

---

# Cached Operator Reordering: A Unified View for Fast GNN Training

---

**Julia Bazinska**  
ETH Zurich  
Zurich, Switzerland  
{firstname}.{lastname}@inf.ethz.ch

**Andrei Ivanov**  
ETH Zurich  
Zurich, Switzerland  
{firstname}.{lastname}@inf.ethz.ch

**Tal Ben-Nun**  
ETH Zurich  
Zurich, Switzerland  
{firstname}.{lastname}@inf.ethz.ch

**Nikoli Dryden**  
ETH Zurich  
Zurich, Switzerland  
{firstname}.{lastname}@inf.ethz.ch

**Maciej Besta**  
ETH Zurich  
Zurich, Switzerland  
{firstname}.{lastname}@inf.ethz.ch

**Siyuan Shen**  
ETH Zurich  
Zurich, Switzerland  
{firstname}.{lastname}@inf.ethz.ch

**Torsten Hoefler**  
ETH Zurich  
Zurich, Switzerland  
{firstname}.{lastname}@inf.ethz.ch

## Abstract

Graph Neural Networks (GNNs) are a powerful tool for handling structured graph data and addressing tasks such as node classification, graph classification, and clustering. However, the sparse nature of GNN computation poses new challenges for performance optimization compared to traditional deep neural networks. We address these challenges by providing a unified view of GNN computation, I/O, and memory. By analyzing the computational graphs of the Graph Convolutional Network (GCN) and Graph Attention (GAT) layers—two widely used GNN layers—we propose alternative computation strategies. We present *adaptive operator reordering with caching*, which achieves a speedup of up to 2.43x for GCN compared to the current state-of-the-art. Furthermore, an exploration of different caching schemes for GAT yields a speedup of up to 1.94x. The proposed optimizations save memory, are easily implemented across various hardware platforms, and have the potential to alleviate performance bottlenecks in training large-scale GNN models.

## 1 Introduction

Neural networks have transformed the way various real-world problems are solved, from computer vision to natural language processing. However, most deep neural network (DNN) approaches do not allow for straightforward processing of structured graph data, such as molecules, social networks or knowledge graphs. There exist complex methods allowing for limited processing of such structured data by regular DNNs.

Some examples include bag-of-atoms [2], which represent chemical compounds, or hierarchical processing of 3D point clouds, which allows the inclusion of local neighborhood information in the computation [30]. Even though these approaches allow processing structured graph data with DNNs, they are lossy representations that do not include the full information on data connections and their properties. To overcome these limitations, a new computational paradigm emerged.

Preprint. Preliminary work.

Graph Neural Networks (GNNs) are a method of performing neural network computation on graph data. They can be used for solving problems such as node classification, graph classification, and clustering [7, 11, 12, 16, 19, 33, 39, 46, 47]. Example applications include recommendations in social networks [13] or document classification in citation networks [26]. Another common use of GNNs is computer vision, where they are used to analyse 3d point clouds [35] or to match image key-points [32].

With the processing of larger datasets and models, optimizing the performance of GNNs becomes crucial. GNNs differentiate from traditional DNNs in performance due to their predominant use of sparse computation. The input graph adopts a sparse adjacency matrix representation. Operations involving an exchange of information between neighboring nodes rely on this graph structure, and thus, operate on sparse data. As a result, GNN model training and evaluation often face memory-bound challenges, unlike dense models. Consequently, traditional dense-oriented performance optimization methods are often not directly suitable [8].

Several factors impact GNN runtime, with memory throughput being only one aspect. Additional considerations include floating point operations executed, potential time-memory trade-offs between saving values for the backward pass, and characteristics of the input graph. Numerous GNN-focused strategies have been developed to address these performance issues, such as operator fusion and reordering [45], or automatic graph optimization [40]. Nonetheless, these approaches lack a systematic method for analyzing the runtime effects of high-level performance optimization decisions. In this work, we propose a unified view of the computational graphs, I/O, and memory utilization of the most common GNN layers. This framework enables us to understand the performance implications of various computation schemes and data formats. Guided by this analysis, we propose alternative computation schemes for two widely used GNN layers: the Graph Convolutional Network layer (GCN) [26] and the Graph Attention layer (GAT) [38].

The main contributions of this work are:

- **Analysis of GNN Performance Factors.** We investigated three aspects that influence GNN performance, which include the optimal selection of sparse data formats, the choice between caching and recomputing intermediate values, and the organization of operations within the computation graph. These considerations shape the performance of GNN implementations depending on the input dataset.
- **Adaptive Computational Scheme.** Drawing insights from theoretical analysis, we leverage the advantages of adaptively selecting caching strategies and operation ordering for the implementation of GCN and GAT. Additionally, our approach integrates the observed impact of selected sparse data formats on a range of empirically evaluated datasets.
- **Implementation of Proposed Computation Scheme.** We implemented GAT and GCN within the DaCe [4] framework. This implementation yielded a training acceleration of up to 2.43x for GCN and up to 1.94x for GAT compared to the state-of-the-art implementations available in PyTorch Geometric [14] (PyG).

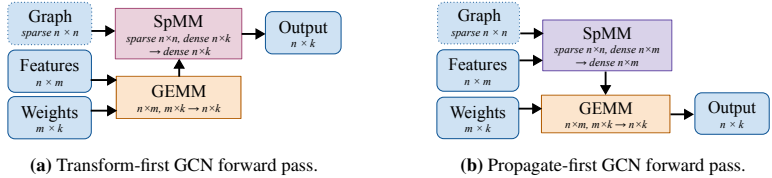
## 2 Background

GNNs are a type of neural networks that are designed to process graph data, such as social networks, citation graphs or spatial structures. In GNN computation, an iterative exchange of information between a node and its neighbors, called *message passing*, takes place. Each node receives messages from its neighbors and computes its new *node feature vector* using an *aggregation function*. Each message-passing iteration is represented as a layer in the neural network.

Model training is performed similarly to classical DNNs, i.e., the value of the loss function is optimized through the use of a gradient descent method that leverages backpropagation. Many variations of GNN layers exist. In this work, we narrow our focus to GCNs and GATs.

Typically, graph data consists of *nodes* characterized by some *node features* and are connected by *edges*. Edges can optionally also have descriptors, called *edge weights*, if they are scalars, or *edge features* if they are vector data. In the following sections, the number of vertices is denoted by  $n$  and the number of edges by  $e$ . The typical way of representing input graphs is to provide the so-called *adjacency matrix*  $\mathbf{A} = (a_{ij})_{1 \leq i, j \leq n} \in \mathbb{R}^{n \times n}$  where  $a_{ij} = 1$  if nodes are adjacent and  $a_{ij} = 0$  otherwise. Furthermore, node features are represented in a *feature matrix*  $\mathbf{X} \in \mathbb{R}^{n \times m}$ .

**GCN.** The Graph Convolutional Network layer is a simple GNN operator that is analogous to a convolution operator. Firstly, the input features for each node are linearly projected into the new



**Figure 1:** Two schemes of computing the forward pass for GCN. Compute nodes of the same color operate on the same shapes, which are indicated on the node in Einstein summation notation.

feature space, and then, for each node, the features of adjacent nodes are aggregated to create the final output features (Appendices G.2 and G.4). In this work, we consider GCNs with summation as the aggregation operation.

**GAT.** The Graph Attention layer is a more complex operator that employs an attention mechanism over the edges connected to each node. After all node features are projected into the new feature space, the *attention weights*, represented as a sparse matrix  $\mathcal{A} \in \mathbb{R}^{n \times n}$ , are computed for each edge. The attention weight for each edge is computed based on the source and destination node features using another parameterized linear projection (Appendices G.2 and G.4).

**SpMM.** The *sparse matrix-matrix multiplication* operator is a multiplication of a sparse matrix  $\mathbf{A} \in \mathbb{R}^{n \times m}$  and a dense matrix  $\mathbf{B} \in \mathbb{R}^{m \times k}$ , resulting in a dense matrix  $\mathbf{C} \in \mathbb{R}^{n \times k}$ ,  $\mathbf{C} = \mathbf{A}\mathbf{B}$ . There exist highly optimized implementations of this operator [29] and it is widely explored in literature [34, 37] (Section 5). The SpMM operator, as explained in Section 2, is a very common operator in GNNs. It represents the propagation of information between nodes. To understand the computational characteristics of SpMM, we look at *operational intensity*. Operational intensity for SpMMs executed in GNNs on datasets considered in this work is no higher than  $1.066 \frac{\text{FLOP}}{\text{byte}}$  (Table 5) which indicates that the computation is memory-bound.

**SDDMM.** Another common operator in GNN computation is the *sampled dense-dense matrix multiplication* [6]. Given a sparse matrix  $\mathbf{A} \in \mathbb{R}^{n \times k}$  and two dense matrices  $\mathbf{B} \in \mathbb{R}^{n \times m}$ ,  $\mathbf{C} \in \mathbb{R}^{m \times k}$ , we can compute  $\mathbf{D} = \mathbf{A} \odot (\mathbf{B} \cdot \mathbf{C})$ , where  $\odot$  represents the Hadamard product and  $\mathbf{D} \in \mathbb{R}^{n \times k}$  is a sparse matrix. Similar to SpMM, this subroutine has existing highly optimized implementations [29]. SDDMM is used in the backward pass of GAT. There, we always need to compute it on matrices with shapes  $\mathbf{A} \in \mathbb{R}^{n \times n}$ ,  $\mathbf{B} \in \mathbb{R}^{n \times f}$  and  $\mathbf{C} \in \mathbb{R}^{f \times n}$ . SDDMM is also memory-bound (Table 6).

### 3 Algorithmic View on Graph Neural Networks

In this section, we provide a unified view of the computational graph, I/O, and memory for the GCN layer and the GAT layer in order to draw conclusions that would allow us to develop faster GNN implementations. By taking a principled, high-level approach with the awareness of the low-level performance impact, we find new ways to execute GNN layers and save compute.

#### 3.1 Analysis of the GCN Computational Graph

In order to compute the output of a GCN layer (Equation (3)), the computation depicted in Figure 1 is necessary. The order of SpMM and GEMM can be altered without impacting the operational outcome. However, it does influence the size of matrices involved in the multiplication. We can either initiate the transformation of features followed by propagation, resulting in the computation of  $\mathbf{A} \cdot (\mathbf{X}\Theta)$ , or opt for the reverse sequence, computing  $(\mathbf{A} \cdot \mathbf{X}) \cdot \Theta$ .

The figure reveals that the GEMM is performed on matrices of equal dimensions in both scenarios. The primary difference resides in the SpMM. In the scenario depicted in Figure 1a the SpMM operates on matrices sized  $n \times n$  and  $n \times k$ . Conversely, the approach in Figure 1b utilizes matrices sized  $n \times n$  and  $n \times m$ . Consequently, the former case entails a computation of  $2(qm + nmk)$  FLOPs, while the latter involves  $2(qk + nmk)$  FLOPs.

To minimize runtime, the optimal strategy is to initiate feature transformation if  $k < m$ , and to start with propagation if  $k \geq m$ . This holds under the assumption that the execution time for an SpMM on matrices of  $n \times n$  and  $n \times t$  scales linearly with  $t$ , where  $t$  is an integer. This assumption is based on the asymptotic linear scaling of FLOPs and memory transfers in SpMM implementations using common sparse formats (Table 5). In reality, the factors influencing runtime are more nuanced and can be dependent on elements such as the exact implementation of the SpMM, the size of the working

**Table 1:** Summary of differences between alternative GCN implementations. Entries in the ‘‘Usage condition’’ column were derived to minimize the total number of operations and memory transfers in the computation. In each pair of alternative schemes, the GEMMs are executed on the same shapes, while the SpMMs are executed on different shapes.

Operation	Scheme (Figure)	GEMM input sizes	SpMM input sizes	# Transient values	Usage condition
GCN forward	Transform-first (1a)	$nm, mk$	$nn, nk$	$nk$	$k < m$
	Propagate-first (1b)		$nn, nm$	$nm$	$k \geq m$
GCN backward with feature gradient	Fused-propagate (12a)	$nm, mk$	$nn, nk$	$nk$	$k < 2m$
	Split-propagate (12b)	$nm, nk$	$2\times: nn, nm$	$2nm$	$k \geq 2m$
GCN backward no feature gradient	Fused-propagate (12a)	$nm, mk$	$nn, nk$	$nk$	$k < m$
	Split-propagate (12b)	$nm, nk$	$nn, nm$	$nm$	$k \geq m$

**Table 2:** Comparison of GCN computation schemes with caching. The case presented in Figures 1b and 12b is omitted because it always executes more operations than the scheme using caching. In each pair of alternative schemes, the GEMMs are executed on the same shapes, while the SpMMs are executed on different shapes.

Operation	Forward scheme (Figure)	Backward scheme (Figure)	GEMM input sizes	SpMM input sizes	# Transients (fwd; bwd)	Usage condition
GCN, with feature gradient	Transform-first (1a)	Fused-propagate (12a)	$nm, mk$	$2\times: nn, nk$	$nk; nk$	$k < m$
	Propagate-first with caching (13a)	Split-propagate with caching (13b)	$nm, nk$	$2\times: nn, nm$	$nm; nm$	$k \geq m$
GCN, no feature gradient	Transform-first (1a)	Fused-propagate (12a)	$nm, mk$	$2\times: nn, nk$	$nk; nk$	$k < \frac{1}{2}m$
	Propagate-first with caching (13a)	Split-propagate with caching (13b)	$nm, nk$	$nn, nm$	$nm; 0$	$k \geq \frac{1}{2}m$

set, the GPU cache sizes and the sparse matrix structure. While we could attempt to model SpMM runtime in a very detailed way, our simpler approach is mostly satisfactory, as can be seen from the results (Section 4).

Extending this analysis to the backward pass, we summarize our recommendations on which scheme should be employed in Table 1. Thus, we propose an *adaptive computation scheme* that executes the operations according to the recommendations. In addition to FLOPs, we also consider the amount of memory required by intermediate values in the computation, which is also shown to be smaller when the recommendation is followed. This way, we minimize both the amount of computation needed and the required amount of memory for executing the computation.

**Caching intermediate values.** The SpMM result  $\mathbf{AX}$  appears both in forward (Equation (3)) and backward (Equation (11)) passes. Thus, to avoid recomputing, the result can be cached and reused in the backward pass. Caching the aggregated features  $\mathbf{AX}$  allows to compute one less SpMM. In Table 2 we present an analysis of the caching scheme illustrated in Figure 13.

In the case described here, caching adds no memory overhead as it avoids storing some intermediates (Appendix A). Moreover, as can be seen in Table 2, computing GCN with caching requires less memory for transients than the same scheme without caching.

