Graph Neural Networks Automated Design and Deployment on Device-Edge Co-Inference Systems[*]

Ao Zhou[1], Jianlei Yang[1], Tong Qiao[1], Yingjie Qi[1], Zhi Yang[2], Weisheng Zhao[1], Chunming Hu[1]
[1]Beihang University, Beijing, China [2]Peking University, Beijing, China

Abstract

The key to device-edge co-inference paradigm is to partition models into computation-friendly and computation-intensive parts across the device and the edge, respectively. However, for Graph Neural Networks (GNNs), we find that simply partitioning without altering their structures can hardly achieve the full potential of the co-inference paradigm due to various computational-communication overheads of GNN operations over heterogeneous devices. We present GCoDE, the first automatic framework for GNN that innovatively Co-designs the architecture search and the mapping of each operation on Device-Edge hierarchies. GCoDE abstracts the device communication process into an explicit operation and fuses the search of architecture and the operations mapping in a unified space for joint-optimization. Also, the performance-awareness approach, utilized in the constraint-based search process of GCoDE, enables effective evaluation of architecture efficiency in diverse heterogeneous systems. We implement the co-inference engine and runtime dispatcher in GCoDE to enhance the deployment efficiency. Experimental results show that GCoDE can achieve up to 44.9× speedup and 98.2% energy reduction compared to existing approaches across various applications and system configurations.

Keywords

GNN, Hardware-Aware, Neural Architecture Search, Co-Inference

1 Introduction

Graph Neural Networks (GNNs) have emerged as the state-of-the-art (SOTA) method for graph-based learning tasks in edge scenarios such as point cloud processing [1] and natural language processing [2]. Additionally, the rising popularity of various sensors in mobile devices also encourages the deployment of GNNs to wireless network edge for tasks like sensing and interaction (e.g., collision prediction in self-driving vehicles [3], speech analytic [4]). However, GNNs often suffer from prohibitive inference costs due to their hungry demands for computational and memory resources [5]. For example, even the optimal GNNs searched by HGNAS [6] for edge devices only achieve about **2** fps processing point cloud data on Raspberry Pi 3B+, insufficient for real-time needs. Deploying such expensive GNNs on resource-constrained edge devices results in excessive workloads, low efficiency, and severe energy consumption, thus limiting their potential.

With the wide application of AI in real life, device-edge co-inference has emerged as a promising paradigm for the large-scale deployment of DNN models at the wireless network edge [7]. By extracting and transmitting intermediate data from the device (e.g., smartphones) to the edge (e.g., edge nodes) for collaborative computation, this approach significantly reduces resource consumption on the device and improves efficiency. Maximizing the benefits of co-inference depends on finding an optimal partitioning point that balances communication and computation trade-offs. In contrast to DNNs, GNNs involve both computation-intensive matrix operations and memory-intensive graph processing [5]. Thus, simply selecting partitioning points on an existing architecture does not fully exploit the collaborative potential. For example, BRANCHY-GNN [8] examines the optimal partition point for GNNs, while reducing communication overhead, does not markedly improve inference efficiency. Therefore, an elegant approach

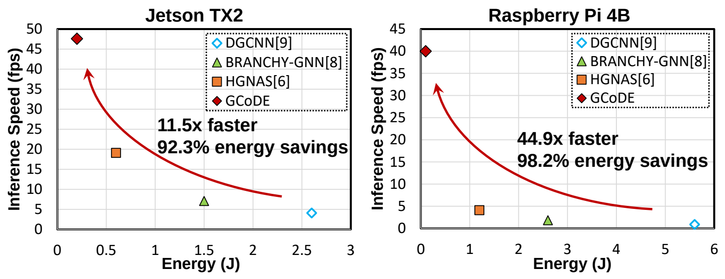


Figure 1: Inference speed vs. energy consumption comparison.

for joint-optimization of architecture design and mapping schemes is warranted.

In this paper, we propose a novel NAS-based GNN architecture-mapping co-design and deployment framework for device-edge hierarchies, namely GCoDE. Given user requirements, GCoDE can efficiently search and deploy optimal GNN architectures with their concomitant mapping schemes for target systems, achieving both accuracy and efficiency under latency and energy constraints. In practice, the design and deployment of custom GNNs for device-edge hierarchies face several challenges. First, designing GNNs for device-edge hierarchies requires balancing the trade-offs between communication and computation. Additionally, the heterogeneity between device and edge leads to diverse GNN execution characteristics, necessitating an effective system performance awareness approach. Moreover, separating architecture design from mapping often results in sub-optimal performance, requiring a co-optimized approach for both. Efficient task dispatching and execution engines are also crucial for maximizing the potential of device-edge deployments. Besides, NAS itself is known for lengthy search times.

To address the aforementioned challenges, our proposed GCoDE framework incorporates device-edge operation mapping into the GNN architecture design for co-optimization. Building on a novel concept, we conceptualize the communication process between the device and edge as a specialized GNN operation. Such a concept is put into good use by **introducing *communicate* as an explicit operation within architecture design space.** The location of each *communicate* operation in the architecture signifies a model split and the subsequent operations mapping between the device and edge. Consequently, the mapping scheme is inherently included in the architectures sampled by GCoDE, and the exploration of the GNN architecture space will also optimize the mapping scheme. Moreover, with the fused architecture space, system performance evaluation becomes a simpler process of architecture evaluation, thus reducing complexity. GCoDE employs two system performance evaluation methods to guide the exploration towards more efficient design configurations. The first approach, based on cost estimation, approximates latency relationships between architectures with low overhead. The second, a GIN-based performance predictor powered by a feature enhancement strategy, provides accurate system latency assessments, ideal for scenarios with strict latency constraints. By leveraging a constraint-based random search strategy, the search process requires only 3 GPU hours. Additionally, GCoDE integrates an efficient co-inference engine, enabling automated GNN deployment and dynamic runtime dispatch. Fig. 1 shows the significant improvements in efficiency and energy savings of GCoDE over existing methods, with Intel i7 and Nvidia 1060 as edge. The main contributions of this paper are as follows:

- **Framework.** To the best of our knowledge, GCoDE is the first framework for automated GNN design and deployment, targeting

[*]This work was supported in part by National Natural Science Foundation of China (Grant No. 62072019) and National Key Laboratory of Spintronics. Corresponding authors are *Jianlei Yang* and *Chunming Hu*, Email: jianlei@buaa.edu.cn, hucm@buaa.edu.cn

