# Layer-Wise Sparse Training of Transformer via Convolutional Flood Filling

Bokyeong Yoon, Yoonsang Han, and Gordon Euhyun Moon(✉)

Sogang University, Seoul, Republic of Korea
{bkyoon,han14931,ehmoon}@sogang.ac.kr

**Abstract.** Sparsifying the Transformer has garnered considerable interest, as training the Transformer is very computationally demanding. Prior efforts to sparsify the Transformer have either used a fixed pattern or data-driven approach to reduce the number of operations involving the computation of multi-head attention, which is the main bottleneck of the Transformer. However, existing methods suffer from inevitable problems, including potential loss of essential sequence features and an increase in the model size. In this paper, we propose a novel sparsification scheme for the Transformer that integrates convolution filters and the flood filling method to efficiently capture the layer-wise sparse pattern in attention operations. Our sparsification approach significantly reduces the computational complexity and memory footprint of the Transformer during training. Efficient implementations of the layer-wise sparsified attention algorithm on GPUs are developed, demonstrating our SPION that achieves up to 2.78× speedup over existing state-of-the-art sparse Transformer models and maintain high evaluation quality.

**Keywords:** Deep Learning · Sparse Transformer · Convolutional Flood Filling

## 1 Introduction

The Transformer is a state-of-the-art deep neural network developed for addressing sequence tasks, originally proposed by Vaswani et al. [18]. One of the main advantages of the Transformer is that, given a sequence of input data points (e.g., a sentence of word tokens), it is able to compute the multi-head attention (MHA) operation in parallel, thereby quickly and accurately capturing long-term dependencies of data points. However, as the sequence length increases, the overall computational cost and memory space required for training the Transformer also increase quadratically [1]. Especially, a MHA sub-layer in the Transformer occupies a substantial portion of the total execution time and becomes the main bottleneck as the sequence length increases. The MHA operation requires a large number of dot-product operations to compute the similarity between all data points in the sequence. However, the dot-product operation inherently has limitations in memory bandwidth since it performs only two floating-point operations


D.-N. Yang et al. (Eds.): PAKDD 2024, LNAI 14646, pp. 158–170, 2024.
https://doi.org/10.1007/978-981-97-2253-2_13

for each pair of data elements read from memory. Hence, in order to mitigate computational complexity and improve data locality of the Transformer, several approaches have addressed the sparsification of computations associated with the MHA operation [2,16,19–21]. However, previous approaches suffer from two primary limitations. First, when the Transformer adopts identical fixed patterns of non-zero entries in the attention matrices across all layers during training, it becomes difficult to effectively capture key features within the sequence. Second, when the Transformer employs additional parameters to learn the sparsity pattern in the attention matrices, both the model size and the computational overhead increase. To address the limitations of previous approaches, we focus on developing a specialized sparse attention scheme that significantly reduces memory consumption and efficiently handles variations of sparse patterns across different types of Transformers and datasets.

In this paper, we present a new sparsity-aware layer-wise Transformer (called **SPION**) that dynamically captures the variations in sparse pattern within the MHA operation for each layer. SPION judiciously explores the sparsity pattern in the MHA operation based on a novel convolutional flood filling method. To precisely detect the characteristics of the sparse pattern in the attention matrices, SPION identifies the shape of sparse pattern by utilizing a convolution filter and the degree of sparsity through a flood filling-based scheme. During the generation of the sparse pattern for each layer, we construct a sparsity pattern matrix with a blocked structure to enhance data locality for the MHA operation that involves sparse matrix multiplication. Furthermore, by capturing layer-specific sparse pattern for each layer, SPION performs layer-wise MHA computations iteratively until convergence is achieved. As the sparse MHA operations contribute significantly to the overall training workload, we develop an efficient GPU implementation for sparse MHA to achieve high performance with quality of results.

We conduct an extensive comparative evaluation of SPION across various classification tasks, including image and text classification, as well as document retrieval involving lengthy sequences. Experimental results demonstrate that our SPION model achieves up to a 5.91× reduction in operations for MHA computation and up to a 2.78× training speedup compared to existing state-of-the-art sparse Transformers while maintaining a better quality of solution.

