MaxK-GNN: Towards Theoretical Speed Limits for Accelerating Graph Neural Networks Training

Hongwu Peng[1,*], Xi Xie[1,*], Kaustubh Shivdikar[2], MD Amit Hasan[1], Jiahui Zhao[1], Shaoyi Huang[1], Omer Khan[1], David Kaeli[2], Caiwen Ding[1]
[*]These authors contributed equally.
[1]University of Connecticut, USA. [2] Northeastern University, USA.
[1]{hongwu.peng, xi.xie, amit.hasan, jiahui.zhao, shaoyi.huang, khan, caiwen.ding}@uconn.edu, [2]{shivdikar.k, d.kaeli}@neu.edu

Abstract

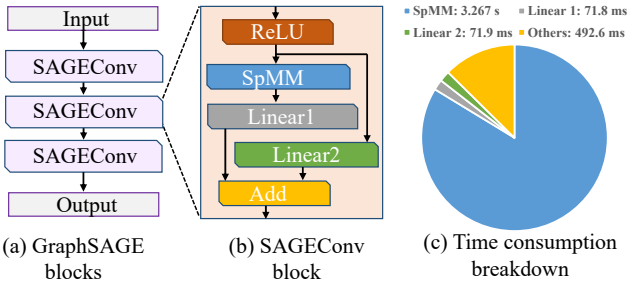
In the acceleration of deep neural network training, the graphics processing unit (GPU) has become the mainstream platform. GPUs face substantial challenges on Graph Neural Networks (GNNs), such as workload imbalance and memory access irregularities, leading to underutilized hardware. Existing solutions such as PyG, DGL with cuSPARSE, and GNNAdvisor frameworks partially address these challenges. However, the memory traffic involved with Sparse-Dense Matrix Matrix Multiplication (SpMM) is still significant.

We argue that drastic performance improvements can only be achieved by the vertical optimization of algorithm and system innovations, rather than treating the speedup optimization as an "after-thought" (i.e., *(i) given a GNN algorithm, designing an accelerator, or (ii) given hardware, mainly optimizing the GNN algorithm*). In this paper, we present MaxK-GNN, an advanced high-performance GPU training system integrating algorithm and system innovation. (i) We introduce the MaxK nonlinearity and provide a theoretical analysis of MaxK nonlinearity as a universal approximator, and present the Compressed Balanced Sparse Row (CBSR) format, designed to store the data and index of the feature matrix after nonlinearity; (ii) We design a coalescing enhanced forward computation with row-wise product-based Sparse Matrix-Matrix Multiplication (SpGEMM) Kernel using CBSR for input feature matrix fetching and strategic placement of a sparse output accumulation buffer in shared memory; (iii) We develop an optimized backward computation with outer product-based and Sampled Sparse Matrix Dense Matrix Multiplication (SSpMM) Kernel.

We conduct extensive evaluations of MaxK-GNN and report the end-to-end system run-time. Experiments show that MaxK-GNN system could approach the theoretical speedup limit according to Amdahl's law. We achieve comparable accuracy to SOTA GNNs, but at a significantly increased speed: 3.22×/4.24× speedup (vs. theoretical limits, 5.52×/7.27×) on Reddit compared to DGL and GNNAdvisor implementations. Our implementation can be found on GitHub[1].

1 Introduction

Graph Convolutional Networks (GCNs), a specific type of Graph Neural Networks (GNNs), have garnered significant attention in recent years due to their unparalleled capability to extract latent information from graph data [1–3]. The field of GCNs manifests in a myriad of important practical applications, including the prediction of cascading power-grid failures [4], traffic forecasting [5], recommendation systems [6, 7], and drug discovery [8]. In the design and acceleration of GNN training, GPU platforms have become the prevalent choice. Conventional GCN acceleration processes a graph feature matrix (X) by multiplying it with a dense, small, weight matrix (W), followed by multiplying the resultant output with a highly sparse and irregular adjacency matrix (A) via Sparse-Dense Matrix Matrix Multiplication (SpMM) [9].



(a) GraphSAGE blocks (b) SAGEConv block (c) Time consumption breakdown

Figure 1. GraphSAGE structure analysis: latency breakdown of full-batch GraphSAGE training on the *ogbn-proteins* dataset over 30 epochs, with 256 hidden dimensions. GPU platform: Nvidia A100.

Addressing the demands for high-performance and efficient GNN systems has led to two primary research trends: algorithmic optimization and hardware-level enhancement. The former encapsulates methods such as graph reordering, e.g., GNNAdvisor [10], run-time community detection, e.g., I-GCN [11], and graph partitioning, e.g., GCoD [12]. Conversely, hardware-level approaches focus on workload balancing and efficient hardware mapping, with specific work tackling workload imbalance stemming from irregular input data with power-law distributed non-zero elements, e.g., AWB-GCN [13], FlowGNN [14], MergePath-SpMM [15], GROW [16], G-CoS [17], ENGN [18] [13–21].

Challenges. Despite the advancements, there are grant challenges. Many existing accelerators, including AWB-GCN [13] and GCoD [12], are FPGA based [11–13] or ASIC based [22,

[1]https://github.com/harveyp123/MaxK-GNN

23] which are typically not open-sourced, and are user-unfriendly. They require specialized hardware such as on-chip distribution networks and comprehensive graph preprocessing support to address the workload imbalance caused by SpMM (referred to as "evil rows") [13]. In comparison, existing GPU systems provide open-sourced and user-friendly implementations, however, they are still far from meeting performance limits. Using the profiling results of full batch GraphSAGE [3] training as an illustration, shown in Fig. 1. The computation and memory demands associated with the SpMM kernel are the major bottlenecks during the training process, contributing to over 83.6% of the total training time. More specifically, the GPU's multi-level memory hierarchy [24] and SpMM's usage of memory-efficient formats (e.g., compressed sparse row (CSR)) create difficulties in shared memory buffering design and hinder the exploitation of memory locality.

Research Gap. We summarize the root causes of the above inefficiencies as: (i) *Memory Traffic Challenges in GPU-based Frameworks:* Existing works adopt a row-wise multiplication approach which employs nonzero-grouping techniques, e.g., GNNAdvisor [10], thereby transferring atomic accumulation into shared memory which resides in a streaming multiprocessor (SM). Although this approach mitigates the cost of atomic accumulation in global memory, it still requires a substantial number of global memory transactions to access the input feature matrix, resulting in total memory traffic scaling linearly with the hidden dimension and number of nonzeros. The linear scaling with original hidden dimension dim_{origin} and nnz exacerbates this problem. MergePath [15] further resolves SpMM workload imbalance issues using a binary search-based warp mapping, but is less effective when the hidden dimension is large. (ii) *Algorithmic Limitations and Resource Waste:* Prevailing algorithmic methods, such as graph partitioning [25] and graph sampling [26], which are tailored to address large-scale graph training challenges, frequently lead to a reduction in accuracy [25] and accrue overhead in communication [25] and subgraph sampling, as well as redundant computing [26]. This architecture-oblivious workflow consistently results in inefficient hardware utilization. These gaps underline the pressing need for **sustainable** acceleration solutions.

Proposed Research. We argue that drastic performance improvements can only be achieved by the vertical integration and optimization of algorithms and system innovations. Our overarching goal is a fundamental shift. Rather than treating the sustainability optimization as an "after-thought" (i.e., *(i) given a GNN algorithm, designing an accelerator, or (ii) given a platform, primarily optimizing the GNN algorithm*), we propose a set of GNN paradigms that work cooperatively at both the algorithm and GPU system levels to deliver strong performance scaling. Our target is a high accuracy, high performance, and low latency GNN training system.

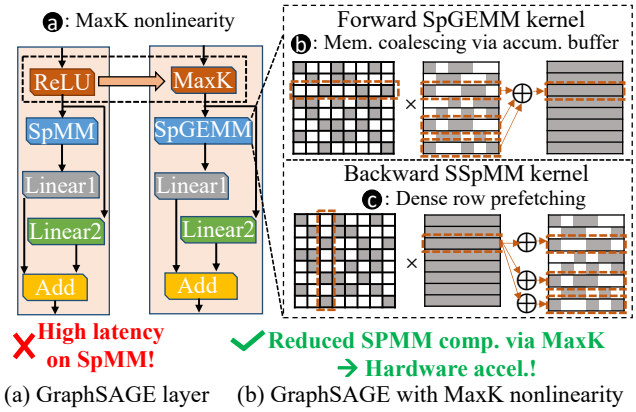


Figure 2. GraphSAGE layer example with (a) ReLU (b) MaxK nonlinearity. SSpMM: sampled sparse matrix dense matrix multiplication.

In this work, we introduce MaxK-GNN, an advanced GPU training system integrating algorithm and system innovations. Our design significantly outperforms the state-of-the-art (SOTA) GPU-based GNN training solutions, including GNNAdvisor [10] and DGL [9]. MaxK-GNN is strategically constructed on the PyTorch framework [27] for its front-end, and further extends the GPU's computational capabilities by customizing the MaxK nonlinearity to select the top-kth element for each node embedding and implementing innovative Sparse Matrix-Matrix Multiplication (SpGEMM) and Sampled Sparse Matrix Dense Matrix Multiplication (SSpMM) kernels using C++/CUDA.

The design of MaxK-GNN system is focused on three core **contributions**, also illustrated in Fig. 2:

ⓐ *Node-Balanced Feature Dimension Reduction through MaxK Nonlinearity:* We introduce the MaxK nonlinearity, and provide a theoretical analysis of MaxK nonlinearity as a universal approximator. We present the Compressed Balanced Sparse Row (CBSR) format, designed to store the data and index of the feature matrix after nonlinearity. This approach not only facilitates memory coalescing, but also significantly reduces traffic on the GPU platform. Experiments show that we can reduce the effective feature map dimension from 256 to 16 with a minor accuracy drop.

ⓑ *Coalescing Enhanced Forward Computation with Row-wise Product-Based SpGEMM Kernel:* This component encompasses: (i) the utilization of the CBSR format for right-hand matrix fetching, leading to a notable memory traffic reduction. For example, Reddit dataset with the original hidden dimension as 256 and MaxK k value as 16, can reduce the global memory traffic by 90.6% compared to SpMM. (ii) the strategic placement of a sparse output accumulation buffer in shared memory, enabling coalesced global memory accumulation on the output matrix, while maintaining the same accumulation efficiency as a conventional SpMM design.

