AdaptGear: Accelerating GNN Training via Adaptive Subgraph-Level Kernels on GPUs

Yangjie Zhou
Shanghai Jiao Tong University
Shanghai, China
yj_zhou@sjtu.edu.cn

Yaoxu Song
Shanghai Jiao Tong University
Shanghai, China
Richard_K@sjtu.edu.cn

Jingwen Leng*
Shanghai Jiao Tong University
Shanghai Qi Zhi Institusion
Shanghai, China
leng-jw@cs.sjtu.edu.cn

Zihan Liu
Shanghai Jiao Tong University
Shanghai, China
altair.liu@sjtu.edu.cn

Weihao Cui
Shanghai Jiao Tong University
Shanghai, China
weihao@sjtu.edu.cn

Zhendong Zhang
Shanghai Qi Zhi Institute
Shanghai, China
zhangzd@sqz.ac.cn

Cong Guo
Shanghai Jiao Tong University
Shanghai Qi Zhi Institusion
Shanghai, China
guocong@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Li Li
Shanghai Jiao Tong University
Shanghai, China
lilijp@sjtu.edu.cn

Minyi Guo*
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

ABSTRACT

Graph neural networks (GNNs) are powerful tools for exploring and learning from graph structures and features. As such, achieving high-performance execution for GNNs becomes crucially important. Prior works have proposed to explore the sparsity (i.e., low density) in the input graph to accelerate GNNs, which uses the full-graph-level or block-level sparsity format. We show that they fail to balance the sparsity benefit and kernel execution efficiency. In this paper, we propose a novel system, referred to as *AdaptGear*, that addresses the challenge of optimizing GNNs performance by leveraging kernels tailored to the density characteristics at the subgraph level. Meanwhile, we also propose a method that dynamically chooses the optimal set of kernels for a given input graph. Our evaluation shows that *AdaptGear* can achieve a significant performance improvement, up to 6.49× (1.87× on average), over the state-of-the-art works on two mainstream NVIDIA GPUs across various datasets.

*Jingwen Leng and Minyi Guo are the corresponding authors.

CF '23, May 9–11, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0140-5/23/05…$15.00
https://doi.org/10.1145/3587135.3592199

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Graph Neural Networks, AI Frameworks, Graphics Processing Unit

ACM Reference Format:
Yangjie Zhou, Yaoxu Song, Jingwen Leng, Zihan Liu, Weihao Cui, Zhendong Zhang, Cong Guo, Quan Chen, Li Li, and Minyi Guo. 2023. AdaptGear: Accelerating GNN Training via Adaptive Subgraph-Level Kernels on GPUs. In *20th ACM International Conference on Computing Frontiers (CF '23), May 9–11, 2023, Bologna, Italy*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3587135.3592199

1 INTRODUCTION

Optimizing graph neural networks (GNNs) performance is a vital task of great interest to both academia and industry. As GNNs have demonstrated success in extending deep learning to graph structures and features, research in this area has advanced rapidly in recent years. Various variants of GNN models have been designed and explored, leading to significant breakthroughs in fields such as chemistry [21], neurology [6], anomaly detection [15, 47], and social networks or recommendations [17, 52]. As graph-structured data continues to grow, it is becoming increasingly imperative to optimize GNN performance to enable real-time analysis and decision-making [69, 70].

One of the important factors in optimizing the performance of GNNs is to understand and utilize the density/sparsity nature of their input graphs [1, 23]. Real-world graphs commonly exhibit community-based structures [19, 44, 54], which can be identified

using existing community-based ordering tools by grouping similar vertices together in ordinal order [3, 4, 7, 35]. This can result in variability in density distribution within the adjacency matrix of a single graph, with higher density on the diagonal reflecting the edge connectivity within a community, and lower density in other locations reflecting the edge connections between communities. Additionally, different input graph datasets can have distinct density characteristics, with the difference in density between different graphs potentially reaching several orders of magnitude [30]. Hence, it is imperative to consider these density characteristics while optimizing the performance of GNNs to ensure efficiency in training and deployment.

Previous studies on exploiting the graph sparsity (i.e., low density) on the modern parallel GPU platform can be divided into two categories based on the granularity of kernel mapping. The first category, referred to as full-graph-level kernel mapping, employs a single optimized kernel for the entire graph [32, 50, 66]. The second category is referred to as block-level kernel mapping. The input graph's adjacency matrix is divided into blocks during preprocessing. The optimal execution mode for each block is determined based on its density. After computation for each block is finished, the results are combined for blocks that correspond to the same set of vertices [63]. However, these approaches are insufficient to fully utilize the graph sparsity for accelerating GNNs. Specifically, the full-graph-level kernel mapping disregards the intra-graph density distribution. The block-level kernel mapping incurs additional overhead in runtime due to the kernel launch and result combination processes.

In this paper, we introduce *AdaptGear*, a novel system that addresses the challenge of optimizing the performance of GNNs by leveraging the unique characteristics of graph density distribution with minimal runtime overhead. The system starts by decomposing the input graph into subgraphs corresponding to intra-community and inter-community based on graph community features in a preprocessing stage. We then employ two key components for computational optimization. The first component, subgraph-level customized kernels, offers diverse kernel formats tailored to the specific characteristics with varying densities of individual subgraphs, resulting in more efficient utilization of computational resources and a more efficient training process. The second component, the adaptive selector, selects the appropriate kernel based on a feedback-driven approach during runtime. This ensures that the optimal kernel is used for each specific input graph, further improving computational performance and avoiding runtime overhead. Based on the innovative design, *AdaptGear* offers a novel and effective approach to optimize the performance of GNNs by utilizing the unique characteristics of graph density distribution.

Overall, our work makes the following contributions:

- We conduct a detailed analysis of the density distribution of both intra- and inter-graphs in order to understand their impact on the sparsity-based GNN execution approaches.
- We propose a set of subgraph-level customized kernels, which are tailored to the density characteristics of specific subgraphs to improve GNNs' computational efficiency.
- We introduce an adaptive selector that can determine the best-performing kernel based on a feedback-driven approach. This ensures that the optimal kernel is used for each specific graph, further improving performance and reducing runtime overhead.
- We propose *AdaptGear* to integrate the above optimizations and evaluate it against state-of-the-art techniques to demonstrate its effectiveness in optimizing GNN performance. Our results show that *AdaptGear* significantly improves performance, with an average improvement of 1.87× compared to existing methods.