## 2 Background and Related Work

### 2.1 Transformer

The encoder-only Transformer, which is one of the variants of the Transformer model, is widely used for various classification tasks using text and image datasets [5,6,17]. In the encoder-only Transformer, each encoder layer consists of a MHA sub-layer and a feed-forward sub-layer. For each encoder layer, the query ($Q \in \mathbb{R}^{L \times D}$), key ($K \in \mathbb{R}^{L \times D}$), and value ($V \in \mathbb{R}^{L \times D}$) are obtained by performing linear transformations on the input embedding. Hereafter, we denote $L$, $D$ and $H$ as the length of input sequence, the size of the embedding

for each data point in the sequence, and the number of heads, respectively. To efficiently perform MHA computation, each $Q$, $K$, and $V$ matrix is divided into $H$ sub-matrices (multi-heads) along the $D$ dimension.

$$A = softmax\left(\frac{Q \times K^T}{\sqrt{(D/H)}}\right) \times V \tag{1}$$

The MHA computation is defined by Eq. 1. After computing the attention for each head, all matrices $A_0$ through $A_{H-1}$ are concatenated to form the final attention matrix $A$. Then, the attention matrix $A$ is passed through the feed-forward sub-layer to produce new embedding vectors, which is then fed into the next encoder layer. Therefore, in terms of computational complexity, the main bottleneck of the encoder layer is associated with processing the MHA sub-layer, which involves a large number of matrix-matrix multiplications across multiple heads. More specifically, the number of operations required for computing the attention matrix $A$ is $2L^2(2D+1) - L(D+1)$, indicating a quadratic increase in operations as the input sequence length ($L$) increases.

**Sparse MHA.** Given the long input sequences, several sparse attention techniques have been proposed to reduce the computational complexity involved in multiplying $Q$ and $K^T$ in the MHA operation. The basic intuition behind sparse attention techniques is that a subset of data points in the long sequence can effectively represent the entire input sequence. In other words, only the highly correlated necessary elements (i.e., data points) in $Q$ and $K$ can be utilized to reduce the computational workload. Therefore, when employing sparse attention, the number of operations required to compute $Q \times K^T$ is $C(2D-1)$, where $C \ll L^2$ represents the number of critical elements in the resulting matrix. In contrast, without sparsification, the original computation of $Q \times K^T$ requires $L^2(2D-1)$ operations.

### 2.2 Related Work on Sparse Attention

Many previous efforts to achieve efficient sparse Transformers have sought to sparsify the MHA operation both before and during model training/fine-tuning.

**Fixed Sparse Pattern.** One of the strategies for performing sparse MHA is to use a predetermined sparsity patterns, where only specific data points in the input sequence are selected to perform the MHA operation before training the model. Several variants of the Transformer model adapt the sliding windows approach in which the attention operations are performed using only the neighboring data points in the matrices $Q$ and $K$. The Sparse Transformer [3] originally employs the sliding windows attention to sparsify the MHA operation. The Longformer [2] is an extension to the Sparse Transformer and introduces dilated sliding windows, which extend the receptive field for computing similarity by skipping one data point at a time while performing the sliding windows

attention. Furthermore, ETC [1] and BigBird [20] incorporate global attention that performs similarity calculations between a given data point and all other data points in the input sequence.

The main advantage of utilizing the fixed sparsity pattern is the reduction in computational overhead and memory footprint. However, the primary problem with a fixed pattern is that it may lose the fine-grained important features and dependencies in the input sequence. For example, the attention mechanisms utilized in Longformer and BigBird have limitations when applied to image classification tasks. Since image data is typically shaped in two dimensions (except for RGB channels), neighboring image pixels (or patches) arranged vertically do not appear side by side in sequential data. Consequently, the sliding window attention-based Longformer is unable to recognize connections between vertically adjacent data points, as it attends only to neighbors within the sequence. Moreover, the global attention used in BigBird only focuses on specific tokens. However, it is possible that there can be other data points besides these specific tokens that are important to all the other data points in a sequence. In such cases, models based on fixed sparse patterns may lose critical information from the data points in the sequence.

