Artifacts Available acm V1.1
Artifacts Evaluated acm Functional V1.1

Understanding GNN Computational Graph: A Coordinated Computation, IO, and Memory Perspective

Hengrui Zhang [*] [1] **Zhongming Yu** [*] [1] **Guohao Dai** [1] **Guyue Huang** [2] **Yufei Ding** [2] **Yuan Xie** [2] **Yu Wang** [1]

Abstract

Graph Neural Networks (GNNs) have been widely used in various domains, and GNNs with sophisticated computational graph lead to higher latency and larger memory consumption. Optimizing the GNN computational graphs suffers from: **(1) Redundant neural operator computation.** The same data are propagated through the graph structure to perform the same neural operation multiple times in GNNs, leading to redundant computation which accounts for 92.4% of total operators. **(2) Inconsistent thread mapping.** Efficient thread mapping schemes for vertex-centric and edge-centric operators are different. This inconsistency prohibits operator fusion to reduce memory IO. **(3) Excessive intermediate data.** For GNN training which is usually performed concurrently with inference, intermediate data must be stored for the backward pass, consuming 91.9% of total memory requirement.

To tackle these challenges, we propose following designs to optimize the GNN computational graph from a novel coordinated computation, IO, and memory perspective: **(1) Propagation-postponed operator reorganization.** We reorganize operators to perform neural operations before the propagation, thus the redundant computation is eliminated. **(2) Unified thread mapping for fusion.** We propose a unified thread mapping scheme for both vertex- and edge-centric operators to enable fusion and reduce IO. **(3) Intermediate data recomputation.** Intermediate data are recomputed during the backward pass to reduce the total memory consumption. Extensive experimental results on three typical GNN models show that, we achieve up to 2.75× end-to-end speedup, 6.89× less memory IO, and 7.73× less memory consumption over state-of-the-art frameworks.

1 Introduction

Graph Neural Networks (GNNs) explore features of vertices and edges using neural operators and relationships through the graph structure. GNNs have shown great potentials in various domains, including Recommendation Systems (Ying et al., 2018; Wang et al., 2019a), Computer Vision (Yan et al., 2018; Qi et al., 2018), Natural Language Processing (Nguyen & Grishman, 2018; Yao et al., 2018), *et al* (Kipf & Welling, 2016; Hamilton et al., 2017).

With the fast development of GNNs, GNN models have evolved into more diversity and complexity in the computational graph, putting forward expensive requirements on both computation and memory resources. For example, training a GNN-based recommendation model consumes 16 GPUs (384 GB memory in total) using days of time (Ying et al., 2018). Improving the performance of GNNs with less resources suffers from: (1) From the computation perspective, GNN models perform neural operators through the graph structure, meaning that the same data of a vertex may be propagated to different edges. Thus, the same operation applied on these edges are executed multiple times for the same vertex data after propagation, leading to **redundant computation** in GNNs. We measure that such redundant computation account for 92.4% of total operators in an EdgeConv model (Wang et al., 2019c), with the detailed setup in Section 7. (2) From the IO perspective, current systems involve writing/reading graph-sized feature data between two graph operators. Operators performed on vertices and edges usually have **inconsistent thread mapping** schemes, which hinder applying fusion for these operators to reduce IO. (3) From the memory perspective, GNN models usually perform concurrent training and inference passes. Thus, **excessive intermediate data** produced during executing fused operators must still be stored for backward, leading to large memory space requirement. We measure in a Graph Attention Network (GAT) (Veličković et al., 2017) model that the intermediate data consume 91.9% of total memory.

To tackle these challenges and accelerate GNN computation with less memory consumption, we need a systematic GNN computational graph optimization framework which considers computation, IO, and memory. DGL (Wang et al.,

[*] Equal contribution [1] Tsinghua University [2] University of California, Santa Barbara. Correspondence to: Guohao Dai <daiguohao@mail.tsinghua.edu.cn>, Yu Wang <yu-wang@tsinghua.edu.cn>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

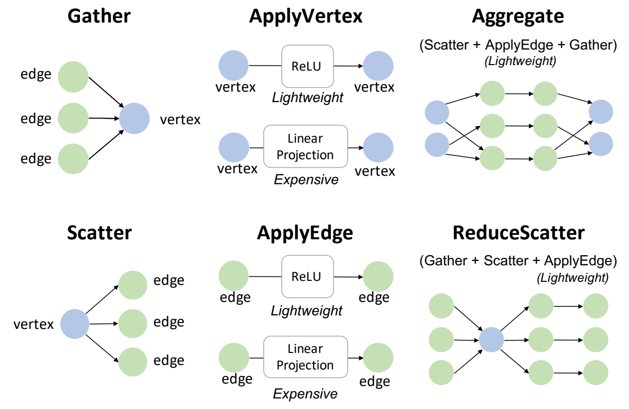


Figure 1. Operators in Graph Neural Networks (GNNs).

2019b) provides two high-level operators, gSpMM and gSDDMM, to express various GNN models, while such an abstraction fails to explore the redundant computation hidden in neural operators performed through the graph structure. FuseGNN (Chen et al., 2020) fuses edge operators to accelerate GNN computation, but it lacks the technique to fuse a vertex-centric operator with an edge-centric one. Huang *et al.,* (Huang et al., 2021) also proposes fusion technique for GNNs, while it cannot handle GNN training because the intermediate data are missing.

All previous studies fail to comprehensively consider the computation, IO, and memory perspectives for both GNN training and inference. Thus, we put forward a systematic framework to accelerate GNN computation and reduce memory consumption on GPUs with following contributions:

- **Propagation-postponed operator reorganization.** Since the redundant computation is caused by performing neural operators on graph structure, we reorganize operator to perform neural operations before propagation and achieve an average of 1.68× speedup.
- **Unified thread mapping for fusion.** Since different thread mapping scheme prohibits fusing vertex-centric and edge-centric operator and further reducing IO, we propose a unified thread mapping scheme for both types of operators and save up to 5.45× memory IO.
- **Intermediate data recomputation.** Since the intermediate data consume the majority of memory but are only stored for the backward pass, we introduce a recomputation mechanism to reproduce intermediate data just before they are needed for backward use and save the memory consumption by up to 2.21×.

We implement three popular GNN models with the techniques above, achieving up to 2.75× speedup, 6.89× less IO, and 7.73× less memory consumption. We even enable running large-scale GNN models with an NVIDIA RTX 2080 GPU (8 GB), which would require the newest NVIDIA RTX 3090 GPU (24 GB) without our technique, with a comparable latency.

Note that in this project, we mainly focus on single-GPU GNN computing, which is the key component of state-of-the-art GNN frameworks such as DGL (Wang et al., 2019b). We focus on GPUs because GPUs are the most commonly-used hardware platform for machine learning in the industry. And we focus on the setting of single-card for mainly two reasons: (1) Many GNN applications only have graphs that can easily fit into the memory of a single GPU, such as proteins or point clouds. (2) For those applications that cannot fit into a single card, such as social networks, there are already well-studied graph partition strategies that can cut the giant graph into small subgraphs so that they can fit into a single device. NeuGraph (Ma et al., 2019) utilizes a straight forward partitioning by tiling the adjacency matrix into equally-sized chunks. ROC (Jia et al., 2020) introduces an online learning strategy based on a cost model that predicts the execution time to discover balanced partitioning. DistGNN (Md et al., 2021) adopt a minimum vertex-cut graph partitioning algorithm and communication avoidance with delayed-update algorithms to optimize GNN training on CPU clusters. As GNN training on multi-card can be divided into out-of-GPU graph partitioning and in-GPU GNN computing, the overall performance still largely depends on the performance of a single GPU, and multi-card GNN training can also benefit from our optimizations.

The following of this paper is organized as follows. Section 2 introduces preliminaries of GNNs. Section 3 introduces an overview of our optimization recipe. Our three techniques, propagation-postponed operator reorganization, unified thread mapping for fusion, and intermediate data recomputation are detailed in Section 4, 5, and 6, respectively. Section 7 presents evaluation results. Section 8 elaborates related work. Section 9 concludes the paper.

2 Preliminaries

2.1 GNN Operators

On a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with the set of vertices $\mathcal{V}$ and edges $\mathcal{E}$, a GNN layer is composed of the following operators:

$$\begin{aligned} m_e &= \texttt{Scatter}(h_v, h_u), (u, e, v) \in \mathcal{E}, \\ m_e^{new} &= \texttt{ApplyEdge}(m_e[, m_e', \cdots]), \\ h_v &= \texttt{Gather}(\{m_e : (u, e, v) \in \mathcal{E}\}), \\ h_v^{new} &= \texttt{ApplyVertex}(h_v[, h_v', \cdots]). \end{aligned}$$

In the above equations, v, u are vertex indices and e is an edge index. h_v refers to feature attached to vertex v, and m_e attached to edge e.

Figure 1 visualizes the definitions of operators. `Gather` is a reduction operation that generates the feature of a vertex from features of edges connecting to it. `Scatter` generates the feature of an edge from features of vertices that the edge

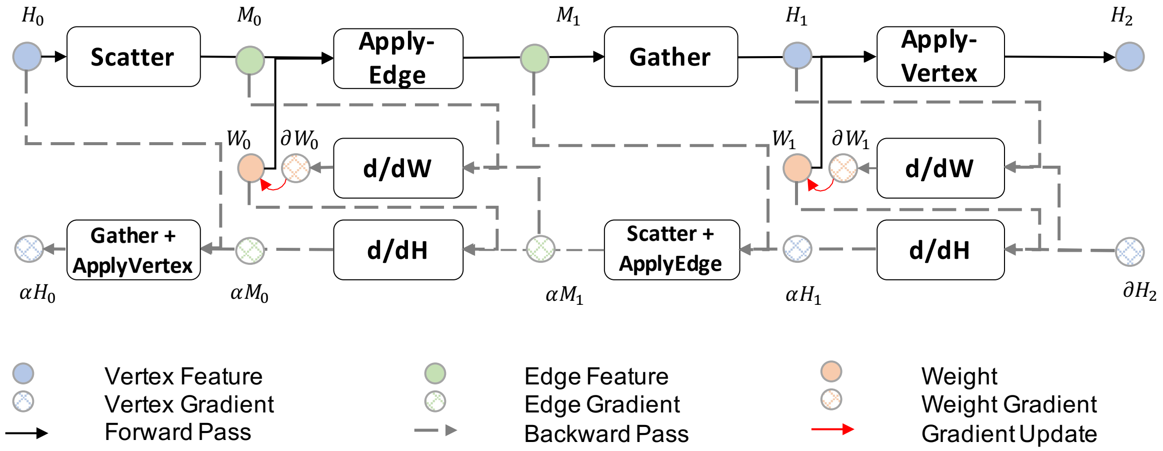


Figure 2. Dataflow in GNN training, showing both the forward pass (top) and backward pass (bottom). All intermediate features are used for backward and need to be stashed in the memory.

connects to. ApplyEdge and ApplyVertex are graph-irrelevant operators that transform the features of each edge and vertex, respectively. We further categorize Apply- operators based on their computation cost: element-wise operations are considered as lightweight Apply-, while computation-intensive operations like linear projection are considered expensive Apply-.

