Proceedings of Machine Learning Research **318** (2026)
The 39th Canadian Conference on Artificial Intelligence

`blksprs`: A Triton Library for Block-Sparse Matrix Operations

Felix Schön* and Hans Tompits**
Institute of Logic and Computation E192-03,
Technische Universität Wien,
Favoritenstraße 9-11, 1040 Vienna, Austria

Abstract

In this paper, we introduce `blksprs`, a Triton-based PyTorch library for block-sparse matrix operations designed for machine-learning approaches. In contrast to existing approaches, `blksprs` supports a significantly wider range of operations, including but not limited to matrix multiplication, softmax, gather, scatter, transposition, (interleaved) repeat, and more. Furthermore, it supports flexible sparsity specification for all input and output matrices of these operations. These features facilitate applications that would have previously been infeasible. We provide a formal specification of `blksprs` and demonstrate that `blksprs` can consistently outperform standard PyTorch and existing Triton implementations. In a practical evaluation, we were able to reduce training time by up to 35% and memory consumption by up to 45% when employing `blksprs` for the training of a Transformer neural network, with minimal implementation overhead.

Keywords: Block-Sparse Matrices, Triton, Transformer, GPU Acceleration

1. Introduction

In recent years, the *Transformer* architecture [1] significantly impacted a large number of fields, including, e.g., natural language processing (NLP) [2–4], computer vision [5, 6], speech processing [7, 8], multimodal learning [3, 9], reinforcement learning [10], and protein structure prediction [11].

At the heart of the Transformer architecture lies the *attention module* [1, 12, 13], which computes similarity scores between pairs of vector representations of input tokens. Computing these similarity scores requires multiplying large matrices, which can be computationally expensive, especially for long input sequences. As a result, recent research has focused on designing more time- or space-efficient methods, such as approximate or structured attention mechanisms. These include techniques like locality-sensitive hashing used in the Reformer [14], low-rank projection in the Linformer [15], and kernel-based optimisation of memory access as used in the FlashAttention [16–18] algorithm.

A promising strategy comes in the form of *block-sparsity* aware operations, as used, e.g., by the *Sparse Transformer* [19] or the *Longformer* [20]. Block-sparse matrices are structured in such a way as to contain regular blocks of elements with a value of 0, enabling predictable sparsity patterns. Unlike sparsity-agnostic implementations, these patterns can be exploited to design functions that compute only the necessary blocks, significantly reducing memory and computational overhead. Previous implementations allowing for these operations include, e.g., OpenAI's block-sparse GPU kernels [21], the kernels included with the Triton [22] language, DeepSpeed Sparse Attention [23], and lower-level libraries such as NVIDIA's cuSPARSE [24]. Existing approaches typically focus on a narrow set of operations, often matrix multiplication or sparse attention only, whereas `blksprs` provides a broader collection of block-sparse operations for machine-learning workflows.

In this paper, we introduce `blksprs`, a Triton-based PyTorch library [22] for block-sparse matrix manipulation operations. In contrast to previous approaches, `blksprs` supports

*schoen@kr.tuwien.ac.at **tompits@kr.tuwien.ac.at

This article is © 2026 by author(s) as listed above. The article is licensed under a Creative Commons Attribution (CC BY 4.0) International license (https://creativecommons.org/licenses/by/4.0/legalcode), except where otherwise indicated with respect to particular material included in the article. The article should be attributed to the author(s) identified above.

a significantly wider range of operations, including matrix multiplication, gather/scatter, split/merge, transposition, and conversion to/from block-sparse representation. Incorporating `blksprs` into an existing codebase requires less engineering effort than stock Triton kernels, as the required auxiliary matrices can be computed on-the-fly with minimal overhead. In contrast to the latter, our library is flexible in its operation, e.g., supporting block-sparsity for all input and output matrices, or adaptation of sparsity layouts and sizes, as we specifically designed `blksprs` with machine-learning purposes in mind. Furthermore, `blksprs` adheres to modern standards, ensuring wide compatibility for different hardware, and fully supports automatic casting between 32-bit and 16-bit float input.

In an experimental evaluation, we show that these advantages do not come at the cost of efficiency: We compare `blksprs` to regular, block-sparsity agnostic PyTorch operation baselines, and to the stock Triton kernels for block-sparse operations. Here, we show that `blksprs` can markedly outperform existing block-sparse implementations, as well as their regular counterparts. Furthermore, we show that by replacing regular attention in a Transformer model with its block-sparse counterpart, training time can be reduced by up to 35% and memory consumption by up to 45%, with minimal engineering effort.

The paper is organised as follows: In Section 2, we provide a brief account of the required mathematical preliminaries and basic notation. Section 3, then, introduces `blksprs`, including the formal specifications of the operations contained in it. In Section 4, we evaluate the performance of `blksprs` compared to existing alternatives and conduct further experiments. Section 5 concludes the paper.

2. **Preliminaries**

As our subsequent discussion involves the concepts of block-sparse matrices, tensors, and operations on these objects, we begin by covering some basic mathematical conventions and notation relevant to these notions.