**Generalizability of the adaptive scheme.** The proposed adaptive scheme with caching can be applied to any GNN computation that includes the chained multiplication depicted in Figure 1. Therefore, our findings could be applicable to other GNN layers employing similar chained multiplication, for example, the Graph Isomorphism Network layer [41] or the GraphSAGE operator [20]. This approach is completely hardware-agnostic, making it suitable for CPU computation, GPU computation, and other accelerators. Although our work operates under the assumption of a summation-based aggregation function for neighboring nodes, it can be extended to accommodate any aggregation function that has the distributive property, e.g. mean.

### 3.2 Analysis of the GAT Computation

The computational characteristics of GAT are different from those of GCN (Appendix G.3). We cannot use a method analogous to what was shown for GCN in subsection 3.1 because it requires computing  $\mathcal{A}(\mathbf{X}\Theta)$ , where  $\mathcal{A}$  is calculated using the value of  $\mathbf{X}\Theta$ . Another difference in relation to

Table 3: Comparison of time-memory trade-offs in GAT.

Cached values	Additional memory use	Saved FLOPs	Saved I/O
Transformed features	$4nhk$	$\mathcal{O}(nhkm)$	$\mathcal{O}(nm + mhh + nhk)$
Transformed features, node attention	$4nh(k + 2)$	$\mathcal{O}(nhkm)$	$\mathcal{O}(nm + mhh + nhk)$
Transformed features, edge attention weights, edge mask	$4nhk + 5qh$	$\mathcal{O}(nhkm + qh)$	$\mathcal{O}(nm + mhh + nhk + qh)$

GCN is that GAT does not consider edge weights of the input graph. Instead, the attention weights, which are computed based on node features, are used.

Similar to Section 3.1, we investigated other opportunities for operator reordering in the GAT operator (Appendix B). We concluded that the optimal scheme is not dependent on the input and is already in use in existing frameworks.

**Multi-head GAT.** GAT layers are often used with multiple heads. From the computational perspective, that results in almost every operator in the computation having an additional dimension. From the performance perspective, it can mitigate the issue of uncoalesced memory accesses if the data is arranged correctly: depending on the computation, the head dimension should be either last or penultimate. Usually when operating on the graph structure, the program often needs to access the memory holding values for neighbors of a given node, which are not contiguous. Thus, when a cache line is loaded, only a single value from it is used because others belong to other nodes. Once enough heads in GAT are used, the loaded cache line holds values for the same node but of different heads. All of those values are used, because the same computation has to be executed on all heads.

Moreover, a model with multiple heads implies that we need to execute SpMM and SDDMM in a *semi-batched* fashion, meaning that the sparse structure is shared across the batch. In the case of SpMM, we need to multiply  $h$  sparse matrices of shapes  $n \times n$  that all share the same sparse structure with a dense matrix of shape  $n \times h \times k$ , where the middle dimension is the batch dimension.

Similarly, regarding SDDMM, we multiply  $h$  sparse matrices sharing the same sparse structure with a result of a multiplication of two dense matrices batched along the middle dimension. Having the batch dimension not as the leading dimension, but as the middle dimension potentially allows for more contiguous memory accesses. However, such semi-batched computation is not directly supported by optimized vendor libraries such as CuSPARSE [29].

**Caching intermediate values.** We analyze caching opportunities for the forward and backward passes of GAT (Appendix C). Caching more variables enables us to reduce computation time in the backward pass, although it does entail a memory cost. The trade-offs discovered between computational benefits and memory costs are summarized in Table 3. We evaluate different caching strategies in subsection 4.2.

## 4 Evaluation

The network architecture used for GCN evaluation was a simple GNN network with two GCN layers and a Rectified Linear Unit (ReLU) activation between them. For GAT, the network architecture consisted of two GAT layers with 8 heads each and an Exponential Linear Unit (ELU) activation function between them. The sizes of the internal hidden representations vary between experiments in order to benchmark different model sizes. For computing the parameter gradients, the mean squared error loss function was used for full networks and a simple sum in case of single layer benchmarking.

**Baseline.** Implementations of GAT and GCN layers from PyTorch Geometric (PyG) were used as baselines. Starting from Pytorch 2.0 and PyTorch Geometric 2.3.0, it is possible to compile the model code to obtain a much faster model. This functionality is supported only for the COO format (Appendix D). Thus, in this work, we report the numbers for compiled PyTorch models with the COO format and not compiled models with CSR format. Additionally, for GAT, we report the results for the dGNN [45] implementation of the operator, which is also available as part of PyTorch Geometric. The dGNN implementation of GAT requires the graph matrix to be stored in both CSC and CSR, thus storing the graph data twice. The dGNN implementation of GAT consists of hand-written CUDA kernels.

Table 4: Graph datasets used in this work.

Dataset	Nodes	Edges	Features	% NNZ	Classes	Avg. node degree
Cora [42]	2,708	10,556	1,433	0.144%	7	7.8
Citeseer [42]	3,327	9,104	3,703	0.082%	6	5.47
PubMed [42]	19,717	88,648	500	0.023%	3	8.99
Flickr [44]	89,250	899,756	500	0.011%	7	5.47
OGB-Arxiv [21]	169,343	1,166,243	128	0.004%	40	13.77

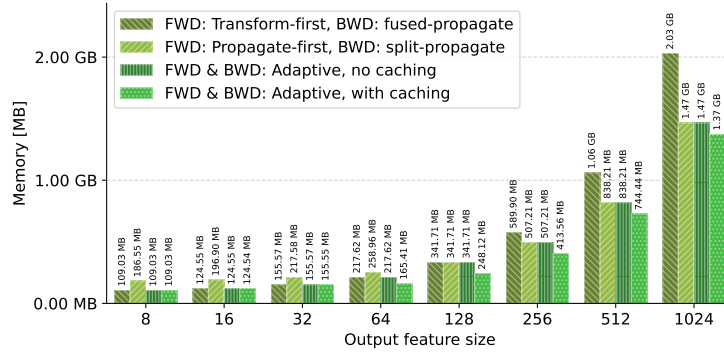


Figure 2: Single GCN layer memory use on the OGB Arxiv dataset.

**Benchmarking.** All of the following experiments were executed on a machine with Intel(R) 6130 @ 2.10GHz CPU, 1.5 TB RAM, and an NVIDIA Tesla V100 16GB PCIe GPU. Each benchmarked computation was run with 10 warm-up iterations and then executed 100 times in 10 blocks in the case of the forward pass or 20 times in 5 blocks in the case of the backward pass (lower because of long execution times) in order to minimize measurement overheads. All reported results for runtimes are medians of  $\frac{1}{100}$  or  $\frac{1}{20}$  of block execution times. Standard deviation is also plotted in the figures but is mostly unnoticeable due to low variance in the results. The datasets used in benchmarking are summarized in Table 4.

### 4.1 Graph Convolutional Network

Our proposed method of choosing the optimal implementation depends on the number of input and output features. Thus, we execute a single GCN layer with varying numbers of output features between 8 and 1024. We use the OGB Arxiv dataset, which has 128 node features, and evaluate layers with varied output sizes. We assess both situations, one where input feature gradients are required and another where they are not necessary. We provide the results for schemes both with and without caching. The dataset is represented in the CSC format.

The threshold for switching between the schemes turned out to be 128 or 256 output features (Appendix F), depending on the computation: whether it is performed without or with feature gradient computation. This threshold matches the suggested usage conditions from Tables 1 and 2. Thus, our adaptive implementation can correctly identify this threshold. As a result, it performs as fast as the fastest scheme.

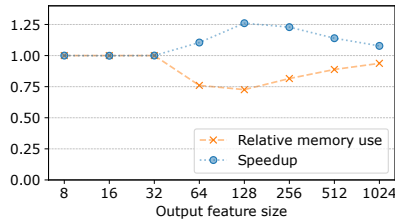


Figure 3: GCN runtime speedup (higher is better) and used memory (lower is better) of the adaptive scheme with caching in comparison to the adaptive scheme without caching.

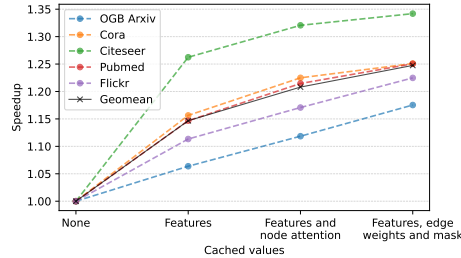
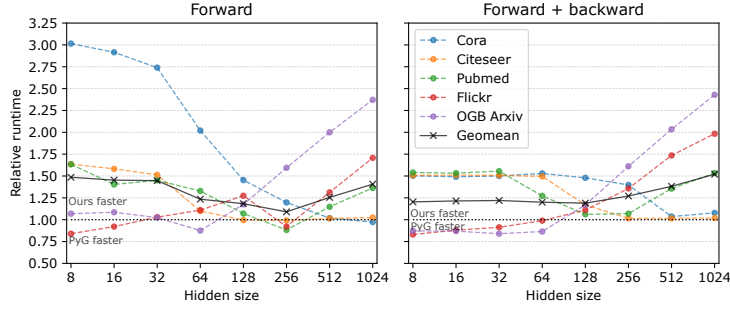


Figure 4: Speedup between different GAT caching schemes for each dataset. Geometric mean is indicated in black.



**Figure 5:** Runtime of our adaptive scheme with caching for GCN relative to the fastest PyTorch Geometric runtime. Geometric mean is indicated in black.

**Memory usage.** In Section 3.1, we mentioned that caching intermediate features does not add extra memory cost. This is backed by Figure 2, which shows memory savings from caching as expected. Caching avoids one SpMM operation, removing the need for an extra array of size  $n \times m$ . Figure 3 illustrates speedup and memory use from caching versus not caching in the same scheme. For output sizes up to 32, both programs act similarly due to the same transform-first, fused-propagate scheme. With larger output sizes, caching becomes valuable: it exhibits a speedup of up to 25% while using 25% less memory. However, for bigger output sizes, the caching advantage lessens as the runtime is dominated by GEMM execution. Thus, adaptive computation benefits certain models, and if the computation scheme allows it, caching intermediate results is always useful.

**Evaluation against baselines.** After proving our adaptive implementation’s optimal computing scheme, we compare it to the baselines on the mentioned two-layer GCN network. We measure the time for a complete forward and backward pass. Runtime results are summarized in Figure 5 (more in Figure 15).

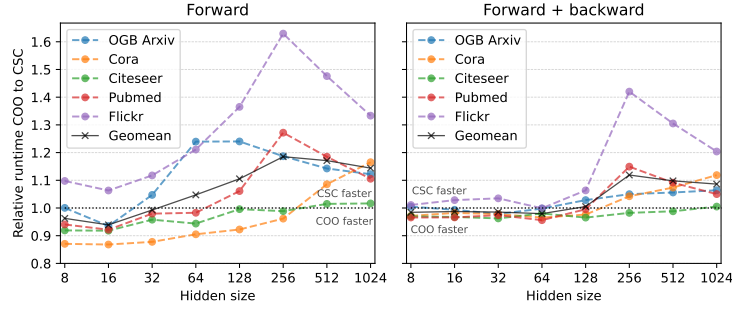
In cases where the hidden size is bigger than the input feature size, we benefit greatly from the adaptive scheme. PyTorch Geometric uses only the transform-first scheme for forward and fused-propagate for backward which clearly stands out in the experimental results. At times, our implementation is slower than PyG. This arises when our approach mirrors PyG’s computation strategy, specifically the transform-first and fused-propagate methods. The primary distinction between our implementation and PyG pertains to the computation of SpMM and ReLU operations. Our method relies on the CuSPARSE library’s optimized subroutine for SpMM, but this prevents us from seamlessly fusing the operator with the subsequent element-wise activation function. In contrast, PyG sacrifices computational flexibility for fine-tuned operator performance. It dynamically generates SpMM code, enabling fusion with the subsequent element-wise operator.

For hidden sizes up to 64, our approach proves faster for smaller datasets like Cora, Citeseer, and Pubmed. Conversely, PyG’s approach excels with larger datasets such as Flickr and OGB Arxiv. For increased hidden sizes, we either benefit from an alternative computing scheme, as seen in Arxiv, Flickr, and Pubmed, or achieve performance nearly on par with PyG, exemplified by Citeseer and Cora. This leads us to conclude that CuSPARSE’s SpMM likely harnesses the small dataset size to achieve higher performance. However, when data grows too large, CuSPARSE’s SpMM performance aligns with PyG’s generated SpMM, albeit without the advantage of fused activation function integration.

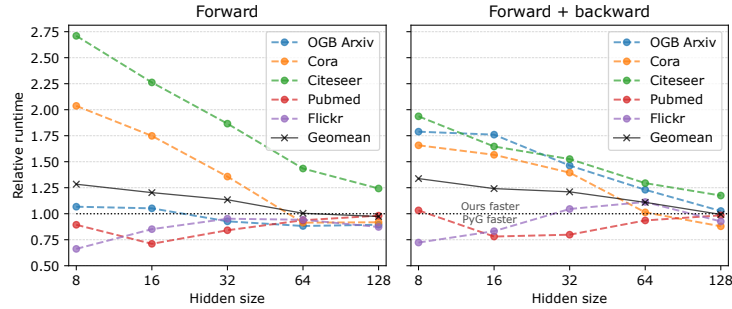
The data format choice significantly affects runtime. Comparing COO and CSC formats (see Figure 6), CSC is up to 1.63 times faster than COO. On average, CSC is 1.14 times faster for forward passes and 1.07 times faster for both passes combined. This impact varies among datasets, with larger hidden sizes favoring CSC due to CuSPARSE’s closed-source CSC SpMM subroutines. These utilize a preprocessing step, distributing GPU thread work fairly. As hidden sizes grow, preprocessing takes less time relative to matrix multiplication, reducing overhead for bigger problems. Unlike CSC, COO SpMM variant doesn’t need this step, performing better for smaller problems like Cora (hidden size 8). However, for larger graphs, CSC excels by loading less data for sparse representation and benefiting from organized non-zero entries in columns, thus enhancing coalescence.

## 4.2 Graph Attention Network

We benchmark GAT with different caching schemes in order to evaluate their time-memory trade-offs. We run a 2-layered GAT model with 8 heads and varied number of hidden features. We



**Figure 6:** Data format comparison: relative runtime of GCN using the COO format to GCN using the CSC format. Values above 1.0 indicate that using CSC is faster, below 1.0 indicate that using COO is faster. Geometric mean across datasets is plotted in black. There is a high variance between datasets but it can be seen that the larger the hidden size, the faster CSC tends to be.



