DF-GNN: Dynamic Fusion Framework for Attention Graph Neural Networks on GPUs

Jiahui Liu[1], Zhenkun Cai[2], Zhiyong Chen[1], Minjie Wang[2]
[1]Shanghai Jiao Tong University, [2]Amazon Web Services
[1]{the-waves, zhiyongchen}@sjtu.edu.cn, [2]{zkcai, minjiw}@amazon.com

Abstract

Attention Graph Neural Networks (AT-GNNs), such as GAT and Graph Transformer, have demonstrated superior performance compared to other GNNs. However, existing GNN systems struggle to efficiently train AT-GNNs on GPUs due to their intricate computation patterns. The execution of AT-GNN operations without kernel fusion results in heavy data movement and significant kernel launch overhead, while fixed thread scheduling in existing GNN kernel fusion strategies leads to sub-optimal performance, redundant computation and unbalanced workload. To address these challenges, we propose a dynamic kernel fusion framework, DF-GNN, for the AT-GNN family. DF-GNN introduces a dynamic bi-level thread scheduling strategy, enabling flexible adjustments to thread scheduling while retaining the benefits of shared memory within the fused kernel. DF-GNN tailors specific thread scheduling for operations in AT-GNNs and considers the performance bottleneck shift caused by the presence of super nodes. Additionally, DF-GNN is integrated with the PyTorch framework for high programmability. Evaluations across diverse GNN models and multiple datasets reveal that DF-GNN surpasses existing GNN kernel optimization works like cuGraph and dgNN, with speedups up to $7.0\times$ over the state-of-the-art non-fusion DGL sparse library. Moreover, it achieves an average speedup of $2.16\times$ in end-to-end training compared to the popular GNN computing framework DGL. We open-source DF-GNN at `https://github.com/paoxiaode/DF-GNN`.

1 Introduction

Graph Neural Networks (GNNs) represent a category of deep learning techniques applied on graph-structured data, playing a vital role in diverse applications. Inspired by the success of the Transformer architecture [1] in LLMs, Attention GNNs (AT-GNNs) have emerged as an important family of Graph Neural Networks. Various AT-GNNs models have been developed, including GAT family [2, 3], Graph Transformer (GT) [4–6], and AGNN [7]. These models have demonstrated enhanced performance for node classification [8, 9], link prediction [10, 11], and graph prediction tasks [12, 13].

AT-GNNs are characterized by their complex computation pattern, typically encompassing three primary steps in each layer: firstly, calculating edge-wise attention scores using node and edge features; secondly, normalizing these scores across each node's local neighborhood; and finally, gathering and aggregating neighbor features, weighted by these normalized attention scores. To facilitate these operations, established GNN systems such as DGL [14] and PyG [15] implement the computation using the message passing paradigm [16], by breaking down GNNs into a series of independent kernels to support various model variants. Nonetheless, executing AT-GNN kernels sequentially on the GPU incurs considerable global memory access and CPU overhead from kernel launches, resulting in inefficient training of AT-GNN models, especially graph transformers, which may require several days to complete. To tackle these issues, several kernel fusion techniques have been proposed for general GNNs. Seastar [17] introduced a CUDA template for fusing multiple sparse operations, thereby omitting the materialization of edge data in GPU global memory. dgNN [18] suggested a unified thread mapping scheme for both vertex and edge-centric operations to facilitate fusion and IO reduction. TLPGNN [19] developed a two-level parallelism paradigm to circumvent atomic operations and enable coalesced memory access. However, applying these techniques to AT-GNNs encounters two significant challenges:

- **Varying compute patterns.** The computational process of AT-GNNs is marked by multiple operations of vertex-centric and edge-centric computations. This frequent change in computational patterns poses a significant challenge for existing kernel fusion strategies, which predominantly rely on uniform thread scheduling across all fused operations. Such a rigid scheduling framework struggles to adapt to the varying computational demands, often leading to inefficiencies. Furthermore, the inflexibility of fixed thread scheduling can result in unnecessary and redundant computations when the feature dimension diminishes in operations such as Softmax.
- **Super node presence.** The computational performance of AT-GNNs is also influenced by the input graph. In real-world graphs, super nodes often exist with a significantly higher number of neighbors compared to other nodes in the same graph. The existence of super nodes leads to a performance bottleneck shift and can quickly deplete the available shared memory, limiting the performance gain from kernel fusion.

To address these challenges, we introduce **DF-GNN**, a framework designed for efficient GPU kernel fusion of AT-GNNs. The key to DF-GNN is *allowing each operation in AT-GNNs to dynamically adopt the optimal thread scheduling strategy, while also enjoying the benefits of kernel fusion, including shared memory usage and reduced kernel launch overhead.* **At the GPU kernel scheduling level**, DF-GNN incorporates a Dynamic Bi-level Thread Scheduling (BTS) design. Dynamic BTS allows each operation to explore its optimal thread scheduling strategy between and within thread blocks. For intra-block scheduling, DF-GNN addresses the specific performance bottlenecks of the three operations in AT-GNNs and devises tailored scheduling strategies. In inter-block scheduling, DF-GNN accounts for performance bottlenecks introduced by super nodes and formulates two general kernel fusion methods, selecting the appropriate method based on the input graph's characteristics. **At the model level**, DF-GNN extends optimization to the backward pass, utilizing the same fusion strategy to optimize the backward computation, enabling acceleration in both forward and backward computations during AT-GNNs training. **At the framework level**, DF-GNN is integrated with the PyTorch [20] framework and provides a series of APIs for easy user access. Users can easily invoke the DF-GNN APIs within their PyTorch models to facilitate efficient model training and inference.

Through comprehensive evaluations on multiple AT-GNNs models and diverse datasets, DF-GNN consistently outperforms existing GNN optimization works such as cuGraph and dgNN, with speedups over the DGL sparse baseline ranging from $1.92\times$ to $7.00\times$. Additionally, DF-GNN achieves an average end-to-end training speedup of $2.16\times$ compared to the DGL framework. These results underscore the effectiveness of DF-GNN in enhancing both kernel-level and end-to-end performance for AT-GNNs training.