**Data-Driven Sparse Pattern.** Sparse patterns in the MHA operation can also be generated by leveraging a data-driven approach, which clusters and sorts the data points of the input sequence during training. Most recently, the LSG Attention [4] extends pretrained RoBERTa [11] for fine-tuning various language related tasks and outperforms Longformer and BigBird models. LSG Attention dynamically captures data patterns by generating sparse masks by utilizing local, sparse, and global attention mechanisms. However, since LSG Attention generates sparse masks at every training step, it results in additional computational overhead during training. Reformer [9] utilizes locality-sensitive hashing to calculate a hash-based similarity and cluster multiple data points into chunks. Similarly, Routing Transformer [16] performs k-means clustering given data points. In the process of training the model, these clustering-based attention approaches also require learning additional functions to identify the relevant dependencies of data points in the input sequence. However, even though utilizing data-driven sparsification techniques during training produces a high quality model, this approach requires additional parameters and operations to learn the sparse patterns, resulting in larger memory space and higher computational cost.

## 3 Motivation: Analysis of Sparse Patterns in MHA

In order to identify common sparsity patterns in the MHA operation, we conducted experiments on the encoder-only Transformer [6] pretrained with the ImageNet-21k and ImageNet-1k datasets. Hereafter, we denote $A^s$ as the attention score matrix obtained after computing $softmax(Q \times K^T / \sqrt{(D/H)})$ in MHA operation. Figure 1 shows $A^s$ from different encoder layers during the inference. Since the sparsity patterns of multiple $A^s$ within the same encoder layer typically

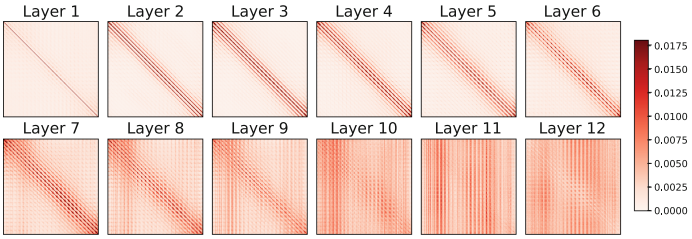


**Fig. 1.** Sparsity patterns in the attention score matrices across different encoder layers during inference of the encoder-only Transformer for image classification.

show similar patterns, we averaged the $A^s$ across multiple heads in each encoder layer. The results clearly show that most of the elements in $A^s$ are close to zero, indicating that only a few data points in the input sequence are correlated to each other. In practice, a number of studies have shown that considering only the critical data points does not adversely affect the convergence of the model [2,14,20]. The characteristics of attention score matrices $A^s$ described below motivate us to develop a new layer-wise sparse attention algorithm.

**Shape of Sparse Pattern.** As shown in Fig. 1, the attention score matrices produced by the different encoder layers exhibit distinct sparsity patterns. For example, in the first to ninth encoder layers, the diagonal elements have relatively large values, similar to a band matrix that stores nonzeros within the diagonal band of the matrix. It is obvious that the MHA operation relies on the self-attention scheme and therefore, the resulting values of the dot-product between linearly transformed vectors for the same data points tend to be larger compared to the resulting outputs produced with different data points. In addition to the diagonal sparsity pattern, encoder layers 10, 11 and 12 show a vertical sparsity pattern, with nonzeros mostly stored in specific columns. This vertical sparsity pattern emerges when the attention operation focuses on the similarity between all data points in $Q$ and particular data points in $K$. In light of these observations, applying the same fixed sparse pattern to all layers may lead to the exclusion of unique essential features that need to be captured individually at different layers. Hence, it is crucial to consider layer-wise sparsification of the MHA based on the sparse pattern observed across different layers. Furthermore, it is necessary to generate domain-specific flexible sparse patterns for various tasks and datasets.

**Degree of Sparsity.** Across different encoder layers, there exists variation not only in the shape of the sparse pattern but also in the number of nonzero elements in attention score matrices. For example, layer 12 has a higher number of nonzero elements compared to layer 1, indicating that layer 12 extensively computes the MHA operation using a larger number of data points in the sequence. Hence, it is crucial to consider varying degrees of sparsity for every encoder layer.

Considering the irregular distribution of non-zero entries in different attention matrices is essential for effectively reducing computational operations while preserving key features across distinct encoder layers.

## 4 SPION: Layer-Wise Sparse Attention in Transformer

In this section, we provide a high-level overview and details of our new SPION that dynamically sparsifies the MHA operation, incorporating the major considerations described in Sect. 3.