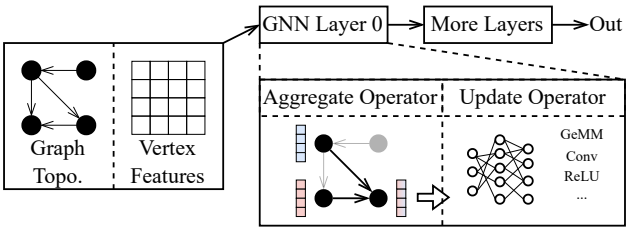


Figure 1: GNN general computation flow.

2 BACKGROUND AND MOTIVATION

This section provides a brief background of graph neural networks (GNNs) and their distinction from traditional deep neural networks (DNNs). We show that GNNs incorporate graph-related inputs so that their execution efficiency varies depending on the storage format of the input graph. We then examine the density distribution of both intra- and inter-graphs, which highlights the need for subgraph-level adaptive kernels in optimizing GNN performance.

2.1 Graph Neural Networks

Graph neural networks (GNNs) have recently gained significant attention in both academia and industry for their ability to effectively learn and infer on graph-structured data in non-Euclidean spaces [10, 70]. As such, achieving high-performance execution of GNNs has become an important research topic [1, 5, 70]. The input of a GNN model is a d-dimensional vector representation for each vertex in the input graph, known as an embedding. These embeddings are designed to have similar values for vertices with similar properties, such as similar subgraph structures, to facilitate efficient reasoning about graph-related problems [29, 41]. To generate these embeddings, GNNs combine feature transformation methods from DNNs with graph-based operations in graph processing that propagate and aggregate information throughout the graph structure.

GNN Computation. The computation flow of GNNs is illustrated in Fig. 1. The input of a GNN model comprises both the topological structure of the input graph and the dense feature embeddings associated with each vertex. The computation within each GNN layer is composed of two primary types of operations, as represented by the following equations:

$$a_v^{(k)} = \text{Aggregate}^{(k)}\left(\left\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\right\}\right)$$
$$h_v^{(k)} = \text{Update}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right)$$

where h_v^k represents the feature vector of vertex v at the k-th layer. The `Aggregate` function performs the aggregation of multiple feature vectors from adjacent vertices into a single feature

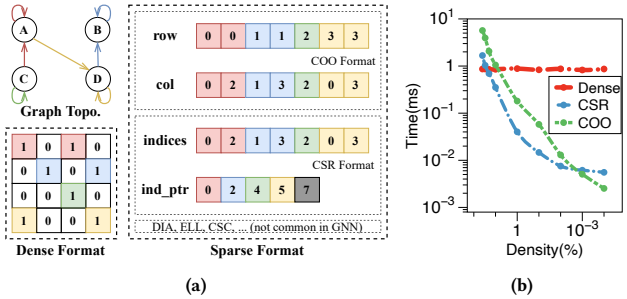


Figure 2: (a) Graph format example: Dense/CSR/COO. (b) Performance comparison of different format for the `aggregate-sum` operator in GCN first layer with Pubmed dataset on A100 GPU.

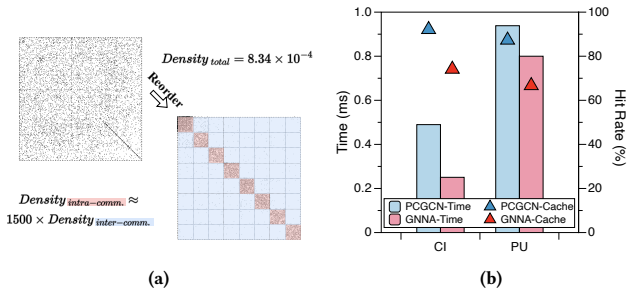


Figure 3: (a) The impact of commmunity-based reordering on the adjacency matrix of Citeseer dataset. (b) Comparison of performance and L2 cache hit rate for the `aggregate-sum` operator in GCN first layer with Citeseer and Pubmed datasets on A100 GPU.

vector using various specific operators, such as `max`, `mean`, and `sum`, which are referred to as `aggregate-max`, `aggregate-mean`, and `aggregate-sum`, respectively. The `Update` function utilizes neural network operations, such as a multilayer perceptron (MLP), to transform each vertex's feature vector into a new feature vector.

Graph Format and Execution Scheduling. As a distinct feature that sets GNNs apart from traditional DNNs, GNNs' input data include graph topology information. This topological data can be represented in various formats, such as the dense format, which utilizes a 2-dimensional array to represent the adjacency matrix of the graph, with a value of 1 indicating a connection between the corresponding source and destination vertices and 0 indicating otherwise. The dense format is convenient for continuous memory accesses but space-inefficient due to the presence of the large number of 0s in the graph. To reduce the storage overhead, GNN systems commonly employ sparse graph formats, such as compressed sparse row (CSR) and coordinate format (COO). The CSR format contains two arrays: the row pointer array and the index columns array. The row pointer array stores the indices of the start of each row in the columns array, while the columns array stores the column indices of the non-zero elements in the matrix. These two arrays can be used together to reconstruct the graph topology. The COO format stores an array of tuple for non-zero elements. Each tuple represents an edge of the graph, with the row and column indices representing the destination and source vertices of the edge, respectively. Fig. 2a shows examples of these different dense and sparse formats.

The choice of graph storage format not only affects the physical organization of the data, but also significantly impacts the scheduling approach that parallelizes the GNNs' execution on GPUs. Specifically, the dense format treats a graph-related operator as a dense operator, while the CSR and COO formats correspond to vertex-parallel [32] and edge-parallel [72] approaches, respectively. Vertex-parallelism is achieved by assigning each thread to a specific vertex and processing all of its associated edges sequentially. In comparison, edge-parallelism is achieved by assigning each thread to a specific edge and processing these computations in parallel.

We show that the optimal graph format choice depends on the specific characteristics of the input graph and requires careful consideration. In order to experimentally analyze how this choice is made, we generate input graphs with various densities using RMAT [9] tool. We adjust the number of edges with a fixed vertex size of 19717, which is the size of Pubmed dataset [5]. Using an A100 GPU [56], we compare performance results for these graphs with different sparsity using CSR, COO, and dense data formats, performing `aggregate-sum` operations. The results shown in Fig. 2b indicate that the dense format has optimal execution efficiency at high density, CSR performs optimally as density decreases, and COO becomes the optimal solution at low density.

