MEGA: More Efficient Graph Attention for GNNs

Weishu Deng and Jia Rao
Department of Computer Science and Engineering
The University of Texas at Arlington, Arlington, TX 76019, USA
weishu.deng@mavs.uta.edu, jia.rao@uta.edu

2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS) | 979-8-3503-8605-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDCS60910.2024.00016

***Abstract*—Graph neural networks (GNNs) have demonstrated effectiveness across diverse application domains by leveraging graph information to uncover intrinsic correlations alongside feature representation. This enables GNNs to explore richer information compared to conventional neural networks, resulting in enhanced predictive performance. However, the integration of graph-structured data into the learning process poses two challenges. First, the sparsity and irregularity of the graph representation result in inefficient and expensive memory accesses on throughput-oriented accelerators, such as GPUs. Second, as GNN training involves interleaved graph operations to extract topological information and neural operations to update node or edge embeddings, the joint optimization of these two operations on accelerators is challenging due to their distinct resource requirements. Among GNN graph operations, graph attention which helps focus GNN training on highly correlated nodes is critical to training performance and model accuracy. However, our profiling of representative GNNs reveals that irregular memory access during graph attention accounts for the dominating overhead in GNN training.**

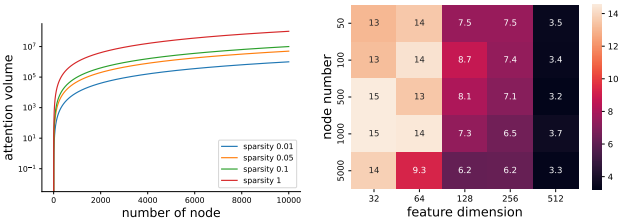
To address this issue, this paper proposes a more efficient graph attention method MEGA to accelerate GNN training. MEGA converts the original graph representation into one that regularizes memory access patterns for graph attention. Specifically, during preprocessing, MEGA traverses a graph to derive a schedule for graph attention and uses the schedule to reorganize the graph representation for optimized memory access. MEGA explores several techniques to balance memory access efficiency and preserve the original graph properties to avoid the loss of model accuracy. Experimental results with representative GNNs and graph data sets show that MEGA consistently outperforms conventional graph attention methods with up to 3x speedup.

***Index Terms*—Graph Neural Networks, GPU Computing, Memory Access patterns, Data transaction**

I. INTRODUCTION

Graph Neural Networks (GNNs) have gained widespread popularity and have been adopted in various machine learning applications, including recommended systems [1], molecular interfaces studies [2], physics system modeling, and disease classification [3]. During the learning process of GNNs, general information can be extracted using conventional neural operators, e.g. linear projection, and pooling layers. Meanwhile, graph operators enforce a local context, which magnifies the inner correlation of data [4] and improves the learning capability.

Graph attention is a mechanism employed in GNNs to assign importance or attention to different nodes and edges to selectively focus on relevant information while disregarding irrelevant or noisy data. The attention mechanism has also been explored in other learning algorithms. The transformer [5], a trending model for natural language processing (NLP), employs a global attention mechanism to compute correlation coefficients between every pair of words in a sentence. As a result, a more accurate semantic correlation can be learned between distinct words. The effectiveness of the attention mechanism in transformers has been demonstrated in extensive experiments and real-world applications [6]. Crucially, the high degree of parallelism and scalability inherent in the transformer mechanism and the effectiveness of global attention enable efficient processing of vast amounts of data, leading to the rapid evolution of large language models (LLMs).



(a) Attention volume. (b) Speedup on GPU.

Fig. 1: Graph Attention vs Global Attention.

It is possible to employ global attention in GNN training with a hypothetical fully connected graph in which edges are represented using an adjacent matrix and non-existing edges are all zeros. While global attention can be done through dense matrix operations and result in efficient and regularized memory accesses, it involves significantly redundant graph data, and its computational cost scales quadratically with the number of nodes as shown in Figure 1a.

On the other hand, graph attention does not incur redundant graph computation but requires that training follows the actual graph paths for performing attention. Such graph topology information and graph traversal utilities are typically provided by a third-party graph library. As real-world graphs exhibit sparse properties [7], graph attention significantly reduces the attention volume compared to global graph attention but at a cost of irregular and expensive memory access.

We quantify the overhead due to irregular memory access in graph attention and compare it with that in global attention in Figure 1b. We perform graph attention and global attention, respectively, on two graphs with the same number of nodes, one real-world, sparse graph, and one fully connected graph. We measure the time required to complete the graph attention

2575-8411/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDCS60910.2024.00016

and global attention and plot their ratio in Figure 1b. A ratio larger than 1 indicates that graph attention takes more time to complete even with less computation than global attention. The results in Figure 1b suggest that as the graph grows bigger and becomes more sparse (e.g., smaller feature dimension), graph attention incurs increasing overhead compared to global attention, highlighting the need to address the memory bottleneck in graph attention.

Recent works proposed ideas to address GNN inefficiency on GPU [8]–[10]. These studies suggest systematic optimizations, modifying low-level libraries through the analysis of input graph data and customizing on-device GNN compute kernels for different graphs with additional scheduling guidance passed to GPUs. In this study, we focus on optimizing the execution efficiency of GNNs on GPU by attaining a nuanced balance between resource efficiency in graph attention and the execution efficiency of global attention. Unlike the existing work, our approach does not necessitate modifications to low-level libraries. Inspired by GPU's high efficiency for dense matrix operations that effectively hide the costly off-chip memory operations [11] [12], we explore novel graph representations to regularize memory access patterns during graph attention, instead of tailoring GPU kernels to accommodate irregular graph accesses. The optimized access pattern not only enables sequential and coalesced memory accesses on GPUs but also facilitates data reuse during GNN training.

To achieve this goal, we propose a More Efficient Graph Attention mechanism (MEGA) to reorganize a graph for optimizing memory accesses in graph attention. The key design of MEGA is a preprocessing stage that determines a graph traversal schedule for graph attention later during GNN training. The preprocessing occurs on the CPU and is decoupled from the interleaved graph and neural operations on the GPU. MEGA derives a new graph representation in which graph vertexes are sorted according to the order they are accessed in the predetermined graph traversal schedule and placed along the diagonal of the new adjacency matrix. This new graph representation enables dense neural operations along the diagonal with aligned, regularized, and coalesced memory access. Similar to global graph attention that involves redundant computation to emulate a fully connected graph, MEGA also requires virtual edges that do not exist in the original graph to determine a graph traversal schedule or a path to (almost) fully cover all vertexes in the graph. MEGA leverages the Weisfeiler-Lehman (WL) method [13] to quantify the similarity (isomorphism) between the original graph representation and the proposed diagonal-oriented representation. MEGA also devise an adaptive diagonal attention approach to dynamically adjust the width of the diagonal (or attention window size) to maintain graph isomorphism.

We have implemented MEGA in PyTorch and evaluated its performance against conventional graph attention approaches. Experimental results with four representative graph datasets and two GNN configurations show that MEGA consistently outperforms conventional graph attention in training speed while incurring moderate space overhead and achieving comparable model performance (accuracy).