### 4.1 Overview of SPION

In SPION, the overall training process is decoupled into three phases: dense-attention training, sparsity pattern generation, and sparse-attention training. Our SPION is capable of sparsifying the MHA operation after training the model for a few steps with dense-attention training. The dense-attention training follows the same training procedure as the original Transformer, without sparsifying the MHA operation. The original dense MHA continues until the attention score matrix $A^s$ of each encoder layer exhibits a specific sparsity pattern. In order to determine the end of the dense-attention training or the start of the sparse-attention training, for each step, we first measure the Frobenius distance between the $A^s$ produced in the previous step $i-1$ and the current step $i$ as defined in Eq. 2.

$$distance_i = \left| \sqrt{\sum (A^s_{i-1})^2} - \sqrt{\sum (A^s_i)^2} \right| \tag{2}$$

Then, we compare the previous $distance_{i-1}$ with the current $distance_i$ to ensure whether a common pattern of nonzeros has emerged. Intuitively, when the difference between $distance_{i-1}$ and $distance_i$ is very small, it is possible to assume that $A^s$ ends up with a specialized sparsity pattern. Hence, if the difference in Frobenius distance between the previous step and the current step is less than a threshold value, we cease the dense-attention training phase. Thereafter, we dynamically generates the sparsity pattern matrix $P$ for each encoder layer based on our novel convolutional flood-filling scheme, as described in Sect. 4.2.

After identifying the sparsity pattern, SPION proceeds with the sparse-attention training phase until convergence by adapting the sparsity pattern. Given the matrices $Q$ and $K$, along with the sparsity pattern matrix $P$, the SDDMM (Sampled Dense-Dense Matrix Multiplication) operation is utilized to accelerate producing the sparsified attention score matrix. Next, the sparse attention score matrix is used to apply the sparse softmax function. After computing the sparse softmax operation, since the sparsified attention score matrix $S^s$ remains sparse, we utilize SpMM (Sparse-Dense Matrix Multiplication) operation to obtain final attention matrix $S$ by multiplying the sparse matrix $S^s$ with the dense matrix $V$. To accelerate both SDDMM and SpMM operations

on GPUs, we utilize the cusparseSDDMM() and cusparseSpMM() functions provided by the NVIDIA cuSPARSE library [13]. Moreover, we implement a custom CUDA kernel for the sparse softmax function by leveraging warp-level reduction to accelerate the sparse-attention training phase.

### 4.2 Sparsity Pattern Generation with Convolutional Flood Fill Algorithm

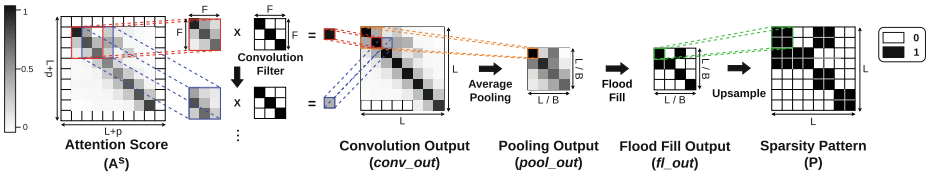


**Fig. 2.** Overview of our convolutional flood filling method for generating the sparsity pattern in the attention score matrix.

To precisely identify the sparsity patterns in the attention score matrix $A^s$, we develop a new convolutional flood fill algorithm that extensively explores the shape, degree and locality of sparsity patterns in $A^s$ for each encoder layer. Figure 2 shows the overview for generating the sparsity pattern in $A^s$. An initial step is to apply a diagonal convolution filter to $A^s$ in order to identify the shape of sparsity pattern. If the diagonal elements in $A^s$ have larger values compared to the others, applying a diagonal convolution filter increases the values of the diagonal elements in the convolution output (*conv_out*). This leads to the emergence of a diagonal sparsity pattern. Otherwise, if the off-diagonal elements, especially the vertical ones, in $A^s$ have larger values compared to the others, applying a diagonal convolution filter results in a vertical sparsity pattern in *conv_out* matrix. In order to ensure that $A^s$ and *conv_out* have the same size ($L \times L$), we adopt zero-padding to $A^s$ during computing the convolution operation defined in Eq. 3.