ⓒ *Optimized Backward Computation with Outer Product-Based SSpMM Kernel Design:* This segment focuses on the acceleration of the computation pattern (sparse × dense = sparse). Leveraging a dense row prefetching technique, we effectively transfer irregular memory accesses from global memory to shared memory. The subsequent irregular shared memory fetching is facilitated using the CBSR index, followed by atomic accumulation of the CBSR data in global memory. The proposed SSpMM design ensures coalesced memory transactions across all stages, substantially reducing global memory consumption by more than 90% (Reddit dataset with original hidden dimension as 256 and k as 16).

We conduct extensive evaluations of MaxK-GNN system within the context of a single-GPU, full-batch, GNN training workload. We report the end-to-end system run-time rather than floating point operations per second (FLOPS) analysis. Experiments show that our MaxK-GNN system could approach the theoretical speedup limit according to Amdahl's law [28]. The performance gaps between our results and the theoretical limits, e.g., 3.22×/4.24× compared to 5.52×/7.27× for Reddit dataset using MaxK-GNN with GraphSAGE, is from the accumulation stage of SpGEMM and dense row prefetching stage of SSpMM, which we think are extremely difficult to further optimize, currently.

The introduced MaxK non-linearity and kernel design are not confined to the specific framework, but exhibit compatible with other SOTA GNN training systems, including PyG [29] and DGL [9]. Furthermore, the adaptability of these novel constructs aligns with current methods employed in graph partitioning [25, 30] and graph sampling techniques [26].

2 Background and Related work

2.1 Graph Convolution Network

Graph Convolutional Networks (GCNs) [2] are stacks of GCNConv layers. An example of a GCNConv layer is shown in Fig. 3. We define a graph $G = (\mathcal{V}, \mathcal{E}, A)$ which contains $|\mathcal{V}|$ nodes and $|\mathcal{E}|$ edges. The adjacency matrix A has the shape of $(|\mathcal{V}| \times |\mathcal{V}|)$, usually with high sparsity. Each non-zero entry $A(i, j)$ corresponds to an edge between i and j. Each node is associated with an $\mathcal{F}$-dimensional feature embedding vector, and $X \in \mathbb{R}^{|\mathcal{V}| \times \mathcal{F}}$ represents the feature embedding matrix for all nodes. The forward propagation of the l-th GCNConv layer can be split into 2 stages: (1) linear transformation $Y^l = X^l W^l$ and (2) feature aggregation $X^{l+1} = \sigma(A' Y^l)$. Where $X^l \in \mathbb{R}^{|\mathcal{V}| \times \mathcal{F}_l}$ is the feature embedding matrix at the l-th layer for all nodes, $W^l \in \mathbb{R}^{\mathcal{F}_l \times \mathcal{F}_{l+1}}$ is the weight matrix for linear transformation which will be learned during the GCN training. The feature aggregation stage calculates the feature embedding matrix for the next layer, where $A' \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ is the normalized and regularized adjacency matrix, σ is the activation function, typically element-wise ReLU. Different varients of GCNs, such as GraphSage [31] and Graph Isomorphism Network (GIN) [32], use similar structure and can reuse the same forward propagation abstraction as GCNs.

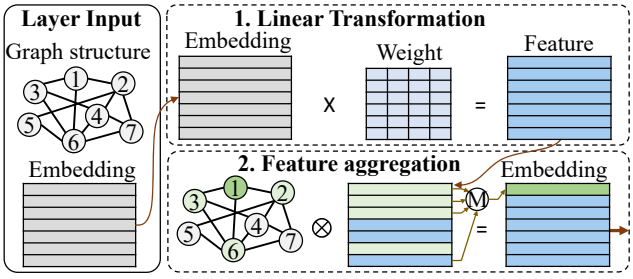


Figure 3. Computational workflow of single GCNConv layer.

2.2 GNN Acceleration

The PyTorch Geometric (PyG) software stack [29] and similar proposals like HP-GNN [33], LL-GNN [34], and FlowGNN [14] utilize message-passing primitives such as *scatter* and *reduce* for GNN training on GPUs and FPGA overlays. These primitives incur substantial memory and storage overheads, leading to inefficiency and poor memory bandwidth utilization. None of these approaches effectively enhance workload balance or address data locality issues within GNN training and inference.

Several existing open-source GPU and FPGA acceleration frameworks are aimed at enhancing GNNs. GNNAdvisor [10] uses warp-level partitioning for distributed neighborhood workload but may cause load imbalance and its kernel performance, mainly improved by the Rabbit order [35], doesn't outperform cuSPARSE [36]. MergePath [15] addresses SpMM workload imbalance with a binary search algorithm, but its efficacy decreases with feature dimensions over 128, common in large graphs. Flow-GNN [14] accelerates GNNs on FPGA platforms but fails to address workload imbalance or provide scalability to larger core count.

Several methods have been developed to tackle large graph problems, such as graph neighborhood and boundary sampling. Betty [26] offers a novel sampler to alleviate memory bottlenecks, and BNS-GCN [25] uses boundary sampling for multi-GPU and multi-node systems; yet both fail to address the SpMM bottleneck. Other research has utilized generalized SpMM for GNN inference acceleration, including AWB-GCN [13], which dynamically balances workload, and I-GCN [11], which enhances locality and reduces off-chip memory access. However, the implementation requires specialized hardware and is not applicable to GPU system.

2.3 Introducing Sparsity in GNN Training

Dropout: As a well-known regularization to prevent overfitting, dropout introduces feature sparsity during training by randomly setting a fraction of input units to 0 [37]. This

operation results in a form of sparsity that is highly irregular and challenging to leverage in an end-to-end hardware design.

Weight Sparsification: Two prevailing weight sparsification approaches in GNN training are *train-and-prune* and *sparse training* [38]. The former optimizes weight parameters to improve inference speed, as exemplified by methods like ADMM-based pruning [39] and LTH-based pruning [40]. The overall training cost (including pretraining) is usually much higher compared to the original model training. Conversely, *sparse training* initiates with a sparsified weight matrix and updates sparse weight locations at specific iterations. The standard scheduler, *drop and grow*, includes techniques like SET [41], RigL [42], and SNFS [43]. Such sparsification in GNN workload, however, introduces irregular patterns that inhibit efficient hardware deployment [44].

Nonlinearity for Sparsification: Nonlinear functions such as ReLU [45] introduce sparsity into the graph training. As detailed in FATReLU [46], adjusting the ReLU threshold can induce greater feature sparsity. Similar to other sparsity forms, this irregularity does not align with hardware characteristics, yielding limited speedup on the graph training system.

3 MaxK-GNN Dataflow

We start with introducing MaxK nonlinearity, and discuss how it can benefit GNN training system in MaxK-GNN dataflow.

3.1 MaxK Nonlinearity as a Universal Approximator

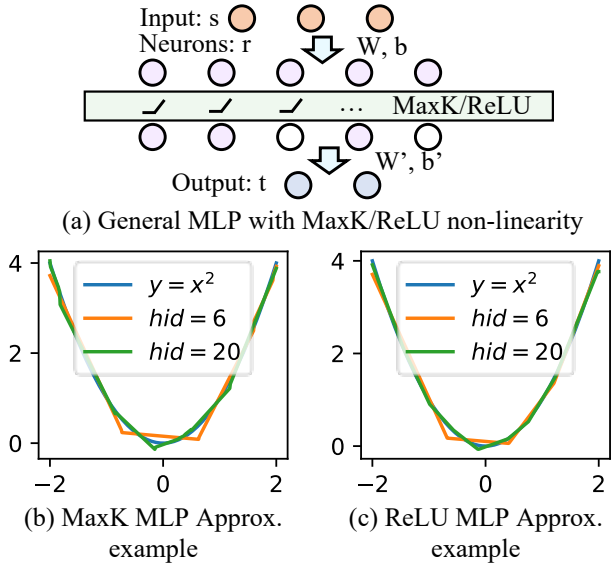
Conventional ReLU operators, which are frequently utilized in GNN architectures, result in an irregularly sparsified feature matrix, thereby hindering its usage for hardware acceleration. To address this challenge, we introduce the MaxK nonlinearity.

MaxK Nonlinearity Definition: *(i) During the forward propagation, MaxK nonlinearity is computed on node-wise feature map to get the maximum k_{th} element and set the rest to 0. (ii) During the backward propagation, the feature gradient uses same feature sparsity pattern as induced in forward.*

$$h(X) = \textit{max-k}_{j\in[1,r]}(X \cdot W + b)_j \tag{1}$$

In addition, the MaxK nonlinear operator is positioned before the SpMM operator, which serves to diminish the computational and memory access overhead associated with SpMM. This nonlinearity also exhibits superior generalization ability in both transductive and inductive graph learning settings. It introduces a regularized sparsity pattern, enabling a more efficient design for hardware acceleration on GNNs.

MaxK, like ReLU, is a piece-wise linear (PWL) function. We use Multilayer Perceptron (MLP) with MaxK nonlinearity to prove that it is a universal approximator. As demonstrated in Eq. 1 and as illustrated in Fig. 4(a), MaxK is applied to the feature map. The input X has a size s, W has dimensions $[s, r]$, with r being the hidden dimension. MaxK maintains the maximum k significant value out of r while preserving the same shape. MaxK preserves the same input and output dimensions and introduces a regularized sparsity pattern, thereby facilitating the design of the supporting hardware. MaxK can approximate any continuous function $f(X)$ of $X \in \mathbb{R}^s$ with a sufficient number of hidden units r.



(a) General MLP with MaxK/ReLU non-linearity

(b) MaxK MLP Approx. example

(c) ReLU MLP Approx. example

Figure 4. MLP with MaxK and ReLU non-linearity. $y = x^2$ function approximation example with different number of hidden units.

Proposition 3.1. *Given any positive integers s and t, two parameter groups W and W' are determined such that $g(X)$ is expressed as a linear combination of r convex PWL functions:*

$$g(X) = h(X) \cdot W' + b' \tag{2}$$

In this equation, $g(X)$ operates as the neural network approximator and denotes the continuous PWL function with r locally affine regions on $\mathbb{R}^s$. Thus, any continuous PWL function $g(X)$ can be represented as a linear combination of PWL functions $h(X)$ (proof can be found in [47, 48]).