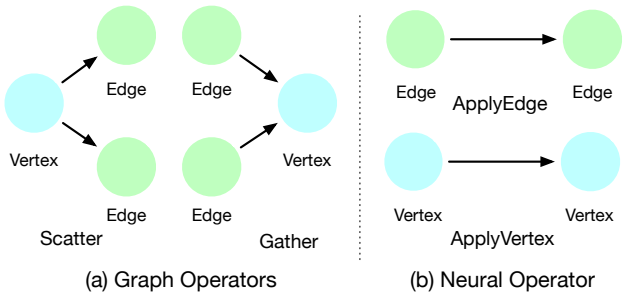


Fig. 2: GNNs primary operators.

II. GRAPH ATTENTION

A. Background

The graph attention layer utilized for updating node features in SoTA GNNs [14] comprises two fundamental operators, depicted in the equation below.

$$h'_u = \text{func (w,h}_u, \text{aggregate}_{v \in \mathcal{N}_{(u)}}(h_v, e_{v,u}))$$

the red part highlights the graph operation that aggregates neighboring information of node h_u with edge weight $e_{v,u}$, and the blue part indicates the subsequent neural operation that updates h'_u for the next layer.

Each operator is depicted in Figure 2. About graph operators shown in part (a), the arrows signify the data flow from vertex embeddings to edge embeddings for scatter operations and vice versa for gather operations. The required embeddings for the target node are typically discretely located in memory indexed by ID, making them challenging to access in a coalesced manner. Graph operations serve as an unavoidable prerequisite, attaining the messages from neighboring nodes. In part (b), the neural operations update all edges or vertices embeddings with element-wise activation, residual connection, or linear projection. In neural operations, identical trainable parameters and operations are shared across all embeddings, eliminating the guidance by graph information for complex data transactions.

Both types of operations are intricately intertwined, with no overlap between them. Delays in either operation can have a detrimental impact on the overall efficiency of the model.

B. Related Work

This section pertains to the latest research works that made significant contributions toward enhancing the GNNs efficiency.

1) Graph Transformer: Graph Transformers have gained popularity in graph learning due to their attributes of high parallelism and robust performance. The graph-agnostic global attention module is adapted for message passing. To enforce the learning of local context, topology information is encoded as structural embeddings [15] [16] [17] or positional embeddings [18] [19] [20] [21] [22]. These embeddings are appended

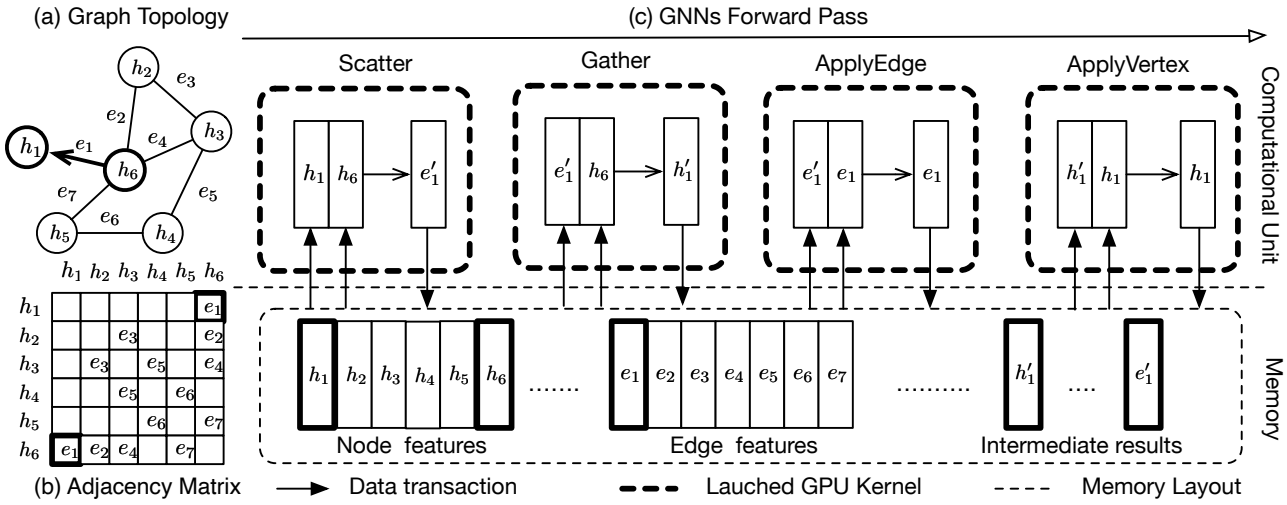


Fig. 3: Graph example for demonstration (a) and its corresponding adjacency format (b) The workflow of the forward pass of a graph attention layer on GPU. Bolden frames in the figures signify the required nodes and edges for computation (c).

to nodes as additional features. However, edge features are often overlooked in these works, and graph Transformers run the risk of overfitting to positional or structural encodings [23].

Although graph transformers execute efficiently, their global attention entails a trade-off in the form of substantial attention complexity, scaling quadratically as $O(n^2)$ with node number n, which becomes limiting as the graph size grows too large. Some studies suggest the use of linear transformers (partial graph attention) as a viable solution to mitigate this challenge. Notable examples include Linformer [24], Reformer [25], Longformer [26], Performer [27], and BigBird [28].

2) Systematic Optimization: Optimizations from a systematic perspective are proposed in various works [8]–[10]. These works identify significant bottlenecks and propose corresponding solutions. To exemplify these bottlenecks, we consider a graph attention update concerning node h_1 in the graph depicted in Figure 3a. The subsequent explanations elaborate on these identified bottlenecks.

Un-coalesced memory access GPUs favor continuous access patterns, thus facilitating coalesced access [29]. The irregularity of graphs poses difficulties in addressing space locality, where required neighboring operands are placed in discontinuous memory locations, shown by the h_1 and h_6 in the memory layout Figure c 3 . This hinders coalesced accessing, resulting in an abundance of costly loads or stores directed to off-chip memory. To improve the locality of the workload, GNNAdvisor [8] employed a reorder policy [30] to rearrange node indexes so that vertices sharing dense connections are assigned consecutive ID values and co-located in memory. As a result, embeddings can be accessed with better locality. Given the diverse spectrum of graph topology types, a universal reordering solution is not adept at effectively addressing the nuances inherent in various graph characteristics [31].

An alternative study [32] discerned that adopting an appropriate graph storage format holds the potential for enhancing the efficiency of GNNs during training.

Significant workload imbalance. Real-world graphs often display a highly skewed degree distribution. Concurrently, aggregation tasks are structured around targeting nodes within the graphs. This arrangement leads to significant variance in the workload assigned to each computation unit, determined by the node degree. To address the issue of imbalanced workload and the potential elongation of tail-latency arising from excessive workload, [8] introduces a neighbor grouping strategy. In this approach, computation units handling an excessive number of neighbor nodes are subdivided into smaller units, aligning with the average node degree. Subsequently, the outcomes generated by multiple computation units are aggregated into global memory using atomic instructions, thereby obviating the need for explicit data exchange. It is noteworthy that the implementation of this solution requires additional instructions to guide the optimization process.