2.2 Intra-Graph Density Analysis

Real-world graphs commonly exhibit community-based structures [19, 44, 54], which can be identified using existing community-based ordering tools [3, 35] by grouping similar vertices together in ordinal order. Each graph community has a group of vertices with strong intra-community connections, but weak connections to other vertices. However, the ordinal numbers of an original graph are often randomly assigned. Community-based reordering aims to reorder the vertices of a graph such that neighboring vertices belong to the same community. This reordering also leads to a differentiated density distribution for different subgraphs within a graph. This phenomenon is illustrated shown in Fig. 3a, through an examination of the Citeseer dataset [20] using METIS [35], a commonly utilized community-based reordering tool. The distribution of the adjacency matrix before reordering is observed to be random and irregular [4]. However, after reordering, the distribution of the adjacency matrices corresponding to intra- and inter-community edges exhibits distinct characteristics. These edges correspond to the intra- and inter-community subgraphs, respectively. These two subgraphs display a significant difference in density, with the density of intra-communities edges (i.e., on the diagonal) higher than the density of inter-communities edges.

Previous research has explored using the graph topology obtained after reordering for high-performance GNN computation. This research can be classified into two categories based on the granularity of kernel mapping. The first category, referred to as full-graph-level kernel mapping, implements a static optimization kernel for the entire graph [50, 66]. The second category, referred to as block-level kernel mapping, invokes the computational kernel independently for each block of the adjacency matrix [63]. The full-graph-level mapping ignores the density distribution pattern

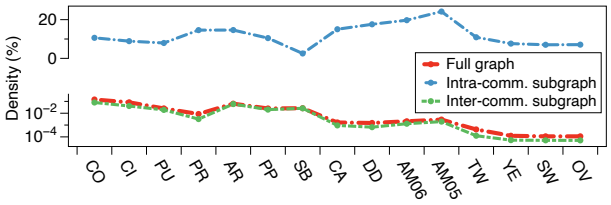


Figure 4: The average density of full, intra-community and inter-community subgraphs for different datasets.

brought by the community-based reordering of the adjacency matrix and treats it as a form of the orthogonal preprocessing optimization method. In contrast, the block-level kernel mapping selects the appropriate execution mode for each block based on its density at a fine-grained level, and it then merges the results of the blocks corresponding to the same set of vertices.

As typical examples of these two categories, we use GNNAdvisor [66] and PCGCN [63] for performance analysis. We collected the execution time of the first layer of GCN and the L2 cache hit rate via nsight system profiler [57] on A100 GPU. The results in Fig. 3b reveal that while PCGCN achieves a higher cache hit rate, it incurs a longer execution time. This is due to the fact that PCGCN employs an overly fine-grained granularity of kernel mapping, which incurs additional runtime overhead of kernel launching and results merging. Therefore, it is essential to identify an appropriate mapping granularity and leverage the distribution characteristics of the reordered adjacency matrix to enhance performance.

2.3 Inter-Graph Density Analysis

In real-world graphs, density distribution variations manifest both at the intra-graph and inter-graph levels. In other words, different graphs have significant differences in density properties.

To investigate these variations, we analyze 15 commonly used graph datasets as shown in Tbl. 1. The number of vertices and edges in each dataset is recorded to provide insight into the scale of the graph. Additionally, the features and class sizes of the datasets vary, which could impact the computational complexity and memory usage of certain graph operations.

Table 1: Details of graph datasets used for evaluation.

dataset	#Vertex	#Edge	#Feat	#Class
cora (CO)	2708	10556	1433	7
citeseer (CI)	3327	9228	3703	6
pubmed (PU)	19717	99203	500	3
PROTEINS_full (PR)	43466	162088	29	2
artist (AR)	50515	1638396	100	12
ppi (PP)	56944	818716	50	121
soc-BlogCatalog (SB)	88784	2093195	128	39
com-amazon (CO)	334863	1851744	96	22
DD	334925	1686092	89	2
amazon0601 (AM06)	403394	3387388	96	22
amazon0505 (AM05)	410236	4878874	96	22
TWITTER-Real-Graph-Partial (TW)	580768	1435116	1323	2
Yeast (YE)	1710902	3636546	74	2
SW-620H (SW)	1888584	3944206	66	2
OVCAR-8H (OV)	1889542	3946402	66	2

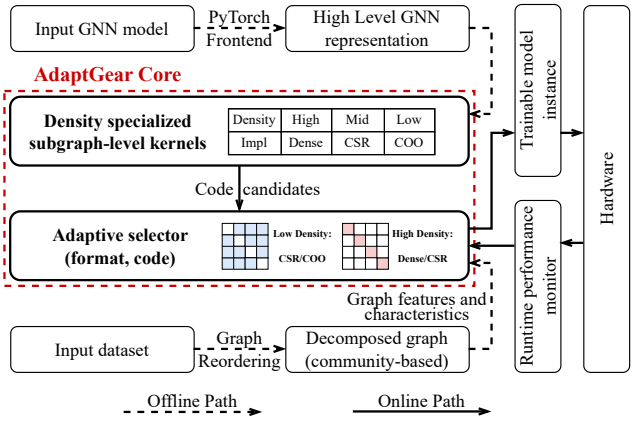


Figure 5: The design of *AdaptGear*.

We quantify and analyze the variability of density distribution for different datasets after community-based reordering using the METIS algorithm. The size of each community is set to 16. As depicted in Fig. 4, the results confirm our previous analysis on intra-graph in Sec. 2.2 that there are distinctions in the density distributions of intra-community subgraphs and inter-community subgraphs. Furthermore, the results demonstrate that these distinctions in characteristics also exist between different datasets. As such, it can be inferred that using a fixed format for all datasets does not result in optimal performance improvement.

3 DESIGN OF *AdaptGear*

In this section, we present our training system, *AdaptGear*, which efficiently exploits both the intra- and inter-graph sparsity to accelerate GNN training. We first present an overview of our system and then present details for its two key components, which are the subgraph-level customized kernels and adaptive selector.

3.1 Overview

Fig. 5 illustrates the overview of *AdaptGear*, a GNN acceleration system that utilizes subgraph-level adaptive kernels. The input GNN model is represented using a PyTorch front-end, while the input graph dataset is decomposed under community-based reordering guidelines. The core design of *AdaptGear* includes two modules: customized CUDA kernel templates and an adaptive code selector.

The first core component of *AdaptGear* is the customization of CUDA kernels, which are designed to handle intra- and inter-graphs with varying density. Instead of using static kernels, *AdaptGear* designs customized kernels for both intra-community and inter-community subgraphs with varying densities. This approach establishes a comprehensive strategy space, offering the potential for high-performance optimizations of GNNs.

The second core component of *AdaptGear* is the adaptive code selector, which selects the optimal CUDA kernel template during runtime. By monitoring and profiling the performance of each

Table 2: Comparison of existing graph operator acceleration methods with our proposed *AdaptGear*.

