2.33	92.6	89.6	109.9 (2.5×↑)	121.2 (30.7%↓)
	[11]	1.81	93.2	90.6	206.4 (1.3×↑)	34.5 (80.3%↓)
	TX2_Acc	1.60	92.9	89.7	50.5 (5.3×↑)	60.5 (65.4%↓)
	TX2_Fast	**1.48**	**92.2**	**88.7**	**36.3 (7.5×↑)**	**57.6 (67.1%↓)**
	TX2_Small	1.34	92.5	89.0	88.4 (3.1×↑)	32.6 (81.4%↓)
Pi	DGCNN	1.81	92.9	88.9	4139.1	457.8
	[5]	2.33	92.6	89.6	3466.0 (1.2×↑)	354.3 (22.6%↓)
	[11]	1.81	93.2	90.6	1961.7 (2.1×↑)	271.1 (40.8%↓)
	Pi_Acc	1.47	92.8	89.3	1165.3 (3.6×↑)	270.2 (41.0%↓)
	Pi_Fast	**1.36**	**92.1**	**88.3**	**557.6 (7.4×↑)**	**257.8 (43.7%↓)**
	Pi_Small	1.40	92.1	88.7	683.4 (6.1×↑)	257.8 (43.7%↓)

HGNAS realizes a more pronounced reduction in peak memory usage on GPU platforms, like RTX3080 and TX2, primarily due to enhanced evaluation stability gained from the combination of predictor outputs and estimation results. Compared to the latency-aware design of Device_Fast, Device_Small reduces peak memory usage by up to 43%, highlighting the effectiveness of the proposed peak memory awareness approach. Meanwhile, as shown in Table III, GNNs designed by HGNAS aiming for lower latency also exhibit reduced peak memory usage, and vice versa. This exposes some correlation between the two metrics, which we will analyze in detail in Sec. IV-E. Moreover, despite the numerous candidates introduced by the fine-grained design space, HGNAS achieves search efficiency comparable to other hardware-aware GNN search frameworks by leveraging an efficient multi-stage search strategy. Specifically, G-CoS [24], with 10^9 candidate architectures, requires 4 GPU hours to complete a search. In contrast, HGNAS, with 10^{12} candidate architectures, requires only 3 GPU hours.

C. Evaluation on MR

To demonstrate the scalability of HGNAS, we conducted additional experiments using the text analysis dataset MR. This dataset is also used for graph classification tasks, featuring unique data characteristics, varying numbers of nodes, and different hardware sensitivities compared to the point cloud dataset ModelNet40. Fig. 11 presents the experimental results, with some data scaled to improve readability due to their large variance. Compared to DGCNN and PAS, our model achieves similar or superior task accuracy, along with improvements in inference efficiency by factors of $4.2\times$, $12.6\times$, $6.9\times$, and $42\times$ on four edge platforms, respectively. Additionally, we achieved a 98% reduction in peak memory usage on both the RTX3080 and Jetson TX2 platforms, which are equipped with GPU resources. Moreover, given the resource limitations of the Raspberry Pi, we relaxed hardware constraints during the search process. This resulted in a substantial accuracy improvement, demonstrating HGNAS's ability to balance accuracy and efficiency. The substantial performance improvement demonstrates HGNAS's advantages in designing efficient GNN models.

