WiseGraph: Optimizing GNN with Joint Workload Partition of Graph and Operations

Kezhao Huang
Tsinghua University

Jidong Zhai
Tsinghua University

Liyan Zheng
Tsinghua University

Haojie Wang
Tsinghua University

Yuyang Jin
Tsinghua University

Qihao Zhang
Tsinghua University

Runqing Zhang
Tsinghua University

Zhen Zheng
Microsoft

Youngmin Yi
University of Seoul

Xipeng Shen
North Carolina State University

Abstract

Graph Neural Network (GNN) has emerged as an important workload for learning on graphs. With the size of graph data and the complexity of GNN model architectures increasing, developing an efficient GNN system grows more important. As GNN has heavy neural computation workloads on a large graph, it is crucial to partition the entire workload into smaller parts for parallel execution and optimization. However, existing approaches separately partition graph data and GNN operations, resulting in inefficiency and large data movement overhead.

To address this problem, we present WiseGraph, a GNN training framework exploring the joint optimization space of graph data partition and GNN operation partition. To bridge the gap between the two classes of partitions, we propose a workload abstraction tailored to GNN, *g*Task, which can not only describe existing GNN partition strategies as special cases but also exploit new optimization opportunities. Based on *g*Tasks, WiseGraph effectively generates partition plans adaptive to input graph data and GNN models. Evaluation on five typical GNN models shows that WiseGraph outperforms existing GNN frameworks by 2.04× and 2.22× for single and multiple GPU training. WiseGraph is publicly available at https://github.com/xxcclong/CxGNN-Compute/.

CCS Concepts: • **Computer systems organization** → *Heterogeneous (hybrid) systems*; • **Computing methodologies** → *Artificial intelligence.*

Keywords: Efficient training method, Graph neural network, Parallelism

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04
https://doi.org/10.1145/3627703.3650063

ACM Reference Format:
Kezhao Huang, Jidong Zhai, Liyan Zheng, Haojie Wang, Yuyang Jin, Qihao Zhang, Runqing Zhang, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2024. WiseGraph: Optimizing GNN with Joint Workload Partition of Graph and Operations. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3627703.3650063

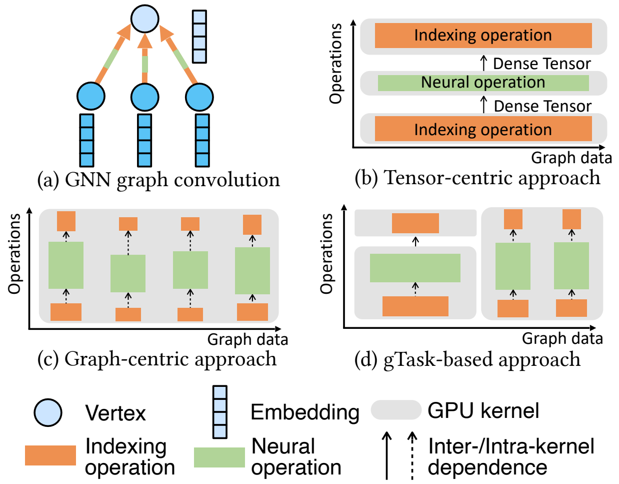
1 Introduction

Graph Neural Network (GNN) has shown promising results in graph-based learning applications while obviating the burdens of manual feature selection or extraction. GNN is becoming a dominant approach for learning applications on graph data, such as social networks [25, 26], recommendation systems [14], molecule structures [13, 46], and knowledge graphs [36, 51].

Unlike classic Deep Neural Networks (DNN), GNN comprises not only dense tensors but also sparse graph data. The input of a GNN typically consists of a graph containing sparse edge connections between vertices, along with dense embedding vectors on vertices and weight parameters from the model itself. Each vertex in the graph carries an embedding vector with a length of hundreds or thousands. To learn from both sparse and dense data, graph convolution is applied in GNNs, which typically consists of two key operations: *indexing operations* and *neural network operations* (*neural operations* for short). As shown in Figure 1(a), indexing operations are used to fetch embeddings from source vertices along graph edges, and then neural operations, such as MLP [15] and Attention [40], are performed to encode the fetched data. At last, the encoded data is reduced to the destination vertex. These operations are organized as a data flow graph (DFG).

As GNN has heavy neural computations on a large graph structure, to efficiently train it with accelerators, such as GPUs, it is crucial to partition the entire workload into smaller parts for parallel execution and optimization. With regard to partition strategies, previous solutions can be categorized into two main types [47].

The first type is **tensor-centric** approach [11, 26, 28], shown in Figure 1(b). In this approach, indexing operations are used to fetch vertex embeddings according to graph data



(a) GNN graph convolution (b) Tensor-centric approach

(c) Graph-centric approach (d) gTask-based approach

Figure 1. The comparison between the approaches of existing work and ours. (a) illustrates GNN computation. Tensor-centric approach (b) partitions operations but regards graph data as a whole. Graph-centric approach (c) partitions graph data but does not partition operations. *g*Task-based approach (d) jointly partitions graph data and operations.

and then produce a dense tensor residing in the GPU memory. Subsequently, a neural operation is executed on this dense tensor. The tensor-centric approach partitions these operations into separate GPU kernels to execute. As these operations work on dense tensors, this approach can achieve high efficiency. However, partitioning all operations into separate kernels can result in significant overhead for data movement and memory consumption (see §2.2).

The second type is a **graph-centric** approach that partitions graph data into multiple parts based on certain rules and maps them to different execution units on GPU, shown in Figure 1(c). For example, vertex-centric [47] and edge-centric [58] approaches partition graph data according to destination vertices and edges respectively. As the graph is partitioned into a number of fine-grained parts, data movement among operations can be reduced by executing all operations within a single GPU kernel. However, current partitions do not consider the operations, resulting in poor parallelism and low efficiency of neural operations.

The fundamental limitation of existing approaches is the separate partition of graph data and operations while ignoring their joint optimization space. Moreover, conventional partition methods are non-adaptive: they use the same partition method for different graphs and operations, ignoring either the complexity of graph data or the diversity of operations. In this work, we propose to *adaptively* explore the *joint* partition space of graph data and operations. Compared to the previous separate partition space, our joint approach constitutes a significantly larger optimization space. As shown in Figure 1(d), we can partition both graph data and operations simultaneously to generate new execution plans.

To explore the joint partition space, we present WiseGraph, a GNN training framework that co-optimizes GNN with regard to graph data and operations. It bridges the partition of graph data and operations through *g*Task, which is a workload abstraction with a subset of edges partitioned from graph data and an operation partition plan. WiseGraph utilizes the information of operations to generate a set of graph partition plans, which batch edges sharing similar patterns in operation execution. After graph partition, WiseGraph leverages *g*Task-level data patterns to optimize GNN operations at different levels and generate candidate operation partition plans. Finally, WiseGraph jointly optimizes with the partitions of graph data and operations by matching a graph partition plan with operation partition plans. In WiseGraph, the main components are described below.

Operation-aware graph data partition. Existing graph-centric approaches partition graph data solely based on graph structure, which can result in low efficiency for operations, as they overlook other graph data that can also have a great impact on the performance. To address this, we propose *operation-aware graph data partition*, which identifies all graph attributes related to memory access in operations and considers them comprehensively when generating graph partition plans and *g*Tasks. Therefore, it can cover existing graph partition plans while discovering other new ones.

Data-pattern-aware operation partition. Existing approaches statically partition operations: they either fuse all operations into one kernel or put them into separate kernels, ignoring the patterns of graph data. However, graph data patterns can significantly influence performance. To address this, we propose *data-pattern-aware operation partition* by exploring *g*Task-level data patterns from graph partition plans. In WiseGraph, there are three key steps for operation partition. It first changes operation organization by transforming the DFG of GNN model. The second is to partition operations into different kernels. The last step is to map operations to different devices for multi-device training. The steps are enabled by the crucial data patterns revealed with *g*Tasks.

Joint optimization of graph and operation partition. With the above techniques exploring the space to partition graph data and operations, WiseGraph generates a number of graph and operation partition plans. For joint optimization, it is infeasible to try all the operation partition plans for every *g*Task due to large graph size. To address this, we propose differentiated joint strategy. For each graph partition plan, we identify those outlier *g*Tasks that are significantly influenced by the irregularity of graph data. Then we separately select operation partition plans for them and reschedule their execution priority and resource.

To summarize, WiseGraph made the following contributions.

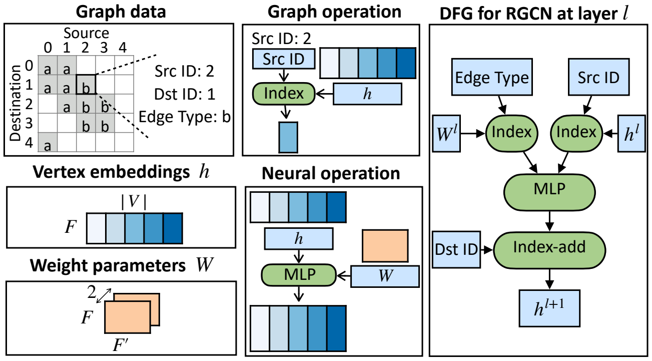
1. WiseGraph proposes a new **joint optimization space** for graph data and operations in GNN, which co-considers graph partition and operation partition;
2. WiseGraph designs *g*Task **as the workload abstraction for joint partition**, which reveals graph data patterns for operation optimization while assigning suitable operation partition plan to individual tasks on graph;
3. WiseGraph develops an end-to-end workflow to partition graph data, optimize operation partitions with data patterns, and search for performant execution plans;
4. This paper presents extensive experiments and results show that WiseGraph gains speedup of 2.04× and 2.27× over state-of-the-art GNN systems on five models on single-GPU and multi-GPU scenarios respectively.