DAC '24, June 23–27, 2024, San Francisco, CA, USA
2024. ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3655938

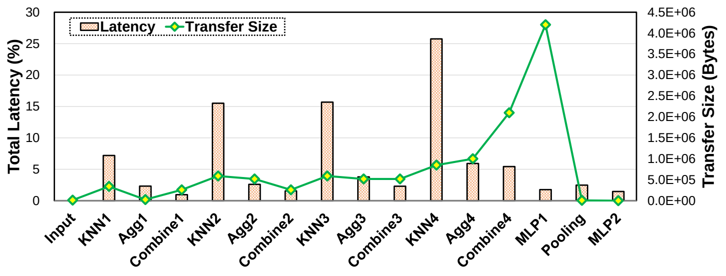


Figure 2: Changes in the required transfer data size and percentage of total latency for each operation in DGCNN.

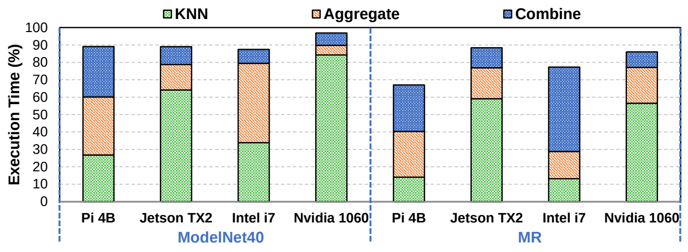


Figure 3: Execution time breakdown of DGCNN across various devices on ModelNet40 and MR datasets.

device-edge co-inference systems. Additionally, GCoDE achieves the co-optimization of architecture and its mapping scheme, balancing objectives like accuracy, latency, and energy.

- **System performance awareness.** To our best understanding, GCoDE is also the first work to achieve system performance awareness for GNNs in heterogeneous wireless edges. The proposed system performance predictor shows over 94.7% accuracy in relative latency relationship prediction among GNN architectures across diverse systems.
- **Evaluation.** Extensive experiments across various applications and deployment systems highlight the superiority of GCoDE, achieving up to 44.9× speedup and 98.2% energy savings without sacrificing accuracy.

2 Motivation and Related Works

This section outlines our three key motivations based on various observations and related research.

Motivation ❶: Exploring trade-offs between computation and communication on device-edge co-inference systems.

Fig. 2 illustrates the variation in computation latency for each operation in the DGCNN [9] architecture on ModelNet40 [10] dataset and Jetson TX2 platform, alongside the data transfer sizes required for partitioning after each operation. As the node feature dimensions expand with each GNN layer, the *KNN* operations become progressively more time-consuming, with the last one taking up over a quarter of the total execution time, aligning with findings in [6]. In addition, the *KNN* operation produces graph data, leading to an increase in transfer size if a subsequent *Aggregate* operation is required. On the other hand, the *Pooling* operation markedly reduces the intermediate data to only 68% of input size, while the higher-dimensional *MLP1* results in a significant increment. As pointed out in [8], a potential resource-efficient deployment solution is to split the DGCNN architecture at the front layers or *Pooling* operation to reduce the communication overhead. The effectiveness of this solution depends on the device and edge performance. Therefore, it is necessary to carefully balance the trade-offs between communication and computation overhead during the architecture design to achieve optimal efficiency [11].

Motivation ❷: Perceiving the heterogeneous hardware sensitivities of GNNs.

Next, we analyze the GNN hardware sensitivity by examining the execution time of DGCNN across various edge platforms. Fig. 3 demonstrates that on the ModelNet40 point cloud dataset [10], the *KNN* operation consumes most of the execution time on both Jetson TX2 and Nvidia 1060. This is due to the intensive and irregular memory accesses, which obscure the parallel

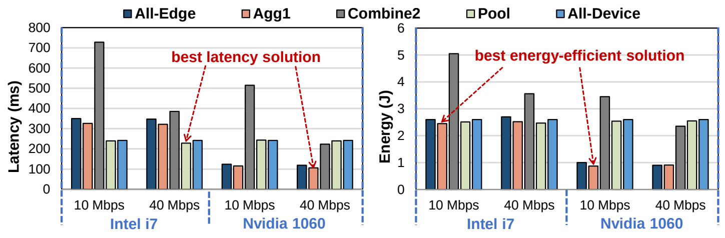


Figure 4: Performance of various partitioning schemes on DGCNN under different heterogeneities. Jetson TX2 serves as the device.

Table 1: Comparison of support features

Supported Features	GCoDE	HGNAS [6]	MaGNAS [12]	BRANCHY [8]
Design Automation	✓	✓	✓	✗
Architecture Exploration	✓	✓	✓	✗
Performance Awareness	✓	✓	✓	✗
▷ Single Device	✓	✓	✗	✗
▷ Heterogeneous	✓	✗	✓	✗
▷ Heterog. Wireless Edge	✓	✗	✗	✗
Multi-Objective Optimization	✓	✓	✓	✗
Device-Edge Deployment	✓	✗	✗	✓
Runtime Optimization	✓	✗	✗	✗

processing capabilities of GPUs. For the Intel i7, the *Aggregate* operation emerges as the main optimization challenge. For the Raspberry Pi, constraints by the lower computing power, all operations are time-consuming. Unlike point cloud data, the text dataset MR features fewer nodes (1024 vs. 17) and larger feature dimensions (3 vs. 300), resulting in distinct execution characteristics. For instance, the *Combine* operation dominates the execution time on the Intel i7.

The above observation highlights the hardware sensitivity of GNNs, emphasizing the need for performance consideration in architecture design. HGNAS [6] presents a GCN-based latency predictor, accurate for single-device evaluations but less effective in heterogeneous environments (see Sec. 4.4). MaGNAS [12] employs a lookup table (LUT) approach for heterogeneous MPSoCs, but this method fails to address runtime overheads [13]. As a result, there remains a deficiency in elegant, scalable, and accurate approaches for assessing GNN co-inference efficiency across device-edge hierarchies.

Motivation ❸: Addressing the gap in automated design and deployment of GNNs on device-edge hierarchies.

