Understanding and Bridging the Gaps in Current GNN Performance Optimizations

Kezhao Huang
Tsinghua University
hkz20@mails.tsinghua.edu.cn

Jidong Zhai
Tsinghua University
zhaijidong@tsinghua.edu.cn

Zhen Zheng
Alibaba Group
james.zz@alibaba-inc.com

Youngmin Yi
University of Seoul
ymyi@uos.ac.kr

Xipeng Shen
North Carolina State University
xshen5@ncsu.edu

Abstract

Graph Neural Network (GNN) has recently drawn a rapid increase of interest in many domains for its effectiveness in learning over graphs. Maximizing its performance is essential for many tasks, but remains preliminarily understood. In this work, we provide an in-depth examination of the state-of-the-art GNN frameworks, revealing five major gaps in the current frameworks in optimizing GNN performance, especially in handling the special complexities of GNN over traditional graph or DNN operations. Based on the insights, we put together a set of optimizations to fill the gaps. These optimizations leverage the state-of-the-art GPU optimization techniques and tailor them to the special properties of GNN. Experimental results show that these optimizations achieve 1.37×−15.5× performance improvement over the state-of-the-art frameworks on various GNN models.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; • **Computer systems organization** → *Single instruction, multiple data.*

Keywords: GNN, Performance Optimizations, Parallelism

1 Introduction

In the last several years, Graph Neural Network (GNN) has emerged as a dominant approach to learn from graphs. It holds state-of-the-art performance across a wide range of prediction tasks on graphs, such as node classification [38], graph classification [45], and link prediction [41]. At the same time, GNN outperforms other methods (e.g., DeepWalk [37], Node2Vec [9]) in graph representation tasks, providing downstream tasks with better node representation. As a result, GNN holds more than 90% of leading positions in graph tasks in Open Graph Benchmark (OGB) [12] and CogDL [2]. Its influence has spanned across many domains, from biology to medicine [8], social networks, personal recommendations [10], knowledge graph processing [39], and so on.

GNN performance (speed) is essential for many of its applications. The importance is amplified by current trends as researchers find that deeper GNN models bring better accuracy [26, 28], and move their attention from small graph datasets to larger datasets [12] to model real-world problems better. The limited computing efficiency of current GNN frameworks has been restraining GNN applications and research. Compromises have to be made on algorithms and hence the resulting accuracy [3, 23, 49].

Although GNN consists of primarily graph operations and neural operations, simply putting graph computing frameworks and DNN frameworks together is insufficient for supporting efficient GNN executions, as it cannot efficiently handle complex interactions between graph operations and neural operations, which is the key to GNN performance. For instance, graph operations and DNN layers interleave in GNN, the dependences complicate kernel fusion, memory performance optimization, scheduling, and load balance. Some GNN models even perform neural operations following graph structure, making it harder to achieve high efficiency due to the mismatch of the dense computational property of neural operations and the sparse patterns in graph operations.

The needs have prompted a strand of recent efforts in developing GNN programming frameworks, such as Pytorch-Geometric (PyG) [5], Deep Graph Library (DGL) [42], NeuGraph [29] AliGraph [52], ROC [17], and G^3 [27]. Although these frameworks have shown some promising advancements, experiments indicate that a large performance potential remains yet to tap into. We find that current frameworks, on Nvidia Tesla V100 GPU, achieve less than 10% of the peak throughput, with only around 50% peak memory bandwidth.

PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02.
https://doi.org/10.1145/3437801.3441585

During up to 87% of the execution time, the occupancy of GPU is less than 10% (detailed in Section 3).

In this work, we provide an in-depth examination of these GNN frameworks. Through a series of experiments, we have identified five major gaps in current frameworks in optimizing GNN performance, especially in handling the special complexities of GNN over traditional graph or DNN operations. These gaps reside in the aspects of data locality, load balance, redundancy, memory footprint, and treatment to varying feature lengths.

Based on those insights, we put together a set of optimizations to fill the gaps. These optimizations leverage the state-of-the-art GPU optimization techniques and tailor them to the special properties of GNN. We develop *locality-aware task scheduling* and *neighbor grouping* to fit the needs of graph operation scheduling in GNN to promote locality and load balance. We schedule the operations on the computation graph through *data visible range adapter* to tackle the complex dependence among operations, which significantly reduces redundant graph data loads, kernel launches, and neighbor traverse. Taking advantage of graph structures, we develop *Sparse fetching* to avoid expanding the feature matrix and redundant computations when performing neural operations in GNN. Finally, we build a tuning framework to choose running configurations based on both the given problem (e.g., feature length, graph size, and computation pattern) and the characteristics of our optimizations.

We evaluate our optimizations with three popular GNN models, GCN [20], GAT [41], and GraphSage [10], with eight diverse graph datasets. Experimental results show that our optimizations produce 1.81×, 14.8×, and 3.76× average speedups over the state-of-the-art GNN frameworks, DGL [42], PyG [5], and ROC [14], on GCN. For models with more complex structures such as GAT, we can achieve 15.5× and 38.6× improvement over DGL and PyG. Even on computation intensive models such as GraphSAGE-LSTM [10], we can achieve 1.37× speedups.

In summary, we make three major contributions in this work.

1. As the first comprehensive study on the complexities of GNN for performance optimizations, we summarize a list of special complexities of GNN that go beyond what traditional graph computing and DNN frameworks can handle.
2. Through empirical studies on several representative GNN frameworks, we identify the major gaps in the state-of-the-art frameworks in optimizing GNN performance, especially in handling the special complexities of GNN over traditional graph or neural operations.
3. We show how to tailor the principles of GPU optimizations to fit the features of GNN, yielding a set of GNN optimizations that advance GNN computation significantly.

2 Complexities in GNN for Optimizations

This section first gives some background on GNN, and then presents its complexities for performance optimizations.

2.1 Overview of GNN

GNN is the neural networks performed on graph data. In a graph used in GNN, nodes represent entities in a problem domain (e.g., a user in a social network), with each carrying a feature vector. Edges between nodes indicate their relationship, quantified with edge weights. For a GNN model, there are mainly two kinds of operations. The first is **graph operation**. A node (usually called **center node**) collects feature vectors of its **neighbor nodes**, performs some operations, such as reductions or other kinds of computations, and updates its own feature vector accordingly. In this way, GNN can encode the graph structure and information using the updated node feature vectors. The second is **neural operations**. Neural operations are performed in two ways, either independently among nodes or in center-neighbor patterns according to the graph structure. The latter makes use of neighborhood relationship, and performs neural operations for each center node with neighbors' features.

A computing layer in GNN consists of both graph operations and neural operations. A simple example is shown in Equation 1, which computes the hidden features of center node v on layer $l+1$. The input for layer l is the hidden features h^l; $u \rightarrow v$ means that there is an edge from node u to node v; e_{uv} is the value on the edge. The layer reduces the features of neighbor nodes of v, and then computes tensor product using weights W^l. At last, an activation function is performed to produce the hidden features for the next layer h^{l+1}.

$$h_v^{l+1} = ReLU((SUM_{u\rightarrow v}(e_{uv} \odot h_u^l)) \otimes W^l) \quad (1)$$

Table 1 lists some commonly used computing layers for updating nodes in GNN. Table 2 lists some common operations for updating edge weights. All computing layers of a GNN form its **computation graph** [14].

Table 1. Common computing layers used in GNN models.

Layer type	Formula
sum	$SUM_{u\rightarrow v}(h_u^l * e_{uv})$
mean	$SUM_{u\rightarrow v}(h_u^l * e_{uv}/D_v)$
pooling	$MAX_{u\rightarrow v}(ACT(W^l \otimes h_u^l \odot e_{uv}))$
MLP [45]	$MLP^{(l)}(SUM_{u\rightarrow v}(h_u^l \odot e_{uv}))$
LSTM [10]	$LSTM_{u\rightarrow v}(h^l)$
Softmax_aggr [25]	$SUM_{u\rightarrow v}(h_u^l \odot softmax(e_{uv}))$

2.2 Complexities for Optimizations

GNN shares some common complexities with traditional graph computing and DNN for performance optimizations,

Table 2. Typical graph operations for computing edge weights in GNN models.