The four operators above are comprehensive enough to express any GNN model, but there are some widely-used combinations of operators, which current GNN systems also provide dedicated optimizations to. We name two most common combinations: Aggregate and ReduceScatter, as defined below. We add them to our operator abstraction operators for the convenience of expressing models.

$$\begin{aligned} h_v^{new} &= \texttt{Aggregate}(\{(h_u, m_e) : (u,e,v) \in \mathcal{E}\}, h_v) \\ &= \texttt{Gather}(\{\texttt{ApplyEdge}(\texttt{Scatter}(h_v, h_u), m_e)\}) \\ m_e^{new} &= \texttt{ReduceScatter}(\{m_{e'} : (u \in N(v), e', v) \in \mathcal{E}\}, h_u) \\ &= \texttt{ApplyEdge}(\texttt{Scatter}(\texttt{Gather}(\{m_{e'}\}), h_u), m_e), \\ &\quad (u,e,v) \in \mathcal{E} \end{aligned}$$

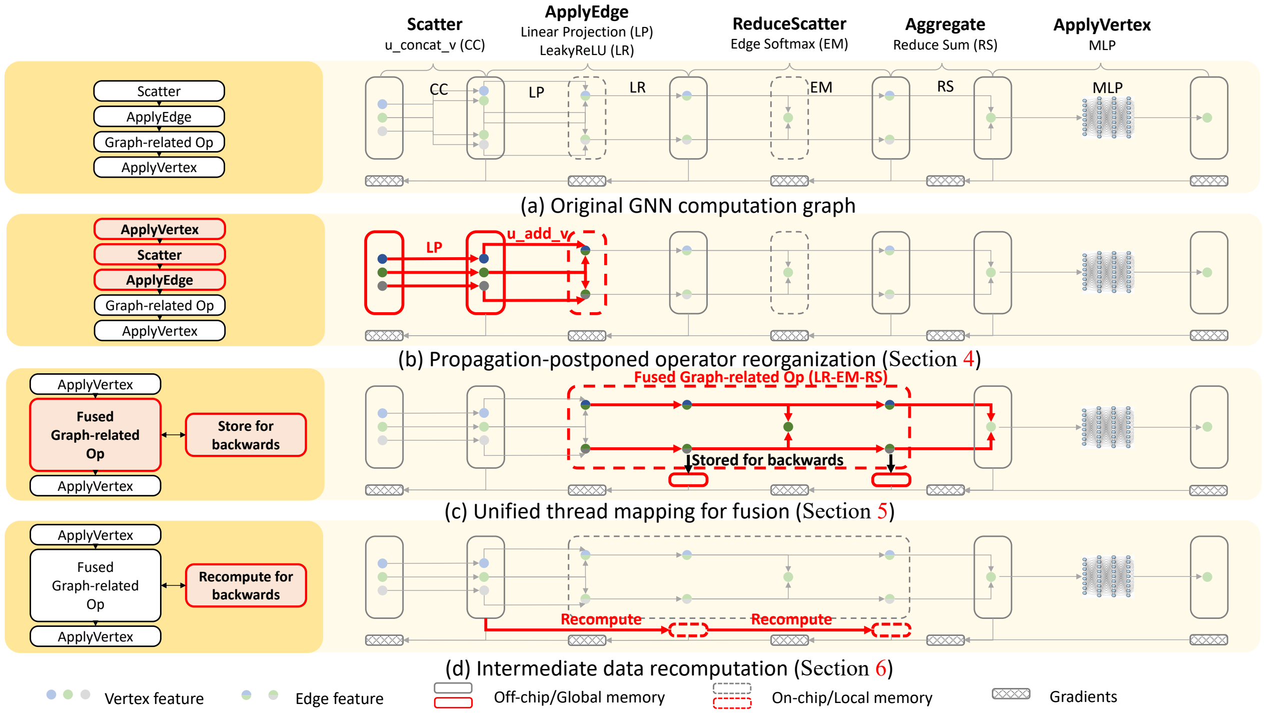
Aggregate generates a new vertex feature by reducing features from its neighbor vertices and edges. ReduceScatter generates a new edge feature by reducing and scattering among the group of edges that connect to the same vertex, a typical example being the edge-softmax operation in the Graph Attention Network (GAT). Current GNN systems widely support fused Aggregate and ReduceScatter implementations when ApplyEdge is lightweight (Huang et al., 2020; Wang et al., 2019b).

Compared with related work, our operator abstraction is both comprehensive and optimization-friendly. In terms of comprehensiveness, the Aggregation-Combination abstraction in previous work (Yan et al., 2020; Wang et al., 2021), equivalent to our Aggregate and ApplyVertex, does not cover ApplyEdge. Therefore, the Aggregation-Combination can only express GNN models without applying neural networks to edge features, such as the vanilla GCN (Kipf & Welling, 2016) or GraphSAGE (Hamilton et al., 2017). Our proposed operator abstraction, in constrast, can construct whatever Aggregation-Combination constructs, and also Graph Attention Network (GAT) (Veličković et al., 2017), EdgeConv (Wang et al., 2019c), and other models with arbitraty message-passing procedure. Figure 3(a) shows how to construct GAT using our operator abstraction, and the construction of more GNN models are elaborated in Appendix to demonstrate its comprehensiveness. In terms of optimization convenience, the abstraction in DGL (Wang et al., 2019b), gSDDMM and gSpMM, can be lowered to any operator-combination that outputs edge and vertex features. respectively. Such general abstraction hinders chances of local, global or adaptive optimizations, e.g. optimizing only Gather part, or fusing the last Scatter in gSDDMM with first Gather in gSpMM. DGL leverages a limited set of built-in operators to tackle optimization challenges in such a general abstraction. On the contrary, this paper uses a fine-grained operator abstraction to express GNN models for generality, and leverage inter-operator optimizations to systematically improve performance.

2.2 Back-Propagation in GNN

The back-propagation algorithm is applied to train GNN models. One can prove that the backward pass of above set of operators still fall into this set. We list the key conclusions below, while the detailed proof is elaborated in Appendix.

- The backward pass of Gather (Scatter) is Scatter and ApplyVertex (Gather and ApplyEdge).
- The backward pass of ApplyEdge (ApplyVertex) is two ApplyEdge (ApplyVertex) operations.



(a) Original GNN computation graph

(b) Propagation-postponed operator reorganization (Section 4)

(c) Unified thread mapping for fusion (Section 5)

(d) Intermediate data recomputation (Section 6)

Figure 3. Design Overview. The left part shows the high-level abstraction of a typical GNN computation flow, and the right part shows the example of a GAT (Veličković et al., 2017) model when applying techniques proposed in this paper. (a) The original computation graph. (b) After applying operator reorganization, the linear projection operator is preposed and applied on vertices to reduce computation. (c) After applying operator fusion with the unified thread mapping scheme, operators are fused while the intermediate data are still stored for the back propagation phase. (d) After applying recomputation, intermediate data are not required to be stored.

The backward pass of `Aggregate` (`ReduceScatter`) can be analyzed by decomposing them into the four fine-grained operators. In summary, we can express both forward and backward pass of GNN using the same operator abstraction. Figure 2 shows a dataflow graph with both passes and expressed by the four basic operators. Figure 2 also shows that the intermediate features are needed for computing gradients in the backward pass of the same layer. Therefore, during the forward pass, all intermediate features must be stashed and later used to calculate parameter gradients in the backward pass. Take the GAT models an example, Figure 3(a) marks all the feature tensors that are stashed and where they are used in the backward pass. State-of-the-art GNN systems follow the general strategy of saving outputs of all operators in the model, and only provide fused implementations for some common operator-combinations to avoid saving an $O(|\mathcal{E}|)$ intermediate tensor (e.g., DGL's built-in edge-softmax for GAT), however, a general approach for reducing memory consumption in training is lacked.

3 Design Overview

This paper proposes a systematic approach to optimize the GNN at inter-op level. In this section, we provide an overview of our designs by walking-through them on the model architecture of Graph Attention Network (GAT) (Veličković et al., 2017). As Figure 3(a) shows, a GAT layer is composed of `Scatter`, `ApplyEdge`, `ReduceScatter`, `Aggregate`, ended by an `ApplyVertex`. We tackle the aforementioned three design challenges with methods below:

Eliminating redundant computation through propagation postponed reorganization. Recall that the redundancy is caused by performing expensive `Apply-` (e.g. linear transformations) many times on the features propagated from the same source. We propose to reorder the operator sequence to eliminate this redundancy by first applying expensive `Apply-` on the vertex features, and then propagating the results to edges. For example, we show that in GAT (Veličković et al., 2017) models, the `Scatter-ApplyEdge` operator sequence can be substituted by linear-projection on vertex features and then `Scatter`-ing the result, as shown in Figure 3(b).

Reducing IO through completely fusing graph-related kernels. We propose to fuse a sequence of operators as long as they are graph-related kernels or lightweight `Apply-`. We choose not to fuse expensive `Apply-` like linear transformations because they can often be tackled with primitives in highly-optimized libraries, e.g. cuBLAS or cuDNN. The challenge here is that vertex-centric and edge-centric operators, i.e. operators that produce vertex- or edge-features, apply vertex-balanced and edge-balanced thread mapping in current GNN systems, respectively. The unmatched thread mapping schemes prohibit reusing intermediate data locally and force dumping data to the DRAM. With novel kernel designs, we show vertex- and edge-balanced thread mapping can both be applied no matter the operator produces vertex- or edge-features. This allows us to choose a unified thread mapping for a sequence of graph-related kernels to fuse them. As shown in Figure 3(c), this step fuses operators like `Scatter`, `ReduceScatter`, `Aggregate` into one single kernel and greatly reduces IO.