**Figure 7:** GAT comparison against baselines: the runtime of our implementation relative to the fastest PyTorch Geometric implementation for the given dataset and hidden size. Values above 1.0 indicate that ours is faster. The geometric mean of the speedups is indicated in black.

evaluate hidden sizes of 8, 16, 32, 64, 128. A model with 8 heads and hidden size of 128 has a comparable number of parameters as a GCN model with 1024 hidden size. We compare against various implementations available in PyTorch Geometric: compiled COO, CSR, dGNN using CSC and CSR, both compiled and not.

**Caching schemes evaluation.** We evaluate four caching schemes as described in Table 3: no caching, caching only the transformed features, caching features and node attention, caching features together with final edge weights and LeakyReLU mask. Aggregated results for different datasets can be seen in Figure 4 (more in Figures 16 and 17).

**Evaluation against baselines.** We choose the fastest caching scheme, which caches the transformed features, edge weights, and the LeakyReLU mask, to benchmark against PyG. We summarize the results in Figure 7 (more in Figure 18), where we show the speedup of our implementation against the fastest of PyG implementations for the given dataset and hidden size.

The PyG implementations use different caching strategies. Compiled edge list and CSR both cache as much as possible, i.e., the transformed features, the edge weights, and the edge mask. In the dGNN implementation, only the transformed features and the node attention are cached. Our implementation outperforms the baselines on average. Similar to the GCN case, our implementation is consistently faster than pure PyG implementations on the smallest dataset and small data sizes.

In all instances except one (Flickr, 128 hidden size), our performance surpasses that of the dGNN implementation. We attribute this trend to two key factors. Firstly, we cache full edge weights, whereas dGNN saves only the node attention and recomputes the values. Secondly, the smaller the dataset and the hidden size, the more dGNN is outperformed by our implementation and vanilla PyG. This highlights the suboptimal handling of smaller computations by dGNN, stemming from inadequate utilization of shared memory resources and a lack of adaptability in kernel-blocking strategies for small data. A closer inspection of the source code further substantiates these observations [1].

However, in some cases our implementation does not outperform PyTorch. That is caused by the fact that our implementation, by committing to highly-optimized subroutines, gives up on possible fusion with sparse operators, which is leveraged by PyTorch, similarly to the case of GCN. Moreover, in multi-head GAT layers, we execute some computation only as batched in the leading dimension,



which requires us to execute additional tensor permutations. It would be more efficient to omit the tensor permutations and execute the computation batched in the middle dimension, which would allow for more coalesced memory accesses. However, we take the former approach due to lack of support for the appropriate batching scheme in CuSPARSE. Future work could overcome this difficulty and produce a more optimized GAT implementation.

Furthermore, the experimental results highlight the need for improvements in compilation support within PyTorch Geometric. We encountered difficulties in compiling the GAT model for both Citeseer and OGB Arxiv due to framework bugs in PyTorch. Additionally, for dGNN, we examined results from both compiled and non-compiled models. Surprisingly, we observed that contrary to expectations, the compiled version performs slower than the non-compiled counterpart on the smallest datasets (Cora and Citeseer). This phenomenon is limited to small graphs and hidden sizes. We hypothesize that the PyTorch compiler might generate suboptimal blocking schemes for CUDA kernels in these cases, leading to underutilization of the GPU.

## 5 Related Work

**Optimizing Sparse Operators.** While our work focuses on high-level GNN computation, the problem of optimizing sparse computation on the operator level has been explored by many works. However, approaches presented in the works below focus solely on low-level optimization of SpMM and thus are not easily generalizable to other computations required by GNNs.

Various approaches aim to optimize sparse matrix-vector multiplication (SpMV) in different contexts. Kreutzer et al. [27] propose SELL- $C$ - $\sigma$ , enhancing ELLPACK for efficient SpMV. Anzt et al. [3] present another ELLPACK variant optimized for SpMV. Multiple other works in this direction exist [5, 9, 10, 17, 36]. These solutions offer inspiration rather than direct solutions due to the SpMV absence in GNNs.

Gale et al. [15] optimize SpMM and SDDMM for Deep Learning. Their focus is on matrices for pruned dense neural networks, with sparsity of 70%-90%. Yet, this is significantly lower than typical matrices representing graphs (< 2% non-zero entries).

For GNNs, Shi et al. [34] propose a SpMM-optimized format using modified COO. Vazquez et al. [37] design an optimized SpMM kernel using ELLPACK-R. Both address thread balance, memory latency, and uncoalesced reads. GE-SpMM [22] also optimizes SpMM for GNNs, allowing various reduce operators and operating on widely-used CSR format. Recently, Besta et al. [6] provided a tensor-based formulation for a broad set of Attentional GNNs.

**High-level optimization.** In our work, we utilize the DaCeML [31] and DaCe frameworks to enable high-level optimization beyond the scope of a single operator. There are other works that also adopt such a high-level approach to optimizing the GNN runtime.

Zhang et al. [45] use a layer-level approach similar to ours, wherein they examine computation graphs of selected GNN layers to pinpoint performance issues. Their solution involves reordering operators in the GAT to reduce computation, but they do not address the GCN and lack dynamic computation adjustments based on matrix size.

Methods to alleviate high memory bandwidth usage exist, including *neighbor grouping* by Huang et al. [23]. This approach assigns neighbor node groups to memory-sharing threads, thereby enhancing data reuse. It addresses GNN performance in a distinct manner compared to our work.

GNN-focused compilers also address sparse data optimization. Graphiler [40] automatically enhances GPU-based GNN computation by employing operator reordering optimization, similar to our approach and that of Zhang et al. [45]. In terms of CPU execution, Graphite [18] optimizes GNNs by overlapping memory and compute tasks to improve data locality. However, it is restricted to CPUs. SparseTIR [43] adopts a data-centered perspective, offering composability in terms of formats and transformations. It tailors hybrid data formats and optimizes computation in alignment with its intermediate representation (IR).

## 6 Conclusion


In this work, we presented a unified view on GNN computational graphs, I/O and memory which allowed us to connect a high-level understanding with the awareness of low-level performance consequences. We used the gained insights to propose GNN optimizations, we implemented them and we showed their benefits in terms of performance.

We proposed the adaptive computational scheme with caching for optimizing the chained multiplication of  $\mathbf{A} \cdot \mathbf{X} \cdot \Theta$  where  $\mathbf{A}$  is a sparse adjacency matrix and  $\mathbf{X}$  and  $\Theta$  are dense matrices. This scheme eliminates redundant computation and reuses cached values to save both compute and memory. We incorporated the scheme in the Graph Convolutional Network layer using the DaCe framework and achieved up to 3.02x speedup (geomean 1.31x) in comparison to the best PyTorch implementation in the forward pass and up to 2.43x speedup (geomean 1.27x) in the backward pass. Furthermore, we evaluated GCN runtime on different sparse data formats and found that the choice of a correct format can result in up to 1.63x speedup.

Moreover, we explored alternative caching schemes for GAT and showed an in-depth analysis of the influence of caching on runtime. Our optimized implementation achieved up to 2.71x speedup (geomean 1.11x) in comparison to the best PyTorch Geometric implementation in the forward pass and up to 1.94x speedup (geomean 1.17x) in the backward pass.

Furthermore, this work provides explicit formulations of the backward passes for both GCN and GAT (Appendix G). Leveraging their mathematical properties was crucial in this work. Thus, further work could also benefit from their availability. Another contribution of this work is the extension of DaCeML [31] which allows for replacements of PyTorch modules with arbitrary code. This functionality facilitates further work in optimization of machine learning models previously not supported by DaCeML, such as GNN models.

## Acknowledgements

We thank Hussein Harake, Colin McMurtrie, Mark Klein, Angelo Mangili, and the whole CSCS team granting access to the Ault and Daint machines, and for their excellent technical support. We thank Timo Schneider for help with computing infrastructure at SPCL. This project received funding from the European Research Council  (Project PSAP, No. 101002047), and the European High-Performance Computing Joint Undertaking (JU) under grant agreements No. 955513 (MAELSTROM) and No. 101034126 (EU-Pilot). This project was supported by the ETH Future Computing Laboratory (EFCL), financed by a donation from Huawei Technologies. This project received funding from the European Union’s HE research and innovation programme under the grant agreement No. 101070141 (Project GLACIATION).

## References

- [1] [n. d.]. dgNN: High-performance backend for GNN layers with Data Flow Graph level optimization. <https://github.com/dgSPARSE/dgNN/>. Accessed: 2023-08-11. 8
- [2] Luis M Antunes, Ricardo Grau-Crespo, and Keith T Butler. 2022. Distributed representations of atoms and materials for machine learning. *npj Computational Materials* 8, 1 (2022), 1–9. 1
- [3] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. 2014. *Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-sigma formats on NVIDIA GPUs*. Technical Report UT-EECS-14-727. 9
- [4] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. 2
- [5] Maciej Besta et al. 2020. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In *IEEE IPDPS*. IEEE, 1122–1132. 9
- [6] Maciej Besta et al. 2023. High-Performance and Programmable Attentional Graph Neural Networks with Global Tensor Formulations. In *ACM/IEEE Supercomputing*. 3, 9
- [7] Maciej Besta, Raphael Grob, Cesare Miglioli, Nicola Bernold, Grzegorz Kwasniewski, Gabriel Gjini, Raghavendra Kanakagiri, Saleh Ashkboos, Lukas Gianinazzi, Nikoli Dryden, et al. 2022. Motif Prediction with Graph Neural Networks. In ACM KDD. *arXiv preprint arXiv:2106.00761*. 2
- [8] Maciej Besta and Torsten Hoefler. 2022. Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis. *arXiv preprint arXiv:2205.09702* (2022). 2
- [9] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. Slimsell: A vectorizable graph representation for breadth-first search. In *IEEE IPDPS*. IEEE, 32–41. 9
- [10] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *ACM HPDC*. 9
- [11] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. 2017. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine* 34, 4 (2017), 18–42. 2

- [12] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. 2020. Machine learning on graphs: A model and comprehensive taxonomy. *arXiv preprint arXiv:2005.03675* (2020). 2
- [13] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. <https://doi.org/10.48550/ARXIV.1902.07243> 2
- [14] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2, 13
- [15] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. arXiv:2006.10901 [cs.LG] 9
- [16] Lukas Gianinazzi, Maximilian Fries, Nikoli Dryden, Tal Ben-Nun, and Torsten Hoefler. 2021. Learning Combinatorial Node Labeling Algorithms. *arXiv preprint arXiv:2106.03594* (2021). 2
- [17] Lukas Gianinazzi, Pavel Kalvoda, Alessandro De Palma, Maciej Besta, and Torsten Hoefler. 2018. Communication-avoiding parallel minimum cuts and connected components. In *ACM SIGPLAN Notices*. 9
- [18] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas. 2022. Graphite: Optimizing Graph Neural Networks on CPUs through Cooperative Software-Hardware Techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (New York, New York) (ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 916–931. <https://doi.org/10.1145/3470496.3527403> 9
- [19] William L Hamilton et al. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* (2017). 2
- [20] William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216 [cs.SI] 4
- [21] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, Jure Leskovec, Regina Barzilay, Peter Battaglia, Yoshua Bengio, Michael Bronstein, Stephan Günnemann, Will Hamilton, Tommi Jaakkola, Stefanie Jegelka, Maximilian Nickel, Chris Re, Le Song, Jian Tang, Max Welling, and Rich Zemel. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *Advances in Neural Information Processing Systems 2020-December* (5 2020). <https://arxiv.org/abs/2005.00687v7> 6
- [22] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. arXiv:2007.03179 [cs.DC] 9
- [23] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Virtual Event, Republic of Korea) (PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 119–132. <https://doi.org/10.1145/3437801.3441585> 9
- [24] J. Kiefer and J. Wolfowitz. 1952. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics* 23, 3 (1952), 462 – 466. <https://doi.org/10.1214/aoms/1177729392> 21
- [25] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] 21
- [26] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016). 2
- [27] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing* 36, 5 (jan 2014), C401–C423. <https://doi.org/10.1137/130930352> 9
- [28] Andrew L. Maas. 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models. <https://api.semanticscholar.org/CorpusID:16489696> 20
- [29] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. CUSPARSE library: A set of basic linear algebra subroutines for sparse matrice. In *GPU Technology Conference*. 3, 5, 17, 18
- [30] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. 2017. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. <https://doi.org/10.48550/ARXIV.1706.02413> 1
- [31] Oliver Rausch, Tal Ben-Nun, Nikoli Dryden, Andrei Ivanov, Shigang Li, and Torsten Hoefler. 2022. DaCeML: A Data-Centric Optimization Framework for Machine Learning. In *Proceedings of the 36th ACM International Conference on Supercomputing (ICS '22)*. 9, 10
- [32] Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. 2019. SuperGlue: Learning Feature Matching with Graph Neural Networks. *CoRR* abs/1911.11763 (2019). arXiv:1911.11763 <http://arxiv.org/abs/1911.11763> 2

- [33] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80. 2
- [34] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. 2020. Efficient Sparse-Dense Matrix-Matrix Multiplication on GPUs Using the Customized Sparse Storage Format. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. 19–26. <https://doi.org/10.1109/ICPADS51040.2020.000133>, 9, 17
- [35] Weijing Shi and Ragunathan (Raj) Rajkumar. 2020. Point-GNN: Graph Neural Network for 3D Object Detection in a Point Cloud. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2
- [36] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *ACM/IEEE Supercomputing*. 9
- [37] Francisco Vázquez, Gloria Ortega López, José-Jesús Fernández, Inmaculada García, and Ester M. Garzón. 2012. Fast Sparse Matrix Matrix Product Based on ELLR-T and GPU Computing. *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications* (2012), 669–674. 3, 9, 17
- [38] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017). 2
- [39] Zonghan Wu et al. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020). 2
- [40] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 515–528. <https://proceedings.mlsys.org/paper/2022/file/a87ff679a2f3e71d9181a67b7542122c-Paper.pdf> 2, 9
- [41] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks? *arXiv:1810.00826 [cs.LG]* 4
- [42] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. 2016. Revisiting Semi-Supervised Learning with Graph Embeddings. *CoRR* abs/1603.08861 (2016). *arXiv:1603.08861* <http://arxiv.org/abs/1603.08861> 6
- [43] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. 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* (Vancouver, BC, Canada) (*ASPLOS 2023*). Association for Computing Machinery, New York, NY, USA, 660–678. <https://doi.org/10.1145/3582016.3582047> 9, 16
- [44] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. *arXiv:1907.04931 [cs.LG]* 6
- [45] Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. 2021. Understanding GNN Computational Graph: A Coordinated Computation, IO, and Memory Perspective. *arXiv:2110.09524 [cs.LG]* 2, 5, 9, 13
- [46] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2020. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2020). 2
- [47] Jie Zhou et al. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81. 2

## Appendix

### A No memory overhead with GCN caching

Usually, caching additional values for the backward pass means using additional memory. However, that is not the case when employing caching in GCN. Looking at Figure 12, it can be seen that in order to compute the gradients without caching, in both schemes we need to store the layer input features of size  $n \times m$  for the backward pass. Nevertheless, as can be seen from Figure 13b, once the cached aggregated features are used, the original layer input features are not needed. Thus, the amount of memory used to store intermediates for the backward pass is exactly the same as in the scheme without caching.