Redundant data transaction. In GNNs with multiple stacked graph attention modules, the operations involving the graph and neural aspects are intricately intertwined. Due to disparate computational characteristics, identical data undergoes repetitive data transactions through frequent graph and neural kernel calls, as exemplified by the loading of h_4 and h_6 two times in Figure 3c. To address this issue, studies [9], [10] introduce intentional kernel fusion. This approach considers factors such as the minimal data visibility range and the equitable distribution of threads within warps, occasionally requiring additional effort for various aggregation approaches.

III. Preliminary

1) Models: The graph attention module is pivotal in discerning distinctions among various GNN configurations, directly impacting their performance and efficiency when

applied across different datasets. Consequently, the ensuing discussion will primarily revolve around the attention modules of the test models as elaborated below.

Gated Graph ConvNet (GCN) [33]: Employs batch normalization and residual connection to update both edges and nodes embeddings. The aggregation coefficient is determined by a normalized value of the sigmoid of updated edge embeddings, allowing node embeddings to be updated afterward.

Graph Transformer (GT) [18]: Adopts the mechanism from the transformer for computing graph attention coefficients. Within its attention module, multiple attention heads are enabled to discover richer information. Embeddings are then updated following a residual connection and layer normalization.

Both configurations frequently launch dependent graphs and neural operations, forming a non-overlapping serial compute graph. Struggling in between any point can throttle the overall efficiency. GT contains more trainable parameters and more frequent neural operations, theoretically achieving stronger learning capability at the cost of more computing effort. From a system perspective, greater occupation by highly efficient kernels can amortize the poor efficiency of the rest. However, end-to-end speedup is not always guaranteed with better execution efficiency, as convergence is not strictly correlated with model efficiency or the volume of trainable parameters across different datasets.

Table I provides an overview of the characteristics of the test models. The parameter volumes indicate the cumulative count of trainable parameters. The count of graph operations encompasses actions like scattering and gathering. GT demonstrates a heightened level of operations in both neural and scattering aspects due to its involvement in additional query, and key operations based on the graph.

TABLE I: Model Configuration Statistics.

	GCN	GT
Parameter Volume	$5d^2$	$14d^2$
Scatter(edges) calls	$\times 1$	$\times 5$
Gather(nodes) calls	$\times 2$	$\times 2$

2) Datasets: The downstream tasks for GNNs encompass graph prediction, node prediction, and edge prediction. Our study concentrates on graph prediction tasks that generate a single scalar value per graph. This scalar value remains invariant across both the original and augmented output data. To conduct our investigation, we employ four graph datasets that encompass both graph regression and graph classification tasks

ZINC dataset [34] and the **AQSOL** dataset [35] are widely recognized molecular graph datasets primarily employed for regression tasks. These tasks involve the prediction of molecular properties, where the node features serve as representations of atom types, and the edge features convey information about the bonds existing between atoms.

CSL dataset [36] comprises 4 types of regular graphs characterized by edges forming a cycle, and it incorporates skip-links connecting nodes. **CYCLES** dataset [37] consists of similar cycles while others do not where the cycles contain fixed lengths. Both of these datasets are synthetically generated with the specific purpose of assessing the expressive capabilities.

Table II provides an overview of datasets. These statistics furnish valuable insights into both the computational and communication volumes associated with graphs and the frequency of data transactions per node. It is noteworthy that all the test datasets demonstrate sparsity characteristics. Furthermore, it is pertinent to observe that the majority of real-world graphs similarly exhibit sparse properties [7].

TABLE II: Graph Statistics.

Datasets	train	validation	test	nodes	edges	sparsity
ZINC	10000	1000	1000	23	50	0.096
AQSOL	7985	996	996	18	36	0.148
CSL	90	30	30	41	164	0.098
CYCLS	9000	1000	10000	49	88	0.036

TABLE III: Graph Statistics.

Datasets	$\mu(\sigma(d))$	$\sigma(d_{min})$	$\sigma(d_{max})$	$\sigma(d_{mean})$	$\mu(\epsilon)$
ZINC	0.5116	0.0059	0.1998	0.0052	0.94
AQSOL	0.6255	0.0987	0.3106	0.0511	0.87
CSL	0.0	0.0	0.0	0.0	1.0
CYCLS	0.4737	0.0	0.5045	0.0241	0.71

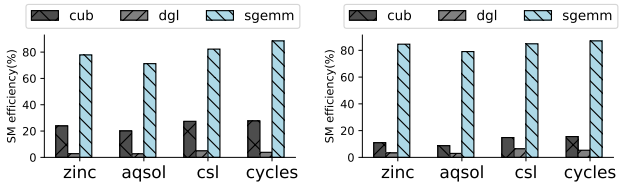
We present our observations on datasets in Table III. The graph datasets exhibit small values for the average degree variance $\mu(\sigma(d))$, indicating a consistent degree distribution shape shared across each instance. This consistency is further supported by the diminutive values of $\sigma(d_{\text{mean}})$, $\sigma(d_{\text{min}})$, and $\sigma(d_{\text{max}})$. Additionally, we conducted the Kolmogorov-Smirnov test [38] on graph degree distributions, where the proximity of these $\mu(\epsilon)$ values to 1 signifies a high degree of similarity among the distributions across different datasets. This characteristic allows us to apply a similar unfolding policy across graphs within each dataset, enabling batching for higher parallelism while minimizing padding waste.

A. Profiling results

To analyze the impact of datasets and model configurations on training efficiency, we gathered various metrics using the Nvidia profiling tool nvprof on the GPU, focusing on various on-device compute kernels.

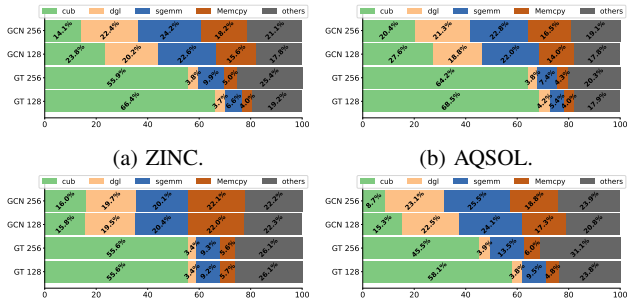
The baseline method involves using the graph library dgl for graph operations. To accelerate the graph attention process in the dgl module, the cub module is utilized for sorting embeddings based on given indices, facilitating fast fetching of neighbors during graph attention for the targeted node. The $sgemm$ kernel represents the matrix multiplication for linear projection, a highly optimized operation on the GPU.

We initiated an evaluation of the Stream Multiprocessor efficiency, a metric indicating GPU utilization concerning



(a) GCN. (b) GT.

Fig. 4: SM Efficiency.



(a) ZINC. (b) AQSOL.

(c) CSL. (d) CYCLES.