Despite significant advances in DNN co-inference research [11], studies focusing on GNNs remain scarce [14]. As shown in Tab. 1, BRANCHY-GNN [8] investigates the splitting and intermediate data compression techniques for GNN. However, the insufficient exploration of novel architectures and hardware-aware strategies leads to sub-optimal performance. While HGANS [6] and MaGNAS [12] introduce hardware-aware NAS for GNNs, they lack consideration of the system heterogeneity and wireless network conditions. A key issue is that detaching the architecture design and mapping, such as partitioning after design, the full potential of collaboration will not be realized. Fig. 4 shows that despite exploring potential partitioning schemes described in Motivation ❶, even the best partitioning points fail to significantly enhance performance under different wireless network conditions. As such, an automated architecture-mapping co-design and deployment framework tailored for GNNs on device-edge co-inference systems is needed, which motivates our investigation in this paper.

3 GCoDE Methodology

3.1 Framework Overview

Fig. 5 provides an overview of our GCoDE framework, comprising three key components: graph neural architecture and mapping co-exploration, system performance awareness, and device-edge deployment. Given user requirements such as system configurations, GCoDE begins by exploring the GNN co-inference design space to locate optimal architectures with tailored mapping schemes. GCoDE organizes the co-inference design space into a supernet, decoupling the training and searching processes via a one-shot approach. Subsequently, GCoDE adopts a constraint-based random search strategy to efficiently explore the design space. During the

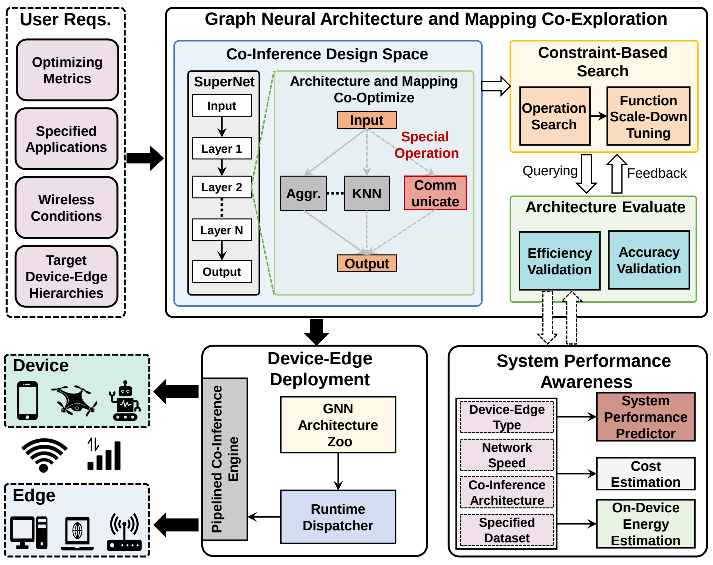


Figure 5: Overview of GCoDE framework.

exploration, each candidate is evaluated based on its accuracy on the validation dataset and its performance (latency and energy consumption) on the target system, ensuring the user requirements for deployment. System performance metrics are derived from our system performance predictor, cost estimation, and energy estimation method, obviating the need for laborious real-time measurements. Following architecture exploration, a set of optimal GNN architectures is prepared for deployment. GCoDE assembles these architectures into an architecture zoo, enabling dynamic adjustments to the deployed architecture via our runtime dispatcher. Deployment leverages our pipelined co-inference engine, which efficiently executes GNN inference tasks between device and edge. Subsequently, we will formulate the optimization process of GCoDE, and introduce these components in detail.

3.2 Problem Formulation

In this work, we aim to co-optimize the accuracy and efficiency of GNN architectures deployed on device-edge co-inference systems. Given the user requirements: the device $\mathcal{D}$, the edge $\mathcal{E}$, anticipated network speed $\mathcal{S}$, latency constraint C_{lat} and on-device energy constraint C_e, the co-optimization process can be formulated as:

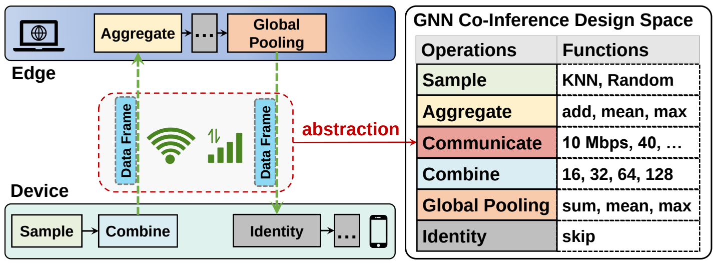
$$\arg\max_{\alpha\in\mathbb{A}} \left\{acc_{val}\left(\mathcal{W}^*,\alpha\right) - \lambda\times\left(\mathcal{P}_{sys}\left(\alpha,\mathcal{D},\mathcal{E},\mathcal{S}\right) + E_{dev}(\alpha,\mathcal{D},\mathcal{S})\right)\right\},$$

$$\begin{aligned} s.t. \quad & \mathcal{W}^* = \arg\max_{\mathcal{W}} \left\{acc_{train}\left(\mathcal{W},\alpha\right)\right\} \\ & \mathcal{P}_{sys} < C_{lat} \quad and \quad E_{dev} < C_e \end{aligned},$$

where $\mathcal{W}$ denotes the model weights, acc_{train} is the training accuracy, acc_{val} is the validation accuracy, α is the selected architecture to be optimized form GNN co-inference design space $\mathbb{A}$, $\mathcal{P}_{sys}$ indicates the inference latency on targeted system, E_{dev} denotes the energy consumption of α on the device $\mathcal{D}$, and λ is a scaling factor used to adjust the optimize propensity between accuracy and efficiency. Note that system performance is jointly determined by the architecture, device-edge configurations, and network conditions.

3.3 Co-Inference Design Space

Fig. 6 illustrates our approach to avoid the detachment of architecture design and mapping by establishing a unified co-inference design space for GNN. Specifically, the primary difference between co-inference and single-device inference lies in the data communication between device and edge, making the selection of an effective communication point vital. To achieve an optimal balance between communication and computation, we introduce a novel concept: communication between device and edge can be treated as a specialized operation within the GNN architecture. Thus, we abstract device communication as a distinct GNN operation and integrate it into the architecture design space. This fusion of architecture and mapping enables GCoDE to explore various flexible collaborative patterns, rather



Operations	Functions
Sample	KNN, Random
Aggregate	add, mean, max
Communicate	10 Mbps, 40, …
Combine	16, 32, 64, 128
Global Pooling	sum, mean, max
Identity	skip

Figure 6: Unified design space of architecture design and mapping.

