GNNavigato: Towards Adaptive Training of Graph Neural Networks via Automatic Guideline Exploration*

Tong Qiao
Beihang University

Jianlei Yang
Beihang University

Chen Bai
CUHK

Yingjie Qi
Beihang University

Wei Sheng Zhao
Beihang University

Chunming Hu
Beihang University

Abstract

Graph Neural Networks (GNNs) succeed significantly in many applications recently. However, balancing GNNs training runtime cost, memory consumption, and attainable accuracy for various applications is non-trivial. Previous training methodologies suffer from inferior adaptability and lack a unified training optimization solution. To address the problem, this work proposes GNNavigator, an adaptive GNN training configuration optimization framework. GNNavigator meets diverse GNN application requirements due to our unified software-hardware co-abstraction, proposed GNNs training performance model, and practical design space exploration solution. Experimental results show that GNNavigator can achieve up to 3.1× speedup and 44.9% peak memory reduction with comparable accuracy to state-of-the-art approaches.

Keywords

GNNs, Training Guidelines, Design Space Exploration

1 Introduction

Graph neural networks (GNNs) have attained significant success across a wide range of graph-based applications, such as node classification [1, 2], link prediction, community detection and flow forecasting. Thanks to the information propagation along edges, GNNs exhibit the ability to capture intricate patterns and relationships within graph data, significantly surpassing traditional deep learning approaches. However, due to the neighborhood explosion, GNNs face more serious challenges than traditional deep learning in terms of accuracy, execution time. Many efforts have been made to address the challenges, which can be generally categorized based on their optimization goals as accuracy centrict optimization, time efficiencycentric optimization, and memory footprint optimization. To minimize feature retrieving traffic, PaGraph [3] and BGL [4] introduce feature caching policies, utilizing free GPU memory for caching. Works such as FastGCN [5], GraphSAINT [6] leverage the locality of graph data for more efficient neighbor sampling [7]. To enhance GNNs computation performance, [8, 9] develop GPU kernels and thread assignment policies, while [10] design accelerators for GNNs training. Additionally, there are many other works focused on optimizations such as workflow pipelining [11], dedicated task scheduling [12], and feature data compression [13].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC ’24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06...
https://doi.org/10.1145/3649329.3656504

Unfortunately, all aforementioned optimization strategies are not sufficient for tackling existing problems. Firstly, without a comprehensive view of GNNs training, most existing approaches perform well only under specific scenarios. As depicted in Fig. 1, existing works typically achieve their claimed excellent performance by making trade-offs among different metrics. The speedup of PaGraph [3] largely depends on the extra consumption of memory resources. And compared to PaGraph [3], 2PGraph [14] achieves 2.45× speedup at the cost of a 3% drop in training accuracy. The adaptability of existing works becomes limited when confronted with diverse application requirements and scenario constraints. Secondly, the collaboration among different optimization strategies is disappointing. Regardless of the difficulties in strategies combination, simply linking multiple strategies may compromise their performance due to incompatibility. Finally, many previous works need careful adjustments in configuration to ensure their performance. This process largely relies on essential expertise and requires significant human effort. To this end, enabling automatic exploration for adaptive solutions with low overhead is valuable, according to the varying requirements of graph-based applications.

In this paper, we introduce a novel GNNs training framework called GNNavigator. GNNavigator can automatically generate effective training guidelines based on application requirements. Our approach distinguishes itself from other GNNs training optimizations by its adaptability to applications prioritizing different performance metrics. Furthermore, many existing optimization strategies can be easily reproduced through simple reconfiguration within GNNavigator framework. Consequently, GNNavigator always achieves excellent performance comparable to or better than previous works. Our contributions are summarized as follows:

- **Unified optimizations abstraction.** We decompose GNNs training into several components, categorizing and abstracting various optimizations according to the decomposition.

- **Reconfigurable runtime backend.** Upon the abstractions, we build a reconfigurable runtime backend to support diverse optimizations by simply reconfiguring.

- **“Gray-box” performance estimating model.** We construct a “gray-box” model, combining theoretical analysis and machine learning, for accurate GNNs training performance estimation.