Avoiding saving intermediate data for backward pass through recomputation. Recall that GNN training requires saving all intermediate features in the forward pass for computing gradients, leading to excessive memory consumption. We borrow the idea of gradient checkpointing in DNN training to tackle this challenge. We selectively save the intermediate features in the forward pass (checkpoints), and recompute the unsaved features just before they are needed in the backward pass, as shown in Figure 3(d). The non-checkpoint features originally requires a memory size of $O(f \times |\mathcal{E}|)$, where f stands for the feature length and $|\mathcal{E}|$ is the number of edges. With recomputation and the aforementioned kernel-fusion technique, we can eliminate this $O(f \times |\mathcal{E}|)$. To maximize the benefit of memory savings and minimize the cost of recomputation, we choose to recompute edge rather than vertex features.

4 Reducing Computation: Propagation-postponed Operator Reorganization

Motivation. Many GNN models perform `Scatter` followed by a computation-intensive neural network (NN) as `ApplyEdge`. The same vertex feature is propagated to all of its adjacent edges, and this duplication causes repeated NN computation in the `ApplyEdge` step.

Challenge. We describe above the intuition why propagation + NN operator causes redundant computation, but we lack a general formulation to identify and eliminate such redundancy. In particular, `Scatter` involves both redundant computation and per-edge unique computation: the redundant part is because multiple edges connected to the same vertex share identical vertex feature as input, and the unique part is because each edge combines features from a unique pair of two vertices. Separating the two parts and reducing the redundant part require a careful surgery on the original computational graph.

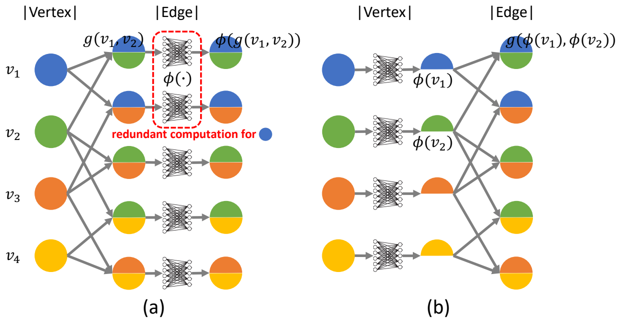


Figure 4. Diagram of the propagation-postponed operator reorganization. (a) Redundant neural operator computation on a same vertex. (b) Operation reorganization to postpone graph operators and eliminate redundant computation.

Insight. Our key insight is that the root of this possible computation redundancy is performing repeated neural computation on features scattered from the same source. Take figure 4(a) as an example. Figure 4(a) shows the computation and data flow for a part of one EdgeConv layer with one `Scatter` operator followed by an `ApplyEdge` operator. Features on vertices are first scattered to edges with function $g(u, v) = u - v$, after that a linear-projection function $\phi(\cdot)$ is applied. Vertex features are scattered and applied $\phi(\cdot)$ independently on different edges. Therefore we might apply $\phi(\cdot)$ to the same feature more than once, which causes possible redundancy in computation.

Approach: identify redundancy. Following our insight that the possible redundancy occurs in the `Scatter-ApplyEdge` phase, we find a sufficient condition to identify this possible redundancy in computing: if the Scatter function g and `ApplyEdge` function ϕ follows the **commutative law** and **distributive law**, there is redundancy in this Scatter-ApplyEdge procedure. Take figure 4(a) as an example. We first compute $g(v_1, v_2)$ and $g(v_1, v_3)$ during Scatter, then compute $\phi(g(v_1, v_2))$ and $\phi(g(v_1, v_3))$. Under the commutative law and distributive law, we obtain $\phi(g(v_1, v_2)) = g(\phi(v_1), \phi(v_2))$ and $\phi(g(v_1, v_3)) = g(\phi(v_1), \phi(v_3))$. Therefore, we actually compute $\phi(v_1)$ more than once. For the whole procedure, the computational expensive function $\phi(\cdot)$ is computed $|\mathcal{E}|$ times.

Approach: eliminate redundancy. We propose propagation-postponed operator reorganization to eliminate this redundancy while keeping functional equivalence. The main idea is, as the redundancy is caused by edges that share the same source performing transformation to the same feature, if we postpone `Scatter` and perform `ApplyFunction` first, we will only perform transformation to the same feature for only once. In figure 4(b), we first compute $\phi(v_1)$, $\phi(v_2)$ and $\phi(v_3)$, then scatter them to edges to compute $g(\phi(v_1), \phi(v_2))$ and $g(\phi(v_1), \phi(v_3))$, which actually change the execution order from `Scatter-ApplyEdge` to `ApplyVertex-Scatter`. For the whole procedure, function g is still computed $|\mathcal{E}|$ times, but the computational expensive function $\phi(\cdot)$ is computed only $|\mathcal{V}|$ times. In most cases, g is arithmetic operator and $\phi(\cdot)$ is linear operator, which means the distributive law and commutative law are met, and we can always eliminate this redundancy with operator propagation-postponed operator reorganization.

Example. In GATConv, the `Scatter-ApplyEdge` computes the attention score between two vertices by concatenating and applying a one-layer neural network mechanism, as in Equation 1:

$$e_{u\to v} = \text{LeakyReLU}\left(\vec{a}^T \left[\vec{h}_u \| \vec{h}_v\right]\right) \quad (1)$$

where $\vec{h}_u, \vec{h}_v \in \mathbf{R}^f$ are the feature vector of the destination and source, and $\vec{a} \in \mathbf{R}^{2f}$ are weight parameters to learn.

As Figure 3(a) shows, a `Scatter` operator u_concat_v is first applied to propagate features to edges and concatenate the feature vector of the source and destination into $\left[\vec{h}_u \| \vec{h}_v\right]$, followed by a LP (Linear Projection) and LeakyReLU to compute the final attention score. The computation cost is $2|\mathcal{E}|f$ for u_concat_v, $4|\mathcal{E}|f$ for LP and $|\mathcal{E}|$ for LeakyReLU , with a total of $6|\mathcal{E}|f + |\mathcal{E}|$.

Although concatenate and the non-linear neural operation do not follow the commutative and distributive law, we find that the LP and concatenate can be seen as two LP followed by an add: $\vec{a}^T \left[\vec{h}_u \| \vec{h}_v\right] = \left[\vec{a_l}^T \| \vec{a_r}^T\right] \left[\vec{h}_u \| \vec{h}_v\right] = \vec{a_l}^T \vec{h}_u + \vec{a_r}^T \vec{h}_v$. Therefore, there is redundancy in computation if we don't perform operator reorganization, and we postpone Scatter and change the execution order from `Scatter-ApplyEdge` into `ApplyVertex-Scatter`. As shown in Figure 3(b), we first apply LP to features on vertices, then scatter them to edges and perform add, followed by a LeakyReLU, which still need to be applied on edges. The total computation cost is reduced to $4|\mathcal{V}|f + 2|\mathcal{E}|$.

5 Reducing IO: Unified Thread Mapping for Fusion

Motivation. GNN systems suffer from excessive global memory writing/reading between production-consumption operators. Take the GAT model in Figure 3 as an example: the edge features produced by the `ApplyEdge` step needs to be written-out to the global memory, and read-in again by the next `ReduceScatter` operator. The output of `ReduceScatter` step is again stored and loaded by the succeeding `Aggregate` kernel. Both procedures involve writing/reading a $O(|\mathcal{E}|)$-sized feature tensor. Kernel fusion is widely exploited to reduce the data movement. In fact, the edge-softmax in current systems are commonly implemented by a hand-optimized fused kernel to reduce IO. Our target is to apply kernel fusion to further eliminate the aforementioned two edge feature store/load, and completely fuse all graph-related operators (`Scatter`, `ApplyEdge`, `ReduceScatter`, `Aggregate`).

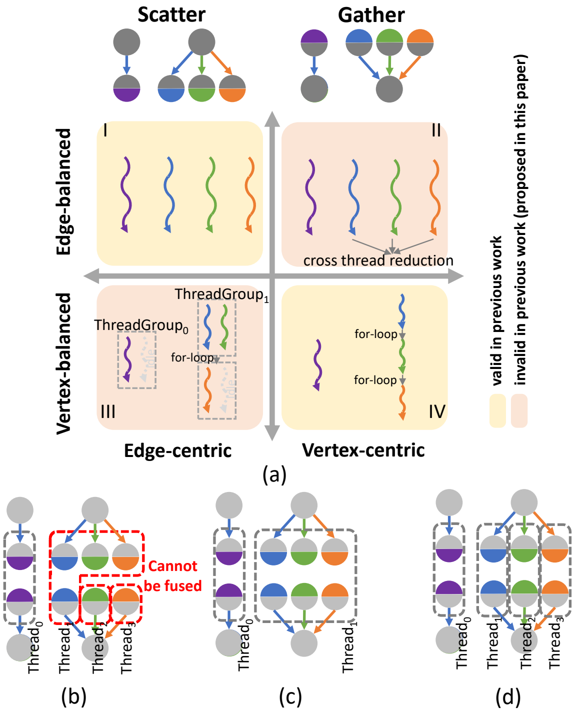


Figure 5. Diagram of the unified thread mapping. (a) We enable different thread mapping schemes for different graph operators. (b) A `Scatter` with the edge-balanced mapping cannot be fused with a `Gather` with the vertex-balanced mapping. (c) Vertex-balanced fusion. (d) Edge-balanced fusion.

