Investigating Node-Balanced Feature Dimension Reduction through Max-Pooling Non-linearity

Pang-Li Yang
py2236@nyu.edu

You-Jun Chen
yc7093@nyu.edu

Abstract

Graph Neural Networks (GNNs) are powerful tools for learning on graph-structured data, with applications in areas such as recommendation systems and drug discovery. However, the inherent sparsity in both graph features and edges presents significant challenges, including workload imbalance and irregular memory access, which can result in underutilized hardware. In this work, we focus on optimizing the most time-consuming part of GNN training—the forward propagation layer. We propose using max-pooling to regularize activations and reduce feature dimensionality, improving GPU resource utilization and accelerating computations. Additionally, we adopt warp-level partitioning to address workload imbalance, ensuring a more even distribution of the computational load across processing units. Our experiments show that the k-max-pooling approach outperforms the top-k method used in MaxK-GNN in terms of speed. The next step will be to evaluate the impact of this approach on accuracy and further investigate how max-pooling for feature reduction affects model performance.

1 Introduction

Graph Neural Networks (GNNs) have emerged as powerful tools for learning on graph-structured data, with applications spanning recommendation systems [12], social network analysis, and drug discovery. However, as GNNs scale to larger graphs, the increasing complexity of computations presents significant challenges, particularly in terms of memory and computational efficiency. Additionally, the inherent sparsity in both graph features and edges often leads to uneven workload distribution across computational threads, making it difficult to fully utilize the parallel processing capabilities of GPUs. This unpredictability hampers optimal resource utilization, limiting the scalability and efficiency of GNNs. Addressing this challenge could unlock substantial improvements in GNN model training, enabling faster computations and more efficient use of available hardware.

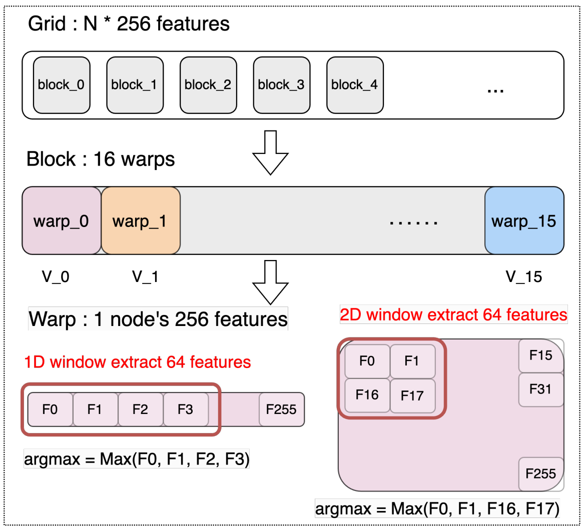
Grid : N * 256 features
block_0
block_1
block_2
block_3
block_4
...
Block : 16 warps
warp_0
warp_1
......
warp_15
V_0
V_1
V_15
Warp : 1 node's 256 features
2D window extract 64 features
1D window extract 64 features
F0
F1
F2
F3
F255
argmax = Max(F0, F1, F2, F3)
F0
F1
F16
F17
F15
F31
F255
argmax = Max(F0, F1, F16, F17)

Figure 1: Warp-Level Partition in Max-Pooling Kernel

In this paper, we propose a novel approach targeting the most time-consuming components of GNN training—specifically the forward kernels. By feeding regular-sized data into these kernels, we optimize GPU resource utilization, such as memory and registers, making them more predictable and efficient. To further improve GNN performance, we adapt insights from MaxK-GNN [11] to tackle workload imbalance during forward propagation through warp-level partitioning. This method ensures a more even distribution of computational load across processing units, thereby enhancing GPU utilization and reducing processing time.

Drawing inspiration from Convolutional Neural Networks (CNNs) [8], where pooling operations enhance speed and accuracy by discarding irrelevant features, we introduce a max-pooling technique for reducing the dimensionality of node features in GNNs. Specifically, max-pooling operations are well-suited for GPU parallelism, enabling efficient computation and accelerating the processing of large-scale graphs. Our experimental results show that the k-max-pooling approach performs significantly faster than the top-k method used in MaxK-GNN. Additionally, the adopted warp-level partitioned kernel outperforms cuSPARSE when the k-value is less than 80% of the original feature dimension.