## B GAT Operator Reordering

For each edge, we need to compute the raw attention weight  $\mathbf{a}^T[\mathbf{x}_i\Theta \parallel \mathbf{x}_j\Theta]$  where  $\mathbf{a} \in \mathbb{R}^{2k}$ ,  $x_i\Theta \in \mathbb{R}^k$ . If we were to compute that expression directly for each edge, the computational complexity would be  $\mathcal{O}(qk)$  where  $q$  is the number of edges.

However, there exists a much more efficient scheme that is widely employed by other works [14, 45] and mainstream GNN frameworks. Instead of computing the whole expression for each edge, we precompute values for each node and only sum them up for each edge.

Firstly, we split the attention weights into two separate vectors:

$$\mathbf{a} = \begin{bmatrix} \mathbf{a}_{\text{src}} \\ \mathbf{a}_{\text{dst}} \end{bmatrix}$$

where  $\mathbf{a}_{\text{src}}, \mathbf{a}_{\text{dst}} \in \mathbb{R}^k$ . This allows us to first compute  $\mathbf{Y} = \mathbf{X}\Theta$  and then for each node  $i$  calculate  $\alpha_i^{\text{src}} = \mathbf{a}_{\text{src}}^T \mathbf{Y}_i$  and  $\alpha_i^{\text{dst}} = \mathbf{a}_{\text{dst}}^T \mathbf{Y}_i$ . Then for each edge between nodes  $i$  and  $j$  we only need to sum  $\alpha_i^{\text{src}} + \alpha_j^{\text{dst}}$ . This way of computing the weights results in computational complexity of  $\mathcal{O}(q + nk)$  which is much more beneficial, given that  $n \ll q$  in graph data.

## C GAT Caching

Similar to GCN, in GAT we can cache intermediate values to avoid recomputing them in the backward pass. As can be seen in Figure 8, there are multiple stages at which we can cache intermediate values. In Figure 9 the scheme for computing the backward pass can be seen.

## D Sparse Formats

Let us assume a matrix  $\mathbf{A} \in \mathbb{R}^{n \times m}$ . All of the formats mentioned below are illustrated in Figure 10.

### D.1 Compressed Sparse Row and Compressed Sparse Column

The Compressed Sparse Row format (CSR) is one of the most commonly used sparse matrix formats. CSR requires storing two vectors describing the sparse structure: the columns array  $\mathbf{c} \in \mathbb{Z}^q$  and the row pointer array  $\mathbf{r} \in \mathbb{Z}^{n+1}$ . The  $i$ -th position in  $\mathbf{r}$  indicates at which index of  $\mathbf{c}$  the  $i$ -th row start. The  $(n + 1)$ -th position of  $\mathbf{r}$  contains the value  $q + 1$  to simplify array processing. The  $j$ -th position in  $\mathbf{c}$  indicates the column of the  $j$ -th value in the dense representation of the matrix. Optionally, a third array can be used to store edge weights: the values array  $\mathbf{v} \in \mathbb{R}^q$ , which is indexed the same way as  $\mathbf{c}$ . In total, this format requires the storage of  $q + n + 1$  integer values for indexing and  $q$  floating point values to represent the non-zero matrix entries. Thus, among the formats described here, the CSR format requires the least memory to store a given sparse matrix.

The CSR format is not particularly well suited to processing on GPUs because it enforces row-wise processing of the matrix but the rows are not even. Thus, the work division between CUDA threads is not straightforward and requires dynamic load balancing.

The Compressed Sparse Column format (CSC) is analogous to the CSR format. Instead of the arrays representing columns and row pointers, it uses one array for rows and one for column pointers.

### D.2 Coordinate Format

The coordinate format (COO) is the simplest sparse matrix format. To represent a matrix in this format, two arrays are needed: the columns array  $\mathbf{c} \in \mathbb{Z}^q$  and the rows array  $\mathbf{r} \in \mathbb{Z}^q$ , that together indicate the coordinates of a given entry in the original dense matrix. Similarly to CSR, a third array storing values can be also used: the values array  $\mathbf{v} \in \mathbb{R}^q$ .

This format has a higher memory requirement than CSR and CSC. It requires storing three matrices of length  $q$  each (two holding integer values and one holding floating point values). However, it allows for a simpler parallelization. The matrix can be processed entry-by-entry, so no tailored scheme is needed to ensure even work division between threads.

### D.3 ELLPACK Format

The ELLPACK format is better suited for processing on GPUs than CSR and COO. Instead of one-dimensional vectors, it uses a two-dimensional matrix to store the sparsity structure:  $\mathbf{Col} = (c_{ij}) \in \mathbb{Z}^{n \times p}$ , where  $c_{ij}$  holds the column index of the  $j$ -th entry in the  $i$ -th row. The number of columns in  $\mathbf{Col}$ , which we denote by  $p$ , is equal to the highest number of non-zero entries in a single

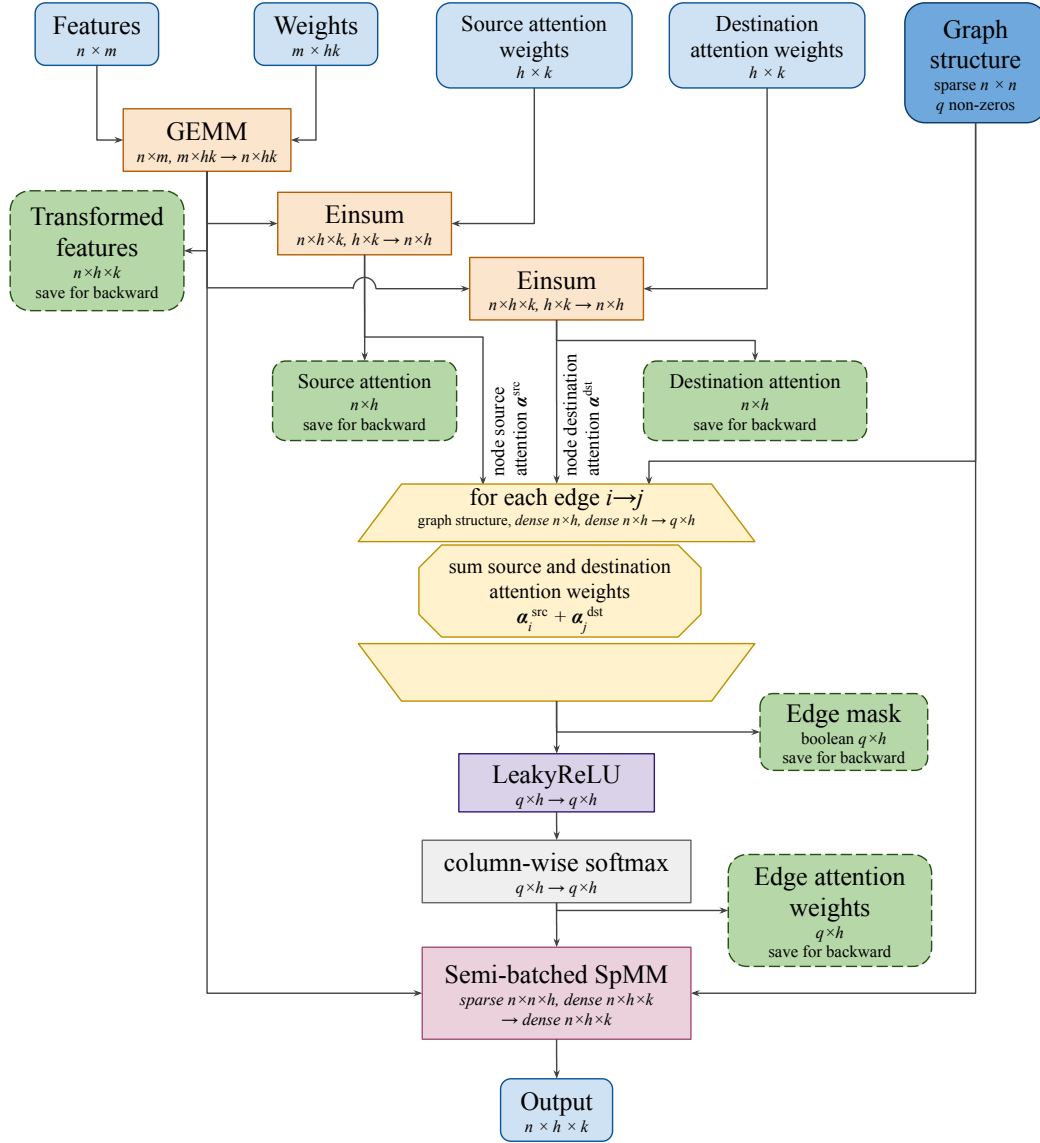
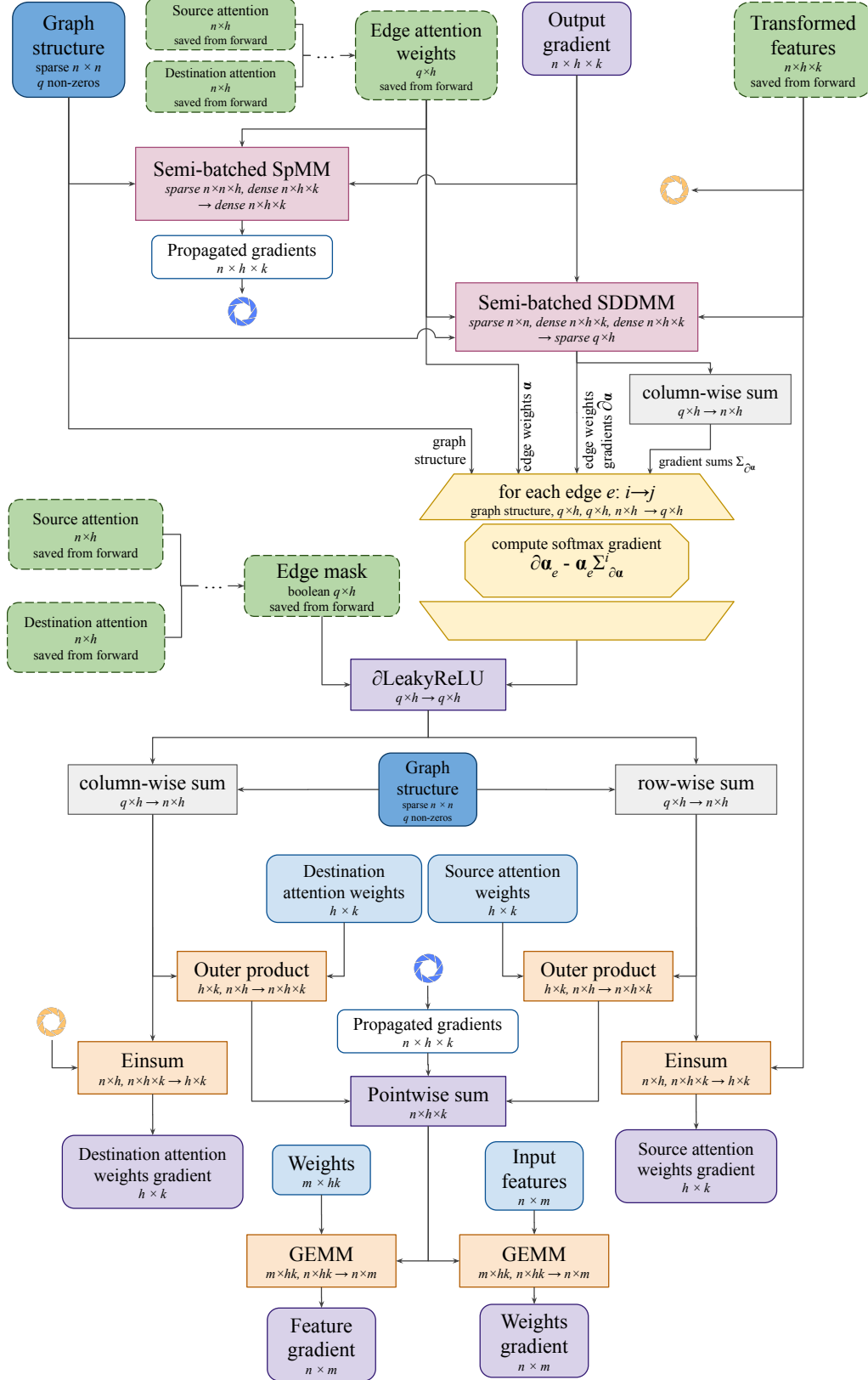
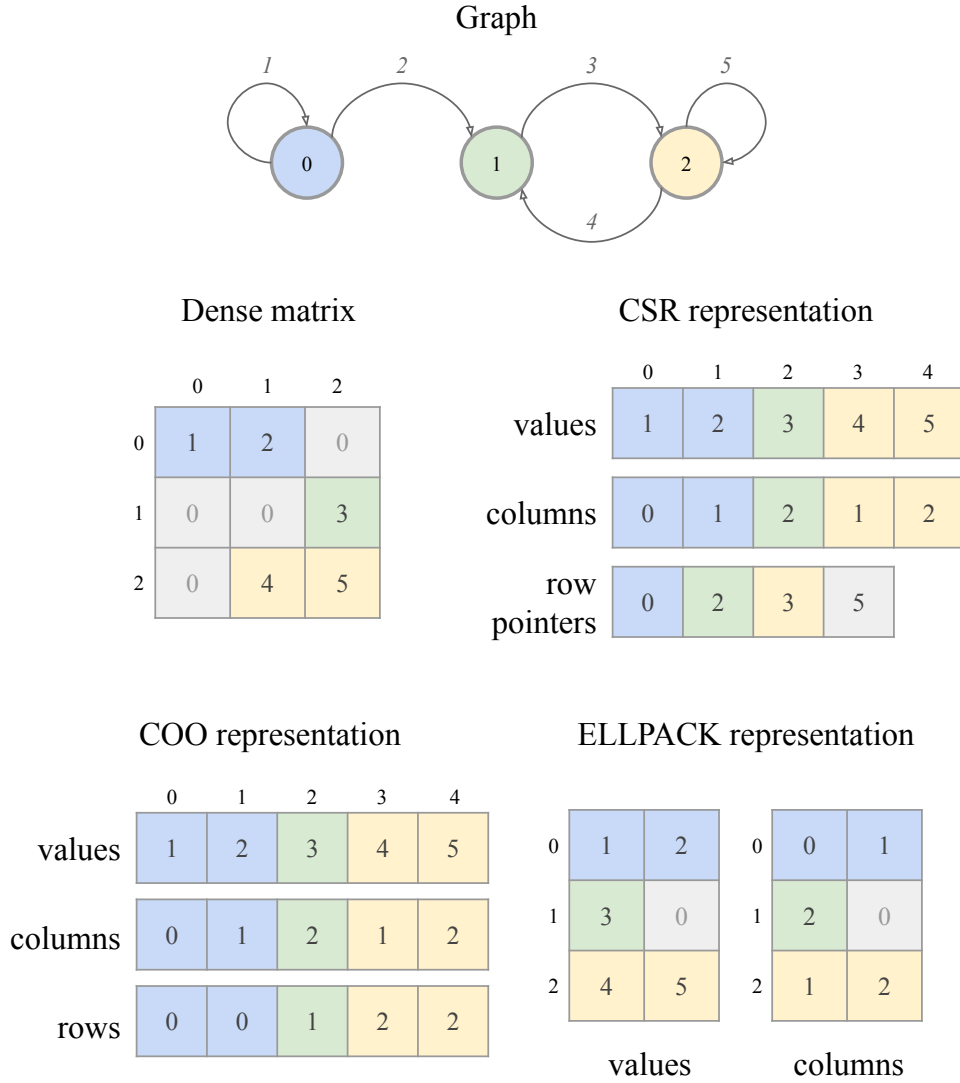


Figure 8: GAT forward scheme. Green boxes with dashed frames indicate values that can be cached for the backward pass.