$$conv\_out(i,j) = \sum_{f=1}^{F} A^s(i+f, j+f) \times filter(f,f) \tag{3}$$

After generating the *conv_out* through a diagonal convolution operation, our algorithm performs average pooling on the *conv_out* using a kernel/block of size ($B \times B$) as defined in Eq. 4.

$$pool\_out\left(\frac{i}{B}, \frac{j}{B}\right) = \frac{1}{B^2} \sum_{p=1}^{B} \sum_{q=1}^{B} conv\_out(i+p, j+q) \tag{4}$$

Instead of analyzing the sparsity pattern of $A^s$ element by element, applying average pooling enables capturing block sparsity pattern, which considers both

the critical data points and their surrounding data points. Hence, since the output of average pooling (*pool_out*) has a smaller size ($L/B \times L/B$) compared to the attention score matrix $A^s$, *pool_out* can be considered as a block-wise abstract sparsity representation of $A^s$.

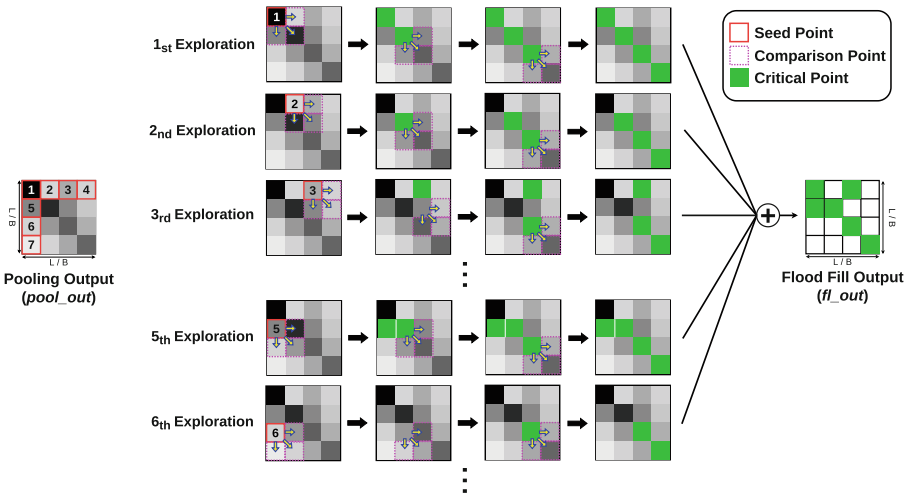


**Fig. 3.** Walk-through example of the identification of critical elements using the flood fill algorithm. (Colour figure online)

In order to dynamically explore the critical elements in the *pool_out*, we develop a novel method inspired by the flood fill algorithm. The flood fill algorithm is originally developed to determine the area connected to a given cell/pixel in a multi-dimensional array and a bitmap image [7]. Unlike the traditional flood fill algorithm, which compares all neighbors of a current element to find the element with the largest value, our scheme only compares to the neighboring elements on the right, below, and diagonally below, as shown in Fig. 3. Essentially, this is because the *pool_out* matrix follows the sequential order from left to right and top to bottom. In the process of capturing the pattern, it is necessary to sequentially follow the important features starting from the top-left corner to the bottom of the matrix. All the elements in the first row and all the elements in the first column of the *pool_out* are used as the seed starting points. From the particular seed point, our algorithm compares the values of elements to its right, below, and diagonally below to extract the element with the largest value in order to check whether the neighbors of the current element are relevant. If the neighboring elements are not relevant to the current element, our algorithm moves to the element diagonally below to continue comparing the neighbors. When the largest value is greater than a predefined threshold, we determine the corresponding element as the critical element (green colored elements in Fig. 3). Here, the threshold is determined by calculating the $\alpha\%$ quantile of *pool_out*. By

utilizing the threshold, our flood fill algorithm ensures that the values of selected critical elements are sufficiently large. After determining the critical elements, our algorithm recursively compares the values of elements to the right, below, and diagonally below the critical element while avoiding duplicate comparisons with elements that have already been selected as critical points. This process is repeated until the selected critical element reaches the end of a row or column in the matrix *pool_out*. After conducting explorations from all seed points, the resulting critical points from each exploration are combined into a single matrix called the flood fill output ($fl\_out$). Therefore, the $fl\_out$ can be considered as an explicit sparsity pattern captured from *pool_out*. $fl\_out$ also represents the compressed block-level sparsity pattern of $A^s$ since the average pooling operation is applied before processing the flood fill algorithm.