2 Background

2.1 Attention GNN Vectorized Formulation

The message passing paradigm presents the computation of GNN from the view of an individual node and its local neighbors. However, this node-centric view can't efficiently capture operations on a graph which consists of a batch of nodes and edges. Thus, a vectorized formulation of AT-GNNs is desired. DGL [14] proposed to generalize classical operations for sparse linear algebra as the vectorized abstraction of the message passing paradigm. Specifically, one generalized Sampled Dense-Dense Matrix Multiplication (SDDMM) is responsible for generating the message on edges, while a generalized Sparse-dense Matrix Multiplication (SpMM) is in charge of message aggregation. Applying the idea to AT-GNNs gives the following formulation:

$$\begin{aligned}
\mathbf{Q}^{N\times d}, \mathbf{K}^{N\times d}, \mathbf{V}^{N\times d} &= f_Q(\mathbf{X}), f_K(\mathbf{X}), f_V(\mathbf{X}) \\
\text{Message: } \mathbf{S}^{N\times N} &= \text{SDDMM}(\mathbf{Q}, \mathbf{K}, \mathbf{A}) \\
&= (\mathbf{Q} *_{\otimes\oplus} \mathbf{K}^\top) \odot \mathbf{A}, \\
\text{Normalize: } \mathbf{P}^{N\times N} &= \text{Softmax}(\mathbf{S}), \\
\text{Aggregate: } \mathbf{O}^{N\times d} &= \text{SpMM}(\mathbf{P}, \mathbf{V}) \\
&= \mathbf{P} *_{\otimes\oplus} \mathbf{V}.
\end{aligned} \tag{1}$$

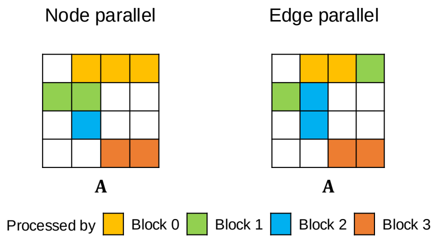

Figure 1: Workload partition in node and edge parallel.

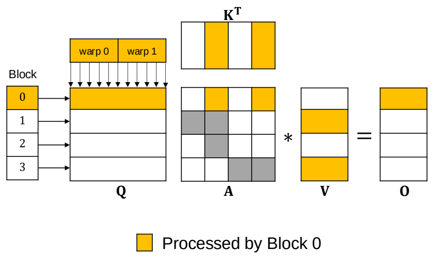

Figure 2: The fused GT convolution kernel under feature parallel fusion strategy.

Here, N is the number of nodes in graph G, d is the node feature dimension, $\mathbf{A}$ represents the adjacency matrix, and $\mathbf{X}$ represents the node feature matrix. The message step first transforms node features $\mathbf{X}$ into $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ matrices and then invokes an SDDMM operation to compute attention weights $\mathbf{S}$. $*_{\otimes\oplus}$ is a generalized matrix multiplication equipped with a scalar addition and a scalar multiplication, while $\odot$ is the Hadamard product (or element-wise multiplication). The flexibility in the choice of $\oplus$ and $\otimes$ leads to different graph convolution variants in computing attention and aggregating messages.

2.2 Existing Optimizations and Limitations

Parallel strategies for GNN operations. In GNN-related workload, inter-block scheduling have two typical strategies: ***Node parallel*** [14], which allocates the workload of a node and its neighbors to one parallel unit, naturally fits the computational pattern of SpMM which aggregates values according to nodes. ***Edge parallel*** [14], in contrast, assigns different parallel units to different edges, achieving ideal workload balance for SDDMM. Figure 1 depicts the difference between node and edge parallel. In addition to these two inter-block methods, ***Feature parallel*** [19] serves as an intra-block scheduling method. Within the thread block, the number of threads corresponds to the feature dimension, with each dimension mapped to a thread. Based on these parallel strategies, various approaches have been proposed to accelerate individual operation kernels such as SpMM or SDDMM. However, without kernel fusion, AT-GNNs involving multiple operations will invoke a series of fragmented kernels. This fragmentation can lead to performance loss, as these kernels fail to efficiently utilize shared memory to minimize data movement and often incur significant CPU overhead from frequent kernel launches.

Kernel fusion. Kernel fusion is a widely used optimization technique that effectively addresses the challenges posed by fragmented kernels, and a series of works represented by FlashAttention [21] have achieved great success in LLMs field. In GNN, research on kernel fusion can be categorized into two primary approaches: one is handcrafted fused kernels [18, 19, 22], manually writing optimized CUDA kernels. These optimizations typically target specific, fixed computation patterns found in certain GNN convolutions. The other is compiler-based frameworks [17, 23, 24], which automatically generate fused kernels based on user-defined functions. However, whether through manual coding or automated generation, existing approaches in GNN field generally employ a fixed parallel strategy for all operations within the fused kernel. This includes inter-block scheduling utilizing node parallel and intra-block scheduling employing feature parallel, as Figure 2 shows. In the following text, we refer to this fixed fusion approach as ***feature parallel fusion strategy***.

The feature parallel fusion strategy encounters two primary hurdles when employed in the context of AT-GNNs, stemming from the irregularity in computation and input data. The first challenge revolves around computation irregularity, where the three operations within AT-GNNs have different optimal parallel strategies. Specifically, SDDMM is suited for the edge parallel strategy, while Softmax and SpMM are better suited for the node parallel strategy. The feature parallel fusion strategy fails to account for these differences. The second challenge arises from the irregularity of input graph. Real-world graphs have diverse characteristics, and some contain super nodes. These variations can shift the performance bottleneck of graph convolution, preventing kernel fusion from achieving optimal performance in certain cases. However, the feature parallel fusion strategy does not consider the data characteristics of the input graphs.

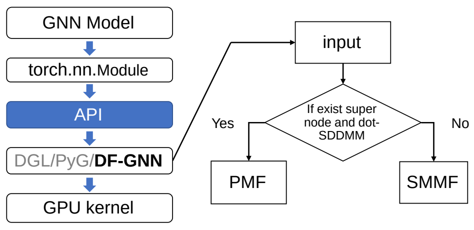

Figure 3: DF-GNN framework.

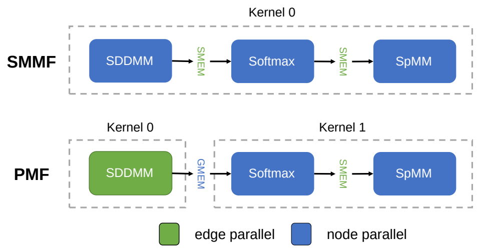

Figure 4: The SMMF and PMF in DF-GNN.