Algorithm 1: Constraint-based search strategy.

```
Input: Target system $Sys$, constraints: $\{C_{lat}, C_e\}$, function setting $f$, max
       search and tuning iteration: $T$, $T_f$.
Output: The best found GNN architectures $\alpha^*$.
Initialize $\alpha^* \leftarrow \emptyset$, $Ops \leftarrow \emptyset$, $func \leftarrow \emptyset$.
Pre-train GNN supernet $\mathcal{N}_{super}$ with $f$.
/* Stage 1: operation search */
for $1 \le t \le T$ do
    while Check($Ops$) do
        $Ops \leftarrow$ Random ($\mathbb{A}$)                     // Sample valid operations
    end
    $\{\mathcal{P}_{sys}, E_{dev}\} \leftarrow$ Evaluate ($Sys, Ops, f$)    // Evaluate performance
    if $\mathcal{P}_{sys} < C_{lat}$ and $E_{dev} < C_e$ then
        $score \leftarrow (acc_{val} - \lambda(\mathcal{P}_{sys} + E_{dev}))$   // Full evaluation
    else
        $score \leftarrow (-1)$                          // Discard failed architectures
    end
    $\alpha^* \leftarrow$ Update($Ops, f, score$)              // Update operations
end
/* Stage 2: function scale-down tuning */
for $1 \le t \le T_f$ do
    $func \leftarrow$ Random($\mathbb{A}, f$)              // scale-down functions from $f$
    $\alpha^* \leftarrow$ Update($Ops, func, acc_{val}$)        // Update functions
end
return $\alpha^*$                                         // Top-performing designs
```

than simply searching for a partition point. Furthermore, this flexible mapping aids in adjusting to system heterogeneity and discovering optimal collaboration patterns during exploration.

Specifically, the GNN co-inference design space $\mathbb{A}$ is organized as a supernet, comprising six operations in each layer: *Sample*, *Aggregate*, *Communicate*, *Combine*, *Global Pooling*, and *Identity*, each having distinct functional properties as shown in Fig. 6. During supernet training, linear layers are used to align the dimensions of all operations within the same layer, which will be removed before search to ensure efficiency.

3.4 Efficient Constraint-Based Search Strategy

Incorporating mapping schemes into the GNN design space leads to numerous invalid candidates during exploration. Examples of this are consecutive *Communicate* operations or a *Global Pooling* operation followed by *Aggregate*. In such cases, intelligent algorithms, such as evolutionary algorithms (EA), face challenges in identifying valid architectures (see Sec. 4.5). Conversely, the simpler random search can yield surprising benefits in such complex design space [15]. Furthermore, random search is more customizable and supports maintaining multiple optimal solutions for various objectives within a single search, catering to the varied application requirements of co-inference (e.g., low energy and low latency). As a result, GCoDE adopts a constraint-based random search strategy to improve exploration efficiency, as shown in Alg. 1. Given user requirements, GCoDE sets an appropriate function configuration for the supernet, guided by the target architecture like DGCNN. Subsequently, GCoDE pre-trains the supernet with a focus on accuracy to establish shared weights for further architecture exploration. Details of the search process are outlined below.

Stage 1: Operation search. This stage focuses on identifying optimal operation sets that adhere to the system performance constraints and accuracy requirements. Specifically, GCoDE checks the validity of each randomly sampled operation set to prevent interference from invalid architectures during the search. Only architectures that pass this validity check

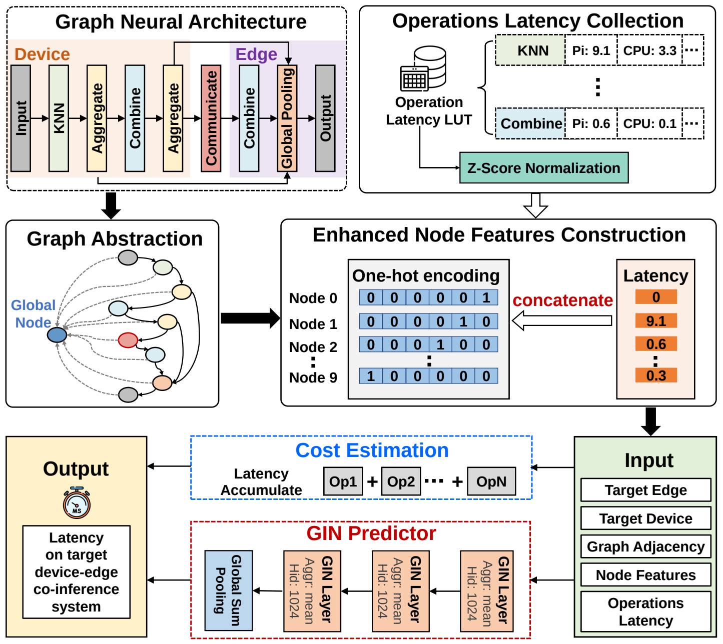


Figure 7: Latency perception for GNNs on device-edge hierarchies.

are subsequently evaluated for inference latency $\mathcal{P}_{sys}$ and energy consumption E_{dev}. Additionally, accuracy assessments are performed only on architectures that meet the specified constraints. To mitigate the influence of varying metrics magnitudes on optimization efficacy, $\mathcal{P}_{sys}$ and E_{dev} are normalized during architecture scoring. This combined approach of validity and performance constraint checks effectively minimizes unnecessary evaluation overheads.

Stage 2: Function scale-down tuning. This stage aims to find more efficient function settings while maintaining accuracy. Practically, architectures that pass the first stage already satisfy the performance requirements. Therefore, GCoDE concentrates on evaluating the impact of scaled-down function settings on accuracy in this stage. Specifically, GCoDE performs scale-down tuning of function setting f, such as reducing the dimensions of *Combine*, to explore more efficient architectures. After the search, GCoDE generates a set of optimal architectures, which are maintained by the GNN architecture zoo to wait for deployment.

3.5 System Performance Awareness

GCoDE integrates system performance awareness methods to guarantee efficiency post-deployment. Specifically, the latency perceive process includes graph abstraction, enhanced node features construction, and inference cost estimation or latency prediction, as depicted in Fig. 7.