As a pioneering solution for deploying GNNs on edge devices, HGNAS excels in various tasks and easily scales to optimize diverse hardware performance metrics. In addressing the crucial energy consumption metrics for edge deployments, HGNAS can scale effectively via incremental predictor training. By integrating energy metrics into the objective function, HGNAS can automatically design energy-efficient GNN models, tailored to various application scenarios. We conducted an energy-efficient GNN architecture search experiment using the Jetson TX2 platform, and the results are shown in Fig. 12. Specifically, energy consumption was estimated by measuring the average power during model inference with the Jtop tool and multiplying it by the inference time. Compared to DGCNN, which consumes 79.9 mJ per inference, the GNN designed by HGNAS offers substantial energy savings of 86.3%, requiring only 10.93 mJ per inference, while maintaining similar accuracy. Compared to PAS, which consumes 90.2 mJ, HGNAS achieves an 87.9% reduction in energy consumption along with a $6.9\times$ speedup. In practice, we have observed that models with lower latency typically exhibit lower energy consumption, as also mentioned in [18]. By analyzing 1000 GNN candidates, we established a strong correlation between energy consumption and latency, evidenced by a correlation coefficient of 0.76. Based on the principle that $energy = power \times latency$ and given minimal power fluctuations for the same inference task, energy consumption is highly dependent on latency. Consequently, our latency-centered search process results in optimal GNNs with correspondingly lower energy consumption.

D. Trade-Off via scaling factors

In this set of experiments, we perform multiple explorations with various scaling factors on ModelNet40 using an Nvidia RTX3080 to demonstrate the effectiveness of the proposed hardware-aware GNN NAS method. The results are presented in Fig. 13, where we use the ratio of α and β to represent the different search orientations. Specifically, when α/β is smaller, the search results are more in favor of lower latency than higher accuracy. Conversely, when α/β is larger, the search results tend to emphasize more on accuracy. This indicates that the scaling factors α and β can adeptly balance accuracy and

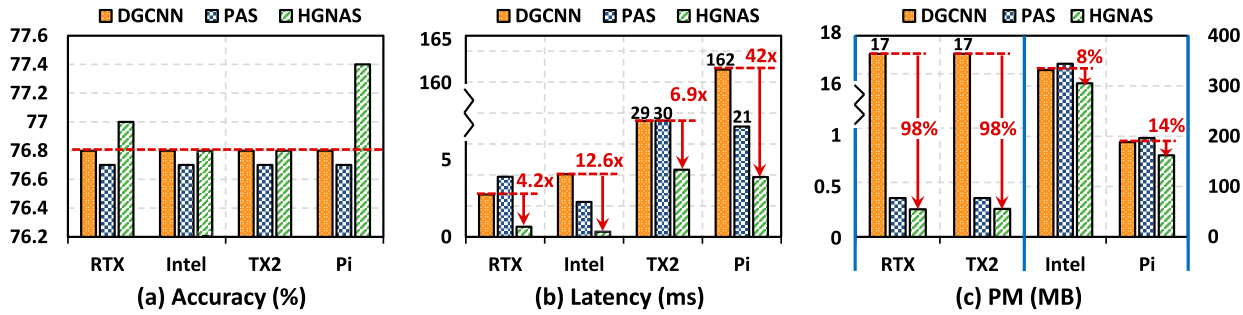

Fig. 11. Comparison of HGNAS and other methods on the MR dataset.

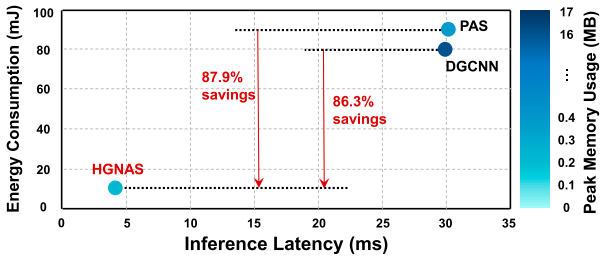

Fig. 12. Evaluation of HGNAS and baselines for inference energy consumption on Jetson TX2.

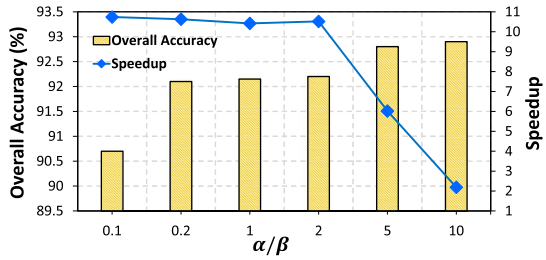

Fig. 13. The trade-off between accuracy and efficiency by scaling factor α and β. Efficiency is represented by the speedup when comparing to DGCNN.

latency, demonstrating the efficacy of the suggested hardware-aware architecture search. In practical applications, this approach offers significant flexibility, as we can dynamically steer the search process towards either accuracy or efficiency by adjusting α and β according to the demands of specific tasks.