2 Background and Motivation

2.1 GNN Basis

GNN inputs. GNN models work on both sparse and dense data. The sparse part is a graph with vertices V and edges E. As shown in Figure 2(a), the graph is represented by an adjacency matrix with edge attributes. Typical edge attributes include *src-id* and *dst-id*, which indicate the IDs of the source and destination vertex connecting to the edge. There can be more edge attributes illustrating other properties of graph data. In this example, *edge-type* distinguishes the type of the connection. The vertices carry pre-learned or learnable embedding vectors h with a length of F. The vertex embeddings are usually represented by a dense tensor with the shape of $[|V|, F]$. Besides the graph data and vertex embeddings, there are also dense weight parameters in the model that get updated iteratively during training.

GNN operations. With both sparse and dense input data, the computation in GNN models is also in combination of sparse and dense, which corresponds to two types of operations, **indexing** and **neural** operations. Shown in Figure 2(b), indexing operations move data according to graph structure, e.g., moving embedding vectors from source vertices to the connected edges. As the graph structure is irregular, indexing operations are sparse computations. Another type is neural operations, which encode vertex embeddings using weight parameters. The neural operations are inherited from the deep neural network (DNN), which includes both lightweight computations such as element-wise operations and heavy computations such as MLP [15], Attention [39], and LSTM [16]. In Figure 2(b), an MLP operation encodes embedding h with weight parameter W. As neural operations are dense computations, their performance is sensitive to data locality and parallelism.



(a) Input data (b) Operations (c) Data flow graph

Figure 2. Illustration of input data (a), operations (b), and DFG (c) for GNN training.

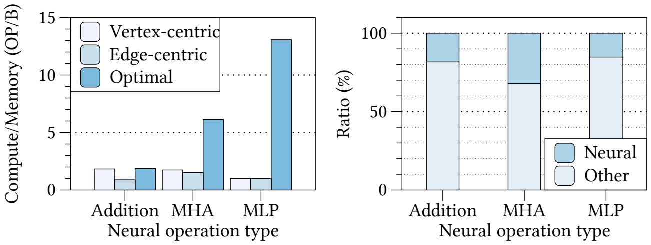
Data flow graph (DFG) of GNN. There are many operations in a GNN model, which is usually represented by a DFG to describe their dependence. A GNN model has multiple layers. In each layer, indexing and neural operations are intertwined, which makes GNN models different from DNN models or graph computing: Embeddings are transmitted according to the graph by indexing operations, while neural operations are performed on the transmitted data. We use RGCN [34], a popular GNN model, as an example for all following illustrations. The computation of RGCN at layer l is shown in Equation (1), and its DFG is depicted in Figure 2(c). It performs MLP on the source vertex feature (h^l) indexed by the edge's source vertex ID (*edge.src-id*) and the weight parameter (W^l) indexed by the edge's type (*edge.type*). After that, it reduces computation results to destination vertex features (h^{l+1}) indexed by destination vertex ID (*edge.dst-id*).

$$h^{l+1}_{edge.dst\_id} \mathrel{+}= \mathrm{MLP}(h^l_{edge.src\_id}, W^l_{edge.type}) \tag{1}$$

2.2 Motivation

To execute the operations of GNN models, graph-centric [42, 43, 47] and tensor-centric [11, 26] approaches partition graph data and operations differently. Graph-centric approaches partition the graph data into a number of fine-grained parts and execute operations for each part in one GPU kernel. Tensor-centric approaches regard the graph data as a whole and use multiple GPU kernels to execute different operations, which is due to the distinct pattern of indexing and neural operations. They process each partition of graph data and operations separately, ignoring the joint optimization space between them.

We elaborate on the main limitations of graph-centric and tensor-centric approaches. Graph-centric (vertex-centric or edge-centric) approaches have low computation efficiency, especially with complex neural operations in GNN model. Figure 3(a) shows the compute/memory ratio for the graph-centric approach on models with different neural operations.



(a) Compute/Memory ratio of vertex- and edge-centric approach

(b) Execution time breakdown for tensor-centric approach

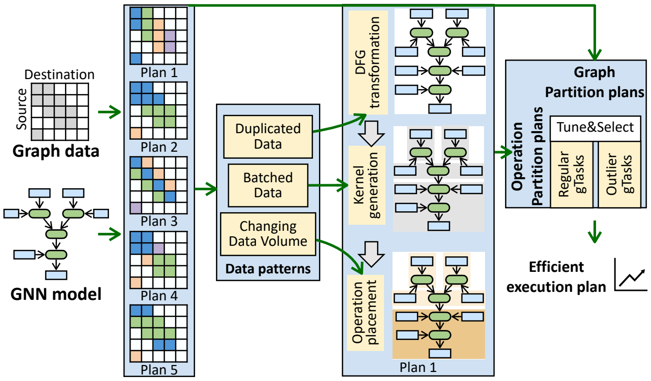
Figure 3. The inefficiency of graph-centric (a) and tensor-centric approaches.

While they are close to the theoretically optimal ratio for simple neural operations (Addition), the gap becomes larger with more complex operations, such as MHA (multi-head attention) and MLP. As a result, graph-centric approach with MLP as neural operation can only reach 1%[1] of the peak performance of GPU. On the other hand, tensor-centric has large redundancy due to graph data. Figure 3(b) shows the normalized execution time breakdown for these models using tensor-centric approaches. The time of executing neural operations (`Neural`) is less than 40%, while most of the time is spent on indexing operations to move data (`Other`) in GPU global memory. Though it has high neural computation efficiency, the tensor-centric approach is still inefficient due to large redundancy of global memory data movement.

3 End-to-end Workflow of WiseGraph

WiseGraph is a GNN framework that explores the joint optimization of graph partition and operation partition, which enables a larger optimization space for GNN models. ***g*Task** is the basis for WiseGraph, which is a workload abstraction with a subset of edges partitioned from graph data and an operation partition plan. With *g*Task, WiseGraph can adaptively generate graph partitions according to both GNN model and graph data, while reveal fine-grained data patterns for operation partition and optimization. The end-to-end workflow of WiseGraph is shown in Figure 4.

The first step of WiseGraph is to generate graph partition plans for *g*Tasks based on graph data and GNN model in Figure 4(a). Figure 4(b) shows multiple generated partition plans: in an adjacency matrix, edges in the same color are partitioned to the same *g*Task. WiseGraph can find and try various graph partition plans, which opens up a large space for graph partition. By comparison, previous work statically sets a fixed graph partition plan, such as vertex-centric partition. WiseGraph achieves that by identifying the edge attributes that are critical to computation efficiency in graph data and then partitioning the graph data into *g*Tasks by applying restrictions on the selected edge attributes.



(a) Input (b) Graph partition (c) Data patterns (d) Operation partition (e) Joint optimization

Figure 4. End-to-end workflow of WiseGraph.

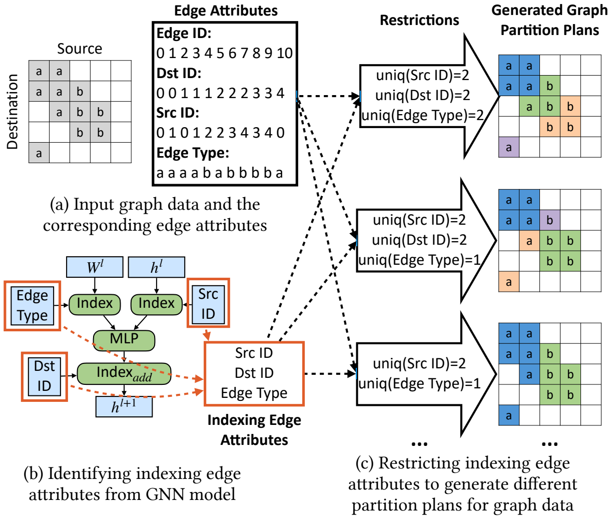
The second step is to generate pattern-aware operation partition plans for *g*Tasks. We find that a graph partition plan with certain restrictions on edge attributes reveals *g*Task-level data patterns. Shown in Figure 4(c), we find three typical types of data patterns. The first is duplicated data, which is about how data is duplicated and leads to computation sharing in a *g*Task. The second is batched data, as there can be a large amount of unique data to be batched for a *g*Task. The third is changing data volume, which is the pattern describing how data volume (size of tensors) changes with operations. Shown in Figure 4(d), these data patterns correspond to three steps of operation partition: With duplicated data, WiseGraph enhances computational sharing through DFG transformation; With batched data, WiseGraph generates efficient kernels to process multiple edges in parallel; With changing data volume, WiseGraph determines an operation placement strategy to scale multi-device training.

Finally, as shown in Figure 4(e), WiseGraph jointly optimizes with both graph and operation partitioning plans. WiseGraph matches the *g*Tasks with different operation partition plans by distinguishing outlier *g*Tasks from regular *g*Tasks. The resulting execution plan can better tackle the irregularity of the graph data for better efficiency.