Graph abstraction. Drawing inspiration from [6], GCoDE abstracts GNN architecture into a directed graph to facilitate architecture graph learning. In this graph, nodes represent various operations, while edges indicate data flow between operations. Additionally, GCoDE introduces self-connections and a global node to enhance graph connectivity.

Enhanced node features construction. To better capture system heterogeneity and network conditions, GCoDE develops enhanced node features for the architecture graph. Specifically, GCoDE maintains an operation latency LUT across various devices, with negligible construction overhead due to the limited number of valid operations. The *communicate* operation latency is calculable based on the transfer data size and the available network bandwidth. GCoDE then concatenates the one-hot encoding of each node with its corresponding latency from the LUT, enriching the initial node features. To mitigate the effect of varying operation magnitudes, latency values are normalized using *z-score normalization* before concatenation.

Cost estimation. With the mapping scheme integrated into the architecture, system performance evaluation becomes efficient and straightforward. Specifically, based on the maintained latency LUT, we can easily accumulate all operation latency in the architecture graph. While this estimation may not include potential runtime overheads compared to measured latency, it effectively captures the relative latency relationship between architectures, crucial for steering the exploration process towards more efficient designs. In application scenarios without strict latency constraints, this scheme is highly cost-effective.

Latency prediction. By abstracting the architecture into a graph, system performance evaluation is transformed into a graph learning problem, an area where GNNs excel. Moreover, GIN [16] outperforms other GNN layers in extracting comprehensive graph information. Consequently, we built a latency predictor using three GIN layers, each with a *mean* aggregation operator. Besides, *Global Sum Pooling* is used as the extractor for all node features. The combination of *mean* aggregation operator with *Global Sum Pooling* enables the predictor to efficiently extract latency information from the total graph. Due to the small number of nodes in the architecture graph, the runtime overhead of the predictor is minimal, measured in milliseconds on GPUs. The highly accurate system latency predictor ensures that the explored architecture meets the strict latency requirements of specific application scenarios (e.g., autonomous driving).

On-device energy estimation. To meet the energy constraints, we employ the energy estimation method described in [11], focusing on the energy consumption of the device. Specifically, the total energy consumption of the device for a single inference is estimated as:

$$E_{total} = E_{idle} + E_{run} + E_{comm} ,$$

where E_{comm} represents the communication energy consumption, computed using power models proposed in [17]. E_{idle} and E_{run} are computational energy consumption in idle and operation executed states, respectively, calculated by multiplying associated power consumption with idle and execution time.

3.6 Device-Edge Deployment

The deployment of GCoDE is based on the integrated architecture zoo, runtime dispatcher, and co-inference engine, as listed in Fig. 5.

GNN architecture zoo. To accommodate diverse runtime requirements, GCoDE maintains a set of optimal GNN co-inference architectures (low energy consumption, low latency, high accuracy, etc.) in an architecture zoo. With the proposed constraint-based search strategy, GCoDE generates all these GNNs in a single search without additional overheads.

Runtime dispatcher. Furthermore, GCoDE dynamically adapts execution architectures via its runtime dispatcher to meet the fluctuating latency and power consumption constraints of the device. This flexibility stems from the GNN architecture zoo, ensuring optimal architecture selection for varied environments.

Co-inference engine. We develop an efficient co-inference engine using Python socket [18], enabling pipelined task execution. Specifically, instead of waiting for the edge to return execution results after processing a data frame, the device immediately begins processing the next frame. This approach effectively mitigates communication latency, significantly boosting the overall processing efficiency of the co-inference application. Additionally, sending and receiving processes are implemented on separate threads. Each thread maintains its own message queue, allowing for better handling of diverse network conditions. To reduce communication overhead, GCoDE compresses all transmitted data based on zlib tool [19].

4 Experiments

4.1 Experimental Setup

Datasets and competitors settings. To evaluate GCoDE, we consider two different application datasets: the point cloud processing benchmark ModelNet40 [10] and the text analysis dataset MR [20], following the evaluation settings in [2, 6]. Our comparison included several baselines: (1) the manually designed DGCNN [9], (2) the manually optimized architecture [1], (3) the GNN device-edge co-inference method BRANCHY-GNN (denoted as BRANCHY) [8], and (4) two GNN NAS framework HGNAS [6] and PNAS [2]. Additionally, to compare architecture-mapping separation designs and joint optimization, we partition hardware-efficient GNNs from existing NAS frameworks and evaluated the efficiency at optimal partition points. For a fair comparison, we used the reported task accuracy in these papers and tested efficiency based on the PyTorch Geometric (PyG) framework

Table 2: **Performance comparison of GCoDE and existing approaches in different modes: Device-Only (D), Edge-Only (E), and Device-Edge Co-Inference (Co). OA and mAcc denote overall and balanced accuracy, respectively.**