Model Names	Formula
Const	$e_{uv}^{con} = 1$
GCN [20]	$e_{uv}^{gcn} = \frac{1}{\sqrt{d_u d_v}}$
GAT [41]	$e_{uv}^{gat} = leaky\_relu((W_l * h_u + W_r * h_v))$
Sym-GAT	$e_{uv}^{sym} = e_{uv}^{gat} + e_{vu}^{gat}$
GaAN [50]	$e_{uv}^{cos} =< W_l * h_u, W_r * h_v >$
Linear [7]	$e_{uv}^{linear} = tanh(sum(W_l * h_u))$
Gene-linear [7]	$e_{uv}^{gene} = W_a * tanh(W_l * h_u + W_r * h_v)$

but also owns some distinctive features. Although a number of GNN frameworks have been developed in recent several years, no previous work has given a systematic examination of both common and special complexities. Such knowledge is essential for providing the right perspective to analyze the strengths and weaknesses of current GNN frameworks.

2.2.1 Common Complexities.

The complexities that GNN inherits from graph computing and DNN can be concisely summarized into two points: (i) irregularity brought by graphs; (ii) large volumes of memory footprint together with heavy workloads in memory access and computations brought by DNN. Nodes in a graph differ in degree and accessed nodes often reside discontinuously in memory, causing difficulties for load balance and memory optimizations. The large volumes of memory footprint and computations, especially when the neural network is getting deeper or wider [26, 44], further worsen the difficulty.

2.2.2 Special Complexities.

GNN has three-fold special complexities, which make the simple combination of traditional graph computing optimizations and DNN optimizations insufficient.

Complex dependence and interleaving patterns. In each GNN layer, there are interleaved graph operations and neural operations of various kinds, which leads to intensive function calls with large overhead of kernel launch and framework scheduling. Many previous optimizations for DNN are difficult to apply in GNN. For instance, none of the existing GNN frameworks can apply effective kernel fusion [22, 51], a common DNN optimization strategy, to GNN because graph operations in GNN have irregular thread mapping patterns, unpredictable memory accesses, and complex dependencies between operations, a clear contrast to regular element-wise and dense operations in DNNs.

Neural operations performed in Graph patterns. Unlike DNNs where neural operations are performed independently on tensors, GNN contains some neural operations that have to be conducted in a center-neighbor pattern [10]. As shown in Figure 1, center nodes collect features from neighbors and perform neural operations (e.g., RNN cells) rather than

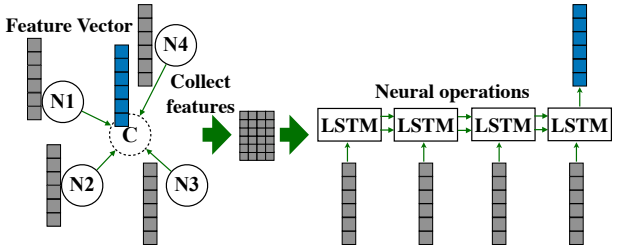

Figure 1. Neural operations performed in a center-neighbor pattern

simple reduction on them. Performing neural operations in a center-neighbor pattern brings the complexity of graph structure into the computation. The highly optimized libraries developed for DNNs support neural operations but cannot work well on such computation patterns. Moreover, the mixture of graph operations and neural operations prompts a large amount of data movements for the need of expanding feature matrices for neural operations according to graph structure.

Varying feature lengths. Feature lengths in a GNN model can range from one to thousands in practice. There can be multiple types of features on each node, such as hidden feature and attention feature (for the computation of self-attention [40, 41]) of different lengths. Meanwhile, the feature length gets changed by transformations at each layer. Although different layers of DNN may see activation maps of different lengths, varying lengths impose some special complexities to optimizations when DNN and graph computations are mixed together at each layer as highly optimized libraries (e.g., cuBLAS and cuDNN) cannot be directly applied and the irregularity and complexity of graph are amplified.

3 Gaps in Existing GNN Frameworks

With the perspective prepared, this section presents the major performance gaps in existing GNN frameworks that we have identified through a set of measurements. We start with our methodology.

3.1 Methodology

We choose Deep Graph Library (DGL) [42] and PyTorch Geometric (PyG) [5] as common GNN frameworks to study. Both have PyTorch [36] as backend. The frameworks represent the state of the art in terms of supported GNN models and performance. We also analyze ROC [14] and NeuGraph [29]. Our experiments use three popular GNN models, GCN [20], GAT [41], and GraphSage [10], with eight diverse graph datasets. The datasets are from Open Graph Benchmark (OGB) [12], the unified benchmark suite for GNN models. Table 3 lists them. With 49M edges on average, the datasets are much larger than the ones (e.g., CORA, Pubmed, Citeseer) studied in previous work whose average number of edges

Table 3. Graph datasets

	#N	#E	avg	max	Var	Density
Citation network						
collab	236K	2.4M	10	671	360	4.2E-5
citation	2.9M	30M	10	1738	221	4.0E-6
arxiv	169K	1.2M	7	13155	4.6K	4.1E-5
Biology network						
protein	133K	79M	597	7750	386K	4.5E-3
ddi	4K	2.1M	501	2234	177K	1.2E-1
ppa	576K	42M	74	3241	9.9K	1.3E-4
Social network						
reddit	233K	115M	492	21657	640K	2.1E-3
Co-purchasing network						
products	2.4M	124M	51	17481	9.1K	2.1E-5

is 37K, and can represent the trend of GNN model research. The datasets also have a broader coverage of domains.

We used DGL 0.4.3post2 and PyG 1.5.0 on Nvidia Tesla V100 with CUDA 10.1, and the deep learning framework is PyTorch 1.5.1.

3.2 Observed Performance Gaps

We make five main observations on the performance gaps in existing GNN frameworks. Some of these gaps also exist in either graph or DNN computings, while others are due to the special complexities brought by the interleaving and dependence of two kinds of computations in GNN. We list them all here to provide a complete understanding of the gaps needed to fill.

Observation 1: Poor locality in graph operations due to edge granularity

Graph operations perform computation according to graph structure, in which an edge indicates data transfer. Let E be the number of edges in a graph and $Feat$ the length of the node feature. Graph operations on one layer can involve $E * Feat$ bytes of data movement in loading node features, which is a dominating performance factor. The total size of node features is however only $N * Feat$ (N is the number of nodes in the graph). In theory, with perfect data reuse, only $N * Feat$ bytes rather than $E * Feat$ need to be loaded. Despite the obvious importance of reuse, existing GNN frameworks take advantage of reuse poorly.

PyG employs edge-wise parallelization[1]. As shown in the upper part of Figure 2, the graph structure is represented in an edge list format. Its first step in graph operations is to expand the feature matrix of $[N, Feat]$ to the feature matrix $[E, Feat]$ for source nodes, duplicating the features of source nodes in the edge list. It then performs a reduction on the feature matrix for source nodes. Two steps are performed in separate kernels; the source feature matrix dimensions are as large as $[E, Feat]$.

[1]From PyG 1.6.0, it implements some of aggregation operations in a center-neighbor pattern just like DGL.