Fig. 5: Execution time.

independent kernels, as depicted in Figure 4. For this assessment, we configured the batch size to 64 and the hidden dimension to 128. As expected, the SM efficiency exhibited by the "$sgemm$" kernel significantly outperforms that of both "cub" and "dgl" kernels by a considerable margin. The overall efficiency is impacted by the computational load introduced by graph operations. Subsequently, We collected run time percentage attributed to each GPU kernel including $Memcpy$ for two batch size settings. As the results of batching within GNNs hold the potential to influence these overhead percentages due to the inherent variations in graph sizes and structures. In Figure 5, the hidden dimension is fixed at 64, and the batch size varies between 128 and 256. The overhead associated with graph kernels witnessed an amortization effect with larger batch sizes, concomitant with an increased allocation of work time to the $sgemm$ kernel. It is important to mention that the CSL dataset maintained a consistent graph size across the entire dataset, resulting in an unaltered overall percentage.

The GT module consistently shows a higher percentage of utilization of graph operations compared to GCN. This difference is attributed to the GT module's execution of five times more scattering operations related to edge features, as illustrated in Table I. Despite the GT module having approximately three times more trainable parameters associated with neural operations, utilization by $sgemm$ is still less than GCN. This observation implies that graph-related operations have a more dominant influence on determining the overall efficiency of the model. Different datasets also yield diverse profiling outcomes. This variance in results is caused by multiple factors, including variations in degree variance and graph sparsity. For instance, the ZINC dataset exhibits a higher percentage of graph operations due to its elevated sparsity value (Table II) and degree mean (Table III).

We collected other system metrics, including the count of Warp-level instructions for global loads, the percentage of stalls attributed to the absence of required input by the instruction, and the number of calls made by each kernel on all major kernels shown in Figure 9. Both graph kernels cub and dgl exhibit a notable deficiency in data locality, evidenced by the substantial percentage of stalls and the excessive volume of global loads observed during their execution. As a result, it becomes abundantly clear that the irregular memory layout emerges as the predominant factor contributing to the efficiency bottleneck encountered within the domain of GNNs.

B. MEGA *Design*

Our proposed approach aims to expedite the end-to-end training process without compromising model performance. The key idea is to enhance runtime GPU memory efficiency. In MEGA , graph data is reorganized to facilitate sequential access while maintaining a trivially relaxed graph isomorphism. This involves assimilating graph information into the reorganized embedding ordering, enabling subsequent efficient window-sized attention within graph operations.

The notations used in this study are provided in Table IV. The input graph is represented in the coordinate format as a list of vertex pairs, where $(v_{\text{src}}, v_{\text{dst}})$ denotes an edge belonging to the set E. For simplicity, we assume the graph to be undirected in the following algorithm, with minor adjustments needed for directed graphs.

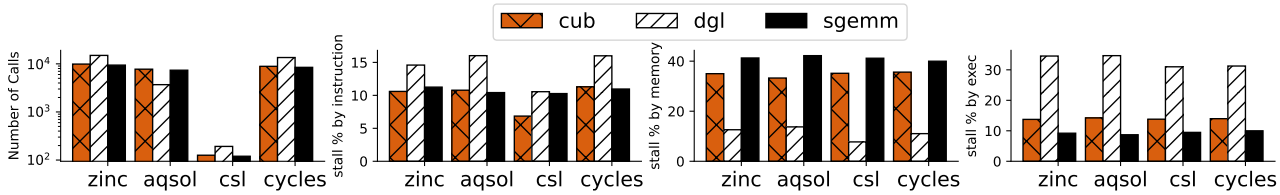
TABLE IV: Notations.

Symbom	Definition
G	Graph topology
V	Set of vertices in G
E	Set of edges in G
n	number vertices; $n = \|V\|$
m	number edges;$m = \|E\|$
$d_{(v)}$	degree of vertex v
$\mathcal{N}_{(v)}$	unvisited neighbors of vertex v
$size(.)$	function returning the size of a set
$\mathcal{P}$	Nodes index within path
U	unvisited nodes index by path
θ	edge coverage percentage

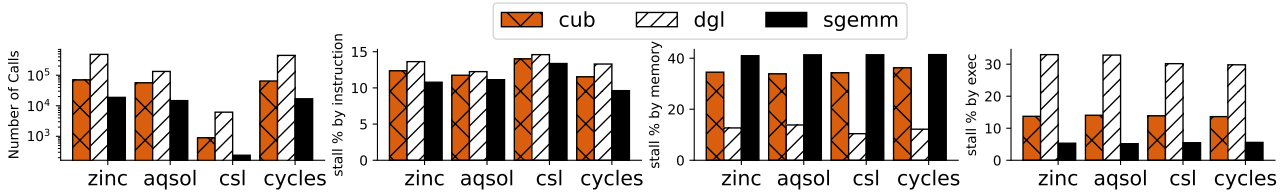
Graph Reorganization

Sparse fetching is frequently employed when accessing data of connected neighbors for a targeted node during graph attention, resulting in a significant volume of costly data access. Drawing inspiration from the notion that one can learn the topology of a maze after traversing it entirely, we propose an alternative perspective on the graph, referred to as a "path". The graph is restructured into a path representation while retaining the original graph properties. By accessing data along the path, the visiting sequence during attention is ensured to be serialized.

To restructure the entire graph, the path must traverse the complete graph. However, accomplishing a traversal of a



(a) GCN.



(b) GT.

Fig. 6: GPU Kernels Profiling.

complete graph without revisiting nodes is extremely difficult and oftentimes infeasible, as extensively studied in Eulerian path analysis [39]. We formulate our traversal objective with a relaxed requirement: The traversing agent is initialized at a specific node in the graph. When a dead end is encountered during traversal, the agent is allowed to jump to any node with unvisited neighbors. Eventually, the traversal terminates once all edges or a certain percentage of edges (denoted as θ) have been visited by the agent.

Preserving Graph Properties

All traversal objectives need to be achieved under a prerequisite: the path-based representation must preserve its isomorphism to ensure the correctness of graph attention.

We leveraged the Weisfeiler-Lehman (WL) method [13] to cross-verify the isomorphism between the original graph and the respective path representation. The key idea of the WL algorithm is to assign unique labels to every vertex. In subsequent steps, vertex labels are repeatedly updated with the concatenation of all labels of neighboring nodes, including self-label. Graph discrepancy can be determined by the difference among the updated labeling strings.

To maintain the local context of the graph in our path format, it is necessary for consecutive node appearances to be connected in the original graph. An exception arises when all neighbors of the current node have been visited, in which case the subsequent node is selected from unconnected nodes. The newly initiated path is connected to the preceding path through a virtual edge, signifying the absence of actual connections in the original graph. This virtual edge also introduces the potential for exploring hypothetical connections, as jumping nodes are frequently selected within densely connected clusters in cases where no direct connections exist in the provided graph. The objective is formulated as follows for picking the subsequent candidate:

$$node_{next} = \underset{v}{\operatorname{argmax}}\, size_{(\mathcal{N}(v)\cap\mathcal{P}[-\omega:])} \tag{1}$$

v denotes subsequent candidates, and ω represents the window size, indicating the visible range for nodes v appearing in the path. The window size can be adaptively tuned based on the mean degree of the input processing graph. As exemplified in Figure 7, the window size of the path is configured to 1. The objective equation identifies the node that maximizes the set intersection. This mechanism ensures correctness in one-hop aggregation and theoretically maximizes isomorphism with multi-hops.