As we use subscripts for both naming and indexing, in order to improve clarity, we make use of the following notation to index scalar elements, as well as vector and matrix slices of matrices and tensors, respectively: Given a matrix $M = (m_{ij}) \in \mathbb{R}^{m \times n}$ and a tensor $T = (t_{hij}) \in \mathbb{R}^{l \times m \times n}$, we use the notation $M_{[i,j]}$ and $T_{[h,i,j]}$ to refer to the scalar elements m_{ij} and t_{hij}, respectively. Furthermore, we extend this notation to refer to vector slices of matrices, and both vector and matrix slices of tensors as follows: Given M and T as above, $M_{[i]}$ refers to the vector $(M_{[i,1]}, \dots M_{[i,n]})$, $T_{[h,i]}$ refers to $(T_{[h,i,1]}, \dots T_{[h,i,n]})$, and $T_{[h]}$ refers to the matrix $M^h = (m^h_{ij})$, where $M^h_{[i,j]} = T_{[h,i,j]}$. We also apply this notation to lists and their elements, e.g., for $L = ((a, b), \dots)$, then $L_{[1,1]} = a$. As usual, $M^\intercal$ denotes the transpose of a matrix M.

Next, we define the notions of *concatenating* and *stacking* vectors and matrices. Given vectors $v^1 \in \mathbb{R}^n$ and $v^2 \in \mathbb{R}^q$, as well as matrices $M^1 \in \mathbb{R}^{m \times n}$ and $M^2 \in \mathbb{R}^{m \times q}$, we define the concatenation operation thus:

$$v^1 \parallel v^2 := (v^1_{[1]}, \dots, v^1_{[n]}, v^2_{[1]}, \dots, v^2_{[q]}) \text{ and } M^1 \parallel M^2 := M^c = (m^c_{ij}), \text{ where}$$

$$m^c_{ij} = \begin{cases} M^1_{[i,j]}, & \text{if } j \leq n, \\ M^2_{[i,g]} & \text{otherwise, where } g = j - n. \end{cases}$$

Furthermore, given k matrices $M^1, \dots, M^k$ (or analogously, vectors), we define the following shorthand:

$$\parallel_{s=1}^{k} M^s := M^1 \parallel (\parallel_{t=s+1}^{k} M^t).$$

The *stacking* of two or more vectors or matrices can then be seen as the concatenation along a new dimension. Given k vectors and matrices, $v^1, \dots, v^k \in \mathbb{R}^n$ and $M^1, \dots, M^k \in \mathbb{R}^{m \times n}$, respectively, we define the *stack* operation as follows:

$$stack(v^1, \ldots, v^k) := M^s = (m^s_{ij}), \text{ where } m^s_{ij} = v^i_{[j]}, \text{ and}$$
$$stack(M^1, \ldots, M^k) := T^s = (t^s_{hij}), \text{ where } t^s_{hij} = M^h_{[i,j]}.$$

3. The `blksprs` Library

We now introduce our `blksprs` library for block-sparse matrix manipulation operations, which is based on the Triton back end for PyTorch [22]. We first lay down the formal specification of `blksprs` and afterwards we discuss a variety of operations facilitated by it.

3.1. Block-Sparse Matrices

The following definition adopts the notion of block-sparse matrices as used, e.g., by the Sparse Transformer [19] or the Longformer [20].

Definition 1. *A matrix $M = (m_{ij}) \in \mathbb{R}^{m \times n}$ is a block-sparse matrix with sparsity block size σ_M and sparsity layout L_M if*

(i) $\sigma_M \in \{2^p \mid p \geq 4, m \bmod 2^p = 0, n \bmod 2^p = 0\}$,
(ii) $L_M = (l_{ij}) \in \{0,1\}^{\frac{m}{\sigma_M} \times \frac{n}{\sigma_M}}$, *where* $\frac{m}{\sigma_M}, \frac{n}{\sigma_M} \in \mathbb{N}$, *and*
(iii) $L_{M\,[i,j]} = 0$ *if and only if* $M_{[e,f]} = 0$, *for each* e *and* f *such that* $(i-1) \cdot \sigma_M < e \leq i \cdot \sigma_M$ *and* $(j-1) \cdot \sigma_M < f \leq j \cdot \sigma_M$.

Intuitively, a value of 0 in L_M indicates a sparse (containing zeroes only) block in M. Note that the requirement of $p \geq 4$ stems from a technical requirement of the Triton language.

Given a block-sparse matrix as in Definition 1, we next introduce the notion of a *compressed representation* of such a matrix. In order to do this, we first define the *lookup table* (*LUT*) U_M based on the sparsity blocks defined by the sparsity layout L_M of a given block-sparse matrix M.

Definition 2. *Let $M = (m_{ij}) \in \mathbb{R}^{m \times n}$ be a block-sparse matrix and L_M its sparsity layout as defined in Definition 1. Furthermore, let $n^{blk}_M = \sum_{i,j} L_{M\,[i,j]}$ be the amount of non-sparse blocks in M, $S = \{(a,b) \in \{1,\ldots,m\} \times \{1,\ldots,n\} \mid L_{M\,[a,b]} = 1\}$ a set of vectors indicating these blocks in M, and $<$ a total ordering of these vectors given by*

$$(a,b) < (c,d) \;:\Longleftrightarrow\; \begin{cases} a < c, \\ \text{or } a = c,\ b < d. \end{cases}$$

Then, $U_M = (x_1, x_2, \ldots, x_{n^{blk}_M})$, with $x_i \in S$, for all $i \in \{1, \ldots, n^{blk}_M\}$, and $x_1 < x_2 < \cdots < x_{n^{blk}_M}$, is the lookup table with respect to the block-sparse matrix M and its sparsity layout L_M.