E. Correlation Between the Measured Latency and Peak Memory Usage

As mentioned before, in HGNAS, we aim to search for resource-efficient GNNs with minimal inference latency and peak memory usage on specific target platforms. In practice, we observe a strong correlation between these two metrics, in which architectures with lower latency also have smaller peak memory usage. To illustrate this point, we randomly sample 1000 GNN architectures in the fine-grained GNN design space and subsequently measure their latency and peak memory usage on four edge devices. As depicted in Fig. 14, the correlation coefficient between the measured latency and peak memory

Fig. 14. Illustration of the correlation relationships between the latency and the peak memory usage.

usage reaches 0.83, suggesting a strong correlation. This observation suggests that GNNs exhibiting low latency also tend to be memory-efficient, especially on edge devices (i.e. Raspberry Pi) with limited computational and memory resources. As such, we initialise the peak memory predictor using the weights from the latency predictor, leading to a substantial reduction in training time. Furthermore, a consistent correlation between these metrics is observed across various devices. This suggests that GNN models optimized by HGNAS for one device are unlikely to exhibit significantly sub-optimal performance on others. This observation offers a valuable trade-off: designers can aim for sub-optimal performance based on predictions from similar devices without incurring the overheads of seeking extreme optimization for each new platform.

F. Prediction Results

In this set of experiments, we evaluate the effectiveness of the proposed GNN hardware performance prediction approach. As illustrated in Fig. 15, our predictor exhibits high accuracy in predicting hardware efficiency for GNNs across various devices on ModelNet40 dataset. Specifically, the MAPE for latency predictions is approximately 6% on the RTX3080, Intel i7-8700K, and Jetson TX2. However, it rises to around 19% on the

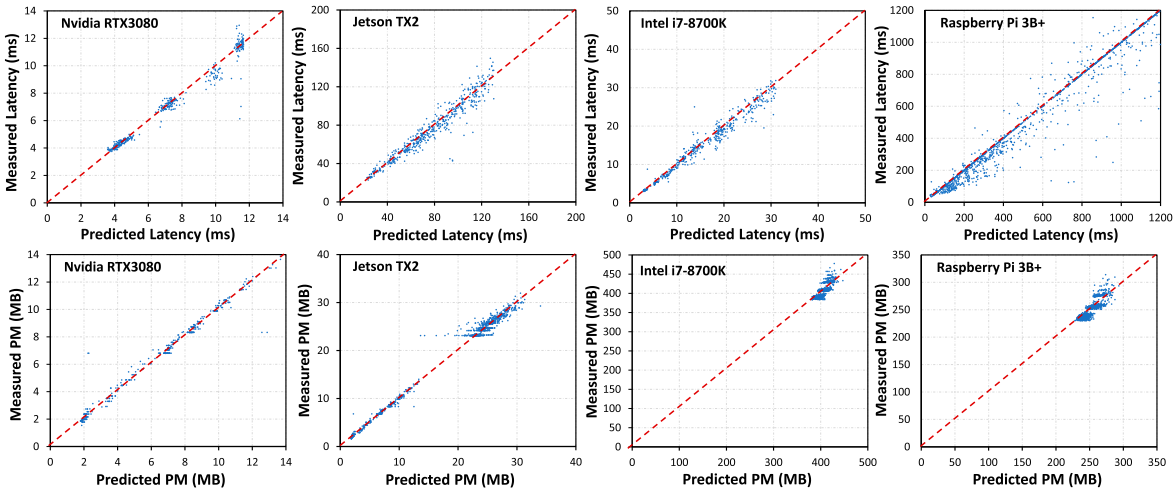

Fig. 15. Predictor's accuracy evaluation on various edge devices. The measured latency and PM denote the actual results collected from real devices. The distance between each blue point and the red dash line reflects the predictor's accuracy.