Limiting vertex revisit Memory redundancy is inevitable when node revisits occur in our graph traversal policy. When the number of revisits reaches a certain threshold, the benefits of serialized memory access are offset by the additional memory that needs to be accessed due to redundancy. Therefore, limiting revisiting in the path is extremely important.

Accomplishing this goal presents a challenge owing to the diverse characteristics exhibited by real-world graphs. These graphs demonstrate a range of degree distributions, such as uniform, normal, and predominantly power distributions [40]. Thus, an adaptive traversing policy becomes imperative to accommodate these varied graph characteristics. Two primary objectives are formulated for constraining revisiting:

- Minimize the incidence of commencing a new path from nodes with an even degree.
- Enlarge the path width when nodes possess a high degree.

Let's represent a set of node degrees in the graph as $D = d_1, \ldots, d_n$. The theoretical lower bound of revisiting number can be optimistically achieved with a window size ω, expressed as $\sum_{d_i \in D} \lceil \frac{d_i}{\omega} \rceil - n$, where n is the node appearance number, ensuring each node appears at least once. Increasing ω enlarges the coverage on popular nodes with more connected neighbors, which are often considered more significant for contributing to the learning context. Reframing the objective equation 1 as our traversing agent action policy yields the following:

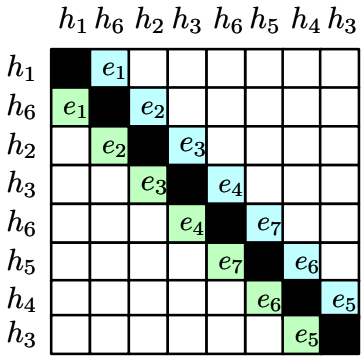


Fig. 7: Path-based Graph Representation in Adjacency Format.

$$\mathcal{P}_{[i+1]} = \underset{v}{\operatorname{argmax}}\, size_{(\mathcal{N}_{(v)} \cap \mathcal{P}_{[i-\omega:i]})} \tag{2}$$

$P_{[i]}$ represents the last visited node so far. A candidate maximizing correlation with the previous ω nodes in the path is heuristically selected for visiting in the subsequent step. Candidates v for equation 2 are filtered according to the following conditions in descending order of priority:

$$v = \begin{cases} \{u|\ u \in U, u \in \mathcal{N}_{(\mathcal{P}_{[i]})}\ \} & \text{if } notEmpty \\ \{u|\ u \in \mathcal{P}_{[:i]}\ , \mathcal{N}_{(u)} > 0\} & \text{if } notEmpty \\ U & \text{else} \end{cases}$$

Initially, the agent scans the unvisited neighbors of node $p_{[i]}$. This process continues until the neighborhood $\mathcal{N}(\mathcal{P}[i])$ is exhausted. Subsequently, candidates are selected from the visited nodes with remaining unvisited neighbors, denoted by $\{u|\ u \in \mathcal{P}_{[:i]}\ , \mathcal{N}_{(u)} > 0\}$. If both pools of previous candidates are empty, nodes are then chosen from the set of unvisited nodes U.

We provide pseudocode shown in Table 1 incorporating the above objectives. Two key data structures are used to minimize the search space. The neighbors dictionary $\mathcal{N}$, initialized at line 1, s initialized at line 1, stores the set of unvisited neighbors for each vertex. In each iteration, the selected candidate is removed from the neighbor set of the previous vertex, and the process is terminated until all neighbor sets are empty, ensuring that the search space monotonically decreases. The stack stk, declared at line 1, stores the visited nodes with remaining unvisited neighbors. The Last-In-First-Out mechanism ensures that the topmost vertex popped from the stack is the most correlated to the recently traversed path. The algorithm continues until all nodes have been visited, and a certain ratio θ of edges have been traversed. The desired outcome exemplified by this algorithm is illustrated in Figure 7, with the input graph shown in Figure 3a.

Lastly, our approach enables the option to drop edges during the traversal process. As elucidated by the work [41], there are instances in which GNNs that rely on fixed topology aggregation may lead to overfitting and a diminishment of their generalization capacity.

Algorithm 1 Objective Graph traversal.

```
0: //return number of neighbors of v in previous ω traversed nodes
0: function CORRELATE(v)
0:     return size of N_(curr) ∩ P_[−ω:]
0: end function
0: //return reordered data sequence
0: function MAIN(G, α, ω)
0:     P ← list
0:     stk ← stack
0:     N ← dict for store Un-visited neighbors
0:     U ← Un-visited vertex set
0:     curr ← starting vertex
0:     for src, dst ← edge_pairs in G do
0:         neighbors[src].append(dst)
0:     end for
0:     P.add(curr)
0:     U.remove(curr)
0:     while not (U ≠ ∅) or visited edge ratio > α do
0:         pre, ns ← curr, neighbors[curr]
0:         if ns ≠ ∅ then
0:             curr ← arg max_v[correlate(v) for v in ns]
0:         else if stk ≠ ∅ then
0:             stk.pop() until N_(stk.top()) ≠ ∅
0:             curr ← stk.pop()
0:         else
0:             curr ← arg max_v[correlate(v) for v in U]
0:         end if
0:         P.add(curr)
0:         N_(curr).remove(pre)
0:         //N_(pre).remove(curr) for un-directed graph
0:         U.remove(curr)
0:         if N_(curr) ≠ ∅ then
0:             stk.push(curr)
0:         end if
0:     end while
0:     return P
0: end function=0
```

C. Adaptive Diagonal Attention

Embeddings with a designated sequence (path-based representation) are prepared for loading into the model, given that the graph is imparted with a specific ordering and avoids the need for topology information. During graph attention, neighboring nodes are placed consecutively in memory, maximizing memory efficiency with more coalesced memory access. This memory alignment aligns seamlessly with diagonal attention, wherein nodes aggregate messages along a windowed diagonal depicted by the colored grid in the adjacency matrix, as illustrated in Figure 7. Nodes with degree values higher than the window size can be accommodated through multiple appearances in the path, as exemplified by nodes h_3 and h_6. For graphs with higher average degrees, the window size can be adaptively enlarged to cover more edges while requiring

fewer node appearances.

This diagonal attention mechanism resonates with the concept presented in `longformer` [26] for NLP tasks. It imposes a fixed attention range on surrounding words, grounded in the assumption that words situated far from the current context contribute minimally to the learning process. The analogy between graph attention and `longformer` is discernible to a certain extent. Popular nodes that have more neighbors exert more influence [42] [43] [44] and receive greater attention, while a majority of nodes share similar low degree value. Akin to crucial words may manifest multiple times in a sentence.