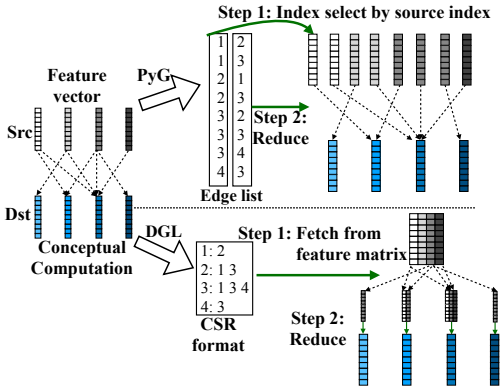

Figure 2. Graph operations in PyG and DGL

DGL avoids the large space usage by employing node-wise parallelization with a center-neighbor pattern. Each task takes charge of the computation of a center node and its neighbor nodes. As shown in the lower part of Figure 2, the graph structure is represented in Compressed Sparse Row (CSR) format, and each task first fetches data from the feature matrix, and then performs reduction to update the feature of the center node. Different tasks are assigned to warps or thread blocks in GPU. If the reduce function is SUM, DGL utilizes cuSPARSE [33] to perform the sum reduction. Otherwise, it conducts the two steps in a single kernel. The single-GPU versions of ROC and NeuGraph also implement graph operations in this way.

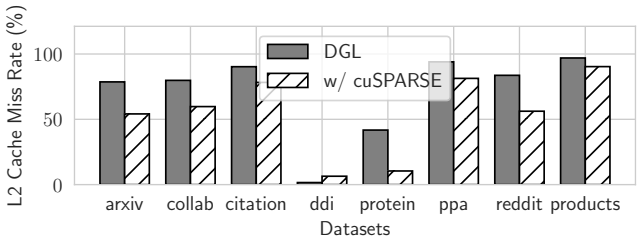

Figure 3. L2 cache miss rates of graph operations in the last layer of GCN in DGL. (It uses cuSPARSE when the reduce function is SUM, shown by "w/ cuSPARSE" bars.)

Figure 3 shows the L2 cache miss rates of graph operations in GCN implemented with DGL, running with eight datasets. Except on the small or already clustered datasets (`ddi`, `protein`), the executions exhibit over 50% L2 cache miss rates. The reason for the poor locality is that DGL statically distributes the tasks of center nodes to underlying computing units, leaving memory access pattern fully determined by the graph, which has an irregular pattern.

Observation 2: Severe workload imbalance

DGL, ROC, and NeuGraph all partition tasks in center node granularity, assigning the computations for a center

Table 4. The percentage of time when #active block is less than a proportion in DGL for graph operations in GAT.

Dataset	<100%	<50%	<10%
arxiv	89.99	89.64	87.87
collab	34.35	33.39	31.83
citation	3.23	1.91	1.15
ddi	74.39	63.79	42.55
protein	14.12	11.11	9.09
ppa	6.49	4.89	3.40
reddit	19.15	17.33	15.21
products	5.70	4.40	3.58

node and all its neighbors to one computing unit. Due to the irregularity of a graph, the nodes in a graph may have large variances in the number of neighbors. Moreover, the feature vector length in GNN is typically large, which makes the computation for each neighbor node heavy. These two factors lead to severe workload imbalance, resulting in long-tail effects. Table 4 reports the percentage of time when the number of active thread blocks is less than 100%, 50%, and 10% of the maximum number that can concurrently run on a GPU. In a substantial part of the executions, the GPU is underutilized. PyG, for its distribution of graph operations in edge granularity, is less subject to load imbalance; but as mentioned earlier, the needed duplications result in large overhead.

Observation 3: Redundancy in memory access and large overhead due to intensive function calls

DGL and PyG build up computation graph using many graph operations. Consider a GAT layer. The core computation is shown in Equation 2. For a center node, it first updates the edge weight with att_{src} and att_{dst} with the activation function $leaky\_ReLU$. It then normalizes the weights of all incoming edges with $softmax$. It finally can apply the edge weights to the features of the neighbor nodes to update the center node. The implementation in DGL breaks the layer into seven steps as shown in Listing 1. PyG shares a similar way of segmenting computations into many steps and kernels. There are hence repeated loading of the graph structures (for DGL) or redundant global memory accesses (for PyG), which cause large overhead.

```
# input: graph, feat_src, att_src, att_dst
# update edge weight
e = graph.u_add_v(att_src, att_dst) # [E, 1]
e = leaky_relu(e) # [E, 1]
# edge softmax
e = exp(e) # [E, 1]
v_acc = graph.reduce_edge("sum", e) # [N, 1]
e_acc = graph.broadcast_edge(v_acc) # [E, 1]
e = div(e, e_acc) # [E, 1]
# aggregation
outf = graph.reduce_vertex("sum", u_mul_e(feat_src
    , e)) # [N, Feat]
```

Listing 1. GAT layer in DGL

$$h_v^{l+1} = SUM_{u \to v}(softmax_v(leaky\_ReLU(att_v + att_u)) \odot h_u^l) \quad (2)$$

Table 5. The percentage of the execution time of expansion and transformation in neural operations (DGL on GraphSAGE-LSTM).

Dataset	Expansion %	Transformation %
arxiv	9.60	25.60
collab	9.70	21.42
citation	7.32	19.02
ddi	8.89	20.85
protein	9.69	23.01
ppa	9.95	24.32
reddit	9.42	22.64
products	8.05	18.77

Observation 4: Large memory footprints and redundant computations when expanding neural operations by graph structure

Neural operations performed in a center-neighbor pattern are based on the graph structure. The frameworks doing that involve large memory footprints and redundant computations. DGL, for instance, breaks computation into two steps: it first expands each center node's neighbor features into a continuous memory space, and then performs neural operations on the expanded matrix in a dense way. The problem is that the first step (expansion) results in large footprints and the next step (transformation) results in large redundancy. After expanding the feature matrix, even if two feature vectors are identical, without this information, the transformation has to be redone on those vectors. Table 5 reports the portion of time taken by the expansion and the transformation in the overall time when DGL processes GraphSAGE-LSTM, which conducts LSTM on the neighbor features of each center node. The cost of the two together is as much as 35%.

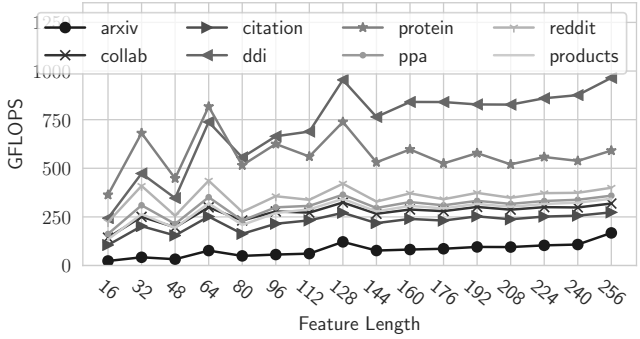

Figure 4. The throughput changes as feature lengths change.

Observation 5: Inefficiency on variant feature lengths

Neither DGL nor PyG adapts the computations based on feature lengths. Their task distribution and the mapping of

computation to threads remain the same for different feature lengths. As the performance of graph operations are very sensitive to locality, utilizing the same schedule for different feature lengths can lead to large performance degradation. As shown in Figure 4, the throughput changes significantly even if the feature length changes slightly.

4 Tailoring Optimizations to Fill the Gaps

After identifying the performance gaps, we further develop a set of optimizations to fill those gaps. Please note that our goal is not to invent fundamentally new GPU optimization techniques. In fact, at the fundamental level, these optimizations all grow out of the common principles already established in the general GPU code optimization community around data locality, load balance, and kernel fusions. The question we try to answer here is how to translate these optimization principles into effective optimizations that fit the properties of GNN, thereby, state-of-the-art of GNN computing can be advanced.

Specifically, we have developed four optimizations, two on GPU resource and locality for graph operations, two on the co-optimizations of graph operations and neural operations.

