GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs

Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding, *University of California, Santa Barbara*

https://www.usenix.org/conference/osdi21/presentation/wang-yuke

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Open access to the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs

Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding
University of California, Santa Barbara

Abstract

As the emerging trend of graph-based deep learning, Graph Neural Networks (GNNs) excel for their capability to generate high-quality node feature vectors (embeddings). However, the existing one-size-fits-all GNN implementations are insufficient to catch up with the evolving GNN architectures, the ever-increasing graph sizes, and the diverse node embedding dimensionalities. To this end, we propose **GNNAdvisor**, an adaptive and efficient runtime system to accelerate various GNN workloads on GPU platforms. First, GNNAdvisor explores and identifies several performance-relevant features from both the GNN model and the input graph, and uses them as a new driving force for GNN acceleration. Second, GNNAdvisor implements a novel and highly-efficient 2D workload management, tailored for GNN computation to improve GPU utilization and performance under different application settings. Third, GNNAdvisor capitalizes on the GPU memory hierarchy for acceleration by gracefully coordinating the execution of GNNs according to the characteristics of the GPU memory structure and GNN workloads. Furthermore, to enable automatic runtime optimization, GNNAdvisor incorporates a lightweight analytical model for an effective design parameter search. Extensive experiments show that GNNAdvisor outperforms the state-of-the-art GNN computing frameworks, such as Deep Graph Library (3.02× faster on average) and NeuGraph (up to 4.10× faster), on mainstream GNN architectures across various datasets.

1 Introduction

Graph Neural Networks (GNNs) emerge to stand on the frontline for handling many graph-based deep learning tasks (*e.g.,* node embedding generation for node classification [9, 14, 23] and link prediction [6, 28, 51]). Compared with standard methods for graph analytics, such as random walks [16, 47] and graph Laplacians [7, 34, 35], GNNs highlight themselves with the interleaved two-phase execution of both graph operations (scatter-and-gather [15]) at the *Aggregation* phase, and Neural Network (NN) operations (matrix multiplication) at the *Update* phase, to achieve significantly higher accuracy [27, 52, 55] and better generality [17]. Yet, the state-of-the-art GNN frameworks [11, 36, 53, 54], which follow a one-size-fits-all implementation scheme, often suffer from poor performance when handling more complicated GNN architectures (*i.e.,* more layers and higher hidden dimensionality in each layer) and diverse graph datasets.

Specifically, previous work that supports both GNN training and inference can be classified into two categories. The first type [36, 54] is built on popular graph processing systems and is combined with NN operations. The second type [11, 53], in contrast, starts with deep learning frameworks and is extended to support vector-based graph operations. However, these existing solutions are still preliminary and inevitably fall short in the following three major aspects, even on common computing platforms such as GPUs.

Failing to leverage GNN input information. GNN models demonstrate great diversity in terms of layer sequences, types of aggregation methods, and the dimension size of node embeddings. These profoundly impact the effectiveness of various system optimization choices. The diversity of input graphs further complicates the problem. Unfortunately, current GNN frameworks [11, 36, 53] follow a one-size-fits-all optimization scheme and fail to craft an optimization strategy that maximizes efficiency for a particular GNN application's settings. Some classical graph systems [2, 3] have exploited input characteristics to facilitate more efficient optimizations, but they only focus on simple graph algorithms like PageRank [45] while having no support for GNN models.

Optimizations not tailored to GNN. While the update phase in GNNs involves NN operations that are dense in computation and regular in memory access, the aggregation phase is usually sparse in computation and highly irregular in memory access. Without dedicated optimization, it will inevitably become the performance bottleneck. Existing GNN frameworks [11, 36, 53] simply extend the optimization schemes from classical graph systems [26, 54], and do not address the difference between GNN and graph processing. For example,

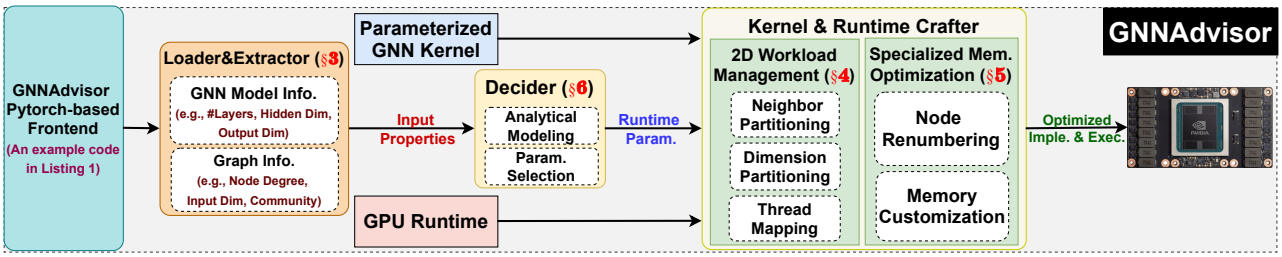

Figure 1: Overview of GNNAdvisor.

each node is associated with an embedding attribute in GNNs while each node only has a single scalar attribute in traditional graph processing. Such difference invokes novel design principles for GNNs towards more aggressive parallelism and more efficient memory optimization.

Listing 1: Example of a 2-layer GCN in GNNAdvisor.

```
import GNNAdvisor as GNNA
import torch
# import other packages ...

# Create a GCN class.
class GCN(torch.nn.Module):
    def __init__(self, inDim, hiDim, outDim, nLayers):
        self.layers = torch.nn.ModuleList()
        self.layers.append(GNNA.GCNConv(inDim, hiDim))
        for i in range(nLayers - 2):
            layer = GNNA.GCNConv(hiDim, hiDim)
            self.layers.append(layer)
        self.layers.append(GNNA.GCNConv(hiDim, outDim))
        self.softmax = torch.nn.Softmax()

    def forward(self, X, graph, param):
        for i in range(len(self.layers)):
            X = self.layers[i](X, graph, param)
            X = self.ReLU(X)
        X = self.softmax(X)
        return X

# Define a two-layer GCN model.
model = GCN(inDim=100, hiDim=16, outDim=10, nLayers=2)

# Loading graph and extracting input propertities.
graphObj, inputInfo = GNNA.LoaderExtractor(graphFile,
                                           model)
# Set runtime parameters automatically.
X, graph, param = GNNA.Decider(graphObj, inputInfo)

# Run model.
predict_y = model(X, graph, param)

# Compute loss and accuracy.
# Gradient backpropagation for training.
```

Poor runtime support for input adaptability. Prior GNN frameworks [11,36,53] rely on a Python-based high-level programming interface for ease of user implementation. These frameworks employ static optimizations through a compiler or manually-optimized libraries. Nevertheless, some critical performance-related information for a GNN is only available at runtime (*e.g.*, node degree and embedding size). Without adaptable designs that can leverage such runtime information, we would easily suffer from an inferior performance because of the largely under-utilized the GPU computing resources and inefficient irregular memory access. This limitation motivates the need for runtime environments with flexible designs to handle a wide spectrum of inputs effectively.

To this end, we propose, GNNAdvisor, an adaptive and efficient runtime system for GNN acceleration on GPUs. GNNAdvisor leverages Pytorch [46] as the front-end to improve programmability and ease user implementation. We show a representative 2-layer Graph Convolutional Network (GCN) [27] in GNNAdvisor at Listing 1. At the low level, GNNAdvisor is built with C++/CUDA and integrated with Pytorch framework by using Pytorch Wrapper. It can be viewed as a new type of Pytorch operator with a set of kernel optimizations and runtime support. It can work seamlessly with existing operators from the Pytorch Framework. Data is loaded with the data loader written in Pytorch and passed as a Tensor to GNNAdvisor for computation on GPUs. Once the GNNAdvisor completes its computation at the GPU, it will pass the data Tensor back to the original Pytorch framework for further processing. As detailed in Figure 1, GNNAdvisor consists of several key components to facilitate the GNN optimization and execution on GPUs. First, GNNAdvisor introduces an input **`Loader&Extractor`** to exploit the input-level information that can guide our system-level optimizations. Second, GNNAdvisor incorporates a **`Decider`** consisting of analytical modeling for automatic runtime parameter selection to reduce manual effort in design optimization, and a lightweight node renumbering routine to improve graph structural locality. Third, GNNAdvisor integrates a **`Kernel&Runtime Crafter`** to customize our parameterized GNN kernel and CUDA runtime, which consists of an effective 2D workload management (considering both the number of neighbor nodes and the node embedding dimensionality) and a set of GNN-specialized memory optimizations.

Note that in this project, we mainly focus on the setting of single-GPU GNN computing, which is today's most popular design adopted as the key component in many state-of-the-art frameworks, such as DGL [53] and PyG [11]. Single-GPU GNN computing is desirable for two reasons: First, many GNN applications with small to medium size graphs (*e.g.*, molecule structure) can easily fit the memory of a single GPU. Second, in the case of large-size graphs that can only be handled by out-of-GPU-core and multi-GPU processing, numerous well-studied graph partition strategies (*e.g.*, METIS [22]) can cut the giant graphs into small-size subgraphs to make them suitable for a single GPU. Therefore, the optimization of

both the out-of-GPU-core (*e.g.*, GPU streaming processing) and multi-GPU GNN computation still largely demands performance improvements on a single GPU. Moreover, while our paper focuses on GNNs, our proposed methodology can be applied to optimize various types of irregular workload (*e.g.*, social network analysis) targeting GPUs as well.