Theorem 3.2. MLP with MaxK Serves as a Universal Approximator. *A MaxK network $g(X)$ with r hidden units can approximate any continuous function $f(X)$ on a compact domain $C \subset \mathbb{R}^s$ with an arbitrarily small approximation error ϵ. In particular, as $\epsilon \to 0$, it follows that $r \to \infty$.*

Universal Approximator Proof: According to the Stone-Weierstrass approximation theorem [49], a PWL function $g(X)$ can approximate any continuous function $f(X)$ with an error ϵ: $|f(v) - g(v)| < \epsilon$. The PWL function $g(X)$, as provided in Proposition 3.1, is composed of r convex PWL function $h(X)$. By configuring a sufficiently large number of

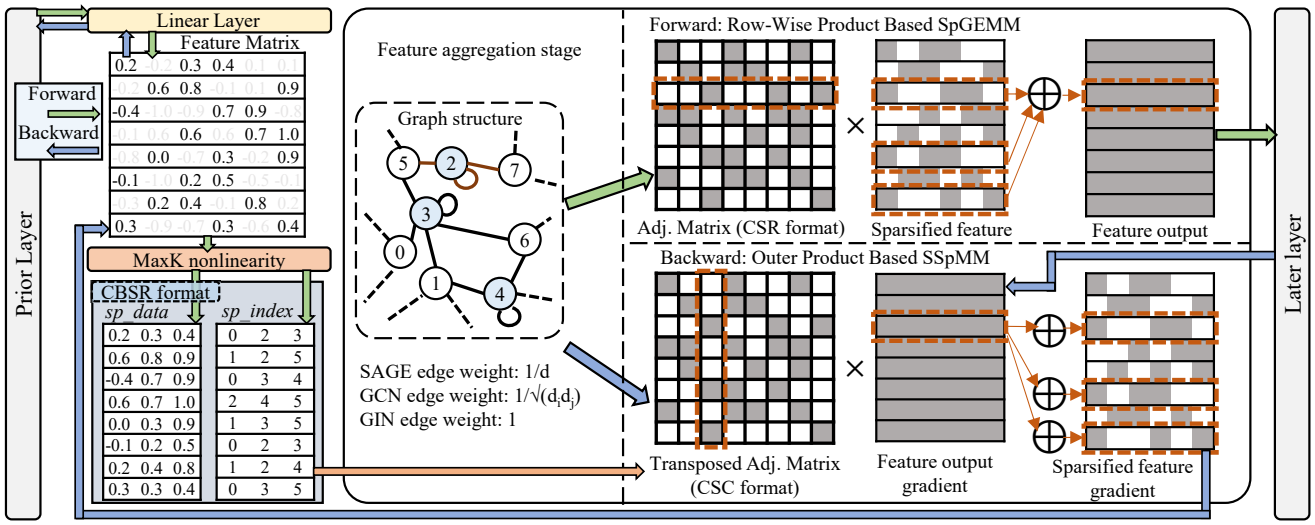


Figure 5. Training dataflow of single MaxK based GNN layer. In the backward computation, the transposed CSC format is equal to original CSR format.

hidden units r and appropriate k values in the MaxK network $g(X)$, the desired approximation error ϵ can be achieved. Consequently, it can be concluded that a MaxK network with r hidden units can provide an arbitrarily close approximation to $f(X)$ on the compact domain $C \subset \mathbb{R}^s$.

Refer to Fig. 4(b) for an illustration of a single-layer MaxK-based network employed for the approximation of the function $y = x^2$. A standard backpropagation algorithm is used to train the neural network until convergence, with the number of hidden units varied to observe the approximation error. For the MaxK nonlinearity, the top $\lceil hid/4 \rceil$ elements are selected and the remainder set to 0. It can be readily observed that as the number of hidden units increases, the approximation error of the MaxK-based neural network approximator decreases.

Key takeaway: *The MaxK-GNN system introduces the regularized sparsity of the embedding feature matrix, thereby considerably speeding up GNN models' SpMM operation. Notably, MaxK is a nonlinearity and does not compromise the precision of the models significantly.*

3.2 MaxK-GNN Training Dataflow

Traditional GNN layers adopt similar backward and forward SpMM computation designs. However, our proposed MaxK-GNN framework incorporates asymmetric forward and backward paths, leveraging MaxK nonlinearity. MaxK nonlinear operator is positioned before the SpMM operator, sparsifying the forward computation path from SpMM to SpGEMM. Similarly, MaxK nonlinearity also increases sparsity of the backward computation path to SSpMM. Such a process significantly reduces the computational and memory overhead associated with matrix multiplications. To leverage this newly introduced embedding sparsity, we present modified forward and backward propagation dataflow and generate a reference kernel design tailored for MaxK-GNN. Fig. 5 illustrates the dataflow of single-layer GCN training with the proposed MaxK nonlinearity, for the sake of simplicity.

Given a GNN layer with single linear and aggregation operations, the forward and backward processes of the aggregation stage are expressed in Eq. 3, where A is the adjacent list and $h(X_{l-1})$ denotes the feature map post linear layer and the MaxK nonlinearity. A could have different expressions according to aggregator type. For instance, the SAGEConv uses $1/d$ (d is the node degree) for the mean aggregator. The backward process inherents the sparsity pattern as we only compute the gradient of $h(X_{l-1})$ non-zero elements.

$$X_l = A \cdot h(X_{l-1}), \quad \frac{\partial L}{\partial h(X_{l-1})} = A^T \cdot \frac{\partial L}{\partial X_l} \tag{3}$$

Forward Computation. For the forward computation, we propose employing a Compressed Balanced Sparse Row (CBSR) format to capitalize on the newly introduced sparsity of the feature output resulting from the MaxK layer. The CBSR format allows for contiguous memory accesses on sparsified feature matrix and improves the system memory bandwidth utilization while mitigating workload imbalance. This format comprises two components: a data segment (*sp_data*) and an index segment (*sp_index*), which are stored in two adjacent memory blocks in the main memory. The next step involves the execution of forward feature aggregation, accomplished by multiplying the graph adjacency list by the sparsified feature matrix. This computation

utilizes a row-wise product-based SpGEMM scheme [50], whereby $X_l[i,:] = \sum_{j=0}^{J} A[i,j] \cdot h(X_{l-1})[j,:]$. Assuming a dense output obviates the costly ESC overhead [51] usually encountered with SpGEMM design. During the row-wise product operation, each element from the left-hand row is multiplied by its corresponding elements in the right-hand row, with the result then accumulated to the output row. This procedure enables the sparsified output accumulation to occur within the on-chip cache, offering significantly lower latency compared to global memory-based accumulation.

Backward Computation. During the feature aggregation's backward SSpMM process, the transposed adjacency matrix, in a CSC format (equivalent to CSR employed in forward), is multiplied with the feature output gradient to yield the sparsified feature gradient. Compared with standard SpGEMM computations, this backward SSpMM computation exhibits a (sparse × dense = sparse) operation. Given that the output sparsity pattern (*sp_index*) aligns with that of the forward process, the backward SpGEMM only requires to compute corresponding data (*sp_data*) located by (*sp_index*).

However, the row-wise product-based multiplication for this computation could lead to substantial irregular global memory access on $\frac{\partial L}{\partial X_l}$ (from Eqn. 3), as elements must be fetched according to the sparse index *sp_index*. To mitigate this, we propose prefetching rows from $\frac{\partial L}{\partial X_l}$ to the on-chip memory, enabling irregular access within the cache, thereby avoiding uncoalesced global memory traffic.

Consequently, we adopt an outer product-based [50] SSpMM method for the backward computation process, where

$$\frac{\partial L}{\partial h(X_{l-1})}[:,:] = \sum_{j=0}^{J} A^T[:,j] \cdot \frac{\partial L}{\partial X_l}[j,:] \tag{4}$$

Each element of the left-hand column is multiplied by a single row and accumulated to the respective output row. This strategy ensures efficient utilization of memory and alleviates the irregular memory access issue.

4 GPU System Support for MaxK-GNN

4.1 Forward SpGEMM GPU Kernel

Compared to conventional SpMM, our proposed CBSR format incorporates a sparsified input embedding matrix, with the aim of reducing both computational and memory demands. Sparse input matrices correlate to an increase in irregular memory accesses to both the input matrices (adjacency matrix and embedding matrix) which could degrade kernel computational, memory efficiency. To counteract the lack of spatial locality in the CSR formatted adjacency list and the CBSR formatted embedding matrix, we use a *warp-level partitioning scheme*. Coupled with an on-chip buffering mechanism, this approach can achieve warp-level balance and coalesced global memory accesses, significantly improving computational efficiency.

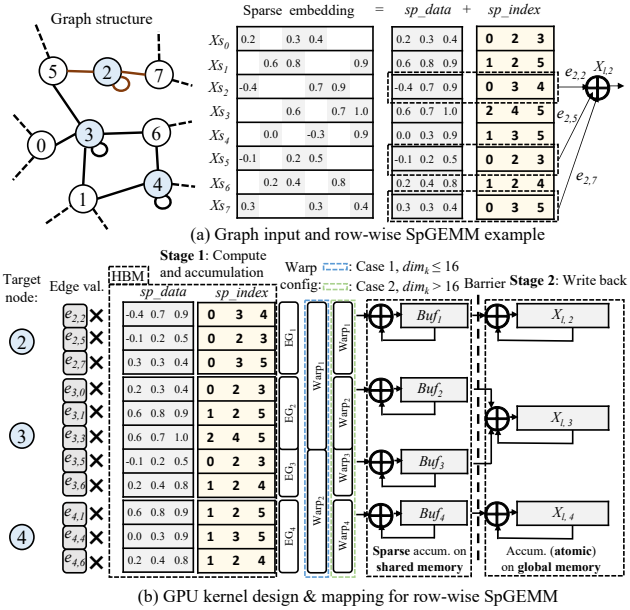


Figure 6. Forward computation kernel with $dim_{origin} = 6$ and $dim_k = 3$. Xs_i: ith row (node) of input embedding Xs, represented in the CBSR format. $X_{l,i}$: ith row (node) of output embedding X_l.