Definition 3. *Given a block-sparse matrix $M = (m_{ij})$ with sparsity block size σ_M, sparsity layout L_M, and lookup table U_M, the tensor $\overline{M} = (\overline{m}_{hij}) \in \mathbb{R}^{n^{blk}_M \times \sigma_M \times \sigma_M}$ is a compressed representation of M if $\overline{M}_{[h,i,j]} = M_{[e,f]}$, where $e = (U_{M\,[h,1]} - 1) \cdot \sigma_M + i$ and $f = (U_{M\,[h,2]} - 1) \cdot \sigma_M + j$.*

In order to reconstruct a regular, non-compressed block-sparse matrix from its compressed representation, we require the notion of a *reverse lookup table*.

Definition 4. *For a block-sparse matrix $M = (m_{ij}) \in \mathbb{R}^{m \times n}$, let $L^f_M \in \mathbb{R}^{e \cdot f}$, with $e = \frac{m}{\sigma_M}$ and $f = \frac{n}{\sigma_M}$, be a flat version of the sparsity layout L_M, defined as*

$$L^f_M = (L_{M\,[1,1]}, L_{M\,[1,2]}, \ldots, L_{M\,[e,f]}),$$

and let $csum_i = \sum_{j=1}^{i} L^f_{M\,[j]}$, for $1 \leq i \leq e \cdot f$, be the cumulative sum up to the ith value.

Then, the reverse lookup table of M is given by $U^r_M = (u_1, \ldots, u_{e \cdot f})$, where

$$u_i = \begin{cases} csum_i, & \text{if } L^f_{M[i]} = 1, \\ -1 & \text{otherwise,} \end{cases}$$

for $1 \leq i \leq e \cdot f$.

We can then use this reverse lookup table to convert between a compressed representation and a regular representation of a block-sparse matrix.

Definition 5. *Given a block-sparse matrix in compressed form $\overline{M} = (\overline{m}_{hij}) \in \mathbb{R}^{n^{blk}_M \times \sigma_M \times \sigma_M}$ with sparsity block size σ_M, sparsity layout L_M, and reverse lookup table U^r_M, we define its non-compressed counterpart as the block-sparse matrix $M = (m_{ij}) \in \mathbb{R}^{m \times n}$, where*

$$m_{ij} = \begin{cases} \overline{m}_{def} & \text{if } d \neq -1, \text{ where } d = U^r_{M[x]},\ e = i \bmod \sigma_M,\ f = j \bmod \sigma_M, \\ & \qquad \text{with } x = \left\lceil \frac{j}{\sigma_M} \right\rceil + \left(\left\lceil \frac{i}{\sigma_M} \right\rceil - 1 \right) \cdot \frac{n}{\sigma_M}, \\ 0 & \text{otherwise.} \end{cases}$$

3.2. **Block-Sparse Matrix Operations**

Having Definitions 1 to 5 at hand, we now define some of the operations supported by `blksprs`. Recall that we denote the compressed version of a block-sparse matrix M by $\overline{M}$. Note that here, the exact dimensions of $\overline{M}$ are dependent on both the sparsity block size and the overall number of non-sparse blocks in M, i.e., $\overline{M} \in \mathbb{R}^{n^{blk}_M \times \sigma_M \times \sigma_M}$. We omit the exact specification for compressed block-sparse matrices over the next section for the sake of brevity. Furthermore, we assume that sparsity block size, which we henceforth denote by σ, is shared across all matrices of a given operation, as `blksprs` fully supports adapting given compressed block-sparse matrices to new sparsity block sizes.

Let us first define some ancillary functions to improve clarity. We start with the following function ϕ which only returns a value from a (potential) block of $\overline{M}$ if the block is not marked as sparse by U^r_M:

$$\phi(v, \overline{M}, h) := \begin{cases} v & \text{if } U^r_{M[h]} \neq -1, \\ 0 & \text{otherwise.} \end{cases}$$

Given sparsity and reverse-sparsity layouts U_M and U^r_N, we next define the auxiliary functions *lr*, *rlr*, and *rlc*. The function *lr* retrieves the reverse sparsity layout value indexed by a given sparsity block defined by h, where b is the number of sparsity blocks in a row of the sparsity layout. Furthermore, *rlr* and *rlc* retrieve the reverse sparsity layout values of a block in the same row (respectively column) as the sparsity block indexed by h. By specifying the respective parameters, using *rlr*, we can index the block in the cth column of the sparsity layout sharing the row of block indexed by h. Analogously, using *rlc*, we can index the block in the rth row sharing the same column as the block indexed by h. The definitions of the functions *lr*, *rlr*, and *rlc* are as follows:

$$\begin{aligned} lr(U_M, U^r_N, h, b) &= U^r_{N[U_{M[h,1]} * b + U_{M[h,2]}]}, \\ rlr(U^r_N, U_M, h, c, b) &= U^r_{N[c + (U_{M[h,1]} - 1)b]}, \text{ and} \\ rlc(U^r_N, U_M, h, r, b) &= U^r_{N[rb + (U_{M[h,2]} - 1)]}. \end{aligned}$$

3.2.1. **Block-Sparse Matrix Multiplication**

The `blksprs` library supports matrix multiplication between two block-sparse matrices. Here, a target sparsity layout can be specified, resulting in the computation of the required output blocks only.