Overall, we make the following contributions:

- We are the first to explore GNN input properties (§3) (*e.g.*, GNN model architectures and input graphs), and give an in-depth analysis of their importance in guiding system optimizations for GPU-based GNN computing.
- We propose a set of GNN-tailored system optimizations with parameterization, including a novel 2D workload management (§4) and specialized memory customization (§5) on GPUs. We incorporate the analytical modeling and parameter auto-selection (§6) to ease the design space exploration.
- Comprehensive experiments demonstrate the strength of GNNAdvisor over state-of-the-art GNN execution frameworks, such as Deep Graph Library (average 3.02×) and NeuGraph (average 4.36×), on mainstream GNN architectures across various datasets.

2 Background and Related Work

In this section, we introduce the basics of GNNs and two major types of GNN computing frameworks: *GPU-based graph systems* and *deep learning frameworks*.

2.1 Graph Neural Networks

Figure 2 visualizes the computation flow of GNNs in one iteration. GNNs compute the node feature vector (embedding) for node v at layer $k+1$ based on the embedding information at layer k ($k \geq 0$), as shown in Equation 1,

$$\begin{aligned} a_v^{(k+1)} &= \textbf{Aggregate}^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)}) \\ h_v^{(k+1)} &= \textbf{Update}^{(k+1)}(a_v^{(k+1)}) \end{aligned} \quad (1)$$

where $h_v^{(k)}$ is the embedding vector for node v at layer k; $h_v^{(0)}$ is computed from the task-specific features of a vertex (e.g., the text associated with the vertex, or some scalar properties of the entity that the vertex represents) via some initial embedding mapping that is used only for this ingest of symbolic values into the embedding space; $a_v^{(k+1)}$ is the aggregation results through collecting neighbors' information (*e.g.*, node embeddings); $\mathbf{N}(v)$ is the neighbor set of node v. The aggregation method and the order of aggregation and update could vary across different GNNs. Some methods [17,27] just rely on the neighboring nodes while others [52] also leverage edge properties, by combining the dot product of the end-point nodes of each edge, along with any edge features (edge type and other attributes). The update function is generally composed of standard NN operations, such as a single fully connected layer or a multi-layer perceptron (MLP) in the form of $w \cdot a_v^{(k+1)} + b$, where w and b are the learnable weight and bias parameters, respectively. The common choices for node embedding dimensions are 16, 64, and 128, and the embedding dimension may change across different layers.

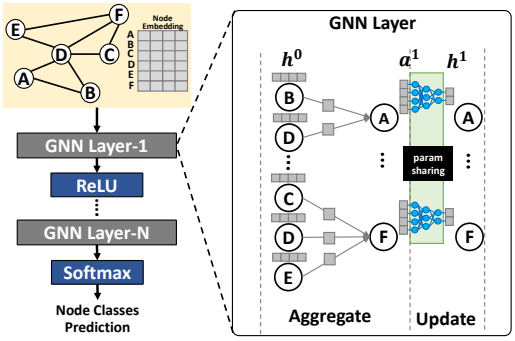

Figure 2: GNN General Computation Flow.

After passing through several iterations of aggregation and update (*i.e.*, several GNN layers), we will get the output embedding of each node, which can usually be used for various downstream graph-based deep learning tasks, such as node classification [9, 14, 23] and link prediction [6, 28, 51]. Note that the initial node embedding for GNN's input layer may come with the original graph dataset or can be generated by a set of graph embedding algorithms, such as [5, 10, 16], which is not included in the computation of GNNs models (generating the hidden and output node embeddings).

2.2 Graph Processing Systems

Numerous graph processing systems [26, 32, 33, 38, 54] have been proposed to accelerate traditional graph algorithms. The major commonalities of these systems include the vertex/node-centric programming abstraction, edge-centric processing, and system optimizations to reduce the computation irregularity (*e.g.*, workload imbalance) and memory access irregularity (*e.g.*, non-coalesced global memory access). However, extending these graph processing systems to support GNN computing meets with substantial challenges.

First, the common algorithm optimizations in graph processing may not benefit GNNs. For example, graph traversal algorithms, such as Breadth-first Search, rely on iterative computing on node frontiers (active neighbors). Therefore, a set of frontier-based optimizations, such push-pull traversal [32,33], and frontier filtering [32,33,54], have been extensively studied. However, GNNs consistently maintain fixed-sized frontiers (all neighbors) of each node across iterations.

Second, the system optimization techniques for graph processing would benefit GNNs only after careful adaption and calibration. For example, node/edge-centric processing [33, 54] and shard-based graph representation [26] are

tailored for processing nodes/edges represented with a single scalar attribute. In GNNs, there's another dimension for data parallelism, namely the embedding dimension, which tends to be large Therefore, previous design trade-offs between the coarse-grained node-level parallelism and node-value locality should be further extended to balance dimension-wise parallelism and node-embedding locality at a finer granularity.

Third, some essential functionalities of GNN computing are missing in graph systems. For example, the node update based on NN computing for both the forward value propagation and the complicated backward gradient propagation is not available in graph systems [26, 29, 32, 33, 38, 49, 54]. In contrast, Pytorch [46] and Tensorflow [1] feature an analytic differentiation function for automatic gradient computations on various deep learning model architectures and functions. Therefore, extending the graph-processing system to support GNN computing requires non-trivial efforts, and thus we develop GNNAdvisor on top of a deep learning framework.

2.3 Deep Learning Frameworks

Various NN frameworks have been proposed, such as Tensorflow [1], and Pytorch [46]. These frameworks provide the end-to-end training and inference support for traditional deep-learning models with various NN operators, such as linear and convolutional operators. These operators are highly optimized for Euclidean data (*e.g.*, image) but lack support for non-Euclidean data (*e.g.*, graph) in GNNs. Extending NN frameworks to support GNN that takes the highly-irregular graphs as the input is facing several challenges.

First, NN-extended GNN computing platforms [11, 53] focus on programmability and generality for different GNN models but lack efficient backend support to achieve high performance. For example, Pytorch-Geometric (PyG) [11] uses the torch-scatter [12] library implemented with CUDA as its major building block of graph aggregation operations. The torch-scatter implementation scales poorly when encountering large sparse graphs with high-dimensional node embedding because its kernel design essentially borrows the design principles of graph-processing systems by using excessive high-overhead atomic operations to support node embedding propagation. A similar scalability problem is also observed in Deep Graph Library (DGL) [53], which incorporates an off-the-shelf Sparse-Matrix Multiplication (SpMM) (*e.g.*, *csrmm2* in cuSparse [39]) for simple sum-reduced aggregation [17, 27] and leverages its own CUDA kernel for more complex aggregation scheme with edge attributes [52, 55].

Second, major computation kernels [11, 53] are hard-coded without design flexibility, which is essential to handle diverse application settings with different input graph sizes and node embedding dimensionality. From the high-level interface, users are only allowed to define the way of composing these kernels externally. Users are not allowed to customize kernels internally based on the known characteristics of GNN model architectures, GPU hardware, and graph properties.

3 Input Analysis of GNN Applications

In this section, we argue that the GNN input information can guide the system optimization, based on our key observation that different GNN application settings would favor different optimization choices. We introduce two types of GNN input information and discuss their potential performance benefits and extraction methods.

3.1 GNN Model Information

While the GNN update phase follows a relatively fixed computing pattern, the GNN aggregation phase shows high diversity. The mainstream aggregation methods of GNNs can be categorized into two types:

The first type is aggregation (*e.g.*, *sum*, and *min*) with only the embeddings of neighbor nodes, as in Graph Convolutional Network (GCN) [27]. For GNNs with this type of aggregation, the common design practice is to reduce the node embedding dimensionality during the update phase (*i.e.*, multiplying the node embedding matrix with the weight matrix) [11, 27, 53] before the aggregation (gather information from neighbor node embedding) at each GNN layer, thereby, largely reducing the data movements during the aggregation. In this case, improving memory locality would be more beneficial, in that more node embeddings can be cached in fast memory (*e.g.*, L1 cache of GPUs) to exploit performance benefits.

The second type is aggregation with special edge features (*e.g.*, weights, and edge vectors that are computed by combining source and target nodes) applied to each neighbor node, as in Graph Isomorphism Network (GIN) [55]. This type of GNN must work on large full-dimensional node embeddings to compute the special edge features at the node aggregation. In this case, the fast memory (*e.g.*, shared memory of GPU Stream-Multiprocessors) is not large enough to exploit memory locality. However, improving computation parallelization (*e.g.*, workload partitioning along the embedding dimension) would be more helpful, considering that workloads can be shared among more concurrent threads for improving overall throughput.