By combining k-max-pooling with warp-level partitioning, we demonstrate substantial reductions in computation time, especially when processing a large number of features. This work offers new insights into optimizing GNNs for both sparse activation regularization and efficient workload balancing, leading to significant improvements in training time and scalability for large-scale graph data. Future work will focus on training with real-world graph data to evaluate the generalizability and accuracy of the proposed method.

2 Literature Survey

2.1 Graph Convolution Networks

Graph Convolution Networks (GCNs) [5], a specific type of Graph Neural Networks(GNNs), consist of stacks of GCN-Convolution layers. An example of a GCN-Convolution layer is shown in Figure 2.

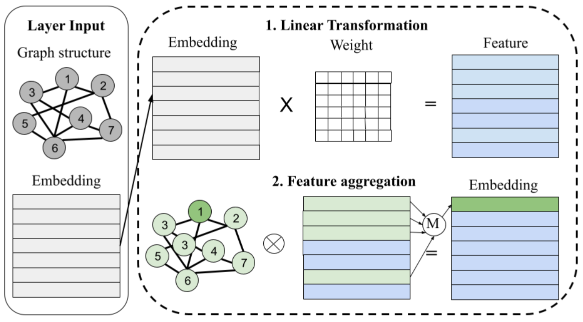

Figure 2: Computational Workflow of a Graph Convolution Network Layer

We define a graph $G = (\mathcal{V}, \mathcal{E}, A)$ which contains $|\mathcal{V}|$ nodes and $|E|$ edges. The adjacency matrix A has the shape of $(|V| \times |V|)$, 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 F -dimensional feature embedding vector, and $X \in \mathbb{R}^{|V| \times F}$ represents the feature embedding matrix for all nodes.

The forward propagation of the l-th GCN-Convolution layer has two stages. The first stage is linear transformation:

$$Y^l = X^l W^l \tag{1}$$

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 second stage is feature aggregation:

$$X^{l+1} = \sigma(A'Y^l) \tag{2}$$

which calculates the feature embedding matrix for the next layer, X^{l+1}. A' is the normalized and regularized adjacency matrix, and σ is the activation function, typically element-wise ReLU.

Different variation of GCNs, such as GraphSage [3] and Graph Isomorphism Network (GIN) [13], use similar structure and can reuse the same forward propagation abstraction as GCNs .

2.2 Introducing Sparsity in GNN Training

Dropout Adding a dropout layer is an effective regularization technique to prevent overfitting by randomly setting a fraction of the input to zero [2]. However, this introduces a form of sparsity that is highly irregular, making it difficult to exploit effectively in system or hardware design.

Non-linearity Functions Similar to dropout, non-linearity functions like ReLU [1] naturally introduce sparsity into graph training. The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{3}$$

However, the location of the non-zero elements is only known during inference time, which makes this form of sparsity irregular and misaligned with hardware characteristics. As a result, it yields limited speedup in training systems.

Weight Reduction and Model Compression To reduce the computational and memory costs of GNNs, a common approach is to remove redundant or less important weights, achieving model compression with minimal accuracy loss [4]. This can be accomplished through *train-and-prune* or *sparse training*. In the train-and-prune approach, the model is first trained in the standard (dense) manner, and after training, weights are removed or set to zero based on specific criteria, as exemplified by methods like ADMM-based pruning [14]. In *sparse training*, training starts with a sparsified weight matrix, and sparse weight locations are updated during specific iterations, as demonstrated by methods like SET [9]. However, both approaches introduce irregular sparsification patterns in GNN workloads, which can hinder efficient hardware deployment.