$\mathcal{S}_L$	Method	OA (%)	mAcc (%)	Mode	Jetson TX2 ⇌ Nvidia 1060		Jetson TX2 ⇌ Intel i7		Pi 4B ⇌ Nvidia 1060		Pi 4B ⇌ Intel i7	
					Latency (ms)	Energy (J)	Latency (ms)	Energy (J)	Latency (ms)	Energy (J)	Latency (ms)	Energy (J)
≤ 40 Mbps	DGCNN [9]	92.9	88.9	D	241.9	2.6	241.9	2.6	1121.8	5.6	1121.8	5.6
				E	118.8 (2.0×↑)	0.9 (65.4%↓)	347.6 (1.4×↓)	2.6 (0.0%↓)	103.5 (10.8×↑)	0.4 (92.9%↓)	333.7 (3.4×↑)	1.0 (82.1%↓)
	[1]	92.6	90.6	D	107.6 (2.3×↑)	1.2 (53.8%↓)	107.6 (2.2×↑)	1.2 (53.8%↓)	851.1 (1.3×↑)	4.3 (23.2%↓)	851.1 (1.3×↑)	4.3 (23.2%↓)
				E	86.6 (2.8×↑)	0.7 (73.1%↓)	321.6 (1.3×↓)	2.5 (3.8%↓)	68.7 (16.3×↑)	0.2 (96.4%↓)	303.4 (3.7×↑)	1.0 (82.1%↓)
	HGNAS [6]	92.1~92.5	88.3~88.8	D	52.1 (4.6×↑)	0.6 (76.9%↓)	52.1 (4.6×↑)	0.6 (76.9%↓)	241.5 (4.6×↑)	1.2 (78.6%↓)	241.5 (4.6×↑)	1.2 (78.6%↓)
				E	79.2 (3.1×↑)	0.6 (76.9%↓)	83.7 (2.9×↑)	0.7 (73.1%↓)	69.0 (16.3×↑)	0.2 (96.4%↓)	71.0 (15.8×↑)	0.3 (94.6%↓)
	BRANCHY [8]	92.0	-	Co	141.2 (1.7×↑)	1.5 (42.3%↓)	140.2 (1.7×↑)	1.5 (42.3%↓)	541.8 (2.1×↑)	2.6 (53.6%↓)	528.1 (2.1×↑)	2.5 (55.4%↓)
	HGNAS [6]+Partition	92.1~92.2	88.3~88.7	Co	52.6 (4.6×↑)	0.5 (80.8%↓)	51.4 (4.7×↑)	0.5 (80.8%↓)	53.7 (20.9×↑)	1.9 (66.1%↓)	106.0 (10.6×↑)	2.1 (62.5%↓)
	GCoDE	**92.1~92.6**	**88.1~89.7**	**Co**	**31.9 (7.6×↑)**	**0.3 (88.5%↓)**	**21.0 (11.5×↑)**	**0.2 (92.3%↓)**	**25.0 (44.9×↑)**	**0.1 (98.2%↓)**	**64.4 (17.4×↑)**	**0.2 (96.4%↓)**
≤ 10 Mbps	DGCNN [9]	92.9	88.9	D	241.9	2.6	241.9	2.6	1121.8	5.6	1121.8	5.6
				E	123.9 (2.0×↑)	1.0 (61.5%↓)	350.1 (1.4×↓)	2.7 (3.8%↑)	107.8 (10.4×↑)	0.4 (92.9%↓)	339.5 (3.3×↑)	1.1 (80.4%↓)
	[1]	92.6	90.6	D	107.6 (2.2×↑)	1.2 (53.8%↓)	107.6 (2.2×↑)	1.2 (53.8%↓)	851.1 (1.3×↑)	4.3 (23.2%↓)	851.1 (1.3×↑)	4.3 (23.2%↓)
				E	93.4 (2.6×↑)	0.7 (73.1%↓)	325.5 (1.3×↓)	2.5 (3.8%↓)	75.8 (14.8×↑)	0.3 (94.6%↓)	307.7 (3.6×↑)	1.0 (82.1%↓)
	HGNAS [6]	92.1~92.5	88.3~88.8	D	52.1 (4.6×↑)	0.6 (76.9%↓)	52.1 (4.6×↑)	0.6 (76.9%↓)	241.5 (4.6×↑)	1.2 (78.6%↓)	241.5 (4.6×↑)	1.2 (78.6%↓)
				E	87.8 (2.8×↑)	0.7 (73.1%↓)	88.3 (2.7×↑)	0.7 (73.1%↓)	70.3 (16.0×↑)	0.3 (94.6%↓)	74.0 (15.2×↑)	0.3 (94.6%↓)
	BRANCHY [8]	92.0	-	Co	141.0 (1.7×↑)	1.5 (42.3%↓)	140.8 (1.7×↑)	1.5 (42.3%↓)	531.8 (2.1×↑)	2.6 (53.6%↓)	544.0 (2.1×↑)	2.6 (53.6%↓)
	HGNAS [6]+Partition	92.1~92.2	88.3~88.7	Co	57.1 (4.2×↑)	0.5 (80.8%↓)	53.8 (4.5×↑)	0.5 (80.8%↓)	72.9 (15.4×↑)	0.9 (83.9%↓)	122.8 (9.1×↑)	1.0 (82.1%↓)
	GCoDE	**92.2~92.8**	**88.7~89.7**	**Co**	**39.0 (6.2×↑)**	**0.3 (88.5%↓)**	**50.2 (4.8×↑)**	**0.5 (80.8%↓)**	**35.6 (31.5×↑)**	**0.1 (98.2%↓)**	**49.3 (22.8×↑)**	**0.2 (96.4%↓)**

[21] under the same experimental conditions, averaging results from 10 runs.

Devices and implementation settings. To compare the efficiency of GCoDE and competitors, we employ four device-edge configurations: Jetson TX2 [22] and Raspberry Pi 4B [23] as device, and Nvidia 1060 GPU [24] and Intel i7-7700 CPU [25] as edge. All devices are connected to a wireless router, with varying network conditions simulated by setting upload bandwidth limits ($\mathcal{S}_L$) at 10 Mbps and 40 Mbps. Furthermore, all implementations and tests are conducted on the PyG framework, ensuring test reliability. The architecture search is conducted over a maximum of 2000 iterations and 10 tuning iterations. For predictor training, we randomly sampled 9K co-inference architectures (70%/30% for training/validation) and trained them for 200 epochs using MAPE as the loss function.

4.2 Evaluation on ModelNet40

GCoDE vs. Existing approaches. Tab. 2 compares the task accuracy, latency, and on-device energy consumption of GCoDE against all baselines. To demonstrate the overall enhancement from architecture-mapping co-design and performance awareness on co-inference efficiency, we also compare the performance of all baselines under various collaboration modes. It is clear to see that the performance improvement of GCoDE in device-edge co-inference significantly surpasses existing manual designs and NAS methods without sacrificing accuracy. In case of better network conditions of 40 Mbps, GCoDE can achieve up to 44.9× speedup compared to DGCNN. Against hardware-efficient GNNs designed by HGNAS for edge devices, GCoDE consistently shows optimal performance, achieving up to 9.7× speedup on the lower-powered Raspberry Pi. Compared with the GNN co-inference method BRANCHY, GCoDE can also achieve up to 21.7× speedup, highlighting the importance of performance-awareness in architecture design. Compared to HGNAS with its best partitioning point selection, GCoDE still achieves up to 2.5× inference speedup. Notably, processing all data at the Edge (Edge-Only mode) often fails to achieve optimal performance, primarily because of high communication overhead and wasted computing power of the device. In case of worse network conditions of 10 Mbps, GCoDE maintains its superiority over all baselines by leveraging its environmental awareness, achieving up to 14.9× speedup compared to BRANCHY. Additionally, GCoDE offers substantial on-device energy savings, outperforming all baselines with up to 98.2% reduction in energy consumption, enhancing the potential of GNNs on resource-limited edge devices. The results show that the architecture and deployment mapping co-design of GCoDE allows both device and edge to achieve their full potential and optimal system performance. Moreover, the advanced system performance awareness method of GCoDE effectively identifies optimal solutions in the device-edge co-inference paradigm, ensuring scalability across various system configurations.