3 Accelerate AT-GNNs with DF-GNN

DF-GNN framework is developed to be fully compatible with the widely-used deep learning framework PyTorch, similar to existing GNN frameworks like DGL and PyG, as illustrated in Figure 3. Users can easily invoke the optimized fused GPU kernel code through the DF-GNN graph convolution API within the *torch.nn.Module*, thereby enhancing the efficiency of GNN models.

The framework supports two general kernel fusion methods for AT-GNNs: Shared Memory Maximization Fusion ***(SMMF)*** and Parallelism Maximization Fusion ***(PMF)***, which maximizes the utilization of shared memory and parallelism, respectively, as depicted in Figure 4. For the SMMF, three operations are fused into a single kernel, sharing the same node parallel inter-block scheduling, with intermediate variables transferred through shared memory. Conversely, the PMF employs edge parallel SDDMM while utilizing node parallel for Softmax and SpMM operations. In this case, Softmax and SpMM are fused, with SDDMM passing its results via global memory. Additionally, DF-GNN enables dynamic selection between SMMF and PMF at runtime with each API call, with the rationale for this choice discussed in detail in Section 4.2.

4 DF-GNN Design

Overview. DF-GNN employs a dynamic bi-level thread scheduling method, enabling operations to adopt their optimal thread scheduling strategies and leverage shared memory for efficient inter-operation communication within a fused kernel. Based on this method, DF-GNN designs tailored thread scheduling strategies for three operations in AT-GNNs, which include warp-balanced SDDMM, redundancy-free Softmax, and vectorized SpMM. Moreover, DF-GNN also fuses operations in backward computation to accelerate end-to-end AT-GNNs training.

4.1 Dynamic Bi-level Thread Scheduling

To address the challenges of AT-GNNs kernel fusion, DF-GNN designs ***Dynamic Bi-level Thread Scheduling***, a fine-grained, dynamic kernel thread scheduling strategy to replace the previous coarse-grained, fixed scheduling schemes, allowing the three operations in AT-GNNs to employ different parallel strategies. For the inter-block scheduling, operations can select between edge parallel and node parallel strategies, based on factors such as the need to avoid atomic operators and the patterns of graph convolution computation. For the intra-block scheduling, operations may apply different strategies based on their own computational characteristics. If an operation uses a different intra-block scheduling than the preceding operation, DF-GNN enforces thread synchronization within the thread blocks and reschedules the threads according to the new scheduling. Additionally, DF-GNN fuses adjacent operations that share the same inter-block scheduling into a single kernel, leveraging shared memory for efficient operation communication within the fused kernel.

The dynamic thread scheduling design offers two primary advantages over the fixed feature parallel fusion strategy in FusedMM [22], TLPGNN [19], and dgNN [18]. Firstly, the bi-level flexible thread scheduling between and within thread blocks expands the optimization space for kernel performance. Each operation can choose optimal scheduling strategies at both levels based on the compute pattern, enhancing the performance gain of each operation within the DF-GNN kernels. Secondly, the flexible combination of different scheduling strategies empowers DF-GNN to consider input data factors when deciding the thread scheduling strategy. For specific input graph characteristics, DF-GNN adopts different kernel scheduling strategies to address their unique bottlenecks, improving the generality of DF-GNN's fusion strategy.

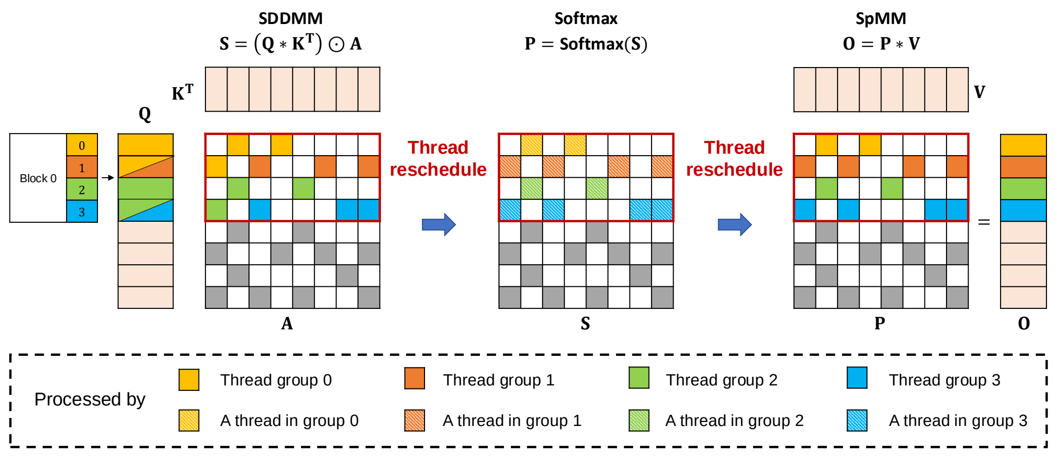

Figure 5: Dynamic Bi-level Thread Scheduling in GT convolution.

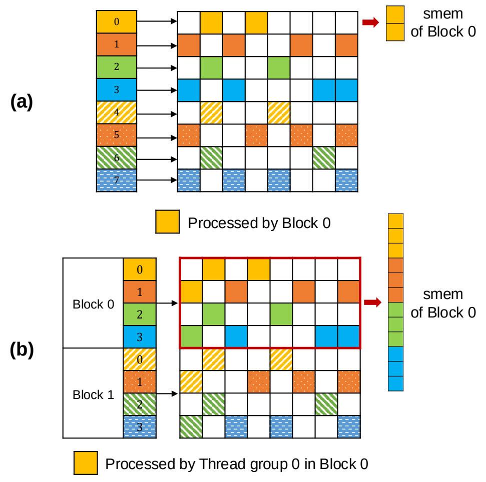

Figure 6: (a) Feature parallel SDDMM. (b) Warp-balanced SDDMM in DF-GNN.

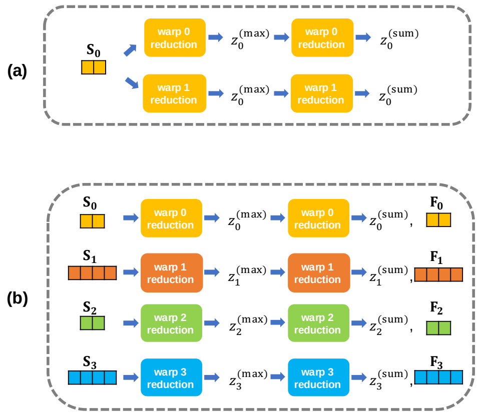

Figure 7: (a) Feature parallel Softmax. (b) Redundancy-free Softmax in DF-GNN.

