HAL open science

Fast inference with Kronecker-sparse matrices

Antoine Gonon, Léon Zheng, Pascal Carrivain, Quoc-Tung Le

To cite this version:

Antoine Gonon, Léon Zheng, Pascal Carrivain, Quoc-Tung Le. Fast inference with Kronecker-sparse matrices. 2024. hal-04584450v3

HAL Id: hal-04584450

https://hal.science/hal-04584450v3

Preprint submitted on 8 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fast inference with Kronecker-sparse matrices

Antoine Gonon*[†], Léon Zheng*[†][‡], Pascal Carrivain*[§], Quoc-Tung Le[†]

October 8, 2024

Abstract

This paper *benchmarks* and *improves* existing GPU matrix multiplication algorithms specialized for Kronecker-sparse matrices, whose sparsity patterns are described by Kronecker products. These matrices have recently gained popularity as replacements for dense matrices in neural networks because they preserve accuracy while using fewer parameters. We present the first energy and time benchmarks for the multiplication with such matrices, helping users identify scenarios where Kronecker-sparse matrices are more time- and energy-efficient than their dense counterparts. Our benchmark also reveals that specialized implementations spend up to 50% of their total runtime on memory rewriting operations. To address the challenge of reducing memory transfers, we introduce a new so-called tiling strategy adapted to the Kronecker-sparsity structure, which reduces reads and writes between levels of GPU memory. We implement this tiling strategy in a new CUDA kernel that achieves a median speed-up of $\times 1.4$, while also cutting energy consumption by 15%. We further demonstrate the broader impact of our results by applying the new kernel to accelerate transformer inference.

1 Introduction

Accelerating the inference and training of deep neural networks is a major challenge given their constantly growing resource requirements. At the very heart of neural network efficiency is the acceleration of matrix multiplication on GPU, which is one of the main operation during both training and inference. For instance, in a forward pass of vision transformers (ViT) [6], between 30% and 60% of the total time is spent in fully-connected layers doing matrix multiplications (see Appendix B.6 for details). One key approach to speed up computation is to enforce *sparsity* constraints on certain weight matrices in the model and to rely on sparse software libraries to perform the matrix-vector multiplications [9].

Among various forms of sparsity, *butterfly sparsity* has emerged as a promising approach for constructing efficient neural networks [4]. Butterfly matrices are structured matrices that can be expressed as products of sparse factors with specific sparsity patterns [8, 13, 15, 25, 26], offering sub-quadratic *theoretical* matrix multiplication complexity. The fast algorithms associated to important linear transforms, such as the Discrete Fourier Transform (DFT) and the Hadamard Transform, heavily exploit this structure [2]. See Figure 1 for an example of the decomposition of the DFT matrix into a product of Kronecker-sparse matrices, in dimension 16.

In this work, we focus on the fundamental building blocks of butterfly matrices: their sparse factors with sparsity patterns defined by Kronecker products. Specifically, these factors have supports of the form $\mathbf{S}_{\boldsymbol{\pi}} = \mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d$, where $\otimes$ is the Kronecker product, $\mathbf{I}_n$ denotes the $n \times n$ identity matrix, $\mathbf{1}_{n\times m}$ is an $n \times m$ matrix of ones, and $\boldsymbol{\pi} = (a, b, c, d)$ is a tuple of integers defining the sparsity pattern (see Definition 2.1 and Figure 2). We introduce the term *Kronecker-sparse matrices* to refer to these sparse factors with Kronecker product-based supports.

We introduce the term *Kronecker-sparse matrices* to precisely capture the computational structures that contribute to the efficiency of butterfly matrices. Existing definitions of butterfly matrices are often either too restrictive—applying only to square matrices with dyadic dimensions [2, 20]—or too general, encompassing dense matrices [15]. However, all definitions agree that butterfly matrices are products of matrices with Kronecker constraints on their supports. Therefore, we focus on these *Kronecker-sparse matrices* in this paper.

*Equal contribution.
[†]ENS de Lyon, CNRS, Université Claude Bernard Lyon 1, Inria, LIP, UMR 5668, 69342, Lyon cedex 07, France.
[‡]valeo.ai
[§]Inria, ENS de Lyon, CNRS, Université Claude Bernard Lyon 1, LIP, UMR 5668, 69342, Lyon cedex 07, France.

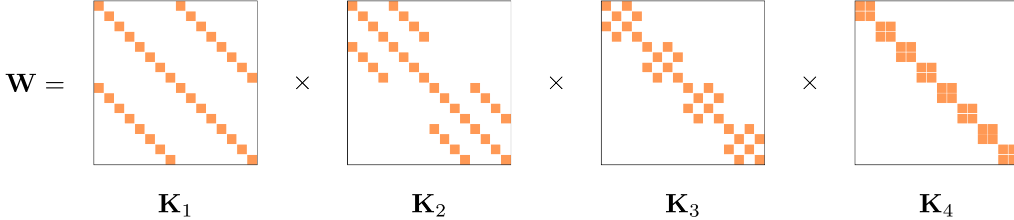


Figure 1: Example of butterfly factorization $\mathbf{W} = \mathbf{K}_1 \dots \mathbf{K}_L$, for $L = 4$. Here, the factor $\mathbf{K}_\ell \in \mathbb{R}^{N \times N}$ (with $N = 2^L$) has support $\mathbf{S}_\ell = \mathbf{I}_{2^{\ell-1}} \otimes \mathbf{1}_{2\times 2} \otimes \mathbf{I}_{2^{L-\ell}}$. This corresponds to the butterfly factorization of the Discrete Fourier Transform matrix $\mathbf{W}$, up to a permutation of its column indices.

In practice, the goal is to reparameterize a dense fully-connected layer $\mathbf{W}$ as a product of *Kronecker-sparse* matrices $\mathbf{W} = \mathbf{K}_1 \dots \mathbf{K}_L$ while having (i) at least the same accuracy for the learning task at hand, (ii) less parameters to store, and (iii) an accelerated inference and training phase. Previous works mostly focused on (i) and (ii) [3, 4, 15, 20]. This work tackles (iii).

Main contributions. (i) We assess for the *first time* the time- and energy-efficiency of `PyTorch` GPU algorithms for multiplying a batch of vectors with a *Kronecker-sparse* matrix, including algorithms specialized for Kronecker-sparsity relying on efficient libraries for batch GEMM[1], block-sparse matrix multiplication and tensor contraction. The benchmark is easy to adapt to include future implementations, and can be used by users to identify situations where Kronecker-sparse matrices can be beneficial.

(ii) The benchmark reveals that specialized implementations spend up to 50% of their total runtime on GPU memory rewriting operations. To address this, we design a new tiling strategy, with tiles adapted to Kronecker-sparsity, implemented in a new open-source[2] CUDA kernel. This reduces the transfers between the different levels of GPU memory, achieving a median speed-up factor of $\times 1.4$ in *float-precision* while also cutting energy consumption by a median of 15%. We also demonstrate the broader impact of our results by showing how the new kernel can be used to speed up the inference of transformers.

(iii) We introduce a heuristic based on theoretical and empirical findings that helps to decide whether a Kronecker-sparsity pattern (a, b, c, d) will be time- and energy-efficient compared to its dense counterpart. This rule, based on the ratio $(b + c)/bc$, paves the way for designing more efficient Kronecker-sparse neural networks in the future, for instance by selecting the most efficient pattern (a, b, c, d) among those with the same number of non-zeros.

Outline. Section 2 introduces the framework to study Kronecker-sparse matrix multiplication, and describes existing GPU algorithms on PyTorch. Section 3 assesses the cost of GPU memory access in these baselines. Section 4 explains how the new CUDA kernel reduces the memory transfer compared to previous existing implementations. Section 5 benchmarks the execution time and energy consumption of existing PyTorch GPU algorithms, and the new kernel, for the multiplication with a Kronecker-sparse matrix. Section 6 concretely illustrates broader implications of this work: the new kernel can be used to speed up the inference of neural networks.