2.3 Exploiting Sparsity in Weights and Activations

One research trend focuses on leveraging sparsity in weights and activations to achieve faster inference and improved performance. Sparse CNN [10] introduces a CNN inference architecture that exploits both weight and activation sparsity to enhance the performance and power efficiency of DNNs by retrieving only non-zero data values from DRAM and on-chip buffers. However, orchestrating a dataflow that efficiently delivers these sparse datasets to an array of multipliers, while maximizing data reuse and multiplier utilization, remains a non-trivial challenge. FATReLU [7] aims to increase activation map sparsity in convolutional neural networks for faster inference but still faces challenges with irregular activation patterns. MaxK-GNN [11] attempts to regularize activation sparsity by applying a top-k non-linearity function to the embedding feature matrix. Experiments have shown that this approach effectively speeds up training without significant accuracy loss. However, their implementation is limited to inputs in the range of 0 to 256 (8-bit integers). In this study, we apply the concept of max-pooling [6] to identify k relatively large feature values for each node, which has the same effect of regularizing activation sparsity, similar to the MaxK-GNN system.

3 Proposed Idea

Our general training framework incorporates non-linear feature approximator kernels between the layers of forward propagation kernels. Besides, the intermediate results (i.e., the extracted approximated features) are stored and afterward used during the backward propagation process. This modification enables us to optimize the two most time-intensive kernels in the GNN training pipeline.

3.1 Max Pooling Approximator

Although ReLU performs well in terms of both efficiency and accuracy in practical machine learning tasks, it has a notable drawback in GNN training: it introduces irregular sparsity in the feature matrix after each hidden layer. This sparsity requires frequent adjustments to the CSR format used for the sparse feature matrix and increases the need for conditional branching in the following CUDA kernel design, both of which can lead to significant computational overhead.

Max-pooling kernel is our proposed approximator, designed to extract f_k output features from f_n input features, based on the weighted features computed during the previous forward propagation step. By carefully configuring the pooling window size w and stride length s, the task size can be reduced to a linear function of f_n, enabling efficient computation.

$$h_{f_k}(X) = maxPooling_{j\in[1,f_n]}(XW + b)_j \quad (4)$$

where $h(X)$ represents the extracted sparse feature matrix, X is the input sparse matrix, and W and b represent the trainable weight and bias parameters, respectively, in the GNN learning framework.

As shown in the equation, our max-pooling kernel follows a node-wise (per vertex in the graph) parallel design. Further, there are two key variations of pooling windows: (i) the dimensionality of the windows, which can be 1D, 2D, or even higher; our focus will be on designing and comparing 1D and 2D implementations, and (ii) whether the windows are overlapping or using channels.

Max-pooling 1D kernel serves as our baseline design, treating the input features as a 1D array and applying a 1D pooling window at different locations, with intervals of the same length.

Algorithm 1 Max Pooling Kernel

```
Require: Input features $D$, indices $I$
Require: Out features $V$, indices $P$
 1: for all blocks $X_{\text{blocks}}$ in $kernel$ do
 2:     Initialize $cache$ in shared memory
 3:     Form 32 threads within all warps
 4:     $cache \leftarrow D_{blocks}$
 5:     Synchronize warp
 6: end for
 7: for all row $D_{\text{row}}$ in $D$ do
 8:     for all $thread_t$ in $warp_w$ do
 9:         for all $window_j$ in $thread_t$ do
10:             Find maximum $max$ in the window
11:             Record position $pos$ for $max$
12:             $V[w * f_k + t + j] \leftarrow max$
13:             $P[w * f_k + t + j] \leftarrow pos$
14:         end for
15:     end for
16: end for
17: Synchronize device
```

Max-pooling 2D kernel is an extension of the 1D design, treating input features as 2D matrices and applying 2D windows, similar to their common use in CNNs. Although GNN features may not necessarily exhibit a continuous relationship with their neighboring features (like surrounding pixels in CNN), using 2D windows offers several advantages. First, it increases the flexibility in designing the window structure. Second, it mitigates potential drawbacks when consecutive features have significantly different value ranges. For instance, if the first few features of a graph capture deterministic characteristics of a node, a 1D window might drop more of these important features compared to a 2D design, which could preserve them more effectively.