Figure 5 illustrates the application of dynamic BTS within the graph transformer model. In this instance, the SMMF approach is utilized, where three operations are fused into a single kernel that shares the same node parallel inter-block scheduling. Each thread block comprises four thread groups (warps) and is responsible for managing the computations of four rows. Meanwhile, the intra-block scheduling is dynamically adjusted for different operations to optimize kernel performance. In SDDMM, the edges across these four rows are evenly distributed among the four thread groups for processing. Following this, thread groups adjust their scheduling to node parallel for both Softmax and SpMM since both of them require row-wise aggregations. Notably, since Softmax does not involve feature dimension, threads within a thread group are flattened to process a single row. While in SpMM, a thread group collectively processes an edge, iterating over the corresponding row.

4.2 Tailored Thread Scheduling

SDDMM. In the existing inter-block thread scheduling strategies, edge parallel evenly distributes edges across all thread blocks, making it most suitable for SDDMM where each edge result is independent. However, edge parallel SDDMM is difficult to fuse with the subsequent Softmax and SpMM operations, which are better suited for node parallel. Consider this, node parallel SDDMM is used in existing feature parallel fusion strategy, which processes edges in one row by a thread block, as shown in Figure 6a. Nevertheless, node parallel SDDMM suffers from workload imbalance between thread blocks due to the varying numbers of neighbors.

Inspired by the workload balancing works in [25, 26], DF-GNN addresses the issue of SDDMM workload imbalance and facilitates kernel fusion by designing the ***warp-balanced SDDMM***, adopting a balanced strategy with intra-block scheduling using edge parallel and inter-block scheduling using node parallel. Figure 6b illustrates the process of warp-balanced SDDMM. In this approach, threads within a thread block are grouped, and these thread groups evenly distribute edges and loop over them to compute the SDDMM result. The computed attention weights of a thread block are then stored in shared memory, ready for subsequent utilization in the Softmax and SpMM operations. Compared to the feature parallel strategy, the tailored SDDMM thread scheduling in DF-GNN ensures warp-level workload balance. Warp-balanced SDDMM uniformly distributes computations for multiple adjacent nodes and enables kernel fusion with subsequent operations, resulting in a significant reduction in overall imbalance and kernel execution time.

Warp-balanced SDDMM employs intra-block edge parallel outperforms existing feature parallel SDDMM. However, the performance of the fused kernel is also influenced by the input graph data and the compute pattern of graph convolution. In some cases, edge parallel SDDMM may be more effective than node parallel warp-balanced SDDMM. The cost of the SDDMM operation can be expressed as $cost_{SDDMM} = \textit{the number of edges} \times \textit{the cost of the operator per edge}$. This implies that the cost for SDDMM increases in two scenarios: first, when the input graph is large, especially if it contains super nodes with numerous neighbors, making it impossible to store SDDMM results in shared memory; and second, when a high-cost edge operator, such as dot, is used in SDDMM. Considering this, if a graph contains super nodes that satisfy the condition $\max_v(\mathcal{N}(v)) \geq \frac{\text{Shared memory capacity}}{\text{Bytes of feature data type}}$ and employs dot-SDDMM, we propose that the performance bottleneck of the fused kernel shifts from CPU launch overhead to unbalanced and low-utilization SDDMM. In these scenarios, DF-GNN utilizes PMF, which employs edge-parallel SDDMM. Conversely, if the condition is not met, SMMF is adopted, incorporating warp-balanced SDDMM, as illustrated in Figure 3.

Softmax. Integrating the Softmax operation into a fused kernel poses a significant challenge due to the absence of the feature dimension. The existing feature parallel fusion strategy, which aligns the number of threads with the feature dimension, is suitable for operations like SDDMM and SpMM that process along this dimension. However, when the feature dimension is absent in the Softmax step, feature dimension alignment introduces duplicated computations. As depicted in Figure 7a, two warps in the same block have to execute identical operations in parallel, including reading S_v from shared memory, performing a max warp reduction, and conducting a sum warp reduction. Additionally, feature parallel fusion fails to cache the reusable Softmax result $\mathbf{F_v}$, resulting in its recomputation during the subsequent SpMM step.

To solve the above challenge in Softmax, DF-GNN designs ***redundancy-free Softmax***, rescheduling threads along the edge dimension rather than the feature dimension to avoid duplicated computations. In this approach, each thread group manages the computation of a row, where threads in a group are flattened to handle the corresponding edges. Warp-level reduction is used to aggregate the attention weights for normalization. The computation of the redundancy-free Softmax is illustrated in Figure 7b. For instance, warp 0 is responsible for processing $\mathbf{S}_0$ of row 0, while warp 1 is assigned to handle $\mathbf{S}_1$ of row 1. Distinct warps do not execute identical tasks, enabling the efficient utilization of hardware resources. Additionally, during the second warp reduction, the redundancy-free Softmax saves the intermediate values $\mathbf{F}_v$ into shared memory. This enables the reuse of these values in the subsequent SpMM step, avoiding the need for recomputation.

SpMM. The SpMM operation, which aggregates the weighted node features to output features, is a key component of the fused AT-GNNs kernel. SpMM operation incorporates with a large number of accesses to global memory, making memory the primary bottleneck affecting SpMM performance.

To address the memory bottleneck, DF-GNN implements ***vectorized SpMM*** which incorporates *vectorized memory access* and *shared memory cache*. Due to the mismatch between the number of threads and the feature dimension, SpMM involves two nested loops: one loop iterates over the feature dimension, and the other iterates over the neighbors. In DF-GNN, the outer loop processes the feature dimensions using vectorized memory access instructions, while the inner loop iterates over neighbors to accumulate intermediate results in shared memory. This shared memory cache reduces the need for repeated global memory accesses. Furthermore, employing vectorized memory access decreases the number of memory access instructions in SpMM. These optimizations significantly lower the memory overhead of SpMM and improve the performance of the fused kernel.