4.1 Task Distributor for Graph Operations

The irregularity of graphs leads to poor locality and workload imbalance in graph operations as previous sections have mentioned. We develop two techniques, *locality-aware task scheduling* and *neighbor grouping*, to address these issues. The key of *locality-aware task scheduling* is to make the tasks of center nodes that have similar neighbor distributions run concurrently to make better use of cache. *Neighbor grouping* addresses load imbalance by further partitioning the workload of each center node into finer granularity.

4.1.1 Locality-aware task scheduling. In *locality-aware task scheduling*, we re-order the tasks of each center node by considering their neighbor distributions. The basic idea is to make tasks with similar neighbor distribution run concurrently as much as possible, so that cache can be better used. This optimization is effective for GNN applications because they have large feature lengths and the graph operations in them can benefit significantly from data reuse. The challenge is how to identify and schedule tasks that share similar distributions efficiently.

We first introduce the similarity metric we use, *Jaccard Similarity*. The Jaccard Similarity of two center nodes is defined as $\frac{|N_a.neighbor \cap N_b.neighbor|}{|N_a.neighbor \cup N_b.neighbor|}$.

There are three steps in the scheduling: First, we use *candidate pair selection* to find the pair of center nodes that have high neighbor similarity, then use *pair merging* to merge the similar nodes into larger clusters and then use *task scheduling* to schedule the center nodes to the real computation.

(1) *Candidate pair selection* To cluster the nodes with similar neighbor distribution, we first find out all similar pairs of center nodes in the graph that have neighbor distribution with high Jaccard Similarity. We adopt Min-Hashing and Locality-Sensitive Hashing (LSH) [24] to efficiently select the pairs. Min-Hashing reduces the computation workload of each pair by converting large neighbor sets to short signatures, while approximately preserving Jaccard Similarity. LSH further segments the signatures into multiple bands, and hashes the different bands into buckets. Nodes in the same buckets are more likely to have large similarity while the ones in different buckets do not. Therefore, LSH reduces the search space of pairs with large similarity. Its high efficiency makes it a suitable choice for our scheduling, as we are finding out the similar nodes on large graphs.

(2) *Pair merging* After pairing up nodes with high Jaccard Similarity, we further put nodes with high similarity into the same cluster using the pairs. At beginning, every node is in a separate *cluster* with itself being the *representative node* of the cluster. The pairs are organized in a priority queue, with the priority defined by the Jaccard similarity. We dequeue the top pair in the queue, and merge the cluster that the two nodes in the pair belong to. If the nodes are both the representative node of their cluster, the clusters will be merged into one, with the representative node of the larger cluster (containing more nodes) as the new representative node. Otherwise, we will pair up the representative node of the two clusters, and enqueue that new pair into the priority queue. We avoid having large clusters by setting an upper bound to the cluster size (32 in our experiments), it also prevents pairs with low similarity to get into the same cluster. The process continues until all clusters are full or the queue gets empty.

(3) *Task scheduling* After merging the pairs, we have a number of clusters and each cluster contains nodes with similar neighbor distribution. We then map the clusters onto different computing units. We try to distribute the tasks of nodes in the same cluster into adjacent computing units (e.g., adjacent thread warps or thread blocks), so that cache can be better reused. We perform such scheduling directly on the graph structure to minimize scheduling overhead.

4.1.2 Neighbor grouping. Besides the distribution of neighbor nodes, another irregularity is the numbers of neighbors of center nodes, an important reason for the load imbalance in Observation 2.

To balance the workload, we schedule the tasks in a fine-grained fashion by partitioning the tasks of center nodes according to their numbers of neighbors. We partition the neighbors of each center node into different groups by setting an upper bound (a tunable parameter) for the number of neighbors in every group. The computation for the neighbor nodes within a group is then assigned to a specific computing unit. As a result, workloads for center nodes that have a large number of neighbors will be distributed to multiple

computing units, resulting in more balanced schedules. This method is called *neighbor grouping*.

Neighbor grouping forms a synergy with *locality-aware task scheduling* in enhancing data locality. As computing units now execute only part of a task of a center node, the working set (the number of node features that GPU will visit at the same time) becomes smaller and the collective memory access is more likely to benefit from *locality-aware task scheduling*.

A problem for *neighbor grouping* is that the computation results of one center node can be put into multiple computing units, which then might be mapped to different SMs. Reduction would need to be performed to collect results across SMs. By analyzing typical GNN applications, we find that most of the reducers for GNN operations (e.g., Max, Mean, and Sum) allow arbitrary orders in the aggregation. That allows each SM to calculate the reduced results independently and use atomic instructions to update the global memory; no data exchanging is needed across SMs.

4.2 Data Visible Range Adapter

As mentioned in Observation 3, the intensive function calls in computation graphs introduce a large amount of redundant memory access. The main reason is that graph operations and neural operations have distinct computation characteristics. Due to the irregularity of graph operations, it is tricky to directly fuse graph operations and neural operations into the same kernel. Typically, for coalesced memory access, each thread of graph operations is in charge of partial computation of the neighbor node features, so outputs are private for each thread. However, regular neural operations are usually performed with a bunch of threads (e.g., a thread block). The mismatch of scopes makes the fusion even harder.

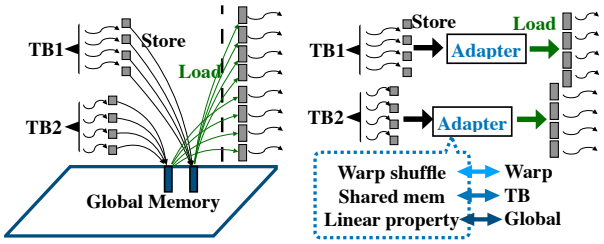

Figure 5. *Data visible range adapters* remove mismatches in the data visibility ranges across kernels. Enabled kernel fusions can save kernel launch overhead, redundant graph loading and the memory access of intermediate data.

The mismatch can be formulated with a concept named *data visible range*. The data visible range refers to in what scope of threads a data item is visible to. It could be thread level, warp level, thread block level, or global level. A mismatch between different operations occurs when the data visible ranges of two operations differ. If the visible range of some data is global in a kernel, a global synchronization (i.e., the end of a kernel on GPU) would be needed before the data can be consumed by another operation. As shown on the left of Figure 5, DGL and PyG ignore the property of data visible ranges, so they put operations in separate kernels by default even if the data only need to be shared within a thread block.

Our proposed *data visible range adapter* tries to identify minimum visible ranges for operations in a model while at the same time address mismatches of visible ranges of adjacent kernels to enable kernel fusion. It leverages inter-thread communication and shared memory to expand visible range from thread to warp and thread block. As the right side of Figure 5 illustrates, we introduce adapters for each thread block. The threads in a thread block put thread-local data into adapters, and the adapters convert them into thread block level. The thread block of the next operation can continue computation without waiting for other blocks or visiting global memory. Two kernels can then be fused together.

Even with *data visible range adapter*, data with global visibility still require global-memory-based communications. We leverage the *linear property* of operations (e.g., Sum, Mean, and Div) to postpone the needed global synchronizations to mitigate the effects. Consider two kernels, K1 and K2. K1 first does normalization of the edge weights by dividing each edge weights with the summation result of all edge weights of a center node, K2 uses the normalized edge weight to scale the neighbor node feature, and performs reduction on the center node. If *neighbor grouping* splits the task of a center node into two SMs, the first step would have to require global visibility of edges for the calculation of the summation. But as the division of the sum can be postponed to later (the linear property), we transform the kernel by moving it to the second kernel (which automatically ensures the summation computation is all done) while keeping other operations running efficiently in a single kernel.

4.3 Sparse Fetching and Redundancy Bypassing

As analyzed in Observation 4, current GNN frameworks perform graph operations and neural operations separately, where graph operations expand feature matrices, and then neural operations perform transformations on the expanded data. That introduces large redundancy in memory access and computation.