Channeled kernel with depth d is a variation of the max-pooling kernel design that replaces overlapping window designs by independently processing each channel, similar to the extraction of R, G, and B channels in a typical CNN framework. The primary goal of this variation is to increase flexibility in handling a broader range of input and output dimensions. Each channel depth d independently identifies the local maximum within its window. For instance, to extract 128 features from 256, a 2-channeled window design extracting 64 features from each channel provides an effective solution. Using shared memory minimizes the overhead of repeated operations, ensuring efficient execution.

Figure 1 illustrates how threads in a warp handle their assigned tasks (pooling windows) under a non-overlapping kernel, in the case where $f_k = 64$ and $f_n = 256$. The underlying 16×16 matrix represents an abstraction of the original n features of a single node, with one warp within a block assigned to process it. Once the local maximum within a given window is identified, the result, along with its location (i.e., the i-th feature), is written to the kernel output based on the corresponding thread index. This warp-level partitioning provides two key benefits: (i) synchronization is required only at the warp level (using `__syncwarp`) between loading data into shared memory and starting computation, significantly reducing the overhead compared to synchronizing all threads in the block, and (ii) GPU resource usage is optimized, as assigning one node per thread would consume excessive shared memory, limiting the number of threads per block.

We benchmark our kernel against the **Top-k approximator kernel**, which also extracts f_k features on a node-wise basis from the previous feature matrix but specifically targets the top f_k maximum values from the f_n features. This kernel is implemented using a warp-level parallel Odd-Even Transposition Sort, storing the top f_k feature values and their sparse locations in intermediate result arrays. Theoretically, the computational complexity of this kernel is quadratic in the number of features, making it slower than our proposed linear max-pooling kernels. The Top-k approximator kernel functions similarly to the Max-pooling approximator kernel.

$$h_{f_k}(X) = topK_{j \in [1, f_n]}(XW + b)_j \qquad (5)$$

where $h(X)$ represents the extracted sparse feature matrix, X is the input sparse matrix, and W and b represent the trainable weight and bias parameters, respectively, in the GNN learning framework.

We use odd-even transposition sort because it works with all data types, and its correctness and performance are predictable. While an optimized approach for the Top-k approximator kernel has been proposed, leveraging binary search to speed up the process, it is limited to 8-bit integer data, i.e., values in the range [0, 255]. This constraint makes it unsuitable for many GNN training targets. Moreover, the same approach cannot be directly applied to general floating-point data, as the threshold range becomes unpredictable, and the number of binary search iterations cannot be consistently determined.

As shown in Algorithm 2, this approach has three notable drawbacks. First, additional shared memory is required solely to store the sparse indices of the feature matrix. This significantly limits the number of blocks dispatched to a streaming multiprocessor (SM) on the GPU, potentially leaving more computational units idle. Second, to eliminate the overhead from the loop's conditional branching, we use an unrolled loop, and it requires iterating over the worst-case scenario—moving the first element to the last—resulting in $\frac{f_n}{2}$ iterations. Third, if we attempt to apply an early break instead of iterating through the worst case, the process requires accessing and modifying a global flag, which involves atomic operations across all threads in the warp. The overhead from these atomic operations could outweigh the speedup from fewer iterations.

3.2 Forward Propagation Kernel

Using our proposed approximator kernels, we can design a kernel for sparse-to-dense matrix multiplication, which generates the result from multiplying

Algorithm 2 Top-K Kernel

```
Require: Input features $D$, indices $I$
Require: Out features $V$, indices $P$
 1: for all blocks $X_{\text{blocks}}$ in $kernel$ do
 2:     Initialize $Vs$ in shared memory
 3:     Initialize $Ls$ in shared memory
 4:     Form 32 threads within all warps
 5:     $Vs \leftarrow D_{blocks}$
 6:     $Ls \leftarrow I_{blocks}$
 7:     Synchronize warp
 8: end for
 9: for all row $D_{\text{row}}$ in $D$ do
10:     for all $thread_t$ in $warp_w$ do
11:         for $i \in \frac{f_n}{2}$ do
12:             for odd $pair_o(a, b)$ in $thread_t$ do
13:                 if $V_s[a] > V_s[b]$ then
14:                     $swap(V_s[a], V_s[b])$
15:                     $swap(L_s[a], L_s[b])$
16:                 end if
17:             end for
18:             Synchronize warp
19:             for even $pair_e(a, b)$ in $thread_t$ do
20:                 if $V_s[a] > V_s[b]$ then
21:                     $swap(V_s[a], V_s[b])$
22:                     $swap(L_s[a], L_s[b])$
23:                 end if
24:             end for
25:             Synchronize warp
26:         end for
27:     end for
28:     $V_{blocks} \leftarrow Vs$
29:     $P_{blocks} \leftarrow Ls$
30: end for
31: Synchronize device
```