TABLE IV
PERFORMANCE OF THE PROPOSED PREDICTOR

Error	Latency Accuracy [%]				PM Accuracy [%]			
Bound	RTX	Intel	TX2	Pi	RTX	Intel	TX2	Pi
±1%	10.3	17.2	11.1	61.4	28.5	37.8	23.7	32.9
±5%	49.6	66.4	48.9	69.2	89.1	96.0	80.2	90.1
±10%	79.3	87.2	79.5	77.6	95.8	99.9	93.8	99.6

Raspberry Pi due to latency measurement fluctuations. Meanwhile, the MAPE for peak memory predictions is approximately 4% on the RTX3080 and Jetson TX2, compared to about 2% on the remaining devices. Table IV details the prediction results on ModelNet40, showing the percentage of predictions within the stipulated error bounds compared to on-device measurements. Across diverse devices, the latency prediction method achieves an approximate 80% accuracy with a 10% error bound, while the peak memory usage prediction method consistently surpasses 90% accuracy. Meanwhile, the peak memory estimation approach achieves a comparable accuracy of more than 90% with a 10% error bound on GPU devices. Furthermore, the integration of predictor outputs and estimation results fortifies the evaluation robustness, as evidenced by the substantial decrease in peak memory usage for the RTX_Small and TX2_Small. In practice, the GNN predictor performs better for GNNs that have faster inference speeds, thereby aiding HGNAS in searching more efficient GNN designs. For hardware performance prediction on the MR dataset, HGNAS achieves higher accuracy. The predictor attains an average accuracy of 88.25% for latency prediction with a 5% error bound across four devices. Additionally, the prediction accuracy for peak memory usage is approximately 99%.

G. Ablation Studies

In this set of experiments, we evaluate both the proposed GNN hardware performance prediction method and the multi-stage search strategy for their efficacy in optimizing

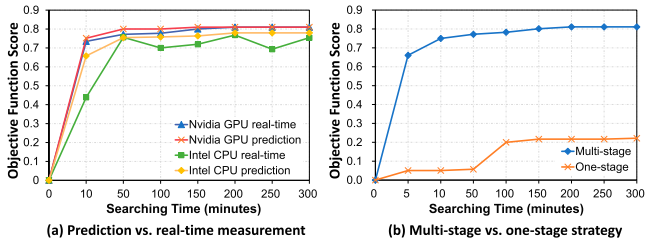

Fig. 16. (a) Performance comparison between real-time measured and prediction-based search. (b) Search time reduction with the multi-stage strategy.

the exploration of GNN architectures within a fine-grained design space.

Prediction vs. real-time measurement. Excessive deployment and communication overheads make real-time measurements of candidate GNNs impractical on most edge devices, whereas our prediction approach assesses them in milliseconds without compromising results. Fig. 16(a) illustrates that our GNN prediction approach effectively enhances search efficiency, as the models searched using both methods yield comparable performance. In particular, our prediction method becomes indispensable when real-time measurements are unfeasible, such as on Raspberry Pi.

Multi-stage vs. one-stage search strategy. In practice, the expansive fine-grained design space presents formidable challenges for efficient exploration; traditional single-stage search strategies often become mired in a myriad of candidate solutions. On the other hand, Fig. 16(b) shows that our multi-stage hierarchical search strategy significantly speeds up the exploration, capable of identifying an optimal GNN architecture within a matter of GPU hours. Specifically, by leveraging the multi-stage search strategy, HGNAS completes a single exploration in approximately 3 hours.