We solve this problem by proposing *sparse fetching* and *redundancy bypassing*. The basic idea is to combine neural operations with graph operations and extract common computation. *Sparse fetching* is a data access method in graph operations, which accesses the data sparsely according to neighbor indexes. Here we attach it to the neural operations and can move the memory access of sparse data from a separate kernel into neural operation kernels.

Instead of accessing the data continuously, the threads first acquire the information of the graph, and use the neighbor index to fetch data to perform neural operations. The

overhead of the memory access can be largely hidden by heavy neural operations afterwards in the same kernel. *Redundancy Bypassing* is further applied on top of *Sparse fetching*. While *Sparse fetching* considers the graph structure in neural operations, *Redundancy Bypassing* utilizes the information of common neighbors to eliminate the redundant transformation by limiting the computation on the node feature matrices, which reduces the computation from $O(E)$ to $O(N)$.

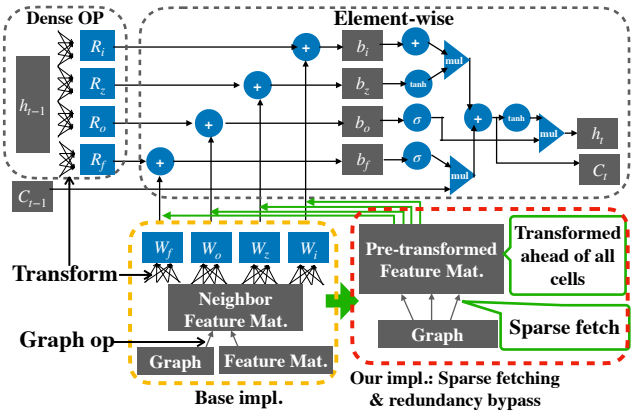

Figure 6. An optimizing example for performing LSTM in a center-neighbor pattern.

Figure 6 shows an example for optimizing GraphSAGE-LSTM in the center-neighbor pattern. The t^{th} cell of LSTM takes the output from the previous cell (h_{t-1}) and the t^{th} neighbor's features of every center node as input. The transformation is firstly done for neighbor features using weight matrix W_f, W_o, W_z, W_i, and for h_{t-1} using weight matrix R_i, R_z, R_o, R_f. It then performs element-wise operations for transformed data and finishes computation for the current LSTM cell. The part in the yellow box shows the base implementation; a graph operation is performed to construct the neighbor feature matrix by collecting the t^{th} neighbor's feature for all center nodes. The transformations are then performed on the neighbor feature matrix before they are fed into the element-wise operations.

However, as shown in the red box, by applying *sparse fetching* at the beginning of the neural network kernel, the graph structure is used to get access to the data in the feature matrix, without performing separate graph operations. Moreover, *Redundancy bypassing* helps to bypass the redundant neural network transformations for the neighbor feature matrix in every LSTM cell. It enables us to apply the transformation ahead of all the LSTM cells for only once. After that, the pre-transformed features are fetched according to graph structure at every LSTM cell. By applying *Sparse Fetching* and *Redundancy Bypassing* to the neural operations following graph structure, we can reduce both redundant data movement and computation.

4.4 Tuning, Adaptation, and Overhead

There are several tunable parameters in the proposed optimizations, such as the upper bound in *neighbor grouping* and the configurations of scheduling and running. The best settings of the optimizations and scheduling could differ for different graphs and feature lengths. We hence construct a tuner to empirically search for appropriate parameter values.

The tuner contains some scheduling and optimization templates to facilitate the search process. The strategy is to first exhaust GPU resources by scheduling more warps and increase the maximum number of thread blocks by limiting their resources such as shared memory usage. As GPU fully occupied, the tuner then adjusts the upper bound of *neighbor grouping*, putting tasks of feature dimension to the same computing unit, and fusing kernels for better locality.

The overhead of our optimizations mainly comes from the analysis performed on the graph structure, which includes the neighbor distribution of center nodes (*locality-aware scheduling*) and their neighbor number (*neighbor grouping*). *Locality-aware scheduling* takes some time (e.g., 600 iterations of a three-layer GCN). It is done offline as we only need to do it once because the graph structure stays invariant. The results however can be used for many runs of the GNN. As an iterative application, GNN usually needs many repeated runs for hyper-parameter tuning [10, 25, 45] and each run may involve thousands of epochs [26].

Neighbor grouping is done online, as its configuration is affected by both graph structure and the given problem. As it only iterates the index in CSR matrix once, which is $O(N)$, the overhead is less than a half epoch and can be done asynchronously with the computation. Multiple-round online tuning is used to determine the appropriate group size upper bound, with a different bound tried in each round. We limit the choices to multiples of 16 and the maximum to be ten times of the average node degree of the given graph. The numbers of rounds needed never exceed 20 in our experiments, negligible compared to the thousands of iterations in GNN executions.

5 Performance Improvements

We implement the optimization techniques introduced in Section 4 and wrap them in PyTorch. This section reports observed performance improvements. The methodology is the same as in Section 3.1. We compare the results with those of DGL and PyG. Our experiments use three representative GNN models, GCN, GAT, and GraphSAGE-LSTM, and eight graph datasets (see Section 3.1 to describe them). As our optimizations do not alter the semantics of the models, the quality of the model outputs remain unchanged. We hence focus on performance (i.e., speed) comparisons. We first report the overall performance and then provide detailed studies on the benefits from each optimization. We report absolute execution time in Section 5.1. For benefits breakdown, we use normalized graphs for readability.

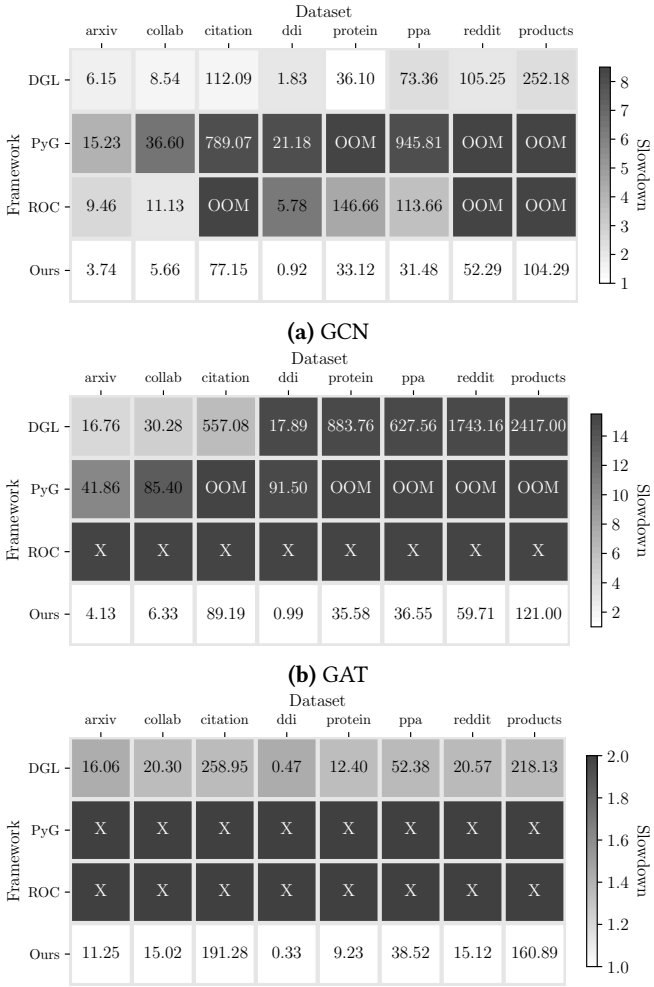

Figure 7. A heatmap of performance slowdowns of three frameworks (DGL, PyG, and ROC) compared to our implementation on different datasets and models. Darker color means larger slowdown the framework has. The numbers are the execution time (ms), lower numbers are better, while the white numbers mean that the slowdown exceeds the threshold, which is 8, 15, 2 respectively for GCN, GAT, and SAGE. "×" means the framework does not implement the model and "*OOM*" means the framework runs out of GPU memory.