![Figure 1: Profiling on existing GNN training frameworks.](image-url)
Algorithm 1: Mini-batch based GNNs training on heterogeneous platforms.

Input: graph \( G(V, E) \), batch size \(|B_0^*|\), initial graph network \( M(L, \Phi_{init}) \), network layers \( L \), network parameters \( \Phi \).
Output: converged graph network \( M(L, \Phi_{trained}) \).

1. for \( i \in [0, |V|/|B_0^*|) \) do
   - Component 1: Sampling on Host
     2. \( G_i(V_i, E_i) \leftarrow \text{SubgraphSampling}(G(V, E), B_0^*) \)
   - Component 2: Transmission
     3. MemcpyHToD(G_i)
   - Component 3: Computation on Device
     4. for \( l \in L \) do
        5. \( a_l^i \leftarrow \text{Aggregate}(G_i, M) \)
        6. \( h_l^i \leftarrow \text{Combine}(a_l^i, M) \)
        7. \( \text{loss} \leftarrow \text{LossFunction}(h_l^i, G_i) \)
        8. Backwards()
     9. end for
   10. end for
11. MemcpyDtoH(M(L, \Phi_{trained}))
12. return \( M(L, \Phi_{trained}) \)

- Adaptive training guidelines. With the assistance of performance estimation, GNNavigator provides training guidelines adaptive to application requirements automatically.

2 Background and Motivations

In this section, we outline the problem boundaries of GNNavigator in Sec. 2.1 and Sec. 2.2, and discuss the motivations inspired by several key observations in GNNs training in Sec. 2.3.

2.1 Mini-batch based GNNs Training

The computation of GNNs can typically be described by their aggregate function and combine function. On a graph \( G(V, E) \) with vertex set \( V \) and edge set \( E \), a vertex \( v \in V \) has a feature vector \( h_v^0 \), and \( V(v) \) represents its neighborhood. Let \( e_{uv} \) be the edge feature at layer \( l - 1 \). The computation of a single GNN layer \( l \) can be formulated as:

\[
\begin{align*}
a_l^i & = \text{Aggregate}\left(h_l^{i-1}, e_{uv} \mid u \in V(v) \cup h_{l-1}^0\right), \\
h_l^i & = \text{Combine}\left(a_l^i\right),
\end{align*}
\]

where \( a_l^i \) is the aggregate result and \( h_l^i \) is the vertex embedding of \( v \) at layer \( l \). By stacking multiple GNN layers together, we can get the final output of graph neural networks [15].

To enable GNNs training on tremendously large-scale graphs, mini-batch based training has been introduced. It conducts training on subgraphs iteratively, to alleviate the ever-increasing requirements in memory. Mini-batch based training first samples a subgraph \( G_i(V_i, E_i) \) from \( G(V, E) \) as mini-batch, and then train the network on a series of mini-batches.

2.2 Heterogeneous Platforms for GNNs Training

GNN training has been explored across diverse hardware platforms. The two most influential frameworks, PyG [16] and DGL [17], both support GNNs training on heterogeneous architectures like CPU-GPU. Aligraph and Euler [18] focus on CPU-only platforms to enable flexible computation patterns. Many other works use FPGAs [19, 20] or even accelerators [10, 21] as computation platforms, leading to a notable reduction in time cost or energy consumption.

However, regardless of the diversity in hardware, it is still the mainstream to train GNNs with heterogeneous platforms. As illustrated in Algo. 1, complex operations such as sampling and file I/O, are executed on general-proposed platforms such as CPUs, which we call host. Massive but simple operations such as aggregate and combine are conducted on dedicated designed platforms such as GPUs or FPGAs, which we call device. Furthermore, host and device can exchange data through host-device links, which can be implied through PCIe or DMA. Remarkably, based on the assumption that data retrieving within a certain platform is always much faster than fetching data from another platform through data links, redundant memory resources on device can be treated as a cache to store partial graph data, aiming to accelerate GNNs training [3, 4].

