RTop-K: Ultra-Fast Row-Wise Top-K Algorithm and GPU Implementation for Neural Networks

Xi Xie[1]*, **Yuebo Luo**[2]*, **Hongwu Peng**[1], **Caiwen Ding**[2]

[1]University of Connecticut, USA
[2]University of Minnesota Twin Cities, USA
{xi.xie, hongwu.peng}@uconn.edu, {luo00466, dingc}@umn.edu

Abstract

Top-k algorithms are essential in various applications, from high-performance computing and information retrieval to big data and neural network model training. This paper introduces RTop-K, a highly efficient parallel row-wise top-k selection algorithm designed for GPUs. RTop-K employs a Binary Search-based approach to optimize resource allocation and provides a scalable solution that significantly accelerates top-k operations. We perform a theoretical analysis of the effects of early stopping in our algorithm, demonstrating that it maintains the accuracy of neural network models while enhancing performance. Comprehensive tests show that our GPU implementation of RTop-K outperforms other row-wise top-k GPU implementations, with minimal impact on testing accuracy when early stopping is applied. Notably, RTop-K achieves speed increases ranging from 4.245× to 9.506× with early stopping, and 3.936× without early stopping, compared to state-of-the-art implementations. The proposed methods offer significant improvements in the training and inference of Graph Neural Networks (GNNs), addressing critical challenges in latency and throughput on GPU platforms.

The implementation can be found on Github[1].

Introduction

Top-k selection is a classic algorithmic challenge that involves identifying the k largest or smallest elements from n input elements based on some predefined ranking criteria. The top-k selection algorithm has been widely applied in many traditional scenarios, such as high-performance computing (HPC) [17], information retrieval (IR) [6], big data [7], and data mining [15]. Today, the top-k algorithm is increasingly applied in the training and inference of neural network models. For example, the Avg-TopK [27] pooling method has achieved more successful results in image classification accuracy and transfer learning models compared to traditional methods. TopK-SGD [22] applied to gradient sparsification techniques significantly reduces the communication traffic without obvious impact on the model accuracy. Combining top-k with sparse training [10] can maintain constant sparsity and perform well while reducing resource requirements. In a study [4], a top-k attention loss function was introduced to address the top-k ranking prediction problem.

Graph Neural Networks (GNNs) have drawn tremendous attention in the past years due to their unique ability to extract latent information from graph data [9]. Practical applications of GNNs include the prediction of cascading power-grid failures [14], traffic forecasting [11], recommendation systems [24], and drug discovery [2]. In the design and acceleration of GNN training and inference, GPU platforms have become the prevalent choice due to their multiple advantages. Firstly, compared to other processing hardware, GPUs provide superior processing power and memory throughput [12]. For example, the NVIDIA A6000 GPU boasts an impressive computation capability of 38.7 Tera FLOPS and a memory throughput 768 GB/s. Secondly, many leading supercomputers (such as Aurora [23] and El Capitan [16]) use GPUs as their primary computing resource. Thirdly, many applications and services related to deep learning and neural networks are developed and deployed on GPU platforms. However, GNN training and inference still typically pose strict challenges on latency and throughput [25].

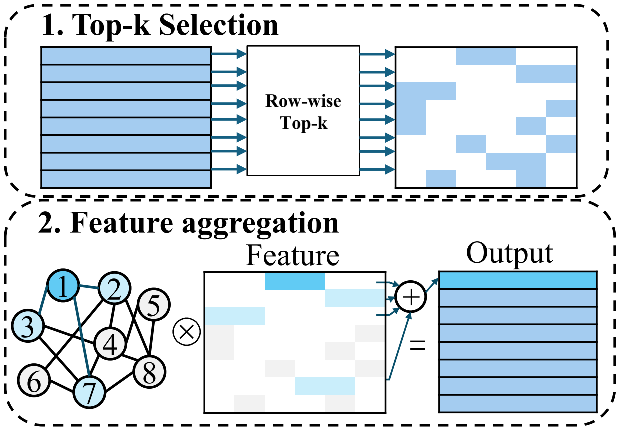

Figure 1: The core operation of MaxK-GNN, which introduces top-k selection into the GNN workflow to provide non-linearity and acceleration.

Recently, MaxK-GNN [19] has achieved great success in

[*] These authors contributed equally.

[1] https://anonymous.4open.science/r/RTopK-4DE7

the acceleration and optimization of GNN training and inference on GPUs. As shown in Figure 1, this work introduces top-k selection before the feature aggregation step in GNNs, which not only provides nonlinearity in GNNs to optimize the model's expressive ability but also demonstrates that performing SPMM operations in GNNs with the top-k-processed right multiplication matrix can achieve several times speedup over traditional workflows while maintaining good model accuracy. The top-k selection operation in MaxGNN necessitates performing a large-scale row-wise top-k computation, i.e., executing top-k operations simultaneously across a batch of vectors on GPUs.