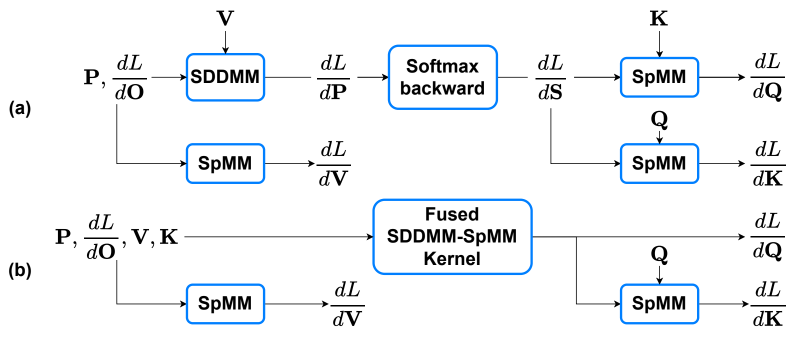

Figure 8: Kernels in GT backward pass. (a) Without any fusion. (b) After DF-GNN fusion.

4.3 Backward Pass Kernel Fusion Optimization

In DF-GNN, we extend the kernel fusion optimization to the backward pass. This extension is motivated by the observation that *the backward pass of SDDMM and SpMM also involve SDDMM and SpMM operations* [14]. The detailed formulations for these backward operations can be found in the appendix B. Figure 8 provides an overview of the GT backward computation. The computation of $\frac{dL}{d\mathbf{Q}}$ consists of three steps: SDDMM, Softmax backward and SpMM, mirroring the SDDMM, Softmax and SpMM pattern in the forward pass. By leveraging this similarity, DF-GNN applies the same fusion strategy to fuse these three kernels into a single fused SDDMM-SpMM kernel. This unified approach eliminates the need for a distinct fusion strategy in the backward pass, reducing the number of required kernels in backward and significantly accelerating the model training.

5 Experiment Evaluation

5.1 Experiment Setup

Table 1: Datasets for evaluation, sorted by avg. degree, * indicates super nodes exist.

Type	Dataset	Abbr.	#Node	#Edge	avg/max degree
Full graph	Citeseer	CS	3.3K	9.2K	2.8 / 99
	Cora	CR	2.7K	10K	3.9 / 168
	Pubmed	PB	19K	88K	4.5 / 171
	Ogbg-ppa*	PP	576K	42M	74 / 3241
	Reddit*	RD	232K	114M	492 / 21657
	Protein*	PR	132K	79M	597 / 7750
Batch graph	COCO-SP	CO	477	2693	5.6
	PascalVOC	PV	479	2709	5.6
	MNIST	MN	71	564	8
	CIFAR10	CF	118	941	8
	CLUSTER	CL	117	4303	36.7
	PATTERN	PN	119	6079	51.1

GNN models. We choose three representative AT-GNNs with the SDDMM-Softmax-SpMM pattern. 1) ***GT*** [4] is similar to the NLP transformer architecture [1], but leverages a sparse attention mechanism by only computing the attention scores between the features of neighboring nodes with dot-SDDMM; 2) ***AGNN*** [7] is similar to GT, but with an L_2 normalization on input node features; 3) ***GAT*** [2] is the most classic AT-GNNs model with add-SDDMM, utilizing attention mechanisms to capture complex relationships within graph-structured data.

Baselines. We compare DF-GNN with the following state-of-the-art baselines in our experiments. 1) ***DGL sparse*** [14] is a sparse operator library under DGL framework and offers a variety of non-fused graph sparse operators, including SpMM and SDDMM; 2) ***PyG*** [15] is a popular GNN framework in which users can utilize message passing abstract to define graph convolutions; 3) ***dgNN*** [18] is a SOTA open-source research work that fuses multiple operators of a GNN models into a single GPU kernel, utilizing a feature parallel fusion strategy, as discussed in Section 2. 4) ***cuGraph*** [27] is an advanced software library developed by NVIDIA, providing APIs to access highly-optimized handcrafted GNN kernels.

Datasets. Table 1 lists the datasets used for evaluation. Batch graph datasets contains numerous small graphs, which are normally used for graph classification tasks. We use the datasets commonly

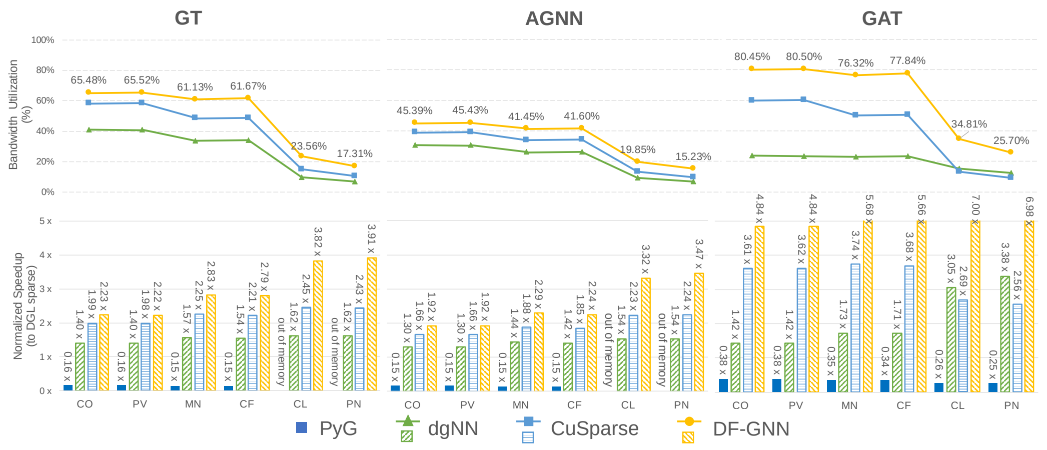

Figure 9: Normalized kernel time speedup and bandwidth utilization on batch graph datasets.