2 Background on Kronecker-sparse matrices

We call a *Kronecker-sparse* matrix any matrix whose support is given by a particular Kronecker product, in line with the building blocks of widespread *butterfly matrices* [2, 3, 4, 7, 12, 15, 15, 20]. Let us emphasize that this Kronecker structure is imposed *only* on the *support*, not on the values of the weights.

Definition 2.1 (Kronecker-sparse matrix)**.** A *Kronecker-sparsity pattern* (or simply Kronecker pattern) is a tuple $\boldsymbol{\pi} := (a, b, c, d) \in (\mathbb{N}_{>0})^4$. A *$\boldsymbol{\pi}$-Kronecker-sparse matrix* (or simply Kronecker-sparse matrix when $\boldsymbol{\pi}$ is clear from the context) is a matrix $\mathbf{K} \in \mathbb{R}^{abd \times acd}$ satisfying $\texttt{supp}(\mathbf{K}) \subseteq \texttt{supp}(\mathbf{S}_{\boldsymbol{\pi}})$, where $\mathbf{S}_{\boldsymbol{\pi}} := \mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d$ (see Figure 2) and where $\texttt{supp}(\mathbf{M}) := \{(i, j), \mathbf{M}_{i,j} \neq 0\}$. The set of $\boldsymbol{\pi}$-Kronecker-sparse matrices is denoted $\Sigma^{\boldsymbol{\pi}}$.

A $\boldsymbol{\pi}$-Kronecker-sparse factor is *sparse* and *structured*. For $\boldsymbol{\pi} = (a, b, c, d)$, it has at most $abcd$ nonzero entries, which yields a sparsity ratio $\frac{abcd}{a^2bcd^2} = \frac{1}{ad}$ since it is of size $abd \times acd$. Kronecker-sparse matrices can represent a wide variety of matrices that have been used to train neural networks, as shown in Table 1.

[1] GEMM stands for General Matrix Multiplication.
[2] The code is available at `https://github.com/PascalCarrivain/ksmm`.

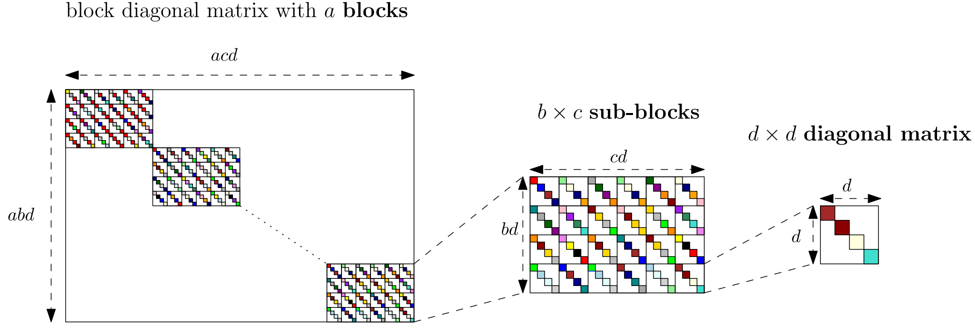


Figure 2: A $\boldsymbol{\pi}$-Kronecker-sparse matrix with $\boldsymbol{\pi} = (a, b, c, d)$ is a block-diagonal matrix with a blocks, where each block itself is a block matrix composed by $b \times c$ diagonal matrices of size $d \times d$. The colored cells correspond to the nonzeros. We color the cells with different colors to indicate that the corresponding weights are free to take different values.

Table 1: Examples of matrices used in neural networks, which can be expressed in terms of products of Kronecker-sparse matrices. For a matrix of the form $\mathbf{W} = \mathbf{K}_1 \dots \mathbf{K}_L$, the column "Kronecker patterns" describes the list of Kronecker-sparsity patterns $\boldsymbol{\pi}_\ell = (a, b, c, d)$ for each Kronecker-sparse matrix $\mathbf{K}_\ell$.

	Matrix size	Kronecker patterns
Dense	$M \times N$	$(1, M, N, 1)$
Low-rank	$M \times N$	$(1, M, r, 1), (1, r, N, 1)$
Square dyadic [2, 20]	$N \times N$ with $N = 2^L$	$(2^{\ell-1}, 2, 2, 2^{L-\ell})_{\ell=1}^L$
Kaleidoscope [4]	$N \times N$ with $N = 2^L$	$(2^{\ell-1}, 2, 2, 2^{L-\ell})_{\ell=1}^L \cup (2^{L-\ell}, 2, 2, 2^{\ell-1})_{\ell=1}^L$
Block butterfly [3]	$N \times N$ with $N = 2^L t$	$(2^{\ell-1}, 2t, 2t, 2^{L-\ell})_{\ell=1}^L$
Monarch [4, 7]	$M \times N$	$(1, M/p, \min(M,N)/p, p), (p, \min(M,N)/p, N/p, 1)$
Deformable butterfly [15]	$M \times N$ with $M = a_1 b_1 d_1$, $N = a_L c_L d_L$	$(a_\ell, b_\ell, c_\ell, d_\ell)_{\ell=1}^L$ s.t. $a_\ell c_\ell d_\ell = a_{\ell+1} b_{\ell+1} d_{\ell+1}$.

2.1 Existing `PyTorch` GPU implementations

Notations. $\mathbf{X} \in \mathbb{R}^{B \times N}$ is the input matrix (batch size B, input dimension N). $\Sigma^{\boldsymbol{\pi}}$ is the set of matrices with Kronecker-sparsity pattern $\boldsymbol{\pi} = (a, b, c, d)$ (Definition 2.1). $\mathbf{0}_{m \times n}$ is the $m \times n$ matrix filled with zeros. For integers $a \leq b$, $[\![a, b]\!] := \{a, a+1, \dots, b\}$. For a matrix $\mathbf{M}$, $\mathbf{M}[I, :]$ is the submatrix restricted to rows I, and $\mathbf{M}[I, J]$ is the restriction to rows I and columns J. Matrix transposition is represented by $\top$. Matrix indices start at zero.

All existing GPU implementations specialized for Kronecker-sparsity build on Algorithm 1, an algorithm tailored to Kronecker-sparsity that decomposes the multiplication with a Kronecker-sparse matrix $\mathbf{K}$ as a permutation of the input (line 3), a multiplication with a permuted representation $\tilde{\mathbf{K}}$ of $\mathbf{K}$ (line 2), and a final permutation of the result (line 1). The permutations are performed to reduce to a multiplication with $\tilde{\mathbf{K}}$, which is more computationally efficient on GPU as it is block-diagonal with dense sub-blocks. Algorithm 1 generalizes to general Kronecker-sparsity patterns $\boldsymbol{\pi} = (a, b, c, d)$ the algorithm suggested by Dao et al. [4] in the specific cases $a = 1$ or $d = 1$. We now describe the concrete `PyTorch` GPU implementations. More details are given in appendix (Appendix D.1) and the full code is available online at `https://github.com/PascalCarrivain/ksmm`.