Kernel Mapping Granularity	Format Strategy	Existing Works	Runtime Overhead
Full-Graph-Level	Static	GNNAdvisor [66], NeuGraph [50]	Low
Block-Level	Adaptive	PCGCN [63]	High
Subgraph-Level	**Adaptive**	**Ours**	**Low**

subgraph kernel during the initial few iterations, the adaptive code selector selects the optimal CUDA kernel template that is best suited to the corresponding inputs.

Table 2 compares our system against existing GNN acceleration works. Previous works either employ full-graph-level execution granularity, not fully exploiting the performance optimization opportunities provided by the distribution of intra-graph densities, or utilize block-level execution strategies that incur substantial runtime overhead. In contrast, *AdaptGear* utilizes subgraph-level granularity with adaptive kernel mapping, effectively leveraging the performance optimization opportunities presented by both intra- and inter-graph density distributions while minimizing runtime overhead as much as possible.

3.2 Subgraph-level Customized Kernel

In this subsection, we describe the design of subgraph-level kernels in *AdaptGear*. The optimization of graph-related operations on GPUs requires two key considerations. Firstly, it is essential to implement an appropriate mapping from computation to the CUDA software abstractions such as CTA (Cooperative Thread Array), thread, etc., to fully leverage the computing resources of the GPU [12]. Secondly, the GPU's memory hierarchy, including global memory, shared memory, and registers, exhibits different access overhead from large to small [53]. To achieve lower access overhead and better performance, efficient memory management is crucial. As an architecture designed to process regular, continuous data, GPU is inefficient when processing irregular sparse data such as graphs [1, 5]. So, customized optimizations are required due to the irregular and sparse nature of GNNs.

As previously mentioned in Sec. 2, there is a diversity in preferred input formats for graphs with different density levels. Furthermore, intra- and inter-community subgraphs exhibit different density distribution properties. To harness these distinct features, we design a series of density-specific subgraph-level kernels.

CSR-based kernel. As shown in the left side of Fig. 6, we present the CSR-based kernel for inter-community subgraphs, which are characterized by low and irregular density distributions. Since the CSR format stores the adjacency matrix in a row-major way, our approach maps a CTA to multiple destination vertices, with each thread accessing the corresponding source neighbor's vertex features serially. To exploit the reuse of topological data within the CTA, we cache the data in shared memory. On the other hand, as the index of source neighbors cannot be determined within a finite range that fits the size of shared memory, we load the vertex features directly from global memory into registers.

Next, we introduce our CSR-based customized kernel for intra-community subgraphs. These subgraphs are characterized by high

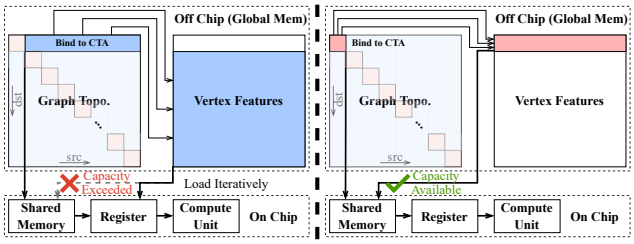


Figure 6: Comparison of kernel execution with CSR format for (left) low-density inter-community subgraph and (right) high-density intra-community subgraph.

density, with edges concentrated in blocks around the diagonal of the adjacency matrix. In light of this observation, we present our customized CSR-based kernel for intra-community subgraphs as shown in the right side of Fig. 6. We map a CTA to a community on a corresponding adjacency matrix. In this mapping relationship between CTA and graph topology, the range of indexes of source vertices that a CTA needs to access is limited and can be determined in advance. Thus, we can pre-load all the vertex features that are required to be accessed into the shared memory. This improves the efficiency of memory accesses, as the intra-community subgraph has a higher density and these features are accessed repeatedly. Furthermore, to avoid excessive shared memory allocation from negatively affecting the parallelism of the CUDA kernel, we apply tiling techniques [37] when the feature size is large.

COO-based kernel. The COO format is distinctive from the CSR format in that it organizes data in an edge-wise manner. Therefore, we design COO-based kernel templates. As demonstrated by the example of the `aggregate-sum` kernel in Algo. 1, this kernel performs computation by allocating threads and conducting element-wise computation. The topology and vertex feature data accessed in the COO kernel are independent between threads, and thus the shared memory caching mechanism is not employed. This approach offers a high degree of parallelism, but destination vertices' updates must be atomic, making it more appropriate for input graph datasets

Algorithm 1: The coo-based `aggregate-sum` kernel.

input : Graph $G = (Row[E], Col[E])$, Vertex Feature Tensor $X[V][F]$
output: Vertex Feature Tensor $Y[V][F]$

```
for edge_id = 0 to E − 1 in parallel do
    for dim_id = 0 to F − 1 in parallel do
        dst_id = Row [edge_id];
        src_id = Col [edge_id];
        Atomic_Add(Y [dst_id][dim_id], X
          [src_id][dim_id]);
    end
end
```

with extremely low density. Consequently, the COO-based kernel is only utilized as a code candidate for inter-community subgraphs.

Dense-based kernel. We also propose a dense format-based approach for the intra-community subgraph. Specifically, we map a CTA to a community's adjacency matrix block and then use sequential access and computation to perform sparse graph operations. For example, by storing the adjacency matrix in dense format and then directly performing in batched GEMM kernel [58] on it with the vertex feature, the result is equivalent to the `aggregation-sum` graph operator. Traditional graph computation methods do not utilize this approach due to the low density of graphs and the resulting large number of invalid accesses and computations. However, in the case of intra-community subgraphs with high density, this method can provide optimal performance gains in some scenarios. The use of Tensor Core for 32-bit computation has been supported only since the beginning of the Ampere architecture. To ensure computational equivalence, we use Tensor Core in the A100 GPU and CUDA Core in the V100 GPU as compute units for the dense format kernel in our subsequent implementations. The mixed precision computational approach enables the utilization of Tensor Core on pre-Ampere architectures, and we leave this as future work.

3.3 Adaptive Selector

After customizing the kernels for high-performance computational optimization of input subgraphs with varying densities on the back-end, the preprocessing stage involves utilizing a community-based reordering tool to decompose the input graph into inter-community and intra-community subgraphs. Specifically, we iterate through each edge of the graph after reordering and calculate the block index of the adjacency matrix where the edge is located using the indexes of the source and destination vertices of the edge during the preprocessing stage. When the block index corresponding to the source vertex is equal to the block index corresponding to the destination vertex, it means that the edge is on the diagonal of the adjacency matrix, i.e., it belongs to the intra- community subgraph. Otherwise, this edge is added to the inter-community subgraph.