H. Insight From GNNs Designed by HGNAS

As demonstrated before, the same GNN model may perform differently across diverse edge computing platforms, which

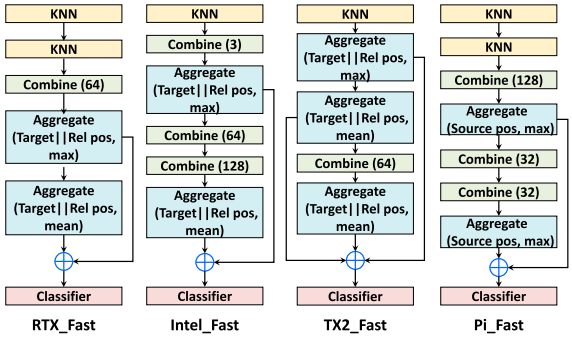

Fig. 17. Visualization of GNN models designed by HGNAS.

implies that the optimization needed is also device-specific. By leveraging the proposed GNN hardware performance predictor, HGNAS effectively identifies device-specific characteristics to successfully determine the optimal GNN architectures. Fig. 17 visualizes the GNN architectures designed by HGNAS for ModelNet40 dataset. Note that adjacent KNN operations are merged during execution to avoid invalid graph construction. The results clearly show that the hardware-efficient architectures designed by HGNAS closely align with the characteristics of the target device, corroborating the GNN computational patterns highlighted in **Observation 3**. For example, given that KNN constitutes a significant portion of execution time on RTX3080 and Jetson TX2, GNN models tailored for these platforms incorporate fewer valid KNN operations. Moreover, the optimal GNNs for Intel CPU involves fewer aggregate operations, while those tailored for the Raspberry Pi prioritize simplifying each operations.

V. Related Works

With the success of GNNs in various edge applications [3], [4], [16], there is growing attention in the research community on enhancing inference efficiency in environments with limited resources [6], [11], [25]. [11] performed a thorough analysis of the computational process of GNNs, finding that the initial layers contribute most significantly to task accuracy, with diminishing returns observed in later layers. They proposed enhancing inference efficiency by integrating a stronger feature extractor in the early stages and simplifying the later stages. Additionally, [5] noted that sampling constitutes the major overhead in GNNs, with frequent duplication of results in later layers from the initial layer. They suggested that reusing sampling results from the first layer could significantly accelerate inference. Nevertheless, manually designing such efficient GNNs requires substantial trial-and-errors, inevitably resulting in significant computational overhead as it necessitates training GNNs from scratch and conducting real-time device measurements to assess metrics such as accuracy and efficiency.

To automate model design and optimization, neural architecture search (NAS) was introduced as an AutoML technique, first applied to GNNs design by GraphNAS [14]. However, their focus on optimizing only accuracy fails to address the needs of real-time applications. Recent efforts have addressed this by introducing hardware-aware NAS methods that aim to optimize both accuracy and efficiency. For instance, G-CoS [24] leverage a hardware estimation approach within a GNN-Accelerator co-search framework to balance hardware efficiency and model accuracy for customized hardware platforms. Additionally, MaGNAS [25] utilizes a lookup table (LUT) method to simultaneously search for optimal GNN architectures and mapping schemes for the MPSoC platform. However, these hardware-aware NAS solutions are typically tailored for specific hardware designs and may not be easily adaptable to general-purpose computing platforms. Conversely, HGNAS uses a predictor approach that can be extended to various platforms, including hardware accelerators such as FlowGNN [38] and GCNAX [39]. This extension is made possible by incremental data collection and training for the predictor.

On the other hand, some studies have explored optimization techniques to handle large-scale graph training problems on distributed edge platforms. For example, SUGAR [40] leverages graph partitioning and subgraph-level training to enhance the training efficiency of GNNs on large-scale graphs, achieving notable results. This method primarily targets the optimization of training processes rather than the design of hardware-efficient architectures. Unlike SUGAR, this paper focuses on exploring methods to automatically design efficient GNN models for real-time inference, specifically for edge devices. More specifically, we aim to design GNNs that are not only accurate but also tailored to the limited computational resources typically available on edge devices.