a sparse matrix (graph adjacency matrix) with a dense, fixed-size matrix (feature matrix) while optimizing performance. The computation process is as follows:

$$X_{next} = A \cdot h(X_{curr}) \tag{6}$$

where $h(X)$ represents the post-approximated feature matrix, and A is the graph adjacency matrix.

The sparse matrix A is stored in Compressed Sparse Row (CSR) format, which includes both a data array and an index segment array in global memory. The kernel then executes a parallel row-wise product operation to efficiently compute and accumulate the new feature values.

$$X_{next}[i,:] = \sum_{k=1}^{K} A[i,k] \cdot h(X_{curr})[k,:] \tag{7}$$

While our approximator kernels restrict the input feature dimension, the graph's inherent sparsity—specifically, the number of connected nodes (edges)—remains a significant factor. To address this, we partition the connected nodes of a target node into fixed-size groups and assign these groups to different warps. For instance, if each warp is configured to handle up to 64 outgoing connections (i.e., 64 sets of sparse matrix multiplication and accumulation tasks), the tasks for a node with 200 outgoing connections would be divided among four warps, handling 64, 64, 64, and 8 connections, respectively. This approach ensures a balanced workload distribution, optimizing performance for parallel computation.

Algorithm 3 Forward Kernel

```
Require: Adjacency $A$, input features $X_{curr}$
Require: Output features $X_{next}$
 1: for all rows $A_i$ in $A$ do
 2:     for assigned warps $w$ in $A_i$ do
 3:         Initialize $S_w$ in shared memory
 4:         Form 32 threads within all warps
 5:         for all connected $e_{i,j}$ in $w$ do
 6:             for all $thread_t$ in $w$ do
 7:                 $S_w[A[j,k]] \leftarrow e_{i,j} X_{curr}[j,k]$
 8:             end for
 9:         end for
10:         $X_{next}[i] \leftarrow_{atomic} S_w$
11:     end for
12: end for
13: Synchronize device
```

As shown in Algorithm 3, we leverage shared memory to compute local aggregation without encountering race conditions. In addition to workload balancing, warp-level parallelism provides another advantage here. First, it helps control the shared memory size within each block, as assigning each thread its own segment of shared memory would consume too much memory, limiting the number of threads per block. Second, when using atomic addition to accumulate the local aggregation back to the target output, potential race conditions occur only at the warp level, reducing the overhead compared to a block-level parallel design.

Graph	# of Nodes	# of Edges
ddi	4267	2135822
Reddit	232965	114615891
Yelp	716847	13954819
ogbn-products	2449029	123718280

Table 1: Experimental graphs and properties.

4 Experimental Setup

Environment Setup Our experiments were conducted on NYU's GPU Computing CUDA2 cluster, running Red Hat Enterprise Linux 9. The system is equipped with two Intel Xeon E5-2660 CPUs (40 cores total), 256 GB of RAM, and two GeForce RTX 2080 Ti GPU cards (each with 11 GB of VRAM, CUDA version 12.4). We wrote our kernels in C++ and CUDA C++ and compiled our source code using g++ 11.5 and nvcc 12.4, targeting compute architecture 7.5.

Two main factors determine the performance of our nonlinear kernels: (1) the size of the vertex, denoted as V, and (2) the input and output dimensions, denoted as f_n and f_k. We will experiment with V based on the graphs shown in Table1 1 and f_n, f_k in the range [32, 256].