Algorithm 1 Kronecker-sparse matrix multiplication

Input: $\boldsymbol{\pi}, \mathbf{X}, \tilde{\mathbf{K}} := \mathbf{P}^\top \mathbf{K} \mathbf{Q}^\top$ with $\mathbf{K} \in \Sigma^{\boldsymbol{\pi}}$, $\mathbf{P} := (\mathbf{I}_a \otimes \mathbf{P}_{b,d})$, $\mathbf{Q} := (\mathbf{I}_a \otimes \mathbf{P}_{c,d})^\top$ cf. (1)
Output: $\mathbf{Y} = \mathbf{X}\mathbf{K}^\top \in \mathbb{R}^{B \times M}$
1: $\tilde{\mathbf{X}} \leftarrow \mathbf{X}\mathbf{Q}^\top$
2: $\tilde{\mathbf{Y}} \leftarrow \tilde{\mathbf{X}}\tilde{\mathbf{K}}^\top$
3: $\mathbf{Y} \leftarrow \tilde{\mathbf{Y}}\mathbf{P}^\top$

Algorithm 2 Equivalent formulation for new tiling strategy

Input: $\boldsymbol{\pi} = (a, b, c, d)$, $\mathbf{K} \in \Sigma^{\boldsymbol{\pi}}$, $\mathbf{X} \in \mathbb{R}^{B \times N}$ $(N := acd)$
Output: $\mathbf{Y} = \mathbf{X}\mathbf{K}^\top \in \mathbb{R}^{B \times M}$ $(M := abd)$
1: $\mathbf{Y} \leftarrow \mathbf{0}_{B \times M}$
2: **for** $(i, j) \in [\![0, a-1]\!] \times [\![0, d-1]\!]$ **do**
3: $\quad \texttt{col} \leftarrow \left\{ i\frac{N}{a} + j + \ell d \,|\, \ell \in [\![0, c-1]\!] \right\}$
4: $\quad \texttt{row} \leftarrow \left\{ i\frac{M}{a} + j + kd \,|\, k \in [\![0, b-1]\!] \right\}$
5: $\quad \mathbf{Y}[:, \texttt{row}] \leftarrow \mathbf{X}[:, \texttt{col}]\mathbf{K}^\top[\texttt{col}, \texttt{row}]$
6: **end for**

bmm and bsr implementations. The first implementations we consider are the one Dao et al. [4], that we call bmm, and a new one that we call bsr. **Note that the original bmm implementation from Dao et al. [4] only works for a pattern $\pi = (a, b, c, d)$ satisfying $a = 1$ or $d = 1$.** We extend it to the general case. Both bmm and bsr implement Algorithm 1 as specified by Table 2. For the multiplication with $\tilde{\mathbf{K}}$ (line 2 in Algorithm 1), bmm relies on batched GEMM NVIDIA routines called through torch.bmm, while bsr relies on the PyTorch block-sparse library.

	bmm	bsr
Storage format for $\tilde{\mathbf{K}}$	3D-tensor of shape (ad, b, c)	2D-tensor of shape (abd, acd) stored in BSR[3] format
Line 1 of Algorithm 1	torch.reshape	
Line 2 of Algorithm 1	torch.bmm	torch.nn.functional.linear
Line 3 of Algorithm 1	torch.reshape	

Table 2: Differences in the implementation of Algorithm 1 between bmm and bsr.

einsum implementation. We propose a new PyTorch implementation specialized for Kronecker-sparsity using tensor contractions [19], inspired by the other specialized implementation[4] given in Dao et al. [4]. It stores the nonzero entries of $\mathbf{K} \in \Sigma^{\pi}$ with a 4D-tensor B_einsum of shape (a, b, c, d), in such a way that the slice B_einsum$[i, :, :, j]$ for $(i, j) \in [\![0, a-1]\!] \times [\![0, d-1]\!]$ stores the entries of $\mathbf{K}$[row, col] where row, col are defined in lines 3 and 4 of Algorithm 2 (Algorithm 2 will be discussed in details in Section 4). The batched matrix multiplication operations at line 5 are then implemented using Einstein summation between this 4D-tensor and a reshaped input tensor.

The above implementations (bmm, bsr, einsum) are specialized for Kronecker-sparsity. We also compare them to the following generic implementations (dense and sparse) that ignore the Kronecker-sparsity.

dense implementation. This ignores the sparsity of $\mathbf{K}$, by storing *all* its entries, including zeros, in a tensor of shape (M, N). The multiplication is done with torch.nn.functional.linear, the default PyTorch implementation for linear layers.

sparse implementation. This exploits the sparsity of $\mathbf{K}$ but not its structure (recall that the sparsity pattern is not arbitrary, but structured as Kronecker products, see Definition 2.1). The nonzero entries of the factor $\mathbf{K}$ are saved in a tensor stored in the Compressed Sparse Row (CSR) format, and the matrix multiplication is done with torch.nn.functional.linear.

2.2 Memory layout convention

Batch-size-first *vs.* batch-size-last. The entries of the input $\mathbf{X} \in \mathbb{R}^{B \times N}$ can be stored either in a PyTorch tensor X_bsf of shape (B, N), or in a PyTorch tensor X_bsl of shape (N, B), in such a way that the entries of the row $\mathbf{X}[k, :]$ are stored in the slices X_bsf$[k, :]$ and X_bsl$[:, k]$. Because of PyTorch's row-major convention, the tensor X_bsf stores in contiguous memory the entries of each row $\mathbf{X}[k, :]$, as opposed to X_bsl that store contiguously the entries of each column $\mathbf{X}[:, i]$. These two different memory layouts are called *batch-size-first* and *batch-size-last*[5] in this paper. Note that the tensor saving the output $\mathbf{Y} = \mathbf{X}\mathbf{K}^{\top}$ will always be in the same memory layout as the input tensor. All the implementations above can be implemented in both ways. While the main point of the paper is to compare the implementations, we will also study the effect of this memory layout convention.

3 Memory accesses in baseline implementations

The specialized implementations bmm and bsr explicitly perform permutation operations corresponding to lines 1 and 3 in Algorithm 1 (see Table 2) to be able to use high-performance multiplication routines for the multiplication with the block-diagonal matrix $\tilde{\mathbf{K}}$ (line 2 in Algorithm 1). This paper assesses for the first time the cost of these memory operations in practice, as we now discuss.

Importance of data transfers. GPU memory management plays a critical role in optimizing performance. Memory in a GPU is organized hierarchically, with global memory being the largest and slowest, followed by shared memory, and finally registers, which are the smallest and fastest [18, Sec. 2.3]. By default, data resides in the global memory of the GPU. Each thread of the GPU runs a kernel that

[4]See their repository github.com/HazyResearch/fly.

[5]By analogy with the recent PyTorch optimization channels last that moves the channels dimension to the last position for convolutional layers.

reads data from global memory into registers, performs register-level computations, and writes the results back to global memory. Therefore, when operations are bottlenecked by memory accesses, it is critical to minimize data transfers between global memory, shared memory, and registers to obtain an efficient GPU implementation [18, Sec. 5.3].

Data transfers in baseline implementations. In this paper, we argue that the baseline `bmm`, `bsr` and `einsum` implementations for Kronecker-sparse matrix multiplication require performing *several passes between global memory and registers that can account for a large proportion of the total runtime* in practice. This suggests that there is room for improvement in the memory accesses of these implementations.

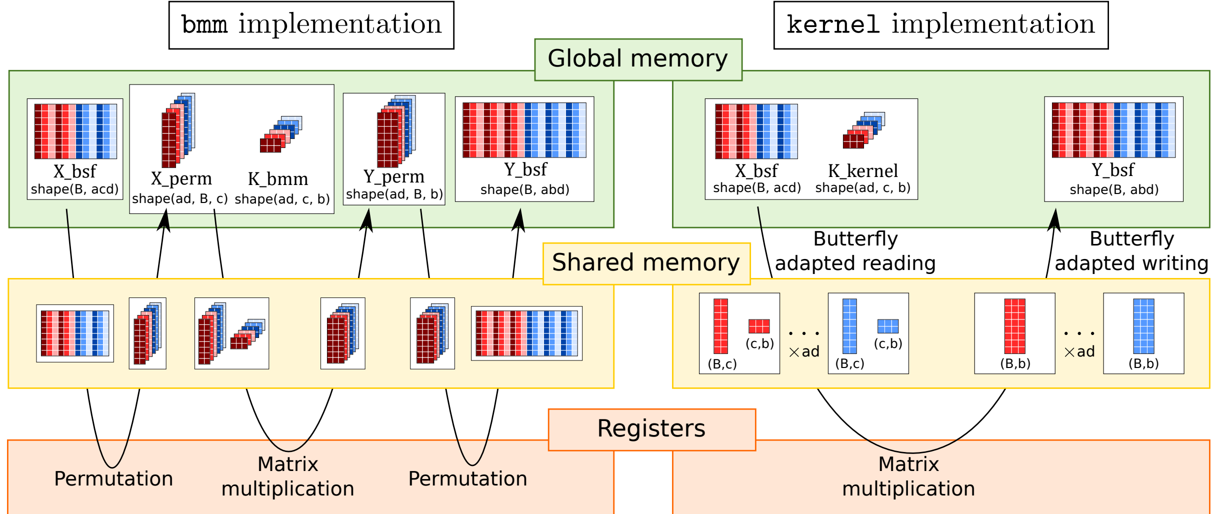


Figure 3: Data flow between the different levels of GPU memory for the `bmm` implementation (Section 2.1) from [4] and the new `kernel` (Section 4).