We illustrate this aggregation-type difference with the mathematical equations for GCN and GIN. With GCN, the output embedding $\mathbf{X}$ is computed as follows:

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{W}, \tag{2}$$

where $\hat{\mathbf{D}}$ is the diagonal node degree matrix; $\mathbf{W}$ is the weight matrix; $\hat{\mathbf{A}}$ is the graph adjacency matrix. For GIN, the output embedding $\mathbf{X}$ for each layer is computed as follows:

$$\mathbf{x}_i' = h\left((1+\varepsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right) \tag{3}$$

where h denotes a neural network, *e.g.*, an MLP, which maps node features x with input embedding dimension and output

embedding dimension; ε is a configurable/trainable parameter depending on the users' demands or application settings; $\mathcal{N}(i)$ denotes the neighbor IDs of the node i.

Assume we have GCN and GIN with hidden dimension 16, and the input dataset has a node embedding dimension of 128. In the case of GCN, we will first do node update (GEMM[1]-based linear transformation) of the node embedding, thus, at the aggregation, we only need to do aggregation on nodes with hidden dimension 16. In the GIN case, we have to do neighbor aggregation on nodes with 128 dimensions then do node update to linearly transform node embedding from 128 to 16 dimensions. Such an aggregation difference would also lead to different optimization strategies, where GCN would prefer more memory optimization on the small node embeddings while GIN would prefer more computing parallelism on the large node embeddings.

To conclude, the type of aggregation in GNNs should be considered for system-level optimization and it can be obtained by GNNAdvisor's built-in parser of GNN model proprieties.

3.2 Graph Information

Node Degree & Embedding Dimensionality: Real-world graphs generally follow the power-law distribution [50] of node degrees. Such distribution already causes workload imbalance in traditional graph processing systems [18, 25, 32]. In GNN aggregation, such workload imbalance would be exacerbated due to the higher dimensionality of the node embeddings if we perform node-centric workload partitioning. Moreover, node embedding would invalidate some cache-based optimizations that are originally applied to graph processing systems, since caches are usually small in size and insufficient to hold enough nodes with their embeddings. For example, in the graph processing scenarios with a scalar attribute for each node, we can improve performance by putting 16×10^3 nodes on the 64KB L1 cache of each GPU thread block. However, in typical GNNs with a 64-dimension embedding for each node, we can only fit 256 nodes on each GPU block's cache.

With node degree and embedding dimensionality information, new optimization opportunities for GNNs may appear because we can estimate the node's workload and its concrete composition based on such input information. If the workload size is dominated by the number of node neighbors (*e.g.*, large node degree), we may customize the design that could concurrently process more neighbors to increase the computing parallelism among neighbors. On the other hand, if the workload size is dominated by node embedding size (*e.g.*, high-dimensional node embedding), we may consider boosting the computing parallelism along the node embedding dimension. Note that the node degree and embedding dimension information can be extracted based on the loaded graph structure and node embedding vectors. GNNAdvisor manages the GNN workload based on such information (Section 4).

[1] General Matrix-Matrix Multiplication.

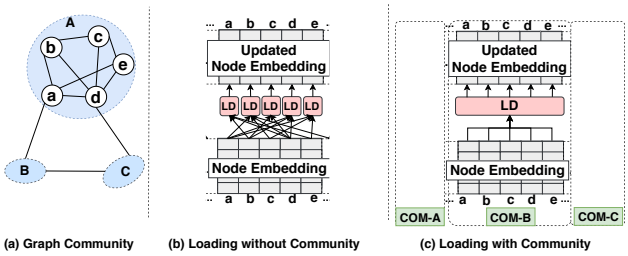

Figure 3: Graph community and its potential benefits. Note that "LD": loading operation. "COM": community.

Graph Community: Graph community [13, 30, 37] is one key feature of real-world graphs, which describes that a small group of nodes tend to hold "strong" intra-group connections (many edges) while maintaining "weak" connections (fewer edges) with the remaining part of the graph. A motivating example of GNN optimization with graph community structure is shown in Figure 3a. Existing node-centric aggregation employed by many graph processing systems [26, 54] is shown in Figure 3b, where each node will first load its neighbors and then do aggregation independently. This strategy can achieve great computation parallelism when each neighbor has a lightweight scalar attribute. In this case, the benefit of loading parallelization would offset the downside of duplicate loading of some shared neighbors. However, in GNN computing where node embedding size is large, this node-centric loading would trigger significant unnecessary memory access since the cost of duplicate neighbor loading is now dominant and not offset by per-node parallelism For example, aggregation of node a, b, c, d, and e would load the embeddings of 15 nodes in total and most of these loads are repeated (both node a and b load the same node d during the aggregation). Such loading redundancy is exacerbated with the increase of embedding dimensionality. On the other side, by considering the community structure of real-world graphs, unnecessary data loading for these "common" neighbors can be well reduced (Figure 3c), where aggregation only requires loads of 5 distinct nodes.

This idea sounds promising, but the effort to realize its benefits on GPUs is non-trivial. Existing approaches [19, 37] of exploiting the graph communities mainly target CPU platforms with a limited number of parallelized threads and MB-level cache sizes for each thread. Their major goal is to exploit the data locality for every single thread. GPUs, on the other side, are equipped with a massive number of parallel threads and KB-level cache sizes per thread. Therefore, the key to exploiting graph community on GPUs is to effectively exploit the data locality among threads by leveraging the L1 cache. Specifically, we need first capture the communities of a graph and then map such locality from input level (node-ID adjacency) to underlying GPU kernels (thread/warp/block-ID

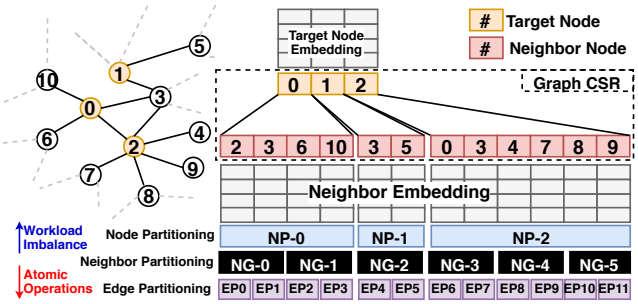

Figure 4: Neighbor Partitioning. Note that "NP": Node Partitioning; "EP": Edge Partitioning; "NG": Neighbor Group.

adjacency). The major hardware-level insight is that threads close in their IDs are more likely to share memory and computing resources, thus, improving the data spatial and temporal locality. GNNAdvisor handles all these details through community-aware node renumbering and GNN-specialized memory optimizations (Section 5).

4 2D Workload Management

GNNs employ a unique space in graph computations, due to the representation of each node by a high-dimensional feature vector (the embedding). GNN workloads grow in two major dimensions: *the number of neighbors* and *the size of the embedding dimension*. GNNAdvisor incorporates an input-driven parameterized 2D workload management tailored for GNNs, including three techniques: *coarse-grained neighbor partitioning*, *fine-grained dimension partitioning*, and *warp-based thread alignment*.

4.1 Coarse-grained Neighbor Partitioning

Coarse-grained neighbor partitioning is a novel workload balance technique tailored to GNN computing on GPUs. It aims to tackle the challenge of *inter-node workload imbalance* and *redundant atomic operations*.

Specifically, based on the loaded graph compressed-sparse row (CSR) representation, our coarse-grained neighbor partitioning will first break down the neighbors of a node into a set of equal-sized neighbor groups, and treat the aggregation workload of each neighbor group (NG) as the basic workload unit for scheduling. Figure 4 exemplifies an undirected graph and its corresponding neighbor partitioning result. The neighbors of Node-0 are divided into two neighbor groups (NG-0 and NG-1) with a pre-determined group size of 2. Neighbors (Node-3 and Node-5) of Node-1 are covered by NG-2, while the neighbors of Node-2 are spread among NG-{3,4,5}. To support the neighbor group, we introduce two components, the neighbor-partitioning module and the neighbor-partitioning graph store. The former is a lightweight module built on top of the graph loader by partitioning the graph CSR into equal-size groups. Note that each neighbor group only covers the neighbors of one target node for ease of scheduling and synchronization. The neighbor-partitioning graph store maintains the tuple-based meta-data of each neighbor group, including its IDs, starting and ending position of its neighbor nodes in the CSR representation, and the source node. For example, the meta-data of NG-2 will be stored as (2, 1, (4, 6)), where 2 is the neighbor-group ID, 1 is the target node ID, (4, 6) is the index range of the neighbor nodes in CSR.

The benefits of applying the aggregation based on partitioning neighbors are three-fold: 1) compared with the more coarse-grained aggregation based on node/vertex-centric partitioning [26], neighbor partitioning can largely mitigate the size irregularity of the workload units, which would improve GPU occupancy and throughput performance; 2) compared with the more fine-grained edge-centric partitioning (used by existing GNN frameworks, such as PyG [11], for batching and tensorization, and graph processing systems [33, 54] for massive computing parallelization), the neighbor-partitioning solution can avoid the overheads of managing many tiny workload units that might hurt the performance in many ways, such as scheduling overheads and the excessive amount of synchronizations; 3) it introduces a performance-related parameter, **neighbor-group** size (*ngs*), which is used for design parameterization and performance tuning. Neighbor partitioning works at a coarse granularity of individual neighbor nodes. It can largely mitigate the workload imbalance problem for low-dimension settings. For high-dimensional node embeddings, we employ a fine-grained dimension partitioning discussed in the next subsection to further distribute workloads of each neighbor group to threads. Note that when the number of neighbors is not divisible by the neighbor group size, it will raise neighbor-group imbalance. Such irregularity can be amortized by setting the neighbor-group size to a small number (*e.g.*, 3).