VI. Conclusion

In this work, we propose HGNAS, the first hardware-aware framework to automatically explore efficient GNNs for resource-constrained edge devices. Our objective is to efficiently search the GNN architecture with the optimal hardware efficiency on target platforms, while satisfying the task accuracy. Specifically, HGNAS adopts a fine-grained design space to facilitate the exploration of high-performance architectures. Additionally, HGNAS integrates a novel GNN hardware performance prediction method to perceive the hardware efficiency of candidate architectures. To further streamline the exploration process, HGNAS incorporates the multi-stage hierarchical search strategy, which reduces a single exploration to a few GPU hours. Extensive experiments show that architectures generated by HGNAS consistently outperform SOTA GNNs, achieving about $10.6\times$ speedup and 82.5% peak memory reduction across edge devices. We believe that HGNAS has made pivotal progress in bringing GNNs to real-life edge applications.

References

[1] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2016, *arXiv:1609.02907*.

[2] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Proc. Neural Inf. Process. Syst. (NIPS)*, vol. 31, 2018, pp. 5171–5181.

[3] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2018, pp. 974–983.

[4] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "PointNet++: Deep hierarchical feature learning on point sets in a metric space," in *Proc. neural Inf. Process. Syst. (NIPS)*, 2017, pp. 5099–5108.
[5] Y. Li et al., "Towards efficient graph convolutional networks for point cloud handling," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, 2021, pp. 3752–3762.
[6] J. Shao, H. Zhang, Y. Mao, and J. Zhang, "Branchy-GNN: A device-edge co-inference framework for efficient point cloud processing," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, 2021, pp. 8488–8492.
[7] A. Zhou et al., "Graph neural networks automated design and deployment on device-edge co-inference systems," in *Proc. ACM/IEEE Des. Automat. Conf. (DAC)*, 2024, *arXiv:2404.05605*.
[8] T. Qiao et al., "GNNavigator: Towards adaptive training of graph neural networks via automatic guideline exploration," in *Proc. ACM/IEEE Des. Automat. Conf. (DAC)*, 2024, *arXiv:2404.09544*.
[9] Y. Wang et al., "Dynamic graph CNN for learning on point clouds," *ACM Trans. Graphics (TOG)*, vol. 38, no. 5, pp. 1–12, 2019.
[10] A. Zhou et al., "Hardware-aware graph neural network automated design for edge computing platforms," in *Proc. ACM/IEEE Des. Automat. Conf. (DAC)*, 2023, pp. 1–6.
[11] S. A. Tailor, R. De Jong, T. Azevedo, M. Mattina, and P. Maji, "Towards efficient point cloud graph neural networks through architectural simplification," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, 2021, pp. 2095–2104.
[12] L. Dudziak, T. Chau, M. Abdelfattah, R. Lee, H. Kim, and N. Lane, "BRP-NAS: Prediction-based NAS using GCNs," in *Proc. Neural Inf. Process. Syst. (NIPS)*, vol. 33, 2020, pp. 10480–10490.
[13] W. Jiang et al., "Accuracy vs. efficiency: Achieving both through FPGA-implementation aware neural architecture search," in *Proc. ACM/IEEE Des. Automat. Conf. (DAC)*, 2019, pp. 1–6.
[14] Y. Gao et al., "Graph neural architecture search," in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI)*, 2020, pp. 1403–1409.
[15] Y. Gao et al., "GraphNAS++: Distributed architecture search for graph neural networks," *IEEE Trans. Knowl. Data Eng. (TKDE)*, vol. 35, no. 7, pp. 6973–6987, Jul. 2023.
[16] L. Wei, H. Zhao, Z. He, and Q. Yao, "Neural architecture search for GNN-based graph classification," *ACM Trans. Inf. Syst. (TOIS)*, 2023.
[17] M. Yan et al., "Characterizing and understanding GCNs on GPU," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 22–25, Jan.–Jun. 2020.