The final pivotal aspect in attaining efficient training is the core component: the adaptive selector. This selector employs a feedback-driven approach to determine the most suitable kernels for different input graphs. Due to the limited number of subgraph-level customized kernels provided by *AdaptGear*, specifically two for intra-subgraph and two for inter-subgraph, and the fact that GNN training requires hundreds or even thousands of iterations with a static topology graph [70], we adopt a feedback-based selection strategy. In the first few iterations of GPU training, we use a monitor to collect the running time of each subgraph kernel, which is then fed back to the runtime scheduler as the basis for kernel selection in the following iteration. Although this feedback collection process may cause some time loss due to monitoring, the performance losses incurred in the early iterations are considered insignificant in the overall context of the training process as evaluated in Sec. 6.3.

4 IMPLEMENTATION

In this section, we elaborate on the implementation details of *AdaptGear*. We start with the front-end programming interface and describe how the back-end integrates with our system.

```
1 import AdaptGear as AG
2 import torch
3
4 # Create a GCN class.
5 class GCN(torch.nn.Module):
6     def __init__(self,...):
7         self.gcn = AG.GCNConv(in_feats, h_feats)
8         ...
9     def forward(self, x, inter_subg, intra_subg):
10        x = self.gcn(x, inter_subg, intra_subg)
11        ...
12
13 # Define a GCN model.
14 model = GCN(...)
15
16 # Loading graph dataset.
17 graph = AG.load_graph(graph_file)
18
19 # Reorder and decompose graph.
20 inter_subg, intra_subg = AG.graph_decompose(graph, method='METIS',
       comm_size=16)
21
22 # Run model.
23 predict_y = model(x, inter_subg, intra_subg)
24
25 # Compute loss and accuracy.
26 # Gradient backpropagation for training.
```

Figure 7: Example of GCN model using *AdaptGear*'s interfaces.

4.1 User-level Interface

AdaptGear utilizes PyTorch as the front-end to enhance programmability and user-friendliness. As an illustration, a representative GCN model is provided as an example for using *AdaptGear* in Fig. 7.

AdaptGear offers two types of interfaces. The first is for the computation of GNN, such as the `AG.GCNConv` in Line 7 of Fig. 7. The second interface is for the preprocessing of graphs, such as the `AG.graph_decompose` in Line 19. Upon examination, it is clear that *AdaptGear* does not substantially deviate from the traditional GNN framework at the front-end level, but only adds a customized computational interface and an additional preprocessing step for graph decomposition. Consequently, it preserves good scalability and lowers the users' learning curve.

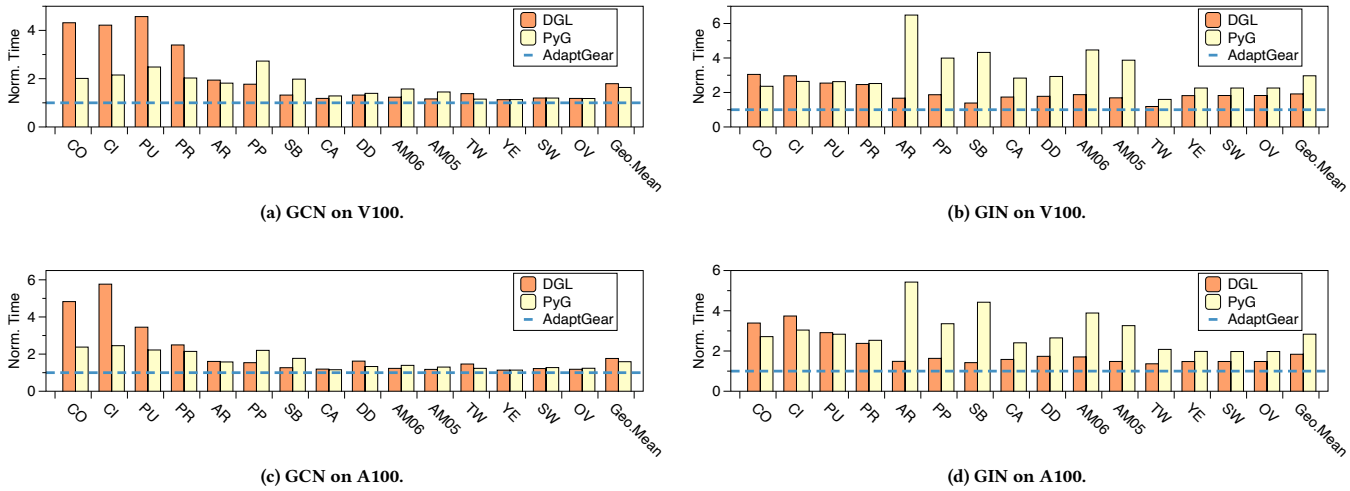
Another feature in *AdaptGear* is the adaptive selector, which operates in a transparent manner to the user during training. This design choice eliminates the need for the user to manually choose the optimal selection strategy, thereby improving ease of use.

4.2 System Integration

In this subsection, we detail the integration of the front-end interface with the back-end implementation in *AdaptGear*.

The front-end computational interface needs to be integrated with the back-end kernel implementation. *AdaptGear* offers a variety of density-customized kernels, which are built using C++/CUDA and integrated into the PyTorch framework through pybind11 [33]. The data is loaded using a Pytorch-based data loader and passed as a Tensor to *AdaptGear*'s back-end for computation on GPUs. Upon completion of the computation, the result tensor is returned to the original PyTorch framework for further uses like loss computations.

The preprocessing interface in *AdaptGear* consists of two stages: reordering and decomposition. The first stage involves the graph



(a) GCN on V100. (b) GIN on V100.

(c) GCN on A100. (d) GIN on A100.

Figure 8: End-to-end normalized training time result on two GPUs. *X*-axis indicates different graph datasets.

reordering using the default algorithm, METIS [35]. This stage allows for the specification of either the community size or the number of communities as parameters, providing flexibility in the reordering process. Furthermore, the specific reordering algorithm used in the backend has potential for future expansion. In the second stage, the graph is decomposed into intra- and inter-community subgraphs by traversing the graph once and dividing the edges based on their source and destination vertex indices.

5 METHODOLOGY

Experiments To evaluate *AdaptGear*, we use two enterprise-level GPUs as our hardware platforms: Tesla V100 [59] and Ampere A100 [56]. Tbl. 3 details our experimental setup.