Given the compressed block-sparse matrices $\overline{A}$ and $\overline{B}$, with the non-compressed counterparts $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, their sparsity layouts and reverse lookup tables L_A, U_A^r, L_B, U_B^r, and the target sparsity layout and lookup table L_C, U_C, we define the block-sparse matrix multiplication as follows:

$$\mathit{matmul}_{bs}(\overline{A}, \overline{B}) := \overline{C} = (\overline{c}_{hij}), \text{ where}$$

$$\overline{c}_{hij} = \sum_{r=1}^{n/\sigma} \sum_{s}^{\sigma} \phi(\overline{A}_{[x,i,s]}, \overline{A}, x) \phi(\overline{B}_{[y,s,j]}, \overline{B}, y),$$

for $x = \mathit{rur}(U_A^r, U_C, h, r, n/\sigma)$ and $y = \mathit{ruc}(U_B^r, U_C, h, r, n/\sigma)$.

3.2.2. **Block-Sparse Matrix Softmax**

The `blksprs` library supports two distinct softmax formulations. We include kernels for a conventional block-sparse softmax that computes the row-wise maximum and row-wise sum over all sparsity blocks and uses them to compute the softmax on a per-block basis, and a "fused" version where we first determine the maximum amount of non-sparse blocks in any row, and then compute the softmax over rows of this size. As the regular block-sparse softmax function is significantly slower for all practical purposes, we discuss the fused version only.

Let n_{mbl} be the maximum amount of blocks in a row for all rows in the sparsity layout U_L. Furthermore, let U_L^s be a version of U_L^r where the reverse sparsity indices are sorted descendingly within what corresponds to a row in the sparsity layout. Informally, this is required to be able to index the first n_{mbl} reverse indices within what corresponds to a row and ensure that no indices pointing to non-sparse blocks follow them.

For an uncompressed block-sparse matrix $M \in \mathbb{R}^{m \times n}$, we first define the vectors u^i which contain exactly the non-sparse values of each row in M and enough zeroes after that to pad them to a length of n_{mbl}:

$$u^i \in \mathbb{R}^{n_{mbl}} = (u_j^i), \text{ for } i \in (1, \dots, m), \text{ where}$$

$$u_j^i = \phi(\overline{M}_{[x,(i-1 \bmod \sigma)+1,(j-1 \bmod \sigma)+1]}, \overline{M}, x), \text{ for } x = U^s_{L[i \cdot n/\sigma + \lfloor \frac{j-1}{\sigma} \rfloor]}.$$

Using these extracted rows, we can then define the fused block-sparse softmax function as follows:

$$\mathit{softmax}_{bsf}(\overline{M}) = \overline{M'} := (\overline{m'}_{def}), \text{ where}$$

$$\overline{M'}_{[x,(i-1 \bmod \sigma)+1,(j-1 \bmod \sigma)+1]} = \mathit{softmax}(v^i)_{[j]},$$

with i, j, and $x = U^s_{L[i \cdot n/\sigma + \lfloor \frac{j-1}{\sigma} \rfloor]}$ as defined as above. Moreover, for a vector $v \in \mathbb{R}^n$, we can then define the *softmax* function as follows:

$$\mathit{softmax}(v) := (v_i'), \text{ where}$$

$$v_i' = \frac{e^{v_i}}{\sum_{r=1}^{n} e^{v_r}}.$$

Intuitively, we first extract a vector slice u^i from the block-sparse matrix, compute the softmax over its entries, and then write the computed values back to their original positions.

3.2.3. **Block-Sparse Transpose**

In order to define the block-sparse transpose function, we first introduce the flow_{bs} function, which reorders the sparse blocks of a compressed block-sparse matrix according to a reverse sparsity layout:

$$\mathit{flow}_{bs}(\overline{M}, U_M, U_N^r) = \overline{M'} = (\overline{m'}_{hij}), \text{ where}$$

$$\overline{m'}_{hij} = \overline{M}_{[x,i,j]}, \text{ for } x = lr(U_M, U_N^r, h, n/\sigma).$$

Using this ancillary function, we can now easily define the block-sparse transposition, as it can be reduced to first independently transposing the blocks of $\overline{M}$, transposing its original sparsity layout, and then reshuffling the blocks based on the reverse sparsity layout of the transposed sparsity layout:

$$\mathit{transpose}_{bs}(\overline{M}) = \mathit{flow}_{bs}(\overline{M}^\intercal, U_M, U_{M^\intercal}^r).$$

Note that we treat the transpose of a tensor as the transpose of all its matrix elements, i.e., for a tensor $M \in \mathbb{R}^{d \times e \times f}$,

$$\overline{M}^\intercal = \overline{M'}, \text{ where}$$
$$\overline{M'}_{[r]} = (\overline{M}_{[r]})^\intercal, \text{ for } 1 \leq r \leq d.$$