**Figure 9:** GAT backward pass using cached values. Source and destination attention can be saved instead of edge mask and edge attention weights to use less memory. The scheme for recomputation of values from the forward pass is omitted for clarity.



**Figure 10:** Example graph and its representations in selected formats.

row of the matrix. Similarly, to store the non-zero matrix values, a two-dimensional matrix is used: the values matrix  $\mathbf{V} = (v_{ij}) \in \mathbb{R}^{n \times p}$ , where  $v_{ij}$  stores the value of the  $j$ -th entry in the  $i$ -th row.

In ELLPACK, a more regular representation is obtained at the cost of storing unnecessary data. This format requires storing  $np$  integer values for indexing and another  $np$  floating point values. Thus, the ELLPACK format is well-suited to processing of matrices that have non-zero values evenly distributed across the rows. Otherwise, a single row that is much longer than the others leads to unnecessarily high storage needs. Due to this, ELLPACK is not well suited for certain graphs, such as graphs with hubs or other subgraphs that imply high variance in node degree. However, the regular structure of the ELLPACK representation makes it easy to process in parallel on GPUs.

#### D.4 Hybrid Formats

As an attempt to mitigate the disadvantages stemming from the use of some of the formats, a hybrid format can be used. There are multiple variants, such as hybrid formats used by Ye et al. [43]. As an example a CSR-COO data format could be constructed to mitigate the issues from irregular row lengths in CSR. In this format, only the first  $t$  elements in a given row are stored in the CSR format, while the remaining entries are stored in the COO format.



Data format	Variables	FLOPs	I/O [bytes]	Operational intensity $\left[\frac{\text{FLOP}}{\text{byte}}\right]$	
				Cora	OGB Arxiv
CSR	$\mathbf{v} \in \mathbb{R}^q$ $\mathbf{r}_{ptr} \in \mathbb{R}^{n+1}$ $\mathbf{c} \in \mathbb{R}^q$ $\mathbf{B} \in \mathbb{R}^{m \times f}$	$\mathcal{O}(qf)$	$\mathcal{O}(q + mf + nf)$	0.621	1.066
CSC	$\mathbf{v} \in \mathbb{R}^q$ $\mathbf{c}_{ptr} \in \mathbb{R}^{m+1}$ $\mathbf{r} \in \mathbb{R}^q$ $\mathbf{B} \in \mathbb{R}^{m \times f}$	$\mathcal{O}(qf)$	$\mathcal{O}(q + mf + nf)$	0.621	1.066
COO	$\mathbf{v} \in \mathbb{R}^q$ $\mathbf{r} \in \mathbb{R}^q$ $\mathbf{c} \in \mathbb{R}^q$ $\mathbf{B} \in \mathbb{R}^{m \times f}$	$\mathcal{O}(qf)$	$\mathcal{O}(q + mf + nf)$	0.612	1.036
ELLPACK	$\mathbf{V} \in \mathbb{R}^{n \times p}$ $\mathbf{Col} \in \mathbb{R}^{n \times p}$ $\mathbf{B} \in \mathbb{R}^{m \times f}$	$\mathcal{O}(qf)$	$\mathcal{O}(np + mf + nf)$	0.236	0.207

**Table 5:** Operational intensity of SpMM  $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ . Example operational intensity values were computed for scenarios realistic in GNNs, i.e., where  $\mathbf{A}$  is the adjacency matrix of a given graph and  $\mathbf{B}$  has  $n$  rows and 64 columns. Detailed information on the graphs can be found in Table 4. To count FLOPs for ELLPACK, we considered only operations on non-zero values.

## E Common Sparse Operators

GNNs are characterized by a substantial amount of sparse computation. There are two sparse operators that are particularly common in GNNs: sparse matrix-matrix multiplication and sampled dense-dense matrix multiplication.

### E.1 Sparse Matrix-Matrix Multiplication

The *sparse matrix-matrix multiplication* operator (SpMM) is a multiplication of a sparse matrix  $\mathbf{A} \in \mathbb{R}^{n \times m}$  and a dense matrix  $\mathbf{B} \in \mathbb{R}^{m \times k}$ , resulting in a dense matrix  $\mathbf{C} \in \mathbb{R}^{n \times k}$ ,  $\mathbf{C} = \mathbf{A}\mathbf{B}$ . There exist highly optimized implementations of this operator [29] and it is widely explored in literature [34, 37] (see more in Section 5). An example implementation using the CSR format can be seen in Listing 1.

```

1 def spmm(A_rowptrs, A_columns, A_values, B):
2     M, K = B.shape
3     N = A_rowptrs.shape[0] - 1
4     C = np.empty((N, K))
5     for i in range(N):
6         for j in range(A_rowptrs[i], A_rowptrs[i + 1]):
7             column = A_columns[j]
8             for k in range(K):
9                 C[i, k] += B[column, k] * A_values[j]
10    return C
    
```

**Algorithm 1:** Example SpMM implementation in NumPy using the CSR format.

The SpMM operator is a very common operator in GNNs. It represents the propagation of information between nodes. To understand the computational characteristics of SpMM, we look at *operational intensity*, defined as the number of floating point operations executed by the program per byte of I/O. Operational intensity for SpMMs executed in GNNs can be found in Table 5. It can be seen that the operational intensity is no higher than  $1.066 \frac{\text{FLOP}}{\text{byte}}$  which indicates that the computation is memory-bound. Operational intensities of CSC-SpMM and CSR-SpMM are the same because the sparse matrix is square. Moreover, it is worth noting that ELLPACK is well-suited to represent typical GNN dataset such as Cora and Arxiv. They both have nodes with much higher degrees than average: for Cora, the maximum node degree is 168 and the average degree is 7.8, while for Arxiv the maximum degree is 436 and the average degree is 13.77. Therefore, their ELLPACK value matrices consist in  $> 96\%$  of zeros, resulting in significantly lower operational intensity when executing SpMM in comparison to other mentioned formats.

### E.2 Sampled Dense-Dense Matrix Multiplication

Another operator that can be encountered in GNN computation is the *sampled dense-dense matrix multiplication* (SDDMM). Given a sparse matrix  $\mathbf{A} \in \mathbb{R}^{n \times k}$  and two dense matrices  $\mathbf{B} \in \mathbb{R}^{n \times m}$ ,

Data format	Variables	FLOPs	I/O [bytes]	Operational intensity $\left[\frac{\text{FLOP}}{\text{byte}}\right]$		
				Cora	OGB	Arxiv
CSR	$\mathbf{v} \in \mathbb{R}^q$ $\mathbf{r}_{ptr} \in \mathbb{R}^{n+1}$ $\mathbf{c} \in \mathbb{R}^q$ $\mathbf{B} \in \mathbb{R}^{n \times f}$ $\mathbf{C} \in \mathbb{R}^{f \times n}$	$\mathcal{O}(qf)$	$\mathcal{O}(qf + n)$	0.567		0.620
CSC	$\mathbf{v} \in \mathbb{R}^q$ $\mathbf{c}_{ptr} \in \mathbb{R}^{m+1}$ $\mathbf{r} \in \mathbb{R}^q$ $\mathbf{B} \in \mathbb{R}^{n \times f}$ $\mathbf{C} \in \mathbb{R}^{f \times n}$	$\mathcal{O}(qf)$	$\mathcal{O}(qf + n)$	0.567		0.613
COO	$\mathbf{v} \in \mathbb{R}^q$ $\mathbf{r} \in \mathbb{R}^q$ $\mathbf{c} \in \mathbb{R}^q$ $\mathbf{B} \in \mathbb{R}^{n \times f}$ $\mathbf{C} \in \mathbb{R}^{f \times n}$	$\mathcal{O}(qf)$	$\mathcal{O}(qf)$	0.562		0.613
ELLPACK	$\mathbf{V} \in \mathbb{R}^{n \times p}$ $\mathbf{Col} \in \mathbb{R}^{n \times p}$ $\mathbf{B} \in \mathbb{R}^{n \times f}$ $\mathbf{C} \in \mathbb{R}^{f \times n}$	$\mathcal{O}(qf)$	$\mathcal{O}(npf)$	0.308		0.325

**Table 6:** Operational intensity of SDDMM  $\mathbf{D} = \mathbf{A} \odot (\mathbf{B}\mathbf{C})$ . Example operational intensity values were computed for scenarios realistic in GNNs, i.e., where  $\mathbf{A}$  is the adjacency matrix of a given graph and  $\mathbf{B}, \mathbf{C}$  have  $n$  rows and 64 columns. Detailed information on the graphs can be found in Table 4. Counting the FLOPs for ELLPACK, we consider only operations on non-zero values.

$\mathbf{C} \in \mathbb{R}^{m \times k}$ , we can compute  $\mathbf{D} = \mathbf{A} \odot (\mathbf{B} \cdot \mathbf{C})$ , where  $\odot$  represents the Hadamard product and  $\mathbf{D} \in \mathbb{R}^{n \times k}$  is a sparse matrix. Similarly to SpMM, this subroutine has existing highly optimized implementations [29]. An example implementation of SDDMM can be found in Listing 2.

The SDDMM operator is used in the backward pass of GAT. There, we always need to compute it on matrices which have shapes  $\mathbf{A} \in \mathbb{R}^{n \times n}$ ,  $\mathbf{B} \in \mathbb{R}^{n \times f}$  and  $\mathbf{C} \in \mathbb{R}^{f \times n}$ . In Table 6 we present estimated operational intensity of SDDMM when used in GAT backward pass computation.

```

1 def sddmm(A_rowptrs, A_columns, A_values, B, C):
2     M, K = B.shape
3     N = A_rowptrs.shape[0] - 1
4     nnz = A_values.shape[0]
5
6     D_values = np.zeros((nnz,))
7     for i in range(N):
8         for j in range(A_rowptrs[i], A_rowptrs[i + 1]):
9             column = A_columns[j]
10            D_values[j] = A_values[j] * np.dot(B[i], C[column])
11     return D_values

```

**Algorithm 2:** Example SDDMM implementation in NumPy using the CSR format.

Similarly to SpMM, SDDMM is an operator with low operational intensity. For both Cora and Arxiv, the operational intensity is below 0.7, independent of the data format. Therefore, SDDMM is also memory-bound. Performing the computation using the ELLPACK format leads to even lower operational intensity due to irregular row sizes in the sparse matrices, same as in the case of SpMM.

## F GCN Evaluation

The results for schemes without caching are presented in Figure 11. It can be seen that the runtime patterns follow our expectations from Table 1. In the case of not computing input feature gradients, as long as the number of input features (128) is bigger than that of the output features, the transform-first forward and fused-propagate backward pass scheme is faster. However, once the number of features surpasses 128, the alternative scheme is faster. The same applies to the case with computing input gradients, but the threshold is shifted to 256. Our adaptive implementation is always as fast as the faster scheme.

The results for schemes with caching are presented in Figure 14. It can be seen that the runtime distribution follows what we expected in Table 2. When not computing input gradients, the scheme using caching is faster once the number of output features exceeds half of the input features. When calculating input gradients, the threshold for adapting the scheme lies at equal input and output feature sizes, which is 128. Again, our adaptive computing scheme proves to match the fastest scheme in every case except the threshold point.

The single suboptimal choice of the scheme at the hidden size of 128 in Figure 14b is due to the fact that we assumed that running two SpMMs matrices matrices of sizes  $n \times n$  and  $n \times t$ , where  $t \in \mathbb{Z}^+$  would take the same time as running one SpMM multiplying matrices of sizes  $n \times n$  and  $n \times 2t$  (limitations of our approach are described in Section 3.1).

## G Implementations

### G.1 GCN Operator Implementation

```

1 def gcn(node_features, rowptrs, columns, edge_vals, weights, output):
2     # GEMM
3     new_features = node_features @ weights
4
5     # SpMM
6     output[:] = 0
7     for i, k in dace.map[0:N, 0:num_out_features]:
8         for j in dace.map[rowptrs[i]:rowptrs[i + 1]]:
9             column = columns[j]
10            mult = new_features[i, k] * vals[j]
11            output[column, k] += mult

```

Algorithm 3: Example GCN operator implementation using NumPy.

### G.2 GCN Forward Pass

More specifically, the GCN operator takes as input an *adjacency matrix*  $\mathbf{A} \in \mathbb{R}^{n \times n}$  and *node features*  $\mathbf{X} \in \mathbb{R}^{n \times m}$ .

The resulting node features  $\mathbf{X}' \in \mathbb{R}^{n \times k}$  are computed as follows:

$$\mathbf{X}' = \mathbf{D}^{-1/2} \mathbf{A} \mathbf{D}^{-1/2} \mathbf{X} \Theta + \mathbb{1}_n \mathbf{b}^T \tag{1}$$

where  $\mathbf{D} \in \mathbb{R}^{n \times n}$  is the diagonal degree matrix,  $\Theta \in \mathbb{R}^{m \times k}$  is the learned parameter matrix,  $\mathbf{b} \in \mathbb{R}^k$  is the learned bias vector, and  $\mathbb{1}_n$  is an  $n$ -element vector of ones.

Optionally, edge weights can be included as entries other than 1 in the adjacency matrix. Often, self-loops are also inserted, so a modified adjacency matrix is used:  $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ . As the graphs are constant during training, the data can be preprocessed as following:

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

where  $\hat{\mathbf{D}}$  is the diagonal degree matrix for a graph with added self-loops. Then, the GCN formulation becomes:

$$\mathbf{X}' = \mathbf{A}' \mathbf{X} \Theta + \mathbb{1}_n \mathbf{b}^T \tag{3}$$

where  $\mathbf{A}'$  is a sparse matrix, while all others are dense.