4 Generating Graph Partition Plans

We first explore the search space for graph partition plans. We take the properties of the GNN model into consideration and only explore graph partition plans that largely influence the execution efficiency of the GNN model. To find out such graph partition plans, our core idea is to analyze the GNN model to first identify the edge attributes that are critical to performance and then apply a series of rules (*restrictions*) on these attributes to generate different graph partition plans in a unified way.

[1] Measured on OGBN-Arxiv, using dense matrix multiplication as the peak GPU performance



(a) Input graph data and the corresponding edge attributes

(b) Identifying indexing edge attributes from GNN model

(c) Restricting indexing edge attributes to generate different partition plans for graph data

Figure 5. Workflow of generating graph partition plans for RGCN; Using the edge attributes in the input graph data (a) and indexing edge attributes identified from the GNN model (b), WiseGraph can apply various restrictions to generate different graph partition plans (c).

4.1 Identifying Key Attributes from GNN Models

Shown in Figure 5(a), the input graph can be represented by multiple edge attributes. For each edge, its ID is indicated by *edge-id*; *src-id* and *dst-id* describe the vertex IDs it is connected to; *edge-type* represents its type.

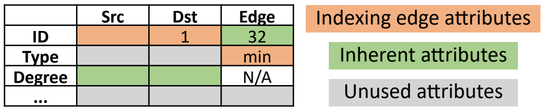
WiseGraph identifies the key attributes according to the GNN model. Shown in Figure 5(b), a GNN model is represented with a DFG, including indexing and neural operations, as well as the input and output tensors. In the DFG, an indexing operation takes edge attributes as input to read/write the tensor pointed by them, and then produces tensor that is used by other operations. The values of these edge attributes determine memory access and computation patterns of the current and subsequent operations. Therefore, these edge attributes are the key factors that influence computation performance, we call them as *indexing edge attributes*, or *indexing attributes* for short.

WiseGraph identifies these indexing edge attributes by analyzing indexing operations in a GNN model. In this example, three indexing edge attributes, i.e., *src-id*, *edge-type*, and *dst-id*, will be used in the following graph partition.

4.2 Applying Restrictions on Edge Attributes

With the indexing edge attributes identified from GNN model, the second step is to generate gTasks and graph partition plans. The key idea is to partition edges with the same/similar values of indexing edge attributes into a gTask. As there are multiple indexing edge attributes to be considered for partition, WiseGraph uses *graph partition table* to organize them in unified way, and applies restrictions on them to generate graph partition plans, where edges partitioned to the same gTask satisfy all the restrictions.

Graph partition table. The edge attributes are organized in the *graph partition table* in a unified way. Shown in Figure 6, the rows of the graph partition table are the edge attributes, which are categorized into three types: Indexing edge attributes are used in indexing operations in the model; Inherent attributes, such as degree of vertices, are not used for indexing but still important for performance; Unused attributes are not used in indexing operations, e.g. type of source and destination vertices, therefore they are not considered in graph partition. The columns of the graph partition table indicate the location of each edge attribute, which can be on the edge itself, or on the destination or source vertex.



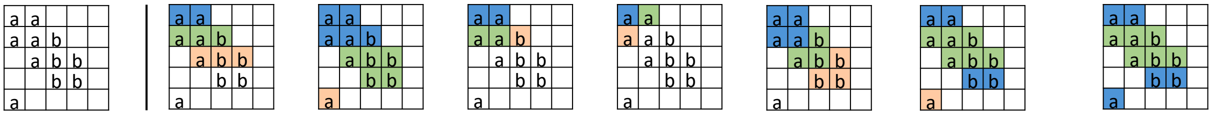
	Src	Dst	Edge
ID	🟧	🟧 1	🟩 32
Type	▨	▨	🟧 min
Degree	🟩	🟩	N/A
...	▨	▨	▨

🟧 Indexing edge attributes
🟩 Inherent attributes
▨ Unused attributes

Figure 6. Graph partition table.

Restrictions. A restriction is to limit **the number of unique values** in an edge attribute for edges within a gTask. For each entry in the graph partition table in Figure 6, there can be three types of restrictions. The first is to restrict the number of unique values with **a specific value**. For example, a restriction of $uniq(dst\text{-}id) = 1$ means that there can only be one unique destination vertex connected by the edges in a gTask; $uniq(edge\text{-}id) = 32$ means there are 32 edges in a gTask. The second is to restrict the number of unique values to be **minimum**. For example, $uniq(src\text{-}id) = min$ means that after satisfying other restrictions, gTasks with a smaller number of unique *src-id* are preferred. The third are the entries with no restriction, a gTask can have unlimited number of unique values in these edge attributes.

As shown in Figure 5(c), applying different restrictions to indexing edge attributes can result in different graph partition plans. Each partition plan generates several gTasks based on graph data, which will be processed in parallel by GPU execution units. Considering the substantial scale of the graph data, we use a greedy method for graph partition to process graph data in a light-weight manner. It can generate the partition plan with computation complexity of $O(E)$ each time. We first sort the edges of the graph according to edge attributes involved in the restrictions. Then we scan these edges in order. If a restriction condition is satisfied after including the current edge, we add it to the current gTask's graph data. If any restrictions are not satisfied after adding the current edge, we stop the graph partition for the current gTask and start a new gTask. Note that the restrictions may not be fully satisfied. But as WiseGraph is adopting a greedy method that keeps adding edge until the restrictions cannot



(a) Ajancency matrix of graph data
(b) uniq(dst-id)=1 (vertex-centric)
(c) uniq(dst-id)=2
(d) uniq(dst-id)=1 &uniq(type)=1
(e) uniq(edge-id)=1 (edge-centric)
(f) uniq(dst-id)=2 &uniq(src-id)=2
(g) uniq(dst-degree)=1
(h) uniq(dst-id)=3 &uniq(dst-degree)=min

Figure 7. The adjacency matrix of graph data and graph partition plans generated from different restrictions; For each graph partition plan, the first three *g*Tasks are marked with colors.

be satisfied, the tasks can still be generated. So there may be outlier *g*Tasks, but they are properly handled as in §6.

4.3 Generated Graph Partition Plans

General space of graph partitions. Generated graph partition plans can cover existing and new graph partitions. For example, previous graph-centric approaches, such as vertex-centric and edge-centric, can be described with graph partition table restrictions in Figure 7(b) and (e). 2D-partition, which is commonly used in graph computing by segmenting the adjacency matrix into 2D grids, is represented in Figure 7(f). Other graph partition plans commonly used can also be generated by applying restrictions to the ID of vertices and edges. With the indexing attributes imported from model, the graph partition table can find graph partition plans that are missing from previous work. For example, Figure 7(d) not only restricts the connected destination vertex ID, but also selects edges of the same type into a *g*Task; Figure 7(g) generates *g*Task with all edges connected to the destination vertices with the same degree. The restriction of *min* applied to edge attributes can also lead to new plans. The example in Figure 7(h) generates *g*Tasks whose edges are connected to K unique destination vertices while minimizing the unique degrees of destination vertex. This enables WiseGraph to generate *g*Task that pads the destination vertices with different degrees for high parallelism.

Comparison with other graph partition methods. Compared with WiseGraph, the other graph partitions methods target at clustering the vertices and minimize the edge-cut among different vertex partitions. The representative includes Metis [23] and Rabbit [1]. With the consideration of redundant computation caused by shared neighbors, Betty [50] further prioritizes partitioning vertices with more shared neighbors into the same cluster, which is achieved by redefining the cost function of edge connections while applying Metis partitioning. The output of all these graph partition methods is a reordered graph so that the vertices are clustered. However, the output of WiseGraph graph partition are *g*Tasks each containing several edges sharing same edge attributes, which are the unit tasks for GNN execution. Metis-style and WiseGraph graph partition work at different levels and can be combined: we can first use Metis-style work to produce the reordered graph with better locality, and then apply WiseGraph graph partition on it to generate *g*Tasks for GNN optimization and execution.

5 Generating Operation Partition Plans

After generating graph partition plans for *g*Tasks, WiseGraph generates candidate operation partition plans for *g*Tasks. Based on graph data patterns revealed by the *g*Task, WiseGraph optimizes three key steps in operation partition, which are DFG transformation, kernel generation, and operation placement.

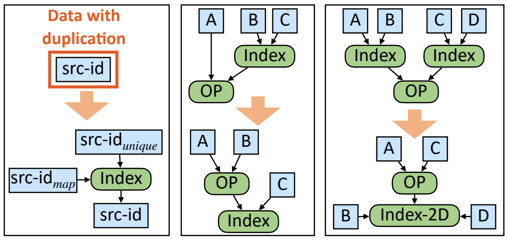
5.1 *g*Task-Level Graph Data Patterns

In graph partition plans, each *g*Task reveals certain data patterns, which are referred to as *g*Task-level data patterns. The data patterns are centered around the *restriction of unique values* for edge attributes, which can be classified into the following three types.

Duplicated data. In a *g*Task, by comparing the number of unique values with the total number of edges, WiseGraph can identify whether there is duplicated data in an edge attribute. For example, given a *g*Task with $uniq(src\text{-}id)$ less than the number of containing edges, there will be duplicated *src-id* across different edges. These duplicated values can result in memory accesses to the same address and computation performed on the same tensors.