Next, to utilize newly generated sparse pattern of $fl\_out$ in the sparse MHA operation, the size of the $fl\_out$ needs to be the same as the size of the attention score matrix $A^s$. Therefore, we upsample the $fl\_out$ using nearest-neighbor interpolation, resulting in each nonzero element in $fl\_out$ forming a block of nonzero elements in the final sparsity pattern matrix ($P$), as shown on the right of Fig. 2. Utilizing a block sparse matrix $P$ improves the quality of model since it incorporates not only the critical elements but also their surrounding elements for MHA operation. Moreover, an optimized blocked matrix multiplication further enhances performance through improved data locality. In the blocked sparsity pattern matrix $P$, critical elements involving MHA calculation are set to 1, and the rest of the non-critical elements are set to 0. Finally, during the sparse MHA, only the elements of $A^s$ that have the same indices with a value of 1 in $P$ will be computed.

## 5 Experimental Evaluation

This section provides both performance and quality assessments of our SPION implementation. Our SPION is compared with various state-of-the-art sparse Transformer models. All the experiments were run on four NVIDIA RTX A5000 GPUs. Our GPU implementation integrates optimized CUDA kernels and CUDA libraries for sparse-attention training with PyTorch, utilizing the ctypes library (CDLL) provided in Python.

**Datasets and Tasks.** We evaluated our SPION on four tasks: image classification, text classification, ListOps, and document retrieval. For image classification, we utilized the CIFAR-10 [10] and iNaturalist 2018 [8] datasets, with the former having images of 32×32 pixels and the latter resized to this resolution, resulting in a sequence length of 1,024. In text classification, we used the AG News [22] and Yelp Reviews [22] datasets, both having a maximum sequence length of 1,024. The ListOps task, based on the dataset from Nangia et al. [12], involves classifying answers from 0 to 9 from sequences of numbers and operators, with a sequence length of 2,048. For the document retrieval task, we used the AAN dataset [15], classifying whether two given documents are related or not,

with a sequence length of 4,096. All the text data is evaluated at the character level.

**Models Compared.** We compared our SPION with four state-of-the-art sparse Transformers: LSG Attention [4], Longformer [2], BigBird [20] and Reformer [9]. In addition, we evaluated variations of our SPION model: SPION-C, SPION-F and SPION-CF. Specifically, the SPION-C model omits the flood filling scheme and selects the top few percent of block elements after convolution filter and average pooling (*pool_out*), facilitating an adjustable sparsity ratio during generating the $P$ matrix. In contrast, the SPION-F model bypasses the convolution filter, applying the flood fill-based algorithm immediately after the average pooling. The SPION-CF integrates both the convolution filter and the flood fill-based approach while generating a sparsity pattern. For all models, the sizes of configuration (window, bucket, block) were determined by the maximum sequence length of dataset. For example, Longformer used 64 or 128, while all other compared models, including our SPION variants, used 32 or 64.

We set the embedding dimension ($D$) to a size of 64. For image classification and document retrieval tasks, we used two layers, whereas for text classification and ListOps tasks, we employed four layers. The batch size was determined by 512 for image classification and text classification, 256 for ListOps, and 32 for document retrieval. Across all experiments, the size of the convolution filter in SPION was fixed at (31×31). As for the threshold in the flood fill algorithm, we configured $\alpha$ to 96 for image classification and text classification tasks, 98 for ListOps, and 99.5 for the document retrieval task. Note that all experimental results presented in this section are averaged over three distinct runs.

### 5.1 Performance Evaluation

**Convergence.** Table 1 shows the accuracy of six models on four different tasks. Our SPION-CF consistently achieved the highest accuracy in all tasks, surpassing the highest accuracy obtained from other compared models by +0.812%, +0.115%, +0.697%, +2.735%, +0.707%, and +0.205% for the evaluation tasks. It is interesting to see that among the SPION variants, incorporating both the convolution filter and flood-filling scheme led to higher accuracy for all tasks. This indicates that the convolution filter and flood-filling method synergize with each other. Additionally, we observed that the flood filling method shows more significant effect on accuracy compared to the convolution filter. This result demonstrates that considering the connectivity between elements is an important factor when capturing sparse patterns.