Above formulation assumes the use of sum as the message aggregation function, so the message passing and aggregation are represented by the matrix multiplication  $\mathbf{A} \cdot \mathbf{X} \Theta$ . Some models alternatively use mean as the aggregation function, which can be represented by  $\mathbf{D} \cdot \mathbf{A} \cdot \mathbf{X} \Theta$ . Another alternative aggregation function is max, which can be represented in the node-wise formulation as following:

$$x_{ik} = \max_{j \in \mathcal{N}(i)} (\mathbf{x}_j \Theta_k^T)$$

Throughout this work we focus on networks with the aggregation function sum.

### G.3 GAT Forward Pass

The GAT operator can be computed as follows:

$$\mathbf{X}' = \mathbf{A} \mathbf{X} \Theta + \mathbb{1}_n \mathbf{b}^T, \tag{4}$$

where  $\mathcal{A} = (\alpha_{ij})$  is the sparse attention weight matrix,  $\Theta \in \mathbb{R}^{m \times k}$  is the learned parameter matrix,  $\mathbf{b} \in \mathbb{R}^k$  is the learned bias vector.

The attention weight of the edge from node  $i$  to node  $j$  is defined as:

$$\alpha_{ij} = \frac{\exp(f(\mathbf{a}^T[\Theta^T \mathbf{x}_i \parallel \Theta^T \mathbf{x}_j]))}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp(f(\mathbf{a}^T[\Theta^T \mathbf{x}_i \parallel \Theta^T \mathbf{x}_k]))}, \quad (5)$$

where  $\mathbf{X} \in \mathbb{R}^{n \times m}$  is the input node features matrix,  $\parallel$  is the concatenation operator,  $\mathbf{a} \in \mathbb{R}^{2k}$  are the learned attention parameters and  $f$  is the Leaky Rectified Linear Unit (Leaky ReLU) [28] with a negative slope parameter of  $\beta \in \mathbb{R}^+$ . It is worth noting that the expression  $\Theta \mathbf{x}_i$  is actually the  $i$ -th row of the matrix  $\mathbf{X}\Theta$ .

It is also possible for the GAT operator to have multiple *attention heads*. Let us denote the number of heads by  $h$ . Using  $h$  attention heads is mathematically equivalent to concatenating results of  $h$  GAT operators with different learned parameters.

Thus, the learned parameters of the layers are:

$$\Theta = [\Theta_1 \dots \Theta_h], \quad (6)$$

$$\mathbf{b} = [\mathbf{b}_1^T \dots \mathbf{b}_h^T]^T, \quad (7)$$

where  $\Theta_q \in \mathbb{R}^{m \times k}$  denotes the weight matrix for the  $q$ -th head and  $\mathbf{b}_q \in \mathbb{R}^k$  represents the bias for  $q$ -th head.

The formulation for the output feature matrix of the  $q$ -th head is as follows:

$$\mathbf{X}'_q = \mathcal{A}_q \mathbf{X} \Theta_q + \not\llcorner_n \mathbf{b}_q^T, \quad (8)$$

where  $q$  indicates the head index,  $\mathbf{X}'_q \in \mathbb{R}^{n \times k}$  is output feature matrix of the  $q$ -th head. The matrix  $\mathcal{A}_q = (\alpha_{qij}) \in \mathbb{R}^{n \times n}$  represents attention weights of the  $q$ -th head which are computed as follows:

$$\alpha_{i,j}^q = \frac{\exp(f(\mathbf{a}_q^T[\Theta_q^T \mathbf{x}_i \parallel \Theta_q^T \mathbf{x}_j]))}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp(f(\mathbf{a}_q^T[\Theta_q^T \mathbf{x}_i \parallel \Theta_q^T \mathbf{x}_k]))}, \quad (9)$$

where  $\mathbf{a}_q \in \mathbb{R}^{2k}$  are the learned attention parameters for the  $q$ -th head.

In the end, outputs for all heads are concatenated to create a single output matrix  $\mathbf{X}' \in \mathbb{R}^{n \times hk}$ :

$$\mathbf{X}' = [\mathbf{X}'_1 \dots \mathbf{X}'_h]. \quad (10)$$

#### G.4 GCN Backward Pass

In order to compute weight updates, we need to compute the gradients of the loss function  $\mathcal{L}$  with respect to the parameters of the GCN layer, namely  $\Theta$  and  $\mathbf{b}$ , as well as the gradients with respect to the input node features  $\mathbf{X}$ , in order to propagate the gradients to the previous layers.

In the derivations, we use the following property. Given some matrices  $\mathbf{A} \in \mathbb{R}^{a \times b}$ ,  $\mathbf{B} \in \mathbb{R}^{b \times c}$  and operation  $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ , and the gradient of some loss function  $\frac{\partial \mathcal{L}}{\partial \mathbf{C}}$ , we can compute the following gradients:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{A}} &= \frac{\partial \mathcal{L}}{\partial \mathbf{C}} \mathbf{B}^T \\ \frac{\partial \mathcal{L}}{\partial \mathbf{B}} &= \mathbf{A}^T \frac{\partial \mathcal{L}}{\partial \mathbf{C}} \end{aligned}$$

Let  $\frac{\partial \mathcal{L}}{\partial \mathbf{X}'} \in \mathbb{R}^{n \times k}$  denote the gradient of the loss function with respect to the layer output  $\mathbf{X}'$ . Then we can express the gradient with respect to the parameters  $\Theta$  and  $\mathbf{b}$  as follows:

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \mathbf{X}^T \mathbf{A}'^T \frac{\partial \mathcal{L}}{\partial \mathbf{X}'}, \quad (11)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial \mathbf{X}'}^T \not\llcorner_n. \quad (12)$$

In case of all layers except the first one, the gradients need to be propagated backward through the model. Thus, we need to also compute the gradient of the loss function with respect to the input node features  $\mathbf{X}$ . The gradient  $\frac{\partial \mathcal{L}}{\partial \mathbf{X}} \in \mathbb{R}^{n \times m}$  can be computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \mathbf{A}'^T \frac{\partial \mathcal{L}}{\partial \mathbf{X}'} \Theta^T. \quad (13)$$

It is worth noting that matrix product  $\mathbf{A}'^T \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{X}'}$  can be interpreted as propagating the gradients backward through the graph, hence the adjacency matrix is transposed in the backward pass.

Once we have computed the gradients  $\frac{\partial \mathcal{L}}{\partial \Theta}$ ,  $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$ , and  $\frac{\partial \mathcal{L}}{\partial \mathbf{X}}$ , we can use them to perform a parameter update step using a selected optimizer, such as Stochastic Gradient Descent [24] or Adam [25].

#### G.4.1 GAT Backward Pass

In order to perform a gradient update step, we need to compute gradients of the loss function w. r. t. learned parameters  $\Theta$ ,  $\mathbf{a}$ ,  $\mathbf{b}$  and the input features  $\mathbf{X}$ , respectively denoted by  $\frac{\partial \mathcal{L}}{\partial \Theta}$ ,  $\frac{\partial \mathcal{L}}{\partial \mathbf{a}}$ ,  $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$  and  $\frac{\partial \mathcal{L}}{\partial \mathbf{X}}$ .

Let us denote  $\mathbf{X}\Theta = \mathbf{M} \in \mathbb{R}^{n \times k}$ , the  $i$ -th row of  $\mathbf{M}$  as  $\mathbf{m}_i$ , and  $\mathbf{a} = \begin{bmatrix} \mathbf{a}_{\text{src}} \\ \mathbf{a}_{\text{dst}} \end{bmatrix}$ , where  $\mathbf{a}_{\text{src}}, \mathbf{a}_{\text{dst}} \in \mathbb{R}^k$ .

For clarity, let us denote the intermediate values  $w_{ij}$ ,  $y_{ij}$ ,  $s_i = \mathbf{a}_{\text{src}} \mathbf{m}_i^T$ , and  $d_j = \mathbf{a}_{\text{dst}} \mathbf{m}_j^T$  in the computation of attention weights:

$$f(\mathbf{a}^\top [\Theta \mathbf{x}_i \parallel \Theta \mathbf{x}_j]) = f(\mathbf{a}_{\text{src}} \mathbf{m}_i^T + \mathbf{a}_{\text{dst}} \mathbf{m}_j^T) = f(s_i + d_j) = f(y_{ij}) = w_{ij} \quad (14)$$

The gradient for  $\mathbf{b}$  is computed in the same way as in case of GCN:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial \mathbf{X}'} \mathbb{K}_n. \quad (15)$$

Then, using  $\mathbf{M} = \mathbf{X}\Theta$ , the gradients for  $\mathbf{X}$  and  $\Theta$  can be computed as follows:

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \mathbf{X}^T \frac{\partial \mathcal{L}}{\partial \mathbf{M}}, \quad (16)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \frac{\partial \mathcal{L}}{\partial \mathbf{M}} \Theta, \quad (17)$$

where  $\frac{\partial \mathcal{L}}{\partial \mathbf{M}}$  is the gradient w. r. t.  $\mathbf{M}$ . In order to compute  $\frac{\partial \mathcal{L}}{\partial \mathbf{M}}$ ,  $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_{\text{src}}}$ ,  $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_{\text{dst}}}$  we need to compute multiple intermediate gradients.

Firstly, the gradient w. r. t. the attention weights, given that  $\mathbf{X}' = \mathcal{A}\mathbf{M}$ :

$$\frac{\partial \mathcal{L}}{\partial \mathcal{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{X}'} \mathbf{M}^T \quad (18)$$

Then, we need to backpropagate through the column-wise softmax.

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \alpha_{ij} \left( \frac{\partial \mathcal{L}}{\partial \alpha_{ij}} - \sum_{u=1}^n \alpha_{iu} \frac{\partial \mathcal{L}}{\partial \alpha_{iu}} \right) \quad (19)$$

The next step is computing the derivative of Leaky ReLU. Its derivative is:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0, \\ \beta & \text{if } x \leq 0. \end{cases} \quad (20)$$

Then the gradient is as follows from the chain rule.

$$\frac{\partial \mathcal{L}}{\partial y_{ij}} = f'(y_{ij}) \cdot \frac{\partial \mathcal{L}}{\partial w_{ij}} \quad (21)$$

Now we just need to compute the gradients for  $\mathbf{s} = \mathbf{a}_{\text{src}} \mathbf{M}^T$  and  $\mathbf{d} = \mathbf{a}_{\text{dst}} \mathbf{M}^T$ . Given that  $y_{ij} = s_i + d_j$ , computing the gradients just requires a summation along an appropriate axis.

$$\frac{\partial \mathcal{L}}{\partial s_i} = \sum_{u=1}^n \frac{\partial \mathcal{L}}{\partial y_{iu}}, \quad \frac{\partial \mathcal{L}}{\partial d_j} = \sum_{u=1}^n \frac{\partial \mathcal{L}}{\partial y_{uj}} \quad (22)$$

Now we can use  $\frac{\partial \mathcal{L}}{\partial \mathbf{d}}, \frac{\partial \mathcal{L}}{\partial \mathbf{s}} \in \mathbb{R}^n$  in order to compute the gradients of the attention parameters:

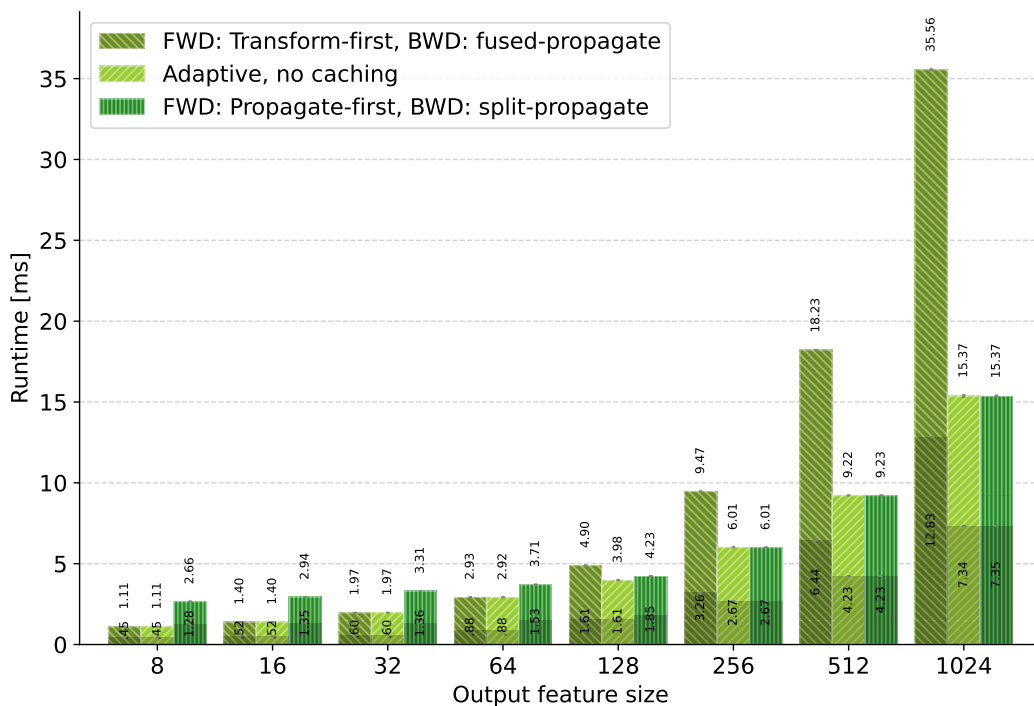
$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_{\text{src}}} = \mathbf{M}^T \frac{\partial \mathcal{L}}{\partial \mathbf{s}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{\text{dst}}} = \mathbf{M}^T \frac{\partial \mathcal{L}}{\partial \mathbf{d}}. \quad (23)$$

Finally, having  $\frac{\partial \mathcal{L}}{\partial \mathbf{d}}, \frac{\partial \mathcal{L}}{\partial \mathbf{s}}$  also allows us to compute the gradient for  $\mathbf{M}$ .