Design Overview. Fig. 6(a) shows an illustration of the row-wise SpGEMM computation with the CBSR format. The computation of node 2's output embedding is based on the multiplication and accumulation of neighboring embeddings (Xs_2, Xs_5, Xs_7) and corresponding adjacency list edge values ($e_{2,2}$, $e_{2,5}$, $e_{2,7}$). The accumulation leverages the index *sp_index* to map multiplication results to the appropriate output positions, which consequently results in a sparse memory access pattern. Therefore, we buffer the partial accumulation result in the on-chip shared memory to mitigate uncoalesced global memory transactions.

The on-chip buffering considerations are encapsulated in the kernel design pseudocode provided in Algorithm 1. The overview workflow is divided into two stages: i) a compute and accumulation stage, and ii) a write back stage. Suppose the original dimensions of the right matrix X_s are denoted as dim_{origin} and the MaxK value selected results in dim_k, leading to a CBSR formatted matrix. This matrix has *sp_data* and *sp_index*, each with a size of $N \times dim_k$. Within the computation and accumulation kernel, the coalesced global memory transactions fetch both *sp_data* and *sp_index*. The parallel multiplication and sparsified accumulation within the warp are conducted within Buf_w, a buffer located in the shared memory. Eventually, Buf_w is atomically added into X_l in global memory using coalesced accesses. This design strategy promotes an efficient memory access pattern, optimizing computational parallelism while conserving memory bandwidth.

Algorithm 1 Forward Computation Kernel Pseudocode

```
for all rows A_row_i in A do
  for all warp partitions P_w in A_row_i do
    Initialize Buf_w in shared memory;
    Form m threads within a warp;
    for each nonzero element e_{i,j} in P_w do
      for all thread_k in warp do
        // Multiply and sparse accumulation to Buf_w, mapped by sp_index
        Buf_w[sp_index[j,k]] += e_{i,j} × sp_data[j,k];
      end for
    end for
  end for
  Reorganize all threads by natural warps;
  for all Buf_w in shared memory do
    // Atomically accumulation with coalesced global memory access
    X_{l,i} += Buf_w;
  end for
end for
```

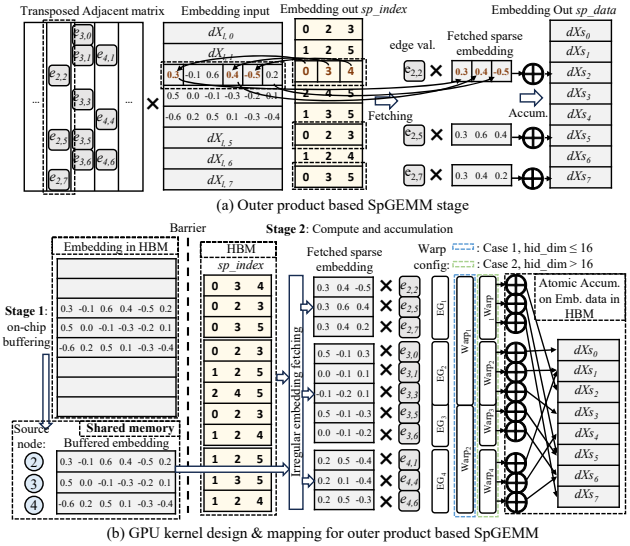


Figure 7. Backward computation kernel with $dim_{origin} = 6$ and $dim_k = 3$. $dX_{l,i}$: ith row (node) of dense input embedding dX_l. dXs_i: ith row (node) of output embedding dXs represented in the CBSR format. Transposed adjacent matrix A^T in the CSC format has same storage format as the original adjacent matrix A in CSR format, thus no extra storage is required.

Warp Level Partition. Illustrated in Algorithm 1, the SpGEMM workload requires a workload-to-warp mapping strategy. Herein, we delve into the warp-level workload partitioning and allocation. We propose a light-weight warp-level partition mapper that operates at $O(n)$ complexity with n being the number of nodes.

As shown in Fig. 6(b), each edge $e_{i,j}$ involved in the computation constitutes a workload unit. Within such a unit, $e_{i,j}$ undergoes a multiplication with the sparse row sp_data_j, followed by accumulation in the buffer Buf_w, indexed by sp_index_j. Subsequently, the workload of each adjacency matrix row A_{row_i} is segmented into Edge Groups (EGs). Each EG reserves a chunk of shared of (dim_{origin}×4) bytes to serve as an intermediate buffer for sparse accumulation.

The workload of each adjacency matrix row A_{row_i} is firstly segmented into Edge Groups (EGs). To optimize EG execution within the computation and accumulation phase, we integrate the hidden dimension into our warp mapping strategy. In scenarios where $dim_k \leq 16$ (as seen in Case 1 of Fig.6(b)), each standard warp comprises $\lfloor \frac{32}{dim_k} \rfloor$ EG workloads, EG is limited to be within the same warp to circumvent memory access conflicts that could occur if an EG straddles multiple warps. In contrast, if $dim_k > 16$ (Case 2 in Fig.6(b)), an EG is processed by a single warp executed iteratively. The execution of each EG is performed by the corresponding warp, with the results aggregated at respective locations in the shared memory buffer, following the indices from sp_index.

In the next phase (stage 2 in Fig. 6), data from each shared memory buffer Buf_w is atomically accumulated into its corresponding X_l output in global memory. Retaining the thread organization of natural warps due to identical dimensions of the shared memory buffers and output embeddings (dim_{origin}), each warp cyclically processes a single row, ensuring an efficient and coalesced computational structure.

4.2 Backward SSpMM GPU Kernel

In the backward computation, we execute a specialized SSpMM operation involving $A^T \times dX_l$ to generate dXs, in CBSR format. Inheriting sp_index from Xs used in the forward computation, the computation needs only sp_data of dXs, signifying a (sparse × dense = sparse) operation with a known output sparse pattern, thereby requiring only data locations indexed by sp_index. A naive row-wise product-based kernel could lead to significant uncoalesced global memory transactions, thereby inhibiting data parallelism.

Design Overview. We propose utilizing an outer product-based SSpMM process to enhance global memory coalescing and computational parallelism. An illustrative dataflow is presented in Fig. 7(a). Here, the third column of A^T, denoted by ($e_{2,2}$, $e_{2,5}$, $e_{2,7}$), multiplies with the third input node embedding, $dX_{l,2}$, indexed by sp_index (represented as $sp\_index_{row\_2}$, $sp\_index_{row\_5}$, and $sp\_index_{row\_7}$). The resulting values are subsequently accumulated into the corresponding output embedding data sp_data (expressed as dXs_2, dXs_5, dXs_7). The transposed adjacency matrix, A^T, is represented in CSC format, mirroring the data structure of the original matrix A in CSR format. Consequently, this approach requires no additional memory for storing the backward gradient computation, thereby optimizing memory utilization. Considering

the irregular indexing produced by sp_index, buffering the dense embedding row dX_l in on-chip memory proves advantageous, as it enhances bandwidth and reduces latency, due to more regular memory access.

Algorithm 2 Backward Computation Kernel Pseudocode

```
for all rows dX_{l,row_i} in dX_l do
  for all workload partitions P_w in A^T_{col_i} do
    // Coalesced global memory read.
    Load dX_{l,row_i} into Buf_w in shared memory;
    Form m threads within a warp
    for each nonzero element e_{i,j} in P_w do
      for all thread_k in warp do
        // Collect data from the buffer Buf_w, indexed by sp_index, and multiply it by e_{i,j}. Subsequently, perform an automatic accumulation to sp_data, ensuring coalesced global memory access.
        sp_data[i,k] += e_{i,j} × Buf_w[sp_index[i,k]]
      end for
    end for
  end for
end for
```

Our proposed design for backward computation kernel, presented in Algorithm 2, encompasses two primary stages. The **first stage** involves loading the dense embedding dX_{l,row_i} into the shared memory buffer Buf_w for each warp. It is crucial to note that this stage facilitates coalesced and continuous global memory transactions, optimizing memory access patterns. The **secondary stage** amalgamates sparse fetching, computation, and atomic accumulation. Sparse fetching encompasses two operations: a) fetching sp_index through coalesced global memory transactions, and b) irregular indexing within the shared memory buffer Buf_w. The fetched vector subsequently undergoes multiplication with the corresponding edge values $e_{i,j}$. Finally, the result is atomically accumulated in the global memory embedding data section sp_data, a coalesced memory transaction that ensures computational efficiency.

Warp Level Partitioning. To enhance computational efficiency and ensure warp-level workload balance, we advocate for an edge-centric grouping process, analogous to the scheme used for the forward SpGEMM. IT is a lightweight process that can be seamlessly applied during the graph loading and preprocessing stage, preserving overall resource efficiency.

During the dense embedding loading stage, each warp fetches the corresponding row of the embedding matrix, dX_{l,row_i}, into shared memory through coalesced global memory access. The required shared memory allocation per warp for floating-point numbers is $dim_{origin} \times 4$ bytes, mirroring the allocation used in the forward workflow. To expedite loading, we utilize a natural warp organization, with each warp iteratively managing its corresponding row. Afterward, all warps arrive at a synchronization barrier.

During the compute and accumulation stage, workload units working with edges and the hidden dimension are reconfigured into EGs, adopting the same partitioning procedure used in the forward computational workflow. In cases where $dim_k \leq 16$ (refer to Case 1 in Fig. 7(b)), each conventional warp manages $\lfloor \frac{32}{dim_k} \rfloor$ EGs, confining each EG to a single warp to prevent shared memory access conflicts. For $dim_k > 16$ (Case 2 in Fig. 7(b)), each EG is handled by a single warp using a loop function. Operations involving sparse fetching, computation, and atomic accumulation are performed within these warps, as outlined in Algorithm 2. This stage exclusively involves coalesced memory read/write operations, thus preserving the efficiency of global memory transactions.