5.1 Overall performance

Figure 7 shows the average times taken by one forward pass of each of three GNN models (three stacked layers in GCN and GAT[2] and one layer in GraphSAGE-LSTM[3]) on each dataset.

[2]The input feature length is 512, with 128 and 64 hidden features, and 32 output features.

[3]The input and output feature lengths are both 32, with the sampled neighbor number as 16

On GCN model, our improvement is 1.81×, 14.8×, and 3.76× compared to DGL, PyG, and ROC respectively. Since the computation pattern in GCN is simple, the improvement mainly comes from the optimizations of graph operations together with computation graph optimization. DGL uses cuSPARSE to perform graph operations in GCN, while PyG expands the feature matrix, leading to large redundancy. It would run out of memory for datasets with many edges as its memory consumption increases linearly. On GAT model, the improvement is 15.5× and 38.6× over DGL and PyG. For the complicated computation graph in GAT, our method saves a large number of memory accesses by kernel fusion and optimizations of graph operations. On GraphSAGE-LSTM, the speedup over DGL is 1.37×; PyG does not support the model. We next provide detailed results and analysis.

5.2 Detailed Analysis

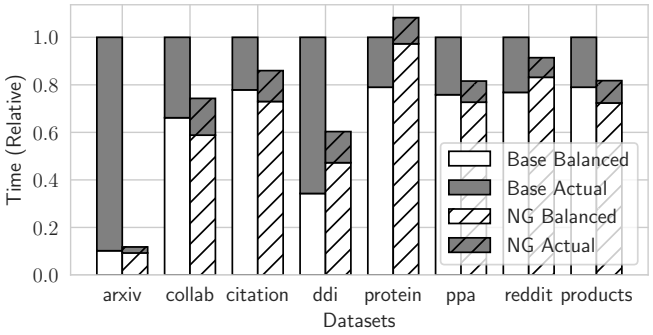

Figure 8. *Neighbor grouping (NG)* enhances load balance on graph operations in the last layer of GCN. "Base": default implementation (as in DGL); "NG": our implementation; "Balanced": execution time in perfect load balance; "Actual": the actual execution time.

Neighbor Grouping. Figure 8 illustrates the benefits of *neighbor grouping* in enhancing load balance. The "Balanced" bars (light-colored portion) show the execution time in perfect load balance, measured by dividing the total time of all thread blocks by the maximum number of active thread blocks the a GPU can run at a moment. The gap between the "Balanced" and actual execution time of the entire kernel is reduced significantly when *neighbor grouping* is applied, showing its benefits in improving the load balance. The light-colored portions of some "NG" bars are higher than those of "Base" bars, due to the extra global memory accesses incurred by *neighbor grouping*. However, the actual time in "NG" is significantly lower than the "Base" bars. The only exception is `Protein`, where the variance of the neighbor number is little, and the benefit by *neighbor grouping* is out-weighted by the overhead, leading to 8% performance decrease.

Locality-aware scheduling. Figure 9 shows the locality improved by task scheduling for graph operations. By itself, it

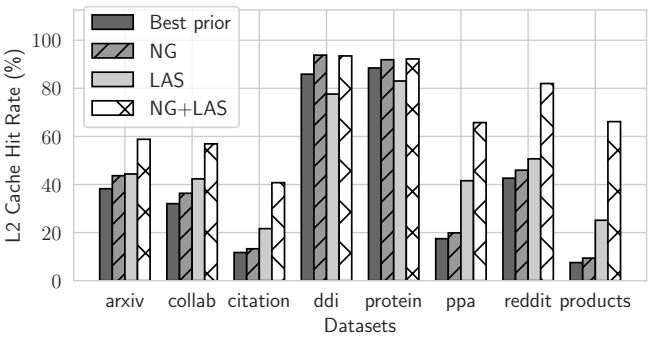

Figure 9. L2 cache hit rates of graph operation in the last layer of GCN. "Best prior": The best L2 cache hit rate other implementations (DGL, PyG and ROC) can achieve; "NG": Applying *Neighbor Grouping*; LAS: Applying *Locality-aware scheduling*; "NG+LAS": Applying both *Neighbor Grouping* and *Locality-aware scheduling*.

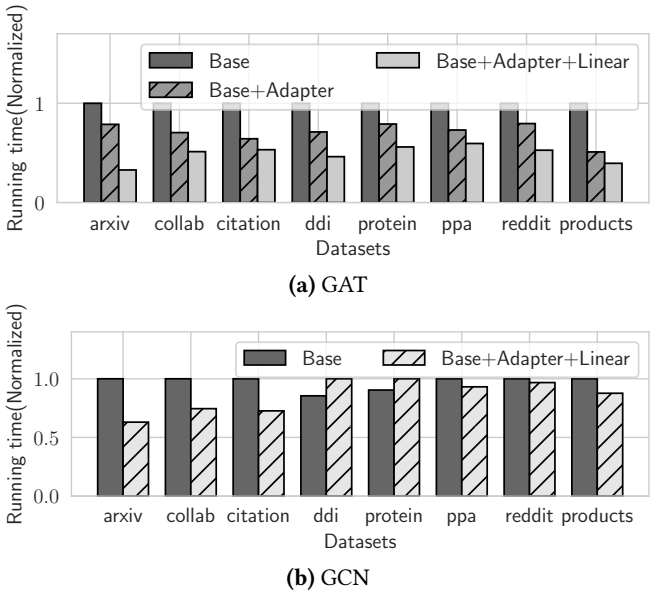

Figure 10. Benefits from *Data visible range adapter* w/ and w/o using linear property on GAT layer and GCN layer. The baseline is our implementation with *Neighbor grouping* and *Locality-aware task scheduling*.

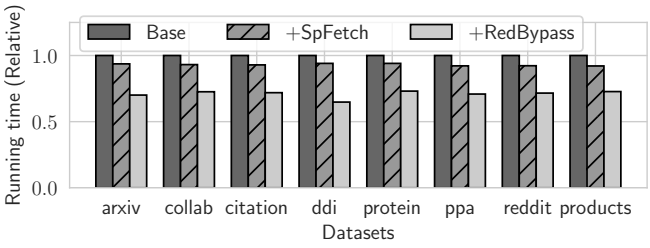

Figure 11. Benefits from *sparse fetching* and *redundancy bypassing* on GraphSAGE-LSTM.

increases the L2 cache hits on six out of the eight cases. Its benefits become a lot more obvious when *neighbor grouping* is applied first. The reason is that, as the active warps are doing computation for thousands of center nodes, the neighbor node features that they may access concurrently are of a large number. Even if the nodes with similar distributions are grouped, the active area is still huge. *Neighbor grouping* helps mitigate the situation, making it more amenable for the scheduling to function by narrowing the active area. Dataset `protein` is a protein dataset with inherent clustered distributions. Performing *locality-aware scheduling* breaks the clustered pattern of it, while the reorder of the computing sequence brings uncertainty to graphs with high density like `ddi`; they hence see a slight decrease of cache hits.

Computation graph scheduling. *Data visible range adapter* helps eliminate the barriers between graph operations on GAT, and the barriers between neural operations and graph operations on GCN, enabling kernel fusions. Figure 10a shows significant improvement for a complex computation graph on GAT. When the optimization takes advantage of the linear property to postpone some operations to later kernels, it produces even more speedups.

GCN has a simple computation graph. The improvement is hence limited (16% as Figure 10b shows), compared to the version with only graph operation optimizations (*Locality aware task scheduling* and *Neighbor Grouping*). `ddi` and `protein` even show a slight performance decrease thanks to their good data locality (shown in Figure 9) which leads to high memory throughput for graph operations in the original version.