To efficiently manage memory for large graphs, we utilized unified memory through `cudaMallocManaged`. This approach was chosen due to the substantial memory requirements of graph data, where traditional memory allocation methods — such as separate host memory allocation (`malloc`) and device memory allocation (`cudaMalloc`)—can result in excessive memory fragmentation or CUDA out-of-memory errors. In addition, we performed a warm-up phase by running the kernel for 10 iterations prior to measurement. This warm-up phase ensures that the GPU is fully initialized, caches are populated, and any performance variability due to initial kernel launches or memory allocations is mitigated.

For the performance measurement of the Max-pooling & Top-k approximator kernels, we conducted a rigorous analysis by calculating the average latency across 100 runs with a fixed random seed $= 123$. This ensures that the observed performance metrics are consistent and representative.

5 Experiments & Analysis

5.1 Max-pooling Kernel Dimensions

Max-pooling kernels of different dimensions can address varying input and output feature requirements. For instance, a 3-channeled 2D kernel can extract 192 features from 256 features, which is not achievable with a 1D kernel. Conversely, a 1D kernel can easily extract 64 features from 192 features, as 192 cannot be reshaped into a square for a 2D kernel. By combining these two types of max-pooling kernels, we can handle a wide range of scenarios. Our experiments begin by comparing their performance across different input and output feature dimensions.

As shown in Table 2, several important characteristics can be observed: (i) 1D and 2D kernels generally perform similarly for the same task, allowing flexibility to switch between them based on the properties or dimensions of the input features without incurring significant overhead; (ii) the input dimension primarily determines execution time, as it directly influences the length and number of pooling windows; and (iii) applying 2D channeled windows to extract more output dimensions increases execution time as the channel depth grows. This is intuitive, as finding the maximum value within the window multiple times requires each previously selected local maximum to be overwritten with a negative value to prevent it from being chosen again, which causes a linear growth requirement of memory accessing.

Input	Output	Dimensions	exec. time
256	192	2D	1.93ms
256	128	1D	0.90ms
		2D	1.00ms
256	64	1D	0.64ms
		2D	0.65ms
128	64	1D	0.44ms
64	32	1D	0.22ms
		2D	0.22ms

Table 2: Max-pooling kernels' performance in different input & output dimensions.

5.2 The Scale of Parallelism

The architecture and resource constraints of NVIDIA GPUs make the assignment of parallel tasks to computational units a critical factor in performance. We experiment with two designs for processing all features of a node: warp-level parallelism and thread-level parallelism. In the former, all required max-pooling windows are executed by a warp (comprising 32 threads), while in the latter, they will be handled by an individual thread.

In Table 3, Our experiments consistently show that warp-level parallelism outperforms thread-level parallelism across all tested graph cases. Although thread-level parallelism simplifies task assignment by eliminating the need to map each thread in a warp to its designated task and avoids requiring warp-level synchronization when loading data from global to shared memory, it has a significant drawback. Thread-level parallelism consumes a large amount of shared memory, limiting the number of threads that can be assigned to a block. Consequently, the total number of threads within a streaming multiprocessor is reduced, making it difficult to hide memory latency effectively through zero-cost context switching.

Graphs	# of nodes	Parallelism	exec. time
ddi	4267	warp	0.02ms
		thread	0.06ms
Reddit	232965	warp	0.65ms
		thread	2.42ms
Yelp	716847	warp	2.07ms
		thread	7.22ms
ogbn-product	2449029	warp	6.87ms
		thread	24.94ms

Table 3: Comparison between warp and thread parallelism.

5.3 Shared Memory

Shared memory is a crucial factor in performance, as it helps reduce the memory footprint on the GPU. In our basic max-pooling design with non-overlapping windows, each data point is accessed only once. However, additional benefits arise when threads within a warp load data from global memory into shared memory using a coherent access pattern. This approach is particularly advantageous for managing the partial random access patterns observed in the 2D kernel design. To assess the impact of shared memory usage, we conducted experiments comparing performance with and without shared memory.

Table 4 presents several key observations: (i) the use of shared memory does not significantly impact the performance of our basic non-overlapping, non-channeled window kernels. This is because the speedup from improved random access using shared memory offsets the overhead of transferring data from global memory, and (ii) shared memory provides a notable performance advantage in channeled window kernels. In this case, shared memory significantly reduces memory footprint by converting multiple global memory accesses into faster SM-level (streaming multiprocessor) hardware communication.