Challenge. The challenge in applying fusion to graph-related operators is the diverged thread-mapping schemes between edge-centric and vertex-centric operators. By edge-centric, we mean the operator whose output is edge features, and by vertex-centric the ones producing vertex features. For example, `Scatter` is an edge-centric operator, `Gather` being vertex-centric, and `ReduceScatter` and `Aggregate` are hybrid of both. We find current GNN systems commonly implement edge-centric operators in edge-balanced thread-mapping, and vertex-centric ones in vertex-balanced thread mapping. As shown in Figure 5(a)I, edge-balanced thread mapping bind parallel workers to different edges. This parallelization strategy naturally matches the edge-centric operator: imagine each worker independently calculate the features for different edges, with no cross-thread communication involved and perfect work-balancing. On the other hand, vertex-balanced thread mapping bind parallel workers to different vertices. This strategy suits the `Gather` operator because the reduction can be carried by the same worker via a sequential loop as Figure 5(a)IV. Although the above two strategies are reasonable when seen separately, the issue comes up when we try to fuse operators with different thread-mapping schemes. As shown in Figure 5(b), the edge-balanced scheme in `Scatter` and the vertex-balanced scheme in `Gather` prohibits reusing the intermediate data in the thread's local scope, because the same thread is assigned to an edge at first but a vertex next.

Insight. Our key insight is that thread-mapping schemes can be decoupled from the operator type: edge-centric operator can also apply vertex-balanced mapping and vise versa. We illustrate these two scenarios in Figure 5(a)II and III. To apply vertex-balanced mapping to edge-centric operator, each worker is assigned to loop over the incoming-edge set of a vertex and compute features for these edges. We can increase the number of threads in the same group to exploit parallelism, since features for each edge can be calculated in parallel. The example in Figure 5(c) reveals a potential issue of imbalanced workload, but the issue is minor as long as we have enough parallelism to fully occupy the GPU, and worth taking if it enables kernel fusion and saves excessive IO. On the other hand, when applying edge-balanced mapping to vertex-centric operator, we need to handle the cross-thread reduction shown in Figure 5(d). Cross-thread reduction can be implemented on GPU via atomic arithmetics. Observe that edge-balanced mapping improves workload balancing, but atomic arithmetics can introduce overhead, which we need to compare against the benefit of kernel fusion.

Approach. Following our insight that both edge-balanced and vertex-balanced schemes can be applied to all operators, we propose to eagerly fuse all graph-related operators with unified thread mapping. By the phrase graph-related, we refer to all operators except the expensive `Apply-` ones such as linear projection. In the GAT example, the sequence of `Scatter`, `ReduceScatter`, `Aggregate` all fall into this definition, and we are able to fuse them into one single kernel by applying the same thread-mapping. In general, we can select between vertex-balanced or edge-balanced mapping based on performance profiling. A special case is when `ReduceScatter` is involved: since an intermediate vertex-feature needs to be reused between two operators, we can only apply the vertex-centric mapping and buffer the vertex-feature in the GPU shared-memory.

Example. In GAT, there are three graph-related operators that have a potential to fuse: `Scatter`, `ReduceScatter` and `Aggregate`. As `ReduceScatter` requests vertex-centric mapping, we apply unified vertex-balanced mapping to fuse these three operators into one kernel, which saves excessive IO. Assuming one GAT layer has h heads and a feature length of f, before operator fusion, the IO of these graph-related operators is $4|\mathcal{E}|h$ for `Scatter`, $3|\mathcal{E}|h$ for `ReduceScatter`, and $3|\mathcal{E}|hf + |\mathcal{V}|hf$ for `Aggregate`, with a total of $|\mathcal{V}|hf + 7|\mathcal{E}|h + 3|\mathcal{E}|hf$. With operator fusion, since the intermediate data are reused, the total IO is reduced to $|\mathcal{V}|hf + 5|\mathcal{E}|h + 2|\mathcal{E}|hf$.

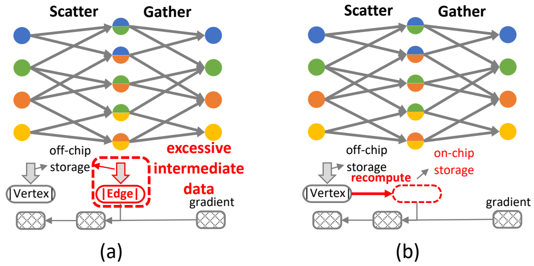


Figure 6. Diagram of the intermediate data recomputation. (a) Edge features are stored for the backward propagation. (b) Edge features are recomputed without storing in the off-chip memory.

6 Reducing Memory: Intermediate Data Recomputation for Training

Motivation. GNN systems suffer from excessive memory consumption, because all the intermediate feature tensors are saved for the backward pass. Section. 5 described our techniques to fuse all graph-related operators in the forward pass. Fusion saves not only IO but also memory since no intermediate tensors need to be written-out and read-in. We intend to extend operator fusion for the back-propagation based training scenario to reduce memory consumption.

Challenge. The challenge of avoiding saving intermediate data is back propagation. The role of intermediate data is two folds: (1) it passes the values on the forward computational graph; (2) it passes the intermediate features in the forward pass to the backward computational graph for gradients computing. We can fuse operators both in forward and backward pass, which solves (1). But this is not enough for training, as intermediate data are still needed for backward.

Take Figure 6(a) as an example, which shows a toy example composed of one `Scatter` step and one `Gather` step, with operator fusion technique already applied. For the forward pass, we've successfully eliminated the $O(|\mathcal{E}|)$ intermediate data produced by `Scatter` with operator fusion technique by fusing the `Scatter-Gather` procedure into one operator, in which the values of the intermediate data are temporarily stored in on-chip memory instead of the off-chip memory. But as we still need this intermediate data for backward propagation, we have to stash the intermediate data in off-chip memory.

Insight. Our key insight is that we can trade memory with computation: if the intermediate data is memory consuming but light weight to compute, we can recompute these intermediate data during the backward pass. Based on this, we propose a recomputing technique to deal with the intermediate data in the backward pass, which solves (2).

Approach. Following our insight that memory can be traded with computation, we propose an empirical criterion $\frac{ComputationCost}{MemoryCost}$ to identify the recomputing opportunity of an operator. If $\frac{ComputationCost}{MemoryCost}$ is no more than $O(1)$, which means we can save one element's memory with no more than one computation, we just recompute the value during the backward pass, because we can save memory with little damage to the runtime latency. Otherwise, we stash the intermediate data as the benefit of recomputing is limited. In the toy example in figure 6(b), we recompute the $O(|\mathcal{E}|)$ intermediate data instead of stashing it because the computation cost of `Scatter` is small. By recomputing, we save $O(|\mathcal{E}|)$ memory consumption with $O(|\mathcal{E}|)$ computation. We will show later by experiments that this overhead is usually no more than 10% in GNN.

Recomputation with fusion. Our recomputing technique usually works for graph-related operators and lightweight `Apply-` operators, which take up much memory space but lightweight to compute. Occasionally, our proposed fusion technique is also applied to graph-related operators and lightweight `Apply-` operators. If we perform fusion without recomputation, we have to stash those needed intermediate data, which still costs a lot of memory space and impair the benefits brought by fusion. With fusion-recomputation combo, we are able to eliminate those intermediate data in the whole training process.

Example. In GAT, three operators are fused: one `Scatter`, one `ReduceScatter` (edge-softmax), and one `Aggregate`. So there are two intermediate data we need to handle: output of `Scatter` and output of $ReduceScatter$, both of which are $O(|\mathcal{E}|)$. As the $\frac{ComputationCost}{MemoryCost}$ of this `Scatter` is only $O(1)$, we can just recompute it during backward propagation. The `ReduceScatter` operator edge-softmax first perform reduction to compute the maximums and the sum of all the exponential as denominator, which is a `Gather`, followed by a $O(1)$ division to compute the final edge value (`Scatter` and `ApplyEdge`). The recomputing score $\frac{ComputationCost}{MemoryCost}$ is $O(|log\frac{|\mathcal{E}|}{|\mathcal{V}|}|)$ for `Gather` and $O(1)$ for `Scatter` and `ApplyEdge`. According to our standard for recomputing, we store all the maximums and denominators during the forward pass, which only takes $O(|\mathcal{V}|)$, and recompute the other results later within $O(1)$ time. By our proposed recomputing technique, two $O(|\mathcal{E}|)$ intermediate data are eliminated at a cost of only $O(1)$ overhead in latency.

7 EXPERIMENT

In this section, we implement our proposed techniques and evaluate them on multiple GNN models and datasets. We (1) demonstrate the overall performance improvements; (2) conduct ablation studies to provide detailed analysis on the benefits brought by each technique; (3) evaluate our implementations on devices with smaller DRAM which wouldn't fit in without our optimization.

7.1 Experimental Setup

7.1.1 Benchmarks

- **Graph Attention Network (GAT)** (Veličković et al., 2017) is one of the most classic GNN models, which adopts attention mechanisms to learn the relative weights between connected vertices instead of the identical or predetermined weights. It first `Scatter` features to edges and compute attention scores with learnable parameters, then perform `ApplyEdge` followed by `Aggregate`.
- **Edge Convolution (EdgeConv)** (Wang et al., 2019c) transforms the point clouds into a k-nearest neighbor graph to represent the topological information, in which points are viewed as vertices and their relative position is modeled as edges. It first `Scatter` vertex features to edges to compute their relative position, then `Apply` neural operations on edges and performs `Gather` to generate vertex embeddings.
- **Mixture Model Network (MoNet)** (Monti et al., 2016) introduces pseudo-coordinates to determine the relative position among vertices to learn the weight function adaptively. It first performs `ApplyEdge` to compute gaussian kernel, followed by `Aggregate`.

We choose these models because we believe they represent the trend that GNN models will evolve into more diversity and complexity, from static edge value without gradient (Kipf & Welling, 2016; Hamilton et al., 2017) to gradient computation on edge feature (Veličković et al., 2017; Monti et al., 2016; Wang et al., 2019c), which improves the expressivity of GNNs.

7.1.2 Baselines

- **Deep Graph Library (DGL)** (Wang et al., 2019b) is one of the mainstream GNN framework on GPUs, which adapts to existing deep learning software such as PyTorch. It outweighs PyG (Fey & Lenssen, 2019) in various GNN models. (Chen et al., 2020)
- **FuseGNN** (Chen et al., 2020) is a system for GNN training on GPUs with efficient CUDA kernel implementations and applies operator fusion technique. As fuseGNN does not implement EdgeConv and MoNet, we only compare with it on GAT.

7.1.3 Datasets

For GAT and MoNet, we use four commonly-used GNN datasets for evaluation, including Cora, Citeseer, Pubmed, and Reddit (Kipf & Welling, 2016; Hamilton et al., 2017). For EdgeConv, we use ModelNet40 classification task with 12,311 meshed CAD models from 40 categories, consisting in predicting the category of a previously unseen shape (Wu et al., 2015; Wang et al., 2019c).