3.2.4. **Block-Sparse Matrix Gather**

Given a block-sparse source matrix $S \in \mathbb{R}^{m \times q}$, index matrix $I \in \{1, \ldots, q\}^{m \times n}$, and their sparsity and reverse sparsity layouts, we define the block-sparse gather operation. Here, the elements $I_{[e,f]}$ of the index matrix point to elements of S to gather at $T_{[e,f]}$:

$$\begin{aligned} \mathit{gather}_{bs}(\overline{S}, \overline{I}) := \overline{T} = (\bar{t}_{hij}), \text{ where} \\ \bar{t}_{hij} = \phi(\overline{S}_{[d,i,f]}, \overline{S}, d), \text{ for } d = U_{M_{\left[\left\lceil \frac{I_{[h,i,j]}}{\sigma} \right\rceil\right]}}^r, f = I_{[h,i,j]} \bmod \sigma. \end{aligned}$$

3.2.5. **Block-Sparse Matrix Scatter**

Given a source matrix $S \in \mathbb{R}^{m \times n}$, an index matrix $I \in \{1, \ldots, q\}^{m \times n}$, and a target matrix $T \in \{0\}^{m \times q}$, we define the block-sparse scatter operation as follows:

$$\begin{aligned} \mathit{scatter}_{bs}(\overline{S}, \overline{I}) := \overline{T} = (\bar{t}_{hij}), \text{ where} \\ \bar{t}_{hij} = \sum_{r}^{n/\sigma} \sum_{s}^{\sigma} \overline{S}_{[x,i,s]} \delta(\overline{I}_{[x,i,s]}, j \cdot U_{T[h,2]}), \text{ for } x = rlr(U_S^r, U_T, h, r, n/\sigma). \end{aligned}$$

Here, $\delta(x, y)$ is the indicator function that equals 1 if $x = y$ and 0 otherwise.

3.3. **Block-Sparse Tensors and Further Functionalities**

In practice, `blksprs` performs operations on block-sparse matrices *in parallel*, in effect representing tensors. In fact, the definitions laid down in the previous section can be generalised to *block-sparse tensors*. However, due to space reasons, we will not discuss this generalisation here.

We also note that the `blksprs` library contains further functionalities designed for machine-learning applications, which we also do not detail here. We only mention that these functionalities include (i) block-sparse repeat, interleaved repeat, split, and merge operations, (ii) sparsity layout adaptation, including conversion between sparsity block sizes, (iii) row-wise operations such as row-wise sum, addition, and maximum, (iv) broadcast addition, (v) efficient sparsity layout creation, (vi) wrapper functions for PyTorch modules such as, e.g., dropout, linear, or normalisation layers, and (vii) validation for combinations of block-sparse tensors and their sparsity layouts.

Furthermore, although not explicitly discussed here, `blksprs` contains kernels and functionalities for backwards passes for most of the functions introduced. This allows for practical use of the library in machine-learning tasks, as automatic gradient computation is supported. Exceptions include helper functions, such as, e.g., row-wise operations, or broadcast operations.

3.4. Practical Advantages of `blksprs`

In addition to the advantages outlined in the previous sections, we want to highlight further merits of the `blksprs` library.

Whereas the Triton functions need to be re-instantiated for every change in sparsity layout, with our `blksprs` library we support automatic sparsity LUT creation for any supported input. In addition to this ease of handling, we allow for manual specification of LUTs to improve performance.

As mentioned previously, the `blksprs` library contains a number of useful auxiliary functions, such as converting from and to a compressed representation, conversion between layouts, wrapper functions, etc. We designed `blksprs` with research purposes in mind, aiming for a high amount of flexibility in order to allow for a wide range of applications.

We furthermore adhered to current PyTorch standards, ensuring wide compatibility. In contrast to the Triton implementations, `blksprs` fully supports automatic casting between 32-bit and 16-bit float input to further improve performance and memory requirements. Although we did not include benchmarks on these more efficient computations, we report that `blksprs` shows very good promise here too.

Lastly, in contrast to other block-sparse implementations making use of the CUDA[1] library [21], Triton is not bound to a specific GPU vendor. This means that in theory `blksprs` could support a wider range of hardware, although we were not able to test this due to lack of access to such cards.

4. Experiments and Evaluation

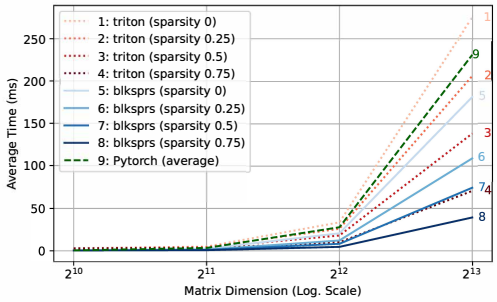
This section covers our experiments performed on the `blksprs` library. Beyond these benchmarks, we conducted extensive correctness tests across various configurations (dimensions, batch sizes, sparsity levels), with special attention to edge cases.