The integration of diagonal attention also mitigates redundant computations associated with bidirectional edges. In conventional implementations, the update of edge embeddings occurs twice during the message-passing process when aggregates are initiated twice by two nodes connected with the same edge. By organizing the adjacency matrix of the original to maintain symmetry around the diagonal in MEGA, results on one side can be reused and applied to the corresponding side, as operands and operations are identical.

IV. EVALUATION

In this section, we present evaluation results aimed at addressing the following inquiries:

- To what extent does the path representation preserve graph characteristics? (IV-B1)
- To what degree does the graph attention achieve optimization in the context of MEGA ? (IV-B2)(IV-B4)
- How effectively is the learning performance preserved when utilizing the path representation? (IV-B4)

A. Evaluation Methodology

Experimental setup.

Experiments were carried out on a GPU server equipped with GeForce GTX 1080 GPUs and a CPU featuring 16 cores (Intel(R) Core(TM) i7-5960X CPU @ 3.00GHz) with PCI-e 3.0x16. The dependency configurations utilized for these experiments adhere to the specifications detailed in the reference [45]. The L2 cache capacity of GTX 1080 GPUs is 2048 KB, which proves inadequate for caching node and edge embeddings. Consequently, the majority of memory access for graph operations is directed to the global memory, which boasts a capacity of 12 GB but incurs higher costs. This discrepancy in efficiency between serialized access and randomized access becomes more pronounced as a result.

The evaluations were conducted using the *Mega* and a baseline method DGL with two model configurations, GT and GCN, across four datasets as detailed in Section III. In each experimental setup, both methods employed models with identical parameter counts. When comparing the end-to-end speedup, we ensured that path representations in *Mega* encompassed all nodes and edges present in the original graph, thus maintaining equitable workloads across respective graphs.

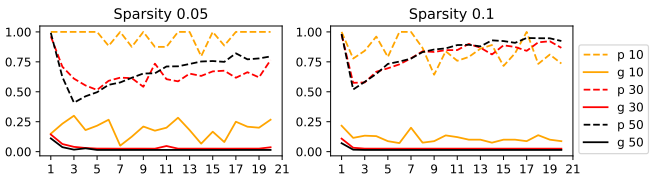


Fig. 8: Isomorphism Evaluation.

B. Evaluation Results

1) ***Isomorphism Results*** *:* The evaluation of graph isomorphism utilizes the WL test, as elaborated in Section III-B. The similarity score is computed between the path representation and the original graph, where a score of 1 indicates complete graph identity. We conducted a comparative analysis between the path-based representation and the full labels set (aggregation results by global attention) to assess the ability of each attention approach to maintain graph characteristics across two prevalent graph sparsity levels illustrated in Figure 8 as 0.05 and 1. Graph sparsity is defined as the ratio of the actual number of edges to the edges of the fully connected graph.

The labels p and g denote path representation and global attention with respective node numbers. As indicated by the results in Figure 8, the path representation consistently ensures identity in 1-hop aggregation. The similarity coefficient may fluctuate as the aggregation hop count grows. Nonetheless, the majority of information is preserved and can be compensated for by the resilience of neural networks, resulting in comparable performance.

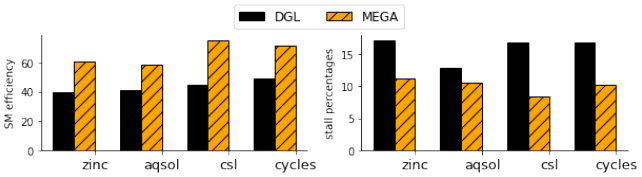
2) ***Systematic Results*** *:* The SM efficiency and the stall percentage due to memory are identified as critical indexes revealing model inefficiency in Section III-A and are utilized to evaluate the optimization of the proposed method. We set the batch size and hidden dimension as 64 and 128, respectively, as these values exhibited the poorest efficiency during the execution of conventional graph attention in previous profiling.

For each metric, we calculated a normalized value to estimate the overall performance. This was achieved by considering the invocation number of kernels participating in the workflow and averaging the metric measurements across each kernel type, employing the following equation:

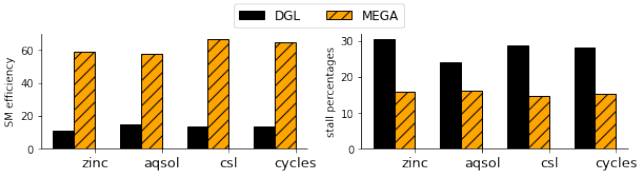
$$Metric = \frac{\sum_{k \in K} metric_k \times n_k}{\sum_{k \in K} n_k}$$

K represents the set of GPU kernel names called during the executions and n_k indicates the number of calls for that kernel.

Based on the evaluation results shown in Figure 9, *Mega* consistently demonstrates stable high SM efficiency and low stall percentage in all settings. The varying efficiency exhibited by DGL under different settings can be attributed to the varied frequency of graph operations across model settings and the distinct data statistics. For example, GT involves five times more aggregation operations than GCN, resulting in the SM efficiency of the DGL method with GT being only a quarter



(a) GCN.



(b) GT.

Fig. 9: memory metrics.

of that of GCN, mainly due to more occupation by inefficient graph operations. Remarkably, the efficiency of *Mega* remains unaffected by these variations.

As anticipated, the diagonal attention along the path representation regularizes graph operations as dense operations, resulting in a reduced occurrence of stalls due to graph irregularities and mitigates performance fluctuations when faced with different datasets or models.

3) ***Runtime Results****:* Results for epoch execution time with common batch sizes settings as 64, 128, and 256, are presented in Figure 10. The dense compute kernel $sgemm$ is identified for providing the best efficiency, and its percentage occupation is also provided to assess the overall efficiency.

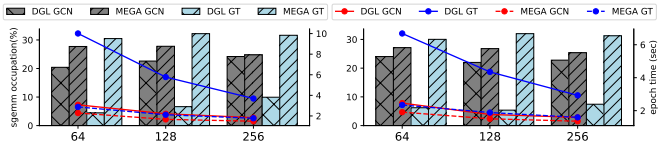
The *Mega* method demonstrates less epoch overhead and greater occupation by $sgemm$ across all settings. However, the epoch speedup achieved by *Mega* does not exhibit an increase as the batch size grows. This observation aligns with our assumption that a larger computation volume for neural operations can amortize the lag caused by graph operations.

As anticipated, *Mega* provides substantial efficiency improvement for GT and comparatively minimal improvement for GCN, as GT involves more graph operations. This provides further compelling evidence that *Mega* effectively bridges the efficiency gap between graph operations and neural operations.

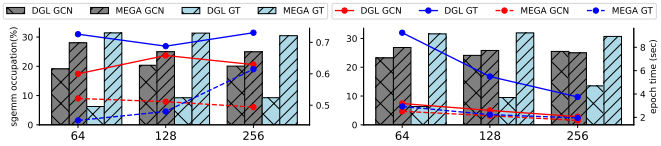
4) ***End-to-End Result****. :* The convergence time serves as the primary criterion for assessing the speedup achieved by the MEGA method. All models employed are configured based on performance-oriented studies respected to test datasets [45] [18].