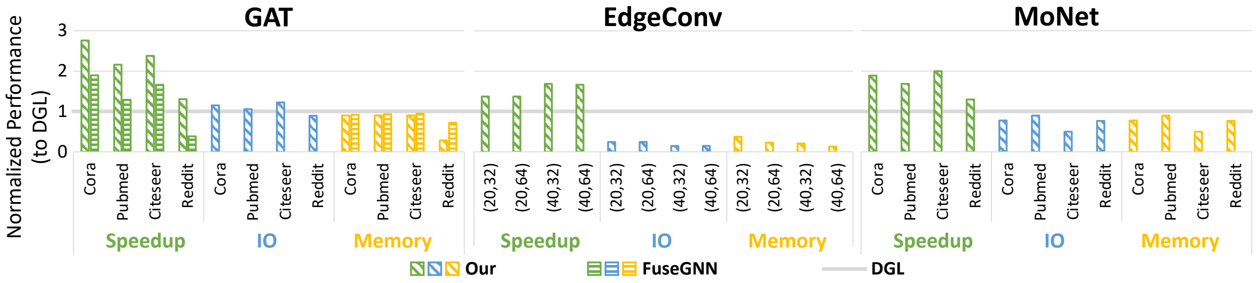


Figure 7. Normalized (to DGL) end-to-end performance on three GNN models from computation (speedup), IO, and memory perspectives.

7.1.4 *Platforms & Metrics*

We implement our proposed technique with a C++ and CUDA backend and a Pytorch-based front-end. Our main evaluation platform is a server with a 10-core 20-thread Intel Xeon Silver 4210 CPU running @ 2.2GHz and an NVIDIA RTX 3090 GPU with CUDA 11. Besides, we use an NVIDIA RTX 2080 GPU to demonstrate our design can achieve comparable performance against RTX 3090.

7.2 End-to-End Performance

GAT. As fuseGNN doesn't support multi-head attention, we use the setting: 2 layers with 128 hidden dimensions for evaluation and the end-to-end training results are shown in Figure 7. Compared with DGL, we achieve an average of 2.07× (up to 2.75×) speedup and save an average of 1.48× (up to 3.53×) memory consumption. Compared with fuseGNN, we achieve an average of 1.85× (up to 3.41×) speedup and save an average of 1.29× (up to 2.55×) less memory consumption. The average IO is increased by 1.3% due to recomputation. On Cora, Citeseer and PubMed, we achieve great speedup mainly because we perform unified vertex-balanced fusion, which is friendly for these datasets. The memory consumption is not greatly saved because what we eliminate is the $O(|\mathcal{E}|)$ intermediate data and the number of edges is small in these datasets. But on Reddit with 233K vertices and 115M edges, we save great memory consumption (3.88GB) compared with DGL (13.7GB) and fuseGNN (9.89GB) mainly because our proposed fusion-recomputation combo eliminates the $O(|\mathcal{E}|)$ intermediate data during training. The memory saving will be more significant if applying multi-head mechanism as in the original paper (Veličković et al., 2017).

EdgeConv. We use the same setting as the original paper (Wang et al., 2019c): EdgeConv layers=4 with hidden dimensions={64, 64, 128, 256}, the number of nearest neighbors k=20/40, and the batch size=32/64, with a total of four different settings, and the end-to-end training results are shown in Figure 7. Compared with DGL, we achieve an average 1.52× (up to 1.69×) speedup and save an average of 4.58× (up to 7.73×) peak memory usage and 5.32× (up to 6.89×) IO. We apply operator organization and operator fusion technique in EdgeConv. As the `Gather` function is max, only an $O(|\mathcal{V}|)$ array is needed for back propagation, and recomputation is not applied to further reduce memory consumption. Note that the training process of EdgeConv consists of two parts: transforming point clouds into a graph on CPU and GNN computing on GPU. As a great portion of the computation is transforming point clouds into a graph, the end-to-end speedup is not as significant as it should be. However, the memory is largely saved because we optimize the graph-related operators which cause large memory consumption. Note that our memory consumption remains unchanged when k changes, for k is the average number of edges for each vertices. By implementing fusion-recomputation combo, we eliminate all the $O(|\mathcal{E}|)$ intermediate data.

MoNet. We use the setting: 2 layers with 16 hidden dimensions, k=3 r=2 for Cora, k=3 r=3 for Pubmed and Citeseer, k=2 r=1 for Reddit, where k is the gaussian kernel size and r is the dimension for pseudo coordinates in gaussian mixture model. As shown in Figure 7, compared with DGL, we achieve an average of 1.69× (up to 2.00×) speedup and save an average of 1.47× (up to 3.93×) peak memory usage and 1.30× (up to 2.01×) IO. Similar with GAT, the performance improvement comes from operator fusion and recomputing. Different from GAT, as MoNet doesn't have `Scatter` in the beginning, operator reorganization is not needed.

7.3 Ablation Studies

Without special declaration, we use the setting as follows. (1) GAT: head=4 with feature dimension=64, on Reddit. (2) EdgeConv: k=40, batch size=64, layer=4 with hidden dimensions={64, 64, 128, 256} for training, layer=1 with feature dimensions=64 if only forward. (3) MoNet: k=2, r=1 with feature dimension=16, on Reddit.

Reorganization. Figure 8 illustrates the benefits of operator reorganization for reducing computation, IO, and memory consumption in GAT and EdgeConv. MoNet has no `Scatter` and therefore no need for operator reorganization. Due to memory limitation of our device, we evalu-

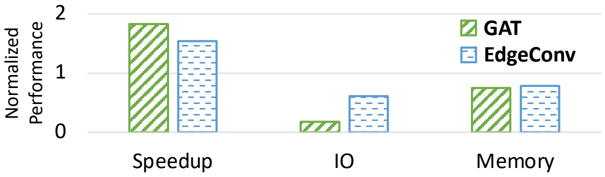


Figure 8. Normalized performance improvements brought by propagation-postponed operator reorganization.

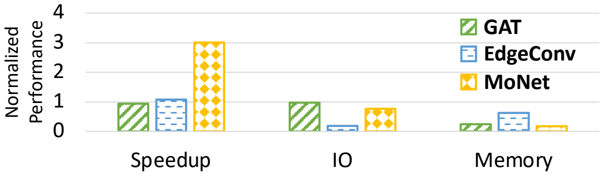


Figure 9. Normalized performance improvements brought by unified thread mapping operator fusion.

ate GAT with Pubmed. The baseline is implemented with Scatter before ApplyEdge, and the implementation with operator reorganization postpone Scatter and perform ApplyVertex first. To clearly show the impacts brought by operator reorganization, We use forward pass for comparison. The experiment results are consistent with theoretical analysis: as redundant computation is eliminated, latency is reduced and redundant IO caused by redundant computation is also eliminated; as we perform ApplyVertex before Scatter, one $O(|\mathcal{V}|)$ and one $O(|\mathcal{E}|)$ intermediate data are generated, but if we perform Scatter first followed by ApplyEdge, two $O(|\mathcal{E}|)$ intermediate data are generated. For the forward pass, operator reorgnization improves latency by 1.68×, IO by 3.06×, and peak memory usage by 1.30× on average.

Fusion. Figure 9 illustrates the benefits brought by operator fusion. We fuse all the graph-related operators with our proposed unified thread mapping scheme, and our proposed fusion technique can be applied to all of these three models. More details about our implementation can be found in appendix. For GAT, fusion has a little negative impact on latency, slightly reduces IO and greatly reduces memory consumption. As we use shared memory to perform operator fusion, which introduces extra overhead and Reddit is a very unbalanced graph, the latency is still largely determined by the unbalanced workload after performing fusion. As the neural operators consumes the major part of IO, the relative IO reduction is not significant. The absolute value of IO reduction and memory reduction are about same level. For EdgeConv, IO and memory consumption are greatly reduced, and latency is slightly improved, mainly because of saving write-in and read-out for intermediate data. As the absolute value of IO in EdgeConv is much smaller than GAT, the relative IO reduction is much more significant. For MoNet, latency, IO, and memory are all significantly saved, mainly because of the largely improved data locality and saving for broadcast. For the forward pass, the operator fusion technique improves latency by 1.68×, IO by 1.16× (up to 5.45×), and peak memory usage by 4.92× on average.

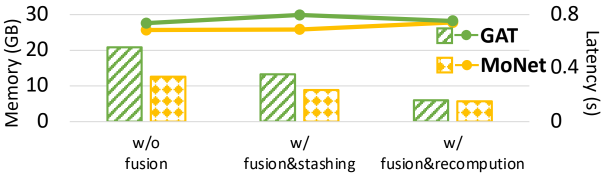


Figure 10. Benefits and overhead brought by intermediate data recomputation. "w/o fusion": disable fusion. "fusion&stashing": fuse operators but stash the needed intermediate data for backward. "fusion&recomputation": perform operator fusion as well as recomputation.

Recomputation. Figure 10 illustrates the benefits brought by intermediate recomputation on GAT and MoNet. As the Gather function in EdgeConv is max, only the indices of the maximum have to be stashed (which is $O(|\mathcal{V}|)$) and there is no need for recomputation. We use three implementations for comparison: (1) without our unified thread mapping operator fusion technique; (2) with the fusion technique but without recomputation technique, which means intermediate data have to be stashed; (3) with both our proposed fusion technique and recomputation technique. For GNN training, only fusion cannot reduce memory consumption, as even if we eliminate some intermediate data during the forward pass with operator fusion, we still need to stash them to perform back propagation. However, with our proposed recomputation technique, we can also eliminate those intermediate data during backward propagation at a small cost of computation. In GAT, recomputation saves 2.21× memory at the cost of slowing down by 7.1%. In MoNet, recomputation saves 1.55× memory and accelerates by 5.9%.

7.4 Evaluation on Different GPUs

With our proposed three techniques, we are able to perform the same training task on devices with much smaller memory capacity. We evaluate our models with the same setting as Section 7.3 on RTX 2080, all of which cannot be done without our proposed techniques due to memory capacity limits. Figure 11 show that our implementation on RTX 2080 can even achieve 1.17× end-to-end speedup over DGL on RTX 3090 with 7.73× less memory for EdgeConv.