Traditional top-k algorithms and their GPU implementations [8, 26, 13] are typically optimized for single queries or limited batched queries, that is, for a large vector or a small batch of large vectors (typically with a batch size not exceeding 100). However, the optimization focus for traditional scenarios differs from the row-wise top-k algorithm required for GNN training and inference. Implementing and optimizing row-wise top-k algorithms on GPUs pose challenges in terms of dispersion, parallelism, and efficiency. Since row-wise top-k involves performing top-k operations on a large batch of vectors simultaneously, and each vector's length corresponds to the hidden dimension of the neural network layer (which usually does not exceed 1024), it is crucial to allocate only a small and appropriate amount of GPU resources for each vector. Under these limited resource constraints, the various optimization methods proposed for large vectors in traditional top-k implementations may be overly complex and inefficient. We should seek simple and efficient algorithms tailored to this scenario.

Additionally, we must consider the requirements and characteristics of applying row-wise top-k in neural networks. We only need to select the values of the top-k elements in each row and their indices in the vector. We do not need to perform sorting at all; neither the k selected elements in each row nor the remaining elements require sorting. Furthermore, given the neural network's tolerant and robust nature, we can explore the feasibility of approximate top-k to further accelerate the overall algorithm.

To efficiently implement row-wise top-k on GPUs for neural network applications, we introduce RTop-K, a highly efficient parallel top-k selection algorithm designed for a large batch of limited-size vectors, with the capability of approximation to further enhance the speed of the row-wise top-k algorithm without compromising the accuracy of the neural network model.

We summarize our contributions as follows:

- We propose a Binary Search-based Top-k Algorithm and provide a theoretical analysis of the effects of early stopping.
- We implement the Binary Search-based Top-k Algorithm on GPU and conduct comprehensive tests, demonstrating that it outperforms state-of-the-art row-wise top-k GPU implementations, with early stopping having minimal impact on the model's testing accuracy.

Preliminary and Related Works

Top-k Algorithms

The heap-based top-k algorithm [3] uses a min-heap to maintain the top-k elements. By iterating through the data, each element is compared with the smallest element in the heap (i.e., the root of the heap). If the current element is larger, it replaces the root, and the heap is restructured to maintain the min-heap property. On the other hand, if the k smallest elements are required, a max-heap is used instead. QuickSelect [5] is a top-k algorithm inspired by the quicksort algorithm, designed to find the k-th largest element in an unordered list. It works by partitioning the data around a pivot element and recursively processing the part that contains the k-th element. The bucket-based top-k algorithm [1] divides the data into several buckets based on the range of values. The top-k elements are then found by sorting the buckets or partially sorting only the relevant buckets. This method is particularly useful for uniformly distributed data. RadixSelect [1] is a variant of radix sort used for selecting the top-k elements. It processes the digits of the numbers starting from the least significant digit to the most significant digit. This method is efficient for fixed-length integer keys. The bitonic top-k algorithm [21] is based on bitonic sorting, a parallel sorting algorithm. It constructs a bitonic sequence (a sequence that first increases and then decreases) and then merges it to find the top-k elements.

When considering these algorithms, we must take into account their suitability for GPU implementation and the optimization requirements for specific problem scenarios. For example, the heap-based top-k algorithm is not well-suited for parallelization on GPUs because it relies on complex tree structure operations and element-wise comparisons and swaps. Although QuickSelect, RadixSelect, and the bitonic top-k algorithm can be successfully implemented on GPUs, they still require considerable data access and resource demands when operating on a vector. This makes it difficult to optimize for row-wise top-k scenarios, where a large batch of limited size vectors requires top-k selection simultaneously, necessitating simplified operations and limited resource usage per vector. The bucket-based top-k algorithm is more friendly to row-wise top-k scenarios but still requires further simplification to enhance performance.

GPU Architecture

The architecture of NVIDIA GPUs consists of an array of multithreaded Streaming Multiprocessors (SMs) designed to execute thousands of threads concurrently. A function that runs on a GPU is called a kernel.

Thread and Memory Hierarchy. NVIDIA GPUs organize threads into warps, with each warp containing 32 threads that execute the same instruction simultaneously. Warps are grouped into blocks, which reside on the same Streaming Multiprocessor (SM) and can communicate via shared memory, a fast on-chip memory space. Blocks are further grouped into grids for specific kernel launches. Threads access data from multiple memory spaces: device memory (large but slower, accessible by all threads), shared memory (low-latency, for communication within a block),