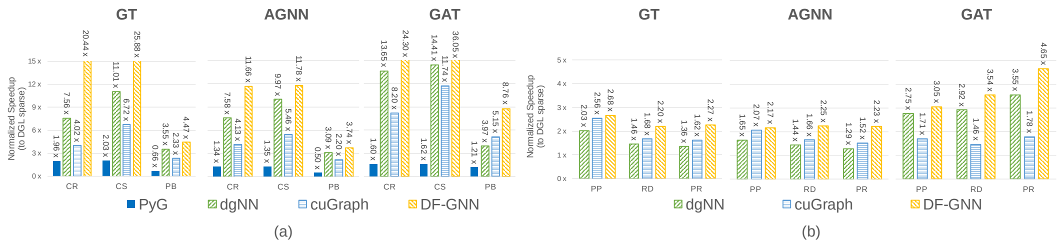

Figure 10: Normalized kernel time speedup on (a) Full graphs and (b) Full graphs with super nodes.

employed for AT-GNNs, including datasets from Benchmarking GNNs [28] (PATTERN, CLUSTER, MNIST, CIFAR10) and Long-Range Graph Benchmark [29] (PascalVOC-SP, COCO-SP). Running AT-GNNs on full graph can be used for node classification tasks. We choose three popular datasets (Cora, Citeseer, and Pubmed), along with three datasets with super nodes (Ogbg-ppa, Reddit, Protein).

Environments. The hardware we used for evaluation is an AWS g5.8xlarge server with one 32-core, 64 thread AMD EPYC 7R32 CPU @ 2.8GHz and a NVIDIA A10G GPU with CUDA 12.2. Regarding the software, we use PyTorch 2.3, torch geometric 2.5 and DGL 2.1.

Model Settings. The feature dimension and batch size are set to 128 and 1024. To ensure a fair comparison, DF-GNN kernels are configured to produce results consistent with the baselines.

5.2 Kernel Performance

Batch graph. Figure 9 shows the speedup and bandwidth utilization of DF-GNN compared to other baselines on the batch graph datasets. For kernel time speedup, DF-GNN achieves the best performance across all datasets, with an average kernel time speedup of 3.7× (and up to 7.0×) compared to DGL sparse across GT, AGNN, and GAT models. Meanwhile, cuGraph and dgNN baselines achieve an average speedup of 2.4× and 1.7×, respectively. The DF-GNN speedup ratio increases with graph degrees, particularly evident in the PATTERN and CLUSTER datasets with high average degrees. This demonstrates that DF-GNN can effectively handle batch graphs with varying degrees, maintaining optimal kernel performance even as the kernel memory access demands rise. Meanwhile, since AT-GNNs kernels are typically memory-bound, memory bandwidth is a crucial determinant of kernel performance. Therefore, we also analysis the bandwidth utilization of different methods, which is calculated as follows: $\frac{\text{Number of read/write bytes}}{\text{Memory bandwidth} * \text{Kernel execution time}}$. Due to the increase in memory access, the bandwidth utilization ratio decreases for all methods as the graph's average degree increases. Compared to all baselines, DF-GNN exhibits the highest bandwidth utilization performance across all datasets, demonstrating its advantages from the perspective of memory access.

Full graph. Figure 10a shows the speedup on the full graph datasets, including the Citeseer, Cora and Pumbed datasets. Overall, DF-GNN achieves an average speedup of 16.3× compared to DGL sparse.

Meanwhile, cuGraph and dgNN baselines achieve average speedups of $8.3\times$ and $11.7\times$, respectively. DF-GNN outperforms the other baselines on all datasets, demonstrating notable robustness. Different from batch graph datasets, DF-GNN, cuGraph, and dgNN achieve higher speedups on full graph datasets. This is because these three graphs are smaller than batch graphs, leading to greater performance gains from kernel fusion. Meanwhile, DF-GNN and dgNN outperform cuGraph due to the cuGraph framework's more complex call stack, which leads to higher overhead for launching CUDA kernels that cannot be hidden with small workloads.

Full graph with super nodes. Figure 10b illustrates the speedup on the full graph datasets with super nodes, including the Ogbg-ppa, Reddit and Protein datasets. Because of the presence of super nodes, DF-GNN employs the PMF method on GT and AGNN convolution, utilizing edge parallel SDDMM to address the performance bottleneck in SDDMM. Among all methods, DF-GNN consistently outperforms the other baselines and achieves an average $2.8\times$ speedup compared to DGL sparse. Additionally, due to the large number of edges in these graphs, the PyG baseline runs out of memory and is not shown. Compared to Figure 10a, the speedup of DF-GNN is reduced because graphs with super nodes are generally larger and exhibit lower kernel launch overhead, diminishing the performance gain from kernel fusion.

Ablation Study. Ablation experiments in Appendix C analyze the performance benefits of each design in DF-GNN and evaluate its robustness across varying feature dimensions, batch sizes and GPU architectures.

5.3 End-to-End Performance

Table 2: The E2E time consumption and speedup of DF-GNN compared to DGL

Time (ms)	Batch graph				Full graph		
	PN	CL	PV	CO	CR	CS	PB
Preprocess	**2.01** (1.20)	**1.81** (1.08)	**1.59** (0.86)	**1.61** (0.85)	-	-	-
Forward	**2.16** (7.86)	**1.94** (7.90)	**3.23** (8.40)	**3.23** (8.64)	**1.99** (7.61)	**2.02** (7.75)	**1.97** (7.69)
Backward	**9.08** (14.9)	**8.72** (14.2)	**8.83** (16.1)	**9.30** (16.4)	**3.87** (8.33)	**3.89** (8.40)	**4.89** (8.28)
Sum	**13.3** (24.0)	**12.5** (23.2)	**13.7** (25.4)	**14.1** (26.0)	**5.86** (15.9)	**5.91** (16.2)	**6.86** (16.0)
E2E Speedup	**1.81x**	**1.86x**	**1.85x**	**1.83x**	**2.72x**	**2.73x**	**2.33x**

Table 2 presents the E2E speedup of DF-GNN in comparison to DGL. For batch graph datasets, achieves an average speedup of $1.84\times$ compared to DGL. However, the preprocess step takes slightly longer for DF-GNN due to the additional conversion to CSR + COO hybrid format for the forward pass and CSC format for the backward pass, whereas DGL only requires the COO format. In the forward and backward pass, DF-GNN attains average speedups of $3.2\times$ and $1.7\times$ respectively compared to DGL. The lower speedup in the backward pass is attributed to the inability to fuse all backward kernels. For full graph datasets, DF-GNN outperforms DGL with an average speedup of $2.6\times$ per training epoch, achieving $3.9\times$ speedup in the forward pass and $2.0\times$ in the backward pass. Similar to its kernel performance, DF-GNN achieves high speedup on full graph datasets because small workloads result in greater performance gains by eliminating kernel launch overhead.