[18] X. Luo, D. Liu, H. Kong, S. Huai, H. Chen, and W. Liu, "SurgeNAS: A comprehensive surgery on hardware-aware differentiable neural architecture search," *IEEE Trans. Comput.*, vol. 72, no. 4, pp. 1081–1094, Apr. 2023.
[19] Z. Wu et al., "3D ShapeNets: A deep representation for volumetric shapes," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2015, pp. 1912–1920.
[20] S. Caiet al., "Rethinking graph neural architecture search from message-passing," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2021, pp. 6657–6666.
[21] W. Zhang et al., "PaSca: A graph neural architecture search system under the scalable paradigm," in *Proc. ACM Web Conf. (WWW)*, 2022, pp. 1817–1828.
[22] B. Zoph and Q. Le, "Neural architecture search with reinforcement learning," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2016, *arXiv:1611.01578*.
[23] Y. Sheng et al., "The larger the fairer? Small neural networks can achieve fairness for edge devices," in *Proc. ACM/IEEE Des. Automat. Conf. (DAC)*, 2022, pp. 163–168.
[24] Y. Zhang, H. You, Y. Fu, T. Geng, A. Li, and Y. Lin., "G-CoS: GNN-accelerator co-search towards both better accuracy and efficiency," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, 2021, pp. 1–9.
[25] M. Odema, H. Bouzidi, H. Ouarnoughi, S. Niar, and M. A. Al Faruque, "MaGNAS: A mapping-aware graph neural architecture search framework for heterogeneous MPSoC deployment," in *ACM Trans. Embedded Comp. Syst.*, New York, NY, USA: ACM, vol. 22, no. 5s, 2023, pp. 1–26.
[26] H. Zhou, B. Zhang, R. Kannan, V. Prasanna, and C. Busart, "Model-architecture co-design for high performance temporal GNN inference on FPGA," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2022, pp. 1108–1117.
[27] R. Liu, J. Yang, Y. Chen, and W. Zhao, "eSLAM: An energy-efficient accelerator for real-time ORB-SLAM on FPGA platform," in *Proc. ACM/IEEE Des. Automat. Conf. (DAC)*, 2019, pp. 1–6.
[28] M. Li et al., "Eventor: An efficient event-based monocular multi-view stereo accelerator on FPGA platform," in *Proc. ACM/IEEE Des. Automat. Conf. (DAC)*, 2022, pp. 331–336.
[29] J. Yang, W. Fu, X. Cheng, X. Ye, P. Dai, and W. Zhao, "S2 engine: A novel systolic architecture for sparse convolutional neural networks," *IEEE Trans. Comput.*, vol. 71, no. 6, pp. 1440–1452, Jun. 2022.
[30] H. Benmeziane, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang, "A comprehensive survey on hardware-aware neural architecture search," 2021, *arXiv:2101.09336*.
[31] Z. Guo et al., "Single path one-shot neural architecture search with uniform sampling," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2020, pp. 544–560.
[32] "PyTorch Memory Analyzer," PyTorch. Accessed: Jul. 2023. Accessed: Jul. 25, 2023. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.cuda.max_memory_allocated.html
[33] "Linux resource tool," GitHub. Accessed: Jul. 2023. Accessed: Jul. 25, 2023. [Online]. Available: https://github.com/RussellLuo/resource
[34] Y. Zhang et al., "Every document owns its structure: Inductive text classification via graph neural networks," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2020, pp. 334–339.
[35] C. Jiang, K. Huang, J. Wu, X. Wang, J. Xiao, and A. Hussain, "PointGS: Bridging and fusing geometric and semantic space for 3D point cloud analysis," *Inf. Fusion*, vol. 91, no. C, pp. 316–326, Mar. 2023.
[36] J. Wu, M. Sun, C. Jiang, J. Liu, J. Smith, and Q. Zhang, "Context-based local-global fusion network for 3D point cloud classification and segmentation," *Expert Syst. Appl.*, vol. 251, 2024, Art. no. 124023.
[37] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," in *Proc. Workshop Int. Conf. Learn. Representations (ICLR)*, 2019, *arXiv:1903.02428*.
[38] R. Sarkar, Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1099–1112.
[39] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 775–788.
[40] Z. Xue, Y. Yang et al., "SUGAR: Efficient subgraph-level training via resource-aware graph partitioning," *IEEE Trans. Comput.*, vol. 72, no. 11, pp. 3167–3177, Nov. 2023.