4.2 Fine-grained Dimension Partitioning

GNN distinguishes itself from traditional graph algorithms in its computation on the node embedding. To explore the potential acceleration parallelism along this dimension, we leverage a fine-grained dimension partitioning to further distribute the workloads of a neighbor group along the embedding dimension to improve aggregation performance. As shown in Figure 5, the original neighbor-group workloads are evenly distributed to 11 consecutive threads, where each thread manages the aggregation along one dimension independently (*i.e.*, accumulation of all neighbor node embeddings towards the target node embedding). If the dimension size is larger than the number of working threads, more iterations would be required to finish the aggregation.

There are two major reasons for using dimension partitioning. First, it can accommodate a more diverse range of

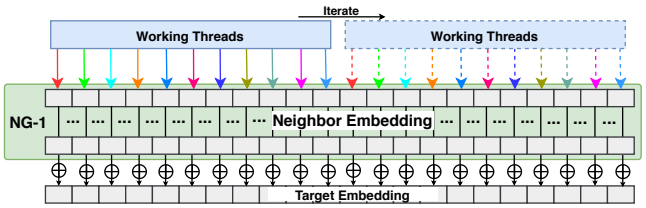

Figure 5: Dimension Partitioning. ⊕: Accumulated add.

embedding dimension sizes. We can either increase the number of concurrent dimension workers or enable more iterations to handle the dimension variation flexibly. This is essential for modern GNNs with increasingly complicated model structures and different sizes of embedding dimension. Second, it introduces another performance-related parameter – the number of working threads (**dimension-worker** (*dw*)) for design customization. The value of this parameter can help to balance the thread-level parallelism and the single thread efficiency (*i.e.*, computation workload per thread).

4.3 Warp-based Thread Alignment

While the above two techniques answer how we balance GNN workloads logically, how to map these workloads to underlying GPU hardware for efficient execution is still unresolved. One straightforward solution is to assign consecutive threads to concurrently process workloads from different neighbor groups (Figure 6a). However, different behaviors (*e.g.*, data manipulation and memory access operations) among these threads would result in thread divergence and GPU underutilization. Threads from the same warp proceed in a single-instruction-multiple-thread (SIMT) fashion and the warp scheduler can only serve one type of instruction per cycle. Therefore, different threads have to wait for their turn for execution until the Stream-Multiprocessor (SM) warp scheduler issues their corresponding instructions.

To tackle this challenge, we introduce a warp-aligned thread mapping in coordination with our neighbor and dimension partitioning to systematically capitalize on the performance benefits of balanced workloads. As shown in Figure 6b, each warp will independently manage the aggregation workload from one neighbor group. Therefore, the execution of different neighbor groups (*e.g.*, NG-0 to NG-5) can be well parallelized without inducing warp divergence. There are several benefits in employing warp-based thread alignment. First, inter-thread synchronization (*e.g.*, atomic operations) can be minimized. Threads of the same warp are working on different dimensions of the same neighbor group, thus no conflicts occur for either global or shared memory accesses by threads from the same warp.

Second, the workload of a single warp is reduced and different warps will process more balanced workloads. Therefore, more small warps can be managed flexibly by SM warp schedulers to improve overall parallelism. Considering the

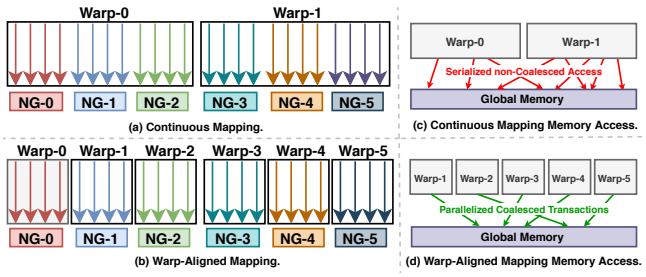

Figure 6: Warp-based Thread Alignment.

unavoidable global memory access of each warp during aggregation, increasing the number of warps can improve SM occupancy to hide latency. Third, memory access can be coalesced. Threads with consecutive IDs from the same warp will access continuous memory addresses in global memory for node embeddings. Therefore, compared with continuous thread mapping (Figure 6c), warp-aligned thread mapping can merge memory requests from the same warp into one global memory transaction (Figure 6d).

5 Specialized Memory Optimization

To further exploit the benefits of 2D workload, we introduce GNN-specialized memory optimizations, *community-aware node renumbering* and *warp-aware memory customization*.

5.1 Community-aware Node Renumbering

To explore the performance benefits of graph community (Section 3.2), we incorporate lightweight node renumbering by reordering node IDs to improve the temporal/spatial locality during GNN aggregation without compromising output correctness. The key idea is that the proximity of node IDs would project to the adjacency of computing units on GPU where they get processed. In GNNAdvisor, our 2D workload management assigns neighbor groups of a node to consecutive warps based on their node ID. If two nodes are assigned with consecutive IDs, their corresponding neighbor groups (warps) would be close to each other in their warp IDs as well. Thus, they are more likely to be scheduled closely on the same GPU SM with a shared L1 cache to improve the data locality on loaded common neighbors. To apply node renumbering effectively, two key questions must be addressed.

When to apply: While graph reordering provides potential benefits for performance, we still need to figure out *what kind of graph would benefit from such reordering optimization*. Our key insight is that for graphs already in a shape approximating block-diagonal pattern in their adjacency matrix (Figure 7a), reordering could not bring more locality benefits, since nodes within each community are already close to each other in terms of their node-IDs. For graphs with a more irregular shape (Figure 7b), where edge connections are distributed among nodes with an irregular pattern, the reordering could

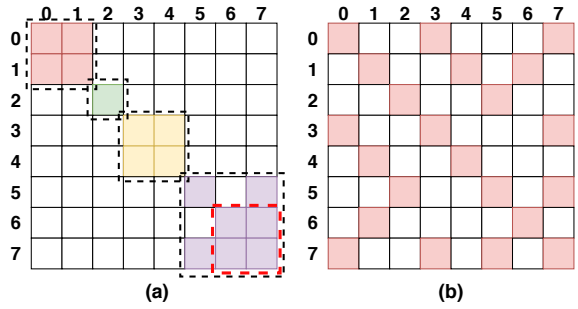

Figure 7: Graph Edge Connection Patterns. Note that each colored square represents the edge between two nodes. Different colors in (a) represent edges from different communities. The red dot-line box indicates the sub-community.

bring notable performance improvement (up to 2× speedup, later discussed in Section 7.5). To this end, we propose a new metric – *Averaged Edge Span* (AES), to determine whether it is beneficial to conduct a graph reordering.

$$\textbf{AES} = \frac{1}{\#E} \sum_{(src_{id}, trg_{id}) \in E} |src_{id} - trg_{id}| \tag{4}$$

where E is the edge set of the graph; $\#E$ is the number of total edges; src_{id} and trg_{id} are the source and target node IDs of each edge. Computing AES is lightweight and can be done on-the-fly during the initial graph loading. Our profiling of a large corpus of graphs also shows that when $\sqrt{AES} > \lfloor \frac{\sqrt{\#N}}{100} \rfloor$ node numbering is more likely to improve runtime performance.

How to apply: We leverage Rabbit Reordering [2], which is a fully parallelized and low-cost graph reordering technique. Specifically, it first maximizes the graph modularity by hierarchically merging edges and clustering nodes. And it then generates node order within each cluster through DFS traversal. Rabbit Reordering has also been proved to outperform other graph clustering approaches [4, 8, 21, 22, 48], including Community-based methods, such as METIS [22], and BFS-based methods, such as Reverse Cuthill-McKee (RCM) [8]) in terms of better quality (data locality) of the captured graph communities, the ease of parallelization, and performance. More importantly, Rabbit Reordering can capture the graph communities hierarchically (*i.e.*, a set of smaller sub-communities are included in a larger community, as exemplified in Figure 7a). Such communities at different levels of granularities would be a good match for the GPU cache hierarchy, where smaller sub-communities (occupying one SM) can enjoy the data locality benefit from the L1 cache, while larger communities (occupying multiple SMs) can enjoy the data locality from the larger L2 cache. We quantitatively discuss such a locality benefit in Section 7.4.

5.2 Warp-aware Memory Customization

Existing works [11, 54] utilize a large number of global memory accesses for reading and writing the embedding and a large number of atomic operations for aggregation (a reduction operation). However, this approach leads to heavy overhead and fails to exploit the potential benefits from shared memory. In particular, when aggregating on a target node with k neighbor groups (each has ngs neighbors with Dim-Dimensional embeddings) into a Dim-dimensional embedding, it involves $O(k \cdot ngs \cdot Dim)$ atomic operations and $O(k \cdot ngs \cdot Dim)$ global memory accesses.