Batched data. Batched data reflects data pattern for *g*Tasks from two aspects. The first is about the number of unique values for each edge attribute, which determines the amount of data involved and the parallelism of *g*Task's computation. The second is the ratio comparing different edge attributes' number of unique values. It determines the data reuse in the computation of *g*Task. For example, for *g*Task generated with $uniq(src\text{-}id) = 32$, the data indexed by *src-id* can form a batch of size 32, which can be processed in parallel.

Changing data volume. For a *g*Task, the number of unique indexing attributes used for data retrieval and for writing can be different, so are the volume of input and output tensors, leading to changing data volume. In multi-device training, where communication is a main bottleneck, data volume in communication can significantly impact performance. Taking *g*Task generated with $uniq(src\text{-}id) = 32\&uniq(dst\text{-}id) =$



(a) Unique value extraxtion (b) Indexing swapping

Figure 8. DFG transformation rules for duplicated data.

2 as an example, the process leads to a reduction in the number of vertices for each *g*Task through computation, indicating to conduct communication subsequent to computation for less communication workload.

5.2 DFG Transformation

When WiseGraph reveals the *g*Task-level pattern of duplicated data, it indicates which attribute values within a *g*Task are shared among edges, which can be used to share and reduce computation workload. To achieve this, WiseGraph employs two DFG transformation rules to generate new DFGs that leverage the duplicated data for computation workload reduction. Our method identifies and executes the necessary computations within a *g*Task while ensuring equivalent results.

Transformation 1: Unique value extraction. From the duplicated data, it extracts the unique values by inserting an additional indexing operation into DFG; A mapping from unique values to duplicated data is used to perform the index operation. Figure 8(a) takes *src-id* as the duplicated data for example. To extract unique values, WiseGraph decompose *src-id* into its unique values ($src\text{-}id_{unique}$) and a tensor ($src\text{-}id_{map}$) storing their mapping relation. An indexing operation is inserted to retrieve the original data: $src\text{-}id = src\text{-}id_{unique}[src\text{-}id_{map}]$. This transformation introduces the unique values from duplicated data into DFG.

Transformation 2: Indexing swapping. It exchanges the execution order of the indexing operation and its subsequent operation. As shown in Figure 8(b), by applying *indexing swapping*, the neural operation (*OP*) that was executed on the indexing operation's output is now performed on indexing operation's input (*B*). By swapping execution order, WiseGraph can perform computation on the extracted unique data. The equivalence is guaranteed as long as the neural operation is invariant to the tensor's dimension used by the indexing operation. For operations with multiple indexing results as inputs, WiseGraph can still perform *indexing swapping* by merging multiple indexing operations into a multi-dimension one (*Index-2D* in this example). In the example from Figure 8(b), the original computation is A[B] ⊗ C[D]. After transformation, the computation is equivalent to (A ⊗ C)[B, D], which simultaneously uses both B and D for indexing the computation result of OP. The multi-dimensional indexing makes *indexing swapping* applicable to broader cases.

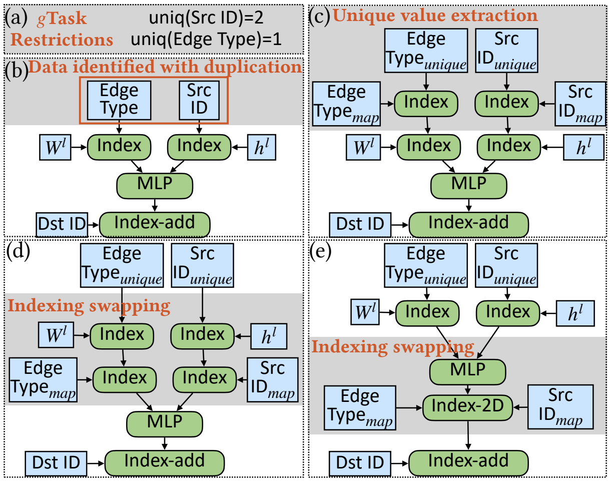


Figure 9. Steps of DFG transformations for RGCN.

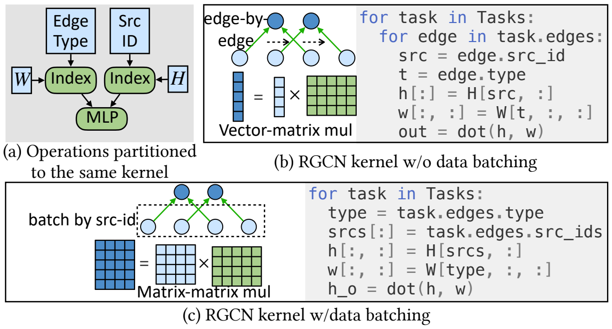
WiseGraph scans the operations on the DFG in topological order and applies transformation rules wherever possible to obtain a new equivalent DFG. By calculating and comparing the workload of transformed DFGs, WiseGraph can choose the one with the least workload as the DFG for execution.

Example. Figure 9 shows how these transformation rules take effect on the DFG of RGCN. With *g*Task and its restrictions in (a), WiseGraph can infer that *src-id* and *edge-type* have duplicated data (b). So WiseGraph applies *unique value extraction* to find out unique values from them and explicitly represent them on the DFG (c). WiseGraph then applies multiple steps of *indexing swapping* to swap execution order between the indexing operation and its subsequent operations ((d) and (e)). After transformations, the MLP operation in the new DFG is directly performed on the unique source vertex embeddings and weight parameters and then produces equivalent results by performing a 2D-indexing operation. A large amount of computation workload for MLP is shared among edges.

5.3 Kernel Generation

The second step of operation partition is to partition operations on DFG into several kernels. If there is only one operation mapped to a kernel, we can directly use existing implementations from DNN frameworks. However, if there are multiple operations partitioned into a kernel, we need to generate efficient GPU kernel code as no off-the-shelf implementation exists.

Existing work [18, 28, 47] can generate code to process multiple operations in a GPU kernel for GNN. However, it is infeasible to apply these approaches in operation partition



(a) Operations partitioned to the same kernel

```
for task in Tasks:
  for edge in task.edges:
    src = edge.src_id
    t = edge.type
    h[:] = H[src, :]
    w[:, :] = W[t, :, :]
    out = dot(h, w)
```

(b) RGCN kernel w/o data batching

```
for task in Tasks:
  type = task.edges.type
  srcs[:] = task.edges.src_ids
  h[:, :] = H[srcs, :]
  w[:, :] = W[type, :, :]
  h_o = dot(h, w)
```

(c) RGCN kernel w/data batching

Figure 10. Kernel generation for RGCN operations.

due to two challenges. First, the need to partition various operations into a kernel poses a significant obstacle. Existing code generation for GNN is limited to handling fixed workflows, such as Source-Edge-Aggregation workflow [28, 47], thereby lacking the flexibility required for arbitrary operation partition. Second, achieving high parallelism is essential for the efficient execution of neural operations. Nevertheless, traditional GNN code generation produces kernels that process workload on an **edge-by-edge** basis, resulting in low parallelism and performance.

To address this, WiseGraph adopts composable micro-kernels [38] guided by the pattern of batched data. To generate kernels that can handle various operations, WiseGraph prepares multiple micro-kernels for data loading and computation, with each micro-kernel representing a specific operation. By composing these micro-kernels, we can generate a GPU kernel with operations partitioned in.

To achieve high parallelism, WiseGraph leverages the pattern of batched data to select the implementation of micro-kernels. These micro-kernels process data in parallel with multiple GPU threads. Both the number of threads used and the amount of data processed by each thread are determined through batched data. By contrast, a micro-kernel without the pattern of batched data can only process computation edge-by-edge, which prevents efficient data access and data reuse for computational workloads.

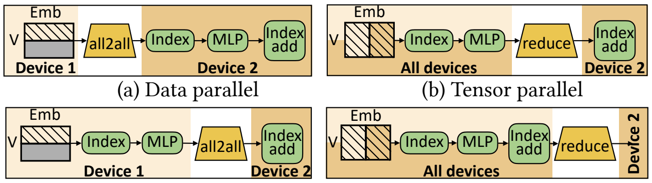
Example. Figure 10 shows an example of generating kernel for RGCN operations (a). Shown in (b), without the pattern of batched data, the kernel processes the gTask workload edge-by-edge. For each edge, it loads the weight matrix w as well as the edge's source vertex embedding vector and performs vector-matrix multiplication on them. In (c), with the data pattern that edges are batched with 4 unique *src-id* in a gTask, WiseGraph generates a GPU kernel that loads 4 vertex embeddings simultaneously. After that, the kernel performs matrix-matrix multiplication on them and updates the destination vertices according to *dst-id*. The performance improvement comes from two aspects. The first is the data reuse: the loaded data of weight matrix is reused for 4 times compared to only one time in non-batched implementation. Second, more efficient hardware resources, i.e., tensor-core, are used for matrix multiplication.