Let us focus on `bmm`, as we will find it to be faster than `einsum` and `bsr`. The data flow of `bmm` is illustrated in Figure 3. There is one pass between the global memory and the registers to perform the permutation with $\mathbf{P}$ (line 3 in Algorithm 1), one for the multiplication with $\tilde{\mathbf{K}}$ (line 2), and another one for the permutation with $\mathbf{Q}$ (line 1).

Estimated time for memory rewritings in `bmm`. We benchmark the relative time spent on memory rewritings in `bmm`, which is, as we will find out later (Section 5), the fastest of the baseline implementations. We find that **the memory rewritings can take up to** 45% **of the total runtime**[6]. This can be seen by looking at the y-axis in Figure 4 (see Appendix B.2 for details on the experiments). We will explain in Section 4 why we plot as a function of the ratio $(b+c)/bc$. We conclude that *it is crucial to optimize the data transfers between the different levels of GPU memory* to improve current implementations.

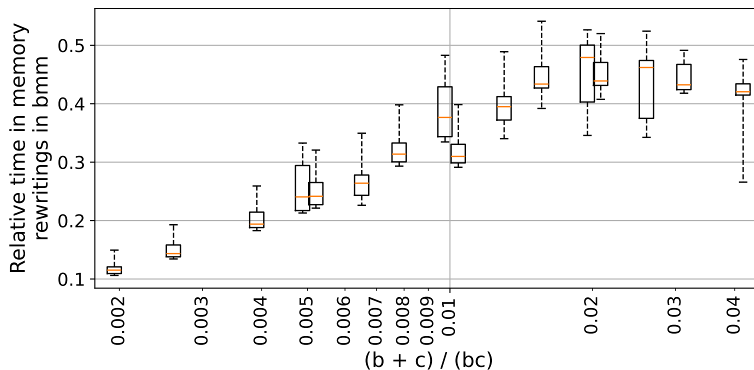


Figure 4: Estimated relative time spent on memory rewritings in `bmm` for the multiplication with $\mathbf{K} \in \Sigma^{\boldsymbol{\pi}}$, for several $\boldsymbol{\pi} = (a, b, c, d)$. We regroup patterns by their value of $(b+c)/(bc)$, and plot a boxplot to summarize the corresponding measurements.

[6]Regardless of the memory layout convention, *batch-size-first* or *batch-size-last*.

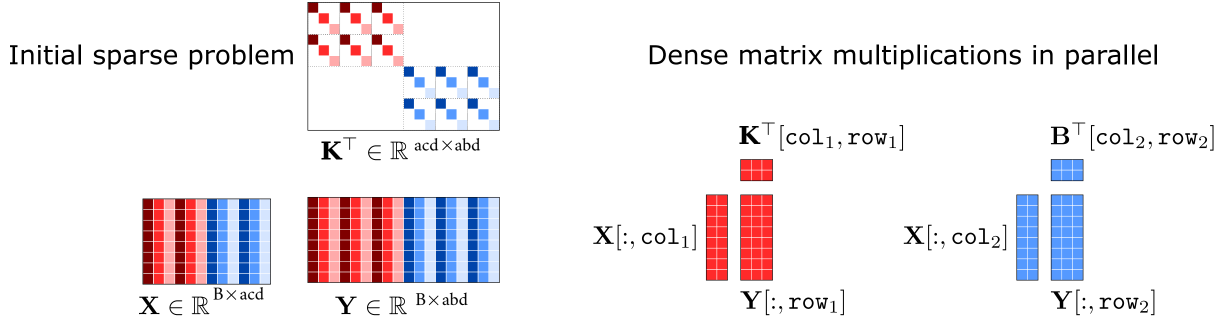


Figure 5: Illustration of Algorithm 2 for sparsity pattern $\boldsymbol{\pi} = (2, 3, 2, 3)$ and batch size $B = 8$. The subsets of rows and columns $(\texttt{row}_1, \texttt{col}_1)$ are associated with the values $(i, j) = (0, 1)$ in the "for" loop of Algorithm 2, whereas $(\texttt{row}_2, \texttt{col}_2)$ are associated with $(i, j) = (1, 1)$.

4 A novel tiling strategy for Kronecker-sparse matrix multiplication with reduced memory transfers

All existing specialized implementations are based on the Algorithm 1 that we introduced in Section 2.1. In Section 4.1, we start by introducing a novel mathematically equivalent reformulation of the multiplication algorithm, Algorithm 2, which corresponds to a new tiling strategy. This strategy allows us to implement the multiplication in a single CUDA kernel, as described in Section 4.2. We then theoretically analyze the memory operations of this new implementation and compare it to existing implementations in Section 4.3. In particular, we exhibit a heuristic to identify efficient Kronecker-sparsity patterns, that will be empirically confirmed later (Section 5).

4.1 A new tiling strategy for Kronecker-sparse matrix multiplication

We propose a new tiling strategy to reduce the cost associated with memory operations. Tiling consists of splitting the matrices into smaller submatrices, or *tiles*, and constructing the result by accumulating the intermediary results obtained on each of these tiles [16, 18]. Our tiling strategy comes from our *mathematically* equivalent reformulation of Algorithm 1 into Algorithm 2, as we now explain.

On Algorithm 2, and why it is equivalent to Algorithm 1. When $d = 1$, the Kronecker-sparse matrix $\mathbf{K}$ is block-diagonal with a dense blocks, as it can be seen from Figure 2. In this special case, Algorithm 2 loops over *each of these blocks*, given by $\mathbf{K}[\texttt{row}, \texttt{col}]$, where the subsets row and col are indexed by $i \in [\![0, a-1]\!]$ in Algorithm 2, and performs the matrix multiplication with the corresponding submatrix of $\mathbf{X}$. **The general case $d \geq 1$ is similar: the Kronecker-sparse matrix $\mathbf{K}$ is,** *up to permutation operations*, **block-diagonal with ad dense blocks, and Algorithm 2 loops over each of these dense blocks**, given by $\mathbf{K}[\texttt{row}, \texttt{col}]$ with row and col defined in lines 3 and 4. See Figure 5 for an illustration. More precisely, the support $\mathbf{S}_{\boldsymbol{\pi}}$ associated with a Kronecker-sparsity pattern $\boldsymbol{\pi} = (a, b, c, d)$ can be reduced to the pattern $\tilde{\boldsymbol{\pi}} = (ad, b, c, 1)$, corresponding to a block-diagonal matrix with ad *dense* blocks of size $b \times c$, by permutations:

$$\mathbf{S}_{\boldsymbol{\pi}} = \underbrace{(\mathbf{I}_a \otimes \mathbf{P}_{b,d})}_{:=\mathbf{P}} \underbrace{(\mathbf{I}_{ad} \otimes \mathbf{1}_{b\times c})}_{=\mathbf{S}_{\tilde{\boldsymbol{\pi}}}} \underbrace{(\mathbf{I}_a \otimes \mathbf{P}_{c,d})^\top}_{:=\mathbf{Q}} = \mathbf{P}\mathbf{S}_{\tilde{\boldsymbol{\pi}}}\mathbf{Q}, \tag{1}$$

where $\mathbf{P}_{p,q}$ for two integers p, q is the so-called (p, q) *perfect shuffle* permutation matrix of size $pq \times pq$ [21] (see Appendix C for details). Therefore, for any $\mathbf{K} \in \Sigma^{\boldsymbol{\pi}}$, we have $\mathbf{K} = \mathbf{P}\tilde{\mathbf{K}}\mathbf{Q}$ with $\tilde{\mathbf{K}} := \mathbf{P}^\top\mathbf{K}\mathbf{Q}^\top \in \Sigma^{\tilde{\boldsymbol{\pi}}}$ that is block-diagonal with ad dense blocks of size $b \times c$. This shows that Algorithm 2 is equivalent to Algorithm 1.