By contrast, we propose a warp-centric shared memory optimization technique. Our key insight is that by customizing shared memory layout according to the block-level warp organization pattern (Figure 7), we can significantly reduce the number of atomic operations and global memory access. First of all, we reserve a shared memory space ($4 \times Dim$ bytes for floating-point embeddings) for the target node of each neighbor group (warp), such that the threads from a warp can cache the intermediate results of reduction in shared memory. Later on, within a thread block, we designate only one warp (called *leader*) for copying the intermediate results of each target node to global memory considering that neighbors of each node can be spread across different warps. The detailed customization procedure is described in Algorithm 1. Specifically, each warp (maintained in *warpPtr*) has three properties: *nodeSharedAddr* (a shared memory address for the aggregation result of a neighbor-group), *nodeID* (the ID of the target node), and *leader* (a boolean flag indicating whether the current warp is a leader warp for flushing out the result from the shared memory to the global memory). The major customization routine (Line 4 to Line 22) handles different warps based on their index position relative to thread blocks. Note that such a shared memory customization is low-cost and is done only once on-the-fly with the regular graph initialization process before the GPU kernel execution.

In our design, when a target node with k neighbor groups (each has ngs neighbors with Dim-dimensional embeddings), it involves $O(Dim)$ atomic operations and $O(Dim)$ global memory accesses. To this end, we can save the atomic operations and global memory access by $(k \cdot ngs)\times$, thus significantly accelerating the aggregation operations. Here, we treat ngs as a hyper-parameter to balance memory access efficiency and computation parallelism, and we further discuss its value selection in Section 6.

6 Design Optimization

The parameters in our GPU kernel configurations can be tuned to accommodate various GNN models with graph data sets. But it is not yet known how to automatically select the parameters which can deliver the optimal performance. In this section, we introduce the analytical model and the auto parameter selection in the **`Decider`** of GNNAdvisor.

Analytical Modeling: The performance/resource analytical model of GNNAdvisor has two variables, workload per

Algorithm 1 Warp-aware Memory Customization.

```
▷ Compute #neighbor-groups (#warps).
warpNum = neighborGroups = computeGroups(ngs);
▷ Compute the number of warps per thread block.
warpPerBlock = floor(threadPerBlock/threadPerWarp)
▷ Initialize tracking variables.
cnt = 0; local_cnt = 0; last = 0;
while cnt < warpNum do
    ▷ Warp in the front of a thread block.
    if cnt % warpPerBlock == 0 then
        warpPtr[cnt].nodeSharedAddr = local_cnt × Dim;
        last = warpPtr[cnt].nodeID;
        warpPtr[cnt].leader = true;
    ▷ Warp in the middle of a thread block.
    else
        ▷ Warp with the same target node as
          its predecessor warp.
        if warpPtr[cnt].nodeID == last) then
            warpPtr[cnt].nodeSharedAddr = local_cnt;
        ▷ Warp with the different target node as
          its predecessor warp.
        else
            local_cnt++;
            warpPtr[cnt].nodeSharedAddr = local_cnt;
            last = warpPtr[cnt].nodeID;
            warpPtr[cnt].leader = true;
        end if
    end if
    ▷ Next warp belongs to a new thread block.
    if (++cnt)%warpPerBlock == 0 then
        local_cnt = 0;
    end if
end while
```

thread (*WPT*), and shared memory usage per block (*SMEM*).

$$\textbf{WPT} = ngs \times \frac{Dim}{dw}, \quad \textbf{SMEM} = \frac{tpb}{tpw} \times Dim \times FloatS \tag{5}$$

where *ngs* and *dw* is the neighbor-group and dimension-worker size (Section 4.2), respectively; *Dim* is the node embedding dimension; *IntS* and *FloatS* are both 4-byte on GPUs; *tpb* is the thread-per-block and *tpw* is the thread-per-warp; *tpw* is 32 for GPUs, while *tpb* is selected by users.

Parameter Auto Selection: To determine the value of the *ngs* and *dw*, we follow two steps. First, we determine the value of *dw* based on *tpw* (hardware constraint) and *Dim* (input property), as shown in Equation 6. Note that we develop this equation by profiling different datasets and GNN models.

$$dw = \begin{cases} tpw & Dim \geq tpw \\ \frac{tpw}{2} & Dim < tpw \end{cases} \tag{6}$$

Second, we determine the value of *ngs* based on the selected *dw* and the user-specified *tpb*. The constraints include making $WPT \approx 1024$ and $SMEM \leq SMEMperBlock$. Note that $SMEMperBlock$ is 48KB to 96KB on modern GPUs [42, 44]. Across different GPUs, even though the number of CUDA cores and global memory bandwidth would be different, the single-thread workload capacity (measured by *WPT*) remains similar. *tpb* is usually chosen as a power of 2 but less than or equal 1024. Our insight based on micro-benchmarking and previous literature [56] shows that smaller blocks (1 to 4 warps, *i.e.*, $32 \leq tpb \leq 128$) can improve SM warp scheduling flexibility and avoid tail effects, thus leading to higher GPU occupancy and throughput. We further demonstrate the effectiveness of our analytical model in Section 7.5.

7 Evaluation

In this section, we comprehensively evaluate GNNAdvisor in terms of the performance and adaptability on various GNN models, graph datasets, and GPUs.

7.1 Experiment Setup

Benchmarks: We choose the two most representative GNN models widely used by previous work [11, 36, 53] on node classification tasks to cover different types of aggregation. 1) Graph Convolutional Network (GCN) [27] is one of the most popular GNN model architectures. It is also the key backbone network for many other GNNs, such as GraphSAGE [17], and differentiable pooling (Diffpool) [57]. Therefore, improving the performance of GCN will also benefit a broad range of GNNs. For GCN evaluation, we use the setting: *2 layers with 16 hidden dimensions*, which is also the setting from the original paper [27]. 2) Graph Isomorphism Network (GIN) [55]. GIN differs from GCN in its aggregation function, which weighs the node embedding values from the node itself. In addition, GIN is also the reference architecture for many other advanced GNNs with more edge properties, such as Graph Attention Network (GAT) [52]. For GIN evaluation, we use the setting: *5 layers with 64 hidden dimensions*, which is the setting used in the original paper [55].

Baselines: we choose several baseline implementations for comparison. 1) Deep Graph Library (DGL) [53] is the state-of-the-art GNN framework on GPUs, which is built upon the famous tensor-oriented platform – Pytorch [46]. DGL significantly outperforms the other existing GNN frameworks [11] over various datasets on many mainstream GNN architectures. Therefore, we make an in-depth comparison with DGL in our evaluation; 2) Pytorch-Geometric (PyG) [11] is another GNN framework in which users can define their edge convolutions when building customized GNN aggregation layers; 3) NeuGraph [36] is a dataflow-centered GNN system on GPUs built on Tensorflow [1]; 4) Gunrock [54] is the GPU-based graph processing framework with state-of-the-art performance on traditional graph algorithms (*e.g.*, PageRank).

Datasets: We cover all three types of datasets, which have been used in previous GNN-related work [11, 36, 53]. Type I graphs are the typical datasets used by previous GNN algorithm papers [17, 27, 55]. They are usually small in the number of nodes and edges, but rich in node embedding information with high dimensionality. Type II graphs [24] are the popular benchmark datasets for graph kernels and are selected as the built-in datasets for PyG [11]. Each dataset consists of a set of

Table 1: Datasets for Evaluation.

Type	Dataset	#Vertex	#Edge	Dim.	#Class
I	Citeseer	3,327	9,464	3,703	6
	Cora	2,708	10,858	1,433	7
	Pubmed	19,717	88,676	500	3
	PPI	56,944	818,716	50	121
II	PROTEINS_full	43,471	162,088	29	2
	OVCAR-8H	1,890,931	3,946,402	66	2
	Yeast	1,714,644	3,636,546	74	2
	DD	334,925	1,686,092	89	2
	TWITTER-Partial	580,768	1,435,116	1,323	2
	SW-620H	1,889,971	3,944,206	66	2
III	amazon0505	410,236	4,878,875	96	22
	artist	50,515	1,638,396	100	12
	com-amazon	334,863	1,851,744	96	22
	soc-BlogCatalog	88,784	2,093,195	128	39
	amazon0601	403,394	3,387,388	96	22

small graphs, which only have intra-graph edge connections without inter-graph edge connections. Type III graphs [27,31] are large in terms of the number of nodes and edges. These graphs demonstrate high irregularity in structure, which is challenging for most of the existing GNN frameworks. Details of these datasets are listed in Table 1.

Platforms & Metrics: We implement GNNAdvisor's backend with C++ and CUDA C and its front-end with Python. Our major evaluation platform is a server with an 8-core 16-thread Intel Xeon Silver 4110 CPU [20] and a Quadro P6000 [42] GPU. Besides, we use Tesla V100 [44] GPU on the DGX-1 system [40] to demonstrate the generality of GNNAdvisor. Runtime parameters of different input settings are optimized by GNNAdvisor `Decider`. To measure the performance speedup, we calculate the averaged latency of 200 end-to-end inference (forward propagation) or training (forward+backward propagation).

7.2 Compared with DGL

In this section, we first conduct a detailed experimental analysis and comparison with DGL on GNN inference, then extend our comparison for GNN training. As shown in Figure 8, GNNAdvisor achieves 4.03× and 2.02× speedup on average compared to DGL [53] over three types of datasets for GCN and GIN on inference, respectively. We next provide detailed analysis and give insights for each type of datasets.