Sparse fetching and redundancy bypassing. The performance improvement from *sparse fetching* and *redundancy bypassing* is shown in Figure 11. For *sparse fetching*, the kernel fetches the features by using the neighbors' indices, which can hurt locality. Thus the overall improvement by *sparse fetching* is limited to below 10%. However, it enables additional bypassing of redundant transformations, which then brings 32% improvement.

Tuning and adaption. Figure 12 shows how the performance of GCN graph operations changes with different feature lengths. As tuning is applied, though the different feature lengths result in distinct memory access pattern, our implementation can achieve good performance.

Online and offline improvement analysis. It is worth noting that the offline pre-processing is optional. It is only needed for one (*Locality-aware task scheduling*) of the four optimizations proposed in this work. The other three optimizations are either online optimizations (*Neighbor grouping*) or kernel code optimizations (*Data visible range adapter*, *Sparse fetching*, and *Redundancy bypassing*).

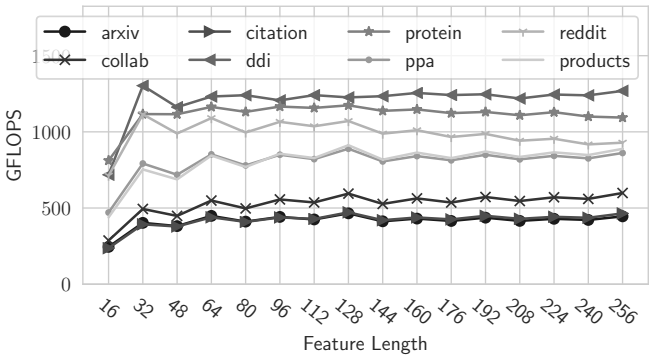

Figure 12. The throughput changes as feature lengths change when tuning is applied.

Even without the offline optimization, the other three optimizations can already produce significant speedups. As shown in Table 6, the online optimization and one of the two kernel optimizations together can already bring up to 8× (2.89× on average) speedups. The offline optimization can bring 1.6× extra speedups, but it is not a must-to-have in scenarios where offline pre-processing is not suitable to use (e.g., graph dynamically changes at every iteration when graph sampling is applied).

Table 6. The speedup of applying different optimizations to the last layer of GAT on different datasets over our unoptimized implementation. "Adp": *Data visible range adapter*, "NG": *Neighbor grouping*, "LAS": *Locality-aware task scheduling*

Dataset	Adp	Adp+NG	Adp+NG+LAS
arxiv	1.07	8.02	9.85
collab	1.31	1.76	2.41
citation	1.43	1.86	2.24
ddi	1.25	2.57	2.86
protein	1.26	1.96	1.83
ppa	1.20	2.20	2.67
reddit	1.15	1.95	2.68
products	1.51	2.83	3.62
AVERAGE	1.27	2.89	3.52

6 Related work

GNN frameworks. PyG [5] and DGL [42] are GNN frameworks widely used by GNN model researchers. They provide user-friendly programming interfaces. Roc [14] and NeuGraph [29] are designed to meet the trend that GNN is applied to larger graphs. They propose pipeline and graph partition methods to scale the computation to multi-GPU and multi-node respectively. Their focus is to reduce the time of communication for training in a distributed way. G^3 [27] demonstrates how to use graph processing frameworks to train GNN, and provides flexible APIs to build GNN models. The previous sections have provided detailed comparisons with previous GNN frameworks.

GPU graph computing frameworks. To make use of GPU's good parallelism, there are a number of studies exploiting GPU for large graph processing. GunRock [43] provides graph processing primitives with high performance. Groute [1] puts graph computing onto multiple GPUs and leverage asynchronous communications to reduce latency. Gluon [4] puts together a communication library for distributed heterogeneous graph processing. Lux [13] supports multi-node GPU graph processing. These works have contributed valuable insights on optimizing graph computing. This work is partially inspired by these studies, but covers the complexities special to GNNs.

Sparse operation optimization. Besides graph computing, sparse operations exist in many other cases, such as SpMV [30], SpMM [46, 47], SDDMM [11] and spGEMM [19]. AsPT [11] focuses on optimizing SpMM and SDDMM by adaptively tiling sparse matrix into dense and sparse tiles. spECK [35] optimizes sparse general matrix-matrix multiplication using light weighted analysis. Tensor Algebra Compiler (TACO) [21] uses compiler techniques to achieve competitive performance with hand-optimized kernels for both sparse tensor algebra and sparse linear algebra. These optimizations all focus on a single primitive kernel. Though they achieve good performance, how they can be applied to GNNs is yet to understand.

Deep neural network performance optimization. There are many previous studies on optimizing DNNs on GPUs. They optimize DNNs through computation graph transformation [15, 16], changing the dimension of parallelism [18], switching the batching method [34], and so on. These methods are all about DNNs only, without covering graphs as inputs and targets. The other kind of DNN optimization is about exploring the sparsity in DNNs, such as optimizations and scheduling based on certain sparse patterns [31, 32, 48]. They mainly focus on performing efficient SpMM on sparse data [6], rather than the special complexities in the combination of graph computing and neural operations.

7 Conclusion

This paper has analyzed the special complexities in optimizing GNNs, and presents an in-depth examination of the performance gaps in existing GNN frameworks. We point out five major gaps in locality, load balance, redundancy, memory footprints, and the lack of adaptation to varying feature lengths. We propose a set of optimizations to fill the gaps. These optimizations stem from some general principles of GPU code optimizations but customize the designs to fit the special properties of GNNs. They include *locality-aware task scheduling*, *neighbor grouping*, *data visible range adapter*, *sparse fetching*, and *redundancy bypassing*. Experiments show

that our optimizations lead to performance improvement of 1.37×–15.5× over the state of the art. As GNNs gain increasing popularity in many domains, our findings in this work can help better meet the demands for the efficiency of many GNN-based applications.

Acknowledgments

We would like to thank anonymous reviewers for their insightful feedback. This work is partially supported by National Key R&D Program of China (2017YFB1003103), National Natural Science Foundation of China (U20A20226), Beijing Natural Science Foundation (4202031), Beijing Academy of Artificial Intelligence (BAAI), and Tsinghua University Initiative Scientific Research Program (20191080594). This material is based upon work supported by the National Science Foundation (NSF) under Grants CNS-1717425, CCF-1703487, CCF-2028850, and the Department of Energy (DOE) under Grant DE-SC0013700. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE. This work is partly supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2018R1D1A1B07050463). Jidong Zhai is the corresponding author of this paper.

References

[1] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/3018743.3018756

[2] Yukuo Cen, Zhenyu Hou, Yan Wang, Qibin Chen, and Jie Tang. 2020. CogDL: An Extensive Research Toolkit for Deep Learning on Graphs. https://github.com/thudm/cogdl

[3] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/3292500.3330925

[4] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 752–768. https://doi.org/10.1145/3192366.3192404

[5] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019). arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[6] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. , Article 17 (2020), 14 pages.

[7] Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. 2020. Graph Neural Architecture Search. (7 2020), 1403–1409. https://doi.org/10.24963/ijcai.2020/195 Main track.