We note that there can be slight numerical differences between the Triton and stock PyTorch implementations, which is due to the way Triton handles computations, and which is fully expected. These numerical differences are negligible and do not negatively impact models using `blksprs` kernels, which we tested by training and comparing two otherwise identical models, one using PyTorch's operations, one using ours.

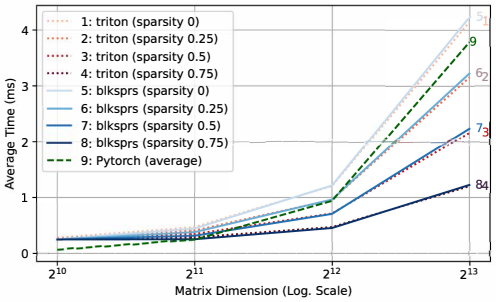
4.1. Performance

As the main advantages of block-sparse approaches come in the form of time- and space-complexity improvements, we are mainly interested in the performance of `blksprs` when compared to its sparsity-agnostic counterparts. We selected four operations (namely *matmul*, *softmax*, *scatter_reduce*, and *repeat_interleave*) to evaluate the performance of `blksprs` and its kernels. These operations were chosen as they represent the core computational primitives for Transformer-based architectures: *matmul* and *softmax* are fundamental to attention computation, *scatter_reduce* is essential for sparse information aggregation, and *repeat_interleave* is required for multi-head attention. For this comparison, we conducted tests using both 16-bit and 32-bit float input. Here, we present only the results of the 32-bit evaluation for the sake of brevity and due to the fact that we saw very similar relative speedups on the 16-bit evaluations. We compare our implementations with the stock, block-sparse agnostic PyTorch implementations, and (wherever possible) with the implementations previously included with Triton. We regard Triton's existing block-sparse kernels as the most direct quantitative baseline, as they provide the closest operation-level comparison to a Triton-based library such as `blksprs`.

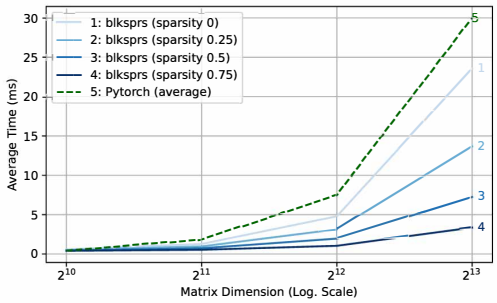
[1] https://developer.nvidia.com/cuda.



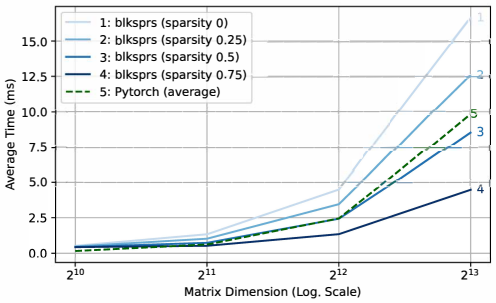
(a) Benchmark results for the *matmul* operation using the PyTorch, Triton, and `blksprs` implementations.



(b) Benchmark results for the *softmax* operation using the PyTorch, Triton, and `blksprs` implementations.



(c) Benchmark results for the *scatter_reduce* operation using the PyTorch and `blksprs` implementations.



(d) Benchmark results for the *repeat_interleave* operation using the PyTorch and `blksprs` implementations.

Figure 1. Results of the benchmark comparing different implementations of four block-sparse and their regular counterpart operations.

We selected four different input matrix sizes, namely $2^{10} \times 2^{10}$, $2^{11} \times 2^{11}$, $2^{12} \times 2^{12}$, and $2^{13} \times 2^{13}$, which we used as input sizes for all matrices assigned to an operation, e.g., source, index, and target matrix for the *scatter_reduce* operation. These sizes correspond to sequence lengths of 1,024 to 8,192 tokens, representing typical ranges encountered in practical Transformer applications from standard natural language processing tasks to long-context scenarios. We used a batch size of 4, stacking four input matrices to obtain an input tensor. Furthermore, we conducted our comparison on four sets of matrices with a random percentage of blocks marked as sparse to evaluate the performance improvements induced by our implementation. For each approach, we tested with input matrices where 0%, 25%, 50%, and 75% of blocks were marked as sparse, respectively. Tests were conducted using the *do_bench*() function included in Triton. Here, we averaged the results over 50 randomly generated input matrices, where for each of them we ran the *do_bench*() function 10 times, removing the first 20% of entries in order to allow the GPU to warm up. All tests were conducted on a single NVIDIA A40 GPU.[2] Figure 1 shows the results of this evaluation, which we will now discuss on a per-operation basis. Note that for all tests we only present a single (averaged) result of the PyTorch implementation, as it is agnostic to matrices with sparse blocks.

[2] https://www.nvidia.com/en-us/data-center/a40.

4.1.1. **Matmul Benchmark**

Figure 1a shows the results of the comparison between the different *matmul* implementations. In contrast to the Triton implementation (which only supports a single of the input matrices or the output matrix to be block-sparse), `blksprs` fully supports sparsity layout specification for both the input and the output. This improves flexibility and enables more sophisticated approaches. In order not to give `blksprs` an unfair advantage in our tests, we limited the sparsity of the input matrices to 25% for all but the first test, where we used completely non-sparse matrices to compare to the stock PyTorch implementation.