Type I Graphs: The performance improvement against DGL is significantly higher for GCN (on average 6.45×) than GIN (on average 1.17×). The major reason is their different GNN computation patterns. For GCN, node dimension reduction (DGEMM) is always placed before aggregation. This largely reduce data movement and thread synchronization overheads during the aggregation phase, which could gain more benefits from GNNAdvisor's 2D workload management and specialized memory optimization for data locality improvements. GIN, on the other side, has aggregation phase that must be finished before the node dimension reduction. Thus, it cannot avoid high-volume memory access and data

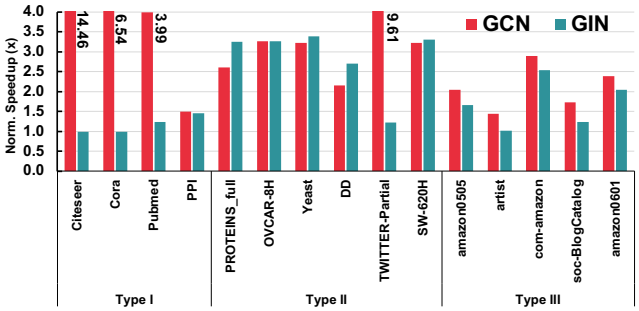

Figure 8: Inference speedup (×) over DGL on GCN and GIN.

movements during the aggregation phase. Therefore, it gets lower benefits from the data locality and the shared memory on GPUs for fast and low-overhead memory access. However, our fine-grained dimension partitioning can still handle these high-dimensional cases effectively.

Type II Graphs: Performance shows less difference between GCN (4.02×) and GIN (2.86×) on the same datasets except for *TWITTER-Partial*, which has the highest node embedding dimension (1323) in Type II graphs. It is worth noticing that the speedup for GIN is consistently better compared with Type I. There are two major reasons: 1) node feature dimension is much lower (average 66.5, excluding *TWITTER-Partial*) versus Type I (average 1421), which can gain more performance benefits from data spatial and temporal locality of our specialized memory optimizations; 2) Type II graphs intrinsically have good locality in their graph structure. The reason is that Type II datasets consist of small graphs with very dense intra-graph connections but no inter-graph edges, plus nodes within each small graph are assigned with consecutive IDs. Therefore, the performance gains of such graph-structure locality can be scaled up when combining with GNNAdvisor's efficient workload and memory optimizations.

Type III Graphs: The speedup is also evident (average 2.10× for GCN and average 1.70× for GIN) on graphs with a large number of nodes and edges, such as *amazon0505*. The reason is the high overhead inter-thread synchronization and global memory access can be well reduced through our 2D workload management and specialized memory optimization. Besides, our community-aware node renumbering further facilitates an efficient workload sharing among adjacent threads (working on a group of nodes) through improving the data spatial/temporal locality. On the dataset *artist*, which has the smallest number of nodes and edges within Type III, we notice a lower performance speedup for GIN. And we find that the *artist* dataset has the highest standard deviation of graph community sizes within Type III graphs, which makes it challenging to 1) use the group community information to capture the node temporal and spatial locality in the GNN aggregation phase, and 2) capitalize on the performance benefits of using such a community structure for guiding system-level optimizations (*e.g.*, warp-aligned thread mapping and shared memory customization) on GPUs, which have a fixed number

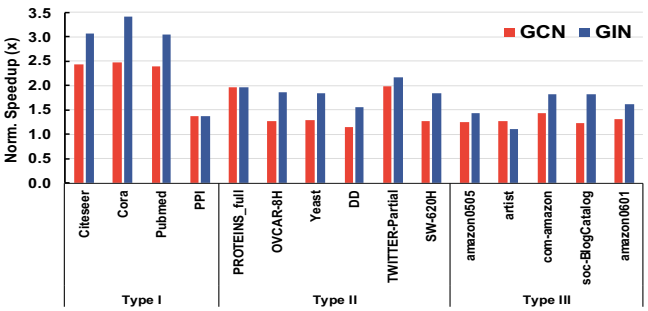

Figure 9: Training speedup (×) over DGL on GCN and GIN.

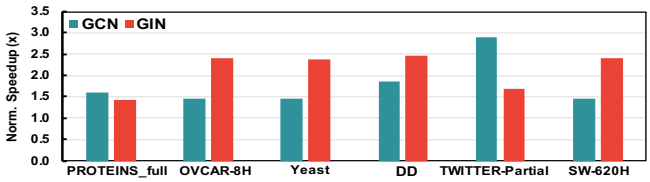

Figure 10: Training speedup (×) over PyG on GCN and GIN.

Table 2: Latency (ms) comparison with NeuGraph (NeuG).

Dataset	NeuG (ms)	Ours (ms)	Speedup
reddit-full	2460	599.69	4.10×
enwiki	1770	443.00	3.99×
amazon	1180	474.57	2.48×

of computation and memory units within each block/SM.

Kernel Metrics: For detailed kernel metrics analysis, we utilize NVProf [41] to measure two performance-critical (computation and memory) CUDA kernel metrics: *Stream Processor (SM) efficiency* and *Cache (L1 + L2 + Texture) Hit Rate*. GNNAdvisor achieves on average 24.47% and 12.02% higher SM efficiency compared with DGL for GCN and GIN, respectively, which indicates that our 2D workload management can strike a good balance between the single-thread efficiency and the multi-thread parallelism that are crucial to the overall performance improvement. GNNAdvisor achieves on average 75.55% and 126.20% better cache hit rate compared with DGL for GCN and GIN, correspondingly, which demonstrates the benefit of specialized memory optimizations.

Training Support: We also evaluate the training performance of GNNAdvisor on all three types of datasets compared with the DGL on both GCN and GIN. Compared with inference, training is more challenging, since it involves more intensive computation with the forward value propagation and the backward gradient propagation, both of which heavily rely on the underlying graph aggregation kernel for computation. As shown in Figure 9, GNNAdvisor consistently outperforms the DGL framework with average 1.61× and average 2.00× speedup on GCN and GIN, respectively, which shows the strength of our input-driven optimizations. The key difference between training and inference of GNNs is two-fold: First, backpropagation is needed in training. This step benefits from our improvements, as the backpropagation step is similar to the forward computation during the inference, and all the proposed methods are still beneficial; Second, training incurs extra memory and data movement overheads for storing/accessing the activations of the forward pass until gradients can be propagated back.

7.3 Compared with other Frameworks

We compare with DGL on all input settings, since DGL is the overall best-performance GNN framework. In this section, we further compare GNNAdvisor with three other representative GNN computing frameworks on their best settings.

Compared with PyG: As shown in Figure 10, GNNAdvisor can outperform PyG with 1.78× and 2.13× speedup on average for GCN and GIN, respectively. For GCN, GNNAdvisor achieves significant speedup on datasets with high-dimensional node embedding, such as *TWITTER-Partial*, through 1) node dimension reduction before aggregation and 2) workload sharing among neighbor partitions and dimension partitions. For GIN, GNNAdvisor reaches 2.45× speedup on datasets with a higher average degree, such as *DD*, since GNNAdvisor can effectively distribute the workload of each node along their embedding dimension to working threads while balancing the single-thread efficiency and inter-thread parallelism. PyG, however, achieves inferior performance because 1) it has poor thread management in balancing workload and controlling synchronization overhead; 2) it heavily relies on the scatter-and-gather kernel, which lacks flexibility.

Compared with NeuGraph: For a fair end-to-end training comparison with NeuGraph that has not open-sourced its implementation and datasets, we 1) use the GPU (Quadro P6000 [42]) that is comparable with the GPU of NeuGraph (Tesla P100 [43]) in performance-critical factors, such as GPU architecture (both have the Pascal architecture) and the number of CUDA cores; 2) use the same set of inputs as NeuGraph on the same GNN architecture [36]; 3) use the datasets that are presented in their paper and are also publicly available. As shown in Table 2, GNNAdvisor outperforms NeuGraph with a significant amount of margin (1.3× to 7.2× speedup) in terms of computation and memory performance. NeuGraph relies on general GPU kernel optimizations and largely ignores the input information. Moreover, the optimizations in NeuGraph are built-in and fixed inside the framework without performance tuning flexibility. In contrast, GNNAdvisor leverages GNN-featured GPU optimizations and demonstrates the key contribution of input insights for system optimizations.

Compared with Gunrock: We make a performance comparison between GNNAdvisor and Gunrock [54] on a single neighbor aggregation kernel of GNNs (*i.e.*, the Sparse-Matrix Dense-Matrix Multiplication (SpMM)) over the Type III graphs. As shown in Figure 11, GNNAdvisor outperforms Gunrock with 2.89× to 8.41× speedup. There are two major reasons behind such a evident performance improvement on the sparse GNN computation: 1) Gunrock focuses on graph-

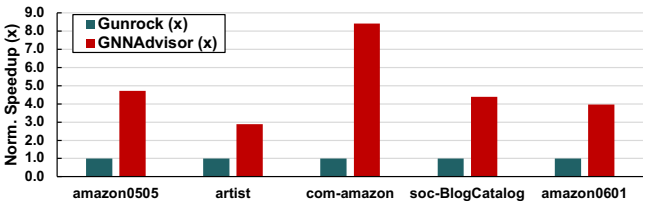

Figure 11: Speedup (×) comparison with Gunrock.

algorithm operators (*e.g.*, frontier processing) but lacks efficient support for handling high-dimensional node embedding; 2) Gunrock leverages generic optimizations without considering the input differences, thus, losing the adaptability for handling different GNN inputs efficiently.