8 Related Work

8.1 GNN Systems

NeuGraph (Ma et al., 2019) first introduces SAGA (Scatter, ApplyEdge, Gather and ApplyVertex) abstraction to describe GNNs. It is the first system that bridges the gap between graph processing systems and DNN systems. After that, GNN systems can be categorized as following types:

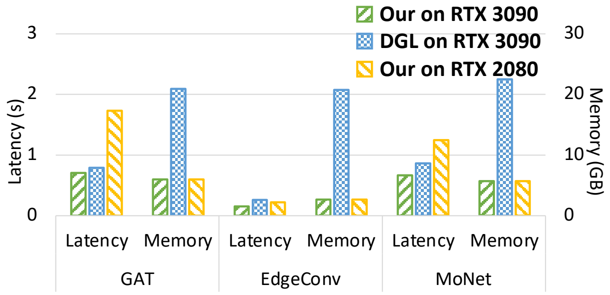


Figure 11. End-to-end performance on different GPUs. Our designs enable running large-scale GNN models with an NVIDIA RTX 2080 GPU, which require the newest NVIDIA RTX 3090 GPU, with a comparable latency.

GNN computation graph optimization includes operator reorganization, operator fusion, data flow optimization, etc., and many efforts have been made to solve the challenges in optimizing GNN computation graph: **(1) Redundant neural operator computation.** Prior work attempts to tackle the computation redundancy via manually modifying the operator combinations to a functionally-equivalent but efficient version. For example, DGL (Wang et al., 2019b) provides a GAT implementation in its GNN-module library, where the `ApplyEdge` (the linear projection) is separated into two functions applied to vertex-features ahead of propagation. However, a theory inside this practice needs to be extracted for optimizing similar scenarios, as we do in this paper. **(2) Inconsistent thread mapping.** Fusion is widely used in conventional Deep Neural Networks (DNNs) (Niu et al., 2021). FuseGNN (Chen et al., 2020) manages to fuse any two edge-centric operators, but lacks the technique to fuse a vertex-centric operator with an edge-centric one, which we address in this paper via unified thread mapping. **(3) Excessive intermediate data.** Huang *et al.,* (Huang et al., 2021) reduces intermediate data during forward but cannot handle back propagation because the intermediate data are missed. FuseGNN (Chen et al., 2020) stashes the intermediate data during forward, but lacks the recomputation technique, which still consumes great memory space.

GNN runtime optimization includes neighbor grouping, graph reordering etc, which introduces a preprocessing procedure to schedule the workload assignment and memory layout. GNNAdvisor (Wang et al., 2021) and Huang *et al.,* (Huang et al., 2021) both utilize neighbor grouping to balance the workloads among GPU threads and blocks and exploit memory locality. GNNAdvisor further use Rabbit Reordering (Arai et al., 2016) to maximize the graph modularity by clustering. By neighbor grouping and graph reordering, the runtime workload balance and memory locality are improved by introducing some preprocessing overhead. Although we mainly focus on GNN computation graph optimizations in this paper, our work can also benefit from these GNN runtime optimizations.

8.2 DNN Systems

TASO (Jia et al., 2019) proposes a novel computation graph optimizer for DNNs that can automatically generate graph substitutions. DNNFusion (Niu et al., 2021) proposes a set of fusion methodologies to work in conjunction with computation graph rewriting for DNN inference. Chen *et al.,* (Chen et al., 2016) introduces the recomputation technique to DNN training to trade computation with memory. Our proposed operator reorganization technique is more of eliminating computing redundancy, while DNN computation graph substitution is more of finding a better substitution. Our unified thread mapping operator fusion technique is also different from operator fusion in DNNs, as GNN introduces graph-related operator, which brings about the divergent thread mapping between edge-centric and vertex-centric operators. And unlike DNN recomputation, which incurs roughly 30% of additional latency (Chen et al., 2016), overhead by our proposed recomputation technique is <10% as we utilize the characteristics of GNN training and graph data.

9 Conclusion

In this paper, we present a thorough study of GNN computational graph optimization. We point out GNN systems suffer from redundant neural operator computation, inconsistent thread mapping, and excessive intermediate data. We propose a systematic framework with propagation-postponed operator reorganization, unified thread mapping for fusion, and intermediate data recomputation. We achieve up to 2.75× end-to-end speedup, 6.89× less memory IO, and 7.73× less memory consumption over state-of-the-art frameworks. We even enable running large-scale GNN models with an NVIDIA RTX 2080 GPU, which would require the newest NVIDIA RTX 3090 GPU without our technique, with a comparable latency. More specifically, we provide an optimization-friendly perspective to understand GNN computational graph, which can be extended to other hardware platforms.

Acknowledgement

This work was supported by National Natural Science Foundation of China (No. 62104128, U19B2019, 61832007); China Postdoctoral Science Foundation (No. 2019M660641); National Key R&D Program of China (No. 2017YFA02077600); Tsinghua EE Xilinx AI Research Fund; Beijing National Research Center for Information Science and Technology (BNRist); Beijing Innovation Center for Future Chips; Biren Technology. This work will be included in dgSPARSE project[1].

[1] The dgSPARSE project (`https://dgsparse.github.io/`) is an open source project for fast and efficient graph processing on GPUs.

REFERENCES

Arai, J., Shiokawa, H., Yamamuro, T., Onizuka, M., and Iwamura, S. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 22–31, 2016.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost, 2016.

Chen, Z., Yan, M., Zhu, M., Deng, L., Li, G., Li, S., and Xie, Y. fuseGNN: Accelerating Graph Convolutional Neural Network Training on GPGPU. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2020.

Fey, M. and Lenssen, J. E. Fast graph representation learning with pytorch geometric. 2019.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive Representation Learning on Large Graphs. In *International Conference on Neural Information Processing Systems (NeurIPS)*, pp. 1025–1035, 2017.

Huang, G., Dai, G., Wang, Y., and Yang, H. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

Huang, K., Zhai, J., Zheng, Z., Yi, Y., and Shen, X. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 119–132, 2021.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 47–62, 2019.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

Kipf, T. N. and Welling, M. Semi-supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907*, 2016.

Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. Neugraph: Parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference (ATC)*, pp. 443–458, 2019.

Md, V., Misra, S., Ma, G., Mohanty, R., Georganas, E., Heinecke, A., Kalamkar, D., Ahmed, N. K., and Avancha, S. Distgnn: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3480856. URL https://doi.org/10.1145/3458817.3480856.

Monti, F., Boscaini, D., Masci, J., Rodolà, E., Svoboda, J., and Bronstein, M. M. Geometric deep learning on graphs and manifolds using mixture model cnns. 2016.

Nguyen, T. and Grishman, R. Graph Convolutional Networks With Argument-Aware Pooling for Event Detection. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

Niu, W., Guan, J., Wang, Y., Agrawal, G., and Ren, B. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pp. 883–898, 2021.

Qi, S., Wang, W., Jia, B., Shen, J., and Zhu, S.-C. Learning Human-Object Interactions by Graph Parsing Neural Networks. *arXiv preprint arXiv:1808.07962*, 2018.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph Attention Networks. *arXiv preprint arXiv:1710.10903*, 2017.

Wang, H., Zhang, F., Zhang, M., Leskovec, J., Zhao, M., Li, W., and Wang, Z. Knowledge-aware Graph Neural Networks with Label Smoothness Regularization for Recommender Systems. *arXiv preprint arXiv:1905.04413*, 2019a.

Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., et al. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. 2019b.

Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., and Solomon, J. M. Dynamic graph cnn for learning on point clouds. 2019c.

Wang, Y., Feng, B., Li, G., Li, S., Deng, L., Xie, Y., and Ding, Y. Gnnadvisor: An adaptive and efficient runtime system for GNN acceleration on gpus. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 515–531, 2021.

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., and Xiao, J. 3d shapenets: A deep representation for volumetric shapes. 2015.

Yan, M., Deng, L., Hu, X., Liang, L., Feng, Y., Ye, X., Zhang, Z., Fan, D., and Xie, Y. Hygcn: A gcn accelerator with hybrid architecture. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 15–29, 2020.

Yan, S., Xiong, Y., and Lin, D. Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition. *arXiv preprint arXiv:1801.07455*, 2018.

Yao, L., Mao, C., and Luo, Y. Graph Convolutional Networks for Text Classification. *arXiv preprint arXiv:1809.05679*, 2018.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph Convolutional Neural Networks for Web-scale Recommender Systems. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pp. 974–983, 2018.

APPENDIX

A GNN Operators

This section formally describes our taxonomy of GNN operators, briefly introduced in Section 2.1 as 4 basic operators: `Scatter`, `Gather`, `ApplyEdge`, `ApplyVertex`, and 2 high-level operators: `ReduceScatter` and `Aggregate`. We further illustrate how to construct popular GNN models from this set of operators.

A.1 Operator Definition

Let a graph be $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents the set of vertices, and $\mathcal{E}$ represents the set of edges. The elements in $\mathcal{E}$ is tuples of (u, e, v), where $u, v \in \mathcal{V}$ and e is a unique id. The tuple (u, e, v) indicates there is an edge indexed by e pointing from u to v. [2] We define four basic operators as follows:

`Scatter`: $m_e = \phi(h_u, h_v), (u, e, v) \in \mathcal{E}$. For every edge, perform a binary operation (function $\phi(\cdot, \cdot)$) on the features attached to the two vertices that the edge connects to.

`Gather`: $h_v = \psi(\{m_e : (u, e, v) \in \mathcal{E}\})$. For every vertex, perform a reduction operation to the features attached to all edges that connects to it.

`ApplyEdge`: $m_e^{new} = f_e(m_e[, m_e', \cdots]), (u, e, v) \in \mathcal{E}$. For every edge, perform the same function f_e that transforms its current feature (and any history features). This operator is graph-irrelevant , meaning that its outcome does not change if the graph structure (connections) changes.

`ApplyVertex`: $h_v^{new} = f_v(h_v[, h_v', \cdots]), v \in \mathcal{V}$. For every vertex, perform the same function f_v that transforms its current feature (and any history features). This operator is also graph-irrelevant like `ApplyVertex`.