As Figure 1a shows, our implementation outperforms both PyTorch and Triton. Furthermore, due to the fact that `blksprs` supports sparsity definitions for the input matrices as described above, it greatly outperforms the Triton implementation for the 25%, 50%, and 75% sparsity cases (where a sparsity of 25% was used for each input matrix). Even when not using block-sparse input matrices, we saw slight improvements when using `blksprs`, while improving other aspects as discussed in Section 3.4.

4.1.2. **Softmax Benchmark**

Concerning the *softmax* benchmark, both our and the Triton implementation perform very similar and within a margin of error to each other. We note that although the performances are similar, our approach has further merits as described in Section 3.4. Figure 1b clearly shows that due to the overhead induced by the block-sparse process, the block-sparse *softmax* function outperforms its regular counterpart starting at about 25% sparsity. We argue that due to the low absolute runtimes (with a difference of less than one millisecond between the no-sparsity cases), we can still recommend replacing the *softmax* function with its block-sparse counterpart, as even in the worst case the penalty is negligible.

4.1.3. **Scatter Reduce Benchmark**

In contrast to the *matmul* and *softmax* functions, Triton does not include block-sparse counterparts of the *scatter_reduce* and *repeat_interleave* functions. Figure 1c shows the comparison between PyTorch's and `blksprs`' *scatter_reduce* operation. This function writes values of an input matrix to specific indices of an output matrix based on an input index matrix, summing up values assigned to the same target index. Here, we use the same dimension for all three of these matrices (which are packed into tensors). Our `blksprs` library clearly outperforms the conventional implementation even for the no-sparsity case and scales very well with increased sparsity percentages. Note that similar to the *matmul* case, we support sparsity layouts for all matrices partaking in the operation. For this benchmark we applied the sparsity percentages only to the input matrices, targeting an output without sparse blocks.

4.1.4. **Repeat Interleave Benchmark**

Figure 1d shows the results of the benchmark for the *repeat_interleave* function. Here, the `blksprs` implementation is outperformed by PyTorch's version up to a sparsity percentage of about 40%. We conjecture that this is due to the increased overhead induced by the large number of loads necessary to evaluate the position of a block in the sparsity layout. We argue that, similarly to the *softmax* case, even in the worst case the performance penalty induced by the block-sparse version is small enough that it can be used in place of the regular version. Furthermore, it eliminates the need to convert from the compressed version to a regular version and back just to apply this operation.

Approach / Seq. Length	1,024	2,048	4,096	8,192
Regular Transformer	0.30 GB	0.93 GB	3.32 GB	12.62 GB
`blksprs` Transformer	0.22 GB	0.59 GB	1.90 GB	6.82 GB

Table 1. Memory requirements for two Transformer models trained on random input data, one using conventional attention, one using attention using `blksprs`.

Approach / Seq. Length	1,024	2,048	4,096	8,192
Regular Transformer	0.31 s	3.35 s	14.60 s	60.02 s
`blksprs` Transformer	1.14 s	3.22 s	9.64 s	38.61 s

Table 2. Training benchmarks for two Transformer models trained on random input data, one using conventional attention, one using attention using `blksprs`.

4.2. **Practical Evaluation**

To assess real-world impact, we compared a standard Transformer [1] against one using `blksprs` block-sparse attention, with input lengths from 1,024 to 8,192 tokens. All tests were run on a locally available NVIDIA RTX 3090.[3]

The results of the first tests are shown in Table 1. Here, we measured the peak memory allocation for training a model on 240 randomly generated input tensors with a batch size of 1 and varying length for both models. The approach making use of our `blksprs` library significantly outperforms the regular approach for all configurations. Here, we were able to record a reduction in memory by a factor of about 0.7 to 0.5 in overall memory consumption.

Table 2 shows the runtime evaluations for data of the same kind as described above. Note that here, we varied the batch size based on the memory requirements of the model, using as many batches as would fit into the GPU's memory. Generally, we saw that `blksprs` scales well with increased batch sizes, where it is able to parallelise the overhead (in the form of, e.g., LUT creation). Although for very small sequences ($1,024$ tokens in length) our approach is outperformed by the regular transformer, starting with a sequence length of $2,048$ we once again see clear improvements when using `blksprs`. Here, we see a runtime reduction of about one third, which scales well with growing sequence sizes.

Note that we only applied `blksprs` to the attention module; further improvements are possible when adapting other model components. These tests demonstrate that `blksprs` can drastically improve performance with minimal implementation effort.

5. **Conclusion**

In this paper, we introduced `blksprs`, a comprehensive Triton-based library for operations on block-sparse matrices and tensors. Our `blksprs` library is designed primarily for machine-learning approaches, ensuring a high degree of flexibility of its operations, a low barrier of implementation, and supports sparsity specification across all inputs and outputs.

Using extensive benchmarks, we were able to show that `blksprs` can consistently outperform existing Triton and (non-sparse) PyTorch implementations, with performance gains scaling effectively with sparsity levels. Using `blksprs`, in a practical evaluation on Transformer architectures replacing standard attention with our block-sparse approach, we were able to reduce training time by up to 35% and memory consumption by up to 45%.