7.4 Optimization Analysis

In this section, we explore and analyze the optimizations used in Sections 4 and 5 in detail.

Neighbor partitioning: From Figure 12a, we can see that with the increase of the neighbor-group size, the running time of GNNAdvisor will first decrease. The increase of the neighbor-group size saturates the computation capability of each thread meanwhile improving the data locality and reducing the number of atomic operations (*i.e.*, inter-thread synchronization overhead). However, when the neighbor-group size becomes larger than a certain threshold (*e.g.*, 32 for the *artist* dataset), each thread reaches its computation capacity upper bound, and further increasing the neighbor-group size offers no more performance benefit instead increases the overall latency.

Dimension partitioning: As shown in Figure 12b, the dimension worker impact is more evident in performance compared with the neighbor-group size at the range from 1 to 16. When the number of dimension worker increases from 16 to 32, the runtime performance shows very minor difference due to the already balanced single-worker efficiency and multi-worker parallelism. Therefore, further increase the number of dimension workers brings no more benefits.

Node renumbering: We demonstrate the benefit of node renumbering by profiling Type III datasets for GCN and GIN. As shown in Figure 12c, renumbering nodes within a graph can bring up to 1.74× and 1.49× speedup for GCN and GIN, respectively. The major reason is that our community-aware node renumbering can increase the data spatial and temporal locality during GNN aggregation.

To quantify such locality benefits, we extract the detailed GPU kernel metric – memory access in terms of read and write bytes from DRAM for illustration. Our CUDA kernel metric profiling results show that node renumbering can effectively reduce the memory access overhead (on average 40.62% for GCN and 42.33% for GIN) during the runtime since more loaded node embeddings are likely to be shared among the nodes with consecutive IDs. We also notice one input case that benefits less from our optimization – *artist*, since 1) the community size inside *artist* displays a large variation (high standard deviation), making it challenging to capture the neighboring adjacency and locality; 2) such a variation hurdles system-level (computation and memory) optimizations to effectively capitalize on the locality benefits of renumbering.

Block-level optimization: We show the optimization benefits of our block-level optimization (including warp-aligned thread mapping, and warp-aware shared memory customization). We analyze two kernel metrics (*atomic operations reduction* and *DRAM access reduction*) on three large graphs for illustration. As shown in Figure 12d, GNNAdvisor can effectively reduce the atomic operations and DRAM memory access by an average 47.85% and 57.93%. This result demonstrates 1) warp-aligned thread mapping based on neighbor partitioning can effectively reduce a large portion of atomic operations; 2) warp-aware shared memory customization can avoid a significant amount of global memory access.

7.5 Additional Studies

Hidden dimensions of GNN: In this experiment, we analyze the impact of the GNN architecture in terms of the size of the hidden dimension for GCN and GIN. As shown in Figure 13a, we observe that with the increase of hidden dimension of GCN, the running time of GNNAdvisor is also increased due to more computation (*e.g.*, additions) and memory operations (*e.g.*, data movements) during the aggregation phase and a larger size of the node embedding matrix during the node update phase. Meanwhile, we also notice that GIN shows a larger latency increase versus GCN, mainly because of the number of layers (2-layer GCN *vs.* 5-layer GIN) that make such a difference more pronounced.

Overhead analysis: Community-aware node renumbering is the major source of overhead for leveraging GNN input information, and other parts are negligible. Here as a case study, we evaluate its overhead on the training phase of GCN on Type III graphs, given the optimization decision from our GNNAdvisor `Decider` (as discussed in Section 5). Here we use training for illustration; inference in a real GNN application setting would also use the same graph structure many times [17, 27, 27] with different node embeddings inputs. As shown in Figure 13b, node-renumbering overhead is consistently small (average 4.00%) compared with overall training time. We thus conclude that such one-time overhead can be amortized over GNN running time, which demonstrates its applicability in real-world GNN applications.

Performance on Tesla V100: To demonstrate the potential of GNNAdvisor in the modern data-center environment, we showcase the performance of GNNAdvisor on an enterprise-level GPU – Tesla V100 [44]. As shown in Figure 13c, GNNAdvisor can scale well towards such a high-end device, which can achieve 1.97× and 1.86× speedup compared with P6000 for GCN and GIN due to more computa-

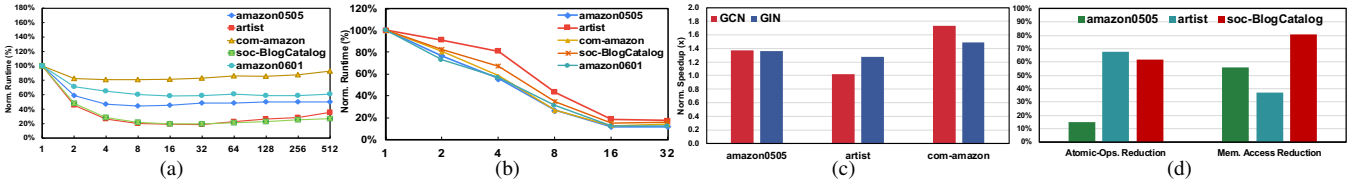

Figure 12: Optimization Analysis. (a) Normalized latency as the neighbor group size (*ngs*) grows (latency at $ngs = 1$ is set as 100%); (b) Normalized latency as the number of dimension workers grows (latency at $dw = 1$ is set as 100%); (c) Normalized speedup when using node renumbering compared to without renumbering; (d) Normalized GPU kernel metrics when using block-level optimizations compared to without block-level optimizations.

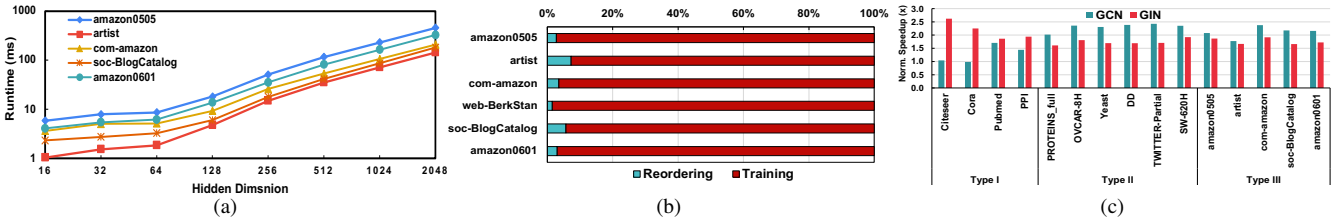

Figure 13: Additional Studies. (a) Latency (*ms*) analysis as the hidden dimension grows on GCN; (b) Overhead (%) analysis for node renumbering; (c) Speedup (×) on Tesla V100 over Quadro P6000 (set as 1×).

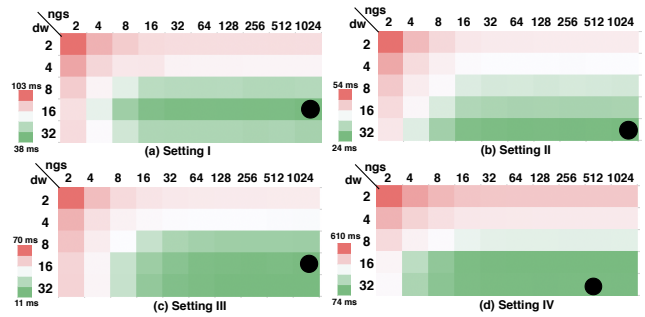

Figure 14: Parameter Selection for Four Settings. Note that the solid-black dot indicates the parameter (*dw* and *ngs*) selected by GNNAdvisor **`Decider`** based on analytical modeling.

tion resources (*e.g.*, 2.6× SMs, and 1.33× CUDA cores, and 1.13× throughput performance) and higher memory bandwidth (*e.g.*, 2.08× peak memory bandwidth). This comparison shows that GNNAdvisor well adapts towards more advanced GPU hardware for seeking better performance. We also foresee that our current work of GNNAdvisor can be extended to the multi-GPU or distributed data center, benefiting overall performance by improving single GPU efficiency.

Parameter selection: To show the effectiveness of our analytical modeling in kernel parameter selection, we consider four different settings: I: *amazon0505* on GCN at P6000 GPU as our base setting; II: *amazon0505* GCN on V100 to demonstrate device adaptation; III: *amazon0505* and *soc-BlogCatalog* on P6000 to demonstrate adaptation to different datasets; IV: *amazon0505* on GIN at P6000 to demonstrate adaptation to a different GNN model architectures. As shown in Figure 14, our parameter selection strategy can pinpoint the optimal low-latency design for the above four settings. This demonstrates the effectiveness of our analytical modeling in assisting parameter selection to optimize the performance of GNN computation.

8 Conclusion

In this work, we propose, GNNAdvisor, an adaptive and efficient runtime system for GNN acceleration on GPUs. Specifically, we explore the potential of GNN input-level information in guiding system-level optimizations. We further propose a set of GNN-tailored system-level optimizations (*e.g.*, 2D workload management, and specialized memory optimizations) and incorporate them into our parameterized designs to improve performance and adaptability. Extensive experiments on a wide range of datasets and mainstream GNN models demonstrate the effectiveness of our design. Overall, GNNAdvisor provides users a handy tool to accelerate GNNs on GPUs systematically and comprehensively.