6 Conclusion

In this paper, we present DF-GNN, a framework designed for the efficient execution of AT-GNNs on GPUs. DF-GNN introduces dynamic bi-level thread scheduling and designs tailored thread scheduling for different operations within AT-GNNs, ensuring the flexibility and generality of fusion strategies. To enhance performance on graphs with super nodes, DF-GNN designs two general kernel fusion methods, SMMF and PMF, which dynamically select appropriate parallel strategies for the SDDMM operation. Additionally, DF-GNN enables kernel fusion in the backward pass of AT-GNNs, further accelerating end-to-end AT-GNN training. DF-GNN also integrates with the PyTorch framework and provides APIs to enhance the user experience. Extensive experiments demonstrate that DF-GNN significantly outperforms existing state-of-the-art GNN frameworks across diverse GNN models and datasets. Furthermore, although DF-GNN concentrates on single-GPU optimization, its optimization framework can be readily adapted for the training of distributed AT-GNNs, given the consistency of the underlying GPU kernels.

Acknowledgments We thank anonymous reviewers for their valuable suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 62222111).

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

[3] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*, 2021.

[4] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *arXiv preprint arXiv:2012.09699*, 2020.

[5] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? *Advances in Neural Information Processing Systems*, 34:28877–28888, 2021.

[6] Ladislav Rampášek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. *Advances in Neural Information Processing Systems*, 35:14501–14515, 2022.

[7] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. Attention-based graph neural network for semi-supervised learning. *arXiv preprint arXiv:1803.03735*, 2018.

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

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

[10] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 31, 2018.

[11] Zhaocheng Zhu, Zuobai Zhang, Louis-Pascal Xhonneux, and Jian Tang. Neural bellman-ford networks: A general graph neural network framework for link prediction. *Advances in Neural Information Processing Systems*, 34:29476–29490, 2021.

[12] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. *arXiv preprint arXiv:1912.09893*, 2019.

[13] Fan-Yun Sun, Jordan Hoffmann, Vikas Verma, and Jian Tang. Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization. *arXiv preprint arXiv:1908.01000*, 2019.

[14] Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[15] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

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

[17] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 359–375, 2021.

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

[19] Qiang Fu, Yuede Ji, and H Howie Huang. Tlpgnn: A lightweight two-level parallelism paradigm for graph neural network computation on gpu. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 122–134, 2022.

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

[21] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[22] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 256–266. IEEE, 2021.

[23] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. Graphiler: Optimizing graph neural networks with message passing data flow graph. *Proceedings of Machine Learning and Systems*, 4:515–528, 2022.

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

[25] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 660–678, 2023.

[26] Zhongming Yu, Guohao Dai, Guyue Huang, Yu Wang, and Huazhong Yang. Exploiting online locality and reduction parallelism for sampled dense matrix multiplication on gpus. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 567–574. IEEE, 2021.

[27] rapidsai. Rapids cugraph, 2024. URL https://github.com/rapidsai/cugraph.

[28] Vijay Prakash Dwivedi, Chaitanya K Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *Journal of Machine Learning Research*, 24(43):1–48, 2023.

[29] Vijay Prakash Dwivedi, Ladislav Rampášek, Michael Galkin, Ali Parviz, Guy Wolf, Anh Tuan Luu, and Dominique Beaini. Long range graph benchmark. *Advances in Neural Information Processing Systems*, 35:22326–22340, 2022.

A Edge Softmax in AT-GNNs

Here is the vectorized formulation of Softmax operation in AT-GNNs, which normalizes the sparse attention weight matrix **S**. To enhance the numeric stability of Softmax, the implementation employs a normalization trick, as illustrated in Equation (2). It involves two rounds of information exchange between nodes and edges: the first round is to obtain the maximum value $z_v^{(max)}$ from the corresponding row, while the second round sums all connected edge values f_{uv}.

$$\begin{aligned}\text{Round.1: } z_v^{(max)} &= \max_{u\in\mathcal{N}(v)}(s_{uv}) \\ f_{uv} &= \exp(m_{uv} - z_v^{(max)}) \\ \text{Round.2: } z_v^{(sum)} &= \sum_{u\in\mathcal{N}(v)} f_{uv} \\ p_{uv} &= \frac{f_{uv}}{z_v^{(sum)}}\end{aligned} \tag{2}$$

B Backward pass in AT-GNNs

Backward pass of SpMM. The backward pass of SpMM in Equation (1), as derived from the loss L, consists of one SDDMM operation and one SpMM operation, which can be represented by the following equations:

$$\frac{\mathrm{d}L}{\mathrm{d}\mathbf{P}} = \left(\frac{\mathrm{d}L}{\mathrm{d}\mathbf{O}} * \mathbf{V}^T\right) \odot \mathbf{I} \tag{3}$$

$$\frac{\mathrm{d}L}{\mathrm{d}\mathbf{V}} = \mathbf{P}^T * \frac{\mathrm{d}L}{\mathrm{d}\mathbf{O}} \tag{4}$$

Backward pass of SDDMM. The backward pass of SDDMM in Equation (1), as derived from the loss L, involves two SpMM operations, represented by the following equations:

$$\frac{\mathrm{d}L}{\mathrm{d}\mathbf{Q}} = \frac{\mathrm{d}L}{\mathrm{d}\mathbf{S}} * \mathbf{K} \tag{5}$$

$$\frac{\mathrm{d}L}{\mathrm{d}\mathbf{K}} = \frac{\mathrm{d}L}{\mathrm{d}\mathbf{S}}^{T} * \mathbf{Q} \tag{6}$$

C Ablation Study

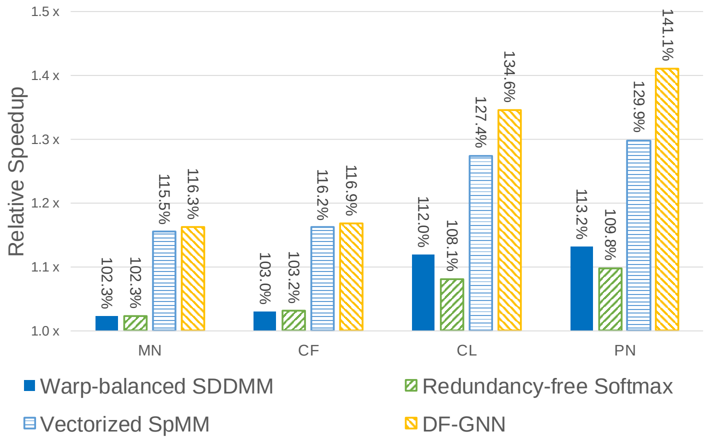

Figure 11: Relative performance improvement brought by proposed optimizations on GT.