2.3 Observations and Opportunities

Algo. 1 lists the overview of mini-batch based GNNs training on heterogeneous platforms. The number of mini-batches \(|V|/|B_0^*|\) is decided in line 1. Given the target vertices set \( B_0^* \) of each iteration, the mini-batch \( G_i(V_i, E_i) \) is deduced according to the specific sampling algorithm (line 2). The sampled subgraph is then transferred to device through links between host and device (line 3). Then, GNNs are trained on device across the sampled mini-batches (line 4 to 8). The trained model is transferred back to host for further processing (line 11).

We outline 4 categories of training optimization opportunities based on Algo. 1, i.e., sampling, transmission, computation, and model design. Three requirements are summarized, given the limitations of previous related approaches.

- Compatibility. The framework should be compatible with many dedicatedly designed GNN training optimizations. For example, FPGA-orient optimization [20] is orthogonal to GNNavigator [9]. However, compatibility permits a feasible joint optimization by combining these two methodologies.

- Adaptability. The framework should be adaptable to various applications. Different GNN applications pertain to various characteristics, emphasizing runtime performance or hardware budgets. It is non-trivial to find a sweet one-for-all solution. Adaptability allows the framework to produce optimal training optimization strategies given different scenarios.

- Automation. The framework should be automated. The automation alleviates heavy labor force input in deciding optimal training optimization parameters. Inspired by BOOM-Explorer [22], we formulate the automation process as a design space exploration (DSE) problem and solve it via our customized surrogate model.

3 GNNavigator Framework

Motivated by the observations and opportunities outlined in Sec. 2.3, we introduce GNNavigator, an adaptive framework automatically fine-tuning GNNs training according to application requirements and hardware constraints. GNNavigator is constructed upon three pivotal techniques: 1) a unified and reconfigurable back-end facilitating efficient strategy cooperation, 2) a "gray-box" performance estimator, and 3) an application-driven design space exploration tailored to requirements and constraints.

3.1 Framework Overview

Fig. 2 provides an overview of GNNavigator, and its general workflow. For better adaptability, GNNavigator requires some essential information, typically related to the applications, as input.

Users should specify the following input items:

- The graph dataset \( G(V, E) \) to be trained on.
- The GNN model \( M(L, \Phi_{init}) \), with explicit network architecture.
- The application requirements like time cost \( T \), memory consumption \( \Gamma \), accuracy \( Acc \), etc., along with the user-defined priorities for different requirements.
- The heterogeneous hardware platforms for GNNs training.

The user inputs are quantitatively analyzed to formulate the parameterized explore targets and runtime constraints, as shown in Step 1. Then, in Step 2, GNNavigator automatically generates GNN
In our abstraction of samplers, it receives a certain number of target vertices $B_l^{l-1}$ as input from layer $l-1$, and fanouts $k^l_i$ neighbors from every vertex $v_l^{l-1} \in B_l^{l-1}$ at a given probability. The output $B_l^l$ at layer $l$ can be formulated as follows:

$$B_l^l = \bigcup_{v_l^{l-1} \in B_l^{l-1}} u \cdot \gamma(p(\eta)(\frac{k^l}{|N'(v_l^{l-1})|}), u \in N'(v_l^{l-1}).$$ (2)

$\gamma(p(\eta))$ is an indicator function to decide whether to select a neighbor $u$, according to a probability $p(\eta)$ specified by the sampling algorithm.

While Eq. 2 is primarily in the form of node-wise sampling, it can generalize other sampling strategies as well. For instance, in the case of layer-wise sampling [5], the number of sampled nodes at layer $l$ can be represented as $\Delta_l$, which is a predetermined value. We can derive the mathematical expectation of $k^l$ from $\Delta_l$ by:

$$E(k^l) = \frac{\Delta_l}{|B_l^{l-1}|} \cdot \mu(p(\eta), B_l^{l-1}),$$ (3)

where $\mu(p(\eta), B_l^{l-1})$ is a coefficient which indicates the probability of multi-vertices in $B_l^{l-1}$ shares a common neighbor in $B_l^l$. In this way, layer-wise sampling has been uniformly abstract as Eq. 2. Locality-aware sampling and subgraph-wise sampling can be more easily integrated into the abstraction, according to their sampling patterns. By setting the neighbor selection probability to a function of data locality $p(\eta)$, we can reproduce biased samplers that prefer a certain subset of $V$. Subgraph-wise sampling strategies like GraphSAIN [6] can be viewed as a special case of node-wise sampling, with many more hops, but only a single neighbor fanout in each hop. To this end, we can unify different sampling strategies to the sampler in Fig. 3, with configurable settings being enumerated.