9 Acknowledgment

We would like to thank our shepherd, Petros Maniatis, and the anonymous OSDI reviewers. This work was supported in part by NSF 1925717. Use was made of computational facilities purchased with funds from the National Science Foundation (OAC-1925717) and administered by the Center for Scientific Computing (CSC). The CSC is supported by the California NanoSystems Institute and the Materials Research Science and Engineering Center (MRSEC; NSF DMR 1720256) at UC Santa Barbara.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI)*, 2016.

[2] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[3] Vignesh Balaji and Brandon Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web (WWW)*, 2011.

[5] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2013.

[6] Hsinchun Chen, Xin Li, and Zan Huang. Link prediction approach to collaborative filtering. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL)*. IEEE, 2005.

[7] De Cheng, Yihong Gong, Xiaojun Chang, Weiwei Shi, Alexander Hauptmann, and Nanning Zheng. Deep feature learning via structured graph laplacian embedding for person re-identification. *Pattern Recognition*, 2018.

[8] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, 1969.

[9] Alberto Garcia Duran and Mathias Niepert. Learning graph representations with embedding propagation. In *Advances in neural information processing systems (NeurIPS)*, 2017.

[10] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. *arXiv preprint*, 2015.

[11] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds (ICLR)*, 2019.

[12] Matthias Fey and Jan E. Lenssen. Pytorch extension library of optimized scatter operations, 2019.

[13] Santo Fortunato. Community detection in graphs. *Physics reports*, 2010.

[14] Jaume Gibert, Ernest Valveny, and Horst Bunke. Graph embedding in vector spaces by node attribute statistics. *Pattern Recognition*, 2012.

[15] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[16] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM international conference on Knowledge discovery and data mining (SIGKDD)*, 2016.

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

[18] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *The VLDB Endowment*, 2014.

[19] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 2000.

[20] Intel. Xeon sliver 4110. `https://ark.intel.com/content/www/us/en/ark/products/123547/intel-xeon-silver-4110-processor-11m-cache\-2-10-ghz.html`.

[21] Konstantinos I Karantasis, Andrew Lenharth, Donald Nguyen, Mara J Garzaran, and Keshav Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[22] George Karypis and Vipin Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. `http://www.cs.umn.edu/~metis`, 2009.

[23] Riesen Kaspar and Bunke Horst. *Graph classification and clustering based on vector space embedding*. World Scientific, 2010.

[24] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016.

[25] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[26] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (HPDC)*, 2014.

[27] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 2017.

[28] Jérôme Kunegis and Andreas Lommatzsch. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009.

[29] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[30] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 2008.

[31] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[32] Hang Liu and H Howie Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[33] Hang Liu and H Howie Huang. Simd-x: Programming and processing of graph algorithms on gpus. In *USENIX Annual Technical Conference (ATC)*, 2019.

[34] Dijun Luo, Chris Ding, Heng Huang, and Tao Li. Non-negative laplacian embedding. In *Ninth IEEE International Conference on Data Mining (ICDM)*, 2009.

[35] Dijun Luo, Feiping Nie, Heng Huang, and Chris H Ding. Cauchy graph embedding. In *The 28th International Conference on Machine Learning (ICML)*, 2011.

[36] 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 *USENIX Annual Technical Conference (ATC'19)*.

[37] Mark EJ Newman. Spectral methods for community detection and graph partitioning. *Physical Review E*, 2013.

[38] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[39] Nvidia. Cuda sparse matrix library (cusparse). `developer.nvidia.com/cusparse`.

[40] Nvidia. Dgx-1. `https://www.nvidia.com/en-us/data-center/dgx-1/`.

[41] Nvidia. Profiling tools. `docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[42] Nvidia. Quardo p6000 gpu. `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-pascal-p6000-data-sheet-us-nv\-704590-r1.pdf`.

[43] Nvidia. Tesla p100. `https://www.nvidia.com/en-us/data-center/tesla-p100/`.

[44] Nvidia. Tesla v100. `https://www.nvidia.com/en-us/data-center/v100/`.

[45] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, 1999.

[46] 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 *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[47] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *The 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2014.

[48] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 2007.

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

[50] Alessandra Sala, Haitao Zheng, Ben Y. Zhao, Sabrina Gaito, and Gian Paolo Rossi. Brief announcement: Revisiting the power-law degree distribution for social graph analysis. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2010.

[51] Tomasz Tylenda, Ralitsa Angelova, and Srikanta Bedathur. Towards time-aware link prediction in evolving social networks. In *Proceedings of the 3rd workshop on social network mining and analysis*, 2009.

[52] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[53] 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. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[54] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 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 (PPoPP)*, 2016.

[55] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

[56] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*, 2018.

[57] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *The 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

A Artifact Appendix

Abstract Summary

GNNAdvisor is an efficient and adaptive runtime system for GNN computing on GPUs. GNNAdvisor consists of two parts. The first part is the host-side CPU program. It is responsible for dataset loading, runtime configuration generation, and invoking the GPU-side program. The second part is the device-side GPU program. It is responsible for the major computation of the GNN model on sparse neighbor-aggregation and dense node-update phase. GNNAdvisor improves the performance of GNN computing with its highly configurable and efficient 2D workload management and specialized memory design. Moreover, the runtime configuration generation on the host-side CPU program makes GNNAdvisor more adaptive towards various kinds of input settings.

Artifact Checklist

- **Link:** `github.com/YukeWang96/OSDI21_AE.git`.
- **Hardware**:
  - Intel CPU x86_64 with host memory >= 32GB. Tested on Intel Xeon Silver 4110 (8-core 16-thread) CPU with 64GB host memory.
  - NVIDIA GPU (arch>=*sm*_60) with devcie memory >= 16GB. Tested on NVIDIA Quadro P6000 (*sm*_61), Tesla V100 (*sm*_70), and RTX3090 (*sm*_86). Note that upon creating this artifact, we mainly evaluate our design on RTX3090. The execution time may be different across different devices but the overall trend of performance (speedup) is similar.
- **OS & Compiler**: Ubuntu 16.04+, GCC 7.5+, CMAKE 3.14+, CUDA 10.2+.

Environment Setup

Step-1: Setup the basic environment. Two options:

- Setup the environment via Docker (**Recommended**).
- Setup via conda and pip.

Details of the above two options can be found in `README.md`.

Step-2: Install GNNAdvisor Pytorch Binding.

- Go to `GNNAdvisor/GNNConv`, then `python setup.py install` to install the `GNNAdvisor` modules.
- Go to `rabbit_module/src`, then `python setup.py install` to install the rabbit reordering modules.

Step-3: Download the graph datasets. Our preprocessed graph datasets in `.npy` format can be downloaded via this link [2] (filename: `osdi-ae-graphs.tar.gz`). Unzip the graph datasets `tar -zxvf osdi-ae-graphs.tar.gz` at the project root directory. Note that node initial embedding is not included, and we generate an all 1s embedding matrix according to users input dimension parameter at the runtime for just performance evaluation.

Experiments

- Running DGL baseline on GNN training (Figure 9).
  - Go to `dgl_baseline/` directory.
  - `./0_run_gcn.sh` and `./0_run_gin.sh` to run DGL and generate `.csv` result for GCN and GIN.
- Running PyG baseline on GNN training (Figure 10).
  - Go to `pyg_baseline/` directory.
  - `./0_run_gcn.sh` and `./0_run_gin.sh` to run PyG and generate `.csv` result for GCN and GIN.
- Running Gunrock for single SpMM (neighbor aggregation) kernel.
  - Go to `Gunrock/` call `./build_spmm.sh`.
  - `./0_bench_Gunrock.py` for profile `spmm`.
- Running GNNAdvisor (Figure 9 and 10).
  - Go to `GNNAdvisor/` directory.
  - `./0_run_gcn.sh` and `./0_run_gin.sh` to run GNNAdvisor and generate `.csv` for GCN/GIN.
- Running some additional studies (Figure 11(a,b,c), and 12(a)). Detailed commands of running all these studies can be found in `README.md`.

Note that accuracy evaluation are omitted for all implementations and each sparse kernels are tested via the unitest.py We focus on the training evaluation of the GNNs, and the reported time per epoch only includes the GNN model forward and backward computation, excluding the data loading and some preprocessing. Since the paper draft submission and the creation of this artifact, DGL has update several of its kernel library (from v0.52 to v0.60). In this comparion we focus on the latest DGL version (v0.60). Based on our profiling on RTX3090 and Quadro P6000, our design would show minor speedup on the simple GCN model (2-layer and 16 hidden dimension), but show more evident speedup on more complicated GIN model (5-layer and 64 hidden dimension), which can still demonstrate the effectiveness of our optimizations. Our observation is that on small Type I graphs, our frameworks achieve significant speedup for both GCN and GIN model on RTX3090 and Quadro P6000. On larger Type II and Type III datasets, our GIN model implementation would show more evident speedups.

[2] https://bit.ly/3ys86a5