Proposed optimization methods. Figure 11 presents the relative speedup obtained when the three optimization methods proposed in Section 4.2. This demonstrates the impact of the three optimization methods on the final kernel performance. Across all four datasets, the SpMM optimization achieves significant performance improvements, indicating that memory access for SpMM is the primary performance bottleneck in the AT-GNNs kernel. Meanwhile, the optimizations for SDDMM and Softmax provided only a 2% to 3% performance improvement for datasets with low node degrees, such as MNIST and CIFAR10. However, for datasets with high node degrees, such as CLUSTER and PATTERN, these optimizations yield approximately a 10% performance gain. This indicates that the performance gain from the SDDMM and Softmax optimizations depend on the properties of the graph. The greater the number of edges in the graph, the higher the performance gains from these optimizations.

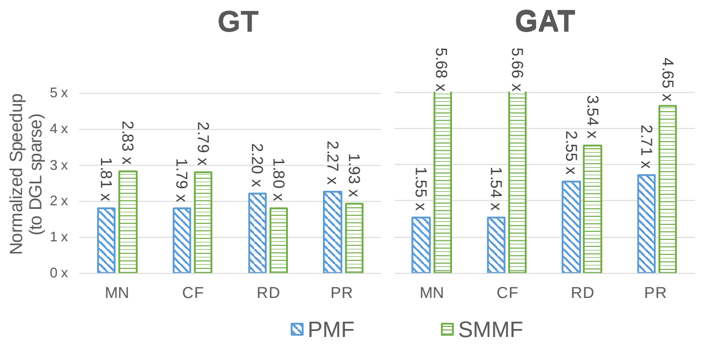

Figure 12: Normalized kernel time speedup of SMMF and PMF in DF-GNN.

SMMF and PMF. Figure 12 shows the speedup of SMMF and PMF. For GT convolution with dot-SDDMM, SMMF outperforms PMF on the MNIST and CIFAR10 datasets. Meanwhile, on

Reddit and Protein datasets with super nodes, PMF shows improved performance compared to SMMF. Nonetheless, the incomplete fusion of kernels in PMF leads to only a modest enhancement when compared to SMMF. For GAT convolution with add-SDDMM, SMMF demonstrates better performance than PMF across all four datasets. This indicates that the PMF method can better handle cases where the input graph contains super nodes and the graph convolution uses dot-SDDMM, demonstrating that the choice between node parallel SDDMM and edge parallel SDDMM in Section 4.2 is reasonable.

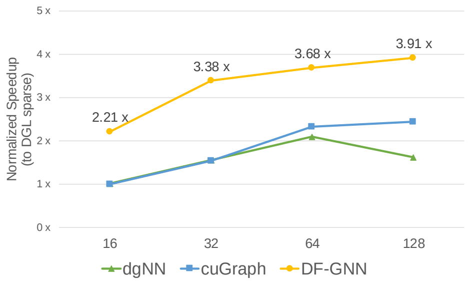

Figure 13: Normalized speedup of GT against different feature dimension (x-axis).

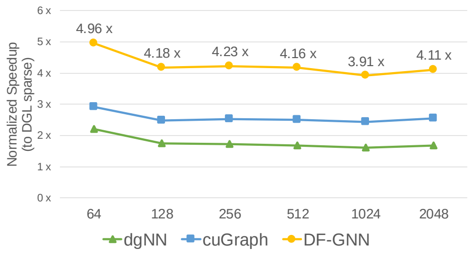

Figure 14: Normalized speedup of GT against different batch sizes (x-axis).

Feature Dimension. Figure 13 shows the normalized kernel time speedup as the feature dimension increases from 16 to 128 on the PATTERN dataset. DF-GNN achieves an average of 3.30× speedup, while the dgNN and cuGraph baselines achieve 1.74× and 1.56× speedup, respectively. DF-GNN consistently achieves better performance compared to the baselines across different feature dimension settings. This demonstrates the extensibility and robustness of DF-GNN in handling varying feature dimensions. The strong performance of DF-GNN is further validated by its ability to outperform state-of-the-art methods under diverse feature dimension configurations.

Batch Size. Figure 14 shows the normalized kernel time speedup as the batch size increases from 64 to 2048 on the PATTERN dataset. Compared to dgNN and cuGraph, DF-GNN achieves an average speedup of 4.26× and exhibits superior performance across all batch sizes. This means that the throughput of DF-GNN can scale proportionally with an increasing batch size, demonstrating its effective handling of graphs with large batch sizes and efficient utilization of GPU resources.

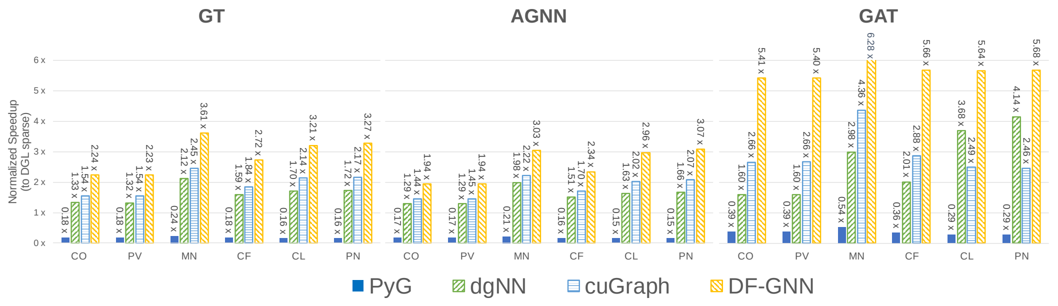

Figure 15: Normalized kernel time speedup on NVIDIA H100 80GB.

Evaluation on Different GPUs. The DF-GNN exhibits robust generalization capabilities, making it applicable across various GPU architectures. Figure 15 illustrates the normalized kernel time speedup on the NVIDIA H100 80GB GPU. Consistent with the experiment results shown in Figure 9, DF-GNN outperforms all baselines across all datasets, achieving an average kernel time speedup of 3.81× (and up to 6.28×) when compared to DGL sparse across the GT, AGNN and GAT models. This underscores the adaptability of DF-GNN's optimization techniques across different GPU architectures, highlighting its potential for widespread applicability in accelerating GNN workloads.