To illustrate real-time speedup in evaluation coordinates, the x-axis represents wall clocks in seconds, while the y-axis represents the loss/accuracy indicating model performance.

MEGA demonstrates acceleration across all datasets to varying degrees. For the zinc dataset, utilizing the GT configuration, the time required by MEGA converges to optimal



(a) zinc & aqsol.



(b) csl & cycles.

Fig. 10: Runtime.

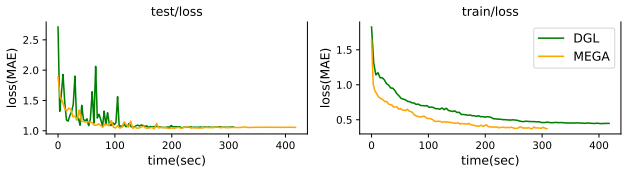


Fig. 11: AQSOL evaluation.

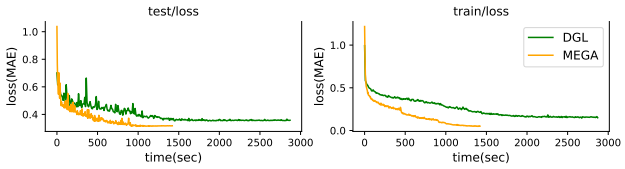


Fig. 12: ZINC evaluation.

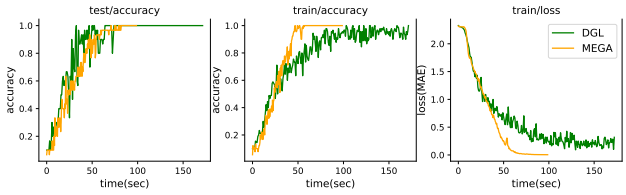


Fig. 13: CSL evaluation.

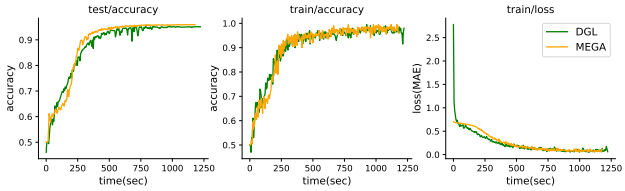


Fig. 14: CYCLES evaluation.

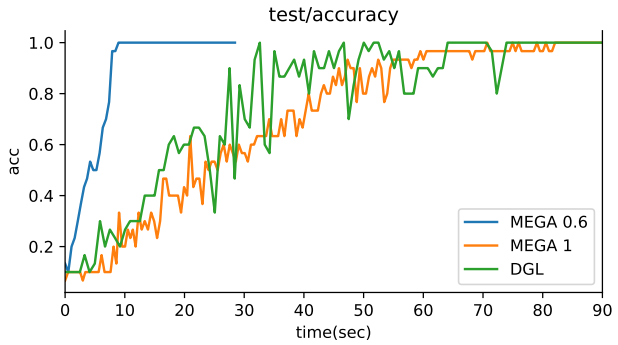


Fig. 15: AQSOL with edges dropping enabled.

results in half the time compared to the baseline approach (Figure 12). Similarly, in the aqsol dataset (Figure 11), csl dataset (Figure 13), and cycles dataset using GCN configuration (Figure 14), MEGA achieves approximately $\times 2.6$, $\times 2.2$, and $\times 1.6$ speedup.

5) ***End-to-End Results with Edge Droping*** *:* We further assess *Mega* with Edge Dropping enabled in our path representation. It is important to note that the benefits of edge-dropping are not universally applicable across all datasets. In our experiments, 20% of edges are randomly dropped within every graph and its respective path representation when tested with the aqsol dataset. Results reveal that *Mega* achieved a notable speedup of 5.9x while maintaining the same level of accuracy, as depicted in Figure 15.

6) ***Distributed Communication Analysis*** *:* In the context of distributed learning, concerning communication analysis, the theoretical communication volume required for a partitioned graph is determined to be $O(\sqrt{\log k \log n})$, as studied in distributed graph communication research [46]. This volume exhibits linear growth with the number of nodes n and the partition number k, this usually requires expensive all-to-all communication. The partition of the path representation in *Mega* reduces this communication overhead to only two communications for adjacent path partitions theoretically, resulting in a distributed manner with $O(k)$ communication effort. Although the path representation introduces an increase in the value of n due to embedding rearrangement, the associated tradeoff is deemed justifiable, given that this value does not surpass a certain degree.

7) ***Limitations*** *:* The principal advantage of our approach lies in its ability to serialize the graph access pattern while minimizing memory overhead, thereby enhancing access efficiency.

To maximize access efficiency, it is imperative to restrict node re-visitation, avoiding the risk of potentially extending overall execution time. Consequently, achieving comprehensive coverage of the graph through a succinct path-based representation becomes paramount. Although real-world graph topologies span a spectrum, our traversal algorithm necessitates meticulous calibration to accommodate diverse graph characteristics.

Neglecting to preserve graph attributes can precipitate a degradation in performance, while failure to curb node redundancy may nullify the gains in access efficiency.

8) ***Discussion & Future work*** *:* In various works aimed at optimizing systems for machine learning, a recurring trade-off between performance and efficiency is observed, e.g., model pruning and model quantization. These methodologies strive to achieve significant efficiency gains while minimizing any compromise to the predictive capabilities of the model. For instance, model pruning techniques, in DeJa vu [47], identify contextual model sparsity, thereby reducing the computational workload associated with model inference.

Our work aims to devise an access-efficient path representation that encapsulates meaningful graph characteristics. Furthermore, empirical studies conducted in SparseGAT [48], demonstrate that a substantial portion of edges within real-world graphs can be eliminated without detrimental effects on model performance. Conversely, the removal of such edges can lead to performance enhancements in certain scenarios, albeit with the caveat of inducing noise during the learning process. MEGA , by extending its capabilities to explore graph sparsity akin to SparseGAT, also alleviates the computational burden associated with graph traversal by reducing the number of edges traversed along a path. For heterogeneous graph scenarios, MEGA can leverage the idea in HAN [49], MEGA can arrange multiple paths to cover distinct node types, subsequently merging hierarchically.

The effectiveness of MEGA also renders it suitable for applications governed by stringent latency constraints. MEGA can be applied with DYGAT [50], facilitates real-time stroke classification conducive to online handwriting interaction.

V. CONCLUSION

We introduce *Mega*, a method that enhances the runtime efficiency of graph attention on GPUs. Throughout our evaluations, we substantiate the effectiveness of *Mega* by comparing it to the conventional message-passing method. Our method reveals a superior memory accessing efficiency, while concurrently maintaining a comparable level of model predicting performance. The path-based graph employed in *Mega* proves to be a nontrivial representation that seamlessly contributes to GNNs learning, and it is a promising avenue for further exploration, demonstrating its potential to accommodate a wide spectrum of datasets. The inherent regularized communication pattern and efficiency in data processing offered by *Mega* allow GNNs to align with the ongoing trend of expanding model sizes.