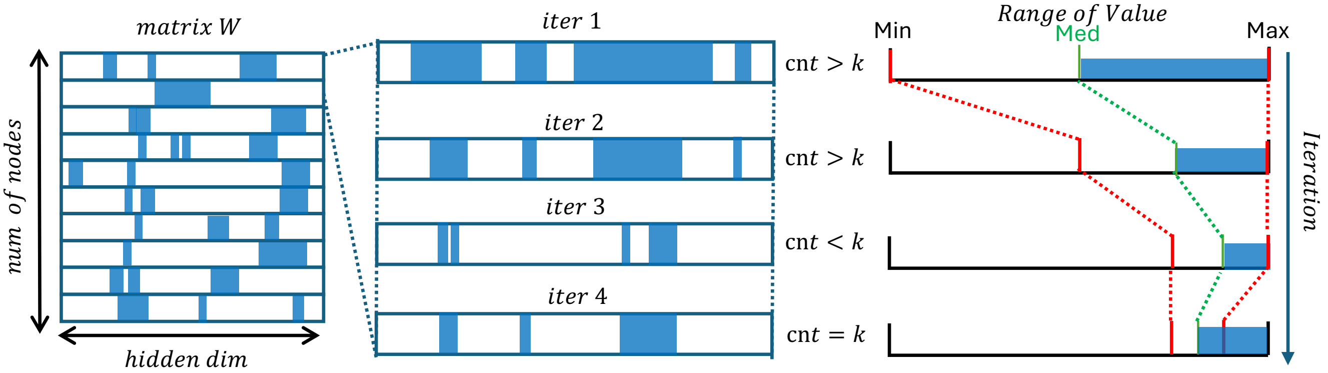

Figure 2: Illustration of the Binary Search-based Top-k Algorithm

and registers (fastest, partitioned among threads on an SM). The usage of registers can affect the number of blocks that can be active on an SM.

Warp-Level Primitives. Warp-level primitives are a set of operations that allow threads within a warp to cooperate and communicate efficiently. These include:

- **Synchronization primitive**: Ensures that all threads reach the same point in execution before proceeding.
- **Shuffle primitive**: Allows threads to exchange values within a warp.
- **Ballot primitive**: Enables threads to collectively determine which threads meet a specified condition by generating a mask representing the threads that satisfy the condition.
- **Counting primitive**: Counts the number of set bits in a given mask, often used in conjunction with the ballot primitive.

The flexible use of warp-level primitives is crucial for designing high-performance kernels, as the efficiency of information sharing through these primitives can even surpass that of using shared memory.

GPU Top-k Implementations

Dr. Top-k [8] is a delegate-centric system that helps reduce the workload of GPU top-k computations, including Radix Select, Bucket Select, and Bitonic Select. It achieves this by dividing the input into sub-ranges and selecting delegates from them, along with performing multi-GPU optimizations. A work [26] proposed two optimization methods, AIR Top-k and GridSelect. AIR Top-k employs an iteration-fused design and adaptive strategy to minimize CPU-GPU communication and memory traffic, while GridSelect uses a shared queue and parallel two-step insertion to decrease costly operations, enhancing parallel top-k efficiency on GPUs. A recent RadixSelect implementation RadiK [13] utilizes an optimization framework tailored for high memory bandwidth and resource utilization, along with an adaptive scaling technique for enhanced robustness, that supports larger k values with high efficiency.

However, the above state-of-the-art GPU implementations are optimized for limited batches of large vectors. For instance, Dr. Top-k, AIR Top-k, and RadiK are designed for scenarios where the vector size is on the order of 2^{20} (about one million elements), and the batch size does not exceed 100. This is not suitable for row-wise top-k applications, where the typical vector size is less than 1024, and the batch size can reach millions.

PyTorch's top-k implementation [20] is suitable for row-wise top-k operations. However, it involves a precise sorting operation for each vector, which can lead to lower efficiency. Despite being able to handle large batch sizes, the exact sorting required for each vector results in a performance bottleneck, making it less efficient for large-scale applications where rapid top-k selection is critical.

RTop-K Framework

The row-wise top-k operation involves finding the largest (or smallest) k elements and their indices in each row of a matrix. Without loss of generality, we assume finding the largest k elements. Suppose a matrix has N rows and M columns; the problem is equivalent to performing top-k selection on N vectors of length M simultaneously. Since N can be extremely large and M is limited, we need to apply a simplified algorithm to each vector, ensuring that the algorithm can execute quickly with very limited computational resources and memory access. We adopt a binary search-based top-k algorithm, which is even more convenient to execute than the bucket top-k algorithm.

Binary Search-based Top-k Algorithm

The algorithm, as illustrated in Fig. 2, first retrieves the min and max values of the vector, and then uses several iterations of binary search to determine a threshold. The algorithm stops when the number of elements filtered by the current threshold equals k. To address the issue where the loop might struggle to exit due to multiple equal or very close elements during filtering, we introduce the condition $max - min > \epsilon$, where $\epsilon = \epsilon' \cdot max$, and ϵ' is a small value representing the precision, such as 10^{-4}. If this condition is not met, the loop will also exit, and the remaining elements will be selected sequentially among those that are equal within the specified precision. Table 1 presents the