Accuracy vs. Latency. Fig. 8 presents the GNN design space exploration results using Jetson TX2 as the device. Compared to all baselines,

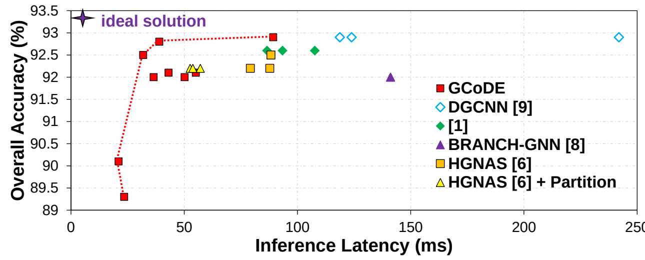


Figure 8: Comparison between the existing approaches and GCoDE in terms of accuracy and latency.

Table 3: Comparison of existing methods and GCoDE on MR.

		GCoDE	BRANCHY [8]	PNAS [2]		PNAS [2]+Partition
Accuracy (%)		**76.1~77.0**	75.5	76.7		76.7
Mode		**Co**	Co	D	E	Co
TX2 ⇌ 1060	Latency (ms)	**8.70**	26.38	29.10	30.70	16.21
	Energy (J)	**0.08**	0.26	0.31	0.28	0.17
TX2 ⇌ i7	Latency (ms)	**8.50**	29.00	29.10	18.60	15.52
	Energy (J)	**0.08**	0.28	0.31	0.19	0.16
Pi ⇌ 1060	Latency (ms)	**4.80**	32.30	13.60	32.00	7.96
	Energy (J)	**0.03**	0.15	0.07	0.14	0.04
Pi ⇌ i7	Latency (ms)	**2.00**	28.70	13.60	28.70	6.90
	Energy (J)	**0.01**	0.14	0.07	0.11	0.04

GCoDE can significantly push forward the Pareto frontier in GNN inference performance, achieving higher accuracy and lower latency. This improvement is attributed to our proposed system performance predictor, which effectively identifies efficient GNN architectures, bringing GCoDE closer to the ideal solution. Additionally, the selection of scaling factor λ during the constraint-based random search process enables users to customize the GNN co-inference architecture for higher accuracy (smaller λ) or lower latency (larger λ), as needed.

4.3 Evaluation on MR

Tab. 3 reports the evaluation results on the MR dataset with high-dimensional node features, under a 40 Mbps network condition, where GCoDE consistently maintains accuracy and efficiency. Compared to the lightweight GNNs designed by PNAS, GCoDE achieves speedups of 3.3×, 2.2×, 2.8×, and 6.8× in four heterogeneous system configurations, respectively. Against co-inference approaches like BRANCHY and PNAS with optimal partitioning points, GCoDE achieves up to 14.3× speedup. Moreover, GCoDE stands out as the most energy-efficient approach compared to all baselines, requiring only 0.01 J on the Raspberry Pi for a single inference. These results strongly demonstrate the effective adaptation of GCoDE to system heterogeneity and its successful balance between communication and computation.

4.4 Evaluation on System Performance Awareness

Fig. 9(a) shows that the proposed system performance predictor achieves 72.4% to 85.3% latency prediction accuracy across various system configurations within a 10% error bound. The high accuracy is attributed to the

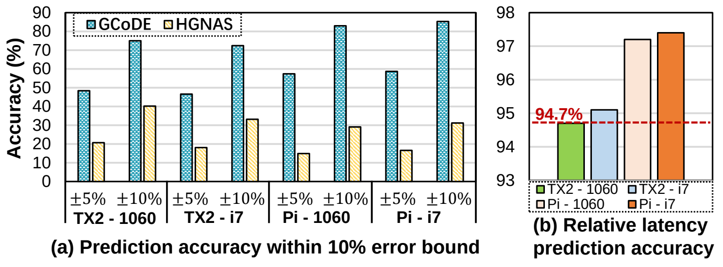


Figure 9: Latency prediction accuracy in co-inference systems.

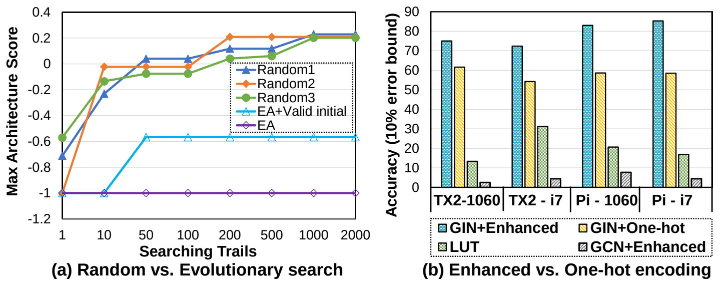


Figure 10: (a) Efficiency comparison between random- and evolutionary search. (b) Prediction accuracy improvement with GCoDE method (GIN + Enhanced feature).

node feature enhancement approach, which considerably enhances the capacity of the predictor to discern hardware sensitivities in heterogeneous devices. In contrast, HGNAS uses a one-hot encoding strategy for node feature initialization, leading to subpar prediction accuracy due to the lack of system heterogeneity information. Furthermore, Fig. 9(b) demonstrates that GCoDE accurately evaluates the relative latency relationship among candidate architectures, achieving more than 94.7% accuracy. The effective capture of the relative latency relationship enables GCoDE to identify the more efficient architectures during the exploration. Besides, our proposed cost estimation method, which requires no training overheads, attains over 88% accuracy in predicting relative latency. The results demonstrate that system performance awareness approaches of GCoDE effectively meet various application requirements and facilitate the exploration of efficient GNN co-inference designs.

4.5 Ablation Studies

Random vs. Evolutionary search. Fig. 10(a) compares the search efficiency of random and evolutionary approaches. The results show that the EA gets stuck in a cycle of identifying valid architectures, thus missing chances to find higher-performing ones. Even with an initial population of valid architectures, its performance remains sub-optimal. In contrast, the proposed constraint-based random search strategy excels, finding optimal architectures within 2000 trails (less than 3 GPU hours). Moreover, it also provides more freedom for GCoDE to explore several optimal solutions during a single search, enabling the building of GNN architecture zoo without additional overheads.