REFERENCES

[1] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: a survey," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–37, 2022.

[2] M. Réau, N. Renaud, L. C. Xue, and A. M. Bonvin, "Deeprank-gnn: a graph neural network framework to learn patterns in protein–protein interfaces," *Bioinformatics*, vol. 39, no. 1, p. btac759, 2023.

[3] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57–81, 2020.

[4] M. E. Newman, "Mixing patterns in networks," *Physical review E*, vol. 67, no. 2, p. 026126, 2003.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[6] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, 2020, pp. 38–45.

[7] F. Chung, "Graph theory in the information age," *Notices of the AMS*, vol. 57, no. 6, pp. 726–732, 2010.

[8] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "Gnnadvisor: An adaptive and efficient runtime system for gnn acceleration on gpus," in *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, 2021.

[9] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, "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*, 2021, pp. 119–132.

[10] H. Zhang, Z. Yu, G. Dai, G. Huang, Y. Ding, Y. Xie, and Y. Wang, "Understanding gnn computational graph: A coordinated computation, io, and memory perspective," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 467–484, 2022.

[11] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[12] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 711–732, 2021.

[13] B. L. Douglas, "The weisfeiler-lehman method and graph isomorphism testing," *arXiv preprint arXiv:1101.5211*, 2011.

[14] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[15] D. Chen, L. O'Bray, and K. Borgwardt, "Structure-aware transformer for graph representation learning," in *International Conference on Machine Learning*. PMLR, 2022, pp. 3469–3489.

[16] G. Bouritsas, F. Frasca, S. Zafeiriou, and M. M. Bronstein, "Improving graph neural network expressivity via subgraph isomorphism counting," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 1, pp. 657–668, 2022.

[17] C. Bodnar, F. Frasca, N. Otter, Y. Wang, P. Lio, G. F. Montufar, and M. Bronstein, "Weisfeiler and lehman go cellular: Cw networks," *Advances in Neural Information Processing Systems*, vol. 34, pp. 2625–2640, 2021.

[18] V. P. Dwivedi and X. Bresson, "A generalization of transformer networks to graphs," *arXiv preprint arXiv:2012.09699*, 2020.

[19] D. Kreuzer, D. Beaini, W. Hamilton, V. Létourneau, and P. Tossou, "Rethinking graph transformers with spectral attention," *Advances in Neural Information Processing Systems*, vol. 34, pp. 21 618–21 629, 2021.

[20] H. Wang, H. Yin, M. Zhang, and P. Li, "Equivariant and stable positional encoding for more powerful graph neural networks," *arXiv preprint arXiv:2203.00199*, 2022.

[21] D. Beaini, S. Passaro, V. Létourneau, W. Hamilton, G. Corso, and P. Liò, "Directional graph networks," in *International Conference on Machine Learning*. PMLR, 2021, pp. 748–758.

[22] D. Lim, J. Robinson, L. Zhao, T. Smidt, S. Sra, H. Maron, and S. Jegelka, "Sign and basis invariant networks for spectral graph representation learning," *arXiv preprint arXiv:2202.13013*, 2022.

[23] L. Rampášek, M. Galkin, V. P. Dwivedi, A. T. Luu, G. Wolf, and D. Beaini, "Recipe for a general, powerful, scalable graph transformer," *Advances in Neural Information Processing Systems*, vol. 35, pp. 14 501–14 515, 2022.

[24] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *arXiv preprint arXiv:2006.04768*, 2020.

[25] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *arXiv preprint arXiv:2001.04451*, 2020.

[26] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[27] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser *et al.*, "Rethinking attention with performers," *arXiv preprint arXiv:2009.14794*, 2020.

[28] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang *et al.*, "Big bird: Transformers for longer sequences," *Advances in neural information processing systems*, vol. 33, pp. 17 283–17 297, 2020.

[29] M. Harris, "How to access global memory efficiently in cuda c/c++ kernels," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

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

[31] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 203–214.

[32] S. Qiu, L. You, and Z. Wang, "Optimizing sparse matrix multiplications for graph neural networks," in *Languages and Compilers for Parallel Computing: 34th International Workshop, LCPC 2021, Newark, DE, USA, October 13–14, 2021, Revised Selected Papers*. Springer, 2022, pp. 101–117.

[33] X. Bresson and T. Laurent, "Residual gated graph convnets," *arXiv preprint arXiv:1711.07553*, 2017.

[34] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[35] M. C. Sorkun, A. Khetan, and S. Er, "Aqsoldb, a curated reference set of aqueous solubility and 2d descriptors for a diverse set of compounds," *Scientific data*, vol. 6, no. 1, p. 143, 2019.

[36] R. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro, "Relational pooling for graph representations," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4663–4673.

[37] A. Loukas, "What graph neural networks cannot learn: depth vs width," *arXiv preprint arXiv:1907.03199*, 2019.

[38] J. Hodges Jr, "The significance probability of the smirnov two-sample test," *Arkiv för matematik*, vol. 3, no. 5, pp. 469–486, 1958.

[39] L. Euler, "The seven bridges of königsberg," *The world of mathematics*, vol. 1, pp. 573–580, 1956.

[40] A. Sala, H. Zheng, B. Y. Zhao, S. Gaito, and G. P. Rossi, "Brief announcement: revisiting the power-law degree distribution for social graph analysis," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 2010, pp. 400–401.

[41] Y. Rong, W. Huang, T. Xu, and J. Huang, "Dropedge: Towards deep graph convolutional networks on node classification," *arXiv preprint arXiv:1907.10903*, 2019.

[42] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[43] H. Cai, V. W. Zheng, and K. C.-C. Chang, "Active learning for graph embedding," *arXiv preprint arXiv:1705.05085*, 2017.

[44] X. Wang and G. Sukthankar, "Multi-label relational neighbor classification using social context features," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 464–472.

[45] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," 2020.

[46] F. Bourse, M. Lelarge, and M. Vojnovic, "Balanced graph edge partition," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1456–1465.

[47] Z. Liu, J. Wang, T. Dao, T. Zhou, B. Yuan, Z. Song, A. Shrivastava, C. Zhang, Y. Tian, C. Re *et al.*, "Deja vu: Contextual sparsity for efficient llms at inference time," in *International Conference on Machine Learning*. PMLR, 2023, pp. 22 137–22 176.

[48] K. Ding, Z. Xu, H. Tong, and H. Liu, "Data augmentation for deep graph learning: A survey," *ACM SIGKDD Explorations Newsletter*, vol. 24, no. 2, pp. 61–77, 2022.

[49] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The world wide web conference*, 2019, pp. 2022–2032.

[50] Y.-T. Yang, Y.-M. Zhang, X.-L. Yun, F. Yin, and C.-L. Liu, "Dygat: Dynamic stroke classification of online handwritten documents and sketches," *Pattern Recognition*, vol. 141, p. 109564, 2023.