Algorithm 1: Binary Search-based Top-k Algorithm

```
Require: Vector $\mathbf{v}$ of size $M$, integer $k$
Ensure: Top-$k$ largest elements and their indices in $\mathbf{v}$
$min \leftarrow \min(\mathbf{v})$
$max \leftarrow \max(\mathbf{v})$
$\epsilon \leftarrow \epsilon' \cdot max$
$mid, cnt \leftarrow 0$
while $max - min > \epsilon$ do
    $mid \leftarrow \frac{min+max}{2}$
    $cnt \leftarrow |\{i \mid \mathbf{v}_i \geq mid\}|$
    if $cnt < k$ then
        $max \leftarrow mid$
    else if $cnt > k$ then
        $min \leftarrow mid$
    else
        break
    end if
end while
if $cnt = k$ then
    $\mathbf{elems}, \mathbf{indices} \leftarrow \{(\mathbf{v}_i, i) \mid \mathbf{v}_i \geq mid\}$
else
    $\mathbf{elems}, \mathbf{indices} \leftarrow \{(\mathbf{v}_i, i) \mid \mathbf{v}_i > mid + \epsilon\}$
    Append the first $k - |\mathbf{elems}|$ pairs of $\{(\mathbf{v}_i, i) \mid mid - \epsilon \leq \mathbf{v}_i \leq mid + \epsilon\}$ to $\mathbf{elems}, \mathbf{indices}$
end if
return $\mathbf{elems}, \mathbf{indices}$
```

Table 1: Cumulative Percentage of Iterations Where the Loop Exits for Different k Values ($\epsilon = 10^{-4}, M = 256$). Results are based on 10^5 repeated experiments for each k.

Iter	k=16	k=32	k=64	k=96	k=128
3	4.13%	2.71%	1.96%	1.34%	1.58%
4	8.98%	5.32%	3.52%	3.00%	2.81%
5	17.90%	10.64%	6.92%	5.84%	5.59%
6	33.86%	21.40%	13.87%	11.72%	11.15%
7	54.43%	38.84%	27.12%	23.29%	22.11%
8	72.38%	59.17%	46.64%	41.35%	39.93%
9	84.53%	76.00%	66.21%	61.48%	60.35%
10	91.88%	86.81%	80.68%	77.37%	76.62%
11	95.81%	93.03%	89.79%	87.64%	87.18%
12	97.89%	96.45%	94.70%	93.57%	93.31%
13	98.97%	98.21%	97.35%	96.70%	96.60%
14	99.52%	99.12%	98.67%	98.34%	98.25%
15	99.76%	99.53%	99.34%	99.20%	99.17%
16	100.00%	100.00%	100.00%	100.00%	100.00%
Avg	7.60	8.29	8.95	9.52	9.60

Algorithm 2: Binary Search-based Top-k Algorithm with Early Stopping

```
Require: Vector $\mathbf{v}$ of size $M$, integer $k$, integer $max\_iter$
Ensure: Top-$k$ largest elements and their indices in $\mathbf{v}$
$min \leftarrow \min(\mathbf{v})$
$max \leftarrow \max(\mathbf{v})$
for $iter \leftarrow 1$ to $max\_iter$ do
    $mid \leftarrow \frac{min+max}{2}$
    $cnt \leftarrow |\{i \mid \mathbf{v}_i \geq mid\}|$
    if $cnt < k$ then
        $max \leftarrow mid$
    else if $cnt > k$ then
        $min \leftarrow mid$
    else
        break
    end if
end for
$\mathbf{elems}, \mathbf{indices} \leftarrow \{(\mathbf{v}_i, i) \mid \mathbf{v}_i \geq min\}$
return first $k$ pairs of $\mathbf{elems}, \mathbf{indices}$
```

statistical results of the iteration counts at which the algorithm exits for different values of k, with the vector's size $M = 256$. For each k value, 10^5 repeated experiments were conducted, with the vector initialized with normally distributed elements. It can be observed that the average iteration at exit ranges from 7.6 to 9.6, and the probability of the iteration count being less than or equal to 13 exceeds 95%. Algorithm 1 summarizes the complete binary search-based top-k algorithm process, but it still contains a number of branching conditions. Given the inherent robustness of neural networks, we can explore the feasibility of incorporating early stopping into the algorithm. We first present the pseudocode for the early stopping algorithm and then conduct a numerical analysis.

As shown in Algorithm 2, the introduction of early stopping further simplifies the algorithm, with the main loop forcefully exiting in no more than max_iter iterations. The collection phase uses min instead of mid as the threshold, ensuring that only one-pass collection is needed, thereby eliminating the need for the two-pass collection process present in the original algorithm (in the branch where $cnt \neq k$). Table 2 summarizes the hit rate and differences between the early stopping top-k selection with different max_iter values and the optimal top-k selection. The experiments were conducted with vectors of size $M = 256$ consisting of normally distributed elements, and 10^5 repeated experiments for each condition. When $max\_iter \geq 5$ for $k \geq 32$ ($max\_iter \geq 6$ for $k = 16$), both the maximum element and the minimum element in the early stopping top-k selection have an average relative error of no more than 5%. For $k \geq 64$, only 4 iterations are needed for the hit rate between the early stopping top-k selection and the optimal top-k selection to exceed 80%. These results indicate that the early stopping top-k selection is numerically stable and controllable. We will further test the impact of early stopping top-k selection on model accuracy in the experimental section.

Table 2: Statistics of early stop top-k selection for Different k Values and Maximum Iterations (M=256). $E_1(\%)$ represents the average relative error between the maximum element in early stop top-k selection and the maximum element in the optimal top-k selection. $E_2(\%)$ represents the average relative error between the minimum element in early stop top-k selection and the minimum element in the optimal top-k selection. Hit(%) represents the overlap ratio between the early stop top-k selection and the optimal top-k selection.