Existing matrix multiplication algorithms specialized to Kronecker-sparsity such as bmm and bsr implement Algorithm 1: they directly store $\tilde{\mathbf{K}}^\top$ instead of $\mathbf{K}^\top$, permute the inputs with $\mathbf{Q}$, multiply with $\tilde{\mathbf{K}}^\top$, and repermute with $\mathbf{P}$, resulting in three passes between the global memory and the registers (Figure 3) and a high cost in memory operations.

Instead, we will rather implement the tiling strategy described in Algorithm 2 by splitting the matrices into blocks as pictured in Figure 5, and incrementally accumulating the result. The key is that this tiling strategy allows us to implement our algorithm in a *single* CUDA kernel, resulting in fewer memory transfers between the different levels of GPU memory. This is illustrated in Figure 3.

4.2 Implementation of the new kernel

We implement Algorithm 2 in a single CUDA kernel exploiting tiling (Figure 5). The kernel performs the multiplications $\mathbf{X}[:, \texttt{col}]\mathbf{K}^\top[\texttt{col}, \texttt{row}]$ in parallel for all the pairs $(\texttt{row}, \texttt{col})$, as defined in Algorithm 2. To perform one of these multiplication, the kernel starts by reading into *global memory* the entries in $\mathbf{X}[:, \texttt{col}]$ and $\mathbf{K}^\top[\texttt{col}, \texttt{row}]$, and load them into *shared memory*. Then, it performs the multiplication, which involves passing the data from shared memory to registers, performing the multiplication, and storing the result in shared memory. The kernel then reads the result from shared memory and accumulates it in the output stored in global memory. Standard CUDA optimizations are applied, see Appendix D.2 for details.

4.3 Efficiency analysis and comparison with existing implementations

Comparing memory operations with other baseline implementations. Thanks to tiling, we were able to implement the multiplication in a single kernel, implying a single pass between the global memory and the registers. This contrasts with the three back and forths made by the implementations of Algorithm 1: one pass between the global memory and the registers to perform the permutation with $\mathbf{P}$ (line 3 in Algorithm 1), one for the multiplication with $\tilde{\mathbf{K}}$ (line 2), and another one for the permutation with $\mathbf{Q}$ (line 1). Concretely, the new kernel only reads *once* each coefficient of $\mathbf{X}$, and writes *once* the result of the multiplication $\mathbf{Y}$, while those baseline implementations read *twice* both $\mathbf{X}$ and $\mathbf{Y}$, and rewrite them *once* (to permute them). This is illustrated in Figure 3.

The new kernel has also fewer global memory accesses than the non-specialized `dense` implementation (Section 2.1), since the `dense` implementation also reads the *zero* entries of $\mathbf{K}$ and the corresponding coefficients of $\mathbf{X}$, while our kernel does not.

Finally, compared to the generic `sparse` implementation, while the new kernel has the same number of memory access, it is expected to be more efficient as it the kernel is aware (and tailored) to the Kronecker-sparsity structure while the `sparse` implementation is agnostic to it.

A theoretical analysis of when the new kernel is expected to be more efficient. Since the new kernel has reduced memory operations, we expect it to be more efficient when there is large proportion of time spent on memory operations in the implementations of Algorithm 1. Consider input and output dimensions N, M and a batch-size B. The permutations of the input and the output (lines 3 and 1 in Algorithm 1) require moving all the entries of the input and output tensors in memory, that is $BN + BM$ entries. The number of scalar multiplications in line 1 of Algorithm 1 is $B \times \#\text{nnz}$ (the batch-size times the number of nonzero in $\mathbf{K}$). For a Kronecker-sparse matrix with sparsity pattern $\boldsymbol{\pi} = (a, b, c, d)$, we have $N = acd$, $M = abd$ and $\#\text{nnz} = abcd$. Therefore, the ratio of the number of memory rewritings over the number of scalar multiplications is:

$$\frac{\texttt{number of memory rewritings}}{\texttt{number of scalar multiplications}} = \frac{BN + BM}{B \times \#\text{nnz}} = \frac{b + c}{bc}. \tag{2}$$

These theoretical considerations suggest that $(b + c)/bc$ is a good proxy for the relative time spent on memory rewritings by the implementations of Algorithm 1. This is empirically confirmed in Figure 4 where we observe a positive correlation.

Implication for neural network design. Since our new kernel reduces the cost of memory rewritings, the Kronecker-sparsity patterns with a large value of $(b + c)/(bc)$ will benefit *the most* from our new implementation. This will be empirically confirmed in Section 5. An important consequence of this is that it provides *a heuristic to identify efficient Kronecker-sparsity patterns* and therefore to *help designing efficient Kronecker-sparse neural networks.*

5 Benchmarking the multiplication with a Kronecker-sparse matrix

We now benchmark the different implementations described so far for Kronecker-sparse matrix multiplication. In particular, we validate numerically the benefits of the new `kernel` implementation, with improved memory transfers, compared to the baselines `einsum`, `bsr` and `bmm`.

Protocol. The benchmark is run in *float-precision* on a subset of 600 sparsity patterns $\boldsymbol{\pi} = (a, b, c, d)$ in $\alpha \times \beta \times \beta \times \alpha$, with $\alpha := \{1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128\}$, $\beta := \{48, 64, 96, 128, 192, 256, 384, 512, 768, 1024\}$, such that $b = c$ or $b = 4c$ or $c = 4b$. These patterns correspond to dimensions of Kronecker-sparse matrices $\mathbf{K} \in \mathbb{R}^{M \times N}$ with $(M, N) = (abd, acd)$ in the linear layers of Transformers (up projection for $b = 4c$, down

projection for $c = 4b$, fully-connected layers for $b = c$) and more generally in any neural network. We choose as batch size $B = 128 \times 196 = 25088$, a standard effective batch size for fully-connected layers in ViTs, corresponding to a number of sequences per batch equal to 128, multiplied by a number of tokens per sequence equal to 196. Further details are given in Appendix B.1.

Table 3: Percentage out of 600 patterns (a, b, c, d) where `algo1` is *faster* than the `algo2` (denoted by $\text{time}(\texttt{algo1}) < \text{time}(\texttt{algo2})$), and the median acceleration factor in such cases (that is, the median ratio $\frac{\text{time of }\texttt{algo2}}{\text{time of }\texttt{algo1}}$). For each implementation, we take the minimum time between the *batch-size-first* and the *batch-size-last* memory layout.