We acknowledge certain limitations of our work. For our experimental evaluation we exclusively utilised NVIDIA GPUs. While Triton's design suggests potential portability to

[3] https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090-3090ti.

other hardware vendors, we were unable to empirically validate this due to lack of access to such accelerators. Furthermore, the benefits of block-sparsity are most pronounced for larger sequence lengths and higher sparsity levels, with overhead potentially outweighing gains for very short sequences or low sparsity scenarios.

We will release `blksprs` and its source code to enable more efficient and novel deep learning approaches. In a companion paper [25], we describe an efficient novel method for additive relative information injection making use of the `blksprs` library for symbolic music composition.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. "Attention is All you Need". In: *Proceedings of the 30th Annual Conference on Neural Information Processing Systems* (*NIPS 2017*). 2017.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (*NAACL-HLT 2019*). Association for Computational Linguistics, 2019, pp. 4171–4186.

[3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. *Language Models are Unsupervised Multitask Learners*. OpenAI Blog. OpenAI Technical Report. 2019. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[4] T. B. Brown et al. "Language Models are Few-Shot Learners". In: *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems* (*NeurIPS 2020*). 2020.

[5] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *Proceedings of the 9th International Conference on Learning Representations* (*ICLR 2021*). OpenReview.net, 2021.

[6] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo. "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows". In: *Proceedings of the 2021 IEEE/CVF International Conference on Computer Vision* (*ICCV 2021*). IEEE, 2021, pp. 9992–10002.

[7] A. Baevski, Y. Zhou, A. Mohamed, and M. Auli. "wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations". In: *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems* (*NeurIPS 2020*). 2020.

[8] A. Gulati, J. Qin, C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu, and R. Pang. "Conformer: Convolution-augmented Transformer for Speech Recognition". In: *Proceedings of the 21st Annual Conference of the International Speech Communication Association* (*Interspeech 2020*). ISCA, 2020, pp. 5036–5040.

[9] J. Lu, D. Batra, D. Parikh, and S. Lee. "ViLBERT: Pretraining Task-Agnostic Visiolinguistic Representations for Vision-and-Language Tasks". In: *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems* (*NeurIPS 2019*). 2019, pp. 13–23.

[10] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch. "Decision Transformer: Reinforcement Learning via Sequence Modeling". In: *Proceedings of the 34th Annual Conference on Neural Information Processing Systems* (*NeurIPS 2021*). 2021, pp. 15084–15097.

[11] J. Jumper et al. "Highly Accurate Protein Structure Prediction with AlphaFold". In: *Nature* 596.7873 (2021), 583–589. ISSN: 1476-4687.

[12] D. Bahdanau, K. Cho, and Y. Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *Proceedings of the 3rd International Conference on Learning Representations* (*ICLR 2015*). 2015.

[13] T. Luong, H. Pham, and C. D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (*EMNLP 2015*). The Association for Computational Linguistics, 2015, pp. 1412–1421.

[14] N. Kitaev, L. Kaiser, and A. Levskaya. "Reformer: The Efficient Transformer". In: *Proceedings of the 8th International Conference on Learning Representations* (*ICLR 2020*). OpenReview.net, 2020.

[15] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma. "Linformer: Self-Attention with Linear Complexity". In: *CoRR* abs/2006.04768 (2020). arXiv: 2006.04768.

[16] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *Proceedings of the 35th Annual Conference on Neural Information Processing Systems* (*NeurIPS 2022*). 2022.

[17] T. Dao. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning". In: *Proceedings of the 12th International Conference on Learning Representations* (*ICLR 2024*). OpenReview.net, 2024.

[18] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao. "FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision". In: *Proceedings of the 38th Annual Conference on Neural Information Processing Systems 2024* (*NeurIPS 2024*). 2024.

[19] R. Child, S. Gray, A. Radford, and I. Sutskever. "Generating Long Sequences with Sparse Transformers". In: *CoRR* abs/1904.10509 (2019). arXiv: 1904.10509.

[20] I. Beltagy, M. E. Peters, and A. Cohan. "Longformer: The Long-Document Transformer". In: *CoRR* abs/2004.05150 (2020). arXiv: 2004.05150.

[21] S. Gray, A. Radford, and D. Kingma. *Block-sparse GPU Kernels*. OpenAI Blog. 2017. URL: https://openai.com/index/block-sparse-gpu-kernels/.

[22] P. Tillet, H. Kung, and D. D. Cox. "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (*MAPL@PLDI 2019*). ACM, 2019.

[23] DeepSpeed Team. *DeepSpeed Sparse Attention*. DeepSpeed Blog. Accessed 2026-04-13. 2020. URL: https://www.deepspeed.ai/2020/09/08/sparse-attention.html.

[24] *CUDA Toolkit 3.2*. Introduces cuSPARSE. NVIDIA, 2010. URL: https://developer.nvidia.com/cuda-toolkit-32-downloads.

[25] F. Schön and H. Tompits. "Efficient Additive Relative Information Attention for Transformer-based Symbolic Music Composition". In: *Proceedings of the 39th Canadian Conference on Artificial Intelligence* (*Canadian AI 2026*). Vol. 318. Proceedings of Machine Learning Research. 2026.