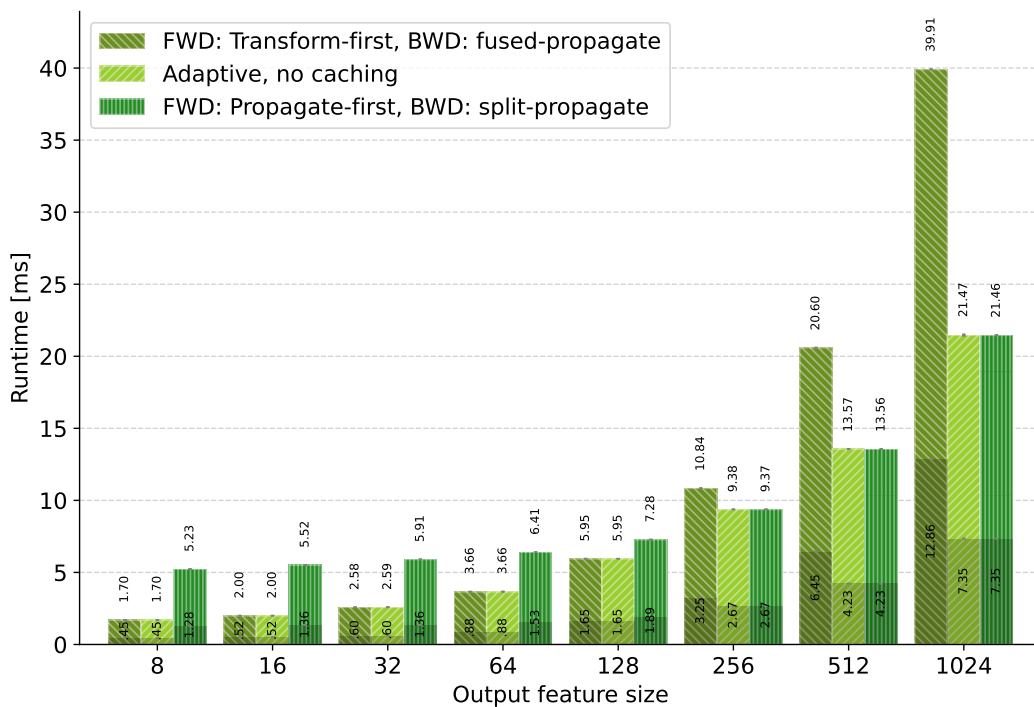
$$\frac{\partial \mathcal{L}}{\partial \mathbf{M}} = \left( \frac{\partial \mathcal{L}}{\partial \mathbf{d}} \right)^T \cdot \mathbf{a}_{\text{dst}} + \left( \frac{\partial \mathcal{L}}{\partial \mathbf{s}} \right)^T \cdot \mathbf{a}_{\text{src}} + \mathcal{A}^T \frac{\partial \mathcal{L}}{\partial \mathbf{X}'} \quad (24)$$

Plugging this value into Equations 16 and 17, we are able to obtain the gradients for the remaining parameters.

## H Figures

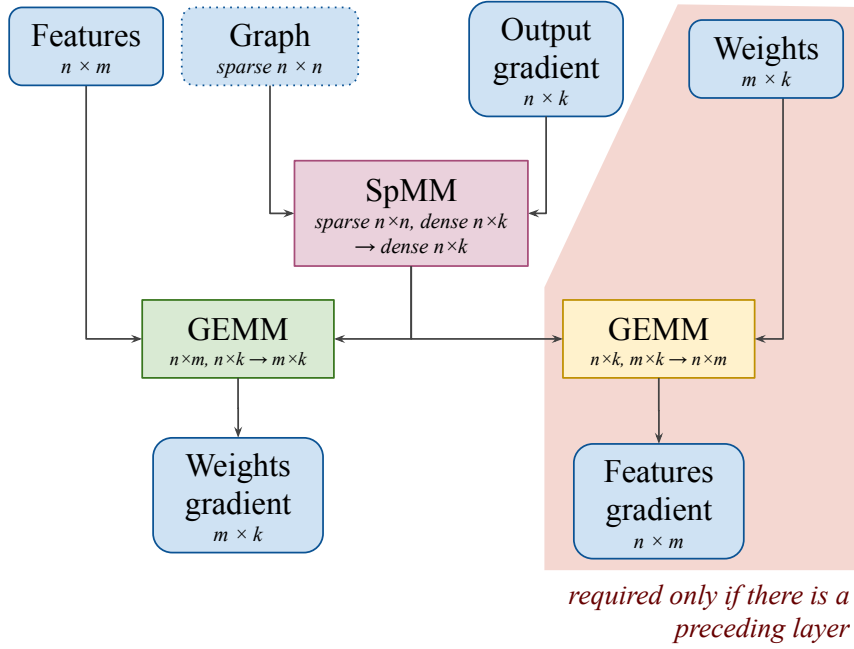


(a) Runtime without calculating input gradients.

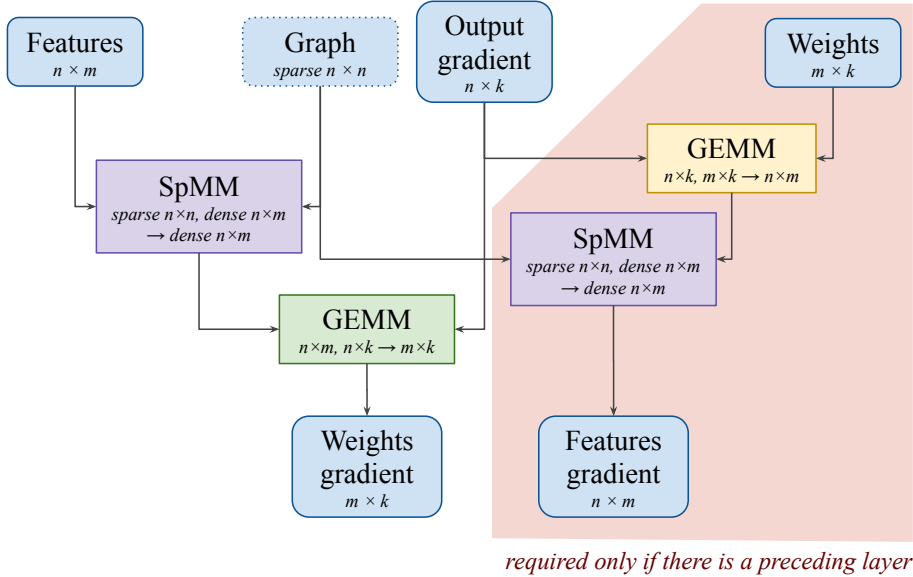


(b) Runtime including calculating input gradients.

**Figure 11:** Single GCN layer runtime on the OGB Arxiv dataset. Darker areas represent the time spent in the forward pass, total bar height is the total time of forward and backward passes.



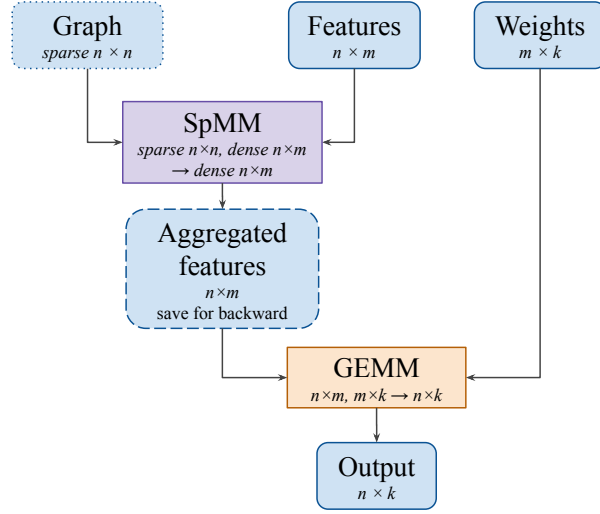
(a) Fused-propagate GCN backward pass, one  $n \times k$  SpMM.



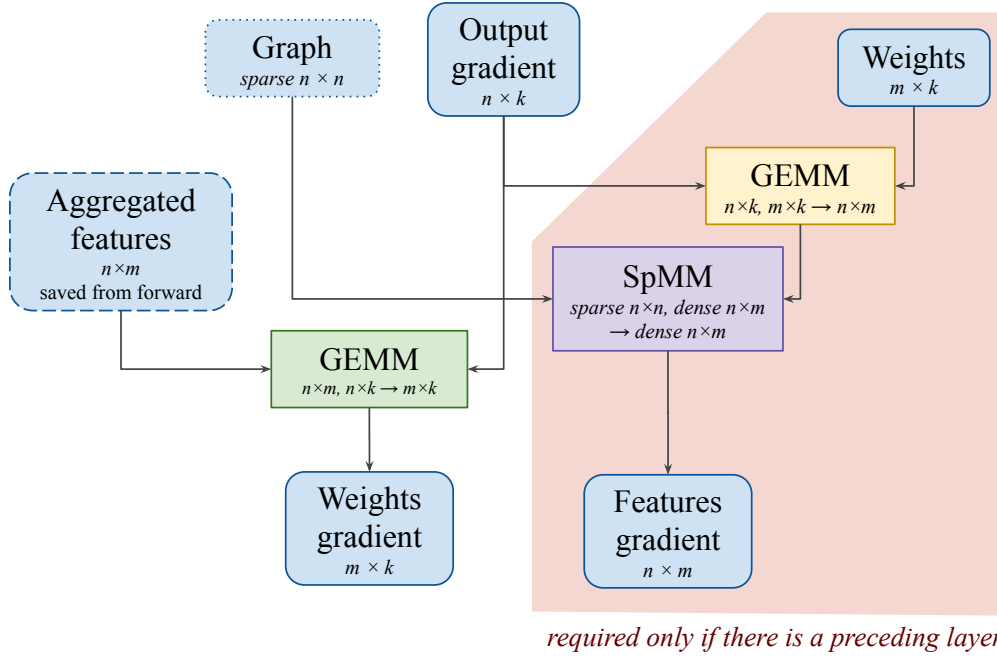
(b) Split-propagate GCN backward pass, two  $n \times m$  SpMMs.

**Figure 12:** Alternative schemes of computing the backward pass for GCN. Red area on the right indicates part of computation that does not need to be executed if the feature gradients are not needed. Compute nodes of the same color operate on the same shapes, which are indicated on the node in Einstein summation notation.



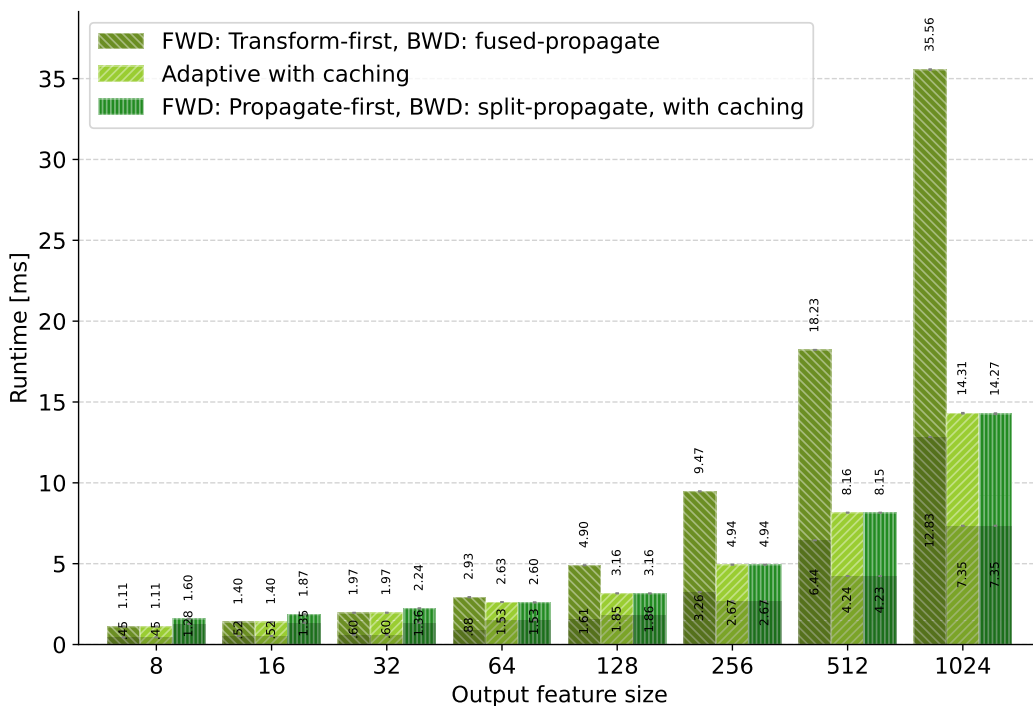


(a) GCN forward pass with caching. One  $n \times k$  SpMM.

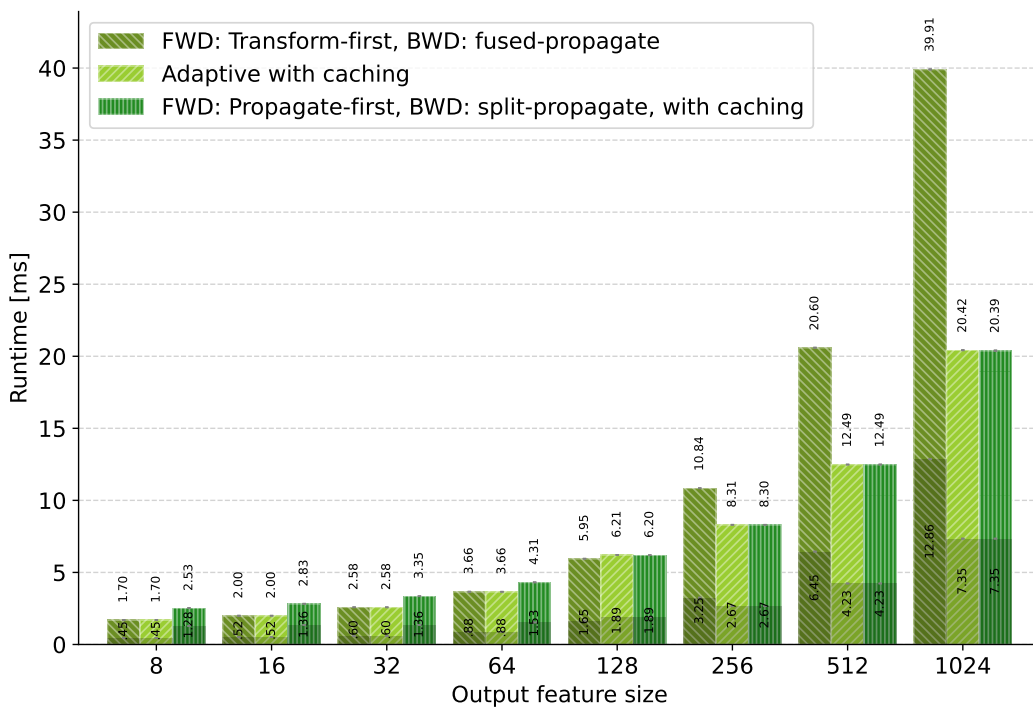


(b) GCN backward pass using cached values. One  $n \times m$  SpMM.