$\text{min time}\begin{pmatrix}\texttt{kernel}\\ \texttt{bmm}\\ \texttt{einsum}\\ \texttt{bsr}\end{pmatrix} < \text{min time}\begin{pmatrix}\texttt{dense}\\ \texttt{sparse}\end{pmatrix}$	$\text{time}(\texttt{bmm}) < \text{min time}\begin{pmatrix}\texttt{einsum}\\ \texttt{bsr}\\ \texttt{dense}\\ \texttt{sparse}\end{pmatrix}$	$\text{time}(\texttt{kernel}) < \text{min time}\begin{pmatrix}\texttt{bmm}\\ \texttt{einsum}\\ \texttt{bsr}\\ \texttt{dense}\\ \texttt{sparse}\end{pmatrix}$
99.67% (×6.57)	92.66% (×1.37)	88.10% (×1.39)

Implementations specialized to Kronecker-sparsity improves over generic implementations. The first line of Table 3 shows that at least one of the implementations specialized to the Kronecker structure among `kernel`, `bmm`, `einsum` and `bsr` improves over the generic `dense` and `sparse` implementation, which do not take into account the Kronecker-sparsity. The speedup increases with the matrix size, see Appendix B.3.

The baseline `bmm` is faster than the other baselines `einsum` and `bsr`. This is shown in the second line of Table 3, where the `bmm` implementation improves over $\min(\texttt{einsum}, \texttt{bsr})$ in 93% of the tested cases. The speedup increases with the matrix size, see Appendix B.4. Therefore, when comparing the new `kernel` implementation to other baselines, we will mainly focus on the comparison between `bmm` and `kernel`.

The new `kernel` implementation is faster than existing baselines. The third row of Table 3 shows that `kernel` is faster than all other baselines in 88% of the tested patterns. This empirically validates the benefits of the reduced memory transfer in the `kernel` implementation. In the following, we provide further details on the influence of the memory layout (*batch-size-first* vs. *batch-size-last*) on this improvement. Additionally, we analyze the patterns $\boldsymbol{\pi} = (a, b, c, d)$ for which the `kernel` outperforms baseline implementations.

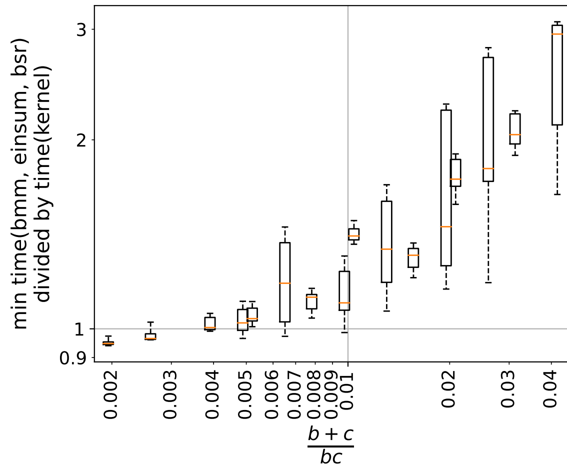


Figure 6: Time speedup factor of `kernel` compared to $\min(\texttt{bmm}, \texttt{einsum}, \texttt{bsr})$. For each implementation, we take the minimum time between the *batch-size-first* and *batch-size-last* memory layouts. We regroup the patterns by their value of $(b + c)/(bc)$, and plot a boxplot to summarize the corresponding measurements.

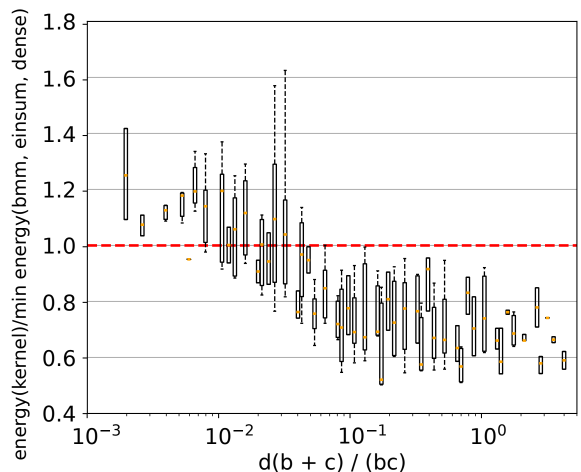


Figure 7: Energy consumed by `kernel` compared to the minimum consumed by `bmm`, `einsum` and `bsr`. For each implementation, we take the minimum energy consumed between the *batch-size-first* and *batch-size-last* memory layouts. We regroup patterns by their value of $d(b + c)/(bc)$.

Impact of the memory layout. For baseline implementations, switching to *batch-size-last* yields a high systematic speedup for `sparse`, high variability in the speedup of `bsr`, and essentially no impact to

negative impact for the other methods, see Appendix B.5 for numerical results. The important part is that it has no impact on `bmm`, and since `bmm` is the fastest baseline implementation (Table 3), switching to *batch-size-last* has no impact on the best of the baseline implementations. However, it yields a systematic speedup (about $\times 2$) for the `kernel` implementation. This acceleration is expected, since the *batch-size-last* memory layout allows for more efficient memory accesses in the `kernel` implementation, as detailed in Section 4.

Analyzing the cases where `kernel` outperforms baselines. As seen in Section 4, the `kernel` has an improved memory access design compared to the rest of the baselines, and it is expected to improve them the most when the ratio $(b+c)/bc$ is large (see (2)). Figure 6 confirms this experimentally: **the `kernel` implementation becomes increasingly time-efficient compared to the baseline implementations as $(b+c)/bc$ increases.**

The `kernel` improves on energy efficiency. Overall, the median energy reduction factor is $\times 0.85$, and the new kernel improves the energy consumption in 72% of the tested cases. The energy measurements are done with the software `pyJoules`. More details about the measurements are in Appendix B.1. It demonstrates that **the `kernel` not only achieves higher time efficiency but also reduces energy consumption** compared to other baselines. This twofold advantage makes the `kernel` an effective solution for improving both performance and sustainability.

A proxy for the energy spent on memory rewritings in the baseline implementations. Figure 7 shows further that the energy efficiency of the `kernel` increases with the value of $d(b+c)/(bc)$. We now give a theoretical explanation for this. For a sparsity pattern $\boldsymbol{\pi} = (a, b, c, d)$, we already discussed that the ratio $(b+c)/bc$ is a good proxy of the relative *time* spent on memory rewritings in practice (see (2)). Since the columns to be rewritten contiguously (i.e., the columns in `col` from Algorithm 2) are equally spaced by d, the energy spent on memory rewritings is expected to increase with d. Multiplying the ratio $(b+c)/bc$ by d can serve as a theoretical proxy for the energy spent on memory rewritings. This is empirically confirmed by the results in Figure 7.

6 Broader implications for neural networks: accelerating inference

The inference of neural networks is claimed to represent 90% of the cost of machine learning at scale according to independent reports from both NVIDIA [10] and Amazon Web Services [11]. We now investigate whether replacing fully-connected layers by products of Kronecker-sparse matrices accelerates the inference. While the same could also apply to other architectures, we will consider Vision Transformers (ViTs) [6]. We find that the computational cost of fully-connected layers is significant in such architectures: depending on the size of the ViT, from 30% to 60% of the total time in a forward pass is spent in fully-connected layers (see Appendix B.6 for details).

Protocol. We benchmark in *float-precision* various components of a ViT-S/16 architecture: a linear layer with bias, an MLP with non-linear activation and/or normalization layers, a multi-head attention module, etc. As in Dao et al. [4], we replace by a product of *two* Kronecker-sparse matrices the weight matrices of linear layers in feed-forward network modules, and the projection matrices for keys, queries and values in multi-head attention modules. We focus on *batch-size-first* as it is the default convention in PyTorch[7]. Details and some additional results are given in Appendix B.7.

Results. We denote by time(`fully-connected`) the inference time with dense matrices (and therefore, with the standard PyTorch implementation). Table 4 shows that time(`kernel`) < time(`bmm`) < time(`fully-connected`) over all the different submodules. **This concretely shows that using Kronecker-sparse matrices and the `kernel` implementation accelerates the inference of standard neural networks.**

Table 4: Acceleration of submodules of a ViT-S/16 using Kronecker-sparse matrices.