Ao Zhou received the B.S. and M.S. degrees in software engineering from Beijing University of Technology, Beijing, China, in 2018 and 2021, respectively. He is currently working toward the Ph.D. degree with the School of Computer Science and Engineering, Beihang University, China. His research interests include GNN acceleration, computer architecture, FPGA accelerator, and heterogeneous computing. He is one of the contributors to the popular graph neural network framework PyTorch Geometric (PyG).

Jianlei Yang (Senior Member, IEEE) received the B.S. degree in microelectronics from Xidian University, Xi'an, China, in 2009, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, China, in 2014. He is currently an Associate Professor with the School of Computer Science and Engineering, Beihang University, Beijing. From 2014 to 2016, he was a Postdoctoral Researcher with the Department of ECE, University of Pittsburgh, PA, USA. His research interests include deep learning accelerators and neuromorphic computing systems. He was the recipient of the first/second place on ACM TAU Power Grid Simulation Contest in 2011/2012. He was a recipient of the IEEE ICCD Best Paper Award in 2013, the ACM GLSVLSI Best Paper Nomination in 2015, the IEEE ICESS Best Paper Award in 2017, and the ACM SIGKDD Best Student Paper Award in 2020.

Yingjie Qi received the B.S. degree in computer science and technology in 2020 from Beihang University, Beijing, China, where he is currently working toward the Ph.D. degree with the School of Computer Science and Engineering. His research interests include graph neural networks acceleration, processing-in-memory architectures, and deep learning compilers.

Tong Qiao received the B.S. degree in computer science and technology in 2020 from Beihang University, Beijing, China, where he is currently working toward the Ph.D. degree with the School of Computer Science and Engineering. His research interests include graph neural networks acceleration and system for machine learning.

Yumeng Shi received the B.S. degree in computer science and technology in 2021 from Beihang University, Beijing, China, where she is currently working toward the M.S. degree with the School of Computer Science and Engineering. Her research interests include deep learning model compression and hardware/software co-design.

Cenlin Duan received the B.S. degree in electronic science and technology from the University of Electronic Science and Technology of China, Chengdu, China, in 2015, and the M.S. degree in software engineering from Xidian University, Xi'an, China, in 2018. She is currently working toward the Ph.D. degree with the School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China. Her research interests include processing-in-memory architectures and deep learning accelerators.

Weisheng Zhao (Fellow, IEEE) received the Ph.D. degree in physics from the University of Paris Sud, Paris, France, in 2007. Currently, he is a Professor with the School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China. In 2009, he joined the French National Research Center, Paris, as a Tenured Research Scientist. Since 2014, he has been a Distinguished Professor with Beihang University. He has published more than 200 scientific articles in leading journals and conferences, such as *Nature Electronics*, *Nature Communications*, *Advanced Materials*, IEEE TRANSACTIONS, ISCA, and DAC. His research interests include the hybrid integration of nanodevices with CMOS circuit and new nonvolatile memory (40-nm technology node and below) like MRAM circuit and architecture design. He was the Editor-in-Chief for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEM I: REGULAR PAPER from 2020 to 2023.

Chunming Hu received the Ph.D. degree in computer science and technology from Beihang University, Beijing, China, in 2006. Currently, he is a Professor and the Dean of the School of Software, Beihang University, Beijing, China. His research interests include distributed systems, system virtualization, mobile computing, and cloud computing. He is currently one of the W3C Board of Directors, and serving as the Deputy Director of W3C China.