4.3 Memory System

In our design, the NVIDIA GPU's shared memory is strategically utilized to mitigate uncoalesced memory accesses and to ensure that all global memory accesses are coalesced. The memory system is structured to store the CSR-formatted adjacent matrix, the embedding matrix, and the CBSR-formatted sparse embedding matrix in global memory (HBM), while the intermediate shared memory serves as a buffer for partial results and sparse fetching. In this section, we examine the global memory transactions for both the forward SpGEMM and the backward SSpMM kernels.

Forward SpGEMM. During the forward SpGEMM computation, the bulk of the computation and sparse fetching is focused on the accumulation process within the shared memory. By implementing a row-wise product-based SpGEMM kernel design, the CBSR-formatted X_s rows are read nnz times, leading to a total global memory traffic of $(4 \times 2 \times dim_k \times nnz)$ bytes for floating-point data and integer index. With smaller dim_{origin}, utilizing uint8 for sp_index allows a reduction in total traffic to $(5 \times dim_k \times nnz)$ bytes. Compared to a row-wise SpMM kernel design, the total **global memory traffic reduction** is calculated as $[(4 \times dim_{origin} - 5 \times dim_k) \times nnz]$ bytes, indicating that lower values of dim_k yield greater reductions.

Additionally, the output atomic accumulation in our proposed SpGEMM kernel aligns with the original row-wise SpMM kernel, where the number of global memory atomic accumulations is given by $(N \times dim_{origin} \times \frac{avgdeg}{w})$, and $avgdeg$ is derived as $\frac{nnz}{N}$ with w representing the hyperparameter for the maximum workload units assigned to an EG.

Backward SSpMM. The backward SSpMM begins with an on-chip buffering stage, allowing the buffered feature gradient row $dX_{l,i}$ to be read only once per SSpMM computation, equivalent to $(N \times dim_{origin})$ memory transactions. Subsequent stages require reading the corresponding rows from sp_index for sparse fetching, and during the compute

Table 1. Graph datasets number of nodes and number of edges information.

Graph Name	# Nodes	# Edges	Graph Name	# Nodes	# Edges	Graph Name	# Nodes	# Edges	Graph Name	# Nodes	# Edges
am	881,680	5,668,682	amazon0505	410,236	4,878,874	amazon0601	403,394	5,478,357	artist	50,515	1,638,396
citation	2,927,963	30,387,995	collab	235,868	2,358,104	com-amazon	334,863	1,851,744	DD	334,925	1,686,092
ddi	4,267	2,135,822	Flickr	89,250	989,006	ogbn-arxiv	169,343	1,166,243	ogbn-products	2,449,029	123,718,280
ogbn-proteins	132,534	79,122,504	OVCAR-8H	1,889,542	3,946,402	ppa	576,289	42,463,862	PROTEINS_full	43,466	162,088
pubmed	19,717	99,203	ppi	56,944	818,716	Reddit	232,965	114,615,891	SW-620H	1,888,584	3,944,206
TWITTER-Partial	580,768	1,435,116	Yeast	1,710,902	3,636,546	Yelp	716,847	13,954,819	youtube	1,138,499	5,980,886

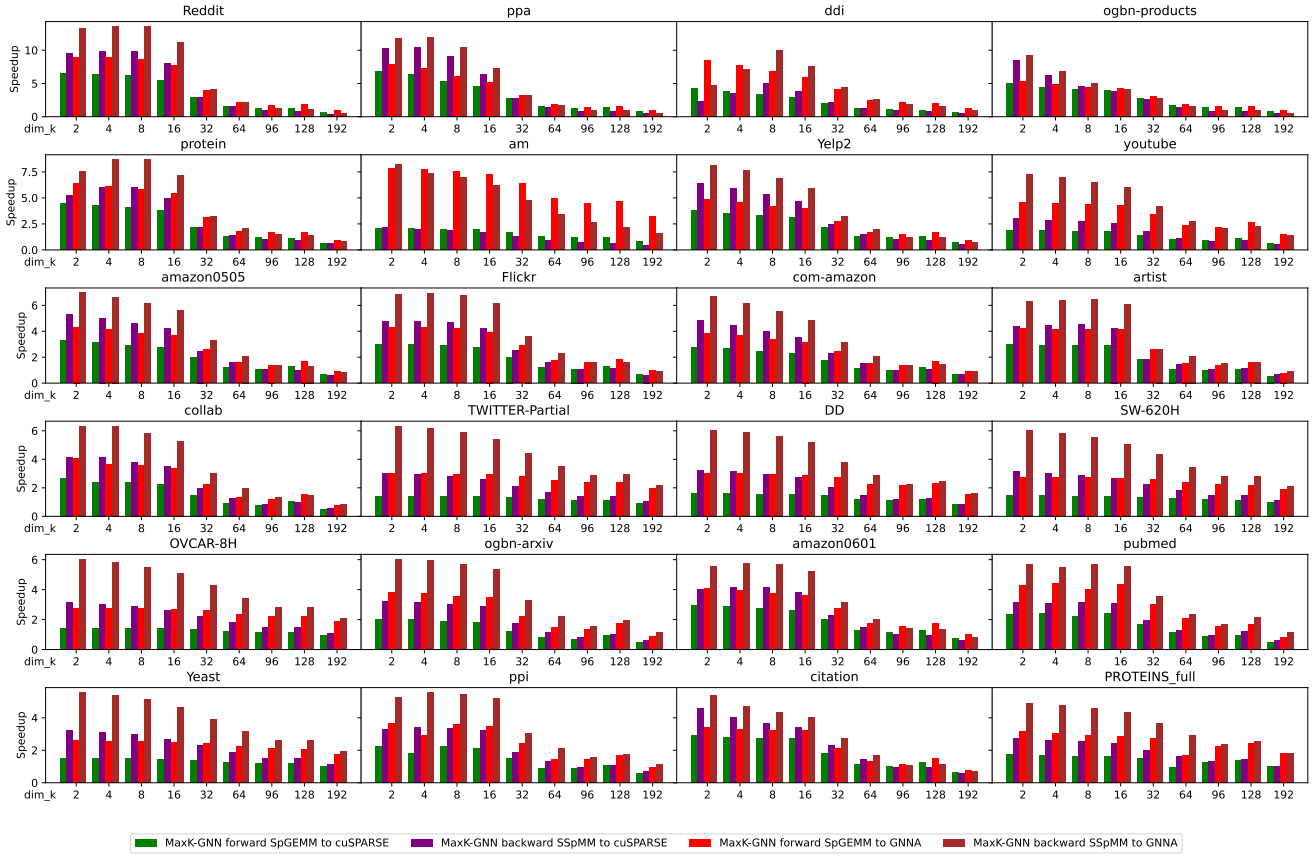


Figure 8. Forward SpGEMM and backward SSpMM speedup over SPMM kernel from cuSPARSE [36] and GNNAdvisor [10]. Original hidden dimension size is 256, we vary dim_k (k value of MaxK) to evaluate kernel speedup.

and accumulation stages, each workload unit performs single read and write operations for its corresponding row in sp_data. Consequently, the total read and write transactions to global memory are approximately $(4 \times N \times dim_{origin} + 5 \times dim_k \times nnz)$ and $(4 \times dim_k \times nnz)$ bytes respectively, when considering uint8 sp_index. Compared to a naive outer product-based SpMM, the **global memory traffic reduction** is $[(4 \times dim_{origin} - 5 \times dim_k) \times nnz]$ for reads and $[(4 \times dim_{origin} - 4 \times dim_k) \times nnz]$ for write transactions, reaffirming that a lower dim_k leads to higher reductions.

4.4 Kernel Profiling

To demonstrate the effectiveness of the proposed design, We provide profiling of SpGEMM and SSpMM kernels, while the experiment setup is outlined in Sec. 5. For similicity, we use Reddit graph as an example and provide the memory system profiling result shown in Table 2. We evaluate the compute kernels by employing the Nsight Compute profiler to generate performance metrics for cuSPARSE SpMM, SpGEMM, and SSpMM kernels when executed on the Reddit graph. Table 2 reports data on the traffic between the L2 cache and global memory, as well as the hit rates for the L1 and L2 caches.

The reported memory traffic reduction of the proposed SpGEMM and SSpMM kernel aligns with the theoretical analysis given in Section 4.3. The proposed MaxK nonlinearity and corresponding kernel support reduces total global memory traffic by close to 86% / 76%, when reducing original

Table 2. MaxK-GNN memory system profiling

dim_org = 256 dim_k = 32	SpMM	SpGEMM	SSpMM
Total Traffic (GB)	100.64	14.54	24.43
L1 cache hit rate (%)	1.44	21.77	28.14
L2 cache hit rate (%)	19.39	34.03	30.03
Peak memory bandwidth utilization (%) (read/write)	94.5/0.41	63.5/1.08	40.2/30.6

hidden dimension from 256 to k as 32. While the traffic is reduced significantly, the bandwidth utilization of read/write is not reduced significantly, as such we are able to achieve **2.9x/2.98x** speedup over cuSPARSE SpMM.

It is worth mentioning that the L1/L2 cache hit rates of cuSPARSE SpMM kernel, our forward SpGEMM kernel, and our backward SSpMM kernel are 1.44%/19.39%, 21.77%/34.03%, and 28.14%/30.30%, respectively. The L1 cache hit rate of our kernels is significantly higher than that of cuSPARSE SpMM kernel, which is due to our rational use of the multi-level memory hierarchy of the GPU.

5 Evaluation

We offer a robust and insightful assessment of MaxK-GNN, highlighting its performance advantages and applicability in the broader context of graph-based learning and computation. Our evaluation strategy begins with an in-depth analysis of both the forward SpGEMM and the backward SSpMM kernels. We carefully assess these components under a range of K values, comparing the performance with existing implementations such as SpMM, as found in GNNAdvisor [10] and cuSPARSE v12.0 [36]. Following this detailed kernel-level examination, we present our end-to-end evaluation. This encapsulates both the accuracy and speedup metrics to showcase the efficacy of the MaxK-GNN framework in addressing general graph learning problems.

5.1 Experimental Setup

Datasets. For SpGEMM & SSpMM kernel benchmark, we select popular benchmark datasets that have been extensively employed in previous studies [10, 29, 31, 53–56]. See Table 1 for more details.