	$\frac{\text{time}(\texttt{bmm})}{\text{time}(\texttt{fully-connected})}$	$\frac{\text{time}(\texttt{kernel})}{\text{time}(\texttt{fully-connected})}$
Linear $N \times N$	0.82	**0.50**
Feed-forward network	0.91	**0.77**
Multi-head attention	0.87	**0.79**
Block	0.90	**0.78**
Kronecker-sparse ViT-S/16	0.89	**0.78**

[7]The insertion of Kronecker-sparse matrices in the *batch-size-last* memory layout would *a priori* require a careful implementation of the rest of the operations in *batch-size-last*, that are for now optimized in *batch-size-first* in PyTorch.

7 Conclusion

This work evaluates the efficiency of existing Kronecker-sparse matrix multiplication algorithms on GPU. The benchmark shows that baseline implementations require costly memory rewrites in global memory, which can account up to half of the execution time in practice. To address this, we propose a new tiling strategy that we implement in a single CUDA kernel. This implies reduced memory transfers between the different levels of the GPU. In practice, this new kernel is faster than previous specialized implementations, while also decreasing energy consumption. Moreover, we provide a simple heuristic to choose Kronecker sparsity patterns (a, b, c, d) that are particularly efficient for this implementation. Finally, we show how the kernel can be used to accelerate the inference of neural networks.

Perspectives. The heuristic provided to identify situations where the kernel is expected to be efficient paves the way to new research directions to design efficient Kronecker-sparse neural network architectures.

While we have focused on finding a new tiling strategy to optimize memory management, the part where we multiply the tiles in our kernel may still have room for improvement, especially in *half-precision*. We hope this will encourage work in that direction.

This paper has also demonstrated that some operations (the generic sparse matrix multiplication of `PyTorch`, and the new kernel, see Figure 12) are particularly performant in *batch-size-last*. This paves the way to revisit other common operations in neural networks within the *batch-size-last* memory layout.

Finally, translating our kernel into OpenCL could enable it to run on AMD hardware and other platforms. We also hope that our benchmark will serve as a baseline for comparing Kronecker-sparse implementations on other hardware, such as CPU, Intelligence Processing Unit, FPGA, etc.

References

[1] Simon Boehm. How to optimize a CUDA matmul kernel for cuBLAS-like performance: A worklog, 2022. https://siboehm.com/articles/22/CUDA-MMM [Accessed: April 2024].

[2] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations. In *ICML*, 2019.

[3] Tri Dao, Beidi Chen, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Re. Pixelated butterfly: Simple and efficient sparse training for neural network models. In *ICLR*, 2022.

[4] Tri Dao, Beidi Chen, Nimit Sharad Sohoni, Arjun D. Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 4690–4721. PMLR, 2022. URL https://proceedings.mlr.press/v162/dao22a.html.

[5] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[6] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2020.

[7] Daniel Y Fu, Simran Arora, Jessica Grogan, Isys Johnson, Sabri Eyuboglu, Armin W Thomas, Benjamin Spector, Michael Poli, Atri Rudra, and Christopher Ré. Monarch mixer: A simple sub-quadratic GEMM-based architecture. In *NeurIPS*, 2023.

[8] Rémi Gribonval, Theo Mary, and Elisa Riccietti. Optimal quantization of rank-one matrices in floating-point arithmetic—with applications to butterfly factorizations. preprint, 2023. URL https://inria.hal.science/hal-04125381.

[9] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1510.00149.

[10] HPCwire. AWS Upgrades its GPU-Backed AI Inference Platform. https://www.hpcwire.com/2019/03/19/aws-upgrades-its-gpu-backed-ai-inference-platform/, March 2019. Accessed: [April 2024].

[11] Jeff Barr. Amazon EC2 Update – Inf1 Instances with AWS Inferentia Chips for High Performance Cost-Effective Inferencing. aws.amazon.com/blogs/aws/amazon-ec2-update-inf1-instances-with-aws-inferentia-chips-for-high-performance-cost-effective-inferencing, 2019. Accessed: [April 2024].

[12] Quoc-Tung Le. *Algorithmic and theoretical aspects of sparse deep neural networks*. PhD thesis, ENS Lyon, 2023. URL https://inria.hal.science/tel-04329531.

[13] Quoc-Tung Le, Léon Zheng, Elisa Riccietti, and Rémi Gribonval. Fast learning of fast transforms, with guarantees. In *ICASSP*, 2022.

[14] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019.

[15] Rui Lin, Jie Ran, King Hung Chiu, Graziano Chesi, and Ngai Wong. Deformable butterfly: A highly structured and sparse linear transform. In *NeurIPS*, 2021.

[16] NVIDIA. Efficient GEMM in CUDA: documentation, 2023. https://github.com/NVIDIA/cutlass/blob/main/media/docs/efficient_gemm.md [Accessed: April 2024].

[17] NVIDIA. Matrix multiplication background user's guide, 2023. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html [Accessed: April 2024].

[18] NVIDIA. CUDA C++ programming guide, 2024. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html [Accessed: April 2024].

[19] Alex Rogozhnikov. Einops: Clear and reliable tensor manipulations with einstein-like notation. In *ICLR*, 2021.

[20] Keivan Alizadeh Vahid, Anish Prabhu, Ali Farhadi, and Mohammad Rastegari. Butterfly transform: An efficient FFT based neural architecture design. In *CVPR*, 2020.

[21] Charles F Van Loan. The ubiquitous kronecker product. *Journal of computational and applied mathematics*, 123(1-2):85–100, 2000.

[22] Phil Wang. Scaled dot-product attention implementation, 2024. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html [Accessed: April 2024].

[23] Phil Wang. Simple ViT implementation, 2024. https://github.com/lucidrains/vit-pytorch/blob/main/vit_pytorch/simple_vit.py [Accessed: April 2024].

[24] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *CVPR*, 2022.

[25] Léon Zheng, Gilles Puy, Elisa Riccietti, Patrick Pérez, and Rémi Gribonval. Butterfly factorization by algorithmic identification of rank-one blocks. *arXiv preprint arXiv:2307.00820*, 2023.

[26] Léon Zheng, Elisa Riccietti, and Rémi Gribonval. Efficient identification of butterfly sparse matrix factorizations. *SIAM Journal on Mathematics of Data Science*, 5(1):22–49, 2023.

Appendices

A Related works

We now review the numerical results we found in the literature about time efficiency of existing algorithms for Kronecker-sparse matrix multiplication.

It is reported in Dao et al. [4] that replacing dense matrices by a product of two Kronecker-sparse matrices led to a twice faster training for image classification and language modeling.

In Fu et al. [7] is reported an acceleration of $\mathbf{X} \mapsto \mathbf{W}^{-1}(\mathbf{K} \odot \mathbf{W}\mathbf{X})$ where $\mathbf{K}$ is some dense weight matrix, $\odot$ is the element-wise multiplication, and $\mathbf{W}$ is the DFT matrix (which admits a factorization in Kronecker-sparse matrices), as soon as the dimensions of $\mathbf{W}$ are at least equal to 4096.

Our study is complementary to these observations: we extensively benchmark the efficiency of the Kronecker-sparse matrix multiplication alone.

B Experiments

B.1 Details on the experiments

The pytorch package version is 2.2 and pytorch-cuda is 12.1.

Matrix sizes. In all our experiments with matrices, we set the batch size to $B = 128 \times 196 = 25088$, a very standard choice for ViTs, as this quantity corresponds to the standard number of tokens per sequence (192) multiplied by the standard number of sequences in a batch of inputs (128). When dealing with a batch of images in neural networks, we choose the standard choice of batch size $B = 128$.