[8] Thomas Gaudelet, Ben Day, Arian R. Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy B. R. Hayter, Richard Vickers, Charles Roberts, Jian Tang, David Roblin, Tom L. Blundell, Michael M. Bronstein, and Jake P. Taylor-King. 2020. Utilising Graph Machine Learning within Drug Discovery and Development. arXiv:2012.05716 [q-bio.QM]

[9] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 855–864. https://doi.org/10.1145/2939672.2939754

[10] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[11] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712

[12] 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. (2020). https://proceedings.neurips.cc/paper/2020/hash/fb60d411a5c5b72b2e7d3527cfc84fd0-Abstract.html

[13] Zhihao Jia, Yongkee Kwon, Galen M. Shipman, Patrick S. McCormick, Mattan Erez, and Alex Aiken. 2017. A Distributed Multi-GPU System for Fast Graph Processing. *Proc. VLDB Endow.* 11, 3 (2017), 297–310. https://doi.org/10.14778/3157794.3157799

[14] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org, Austin, TX, USA, 187–198. https://proceedings.mlsys.org/book/300.pdf

[15] Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, and Alex Aiken. 2020. Redundancy-Free Computation for Graph Neural Networks. (2020), 997–1005. https://doi.org/10.1145/3394486.3403142

[16] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[17] Zhihao Jia, James J. Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, Ameet Talwalkar, Virginia Smith, and Matei Zaharia (Eds.). mlsys.org, Stanford, CA, USA. https://proceedings.mlsys.org/book/276.pdf

[18] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. 1 (2019), 1–13. https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf

[19] Peng Jiang, Changwan Hong, and Gagan Agrawal. 2020. A Novel Data Transformation and Execution Strategy for Accelerating Sparse Matrix Multiplication on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 376–388. https://doi.org/10.1145/3332466.3374546

[20] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th*

International Conference on Learning Representations (Palais des Congrès Neptune, Toulon, France) *(ICLR '17)*. Palais des Congrès Neptune, Toulon, France. https://openreview.net/forum?id=SJU4ayYgl

[21] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A Tool to Generate Tensor Algebra Kernels. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 943–948.

[22] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* (2017).

[23] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alexander Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. *CoRR* abs/1903.12287 (2019). arXiv:1903.12287 http://arxiv.org/abs/1903.12287

[24] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press. http://www.mmds.org/

[25] G. Li, M. Muller, A. Thabet, and B. Ghanem. 2019. DeepGCNs: Can GCNs Go As Deep As CNNs?. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE Computer Society, Los Alamitos, CA, USA, 9266–9275. https://doi.org/10.1109/ICCV.2019.00936

[26] Guohao Li, Chenxin Xiong, Ali K. Thabet, and Bernard Ghanem. 2020. DeeperGCN: All You Need to Train Deeper GCNs. *CoRR* abs/2006.07739 (2020). arXiv:2006.07739 https://arxiv.org/abs/2006.07739

[27] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. 2020. G3: When Graph Neural Networks Meet Parallel Graph Processing Systems on GPUs. *Proc. VLDB Endow.* 13, 12 (2020), 2813–2816. http://www.vldb.org/pvldb/vol13/p2813-liu.pdf

[28] Meng Liu, Hongyang Gao, and Shuiwang Ji. 2020. Towards Deeper Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 338–348. https://doi.org/10.1145/3394486.3403076

[29] 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 19)*. USENIX Association, Renton, WA, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[30] D. Merrill and M. Garland. 2016. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 678–689. https://doi.org/10.1109/SC.2016.57

[31] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. 2017. Variational Dropout Sparsifies Deep Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, Sydney, NSW, Australia, 2498–2507.

[32] Sharan Narang, Eric Undersander, and Gregory F. Diamos. 2017. Block-Sparse Recurrent Neural Networks. *CoRR* abs/1711.02782 (2017). arXiv:1711.02782 http://arxiv.org/abs/1711.02782

[33] Nvidia. 2020. The API reference guide for cuSPARSE. https://docs.nvidia.com/cuda/cusparse/index.html

[34] Yosuke Oyama, Tal Ben-Nun, Torsten Hoefler, and Satoshi Matsuoka. 2018. Accelerating Deep Learning Frameworks with Micro-Batches. In *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*. IEEE Computer Society, Belfast, UK, 402–412. https://doi.org/10.1109/CLUSTER.2018.00058

[35] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. 2020. SpECK: Accelerating GPU Sparse Matrix-Matrix Multiplication through Lightweight Analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 362–375. https://doi.org/10.1145/3332466.3374521

[36] 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. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., 8026–8037. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[37] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA) *(KDD '14)*. Association for Computing Machinery, New York, NY, USA, 701–710. https://doi.org/10.1145/2623330.2623732

[38] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Mag.* 29, 3 (2008), 93–106. https://doi.org/10.1609/aimag.v29i3.2157

[39] Zequn Sun, Chengming Wang, Wei Hu, Muhao Chen, Jian Dai, Wei Zhang, and Yuzhong Qu. 2020. Knowledge Graph Alignment Network with Gated Multi-Hop Neighborhood Aggregation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, New York, NY, USA, 222–229. https://aaai.org/ojs/index.php/AAAI/article/view/5354

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[41] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, Vancouver, BC, Canada. https://openreview.net/forum?id=rJXMpikCZ

[42] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[43] 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. *SIGPLAN Not.* 51, 8, Article 11, 12 pages. https://doi.org/10.1145/3016078.2851145

[44] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. 2020. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020), 1–21. https://doi.org/10.1109/TNNLS.2020.2978386

[45] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, New Orleans, LA, USA. https://openreview.net/forum?id=ryGs6iA5Km

[46] Carl Yang, Aydin Buluç, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed*

Computing, August 27-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11014), Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer, Turin, Italy, 672–687. https://doi.org/10.1007/978-3-319-96983-1_48

[47] Carl Yang, Aydin Buluç, and John D. Owens. 2019. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *CoRR* abs/1908.01407 (2019). arXiv:1908.01407 http://arxiv.org/abs/1908.01407

[48] Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. 2019. Balanced Sparsity for Efficient DNN Inference on GPU. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, Honolulu, Hawaii, USA, 5676–5683. https://doi.org/10.1609/aaai.v33i01.33015676

[49] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, Addis Ababa, Ethiopia. https://openreview.net/forum?id=BJe8pkHFwS

[50] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. 2018. GaAN: Gated Attention Networks for Learning on Large and Spatiotemporal Graphs. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018*, Amir Globerson and Ricardo Silva (Eds.). AUAI Press, Monterey, California, USA, 339–349. http://auai.org/uai2018/proceedings/papers/139.pdf

[51] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. 2020. FusionStitching: Boosting Memory Intensive Computations for Deep Learning Workloads. *CoRR* abs/2009.10924 (2020). arXiv:2009.10924 https://arxiv.org/abs/2009.10924

[52] 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. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105. https://doi.org/10.14778/3352063.3352127

A Appendix

A.1 Abstract

This section is mainly the guideline to perform artifact evaluation for this paper. We first describe the directory tree of our code, which contains the code of both related work and our work. Then, we present the check-list for the evaluation. Finally, experiment workflow shows how to access the source code and how to use the scripts to perform detailed validation.

A.2 Description

The following is the directory tree of the code, which contains eight sub-directories as follows

A.3 Artifact check-list

- Algorithm: Forward phase of graph neural networks on GPU and the optimizations
- Program: CUDA and C/C++ code
- Compilation: nvcc10.1 with -O2 and –use_fast_math flag.
- Binary: CUDA executable
- Data set: Graphs for GNN tasks from Open Graph Benchmark (OGB)
- Run-time environment: Debian 4.19 with CUDA SDK 10.1 installed
- Hardware: Any NVIDIA GPUs with compute capability >=5.0 (Recommended GPU: NVIDIA Tesla V100-PCIe-32GB)
- Expected memory requirements to run the artifact: with 200GB main memory and 32GB GPU memory
- Expected time to run experiments (end-to-end): 2 hours
- Publicly available: Yes

A.4 Experiment workflow

For the convenience of the artifact evaluation, we only provide a few simple scripts in each sub-directory. Below are the steps to download our code, run the experiments, and observe the results.

A.4.1 Download the code. git clone https://github.com/xxcclong/GNN-Computing.git

A.4.2 Build and run. To build and run the code, follow the instructions of README.md in the code and reproduce the results shown in Section 5.