For end-to-end MaxK-GNN system evaluation, we benchmark five datasets that range from small to medium-scale graphs. Specifically, the selected datasets are Flickr [57], for the categorization of image types based on descriptors and shared attributes; Yelp [58], for the classification of user-generated reviews pertaining to businesses and services; Reddit [59], for community prediction using posts' content and users' comments; ogbn-products [1], for Amazon product classification via customer reviews; and ogbn-proteins [1], for the prediction of protein function presence. This suite of datasets not only covers a broad spectrum of applications but also facilitates the understanding of how MaxK-GNN performs under various scenarios.

Models. For SpGEMM & SSpMM kernel benchmarks, we employ an original hidden dimension of 256. This evaluation involves an assessment of the MaxK sparsified feature matrix across a set of k values, including $[2, 4, 8, 16, 32, 64, 96, 128, 192]$. The selected parameters ensure a comprehensive examination of the impact of various sparsity levels on the system's performance.

For the end-to-end evaluation of the MaxK-GNN system, we integrate MaxK nonlinearity with three graph models: GraphSAGE [31], GCN [2], and GIN [52]. The GIN model [52] is noteworthy for its unique aggregation function, serving as a reference for advanced GNNs such as Graph Attention Networks (GAT) [60]. To ensure fairness, all models are trained in full batch graph learning mode with the MEAN aggregator for GraphSAGE. The detailed parameter settings of MaxK-GNN are given in Table 3. Our evaluation spans $k = [2, 4, 8, 16, 32, 64, 96, 128, 192]$ to find the best trade-off between model accuracy and system speedup, highlighting the versatility of our approach.

Table 3. MaxK-GNN training setup

Dataset	Flickr	Yelp	Reddit	OGB	
				products	proteins
Layers	3	4	4	3	3
Hid. dim.	256	384	256	256	256
Epochs	400	3000	3000	500	1000
LR/Drop	0.001/0.2	0.001/0.1	0.01/0.5	0.003/0.5	0.01/0.5

Environment Setup. We construct the MaxK-GNN by implementing the SpGEMM & SSpMM kernels using C++ and CUDA C. The front-end is built with Python/Pytorch to provide a user-friendly interface. Our principal evaluation platform consists of a high-performance server equipped with 32-core AMD EPYC 7513 CPU and state-of-the-art NVIDIA A100 80GB GPU [61]. This computing environment is utilized for both kernel-level and system-level evaluations.

For the performance measurement of the SpGEMM & SSpMM kernels, we conduct a rigorous analysis by calculating the average latency across ***1000 runs***. This ensures that the observed performance metrics are consistent and representative. In the context of the comprehensive evaluation of the MaxK-GNN system, our approach is further extended. We measure the latency of the training phase with epochs given in Table 3, including both forward and backward propagation, 50 times, and subsequently compute the average.

5.2 MaxK-GNN Kernel Evaluation

We presented forward SpGEMM and backward SSpMM kernel evaluation in Fig. 8. Both SpGEMM and SSpMM kernels exhibit a significant speedup compared to the SpMM kernels from cuSPARSE [36] and GNNAdvisor [10]. Note that the

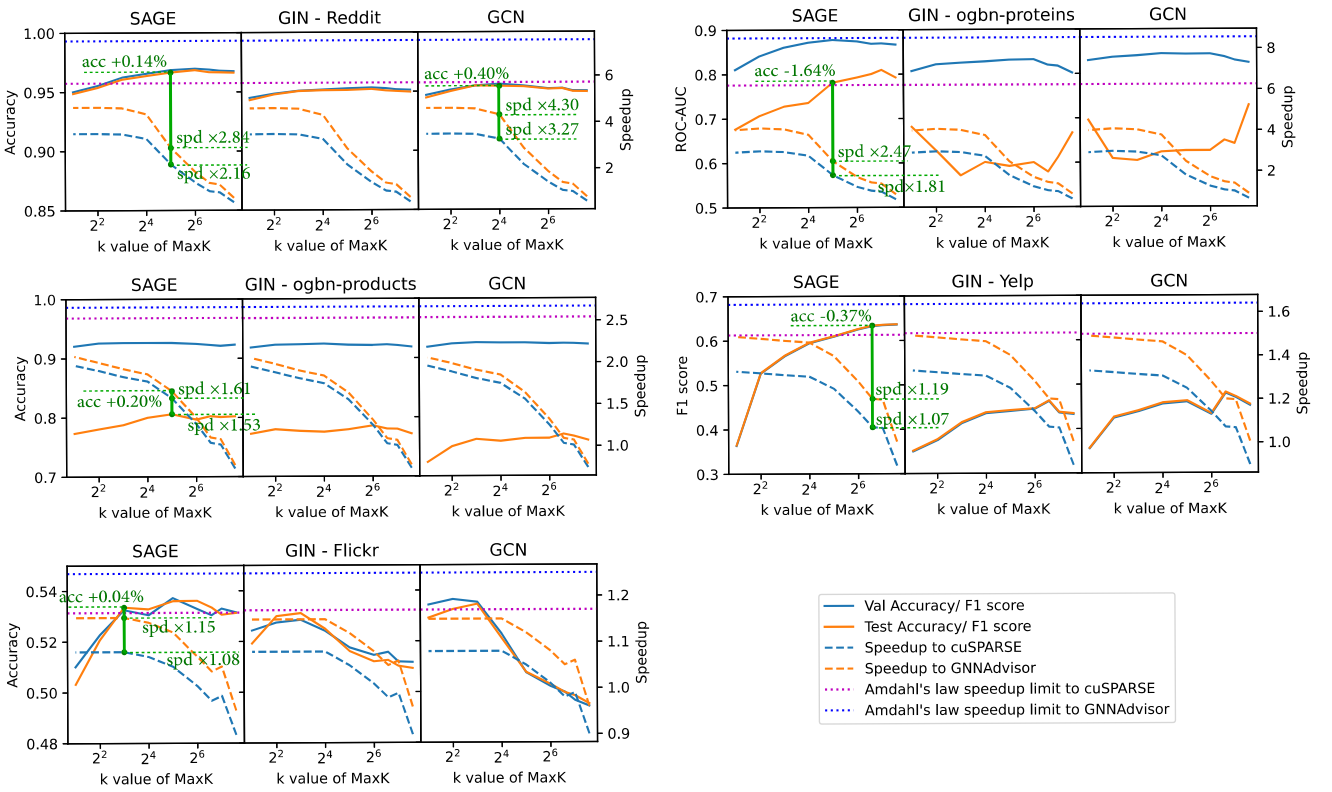


Figure 9. End-to-end MaxK-GNN system evaluation for GraphSAGE [31], GCN [2], and GIN [52] models and Reddit, ogbn-proteins, ogbn-produces, Yelp, and Flickr datasets. MaxK nonlinearity setting: $k = [2, 4, 8, 16, 32, 64, 96, 128, 192]$. We compare the end to end training speedup over DGL with cuSPARSE [9] framework and GNNAdvisor [10] implementation. Speedup: spd.

original hidden dimension is 256 and we vary k values for the benchmark. Overall, the result shows that as k decreases, the speedup increases.

For the SpGEMM kernel, when k is decreased to a certain extent, such as 8, the output accumulation stage becomes the performance bottleneck, hence a further decrease in k leads to a speedup saturation. Nevertheless, our SpGEMM kernel exhibits impressive acceleration performance, especially on graphs with larger average degrees. For graphs with an average degree greater than 50, such as ogbn-proteins, ddi, Reddit, ppa, and ogbn-products, the average speedup of the SpGEMM kernel at $k = 8, 16, 32, 64$ is 4.63×, 4.15×, 2.54×, 1.46×, respectively, as compared to the cuSPARSE [36], and 6.39×, 5.71×, 3.50×, 2.02×, respectively, as compared to the GNNAdvisor [10], demonstrating its high acceleration efficiency. When $k \leq 128$, the SpGEMM kernel can effectively bring speedup to 92.2% of all test cases compared to cuSPARSE [36], and 100% of all test cases compared to GNNAdvisor [10]. The result demonstrates its high generalization.

The backward SSpMM kernel fully exploits the global memory access coalescing and traffic reduction by combining the outer product-based approach with dense row prefetching. The SSpMM kernel achieves better speedup performance than the forward SpGEMM kernel. For graphs with average degrees greater than 50, the average speedup of the SSpMM kernel at $k = 8, 16, 32, 64$ is 6.93×, 5.39×, 2.55×, 1.46× respectively, as compared to the cuSPARSE [36] and 9.57×, 7.46×, 3.55×, 2.04×, respectively, as compared to the GNNAdvisor [10]. Given that a MaxK-GNN layer training pipeline requires computing both forward SpGEMM and the backward SSpMM, the SSpMM kernel speedup can also benefit the end-to-end training speedup. When $k \leq 128$, the SSpMM kernel can effectively bring speedup to 87.5% of all test cases compared to cuSPARSE [36], and 98.4% of all test cases compared to GNNAdvisor [10]. It also exhibits significant versatility.

5.3 End-to-end MaxK-GNN Evaluation

MaxK Nonlinearity Kernel. For MaxK nonlinearity kernel implementation, we customize a high-performance pivot-based $k-th$ value selection kernel. The kernel buffers each node's embedding in shared memory and finds the *min* and *max* values Then the kernel selects a pivot equal to $(min + max)/2$ and counts the number of elements that are

greater than the pivot. The algorithm iterates based on the pivot value until the number of elements (that are greater than the pivot) is equal to k. The feature map distribution is not fully randomized, and follows a normal distribution. With 256 original hidden dimensions, we observe that the pivot based algorithm usually converges when running feature map MaxK selection in less than 10 iterations. Given that the comparison operations and pivot selection are conducted in shared memory, the total global memory traffic would be similar to element-wise operations such as ReLU. The overall MaxK nonlinearity kernel has an average cost of less than 2% of the SpGEMM kernel run time. We provide an example of kernel run-time on the Reddit within Table 4.

Table 4. MaxK nonlinearity kernel profiling

dim_org = 256 dim_k = 32	SpMM	SpGEMM	SSpMM	MaxK
Latency (ms)	44.98	15.49	15.07	0.261

Morever, the MaxK nonlinearity kernel is applied during forward path, and the sparse matrix index can be shared with backward propagation process. We need to "recompress" feature into CBSR format for each GNN layer during the forward path. However, the MaxK nonlinearity kernel has little overhead compared to SpMM and SpGEMM/SSpMM, and it will not become the critical path during the training pipeline.