Enhanced vs. One-hot encoding strategy. Fig. 10(b) shows that while the LUT method maintains accuracy in perceiving relative latency relationships, it is far from the truth latency value. Additionally, GIN outperforms GCN in latency learning for architecture graphs due to its superior graph information learning capability. Moreover, with the powerful graph information extraction capability of GIN, the one-hot encoding strategy obtains an improvement in accuracy, but is still ineffective. In contrast, our feature enhancement method significantly improves prediction accuracy in diverse heterogeneous co-inference systems.

4.6 Insight from GNNs Designed by GCoDE

Fig. 11 visualizes the GNN designs by GCoDE tailored for the TX2-i7 co-inference system. The results clearly show that GNNs designed by GCoDE with system-aware and architecture-mapping co-optimization, align effectively with heterogeneous hardware characteristics and balance communication-computation trade-offs, mirroring the observations in the

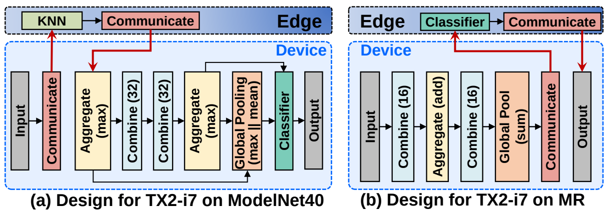


Figure 11: Visualization of GNNs designed by GCoDE.

Motivation section. For the ModelNet40 dataset, GCoDE maps *KNN* operation, which is inefficient on Jetson TX2, to the *KNN*-friendly Intel i7, achieving improved performance with minimal communication costs (low feature dimension). For the MR dataset, GCoDE allocates *Combine* operation, which is a bottleneck on Intel i7, to Jetson TX2 and transfers the intermediate data to Intel i7 after feature dimension reduction by *Global Pooling*. Additionally, all designed architectures by GCoDE have been significantly simplified by eliminating redundant operations and function scale-down tuning. Moreover, the co-inference engine executes operations on the device and edge in a pipelined manner, effectively mitigating communication overhead.

5 Conclusions

This paper introduces GCoDE, the pioneering system-aware automated framework for designing and deploying GNNs on device-edge co-inference systems. GCoDE can automatically design tailored GNN architectures and deploy them on the target system using an efficient co-inference engine and runtime dispatcher. GCoDE builds a unified architecture-mapping co-design space, leveraging constraint-based search strategies and accurate system performance awareness approaches to identify optimal solutions. Extensive experiments show that GCoDE achieves superior accuracy, inference speed, and energy efficiency across diverse applications and systems, surpassing current SOTA methods with up to 44.9× speedup and 98.2% energy savings. We believe that GCoDE has made an important heuristic step towards the design and deployment of efficient GNNs for large-scale wireless network edge applications.

References

[1] Yawei Li et al. Towards efficient graph convolutional networks for point cloud handling. In *Proceedings of ICCV*, 2021.
[2] Lanning Wei et al. Neural architecture search for GNN-based graph classification. *ACM TOIS*, 2023.
[3] Shih-Yuan Yu et al. Scene-graph augmented data-driven risk assessment of autonomous vehicle decisions. *IEEE TITS*, 2021.
[4] Feiyu Chen et al. Multivariate, multi-frequency and multimodal: Rethinking graph neural networks for emotion recognition in conversation. In *Proceedings of CVPR*, 2023.
[5] Yongan Zhang et al. G-CoS: GNN-accelerator co-search towards both better accuracy and efficiency. In *Proceedings of ICCAD*, 2021.
[6] Ao Zhou et al. Hardware-aware graph neural network automated design for edge computing platforms. In *Proceddings of DAC*, 2023.
[7] Jingyi Li et al. Roulette: A semantic privacy-preserving device-edge collaborative inference framework for deep learning classification tasks. *IEEE TMC*, 2023.
[8] Jiawei Shao et al. BRANCHY-GNN: A device-edge co-inference framework for efficient point cloud processing. In *Proceddings of ICASSP*, 2021.
[9] Yue Wang et al. Dynamic graph CNN for learning on point clouds. *ACM TOG*, 2019.
[10] Zhirong Wu et al. 3D ShapeNets: A deep representation for volumetric shapes. In *Proceedings of CVPR*, 2015.
[11] Mohanad Odema et al. LENS: Layer distribution enabled neural architecture search in edge-cloud hierarchies. In *Proceddings of DAC*, 2021.
[12] Mohanad Odema et al. MaGNAS: A mapping-aware graph neural architecture search framework for heterogeneous MPSoC deployment. *ACM TECS*, 2023.
[13] Hadjer Benmeziane et al. A comprehensive survey on hardware-aware neural architecture search. *arXiv preprint arXiv:2101.09336*, 2021.
[14] Yingjie Qi et al. Architectural implications of GNN aggregation programming abstractions. *IEEE CAL*, 2023.
[15] Kaicheng Yu et al. Evaluating the search phase of neural architecture search. In *Proceedings of ICLR*, 2020.
[16] Keyulu Xu et al. How powerful are graph neural networks? In *Proceedings of ICLR*, 2019.
[17] Junxian Huang et al. A close examination of performance and power characteristics of 4G LTE networks. In *Proceedings of MobiSys*, 2012.
[18] Python socket. [Online]. Available: https://docs.python.org/3/library/socket.html.
[19] zlib tool. [Online]. Available: https://www.zlib.net.
[20] Yufeng Zhang et al. Every document owns its structure: Inductive text classification via graph neural networks. In *Proceedings of ACL*, 2020.
[21] Matthias Fey et al. Fast graph representation learning with PyTorch Geometric. In *Proceedings of ICLR*, 2019.
[22] NVIDIA. Jetson TX2. [Online]. Available: https://www.nvidia.com.
[23] Raspberry Pi 4B. [Online]. Available: https://www.raspberrypi.com.
[24] NVIDIA. GeForce GTX1060. [Online]. Available: https://www.nvidia.com.
[25] Intel. Core i7-7700 Processor. [Online]. Available: https://www.intel.com.