Input	Output	Shared Mem.	exec. time
256	64	w/	0.64ms
		w/o	0.64ms
256	128	w/	1.00ms
		w/o	1.20ms
256	192	w/	1.93ms
		w/o	2.58ms

Table 4: Comparison between using shared memory or not.

An additional advantage of using shared memory in the channeled window scenario comes from the need to overwrite selected data with negative values. Shared memory ensures that the raw input data remains unmodified while enabling faster access compared to global memory.

5.4 Compare with Top-K approximator

Max-pooling approximator kernels are designed to be faster than Top-K kernels due to the difference in workload complexity: linear in feature size for max-pooling and quadratic in feature size for Top-K in our implementations. While there are sorting optimizations for Top-K kernels, such as odd-even mergesort or bitonic mergesort, none achieve a linear time complexity.

As shown in Table 5, the execution time of the Top-K kernel is primarily determined by the input dimension f_n. This is because the most computationally intensive task is sorting all the features, which overshadows the time required to write the sorted data from shared memory back to the output arrays.

Compared to Table 2, there is a notable performance improvement in max-pooling kernels. Figure 3 provides several important insights. First, the speedup increases as the size of input features grows. This is because the disparity between the linear task complexity of max-pooling kernels and the quadratic task complexity of Top-K kernels becomes more pronounced with larger input sizes. Second, deeper channeled window designs,

Input	Output	exec. time
256	192	18.89ms
256	128	18.85ms
256	64	18.81ms
128	64	4.21ms
64	32	1.37ms

Table 5: Top-K kernels' performance in different input & output dimensions.

such as the configurations represented by the first and fourth bars (2-channeled) and the fifth bar (3-channeled), consistently demonstrate speedup despite requiring multiple repeated checks within the same window. This efficiency is maintained because the overall workload remains linear. Finally, the performance difference between max-pooling and Top-K kernels diminishes as the input size decreases, indicating comparable performance for smaller inputs.

These observations lead to the conclusion that max-pooling kernels are most effective in the early layers of the training framework, where the objective is to extract a broad range of potential features while minimizing computational costs. Conversely, in the later layers, where the goal shifts to refining and selecting the most dominant features, Top-K kernels become a more suitable choice. Although slightly slower than max-pooling kernels, Top-K kernels excel in feature extraction, making them optimal for tasks requiring high precision in the final stages of training.

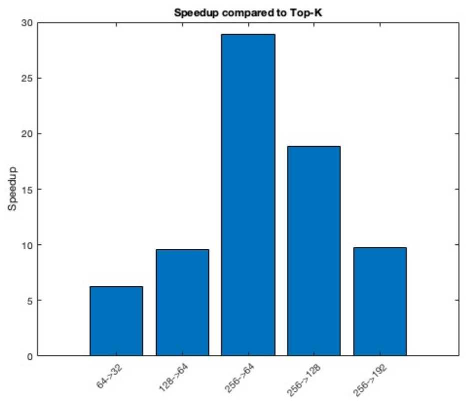

Figure 3: Speedup from Top-K to Max-pooling

5.5 Fixed Input Feature Forward Kernel

We benchmark our forward kernels, which utilize fixed-size feature embeddings, against cuSparse kernels designed for variable-size sparse feature embeddings. This comparison aims to evaluate whether preprocessing feature dimensions significantly impacts GNN training performance.

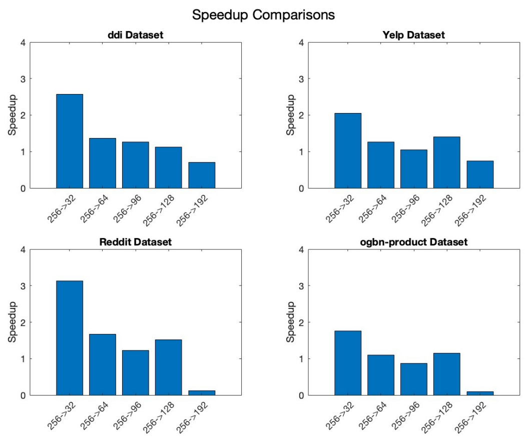

Figure 4: Forward kernel speed under different graphs

As shown in Figure 4, there is a clear trend: the greater the degree of downscaling applied, the higher the speedup achieved. Specifically, using approximators to extract 64 or fewer features consistently results in a positive speedup. However, when the extracted dimension remains close to the original, no significant speedup can be guaranteed. This outcome is likely due to the inherent quality and performance optimization of our implementations compared to the cuSparse library.