Accuracy and Speedup. To comprehensively evaluate the MaxK-GNN framework, we conduct experiments using three representative GNN models: GraphSAGE[31], GCN[2], and GIN[52]. We use five diverse datasets to benchmark performance. The ReLU-based **baseline** model's setting and accuracy utilized in our evaluation section is aligned with the SOTA full-batch training accuracy. Specifically, our baseline performance matches the results presented in Table 3 of reference [62] and the GraphSAGE row in Table 4 of reference [25]. We test MaxK-GNN system with $k = [2, 4, 8, 16, 32, 64, 96, 128, 192]$, as shown in Figure 9. In the figure, we provide speedup limit lines calculated using Amdahl's law [28]: $S = 1/(1 - p\_SpMM)$, where S is the theoretical speedup limit and p_SpMM represents the percentage of execution time taken by the SpMM operator within the full GNN training pipeline. This allows us to contextualize the empirical speedups achieved by MaxK-GNN.

The theoretical speedup limits attainable differ between datasets. Reddit and ogbn-proteins allow greater theoretical speedup due to their characteristics. Using a lower k value with these datasets leads to a slight accuracy decline but permits substantial system speedup exceeding 3x with a suitable k value selection. The ogbn-produces, Yelp, and Flickr datasets have relatively lower theoretical speedup limits. For these datasets, MaxK-GNN achieves 1.1-2× speedup without significant accuracy loss. Lowering k values trades off some accuracy for larger speedups on datasets with higher theoretical limits like Reddit and ogbn-proteins. However, even on datasets with lower speedup limits, MaxK-GNN provides 1.1-2× speedups with minimal accuracy impact.

We select the best performing k values from MaxK-GNN framework, aiming to further investigate the relationship between accuracy and system speedup. We compare the results against a ReLU-based baseline GNN model, which has been implemented in the DGL framework [9]. The results are encapsulated in Table 5.

GraphSAGE (SAGE) on Reddit has a theoretical speedup limit of 5.52×/7.27× compared to cuSP./GNNA. respectively, following Amdahl's law [28]. We utilize MaxK-GNN with $k = 32$ and attain speedup factors of 2.16×/2.84×, resulting in enhanced accuracy for the GraphSAGE model. In cases where MaxK-GNN is implemented with $k = 16$, speedup factors of 3.48×/4.58× are achieved with the GCN model setting, while simultaneously elevating the accuracy by 0.44%. GraphSAGE (SAGE) on Yelp possesses a lower Amdahl's law speedup limit, 1.46×/1.59×, compared to cuSP./GNNA. respectively. The dataset requires a relatively higher k value to uphold accuracy performance. With the original hidden dimension of the Yelp dataset being 384, MaxK-GNN with GraphSAGE & $k = 96$ achieves 1.07×/1.19× speedup relative to cuSP./GNNA. baselines, while maintaining comparable accuracy. The Flickr dataset also manifests a lower Amdahl's law speedup limit, which is 1.16×/1.24× compared to cuSP./GNNA. respectively. However, MaxK-GNN with GraphSAGE & $k = 8$ acquires a 1.08×/1.15× speedup, accompanied by greater accuracy.

The results collectively show that our MaxK-GNN system could approach the theoretical speedup limit. The performance gaps between our results and Amdahl's law theoretical limits, i.e., 3.22×/4.24× compared to 5.52×/7.27× for Reddit dataset using GraphSAGE, is from the essential accumulation stage of SpGEMM and dense row prefetching stage of SSpMM, which are extremely difficult to further optimize.

Further Discussion on Accuracy. In Table 5 and Figure 9, we follow the standard train/val/test split setting and obtain average accuracy over five random seeds for graph training. While most models and datasets demonstrate stable behavior, exceptions occur, notably with the ogbn-proteins dataset when using GCN/GIN models. Upon detailed examination, we attribute the inconsistent behavior observed in the ogbn-proteins dataset to inherent characteristics of the dataset itself. Specifically, within a certain range of the convergence region for the ogbn-proteins dataset, we observe high variance in test accuracy, which in turn leads to unstable ROC-AUC performance. Importantly, this observation is not exclusive to the MaxK-GNN model; we have also identified similar instability in the baseline models without MaxK.

K **Value Selection.** Empirically, we could select $k = 32$ for 256 original hidden dimension to align similar accuracy with ReLU baseline model and obtain significant kernel speedup. Such K selection corresponds to 87.5% feature sparsity.

Table 5. End to end MaxK-GNN accuracy & speedup evaluation and comparison with ReLU based baseline model implemented in DGL [9]

dataset		Reddit			ogbn-proteins			ogbn-products			Yelp			Flickr		
model	method	k	Acc (%)	Latency (ms/epoch) Speedup (cuSP./GNNA.)	k	AUC	Latency (ms/epoch) Speedup (cuSP./GNNA.)	k	Acc (%)	Latency (ms/epoch) Speedup (cuSP./GNNA.)	k	F1 score	Latency (ms/epoch) Speedup (cuSP./GNNA.)	k	Acc (%)	Latency (ms/epoch) Speedup (cuSP./GNNA.)
SAGE	baseline	-	96.51	54.9 (1x/1.32x)	-	0.7976	23.4 (1×/1.37×)	-	80.39	133.6 (1×/1.05×)	-	0.6376	36.5 (1×/1.11×)	-	53.31	3.52 (1×/1.06×)
	MaxK-GNN	32	96.65	25.5 (2.16×/2.84×)	64	0.7928	18.6 (1.25×/1.71×)	32	80.59	87.1 (1.53×/1.61×)	96	0.6339	34.2 (1.07×/1.19×)	32	53.6	3.35 (1.05×/1.12×)
	MaxK-GNN	16	96.37	17 (3.22×/4.24×)	32	0.7812	12.9 (1.81×/2.47×)	16	80	77.9 (1.72×/1.80×)	32	0.61	29.6 (1.23×/1.37×)	8	53.35	3.26 (1.08×/1.15×)
GCN	baseline	-	95.02	54.5 (1×/1.32×)	-	0.646	23.2 (1×/1.37×)	-	76.58	129.6 (1×/1.05×)	-	0.4718	34.3 (1×/1.12×)	-	49.78	3.42 (1×/1.06×)
	MaxK-GNN	16	95.42	16.7 (3.27×/4.30×)	16	0.6236	8.43 (2.75×/3.77×)	32	76.34	83.2 (1.56×/1.64×)	96	0.4819	32.0 (1.07×/1.20×)	8	53.45	3.17 (1.08×/1.15×)
	MaxK-GNN	8	95.46	15.7 (3.48×/4.58×)	2	0.6958	7.94 (2.92×/4.00×)	8	76.21	71.6 (1.81×/1.91×)	32	0.4628	27.5 (1.25×/1.40×)	4	53.25	3.16 (1.08×/1.15×)
GIN	baseline	-	95.07	54.6 (1×/1.32×)	-	0.583	23.3 (1×/1.37×)	-	77.79	130.7 (1×/1.05×)	-	0.4578	34.9 (1×/1.12×)	-	50.78	3.35 (1×/1.07×)
	MaxK-GNN	16	95.11	16.7 (3.27×/4.32×)	4	0.6277	7.81 (2.98×/4.07×)	8	77.69	72.7 (1.80×/1.89×)	96	0.464	32.7 (1.07×/1.20×)	8	53.11	3.10 (1.08×/1.15×)
	MaxK-GNN	8	95.05	15.8 (3.47×/4.57×)	2	0.6812	7.97 (2.92×/3.99×)	4	77.98	69.8 (1.87×/1.97×)	32	0.4422	28.1 (1.24×/1.39×)	4	52.99	3.09 (1.08×/1.15×)

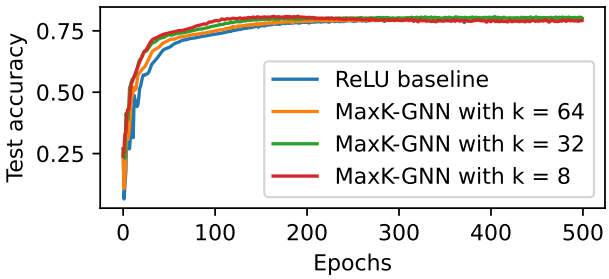


Figure 10. Convergence curves of full-batch training on ogbn-product dataset for (i) ReLU baseline model (ii) MaxK-GNN with $k = 64$ (iii) MaxK-GNN with $k = 32$ (iv) MaxK-GNN with $k = 8$

Convergence Analysis of MaxK-GNN. To examine the convergence performance of the MaxK-GNN training, we show a case study on the ogbn-products dataset with a full batch setting. The results in Fig. 10 show that the MaxK-GNN training, specifically at $k = 64$, $k = 32$ and $k = 8$, demonstrates convergence behavior similar or even better than the baseline model employing ReLU nonlinearity. With a lower k value, the convergence speed is slightly faster.

6 Conclusion and Future Work

Recent advancements in MLaaS (Machine Learning as a Service) have primarily centered on accelerating inference on various platforms [38, 63–76]. In this paper, we present MaxK-GNN, a high-performance GPU training system integrating algorithm and system innovation. (i) We introduce the MaxK nonlinearity and provide a theoretical analysis of MaxK nonlinearity as a universal approximator, and present the Compressed Balanced Sparse Row (CBSR) format, designed to store the data and index of the feature matrix after nonlinearity; (ii) We design a coalescing enhanced forward computation with row-wise product-based SpGEMM Kernel using CBSR for input feature matrix fetching and strategic placement of a sparse output accumulation buffer in shared memory; (iii) We develop an optimized backward computation with outer product-based and SSpMM Kernel. Experiments show that our MaxK-GNN system could approach the theoretical speedup limit according to Amdahl's law. We achieve comparable accuracy to existing GNNs, but at a significantly increased speed: 3.22×/4.24× speedup (vs. theoretical limits, 5.52×/7.27×) on Reddit.

The proposed MaxK nonlinearity could be potentially expanded to more DNN architectures such as CNNs and Transformers, to provide regularly sparsified feature map for acceleration.