Through composing the above four operators, we also propose two high-level operators that are widely seen in GNN models:

`Aggregate`:
$h_v^{new} = \psi(\{f_e(\phi(h_u, h_v), m_e)\}), (u, e, v) \in \mathcal{E}$. It is a sequence of three basic operators: `Scatter` to generate edge features, `ApplyEdge` to transform the edge feature or combine it with any history features, and finally `Gather` to reduce edge features and generate new vertex features. A typical example is the neighborhood feature-reduction in vanilla GCN, where each vertex takes the sum of all its neighbor-vertices' features, essentially $h_v^{new} = sum(\{w_e \cdot h_u : (u, e, v) \in \mathcal{E}\})$. This step can be expressed by `Aggregate` by binding ϕ as copying source-vertex's feature, f_e as multiplying the edge weight w_e, and ψ as summation.

`ReduceScatter`:
$m_e^{new} = f_e(\phi(\psi(\{m_e\}), h_u), m_e'), (u, e, v) \in \mathcal{E}$. It is a sequence of three basic operators: `Gather` to reduce edge features into vertex features based on the vertex's adjacent edge group, and `Scatter` to broadcast the reduction results to all edges, and finally `ApplyEdge` to combine the broadcast values and any history features into new edge features. This operation can be used when the edge features are normalized within a neighborhood set, as happens in the edge-softmax. Edge-softmax performs $m_e^{new} = softmax(\{m_e' : (u \in \mathcal{N}(v), e', v\})[e]$, where

$$softmax(x_1, \cdots, x_n)[i] = \frac{e^{(x_i - \max_k(x_k))}}{\sum_{j=1}^{n} e^{(x_i - \max_k(x_k))}}$$

. This step can be expressed by the following code snippet:

`RS1`: $\psi \leftarrow max$, $\phi \leftarrow$ copy, $f_e \leftarrow$ substraction,
`RS2`: $\psi \leftarrow sum$, $\phi \leftarrow$ copy, $f_e \leftarrow$ division.

A.2 Construct GNN Models

GCN

Vanilla GCN is defined as:

$$h_v^{(l+1)} = \sigma \left(b^{(l)} + \sum_{u \in \mathcal{N}(v)} e_{uv} h_u^{(l)} W^{(l)} \right)$$

where σ is an activation function, b is a bias, and W is weight to learn. With four basic operators, we first perform `ApplyVertex`, then copy source vertex's feature to edges (`Scatter`) and multiply the edge weights (`ApplyEdge`) to obtain $e_{uv} h_u^{(l)} W^{(l)}$, followed by a gather with summation (`Gather`) and an activation (`ApplyVertex`), as shown in figure 12(a). Figure 12(b) shows how to describe the same procedure with an high-level opeartor `Aggregate`.

GAT

GAT is defined as:

$$h_v^{(l+1)} = \sum_{u \in \mathcal{N}(v)} e_{uv} W^{(l)} h_u^{(l)}$$

$$e_{ij}^{l} = \text{edge-softmax}\left(\text{LeakyReLU}\left(\vec{a}^T \left[W h_i \| W h_j\right]\right)\right)$$

where W and a are learnable parameters. Figure 12(c) shows one way to compute this. Assume the input node feature vectors are concatenated into a feature matrix $\boldsymbol{H}^{(l)} \in \mathbb{R}^{n \times f^{(l)}}$, and operator reorganization technique is already applied. We first perform a dense matrix matrix multiplication to transform this feature matrix into $\widetilde{\boldsymbol{H}^{(l)}} = \boldsymbol{H}^{(l)} \times \boldsymbol{W}^{(l)} \in \mathbb{R}^{n \times f^{(l+1)}}$ with torch.nn.linear. We decompose the weight vector $\boldsymbol{a} \in \mathbb{R}^{2f^{(l+1)}}$ into $[\boldsymbol{a}_l || \boldsymbol{a}_r]$

[2] Here we assume directed edges, but can generalize the theory to undirected edges by seeing each edge $u \leftrightarrow v$ as two directed ones $u \rightarrow v$ and $v \rightarrow u$.

and compute attention scores $\boldsymbol{A}_l = \widetilde{\boldsymbol{H}^{(l)}} \times \boldsymbol{a}_l \in \mathbb{R}^{n\times 1}$ and $\boldsymbol{A}_r = \widetilde{\boldsymbol{H}^{(l)}} \times \boldsymbol{a}_r \in \mathbb{R}^{n\times 1}$.

After that, $\boldsymbol{M}_0 \in \mathbb{R}^n$ are generated by

$$\boldsymbol{M}_0 = \text{u\_add\_v}(\boldsymbol{A}_l, \boldsymbol{A}_r)$$

An `ApplyEdge`operator is then applied to generate

$$\boldsymbol{M_1} = \text{LeakyReLU}(\boldsymbol{M}_0) \in \mathbb{R}^n$$

followed by a `ReduceScatter` operator to generate

$$\boldsymbol{M_2} = \text{edge\_softmax}(\boldsymbol{M}_1) \in \mathbb{R}^n$$

An `Aggregate`operator is performed to generate

$$\boldsymbol{H}^{(l+1)} = \text{reduce\_sum}(\boldsymbol{M_2}, \widetilde{\boldsymbol{H}^{(l)}}) \in \mathbb{R}^{n\times f^{(l+1)}}$$

In our implementation, we fuse the computation of $\boldsymbol{M}_0, \boldsymbol{M}_1, \boldsymbol{M}_2, \boldsymbol{H}^{(l+1)}$ into one operator, as shown in figure 12(d).

EdgeConv

Figure 12(e) shows one way to compute EdgeConv. The mathematical definition of one EdgeConv layer is

$$h_v^{(l+1)} = \max_{u\in\mathcal{N}(v)} \left(\Theta \cdot \left(h_u^{(l)} - h_v^{(l)}\right) + \Phi \cdot h_v^{(l)}\right)$$

where $\mathcal{N}(v)$ is the neighbor of v.Θ and Φ are linear layers. In SOTA gnn framework DGL, one edgeconv layer is computed as shown in figure 12(e). Define the input node feature matrices as $\boldsymbol{H}^{(l)} \in \mathbb{R}^{n\times f}$. The $(h_u^{(l)} - h_v^{(l)})$ is computed by

$$\boldsymbol{E}^{(l)} = \text{u\_sub\_v}\left(\boldsymbol{H}^{(l)}\right) \in \mathbb{R}^{e\times f^{(l)}}$$

followed by one linear `ApplyEdge`

$$\boldsymbol{E}_{\Theta}^{(l)} = \Theta \cdot \boldsymbol{E}^{(l)} \in \mathbb{R}^{e\times f^{(l+1)}}$$

An linear `ApplyVertex` is performed to compute $\Phi \cdot h_v^{(l)}$:

$$\boldsymbol{N}_{\Phi}^{(l)} = \Phi \cdot \boldsymbol{H}^{(l)} \in \mathbb{R}^{n\times f^{(l+1)}}$$

followed by

$$\boldsymbol{E}_{\Theta+\Phi}^{(l)} = \text{e\_add\_v}\left(\boldsymbol{E}_{\Theta}^{(l)}, \boldsymbol{N}_{\Phi}^{(l)}\right) \in \mathbb{R}^{e\times f^{(l+1)}}$$

In the end, a reduce function is called to update the node features

$$\boldsymbol{H}^{(l+1)} = \text{reduce\_max}\left(\boldsymbol{E}_{\Theta+\Phi}^{(l)}\right) \in \mathbb{R}^{n\times f^{(l+1)}}$$

GMMConv

GMMConv is defined as:

$$m_{uv} = f(x_u, x_v), x_u \in \mathcal{N}(v)$$

$$w_k(m) = exp(-\frac{1}{2}(m-\mu_k)^T \Sigma_k^{-1}(m-\mu_k))$$

f here is a linear projection, Σ_k is a covariance matrix of the gaussian kernel, μ_k is the mean of the gaussian kernel. By setting covariance matrix and mean as parameters with gradient, GMMConv could learn weight w_k in training process (`ApplyEdge`).

$$h_v^{(l+1)} = \frac{1}{K}\sum_{u\in\mathcal{N}(v)}\sum_{k}^{K} w_k(m_{uv}) h_{u_k}^{(l)}$$

To get node feature, GMMConv multiplies node embedding with gaussian weight, followed by gathering the sum of multi-kernels of embeddings (`Gather`).

B Back-propagation of GNN Operators

In this subsection, we derive the backward pass of the four GNN operators, and show that they can still be constructed by the four basic operators. Here $\circ$ represents composition of operators where the latter operator gets applied first.