5.4 Operation Placement

Multi-device training. In multi-device training for large graphs, the vertex embeddings should be partitioned to multiple devices due to limited per-device memory capacity. For every layer of GNN, each device stores part of source vertex embeddings and is responsible for computing part of destination vertex embeddings. Shown in Figure 11(a) and (b), there are two ways to partition embeddings into devices, either at vertex dimension (V) or embedding dimension (*Emb*), which corresponds to data parallel [3, 32, 42] and tensor parallel [12] training. For vertex dimension partition and data parallel training, the embeddings of different vertices are partitioned to different devices. For embedding dimension partition and tensor parallel training, each device holds all vertices' parts of the embedding. To perform operations on partitioned embeddings, the distributed implementation is needed, which involves not only computation but also communication to prepare necessary data. For data parallel execution in (a), the indexing operation requires *all-to-all* communication because some vertices are not stored locally. On the other hand, for tensor parallel, the neural operation requires additional communication. For the example in (b), MLP needs the communication operation of *reduce-scatter* to generate complete results. As communication bandwidth is lower than computation throughput, communication can be a training bottleneck. Therefore, reducing communication workload is important for multi-device GNN training and is considered by WiseGraph in operation partition.

Operation placement. An important property is observed from these communication operations: they either move or reduce data, thus the execution order of communication and computation can be switched while producing equivalent results. For example, instead of transmitting data to local devices and performing computations on them, we can perform computations on the data before transmitting data to local devices. By changing the execution order, operation placement is shifted from local devices to remote devices and the communication workload changes from the operation's input to output. Due to the changing data volume, the amount of communication data is also changed. WiseGraph determines the placement of operations by considering the changing data volume of each operation on the DFG. The calculation considers the changing data volume at both vertex and embedding dimensions.

Example. Figure 11(c) and (d) show the example to change operation placement for data and tensor parallel. For data parallel in (c), WiseGraph takes into account the changing data volume and, if the volume decreases at the embedding dimension, it will place the MLP on device 1 by performing



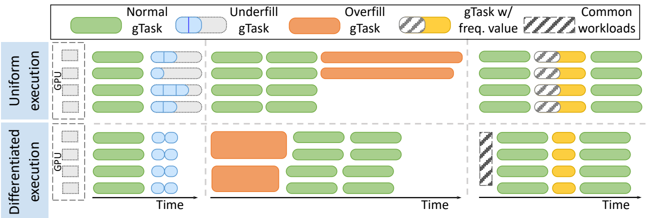
(a) Data parallel

(b) Tensor parallel

(c) Data parallel with new operation placement

(d) Tensor parallel with new operation placement

Figure 11. Parallel strategies and operation placement of RGCN.



(a) Underfill gTask

(b) Overfill gTask

(c) gTask with frequent value

Figure 12. Differentiated scheduling of outlier *g*Tasks.

all-to-all on the MLP output. For tensor parallel in (d), WiseGraph will place index-add on all devices given that data volume decreases at vertex dimension with it.

Comparison with other parallelization approaches. Numerous parallelization techniques and their combinations have been proposed for the training DNNs [20, 33, 35, 56], where tensors and computations are both dense. The determination of an optimal strategy, encompassing tensor partitioning methods, involves either solving [35] or exploring [56] potential solutions. In contrast, the efficiency of multi-device GNN training is significantly influenced by the characteristics of input graph data, necessitating a strategy that also accounts for these data attributes. DistDGL [55] introduces a data parallel approach to graph data processing, employing *all-to-all* communication to aggregate necessary vertex features across workers. Meanwhile, PipeGCN [41] implements pipeline parallelism to reduce communication overhead, and P^3 [12] leverages tensor parallelism for the input layer before switching to data parallelism for subsequent layers for better communication efficiency. Although these methods are capable of processing graph data during execution, they fail to optimize strategy setting based on graph data characteristics. Consequently, this leads to considerable redundancy and variable speed enhancements across different graph structures.

6 Joint Optimization

After generating graph partition plans (§4) and operation partition plans (§5), we need to jointly optimize with them for efficient execution. The key problem is as follows: given a graph partition plan and a set of operation partition plans, how to identify an execution plan that minimizes the execution time. Due to graph irregularity, applying the same operation partition plan to generated *g*Tasks can result in a mismatched and inefficient execution. On the other hand, for a large graph, it is infeasible to try different operation partition plans for every *g*Task. Thus, for joint optimization, WiseGraph should find a light-weight approach to process large graphs while tackling their irregularity.

We observe that, after partitioning graph data into a number of *g*Tasks, most of *g*Tasks are similar (regular *g*Task), with some outlier *g*Tasks due to the irregularity of graphs. This is due to the power-law distribution of graph data [7, 10, 59]: most of the vertices have a moderate degree, and the *g*Tasks generated from them have similar number of edges and unique edge attribute values. So instead of selecting an operation partition plan for every *g*Task, we can identify *regular* and *outlier* *g*Tasks and select one for each type separately, which largely reduces the complexity of joint optimization.

6.1 Outlier *g*Task Identification

To identify the outliers from generated *g*Tasks, we categorize the outlier *g*Tasks into the following three types.

Underfill *g*Task: Insufficient number for restricted edge attributes. The restriction from *g*Task's graph partition table has a preset number of unique values. However, it is possible that there is not enough data to reach the restriction. For example, *g*Task restricting *dst-id=1&edge-id=K* can have underfill *g*Tasks containing a destination vertex with less than K adjacent vertices. A significant amount of idle time and resource underutilization may be incurred due to these underfill *g*Tasks.

Overfill *g*Task: Extremely large number of unrestricted edge attributes. For an unrestricted indexing attribute, a *g*Task can have a much larger number of edges than other *g*Tasks and becomes an overfill *g*Task. In RGCN example, restriction *src=K&edge-type=1* can result in overfill *g*Tasks with high-degree source vertices grouped into the same *g*Task. Overfill *g*Task can lead to severe load imbalance.

***g*Task with frequent values: Edge attributes frequently appear in many *g*Tasks.** Some values of indexing edge attributes can frequently appear in a large number of *g*Tasks. Taking restriction *dst-id=1&edge-num=K* as an example, the *dst-id* of a high-degree vertex will appear in multiple *g*Tasks, as each of *g*Tasks only holds K edges connected to this vertex. There can be data races for *g*Tasks with frequent values, which further results in low memory efficiency.

6.2 Differentiated Outlier *g*Task Scheduling

After identifying the outlier *g*Tasks, WiseGraph reschedules computation resource and execution priority for them to address the resulting resource inefficiency and workload imbalance.

WiseGraph changes computation resource assigned to the outlier *g*Tasks according to their types. Shown in Figure 12(a), underfill *g*Tasks have a redundant workload due to the assumption of batched data. WiseGraph further breaks these *g*Tasks into individual edges and performs edge-wise computation on them to eliminate the redundancy. For the overfill *g*Tasks in (b), WiseGraph allocates more computation resource for them by launching a separate GPU kernel that increases thread blocks and shared memory size. For *g*Tasks with frequent values in (c), WiseGraph extracts common workloads shared by them and computes them in advance. After that, the execution of these *g*Tasks only needs to fetch the pre-computed data for these frequent values.

Besides adjusting computation resource, WiseGraph also reschedules the execution order for the outlier *g*Tasks, which is based on the amount of workload. Imbalanced workload results in long-tail effect: shown in (b), the overfill *g*Tasks still reside on execution units while other execution units are idle. Increasing the priority of execution for overfill *g*Tasks can lead to a more balanced workload. In reserve, shown in (a), running edge-wise computation for underfill *g*Tasks with lower priority can also balance workload.

6.3 Strategy Selection

WiseGraph searches operation partition plans for regular and outlier *g*Tasks respectively, measures execution time, and selects one with the least execution time. To reduce the tuning overhead, WiseGraph applies the following methods. The first is pruning. There can be many inefficient execution plans not worth trying in random search. WiseGraph prunes these execution plans using a cost model. The cost model comprises three essential components: computation workload, memory access volume, and parallelism. The computation workload is determined by analyzing floating-point operations (FLOP) based on the operation type and input data size. Memory access amount is calculated by considering the shape of input tensors. Parallelism is estimated in relation to the number of *g*Task. Inefficient execution plans (e.g., a large amount of workload or low parallelism) will be ruled out without testing. The second is caching. WiseGraph caches parameters and generated kernel for the same configuration to avoid repeated compilation and testing of GPU kernels. The third is efficient graph data processing. After setting the restrictions for *g*Tasks, the organization of graph data should be changed accordingly. CPU processing will become the bottleneck for large graph data. WiseGraph processes graph data in parallel using GPU, which can largely reduce overhead.

Working with sampled graph training. WiseGraph mainly targets at full graph training. However, besides training with a full graph, sampled graph training is also a promising training method for GNN, which generates a subgraph from graph data at each iteration. Compared with the full graph training, the graph data changes at every iteration, making costly parameter tuning on graph data impractical. To make WiseGraph's optimizations appliable to sampled graph training, we observe that, given graph data and a sampling method, the sampled subgraphs share a similar pattern and adapt to the same set of parameters [14, 21, 53]. Therefore, WiseGraph first tunes the parameters on multiple sampled subgraphs, and then applies the optimal graph and operation partition plan to all other sampled graphs without tuning. The graph processing according to the graph partition plan is performed together with graph sampling, which is performed asynchronously on CPU. WiseGraph is unable to tackle the situation where graph structure changes dramatically at every iteration. However, current training methods on sampled/evolving graphs have similar graphs among iterations [5, 6, 21, 60].