7 Acknowledgments

This research was supported in part by the Northeastern University Institute for Experiential AI, the NSF IUCRC Center for Hardware and Embedded Systems Security and Trust (CHEST), NSF CNS-2247893, the Semiconductor Research Corporation (SRC) Artificial Intelligence Hardware program, and Advanced Micro Devices (AMD).

References

[1] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.

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

[3] Jielun Liu, Ghim Ping Ong, and Xiqun Chen. Graphsage-based traffic speed forecasting for segment network with sparse data. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[4] Yuxiao Liu, Ning Zhang, Dan Wu, Audun Botterud, Rui Yao, and Chongqing Kang. Guiding cascading failure search with interpretable graph convolutional network. *Computing Research Repository (CoRR) in arXiv*, abs/2001.11553, 2020.

[5] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *Expert Systems with Applications*, page 117921, 2022.

[6] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)*, 2020.

[7] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[8] Pietro Bongini et al. Molecular generative graph neural networks for drug discovery. *Neurocomputing*, 450:242–252, 2021.

[9] Amazon AWS. Deep Graph Library (DGL). Retrived from https://www.dgl.ai/. Accessed: 2019, May 1.

[10] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, 2021.

[11] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1051–1063, 2021.

[12] Haoran You, Tong Geng, Yongan Zhang, Ang Li, and Yingyan Lin. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 460–474. IEEE, 2022.

[13] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.

[14] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. Flowgnn: A dataflow architecture for universal graph neural network inference via multi-queue streaming. *arXiv preprint arXiv:2204.13103*, 2022.

[15] Mohsin Shan, Deniz Gurevin, Jared Nye, Caiwen Ding, and Omer Khan. Mergepath-spmm: Parallel sparse matrix-matrix algorithm for graph neural network acceleration. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 145–156. IEEE, 2023.

[16] Ranggi Hwang, Minhoo Kang, Jiwon Lee, Dongyun Kam, Youngjoo Lee, and Minsoo Rhu. Grow: A row-stationary sparse-dense gemm accelerator for memory-efficient graph convolutional neural networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 42–55. IEEE, 2023.

[17] Yongan Zhang, Haoran You, Yonggan Fu, Tong Geng, Ang Li, and Yingyan Lin. G-cos: Gnn-accelerator co-search towards both better accuracy and efficiency. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[18] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Transactions on Computers*, 70(9):1511–1525, 2020.

[19] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)*, 54(9):1–38, 2021.

[20] Shengwen Liang, Cheng Liu, Ying Wang, Huawei Li, and Xiaowei Li. Deepburning-gl: an automated framework for generating graph neural network accelerators. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[21] Adam Auten, Matthew Tomei, and Rakesh Kumar. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[22] Mingyu Yan et al. Hygcn: A gcn accelerator with hybrid architecture. In *HPCA*, 2020.

[23] Ilkwon Byun, Dongmoon Min, Gyu-hyeon Lee, Seongmin Na, and Jangwoo Kim. Cryocore: A fast and dense processor architecture for cryogenic computing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 335–348. IEEE, 2020.

[24] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[25] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. *Proceedings of Machine Learning and Systems*, 4:673–693, 2022.

[26] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. Betty: Enabling large-scale gnn training with batch-level graph partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 103–117, 2023.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8026–8037, 2019.

[28] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

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

[30] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 257–266, 2019.

[31] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[32] Yifan Hou, Jian Zhang, James Cheng, Kaili Ma, Richard TB Ma, Hongzhi Chen, and Ming-Chang Yang. Measuring and improving the use of graph information in graph neural networks. In *International Conference on Learning Representations*, 2019.

[33] Yi-Chien Lin, Bingyi Zhang, and Viktor Prasanna. Hp-gnn: Generating high throughput gnn training implementation on cpu-fpga heterogeneous platform. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 123–133, 2022.

[34] Bingyi Zhang, Hanqing Zeng, and Viktor Prasanna. Low-latency mini-batch gnn inference on cpu-fpga heterogeneous platform. *arXiv preprint arXiv:2206.08536*, 2022.

[35] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed*

Processing Symposium (IPDPS), pages 22–31. IEEE, 2016.

[36] Nvidia. Cuda sparse matrix library (cusparse). developer.nvidia.com/cusparse.

[37] Pierre Baldi and Peter J Sadowski. Understanding dropout. *Advances in neural information processing systems*, 26, 2013.

[38] Hongwu Peng, Deniz Gurevin, Shaoyi Huang, Tong Geng, Weiwen Jiang, Orner Khan, and Caiwen Ding. Towards sparsification of graph neural networks. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 272–279. IEEE, 2022.

[39] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. A systematic dnn weight pruning framework using alternating direction method of multipliers. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 184–199, 2018.

[40] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[41] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12, 2018.

[42] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *ICML*, pages 2943–2952. PMLR, 2020.

[43] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

[44] Tianlong Chen, Yongduo Sui, Xuxi Chen, Aston Zhang, and Zhangyang Wang. A unified lottery ticket hypothesis for graph neural networks. In *International Conference on Machine Learning*, pages 1695–1706. PMLR, 2021.

[45] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[46] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Nir Shavit, and Dan Alistarh. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In *International Conference on Machine Learning*, pages 5533–5543. PMLR, 2020.

[47] Shuning Wang. General constructive representations for continuous piecewise-linear functions. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 51(9):1889–1896, 2004.

[48] Ian Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. In *International conference on machine learning*, pages 1319–1327. PMLR, 2013.

[49] Louis De Branges. The stone-weierstrass theorem. *Proceedings of the American Mathematical Society*, 10(5):822–824, 1959.

[50] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780. IEEE, 2020.

[51] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):1–20, 2015.

[52] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[53] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[54] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

[55] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels. 2016.

[56] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[57] Julian McAuley and Jure Leskovec. Image labeling on a network: Using social-network metadata for image classification. 07 2012.

[58] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2020.

[59] Jure Leskovec William L. Hamilton, Rex Ying. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[60] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[61] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[62] Zirui Liu, Chen Shengyuan, Kaixiong Zhou, Daochen Zha, Xiao Huang, and Xia Hu. Rsc: accelerate graph neural networks training via randomized sparse computations. In *International Conference on Machine Learning*, pages 21951–21968. PMLR, 2023.

[63] Xi Xie, Hongwu Peng, Amit Hasan, Shaoyi Huang, Jiahui Zhao, Haowen Fang, Wei Zhang, Tong Geng, Omer Khan, and Caiwen Ding. Accel-gcn: High-performance gpu accelerator design for graph convolution networks. *arXiv preprint arXiv:2308.11825*, 2023.

[64] Hongwu Peng, Shaoyi Huang, Tong Zhou, Yukui Luo, Chenghong Wang, Zigeng Wang, Jiahui Zhao, Xi Xie, Ang Li, Tony Geng, et al. Autorep: Automatic relu replacement for fast private network inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5178–5188, 2023.

[65] Suiyao Chen, Jing Wu, Naira Hovakimyan, and Handong Yao. Recontab: Regularized contrastive representation learning for tabular data. In *NeurIPS 2023 Second Table Representation Learning Workshop*, 2023.

[66] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, and Tri Dao. Medusa: Simple framework for accelerating llm generation with multiple decoding heads. https://github.com/FasterDecoding/Medusa, 2023.

[67] Kiran Thorat, Jiahui Zhao, Yaotian Liu, Hongwu Peng, Xi Xie, Bin Lei, Jeff Zhang, and Caiwen Ding. Advanced language model-driven verilog development: Enhancing power, performance, and area optimization in code synthesis. *arXiv preprint arXiv:2312.01022*, 2023.

[68] Suiyao Chen, Nan Kong, Xuxue Sun, Hongdao Meng, and Mingyang Li. Claims data-driven modeling of hospital time-to-readmission risk with latent heterogeneity. *Health care management science*, 22:156–179, 2019.

[69] Hongwu Peng, Ran Ran, Yukui Luo, Jiahui Zhao, Shaoyi Huang, Kiran Thorat, Tong Geng, Chenghong Wang, Xiaolin Xu, Wujie Wen, et al. Lingcn: Structural linearized graph convolutional network for homomorphically encrypted inference. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[70] Suiyao Chen, Lu Lu, Yisha Xiang, Qing Lu, and Mingyang Li. A data heterogeneity modeling and quantification approach for field pre-assessment of chloride-induced corrosion in aging infrastructures. *Reliability Engineering & System Safety*, 171:123–135, 2018.

[71] Hongwu Peng, Shanglin Zhou, Yukui Luo, Nuo Xu, Shijin Duan, Ran Ran, Jiahui Zhao, Shaoyi Huang, Xi Xie, Chenghong Wang, et al. Rrnet: Towards relu-reduced neural network for two-party computation based private inference. *arXiv preprint arXiv:2302.02292*, 2023.

[72] Jing Wu, Ran Tao, Pan Zhao, Nicolas F Martin, and Naira Hovakimyan. Optimizing nitrogen management with deep reinforcement learning and crop simulations. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1712–1720, 2022.

[73] Hongwu Peng, Shaoyi Huang, Shiyang Chen, Bingbing Li, Tong Geng, Ang Li, Weiwen Jiang, Wujie Wen, Jinbo Bi, Hang Liu, et al. A length adaptive algorithm-hardware co-design of transformer on fpga through sparse attention and dynamic pipelining. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1135–1140, 2022.

[74] Zheyu Yan, Yifan Qin, Wujie Wen, Xiaobo Sharon Hu, and Yiyu Shi. Improving realistic worst-case performance of nvcim dnn accelerators through training with right-censored gaussian noise. *arXiv preprint arXiv:2307.15853*, 2023.

[75] Hongwu Peng, Shanglin Zhou, Yukui Luo, Nuo Xu, Shijin Duan, Ran Ran, Jiahui Zhao, Chenghong Wang, Tong Geng, Wujie Wen, et al. Pasnet: Polynomial architecture search framework for two-party computation-based secure neural network deployment. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[76] Yi Sheng, Junhuan Yang, Lei Yang, Yiyu Shi, Jingtong Hu, and Weiwen Jiang. Muffin: A framework toward multi-dimension ai fairness by uniting off-the-shelf models. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.