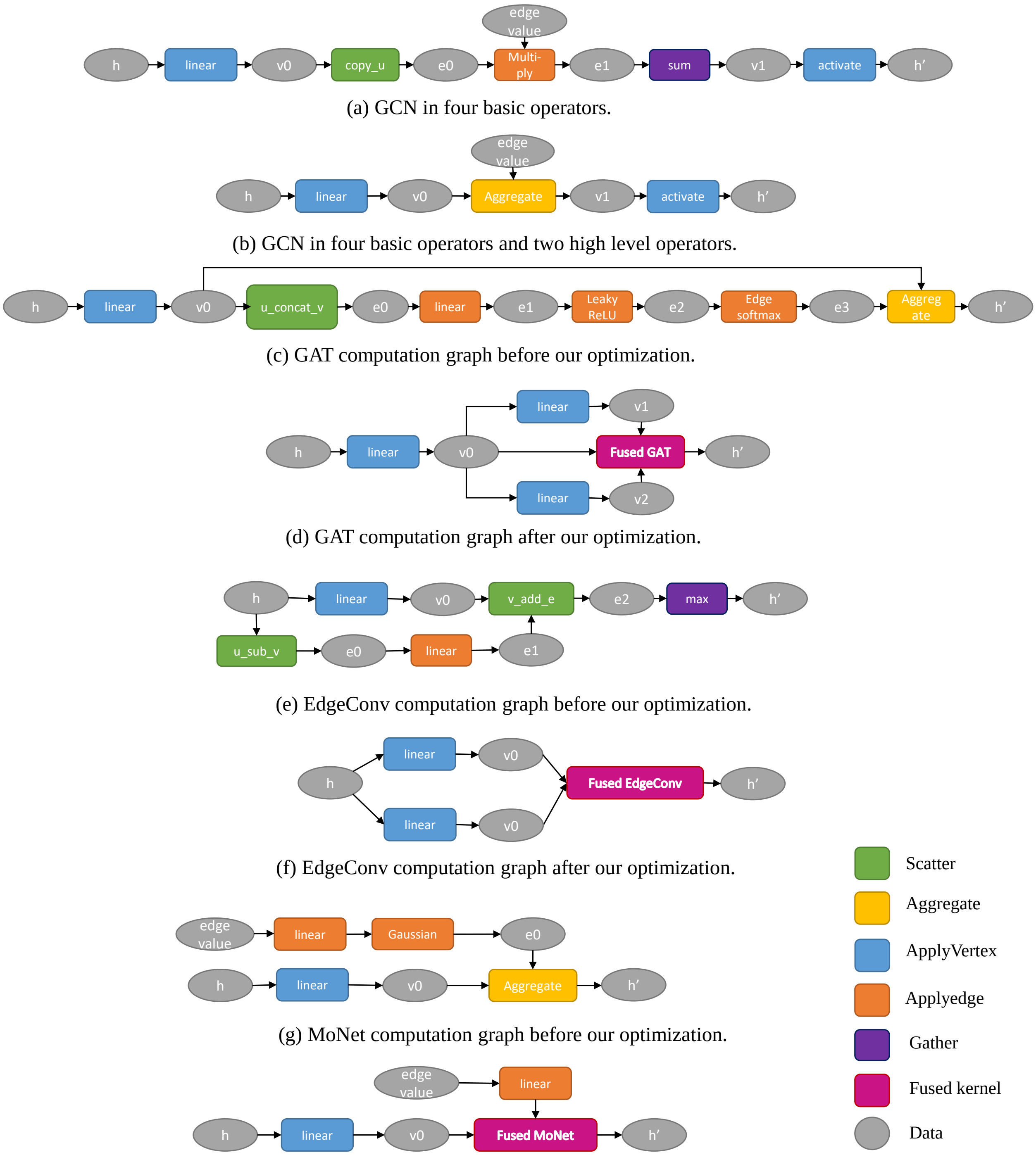
`Gather`: The backward pass of `Gather` is a `Scatter` followed by an `ApplyEdge`.

$$\begin{aligned}\textbf{Forward: } h_v =& \psi(\{m_e : (u,e,v) \in \mathcal{E}\}),\\ \textbf{Backward: } \text{grad}m_e =& \text{grad}h_v \times \frac{\partial \psi}{\partial m_e}\\ =& \texttt{ApplyEdge}_{f_e \leftarrow (\times \text{grad}m_e)}\\ & \circ \texttt{Scatter}_{\phi\leftarrow \text{copy\_v}}\end{aligned}$$

`Scatter`: The backward pass of `Scatter` is a `Gather` followed by an `ApplyVertex`.

$$\begin{aligned}\textbf{Forward: } m_e =& \phi(h_u, h_v), (u,e,v) \in \mathcal{E},\\ \textbf{Backward: } \text{grad}h_v =& \sum_{(v,e,u)\in\mathcal{E}} \text{grad}m_e \times \frac{\partial\phi}{\partial h_v}\\ &+ \sum_{(v,e',u')\in\mathcal{E}} \text{grad}m_{e'} \times \frac{\partial\phi}{\partial h_u}\\ =& \texttt{ApplyEdge}_{f_e\leftarrow(\times[\frac{\partial\phi}{\partial h_v}, \frac{\partial\phi}{\partial h_u}]^T)}\\ & \circ \texttt{Gather}_{\phi\leftarrow[\sum \text{grad}m_e, \sum \text{grad}m_{e'}]}\end{aligned}$$

`Apply-`: The backward pass of graph-irrelevant `Apply-` is also graph-irrelevant , and can be derived in the same way



(a) GCN in four basic operators.

(b) GCN in four basic operators and two high level operators.

(c) GAT computation graph before our optimization.

(d) GAT computation graph after our optimization.

(e) EdgeConv computation graph before our optimization.

(f) EdgeConv computation graph after our optimization.

(g) MoNet computation graph before our optimization.

(h) MoNet computation graph after our optimization.

Figure 12. Construct GNN models with GNN operators.

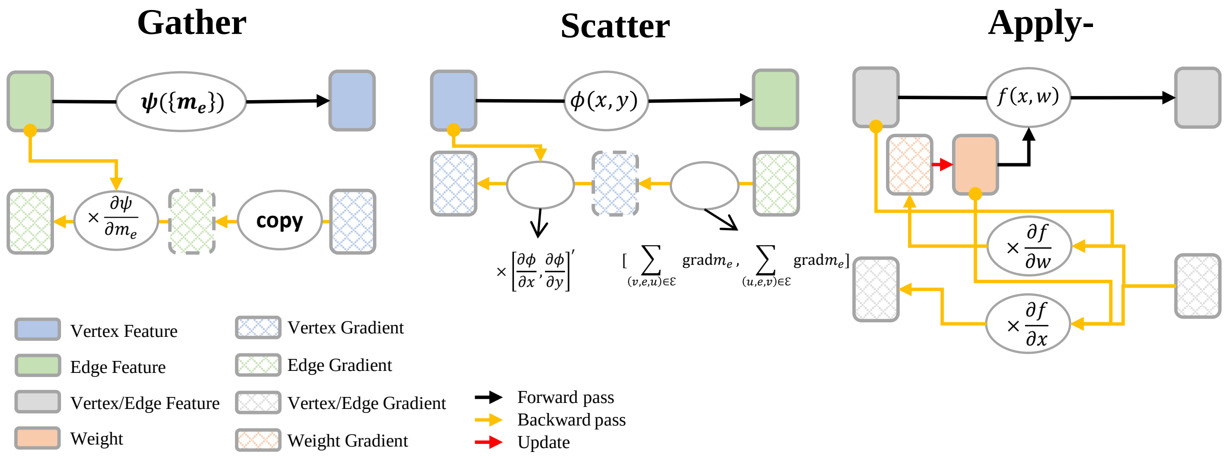


Figure 13. Back-propagation dataflow of GNN operators.

as operators in neural networks.

$$\textbf{Forward: } y = f(x, w)$$

$$\textbf{Backward: } \text{grad}w = \text{grad}y \times \frac{\partial f}{\partial w}$$

$$\text{grad}x = \text{grad}y \times \frac{\partial f}{\partial x}$$

Hence the backward of `Apply-` is two `Apply-`, one calculating the gradient of input and one for the gradient of weight parameters.

Figure 13 visualizes the forward-backward dataflow of each GNN operator.

A Artifact Appendix

A.1 Abstract

Our work propose a systematic methodology to optimize the computational graph for GNNs on GPUs. Our work consists of two parts. The first part is the GPU kernels which are responsible for the major computation of GNN models and are implemented with our proposed operator reorganization, operator fusion, and re-computation technique. The second part is the python code that wraps the kernels to provide a PyTorch-based front-end, and uses them as building blocks to build up different GNN models. Our work improves the performance of GNN computing with careful designs and surgeries in GNN computational graph to reduce computation, IO, and memory consumption, while preserves functional equivalence. Moreover, we show how to analyze and optimize GNN computational graph with three examples and our proposed techniques can also be applied to many other GNN models.

A.2 Artifact check-list (meta-information)

- **Program:** `https://github.com/mlsysAE2022/ae_mlsys_gnn`.
- **Hardware:**
  - Intel CPU x86_64 with host memory $\geq$ 64GB. Tested on Intel Xeon Silver 4210 (10-core 20-thread) CPU with 512 GB host memory.
  - NVIDIA GPU with device memory $\geq$ 24GB. Tested on RTX3090 and RTX2080. We mainly evaluate our design on RTX3090 and the execution time may be different across different devices but the peak memory usage remains same.
- **Compilation:** Ubuntu 18.04+, CUDA 11.0+.

A.3 Description

A.3.1 How delivered

The source code and scripts are available at *https://github.com/mlsysAE2022/ae_mlsys_gnn*.

A.3.2 Hardware dependencies

Our implementation works on Intel x86 CPUs and Nvidia GPUs.

A.3.3 Software dependencies

- CUDA 11.0+
- PyTorch 1.8.0+
- DGL 0.7.0+
- Ninja 1.10+
- GPUtil 1.4+

A.4 Installation

To build our software, you need to install Ninja and PyTorch as shown in dependencies. We use the just-in-time compilation of the pytorch cpp-extension work flow.

A.5 Experiment workflow

- Go to `script/` directory.
- **IO result:** `./io.sh` to run all the IO results. Generate `figure7_io.csv`, `figure8_io.csv`, and `figure9_io.csv`.
- **Figure 7 result:** `./figure7.sh` to run end-to-end experiments on three GNN models. Generate `figure7.csv`.
- **Figure 8 result:** `./figure8.sh` to run ablation study for operator reorganization. Generate `figure8.csv`.
- **Figure 9 result:** `./figure8.sh` to run ablation study for operator fusion. Generate `figure9.csv`.
- **Figure 10 result:** `./figure10.sh` to run ablation study for intermediate variable re-computation and generate `figure10.csv`.
- **Figure 11 result:** run `figure11_3090.sh` on RTX3090 and `figure11_2080.sh` on RTX2080. Generate `figure11_3090.csv` and `figure11_2080.csv`
- **fuseGNN result:** run `training_main.py` in the `gcnLib` submodule. Use the better result of `gas` and `gar` in `--mode`.

A.6 Evaluation and expected result

Once you have run the experiment workflow, you can see the `.csv` result under the `script/` directory. The latency and memory results are stored in `figureX.csv`. The IO results can be seen in the corresponding `figureX_io.csv`.

Example output is given in the folder `example_data`. We ran an extra experiment on Tesla V100 (16GB) to show how to run the experiments on smaller devices, although some of the results are missing because of CUDA out of memory, such as MoNet from DGL on Reddit. Although our implementations consume less than 8 GB device memory, our baselines such as DGL consume device memory as large as 23.1GB (MoNet from DGL on Reddit), so a minimum of 24 GB device memory is needed to run the whole experiment. The memory consumption will remain the same across different devices, and although the absolute value of latency may vary among devices, the speedup ratio between our implementations and baselines remains similar to the results in the paper.

A.7 Methodology

Submission, reviewing and badging methodology:

- `http://cTuning.org/ae/submission-20190109.html`
- `http://cTuning.org/ae/reviewing-20190109.html`
- `https://www.acm.org/publications/policies/artifact-review-badging`