	$k = 16$			$k = 32$			$k = 64$			$k = 96$			$k = 128$		
Iter	$E_1(\%)$	$E_2(\%)$	Hit(%)	$E_1(\%)$	$E_2(\%)$	Hit(%)	$E_1(\%)$	$E_2(\%)$	Hit(%)	$E_1(\%)$	$E_2(\%)$	Hit(%)	$E_1(\%)$	$E_2(\%)$	Hit(%)
2	12.6	20.17	45.85	13.46	30.68	37.81	7.12	25.03	51.78	4.42	17.80	69.59	4.6	24.73	70.93
3	8.01	13.13	54.29	6.22	13.19	60.32	4.44	12.40	69.04	3.39	12.94	74.41	2.78	13.23	79.33
4	4.93	7.64	68.35	3.47	7.05	74.46	2.47	6.55	80.51	1.99	6.82	84.33	1.6	7.24	87.34
5	3.52	5.20	77.36	2.20	4.31	83.19	1.47	3.70	87.88	1.18	3.91	90.49	0.97	4.29	92.34
6	2.90	4.33	81.57	1.62	3.17	87.62	0.99	2.39	91.83	0.77	2.57	93.77	0.62	2.90	95.03
7	2.67	4.10	83.17	1.38	2.79	89.51	0.79	1.87	93.68	0.61	2.00	95.33	0.47	2.30	96.35
8	2.61	4.06	83.68	1.31	2.69	90.19	0.71	1.72	94.35	0.55	1.82	95.94	0.41	2.11	96.86

GPU Implementation

Both Algorithm 1 and Algorithm 2 are well-suited for GPU implementation, where a single warp processes a single vector of size M. Fig. 3 illustrates the GPU implementation design for Algorithm 2, which can be divided into three stages: loading, searching, and selecting.

Loading stage: In this stage, each vector is loaded from global memory into the corresponding shared memory, maintaining efficiency through coalescing memory access. A synchronization barrier is set at the end of this stage.

Searching stage: In this stage, each vector is handled by a single warp, assuming the warp contains w threads (Fig. 3 illustrates $w = 4$, while in actual hardware environments $w = 32$). The first step is to obtain the maximum and minimum elements of the vector. The vector is first divided into $\lceil M/w \rceil$ parts, with each thread responsible for extracting the maximum and minimum elements within its assigned part. Then, a tree-reduction using the shuffle primitive is performed in five steps to obtain the maximum and minimum elements across the entire warp, and these values are broadcasted to all threads within the warp. The second step is to perform binary search according to Algorithm 2 to obtain the selection threshold. In each iteration, the count of elements above the current threshold is accumulated and broadcasted using the same approach. After a specified number of iterations, the final threshold is obtained.

Selecting stage: A single warp traverses the entire vector in one pass. The ballot primitive is used to identify the elements and their indices that meet the selection threshold, and these are dumped into the output buffer. The pop count primitive is then used to count the number of selected elements to ensure that only the top-k pairs are dumped.

This design requires no data writes outside of registers, except for loading the vector and dumping the results. During the searching and selecting stages, warp-level primitives are utilized to achieve highly optimized inter-thread collaboration. Moreover, each warp operates independently in parallel, maintaining high overall efficiency. The implementation for Algorithm 1 follows the same workflow, with adjustments made to the termination condition of the loop in the searching stage. Additionally, the selecting stage may require a potential two-pass selection, which is accomplished by repeating the selection process with a different threshold.

Experiments

Setup and Configuration

The CUDA source code of RTop-K of the Binary Search-based Top-k Algorithm with Early Stopping is compiled utilizing NVCC, version 12.2, and the execution is carried out on an NVIDIA A6000 platform equipped with Ubuntu 22.04. The tests cover various input matrix dimensions, with the number of rows N ranging from 2^{14} to 2^{20}, hidden dimensions M ranging from 256 to 768, and k values ranging from 16 to 128.

The performance of RTop-K with different early stopping settings is compared against the row-wise top-k implementation provided by PyTorch [20], which is the state-of-the-art row-wise top-k implementation that can support a large number of rows. The latency measurements are conducted using the Nsight Compute [18] tool.

Result Analytics

Fig. 5 presents a comprehensive evaluation of RTop-K compared to the PyTorch implementation. It can be observed that RTop-K demonstrates significant speed improvements over PyTorch across various early stopping max_iter settings. Even with no early stopping ($\epsilon = 10^{-4}$), RTop-K consistently outperforms PyTorch in all scenarios. Moreover, the speed of RTop-K remains nearly unaffected by different values of k. Table 3 summarizes the average speed-up of RTop-K relative to PyTorch, showing a speed increase of $4.245\times$ to $9.506\times$ with early stopping, and $3.936\times$ with no early stopping.