To further evaluate the refined training framework against the traditional one, Table 6 provides detailed execution times comparing our forward kernels with the cuSparse implementation. The Reddit graph dataset, which was also used in the previous experiments on approximator kernels, serves as the benchmark for this analysis. It is evident that the forward kernels are considerably more time-consuming than the approximator kernels compared to Table 2. As a result, integrating both layers (approximator and forward kernels) into the GNN training framework demonstrates a substantial overall performance improvement.

6 Conclusions

In this paper, we demonstrated that the max-pooling approximator outperforms the top-k approximator in terms of speed. By utilizing varying dimensions and channeled windows, we effectively extract most of the commonly used feature dimensions in GNN training frameworks. The speedup is particularly pronounced when downscaling features by 50% or more. This leads us to conclude the

Input	Output	Kernels	exec. time
256	32	cuSparse	180.83ms
		forward	57.75ms
256	64	cuSparse	180.83ms
		forward	108.53ms
256	96	cuSparse	180.83ms
		forward	147.42ms
256	128	cuSparse	180.83ms
		forward	119.17ms
256	192	cuSparse	180.83ms
		forward	215.82ms

Table 6: Comparison between our fixed feature forward kernels and cuSparese kernels

optimal use cases for these kernels: max-pooling approximator kernels are most effective in the early layers, while top-k approximator kernels are better suited for the later layers.

We also showed that combining our approximator kernels with a modified forward kernel optimized for sparse-to-dense matrix multiplication achieves significant speedups compared to the traditional combination of ReLU activation and cuSparse kernels. The computational overhead introduced by additional approximator kernels is negligible compared to the substantial performance gains achieved by our forward kernels. As a result, the proposed modifications provide notable benefits to the general GNN training framework.

Future research is expected to focus on improving training performance in terms of both accuracy and convergence. While related studies suggest that eliminating less significant features does not compromise model accuracy, the impact of substituting top-k approximators with max-pooling approximators remains an open question. Additionally, optimizing the configuration and combination of max-pooling windows presents an intriguing challenge. The inherent noise introduced by max-pooling approximators—where a fixed number of prominent features are not always guaranteed to pass to the next layer—could lead to novel and unexpected improvements in training dynamics.

References

[1] Abien Fred Agarap. 2018. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375.

[2] Pierre Baldi and Peter Sadowski. 2013. Understanding dropout. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 2814–2822, Red Hook, NY, USA. Curran Associates Inc.

[3] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216.

[4] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626.

[5] Thomas N. Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25.

[7] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Nir Shavit, and Dan Alistarh. 2020. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5533–5543. PMLR.

[8] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278 – 2324.

[9] Decebal Mocanu, Elena Mocanu, Peter Stone, Phuong Nguyen, Madeleine Gibescu, and Antonio Liotta. 2018. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9.

[10] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. Scnn: An accelerator for compressed-sparse convolutional neural networks. *SIGARCH Comput. Archit. News*, 45(2):27–40.

[11] Hongwu Peng, Xi Xie, Kaustubh Shivdikar, Md Amit Hasan, Jiahui Zhao, Shaoyi Huang, Omer Khan, David Kaeli, and Caiwen Ding. 2024. Maxk-gnn: Extremely fast gpu kernel design for accelerating graph neural networks training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 683–698, New York, NY, USA. Association for Computing Machinery.

[12] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph neural networks in recommender systems: A survey. *ACM Comput. Surv.*, 55(5).

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

[14] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. 2018. A systematic DNN weight pruning framework using alternating direction method of multipliers. *CoRR*, abs/1804.03294.