Baselines We choose four baseline implementations for comparison: 1) Deep Graph Library (DGL) [64] is the state-of-the-art GNN framework that works for multiple DL frameworks. We choose PyTorch version in this work. 2) Pytorch-Geometric (PyG) [18] is another GNN framework which is built upon PyTorch. 3) GNNAdvisor [66] accelerates GNNs on GPUs with handwritten full-graph-level CUDA kernel implementations. 4) PCGCN [63] utilizes block-level adaptive kernel to leverage the intra-graph hybrid density distribution to accelerate GCN.

Benchmarks We use GCN [41] and GIN [71] as representative GNN models in our study. We follow the default configuration for layers, hidden features, and training parameters as outlined in their original papers for all baselines and *AdaptGear* to ensure a fair comparison. We run each benchmark 200 iterations of end-to-end training and present the average results to isolate the effects of randomness.

Table 3: Detailed experimental setup.

GPU	NVIDIA Tesla V100(80 SMs)	Nvidia Ampere A100 (108 SMs)
CPU	Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
OS	Ubuntu 18.04.5 (kernel 5.4.0)	Ubuntu 20.04.2 (kernel 5.11.0)
Driver	GPU Driver: 470.57	GPU Driver: 515.48.07
Software	CUDA: 11.6; Pytorch: 1.13	

Datasets We use 15 graph datasets that have also been used in many previous GNN optimization works [38, 46, 62]. The total count, sparsity, input feature size, and output classes vary significantly among these datasets. As such, our chosen datasets are sufficient to represent the graph in real-world scenarios. Tbl. 1 provides detailed information for these datasets.

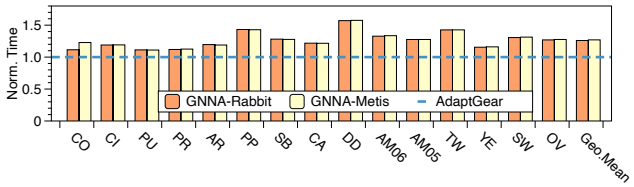
6 EVALUATION

In this section, we aim to evaluate the following points:

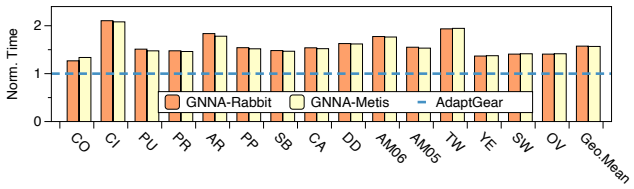
- What are the end-to-end performance improvements brought by *AdaptGear* compared to existing GNN frameworks and manual optimization efforts?
- How much does each design module in *AdaptGear* contribute to the overall performance improvement?
- How much is the additional overhead introduced by the design of *AdaptGear*?

6.1 End-to-end Performance Comparisons

State-of-the-Art GNN Frameworks. We first compare the end-to-end training performance of *AdaptGear* with two widely adopted GNN frameworks, including DGL [64] and PyG [18]. The results, as shown in Fig. 8, compare their normalized end-to-end execution time on two GPUs. To ensure the fairness of comparison, we use the same METIS community size setting of 16 for both the baselines and *AdaptGear*. Our results indicate that on both GPUs, *AdaptGear* achieves significant performance improvements over the baselines, with geometric average speedup values of 1.83× and 2.16× over DGL and PyG, respectively. This improvement is due to *AdaptGear*'s efficient exploitation of the density distribution at both the intra- and inter-graph levels. For each GNN model, *AdaptGear* achieves an average improvement of 1.69× and 2.33× on GCN and GIN, respectively. The more significant improvement



(a) GCN.



(b) GIN.

Figure 9: Performance comparisons between GNNAdvisor and *AdaptGear* on A100 GPU. *X*-axis indicates different graph datasets.

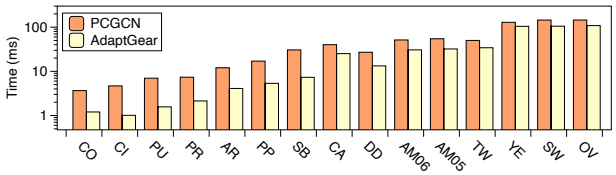


Figure 10: Performance comparisons of PCGCN and *AdaptGear* for GCN on A100 GPU. *X*-axis indicates different graph datasets.

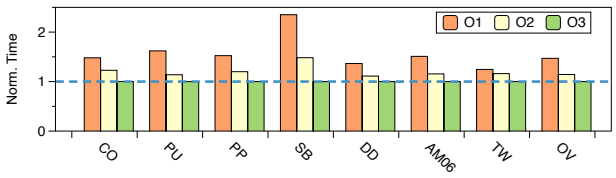


Figure 11: Execution time of different *AdaptGear* versions on A100.

on GIN is attributed to its higher proportion of time spent on graph-related operations, which are the main optimization scope of *AdaptGear*.

Manual Optimization. To demonstrate the performance benefits of *AdaptGear* compared to existing GNN manual optimization efforts, we choose GNNAdvisor [66] and PCGCN [63] as the full-graph-level and block-level acceleration baselines, respectively.

GNNAdvisor [66] uses rabbit-sort [3] as its default preprocessing step for reordering. To eliminate the impact of the preprocessing tool, we collect performance results from two reordering preprocesses for GNNAdvisor, one using rabbit-sort (referred to as `GNNA-Rabbit`) and the other using METIS (referred to as `GNNA-Metis`). Due to the space limit, we only show the results on the A100 GPU. As shown in Fig. 9, *AdaptGear* achieves an average 1.40× and 1.41× performance gain over `GNNA-Rabbit` and `GNNA-Metis` on A100 GPU. This result shows that *AdaptGear* yields good performance optimization results for different preprocessing methods. It is worth noting that the results trend for the V100 GPU is similar. Specifically, the geometric average performance speedup ratios of *AdaptGear* compared to `GNNA-Rabbit` and `GNNA-Metis` on the V100 GPU are 1.39× and 1.38×, respectively.

PCGCN [63] optimizes GCN performance through a block-level approach. However, the METIS parameters used are not clearly specified in their paper. Therefore, we traverse the METIS parameters within the range of 2 to 1024 at multiples of 2 intervals for PCGCN and present the optimal one as the final performance results. The results on A100, as shown in Fig. 10, indicate that despite providing PCGCN with a wide range of reordering parameters, its performance remains lower compared to that of *AdaptGear* for all datasets. Specifically, the geometric average performance speedup ratios of *AdaptGear* over PCGCN on the A100 and V100 GPUs are 2.30× and 2.59×, respectively.

Our comparison with these baselines highlights that the subgraph-level kernel mapping granularity utilized by *AdaptGear* is the more effective for optimizing the computation in GNNs.