Table 4 summarizes the accuracy of four real-world MaxK-GNN based GraphSage models and the proportion of time spent on row-wise top-k operations during training. It is evident that row-wise top-k operations account for a substantial portion of the training time, ranging from 11.66% in Reddit to 26.86% in Flickr. This indicates that optimizing top-k operations in real-world MaxK-GNN based models is meaningful for their training. The impact of applying RTop-K with different early stopping settings in the actual training

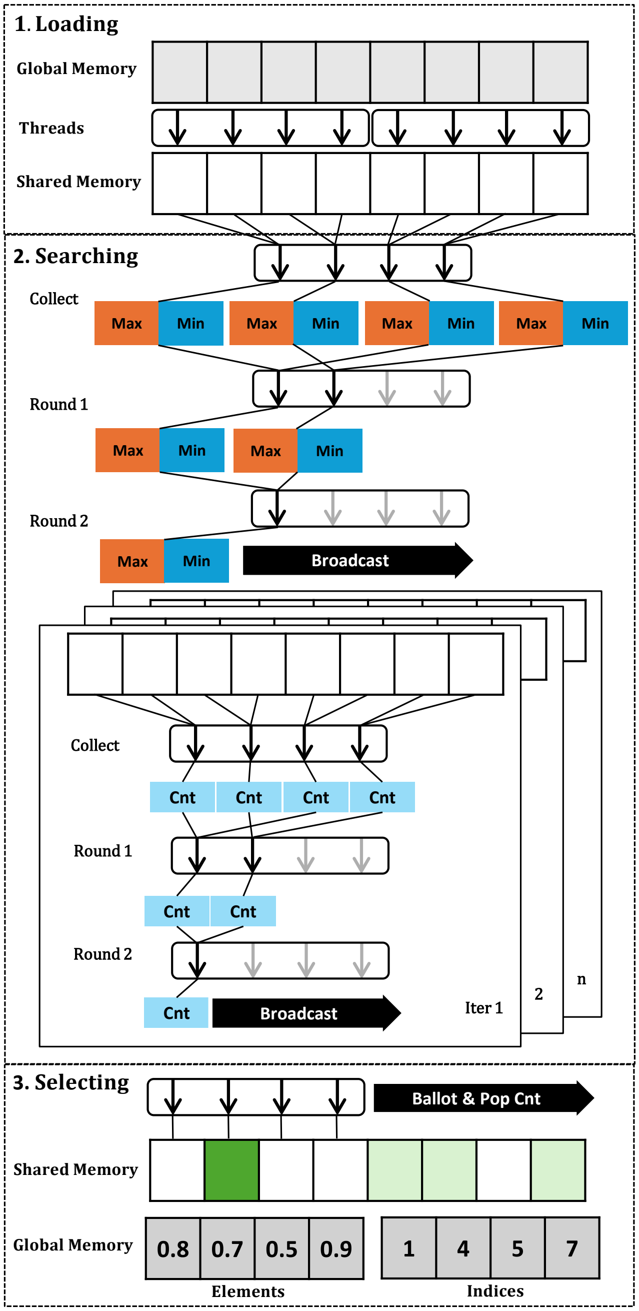

Figure 3: Example of MaxK hardware implementation

of these four models is shown in Fig. 4. It can be observed that the testing accuracy of the models remains good, and in the cases of the Reddit, Flickr, and Yelp models, the testing accuracy of RTop-K with early stopping even surpasses that of the optimal top-k. This demonstrates the successful effectiveness of RTop-K in real-world applications.

Conclusion

In this paper, we presented RTop-K, a highly efficient parallel row-wise top-k selection algorithm for GPUs. By employing a Binary Search-based approach, RTop-K significantly accelerates top-k operations while maintaining the accuracy of neural network models, as confirmed by our theoretical analysis. Comprehensive testing showed that RTop-K outperforms state-of-the-art GPU implementations, achieving speed-ups of 4.245× to 9.506× with early stopping and 3.936× without early stopping. These findings demonstrate RTop-K's potential to improve the performance of Graph Neural Networks (GNNs) by addressing challenges in latency and throughput on GPU platforms.

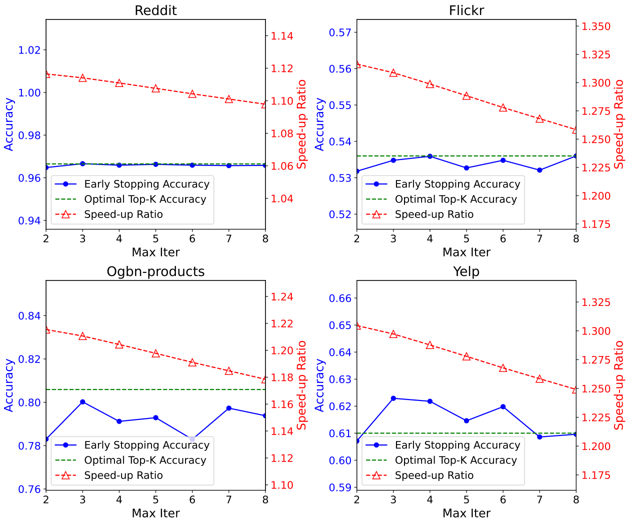

Figure 4: Testing accuracy and overall speed-up ratio of applying RTop-K to GraphSage training on different graphs.