**Speedup and Memory Reduction.** Table 2 shows the time and memory required to train each model across four tasks, using the CIFAR-10 dataset for image classification and Yelp-review dataset for text classification. Our SPION-CF, compared to BigBird, achieved speedups of up to 2.78× per training step. Particularly in longer sequence tasks like ListOps and document retrieval,

**Table 1.** Classification accuracy (%) of various tasks

| Model | Image Classification | | Text Classification | | ListOps | Document Retrieval |
|---|---|---|---|---|---|---|
| | CIFAR-10 | iNaturalist | Yelp-Review | AG-News | | |
| LSG Attention | 43.435 | 53.758 | 80.449 | 79.371 | 39.137 | 80.538 |
| Longformer | 42.845 | 54.269 | 80.858 | 75.465 | 39.620 | 80.595 |
| BigBird | 41.978 | 52.964 | 76.813 | 74.177 | 39.658 | 80.396 |
| Reformer | 40.824 | 51.298 | 74.625 | 67.164 | 38.058 | 78.891 |
| SPION-C | 41.355 | 53.759 | 76.971 | 72.796 | 40.290 | 80.266 |
| SPION-F | 43.846 | 53.553 | 81.339 | 81.645 | 40.160 | 80.423 |
| SPION-CF | **44.247** | **54.384** | **81.555** | **82.106** | **40.365** | **80.800** |

SPION showed greater efficiency. Against LSG-Attention, SPION excelled in tasks with shorter sequences, and also required less training time for longer sequences. Table 2 further shows SPION-CF reducing memory footprint by over 7.24× in comparison to BigBird on ListOps task. Our SPION variants consistently had the smallest memory footprints. Remarkably, despite memory savings, SPION-CF maintained the highest accuracy across all tasks.

**Table 2.** Comparison of elapsed time (ms) and memory usage (GB) per step for training on various classification tasks

| Model | Image Classification | | Text Classification | | ListOps | | Document Retrieval | |
|---|---|---|---|---|---|---|---|---|
| | Time | Memory | Time | Memory | Time | Memory | Time | Memory |
| LSG Attention | 97.064 | 12.390 | 233.947 | 31.532 | 197.602 | 31.835 | 104.699 | 11.652 |
| Longformer | 154.511 | 26.418 | 385.044 | 63.022 | 480.873 | 91.038 | 208.290 | 29.645 |
| BigBird | 217.889 | 35.458 | 503.207 | 82.698 | 539.533 | 104.134 | 276.082 | 34.300 |
| Reformer | 138.936 | 18.545 | 354.041 | 49.067 | 309.772 | 50.614 | 144.285 | 19.590 |
| SPION-C | 87.471 | 6.536 | 217.360 | 13.770 | **187.382** | **14.259** | 106.608 | 5.854 |
| SPION-F | **85.298** | 6.550 | 212.343 | **13.745** | 197.928 | 15.267 | 106.111 | 5.877 |
| SPION-CF | 86.257 | **6.530** | **206.387** | 13.769 | 194.256 | 14.368 | **103.395** | **5.799** |

### 5.2 Computational Complexity Analysis

Figure 4 shows the FLOPs (Floating Point Operations) for computing sparse attention matrix $S$ for various sparse Transformers. Note that all sparse Transformers maintain a total of $C$ non-zero elements in the sparse attention matrix $S$. Therefore, the total operations required for computing $S$ for each head in all compared sparse Transformers is the same as $2C(2D + 1) - L(D + 1)$. As shown in Fig. 4, our SPION-CF achieves up to a 5.91× reduction in FLOPs compared to BigBird on the document retrieval task. This result demonstrates the

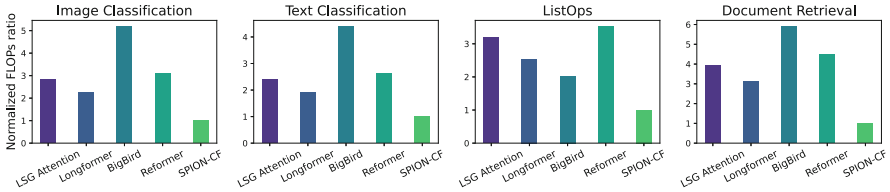


**Fig. 4.** Comparison of FLOPs required for computing the attention score matrix on various tasks. All the results are normalized to SPION-CF.

effectiveness of our convolutional flood-fill attention scheme, which dynamically captures relevant elements specific to the dataset and task. Furthermore, unlike the Longformer and BigBird models that apply the same fixed sparse pattern for every layer, our layer-wise sparse MHA enables the avoidance of unnecessary computations with non-relevant elements at each layer.