6.2 Performance Improvement Breakdown

AdaptGear incorporates two performance optimization modules, including subgraph-level customized kernels and an adaptive selector. To evaluate the effectiveness of these modules, we design relevant baselines and conduct experiments to quantify their contribution. We define three optimization versions of *AdaptGear*. The `O1` version utilizes a static CSR kernel at the full-graph level, while the `O2` version employs static CSR kernels for intra-community subgraphs and COO kernels for inner-community subgraphs. The `O3` version incorporates subgraph-level adaptive sparse kernels.

We use the GCN model in this experiment and collect the performance results on A100 GPU. Due to the space limit, we omit the results for the other models and hardware. As illustrated in Fig. 11, the results show that the implementation of different optimization versions can lead to significant variations in performance improvement across different datasets. This highlights the crucial role that subgraph-level kernels and the adaptive selector play in the optimization process of *AdaptGear*.

6.3 Additional Studies

Runtime Overhead. There are two sources of additional runtime overhead in *AdaptGear*. The first is graph preprocessing, which includes graph reordering and graph decomposition, and needs to be performed only once before the training starts. The second is the adaptive selector, which monitors performance in the early iterations of GNN training. However, the overhead from both processes has a negligible impact compared to the overall GNN training process. To illustrate, we evaluate the overhead using the amazon0601 dataset. The overhead of the graph decomposition is 0.08 seconds, while the overhead of the graph reordering is 0.59 seconds. Furthermore, the runtime monitoring in our adaptive selector incurs less than 0.1 seconds in the early training iterations. Overall, the magnitude of these overheads is insignificant compared to the hours or even days required [16, 70] for the GNN training.

Memory Overhead. The graph decomposition potentially generates additional storage overhead for the topology. However, this

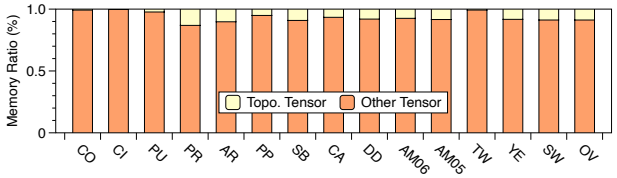


Figure 12: Memory overhead for storing subgraph topology (denoted as Topo. Tensor) in different datasets.

overhead is comparatively small as the majority of memory usage during GNN training is occupied by vertex features and gradient-related data [74]. To quantify the additional memory overhead, we measure the maximum memory overhead via PyTorch Profiler [60] for the GCN model on A100 GPU and compare it with the overhead for the additional subgraph topology storage. As shown in Fig. 12, the percentage of memory occupied by additional topological data is only 4.47% on average, which we believe is acceptable.

7 RELATED WORK

Graph Processing on GPUs. GPUs are crucial computing platforms in various task scenarios [42, 51]. Many studies have focused on optimizing GPUs at the architecture [27, 28, 45, 67], scheduling [13, 14, 26, 49, 68, 75], and algorithm [24, 25, 77] levels. Numerous graph processing systems [22, 39, 40, 43, 55, 61, 65] have been proposed to accelerate traditional graph algorithms on GPUs. These efforts have also explored various parallelization strategies, including vertex parallelism and edge parallelism for graph processing. There are also efforts to explore dynamic parallelization strategies through domain-specific language (DSL) [8, 73].

However, GNNs differ from traditional graph algorithms in terms of their graph operation characteristics and the dimensionality of their feature embeddings, leading to a parallelization strategy space that exceeds the capabilities of traditional graph processing systems.

GNN Frameworks. DGL [64] and PyG [18] are two popular GNN frameworks, which both employ a message-passing programming interface based on DNN frameworks. G^3 [48] focuses on utilizing graph processing frameworks for training GNNs on GPUs. Despite these efforts, the existing frameworks do not fully exploit the opportunities presented by graph density characteristics for optimizing graph kernel computations and hence cannot achieve optimal performance on GPUs.

On the other hand, other frameworks such as Roc [34], NeuGraph [50], and AliGraph [78] aim to address large-scale distributed GNN processing. However, their designs are orthogonal to *AdaptGear*. For handling large graphs requiring multiple GPUs to process, various well-established graph partitioning techniques can divide the graph into smaller subgraphs suitable for single-GPU training [2, 35, 36]. Therefore, the optimization of single-GPU training is equally beneficial for multi-GPU scenarios.

Graph Kernel Optimization. There are also some works that try to explore the graph kernels in GNNs to optimize GNNs. GNNAdvisor [66] provides customized kernels with scalable parameters for accelerating GNNs on GPUs. GE-SpMM [32] focuses on optimizing SPMM-like graph kernels in GNNs, while FeatGraph [31] extends TVM [11] to execute SPMM-like and SDDMM-like graph kernels on GPUs. uGrapher [76] addresses the challenges posed by the dynamics of input graphs and operators by providing a unified abstraction for all graph operations. However, the previous optimization efforts apply to entire input graphs and overlook the performance optimization opportunities presented by variations in the density distribution of intra-graphs. Additionally, PCGCN [63] adopts a block-level approach to leverage the intra-graph density distribution. However, it leads to additional accumulation operations, resulting in high runtime overhead in some scenarios.

Unlike all existing work, *AdaptGear* propose a set of subgraph-level customized kernels and adaptively select the current most suitable kernel for a given input, thus achieving a general GNN high-performance computation optimization.

8 CONCLUSION

In this work, we propose *AdaptGear*, a novel high-performance GNN training system that exploits both the intra- and inter-graph sparsities via adaptive subgraph-level kernels. Our system achieves a significant average speedup of 1.87× compared to various state-of-the-art works, including GNN frameworks and manual optimizations. The exceptional performance of *AdaptGear* is attributed to two key components: subgraph-level customized kernels that offer diverse kernel formats optimized for subgraphs of varying densities and an adaptive selector that selects the appropriate kernel for the input subgraph. Through the detailed evaluation, we demonstrate the effectiveness of *AdaptGear* and its potential as a common methodology for high-performance GNN training.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China under Grant 2021ZD0110104, the National Natural Science Foundation of China (NSFC) grant (62222210, U21B2017, and 620722-97). The authors would like to thank the anonymous reviewers for their constructive feedback for improving the work. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–38.
[2] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. 120–124.
[3] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.
[4] Vignesh Balaji and Brandon Lucia. 2018. When is Graph Reordering an Optimization?. In *IEEE International Symposium on Workload Characterization (IISWC)*.
[5] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful A Mojumder, Kihoon Jung, José L Abellán, Yash Ukidave, Ajay Joshi, John Kim, et al. 2021. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–23.
[6] Alaa Bessadok, Mohamed Ali Mahjoub, and Islem Rekik. 2022. Graph neural networks in network neuroscience. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).
[7] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. 587–596.