Transmission strategies. Regardless of the implementation of transmission strategies, they always ensure the required data being on the device when computing. Note that not all required data needs transmission. An abstraction can be drawn, according to the gap between the required data volume and actual transferred data volume. The transmission strategies typically leverage the free memory resources on device as a cache to alleviate redundant data transmission. Despite their substantial differences in cache updating policies, we can consistently abstract them as follows. First, the device cache is initialized according to the available memory resource on device. Given a mini-batch, the device cache figures out which part of the mini-batch has been cached. The remaining part is filtered out from the host and transferred to the device through host-device links. With all essential data on device, training on the mini-batch can be conducted. Finally, the device cache is updated according to the cache updating policy. Uniformly, part of the configurable settings on transmission are listed in the device cache in Fig. 3. We can distinguish different transmission strategies by configuring the settings properly.
**Figure 4: Automatic guidelines exploration.**

Computation and model design optimizations. GNN models typically embed graph topological information through aggregate functions and enable feature learning by combine functions. Let along the detailed design of models, abstraction of model design optimizations can be formulated based on the time complexity and spatial complexity of aggregate and combine functions, as shown in Eq. 1. Similarly, computation optimizations are abstracted by their maximum throughput and available memory resources. Although the computation optimizations show significant diversity in their targeting device platforms, ranging from GPUs to FPGAs, we find that they can all be measured by their computing capability and available resources.

In Fig. 3, both computation optimizations and model design optimizations are mapped to the device, for actual execution and the following performance estimation.

Reconfigurable runtime backend. Thanks to its reconfigurability, the runtime backend can represent itself with diverse performance, making it adaptive to a wide range of applications.

Fig. 3 depicts the mapping relationships between the four optimization categories and backend components, marked by red words. Within each backend component, there are many reconfigurable settings that can be freely adjusted to simulate existing approaches, marked with blue dash-line rectangles. For instance, by disabling cache update policy, and properly configuring the cache ratio, the backend generally reproduces the approach proposed in PaGraph [3]. Similarly, many existing works can be conveniently reproduced, by applying the configurations setting templates shown in Fig. 3. Furthermore, the unified runtime backend allows for a flexible combination of optimizations from different categories and greatly outperforms existing works in its compatibility. Note that the runtime backend can even incrementally support future optimizations only if they submit to our abstraction. Additionally, all reconfigurable parameters in the runtime backend make up the design space, which will be discussed in detail in Sec. 3.3.

### 3.3 Automatic Guidelines Generation

GNNavigator leverages multi-objective design space exploration (DSE) to automatically generate the training guidelines, as shown in Fig. 4. It benefits from multi-objective DSE in two key aspects. Firstly, in terms of fine-tuning GNN training, the substantial burden of human labor is mitigated through automatic exploration. Secondly, in terms of adaptability, the explorer generates guidelines that satisfy user demands by emphasizing different explore targets. Notably, all the reconfigurable settings in Fig. 3 constitute the design space, represented by blue dash-line rectangles. Moreover, to accelerate the exploration, a "gray-box" performance estimator is established based on the unified abstractions of optimization strategies.

**Gray-box performance estimator.** The estimator predicts GNN training performance in a "gray-box" manner, combining purely theoretical analysis (white-box) and machine learning methods (black-box) together.

As shown in Fig. 4, the estimator makes predictions based on 1) the specific values of all configurable settings, which will be represented as candidate in design space in our following illustration and 2) the pre-determined settings in runtime, usually determined by applications. To ensure the accuracy of estimation, the estimator first theoretically analyzes the data dependence between its inputs and performance $\text{Perf}\{T, \Gamma, \text{Acc}\}$. Considering the complexity and randomness of graph, black-box models based on machine learning are introduced to estimate some key intermediate variables that influence $T, \Gamma$, and Acc. We will present the methodology of constructing "gray-box" models on performance $\text{Perf}$ as follows.