**Figure 13:** GCN computation scheme using caching to avoid recalculation. Red area on the right indicates part of computation that does not need to be executed if the feature gradients are not needed. Compute nodes of the same color operate on the same shapes, which are indicated on the node in Einstein summation notation.



(a) Runtime without calculating input gradients.



(b) Runtime including calculating input gradients.

**Figure 14:** Single GCN layer runtime on the OGB Arxiv dataset *using caching*. Darker areas represent the time spent in the forward pass, total bar height is the total time of forward and backward passes.

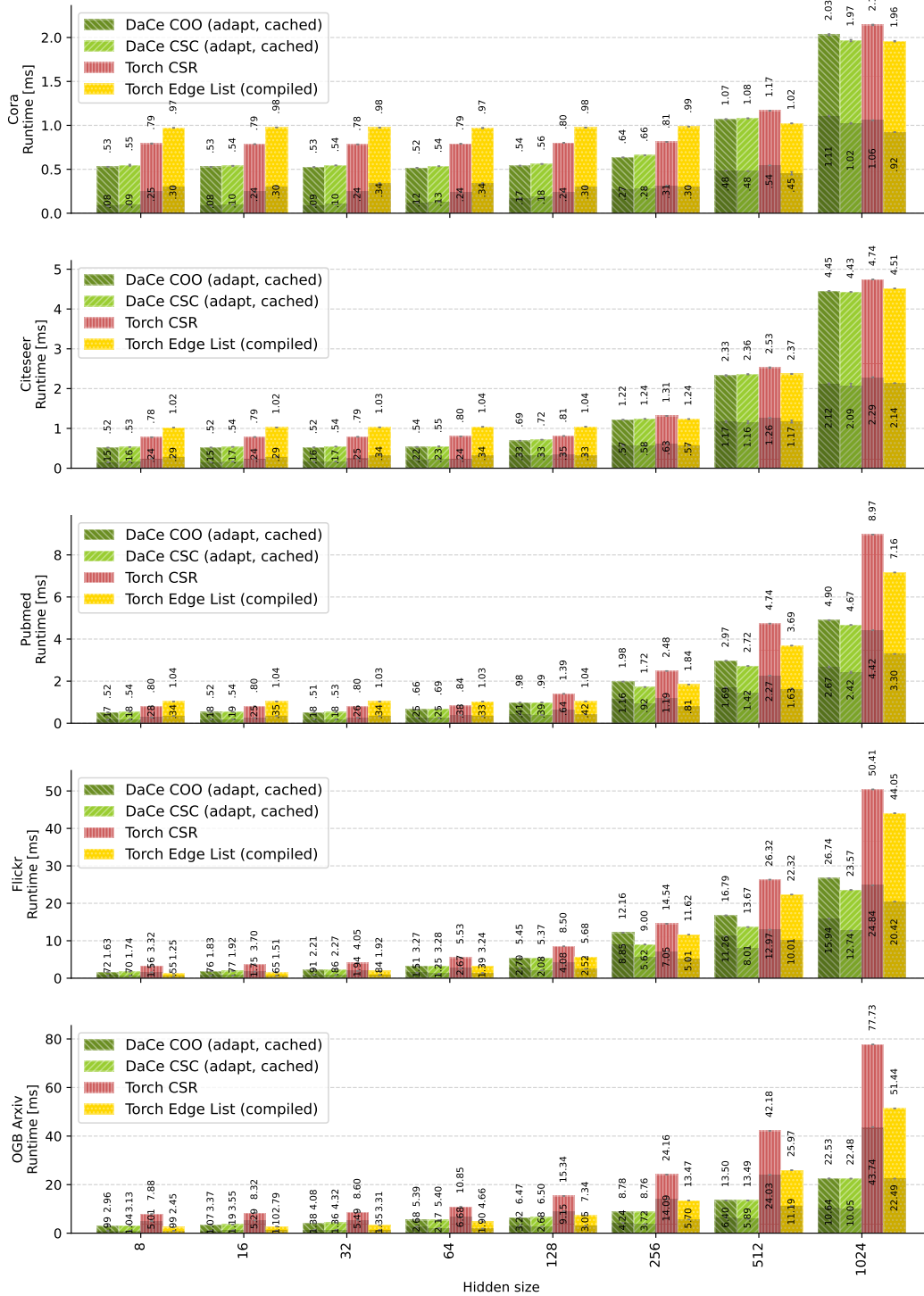


Figure 15: Detailed GCN runtime results, comparison of PyTorch Geometric and our work. Darker regions indicate the forward pass, total bar height is the sum of forward, loss and backward runtime.

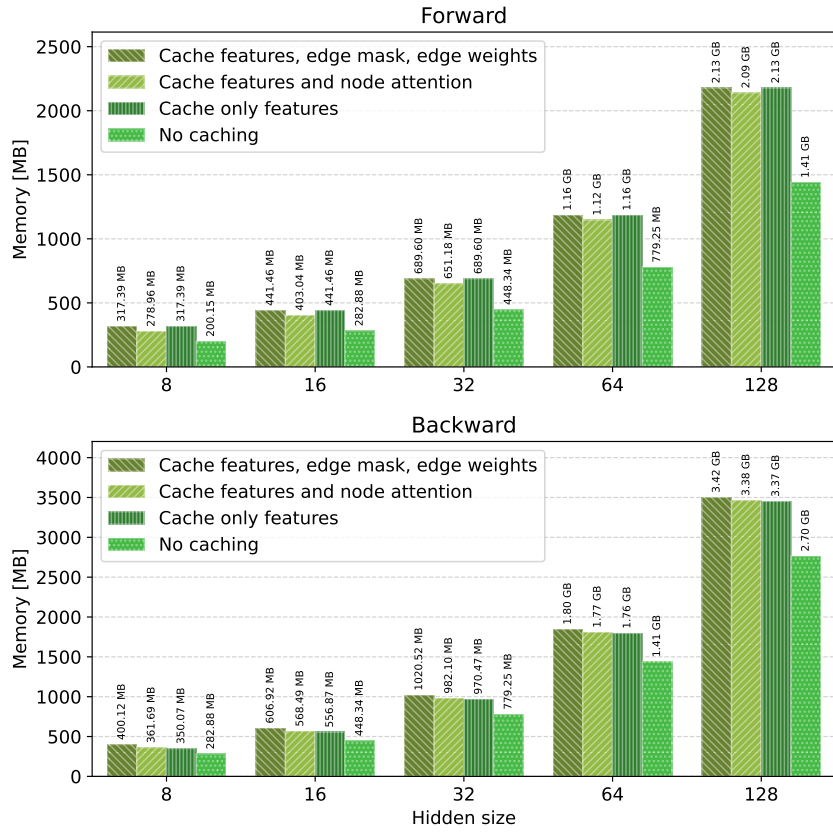
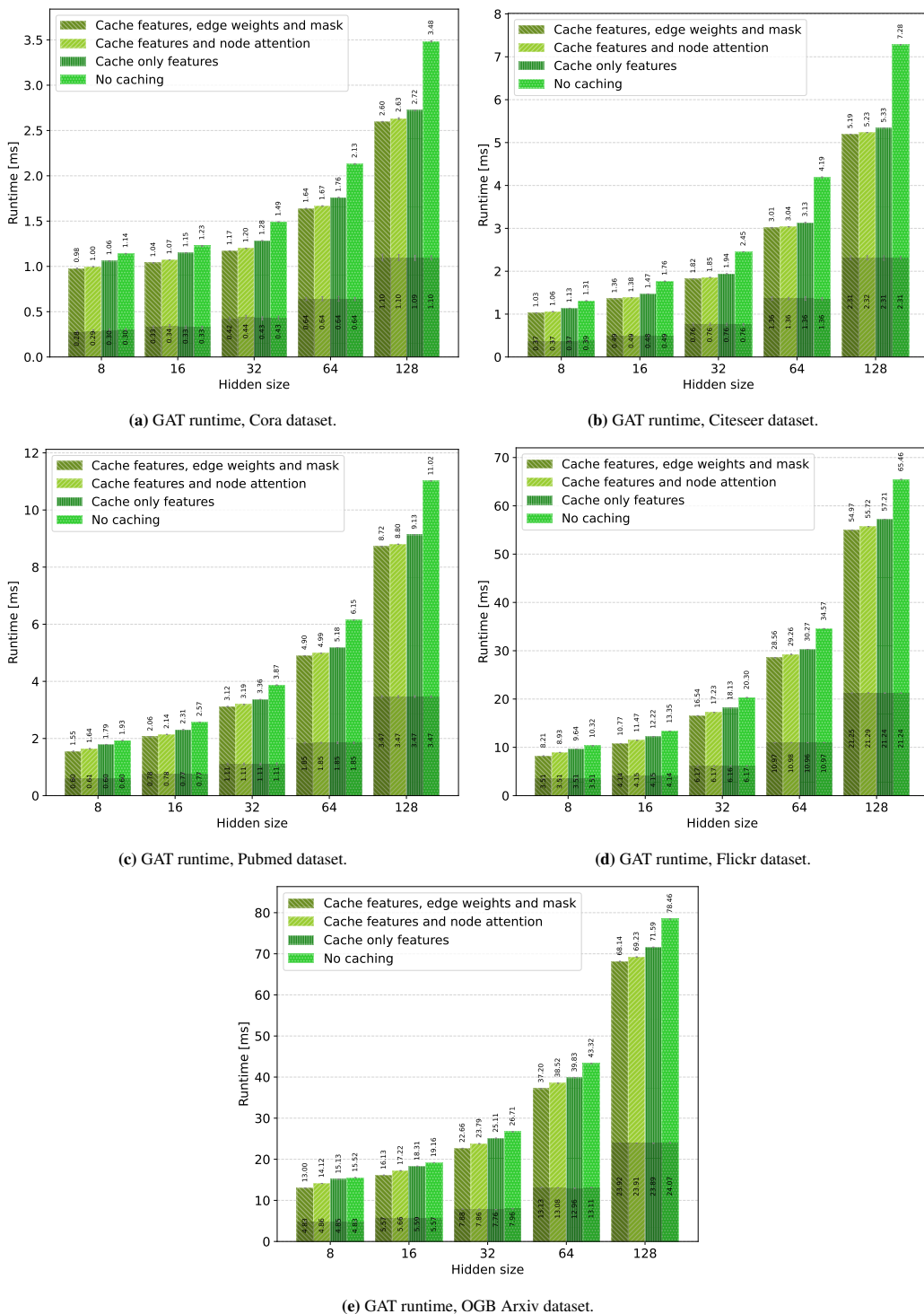
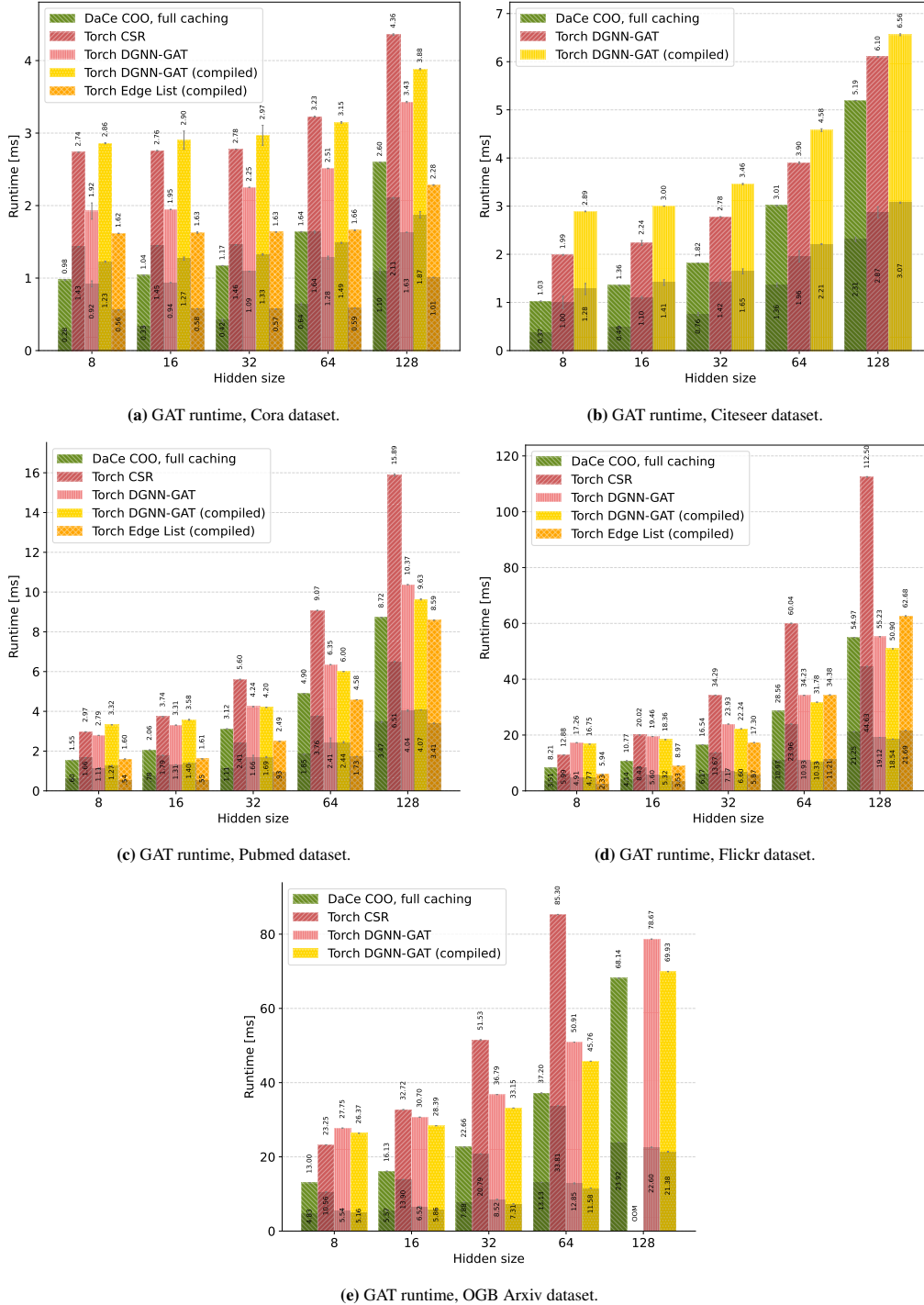


Figure 16: GAT memory use of different caching schemes on the OGB Arxiv dataset.



**Figure 17:** Detailed GAT runtime results, comparison of various caching schemes. Darker regions indicate the forward pass, total bar height is the sum of forward, loss and backward runtime.



**Figure 18:** Detailed GAT runtime results, comparison of our DaCe implementation against PyTorch. Darker regions indicate the forward pass, total bar height is the sum of forward, loss and backward runtime. Some data points for the compiled PyTorch implementation using edge lists are missing due to compilation errors.