[8] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2021. Compiling Graph Applications for GPUs with GraphIt. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 248–261.

[9] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[10] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin P. Murphy. 2020. Machine Learning on Graphs: A Model and Comprehensive Taxonomy. *ArXiv* abs/2005.03675 (2020).

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[12] Shane Cook. 2012. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

[13] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. {DVABatch}: Diversity-aware {Multi-Entry}{Multi-Exit} Batching for Efficient Processing of {DNN} Services on {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 183–198.

[14] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2021. Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[15] Yingtong Dou, Zhiwei Liu, Li Sun, Yutong Deng, Hao Peng, and Philip S Yu. 2020. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 315–324.

[16] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2020. Benchmarking graph neural networks. (2020).

[17] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference*. ACM, 417–426.

[18] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[19] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.

[20] C Lee Giles, Kurt D Bollacker, and Steve Lawrence. 1998. CiteSeer: An automatic citation indexing system. In *Proceedings of the third ACM conference on Digital libraries*. 89–98.

[21] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[23] Chuangyi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xinyu Chen, Xiaofei Liao, and Hai Jin. 2019. A Survey on Graph Processing Accelerators: Challenges and Opportunities. *Journal of Computer Science and Technology* 34 (2019), 339–371.

[24] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[25] Cong Guo, Yuxian Qiu, Jingwen Leng, Xiaotian Gao, Chen Zhang, Yunxin Liu, Fan Yang, Yuhao Zhu, and Minyi Guo. 2022. SQuant: On-the-Fly Data-Free Quantization via Diagonal Hessian Approximation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=JXhROKNZzOc

[26] Cong Guo, Yuxian Qiu, Jingwen Leng, Chen Zhang, Ying Cao, Quanlu Zhang, Yunxin Liu, Fan Yang, and Minyi Guo. 2022. Nesting Forward Automatic Differentiation for Memory-Efficient Deep Neural Network Training. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 738–745.

[27] Cong Guo, Chen Zhang, Jingwen Leng, Zihan Liu, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2022. Ant: Exploiting adaptive numerical data type for low-bit deep neural network quantization. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1414–1433.

[28] Cong Guo, Yangjie Zhou, Jingwen Leng, Yuhao Zhu, Zidong Du, Quan Chen, Chao Li, Bin Yao, and Minyi Guo. 2020. Balancing efficiency and flexibility for DNN acceleration via temporal GPU-systolic array integration. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[29] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[30] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[31] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. Featgraph: A flexible and efficient backend for graph neural network systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[32] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[33] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11–Seamless operability between C++ 11 and Python. *URL: https://github. com/pybind/pybind11* (2017).

[34] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

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

[36] George Karypis and Vipin Kumar. 1998. Multilevel algorithms for multi-constraint graph partitioning. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE, 28–28.

[37] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. 2017. Cutlass: Fast linear algebra in cuda c++. *NVIDIA Developer Blog* (2017).

[38] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. 2016. Benchmark Data Sets for Graph Kernels. http://graphkernels.cs.tu-dortmund.de

[39] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European conference on computer systems*. 169–182.

[40] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

[41] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[42] Zhen Kong, Cheng-Zhong Xu, and Minyi Guo. 2011. Mechanism design for stochastic virtual resource allocation in non-cooperative cloud systems. In *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 614–621.

[43] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[44] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. 2008. Benchmark graphs for testing community detection algorithms. *Physical review E* 78, 4 (2008), 046110.

[45] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran, Pradip Bose, Quan Chen, Minyi Guo, and Vijay Janapa Reddi. 2020. Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 44–57.

[46] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[47] Ao Li, Zhou Qin, Runshi Liu, Yiqun Yang, and Dong Li. 2019. Spam Review Detection with Graph Convolutional Networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2703–2711.

[48] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. 2020. G3: when graph neural networks meet parallel graph processing systems on GPUs. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2813–2816.

[49] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 388–401.

[50] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[51] Shiheng Ma, Jianhui Ding, Weijia Jia, Kun Wang, and Minyi Guo. 2017. Transt: Type-based multiple embedding representations for knowledge graph completion.

In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2017, Skopje, Macedonia, September 18–22, 2017, Proceedings, Part I 10*. Springer, 717–733.

[52] Kelong Mao, Jieming Zhu, Xi Xiao, Biao Lu, Zhaowei Wang, and Xiuqiang He. 2021. UltraGCN: ultra simplification of graph convolutional networks for recommendation. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 1253–1262.

[53] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2016), 72–86.

[54] Mark EJ Newman. 2013. Spectral methods for community detection and graph partitioning. *Physical Review E* 88, 4 (2013), 042822.

[55] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for GPU-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

[56] NVIDIA. 2021. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[57] NVIDIA. 2021. Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[58] CUDA Nvidia. 2008. Cublas library. *NVIDIA Corporation, Santa Clara, California* 15, 27 (2008), 31.

[59] Tesla NVIDIA. 2017. Nvidia tesla v100 gpu architecture.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[61] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[62] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.

[63] Chao Tian, Lingxiao Ma, Zhi Yang, and Yafei Dai. 2020. Pcgcn: Partition-centric processing for accelerating graph convolutional network. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 936–945.

[64] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[65] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[66] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[67] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1083–1095.

[68] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of service support for fine-grained sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 269–281.

[69] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2020. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)* (2020).

[70] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[71] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[72] Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. 2022. Understanding gnn computational graph: A coordinated computation, io, and memory perspective. *Proceedings of Machine Learning and Systems* 4 (2022), 467–484.

[73] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[74] Ao Zhou, Jianlei Yang, Yeqi Gao, Tong Qiao, Yingjie Qi, Xiaoyi Wang, Yunli Chen, Pengcheng Dai, Weisheng Zhao, and Chunming Hu. 2021. Brief industry paper: Optimizing memory efficiency of graph neural networks on edge computing platforms. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 445–448.

[75] Xiaojie Zhou, Kun Wang, Weijia Jia, and Minyi Guo. 2017. Reinforcement learning-based adaptive resource management of differentiated services in geo-distributed data centers. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 1–6.

[76] Yangjie Zhou, Jingwen Leng, Yaoxu Song, Shuwen Lu, Mian Wang, Chao Li, Minyi Guo, Wenting Shen, Yong Li, Wei Lin, et al. 2023. 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*. 878–891.

[77] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 214–225.

[78] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.