## 6 Conclusion

Due to the compute-intensive nature of the Transformer, that must perform a large number of MHA operations, we develop a novel sparsification scheme. This scheme leverages convolution filters and the flood fill algorithm to reduce the overall complexity of MHA operations. Our sparsification technique dynamically identifies the critical elements in the attention score matrix on a layer-wise basis. This method is applicable to many other Transformer models, not limited to the encoder-only Transformer. Experimental results on various datasets demonstrate that our sparse MHA approach significantly reduces training time and memory usage compared to state-of-the-art sparse Transformers, while achieving better accuracy on various classification tasks.

**Acknowledgments.** This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(RS-2024-00259099) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation), and in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1G1A1092597).

## References

1. Ainslie, J., Ontanon, S., et al.: Etc: encoding long and structured inputs in transformers. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 268–284 (2020)
2. Beltagy, I., Peters, M.E., Cohan, A.: Longformer: the long-document transformer. arXiv preprint arXiv:2004.05150 (2020)
3. Child, R., Gray, S., Radford, A., Sutskever, I.: Generating long sequences with sparse transformers. arXiv preprint arXiv:1904.10509 (2019)

4. Condevaux, C., Harispe, S.: Lsg attention: extrapolation of pretrained transformers to long sequences. In: Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 443–454 (2023)
5. Devlin, J., Chang, M.W., et al.: Bert: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (2019)
6. Dosovitskiy, A., Beyer, L., et al.: An image is worth 16×16 words: transformers for image recognition at scale. In: International Conference on Learning Representations (2021)
7. Goldman, R.: Graphics gems, p. 304 (1990)
8. iNaturalist 2018 competition dataset. (2018)
9. Kitaev, N., Kaiser, Ł., Levskaya, A.: Reformer: the efficient transformer. In: Proceedings of the International Conference on Learning Representations (2020)
10. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images (2009)
11. Liu, Y., Ott, M., et al.: Roberta: a robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 (2019)
12. Nangia, N., Bowman, S.R.: Listops: a diagnostic dataset for latent tree learning. arXiv preprint arXiv:1804.06028 (2018)
13. Nvidia: the api reference guide for cusparse, the cuda sparse matrix library. Technical report (2023). https://docs.nvidia.com/cuda/cusparse/index.html
14. Qiu, J., Ma, H., et al.: Blockwise self-attention for long document understanding. In: Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 2555–2565 (2020)
15. Radev, D.R., Muthukrishnan, P., et al.: The ACL anthology network corpus. Lang. Res. Eval. **47**, 919–944 (2013)
16. Roy, A., Saffar, M., et al.: Efficient content-based sparse attention with routing transformers. Trans. Assoc. Comput. Linguist. **9**, 53–68 (2021)
17. Tay, Y., Dehghani, M., et al.: Efficient transformers: a survey. ACM Comput. Surv. **55**(6), 1–28 (2022)
18. Vaswani, A., Shazeer, N., et al.: Attention is all you need. Adv. Neural Inf. Process. Syst. **30** (2017)
19. Wang, S., Li, B.Z., et al.: Linformer: self-attention with linear complexity. arXiv preprint arXiv:2006.04768 (2020)
20. Zaheer, M., Guruganesh, G., et al.: Big bird: transformers for longer sequences. Adv. Neural. Inf. Process. Syst. **33**, 17283–17297 (2020)
21. Zhang, H., Gong, Y., et al.: Poolingformer: long document modeling with pooling attention. In: International Conference on Machine Learning, pp. 12437–12446. PMLR (2021)
22. Zhang, X., Zhao, J., LeCun, Y.: Character-level convolutional networks for text classification. Adv. Neural Inf. Process. Syst. **28** (2015)