Table 3: Average speed up of RTop-K compared to PyTorch implementation ($\epsilon = 10^{-4}$ for no early stopping).

Max Iter	Avg Speed Up	Max Iter	Avg Speed Up
2	9.506	6	5.256
3	8.216	7	4.699
4	6.965	8	4.245
5	6.002	no early stopping	3.936

Table 4: Graph data and the testing accuracy/F1 score of the MaxK-GNN based GraphSage model along with the percentage of time spent on row-wise top-k operations during training.

Graph	# Nodes	Acc/F1 score (%)	Top-k time (%)
ogbn-products	2449029	80.59	19.81
yelp	716847	61	26.09
reddit	232965	96.65	11.66
flickr	89250	53.6	26.86

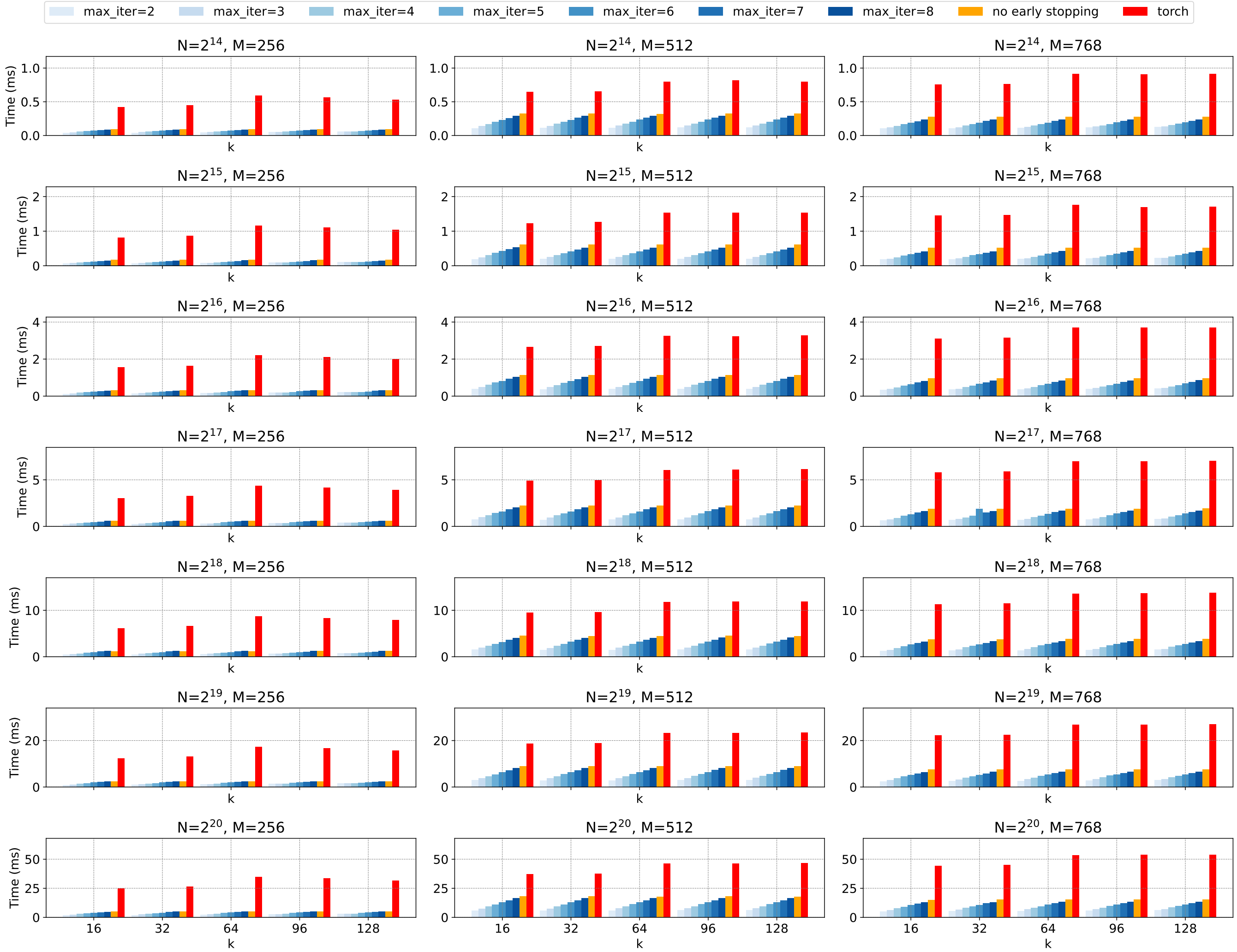

Figure 5: Comparison of execution time (ms) between RTop-K with different early stopping max_iter and without early stopping ($\epsilon = 10^{-4}$), against PyTorch for various (N, M, k) configurations.

References

[1] Alabi, T.; Blanchard, J. D.; Gordon, B.; and Steinbach, R. 2012. Fast k-Selection Algorithms for Graphics Processing Units. *Journal of Experimental Algorithmics*, 17: 4–2.

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

[3] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2009. *Introduction to Algorithms*. MIT press.