Note that the operations on $\text{device}$ are independent of those on $\text{host}$. The epoch time can be formulated as:

$$T = n_{\text{iter}} \cdot \max\{t_{\text{sample}} + t_{\text{transfer}}, t_{\text{replace}} + t_{\text{compute}}\}, \quad (4)$$

where $t_{\text{sample}}$, $t_{\text{transfer}}$, $t_{\text{replace}}$, and $t_{\text{compute}}$ represent the time cost of sampling, transmission, cache updating, and computation on $\text{device}$, respectively, and $n_{\text{iter}}$ is the number of mini-batches within an epoch. Let us begin with $t_{\text{replace}}$. In scenarios requiring cache replacement, the cache updating overhead is mainly influenced by cache volume $r \cdot |V|$, and volume of replaced stale data $|V|((1 - hit))$.

$$t_{\text{replace}} = f_{\text{replace}}(r \cdot |V|, |V|((1 - hit)), \text{Device}). \quad (5)$$

The hit represents the average cache hit rate, and we use $\text{Host Device}$ to indicate the hardware information of $\text{host}$ and $\text{device}$ respectively.

In a similar fashion, $t_{\text{transfer}}$ is determined by the volume of data awaiting transmission $n_{\text{attr}} |V|((1 - hit))$. $t_{\text{sample}}$ is primarily affected by changes in subgraph size $|V| - |B^0|$, which represents the transition from the original target vertices to the final mini-batch. Lastly, mini-batch size $|V|$ and the GNN model $M(L, \Phi)$ together determine $t_{\text{compute}}$. We demonstrate the formulations of $t_{\text{replace}}$, $t_{\text{transfer}}$, and $t_{\text{sample}}$, as follows:

$$t_{\text{transfer}} = f_{\text{transfer}}(n_{\text{attr}} |V|((1 - hit)), \text{Host}, \text{Device}), \quad (6)$$

$$t_{\text{sample}} = f_{\text{sample}}(|V| - |B^0|, \text{Host}), \quad (7)$$

$$t_{\text{compute}} = f_{\text{compute}}(\Gamma, r, \text{Device}). \quad (8)$$

The term $n_{\text{attr}}$ denotes the attribute dimensions of an individual node.

Remarkably, the functions $f_{\text{compute}}$, $f_{\text{replace}}$, $f_{\text{transfer}}$, and $f_{\text{sample}}$ can all be estimated using a pre-trained black-box model. In this way, the performance estimator can predict the execution time of GNN training with negligible latency.

The prediction of device memory consumption $\Gamma$, can also be decomposed to sub-tasks of estimating $\Gamma_{\text{model}}$, $\Gamma_{\text{cache}}$, and $\Gamma_{\text{runtime}}$ respectively,

$$\Gamma = \Gamma_{\text{model}} + \Gamma_{\text{cache}} + \Gamma_{\text{runtime}}, \quad (9)$$

where $\Gamma_{\text{model}}$, $\Gamma_{\text{cache}}$, and $\Gamma_{\text{runtime}}$ are formulated as follows:

$$\Gamma_{\text{model}} = \Phi, \quad (10)$$

$$\Gamma_{\text{cache}} = f_{\text{cache}}(r \cdot |\text{attr}|), \quad (11)$$

$$\Gamma_{\text{runtime}} = f_{\text{runtime}}(|V|, \Phi). \quad (12)$$

$\Gamma_{\text{model}}$ reveals the static memory consumption of GNNs, directly related to $|\text{init}|$. $\Gamma_{\text{cache}}$ represents the cache memory consumption, and $\Gamma_{\text{runtime}}$ indicates the memory footprint of mini-batch computation phrase.
Table 1: Performance of GNNavigator across different tasks.