Table 1. Evaluated graph datasets. **Dim.**: dimension of input vertex embedding; **#Class**: number of classification results

	Dataset	#Vertices	#Edges	Dim.	#Class
Single GPU	**Arxiv (AR)**	169K	2.3M	128	40
	Products (PR)	2.4M	123M	100	47
	Reddit (RE)	233K	114M	602	41
	Papers-sample (PA-S)	1.2M	1.5M	128	172
	FriendSter-sample (FS-S)	1.4M	1.6M	384	64
Multi	**Papers (PA)**	111M	1.6B	128	172
	FriendSter (FS)	66M	3.6B	384	64

7 Evaluation

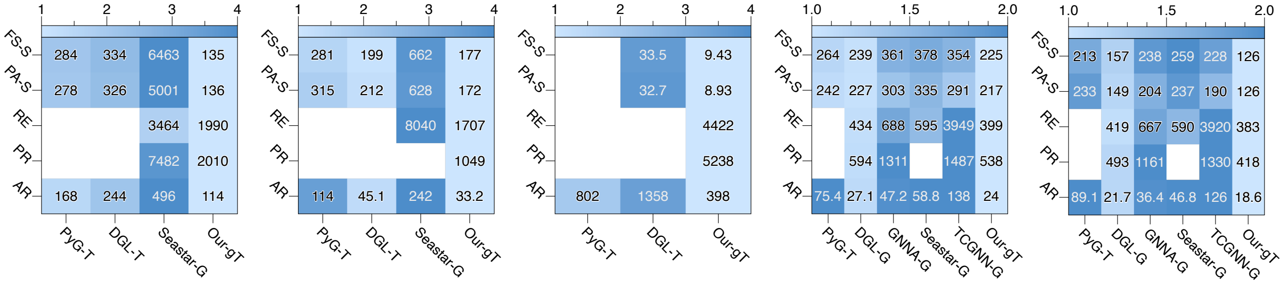
In this section we aim to evaluate the following points:

- Can WiseGraph improve the end-to-end training performance of GNN while preserving model accuracy?
- How do WiseGraph's designs, i.e., graph partition, operation partition, and joint optimization, contribute to performance improvement?
- Can WiseGraph generate *g*Tasks and optimize execution plans in acceptable run time?
- What are the best *g*Tasks found for each GNN model?

7.1 Experimental Setup

Dataset. We evaluate the efficacy of WiseGraph on seven datasets shown in Table 1 collected from GNN benchmark Open Graph Benchmark (OGB) [17] and other sources [25, 49]. Five graph datasets with moderate size are for single-GPU training. Among them, two datasets (PA-S and FS-S) are sampled from large graphs (PA and FS) using a seed vertex size of 1000 and a fan-out of 20-15-10. Two larger datasets (PA and FS) with billions of edges are for multi-GPU training.

Models. We use five models in the evaluation: RGCN [34], GAT [40], SAGE-LSTM [14, 16], SAGE [14], and GCN [24]. Each model has three layers with hidden dimension at 256, which is a typical setting [4, 17, 52, 57] to achieve high accuracy. For multi-GPU full graph training, the hidden dimension is set as 32 to avoid memory issues [3, 44]. RGCN,



(a) RGCN (b) GAT (c) SAGE-LSTM (d) SAGE (e) GCN

Figure 13. Comparison of per-iteration execution time (ms) with other frameworks on different models. X-axis are different frameworks and their partition method (Tensor-centric (T), Graph-centric (G), and *g*Task-based (gT)); Y-axes are different datasets; Lower numbers (lighter colors) are better. White blocks represent the out-of-memory error.

GAT, and SAGE-LSTM perform complex neural computations (MLP, Attention, and LSTM), while SAGE and GCN simply perform addition as the neural computation.

Hardware and environment. We evaluate WiseGraph on a server with four NVIDIA Tesla A100-PCIe GPUs and two AMD EPYC 7742 64-Core processors. All experiments run with CUDA 11.7, NCCL [30] v2.11.4, PyTorch [31] 2.0.1.

Experiment configuration. For single-GPU performance test, all data (graph data, vertex embeddings, and weight parameters) are stored on GPU memory. We measure the time of computation for each training epoch. For multi-GPU performance test, vertex embeddings are stored on different devices and managed by GNN systems. The measured execution time includes computation and communication time using four GPUs. The execution time is measured by averaging 100 iterations. For the accuracy test, we set the same accuracy-related hyper-parameters for different systems.

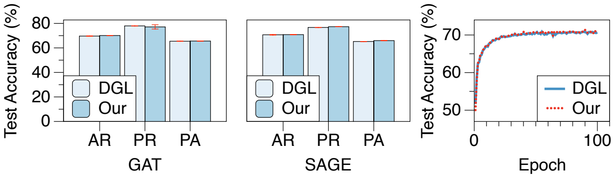
Baselines. We compare WiseGraph with a wide range of GNN systems. For single-GPU training, we compare WiseGraph with PyG@2.5 [11], DGL@1.0.0 [42], GNN-Advisor (GNNA) [43], SeaStar [47], and TC-GNN [45]. Among them, DGL uses both graph-centric and tensor-centric approaches depending on the type of GNN model; PyG is tensor-centric while the others are graph-centric. For multi-GPU training, we compare to DGL [42], ROC [22], DGCL [3], MGG [44] on full graphs, and with an emulated version of P^3 [12] which targets at sampled graph training. P^3 is emulated by reproducing the hybrid parallelism as mentioned in the paper, which includes the tensor parallel for the first layer and data parallel for the other layers.

7.2 Overall Performance

Single-GPU training. Figure 13 shows the epoch time for single GPU training. WiseGraph can achieve 2.04× improvement over state-of-the-art systems on average, which is 2.64× for models with complex neural operations (RGCN, GAT, SAGE-LSTM) and 1.13× for the other simple models that are well optimized. And WiseGraph is more memory-efficient and can run complex models on large graphs. From Figure 13(a) and (b), by comparing two tensor-centric approaches (PyG and DGL) and graph-centric approach (SeaStar), we find that the graph-centric approach is more memory-efficient, while tensor-centric suffers more from out-of-memory (OOM) on graph datasets with a large number of edges. However, tensor-centric is faster than graph-centric due to better neural computation efficiency, but it still introduces redundancy. WiseGraph co-considers graph data and neural operations with *g*Task, which achieves high parallelism while eliminating redundancy. Figure 13(d) and (e) show performance for GNN models using element-wise operations as neural computation. The graph-centric approaches are more effective than tensor-centric ones on these models, as the neural efficiency is less important and data movement is the main bottleneck. For these models, WiseGraph partitions *g*Task with a balanced workload when performing optimizations for kernel generation, so it still achieves a speedup of 1.13×.

Table 2. Multi-GPU training epoch time in seconds

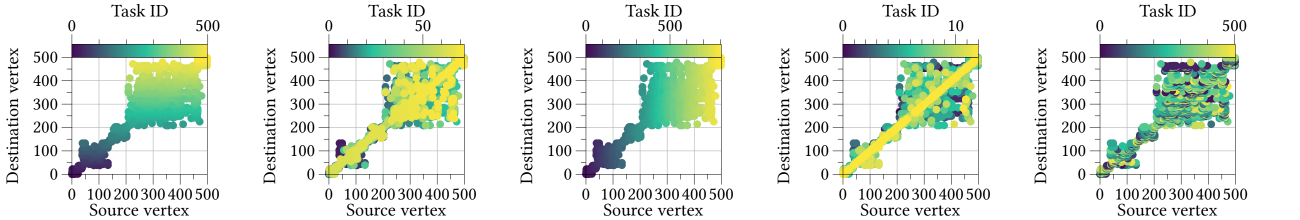
Datasets	DGL	ROC	DGCL	P^3	WiseGraph
PA	23.12	14.95	26.31	N/A	5.98
FS	16.068	12.56	21.63	N/A	6.18
PA-S	30.95	N/A	N/A	37.57	15.34
FS-S	7.39	N/A	N/A	2.80	1.71



(a) Test accuracy on different datasets and models (b) Accuracy curve on AR

Figure 14. Accuracy comparison between DGL and WiseGraph

Multi-GPU training. Table 2 shows the epoch time for multi-GPU training on four GPUs connected via PCIe-4.0.



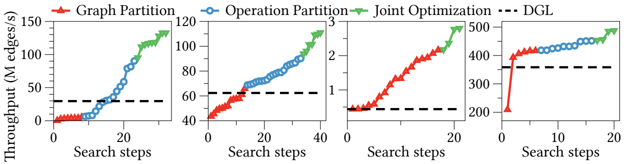
(a) Vertex-centric partition (b) gTask partition for RGCN (c) gTask partition for GAT (d) gTask partition for LSTM (e) gTask partition for SAGE&GCN

Figure 15. Vertex-centric partition and different *g*Task-based graph partitions adaptive to GNN models.

To perform full-graph training on PA and FS, using WiseGraph can achieve 2.27× speedup over the best of other systems. For sampled-graph training on PA-S and FS-S, an epoch will train all vertices in the training set. WiseGraph can always achieve the best performance and shows 1.83× speedup. While the hybrid parallelism selected by P^3 sometimes shows lower performance than the naive data parallel used by DGL. We also compare WiseGraph with MGG [44] for full graph inference, as MGG currently only has forward implementation. Inferring labels for all vertices on PA takes 8.71 seconds with WiseGraph while 25.24 seconds with MGG. The speedup of 2.90× comes from WiseGraph’s operation placement strategy and more efficient computation.