[4] Cui, C.; Zong, J.; Ma, Y.; Wang, X.; Guo, L.; Chen, M.; and Yin, Y. 2021. Tri-Branch Convolutional Neural Networks for Top-k Focused Academic Performance Prediction. arXiv:2107.10424.

[5] Dashti, A.; Komarov, I.; and D'Souza, R. M. 2013. Efficient computation of k-Nearest Neighbour Graphs for large high-dimensional data sets on GPU clusters. *PLoS One*, 8(9): e74113.

[6] Ding, S.; and Suel, T. 2011. Faster Top-k Document Retrieval Using Block-Max Indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 993–1002.

[7] Gaihre, A.; Pandey, S.; and Liu, H. 2019. Deanonymizing Cryptocurrency with Graph Learning: The Promises and Challenges. In *Conference on Communications and Network Security (CNS)*, 1–3. IEEE.

[8] Gaihre, A.; Zheng, D.; Weitze, S.; Li, L.; Song, S. L.; Ding, C.; Li, X. S.; and Liu, H. 2021. Dr. Top-k: delegate-centric Top-k on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21. New York, NY, USA: Association for Computing Machinery. ISBN 9781450384421.

[9] Hu, W.; Fey, M.; Zitnik, M.; Dong, Y.; Ren, H.; Liu, B.; Catasta, M.; and Leskovec, J. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33: 22118–22133.

[10] Jayakumar, S. M.; Pascanu, R.; Rae, J. W.; Osindero, S.; and Elsen, E. 2021. Top-KAST: Top-K Always Sparse Training. arXiv:2106.03517.

[11] Jiang, W.; and Luo, J. 2022. Graph neural network for traffic forecasting: A survey. *Expert Systems with Applications*, 117921.

[12] Li, A.; Liu, W.; Wang, L.; Barker, K.; and Song, S. L. 2018. Warp-consolidation: A Novel Execution Model for GPUs. In *International Conference on Supercomputing*.

[13] Li, Y.; Zhou, B.; Zhang, J.; Wei, X.; Li, Y.; and Chen, Y. 2024. RadiK: Scalable and Optimized GPU-Parallel Radix Top-K Selection. In *Proceedings of the 38th ACM International Conference on Supercomputing*, ICS '24, 537–548. New York, NY, USA: Association for Computing Machinery. ISBN 9798400706103.

[14] Liu, Y.; Zhang, N.; Wu, D.; Botterud, A.; Yao, R.; and Kang, C. 2020. Guiding Cascading Failure Search with Interpretable Graph Convolutional Network. *Computing Research Repository (CoRR) in arXiv*, abs/2001.11553.

[15] Malkov, Y. A.; and Yashunin, D. A. 2018. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4): 824–836.

[16] Moss, S. 2021. El Capitan Supercomputer to Feature AMD Chips, Break 2 Exaflops Barrier. Available at https://www.datacenterdynamics.com/en/news/el-capitan-supercomputer-feature-amd-chips-break-2-exaflops-barrier/. Accessed: 2021, Jan 31.

[17] Muneer. 2021. arrayfireRequest. Available at https://groups.google.com/g/arrayfire-users/c/oDtQcI7afZQ/. Accessed: 2021, Mar 17.

[18] NVIDIA. 2023. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute. Accessed: 2023-08-20.

[19] Peng, H.; Xie, X.; Shivdikar, K.; Hasan, M. A.; Zhao, J.; Huang, S.; Khan, O.; Kaeli, D.; and Ding, C. 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*, 683–698.

[20] Pytorch. 2024. torch.topk. https://pytorch.org/docs/stable/generated/torch.topk.html. Accessed: 2024-08-15.

[21] Shanbhag, A.; Pirk, H.; and Madden, S. 2018. Efficient Top-K Query Processing on Massively Parallel Hardware. In *Proceedings of the 2018 International Conference on Management of Data*, 1557–1570. ACM.

[22] Shi, S.; Chu, X.; Cheung, K. C.; and See, S. 2019. Understanding Top-k Sparsification in Distributed Deep Learning. arXiv:1911.08772.

[23] The Verge. 2021. Americas First Exascale Supercomputer to be Built by 2021. Available at https://www.theverge.com/2019/3/18/18271328/supercomputer-build-date-exascale-intel-argonne-national-laboratory-energy. Accessed: 2021, Jan 31.

[24] Wu, S.; Sun, F.; Zhang, W.; Xie, X.; and Cui, B. 2020. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)*.

[25] Xie, X.; Peng, H.; Hasan, A.; Huang, S.; Zhao, J.; Fang, H.; Zhang, W.; Geng, T.; Khan, O.; and Ding, C. 2023. Accel-GCN: High-Performance GPU Accelerator Design for Graph Convolution Networks. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 01–09.

[26] Zhang, J.; Naruse, A.; Li, X.; and Wang, Y. 2023. Parallel Top-K Algorithms on GPU: A Comprehensive Study and New Methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23. New

York, NY, USA: Association for Computing Machinery. ISBN 9798400701092.

[27] Özdemir, C. 2023. Avg-topk: A new pooling method for convolutional neural networks. *Expert Systems with Applications*, 223: 119892.