<table>
<thead>
<tr>
<th>Applications (Dataset+Model)</th>
<th>Method</th>
<th>Time (/s)</th>
<th>Memory (T/GB)</th>
<th>Accuracy (T/%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR + SAGE</td>
<td>PyG [16]</td>
<td>9.27</td>
<td>1.25</td>
<td>90.55%</td>
</tr>
<tr>
<td></td>
<td>Pa-Full [3]</td>
<td>5.44 (1.7x ↑)</td>
<td>2.11 (69.1% ↑)</td>
<td>90.42%</td>
</tr>
<tr>
<td></td>
<td>Pa-low [3]</td>
<td>8.39 (1.1x ↑)</td>
<td>1.34 (7.5% ↑)</td>
<td>90.58%</td>
</tr>
<tr>
<td></td>
<td>2P [14]</td>
<td>5.18 (1.8x ↑)</td>
<td>0.88 (29.7% ↓)</td>
<td>90.36%</td>
</tr>
<tr>
<td></td>
<td>Bal</td>
<td>3.67 (2.5x ↑)</td>
<td>1.22 (3.5% ↓)</td>
<td>91.19%</td>
</tr>
<tr>
<td></td>
<td>Ex-TM</td>
<td>3.95 (2.3x ↑)</td>
<td>0.79 (37.3% ↓)</td>
<td>90.37%</td>
</tr>
<tr>
<td></td>
<td>Ex-MA</td>
<td>5.12 (1.8x ↑)</td>
<td>0.86 (29.7% ↓)</td>
<td>91.22%</td>
</tr>
<tr>
<td></td>
<td>Ex-TA</td>
<td>3.59 (2.6x ↑)</td>
<td>1.64 (31.1% ↓)</td>
<td>91.24%</td>
</tr>
<tr>
<td>AR + GAT</td>
<td>PyG [16]</td>
<td>7.68</td>
<td>1.32</td>
<td>79.28%</td>
</tr>
<tr>
<td></td>
<td>Pa-Full [3]</td>
<td>3.78 (2.0x ↑)</td>
<td>1.80 (36.3% ↑)</td>
<td>79.23%</td>
</tr>
<tr>
<td></td>
<td>Pa-low [3]</td>
<td>7.04 (1.1x ↑)</td>
<td>1.43 (36.7% ↑)</td>
<td>79.23%</td>
</tr>
<tr>
<td></td>
<td>2P [14]</td>
<td>3.51 (2.2x ↑)</td>
<td>0.84 (36.3% ↓)</td>
<td>75.95%</td>
</tr>
<tr>
<td></td>
<td>Bal</td>
<td>3.53 (2.1x ↑)</td>
<td>0.87 (31.4% ↓)</td>
<td>80.03%</td>
</tr>
<tr>
<td></td>
<td>Ex-TM</td>
<td>2.43 (3.1x ↑)</td>
<td>0.99 (44.9% ↓)</td>
<td>76.42%</td>
</tr>
<tr>
<td></td>
<td>Ex-MA</td>
<td>3.82 (2.0x ↑)</td>
<td>0.91 (31.1% ↓)</td>
<td>81.16%</td>
</tr>
<tr>
<td></td>
<td>Ex-TA</td>
<td>2.85 (2.7x ↑)</td>
<td>0.99 (25.0% ↓)</td>
<td>79.87%</td>
</tr>
<tr>
<td>RD2 + SAGE</td>
<td>PyG [16]</td>
<td>3.49</td>
<td>5.80</td>
<td>61.44%</td>
</tr>
<tr>
<td></td>
<td>Pa-Full [3]</td>
<td>2.98 (1.2x ↑)</td>
<td>5.87 (1.3% ↓)</td>
<td>61.38%</td>
</tr>
<tr>
<td></td>
<td>Pa-low [3]</td>
<td>3.46 (1.0x ↑)</td>
<td>5.80 (1.1% ↓)</td>
<td>61.45%</td>
</tr>
<tr>
<td></td>
<td>2P [14]</td>
<td>3.53 (1.0x ↑)</td>
<td>5.80 (1.2% ↓)</td>
<td>60.51%</td>
</tr>
<tr>
<td></td>
<td>Bal</td>
<td>2.98 (1.2x ↑)</td>
<td>5.87 (1.3% ↓)</td>
<td>61.43%</td>
</tr>
<tr>
<td></td>
<td>Ex-TM</td>
<td>3.21 (1.1x ↑)</td>
<td>5.84 (0.8% ↓)</td>
<td>61.07%</td>
</tr>
<tr>
<td></td>
<td>Ex-MA</td>
<td>3.23 (1.1x ↑)</td>
<td>5.85 (0.9% ↓)</td>
<td>61.71%</td>
</tr>
<tr>
<td></td>
<td>Ex-TA</td>
<td>2.98 (1.2x ↑)</td>
<td>5.87 (1.3% ↓)</td>
<td>61.43%</td>
</tr>
</tbody>
</table>