Accuracy test. Figure 14 shows the test accuracy that DGL and WiseGraph can achieve. For a fair comparison, WiseGraph follow the same hyper-parameters from DGL. From (a), WiseGraph and DGL achieve similar accuracy on both models and all OGB datasets, with an accuracy difference within 1%. The test accuracy also matches that on OGB leaderboard [17]. (b) shows a similar trend of the accuracy curves on AR with SAGE in 100 epochs.

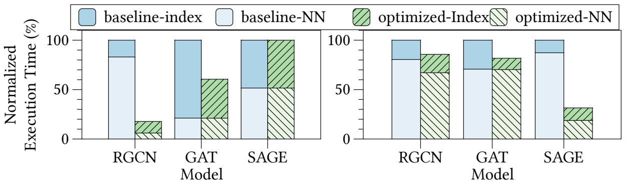
7.3 Optimization Analysis



(a) RGCN (b) GAT (c) LSTM (d) GCN

Figure 16. Throughput with search stages and steps.

Performance improvement breakdown. Figure 16 shows how throughput (processed edges per second) improves with WiseGraph’s search stages and steps. Given a GNN model and graph data (tested on AR), the three stages are graph data partition, operation partition, and joint optimization, each with several tuning steps. The initial point for each model is an edge-centric graph partition with the original user-defined DFG and naive kernels provided by PyTorch and WiseGraph. The black line is the throughput achieved by DGL. Graph partition stage can largely improve performance for SAGE-LSTM (c) and GCN (d), which comes from the balanced and batched workload in each *g*Task. For RGCN (a) and GAT (b), as GPU kernels are still inefficient, the improvement is not significant. The operation partition stage has little contribution for GCN and SAGE-LSTM, as their neural operation is either too simple to be optimized (Addition in GCN) or too complex that PyTorch’s implementation is already well-optimized (LSTM). However, for other models, operation partition can largely improve throughput, especially for RGCN, the improvement can reach up to 15× compared with graph partition only. Finally, with joint optimization, all models’ performance can be further improved.

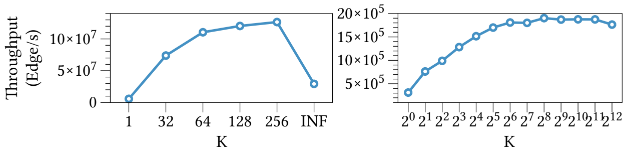


(a) AR (b) PA-S

Figure 17. The normalized execution time without and with the awareness of data duplication in DFG transformations.

Figure 17 shows the execution time breakdown of GNN models. The execution time is classified by indexing operations and neural operations. As different operations are fused into one kernel, some execution time breakdown is estimated with workload and operation efficiency. The base implementation (blue) uses the original DFG and the optimized version uses the transformed DFG with the same set of kernels provided by WiseGraph. The elimination of duplicated workloads is dependent on graph data and model architecture. For RGCN on AR, the neural computation can be reduced by 92.7% as WiseGraph finds many source vertices sharing the same type. For SAGE, though there is no duplication on AR, WiseGraph can address duplication of 78.5% in PA-S, which has less number of destination vertices than source vertices.

Figure 18 shows how different data batching (different K) influences the efficiency of generated kernels for one RGCN and SAGE-LSTM layer. When K is set to 1, the throughput is extremely low, as there is only one edge in a *g*Task and data is not batched. By increasing K, the generated kernel



(a) RGCN (b)SAGE-LSTM

Figure 18. The throughput with differently batched data; (a) RGCN with *g*Task generated with *uniq(src-id)=K&uniq(edge-type)=1*; (b) SAGE-LSTM with *g*Task generated with *uniq(dst-degree)=min&uniq(dst-id)=K*; X-axis shows the value of *K*.

processes batched edges in each *g*Task, and the performance is largely improved. For RGCN, if *K* continues to increase and is set to *INF*, all edges sharing the same edge type will be put into a task, which is equivalent to the tensor-centric approach. As shown in (a), data batching in a *g*Task leads to an improvement of 4.33× compared with the better one in non-batched or tensor-centric approach. In (b), batching edges in LSTM brings better parallelism and 6.10× higher throughput.

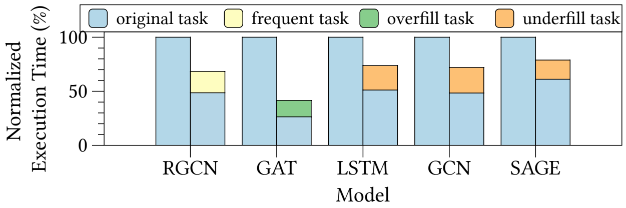
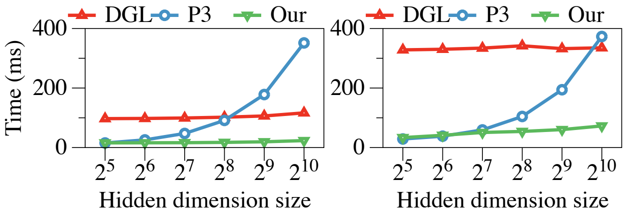


Figure 19. Differentiated execution.

Figure 19 shows the improvement after applying differentiated *g*Task execution on AR. Due to different restrictions in *g*Task generation, WiseGraph finds *g*Task with frequent value for RGCN, overfill *g*Task for GAT, and underfill *g*Task for the rest. The baseline (left bar) is the uniform execution that runs all *g*Tasks with the same searched operation partition. 52.9% of execution time is spent on outlier *g*Tasks on average. After applying differentiated execution (right bar), the execution time of outlier *g*Tasks can be reduced by 60.7% and the overall execution time is reduced by 33.1%.



(a) PA-S (b) FS-S

Figure 20. Execution time with varying hidden dimensions.

Figure 20 shows the impact of varying hidden dimension sizes on the performance of multi-device training. We evaluate the execution time for the first layer of GCN on PA-S and FS-S. The varying hidden dimension leads to different communication data volume and computation workload. By taking into account the pattern of changing data volume for operation placement, WiseGraph consistently achieves the shortest execution time. In contrast, the static parallel strategy employed by DGL and P^3 turns to be inefficient for certain hidden dimension sizes.

Table 3. The processing time for training SAGE in WiseGraph

Processing Time (Second)	PA	AR
Environment Setup	1.2	
Train Initialization	8.3	7.2
Disk to DRAM	40.5	2.7
Convergence Time	18915	662
Joint Optimization	100	12

Graph partition results. Figure 15 visualizes the vertex-centric graph partition and graph partition plans found by WiseGraph on AR. A scatter means one edge, and the scatters in the same color are partitioned to the same *g*Task. (a) is vertex-centric, in which the task ID increases with the destination vertex ID. (b)-(e) show the *g*Task-based graph partition plan found by WiseGraph. For RGCN in (b), *edge-type* is an important factor used by *g*Task; For GAT in (c), WiseGraph finds that edges sharing the same source vertex should be grouped. For SAGE-LSTM in (d), destination vertex degree and edge ID are used for graph partition. The vertices with similar degrees are grouped together. For SAGE and GCN, WiseGraph finds limiting the edge number per *g*Task is beneficial to performance.

7.4 Discussion

Overhead analysis. Table 3 compares the pre-processing time for joint optimization with other necessary steps for GNN training, such as loading vertex embeddings from disk to DRAM. The joint optimization pre-processing time includes graph partition, operation partition, and tuning on various plans. The convergence time is measured by training and evaluating for 100 epochs. We can conclude that the time for joint optimization is comparable with these necessary steps and is much less than the convergence time (<2%). And WiseGraph is more light-weight than other graph partition works such as Metis, which takes over one hour to process PA. Moreover, as the joint optimization is one-shot for a model and graph data, the overhead is acceptable.

Applying WiseGraph to sampled graph training. To make WiseGraph practical in sampled graph training, we have two considerations. First, different sampled subgraphs can share the same partition plan of graph data and operation; Second, the overhead of partition can be overlapped.

Figure 21(a) demonstrates the first point. Compared to full optimization, reusing the partition strategies of graph data and operations that are searched from other sampled graphs can still achieve 91% of the performance. Figure 21(b) shows the overhead of graph sampling and partitioning. Though partition introduces certain overhead, with more CPU threads (24 out of the total 128), the overhead is less than the epoch time of training and can be fully overlapped. Additionally, apart from graph sampling, only one CPU thread is utilized to initiate GPU kernels. As a result, CPU resource is underutilized, presenting the opportunity to allocate it and mitigate graph partitioning overhead.

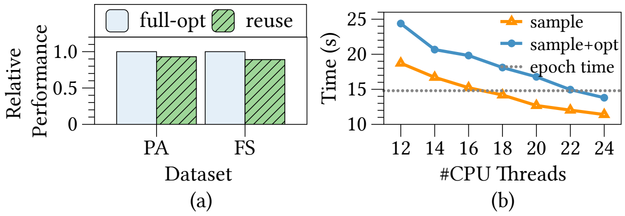


Figure 21. Using WiseGraph for sampled graph training.

8 Related Work

Graph data partition. Partitioning graph data is proved to be effective for locality [2, 54, 59] and balanced workload [7, 29] in graph computing. In GNN, as per-edge workload is heavier, graph data partition is also important for parallelization. Tensor-centric [11, 26, 28] approach uses tensors to represent graph data as a whole. Vertex-centric [42–45, 47] is a commonly used graph-centric approach that partitions graphs by vertices. Other graph partition methods for GNN, such as edge-centric [37], neighbor grouping [19], 2D partition [18], and clustering [27], improve workload balance and locality, and they are solely based on the graph structure. μGrapher adaptively applies vertex-centric and edge-centric to GNN operations, but its optimization space is limited with two strategies.

Optimizations for GNN operations. DGL [42] and Rubik [9] work on DFG for better execution order; Graphiler [48] sets fusion rules for DFG; FeatGraph [18] generates GNN kernels using TVM [8]. P^3 [12] is the tensor parallel for GNN training, which can reduce communication workload in multi-device training. It statically set the parallel strategy, sometimes leading to more communication workloads.

Joint optimization for graph data and operations. GNNAdvisor [43] explores the data pattern of the input graph structure for optimization while utilizing the model information at the same time. However, it explores coarse-grained data patterns from the entire graph and only extracts simple information such as the embedding length from models.

9 Conclusion

This paper presents WiseGraph, a GNN training framework exploring the joint optimization space of graph data and GNN operation partitions, which includes existing GNN workload partition strategies as special cases and reveals new optimization with gTask. Evaluation on five typical GNN models shows that WiseGraph outperforms existing GNN frameworks by 2.04× and 2.22× for single and multiple GPU training.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Dr. Laurent Bindschaedler, for their valuable suggestions. This work is supported by National Key R&D Program of China under Grant 2023YFB3001501, NSFC for Distinguished Young Scholar (62225206), and National Natural Science Foundation of China (U20A20226, 62302251). This work is supported by the 2023 sabbatical year research grant of the University of Seoul. Jidong Zhai is the corresponding author of this paper (zhaijidong@tsinghua.edu.cn).

References

[1] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31, 2016.

[2] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 235–248, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: An efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 130–144, New York, NY, USA, 2021. Association for Computing Machinery.

[4] Yukuo Cen, Zhenyu Hou, Yan Wang, Qibin Chen, and Jie Tang. Cogdl: An extensive research toolkit for deep learning on graphs, 2020.

[5] Jianfei Chen and Jun Zhu. Stochastic training of graph convolutional networks, 2018.

[6] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling, 2018.

[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018. USENIX Association.

[9] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, et al. Rubik: A hierarchical architecture for efficient graph neural network training. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[10] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262, aug 1999.

[11] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.

[12] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 551–568, 2021.

[13] Thomas Gaudelet, Ben Day, Arian R. Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy B. R. Hayter, Richard Vickers, Charles Roberts, Jian Tang, David Roblin, Tom L. Blundell, Michael M. Bronstein, and Jake P. Taylor-King. Utilising graph machine learning within drug discovery and development, 2020.

[14] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.

[15] Simon Haykin. *Neural networks: a comprehensive foundation.* Prentice Hall PTR, 1994.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. 2020.

[18] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. aug 2020.

[19] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. Understanding and bridging the gaps in current gnn performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 119–132, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[21] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 311–326, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, March 2-4, 2020*, pages 187–198, Austin, TX, USA, 2020. mlsys.org.

[23] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[24] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17, Palais des Congrès Neptune, Toulon, France, 2017.

[25] Srijan Kumar, William L Hamilton, Jure Leskovec, and Dan Jurafsky. Community interaction and conflict on the web. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 933–943. International World Wide Web Conferences Steering Committee, 2018.

[26] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alexander Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *CoRR*, abs/1903.12287, 2019.

[27] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 401–415, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, Renton, WA, July 2019. USENIX Association.

[29] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *ACM Sigplan Notices*, 47(8):117–128, 2012.

[30] NVIDIA. Nccl: Optimized primitives for collective multi-gpu communication, 2022.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 8026–8037. Curran Associates, Inc., 2019.

[32] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. SANCUS: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proc. VLDB Endow.*, 15(9):1937–1950, 2022.

[33] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 3505–3506. ACM, 2020.

[34] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017.

[35] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[36] Zequn Sun, Chengming Wang, Wei Hu, Muhao Chen, Jian Dai, Wei Zhang, and Yuzhong Qu. Knowledge graph alignment network with gated multi-hop neighborhood aggregation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 222–229, New York, NY, USA, 2020. AAAI Press.

[37] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter Pietzuch, and Luo Mai. Quiver: Supporting gpus for low-latency, high-throughput gnn serving with workload awareness, 2023.

[38] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[40] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, Vancouver, BC, Canada, 2018. OpenReview.net.

[41] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv preprint arXiv:2203.10428*, 2022.

[42] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[43] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus, 2020.

[44] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. MGG: Accelerating graph neural networks with Fine-Grained Intra-Kernel Communication-Computation pipelining on Multi-GPU platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 779–795, Boston, MA, July 2023. USENIX Association.

[45] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 149–164, Boston, MA, July 2023. USENIX Association.

[46] David S Wishart, Yannick D Feunang, An C Guo, Elvis J Lo, Ana Marcu, Jason R Grant, Tanvir Sajed, Daniel Johnson, Carin Li, Zinat Sayeeda, et al. Drugbank 5.0: a major update to the drugbank database for 2018. *Nucleic acids research*, 46(D1):D1074–D1082, 2018.

[47] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. Seastar: Vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 359–375, New York, NY, USA, 2021. Association for Computing Machinery.

[48] Zhiqiang Xie, Zihao Ye, Minjie Wang, Zheng Zhang, and Rui Fan. Graphiler: Acompiler for graph neural networks. 2021.

[49] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth, 2012.

[50] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. Betty: Enabling large-scale gnn training with batch-level graph partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 103–117, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. Qa-gnn: Reasoning with language models and knowledge graphs for question answering. *arXiv preprint arXiv:2104.06378*, 2021.

[52] Jiaxuan You, Rex Ying, and Jure Leskovec. Design space for graph neural networks. In *NeurIPS*, 2020.

[53] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Accurate, efficient and scalable graph embedding. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019.

[54] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[55] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.

[56] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 559–578. USENIX Association, 2022.

[57] Kaixiong Zhou, Qingquan Song, Xiao Huang, and Xia Hu. Auto-gnn: Neural architecture search of graph neural networks. *arXiv preprint arXiv:1909.03184*, 2019.

[58] Yangjie Zhou, Jingwen Leng, Yaoxu Song, Shuwen Lu, Mian Wang, Chao Li, Minyi Guo, Wenting Shen, Yong Li, Wei Lin, Xiangwen Liu, and Hanqing Wu. Ugrapher: High-performance graph operator computation via unified abstraction for graph neural networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 878–891, New York, NY, USA, 2023. Association for Computing Machinery.

[59] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.

[60] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. *Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks*. Curran Associates Inc., Red Hook, NY, USA, 2019.

A Artifact Appendix

A.1 Abstract

This artifact includes the source codes and experiments for replicating the evaluations in this paper.

A.2 Description & Requirements

A.2.1 How to access. WiseGraph is publicly available at https://github.com/xxcclong/CxGNN-Compute.

A.2.2 Hardware dependencies.

- NVIDIA Tesla A100-PCIe GPU
- Memory > 400GB
- Disk space > 100GB

A.2.3 Software dependencies. We list the most important software we used:

- CUDA 11.7
- Torch==2.0.1+cu117
- dgl==1.0.0
- torch_geometric==2.5.0

They can be installed by following the instructions.

A.2.4 Benchmarks. The dataset used are from OGB [17], the detailed information is listed in Table 1. You can access the processed data following instructions.

A.3 Set-up

Install CxGNN-Compute according to its README.

A.4 Evaluation workflow

Using scripts in `CxGNN-Compute/test/ae`, all experiments can be executed.

A.4.1 Major Claims.

- (C1): WiseGraph achieves 2× speedup over baselines, as shown in Figure 13. This is proven by experiment (E1).
- (C2): WiseGraph's optimization does not affect the accuracy of the trained model, as shown in Figure 14. This is proven by experiment (E2).
- (C3): Each of WiseGraph's data-aware optimizations provides speedup according to our ablation study (Figure 17, Figure 18, and Figure 20). This is proven by experiment (E3), which contains three sub-experiments for the optimizations of duplicated data, batched data, and changing data volume respectively.

A.4.2 Experiments. Experiment (E1) [30min]: Overall performance of WiseGraph, PyG, and DGL on different datasets and models. Follow the instructions to prepare datasets, run the experiment, and collect the results.

Experiment (E2) [30min]: Accuracy of WiseGraph and DGL. Follow the instructions to prepare datasets, run the experiment, and collect the results.

Experiment (E3) [30min]: Ablation study of WiseGraph. Follow the instructions to run the experiment and collect results. There are three sub-experiments in it. Results will be visualized using ipython.

A.5 Notes on Reusability

There are multiple ways to reuse WiseGraph for GNN training. The simplest approach is to directly reuse WiseGraph to train the five pre-supported GNN models in the code. If modifications to the model architecture are required, one can either make modifications within WiseGraph or reuse the efficient GNN operators present in WiseGraph. If there is a need to add CUDA operators for GNN, one can reuse the CPP code or Triton code in WiseGraph.

A.6 General Notes

If you encounter any problems on setting up or reproducing, please refer to troubleshooting.