4.1 Evaluation Setup

**Baselines.** To evaluate the effectiveness of GNNavigator, we consider 3 baselines: PyG [16], a state-of-the-art GNN framework on both CPUs and GPUs; PaGraph [3], a GNNs computation framework with static cache; 2PGraph [14], a CPU-GPU heterogeneous framework, with cache-aware sampling to accelerate training. PaGraph [3], 2PGraph [14], and original PyG [16] are all reproduced on our runtime backend for a fair comparison. The reproductions achieve similar results as the ones they report. Note that the volume of available GPU memory will significantly influence the performance of PaGraph [3]. Therefore, we measure the performance of PaGraph [3] under ideal circumstances (Pa-Full) and resource-limited circumstances (Pa-Low) respectively.

**Datasets and platforms.** Our experiments are conducted on datasets with various scales, including Ogbn-arxiv (AR), Ogbn-products (PR) [23], Reddit2 (RD2), and representative graph neural networks, including GCN, GAT, GraphSAGE (SAGE). We test the performance of the runtime backend on different devices such as RTX 4090, A100, and M90, and further set manual constraints to simulate various scenarios of application. The time cost and memory footprint are measured by PyTorch profiler [24].

**Performance estimator settings.** The performance estimator is trained on the ground-truth performance covering the whole design space. For fairness, the estimator is established upon the performance across all the datasets available, except the one waiting for estimation. Specifically, to embed more prior knowledge, we randomly generate some power-law graphs and profile the training on them as data enhancement to optimize our performance estimator.

4.2 Overall Performance

GNNavigator provides guidelines from a comprehensive view of performance $\text{Perf} (T, \Gamma, \text{Acc})$. As shown in Tab. 1, when highlighting a balance among $T, \Gamma, \text{Acc}$, GNNavigator generally achieves similar or superior performance compared to the baselines across various GNNs training tasks, denoted as 'balance' (Bal). GNNavigator can further improve training performance in certain metrics, with a marginal trade-off in others, which is marked as 'extreme' (Ex) in Tab. 1. And, Ex-TM, Ex-MA, Ex-TA denote the different priorities of generated guidelines. For example, Ex-TM emphasizes time $T$ and memory $\Gamma$, leading to up to...
5 Conclusions

We present GNNavigator, a GNN training framework with excellent adaptability to various application requirements. GNNavigator automatically generates training guidelines, consistently delivering promising performance across different scenarios. We draw unified abstractions from various optimizations and build a reconfigurable runtime backend based on the abstractions. To accurately predict training performance on our backend, we construct a gray-box performance estimator. GNNavigator further enables automatic exploration of training guidelines adapted to application requirements. Our experiments demonstrate that GNNavigator outperforms state-of-the-art works by up to 3.1× speedup and at most 44.9% reduction in memory consumption, with comparable accuracy.

References


Table 2: Validation of estimator prediction.

<table>
<thead>
<tr>
<th>Prediction Validation</th>
<th>Performance Metric</th>
<th>Reddit</th>
<th>Reddit2</th>
<th>Ogbn-products</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2 Score</td>
<td>Time Cost (T)</td>
<td>0.837</td>
<td>0.782</td>
<td>0.728</td>
</tr>
<tr>
<td></td>
<td>Memory (M)</td>
<td>0.924</td>
<td>0.910</td>
<td>0.7307</td>
</tr>
<tr>
<td>MSE</td>
<td>Accuracy (Acc)</td>
<td>0.0292</td>
<td>0.0249</td>
<td>0.0156</td>
</tr>
</tbody>
</table>