Matrix entries. The coordinates of any Kronecker-sparse matrix $\mathbf{K} \in \mathbb{R}^{abd \times acd}$ with sparsity pattern (a, b, c, d) are drawn i.i.d. uniformly in $[-\frac{1}{\sqrt{c}}, \frac{1}{\sqrt{c}}]$, corresponding to the initialization used for training in Dao et al. [4]. The coordinates of the inputs $\mathbf{X}$ are drawn i.i.d. according to a standard normal distribution $\mathcal{N}(0, 1)$.

Benchmarking time execution. All the experiments measuring time execution of a Kronecker-sparse matrix multiplication algorithm (Tables 3, 4, 5 and 8, Figures 4, 6 and 9 to 17) are performed on a NVIDIA A100-PCIE-40GB GPU associated with an Intel(R) Xeon(R) Silver 4215R CPU @ 3.20GHz with 377G of memory. The full benchmark took approximately 3 days in an isolated environment, ensuring that no other processes were running concurrently.

Measurements are done using the PyTorch tool `torch.utils.benchmark.Timer`. The medians are computed on at least 10 measurements of 10 runs. In 94.2% of the cases, we have an interquartile range (IQR) that is at least 100 times smaller than the median (resp. 98% for 50 times smaller, and 99.7% for 10 times smaller).

Benchmarking energy consumption. Measurements of the energy consumption (Figure 7) is done on a NVIDIA Tesla V100-PCIE-16GB GPU associated with an Intel(R) Xeon(R) Silver 4215R CPU @ 3.20GHz with 754G of memory. The full benchmark took approximately 1.5 days in an isolated environment. Measurements are made using the pyJoules software toolkit. The medians are computed on 10 measurements of at least 16 runs. In 96% of the cases, the IQR is at least 10 times smaller than the median, and 5 times smaller in all the cases.

Kronecker-sparsity patterns benchmarked for time measurements (Section 5). The considered patterns are generated by the Python code written in Figure 8. In all the cases, we only consider patterns (a, b, c, d) with $b = c$ or $b = 4c$ or $c = 4d$ to have an input size N and an output size M such that $N = M$ or $N = 4M$ or $M = 4N$. This choice is motivated by the fact that fully-connected layers in ViTs satisfy have input and output sizes satisfying these constraints.

The first "for" loop in Figure 8 generates a wide range of patterns (a, b, c, d) with $a = 1$, as this represents the simplest scenario. Indeed, the case $a > 1$ simply corresponds to repeating a times the case $a = 1$ in parallel.

The second "for" loop in Figure 8 generates patterns with $a > 1$ offering fewer choices for d to keep the benchmark concise in terms of execution time. This loop also imposes additional conditions on b and c (line 28 of the code) that we now explain. Many graphs are plotted based on the ratio $(b + c)/bc$, as introduced in Equation (2). Because of that, our goal was to include as many distinct ratios $(b + c)/bc$ as possible while keeping the benchmark brief. We excluded certain (b, c) values because they resulted in a ratio that was very close to one already in the benchmark and were more computationally intensive.

```python
import itertools

batch_size = 25_088
size_limit = 2_147_483_647

a_list = [1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128]
b_list = [48, 64, 96, 128, 192, 256, 384, 512, 768, 1024]
c_list = [48, 64, 96, 128, 192, 256, 384, 512, 768, 1024]
d_list1 = [1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128]
d_list2 = [4, 16, 64]

def get_patterns_benchmark():
    patterns_list = []

    def add_pattern(a, b, c, d):
        if batch_size * a * c * d <= size_limit and \
           batch_size * a * b * d <= size_limit and \
           a * b * c * d <= size_limit:
            patterns_list.append((a, b, c, d))

    for b, c, d in itertools.product(b_list, c_list, d_list1):
        a = 1
        if (b == c or b == 4 * c or c == 4 * b):
            add_pattern(a, b, c, d)

    for a, b, c, d in itertools.product(a_list, b_list, c_list, d_list2):
        if a != 1 and \
           (b, c) not in [(1024, 256), (256, 1024), (128, 512), (512, 128),
               (64, 256), (256, 64)] and \
           (b == c or b == 4 * c or c == 4 * b):
            add_pattern(a, b, c, d)

    return patterns_list
```

Figure 8: Python code to generate the patterns benchmarked for the execution time in the numerical experiments of Section 5.

Patterns benchmarked for energy measurements (Section 5). For the energy measurements, the goal is to have diverse sparsity patterns (a, b, c, d) corresponding to many different ratios $d(b + c)/bc$ to observe the trend in Figure 7, while keeping the benchmark as short as possible. We chose to consider the cartesian product of

```python
a_list = [1, 4, 16, 32, 64]
b_list = [48, 64, 96, 128, 192, 256, 384, 512, 768, 1024]
c_list = [48, 64, 96, 128, 192, 256, 384, 512, 768, 1024]
d_list = [1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64]
```

by skipping as in Figure 8 all the patterns with

```python
(b,c) in [(1024 , 256) , (256 , 1024) , (128 , 512) , (512 , 128) , (64 , 256)
    , (256 , 64)]
```

and also all the patterns such that

```python
 b != c and b != 4 * c and c != 4 * b
```

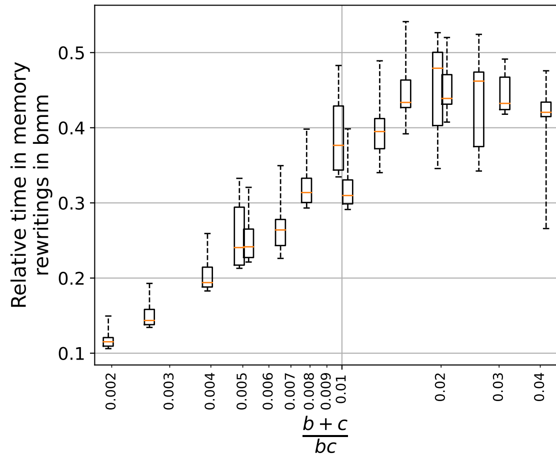
for the same reasons as explained above for time measurements.

Details on boxplots. In all boxplots (Figures 4, 6 to 7 and 9 to 17), the orange line corresponds to the median, the boxes to the first and third quartile and the whiskers to the 5th and the 95th percentile. Outliers are not represented on the graph.

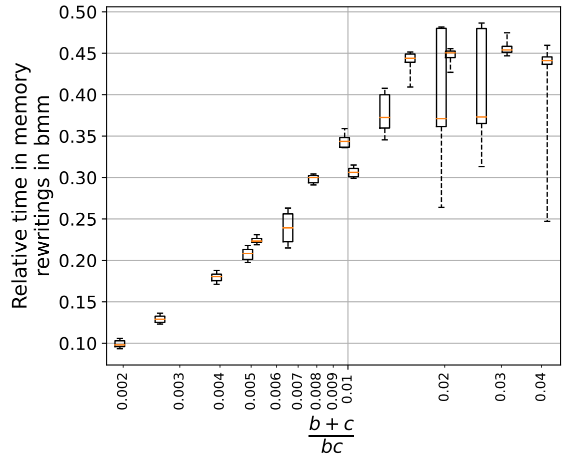
B.2 Estimating the time for memory rewritings in the `bmm` implementation (Section 3)

Protocol. Given a Kronecker-sparsity pattern $\boldsymbol{\pi} = (a, b, c, d)$, an associated $\boldsymbol{\pi}$-Kronecker-sparse matrix $\mathbf{K}$ (Definition 2.1) and an input $\mathbf{X} \in \mathbb{R}^{B \times acd}$ for some batch size B, we first measure the time Δt to compute $\mathbf{Y} := \mathbf{X}\mathbf{K}^\top$ using the `bmm` implementation. Then, we measure the time $\Delta \tilde{t}$ to perform only the multiplication operations $\mathbf{Y}[:, \texttt{row}] = \mathbf{X}[:, \texttt{col}]\mathbf{K}^\top[\texttt{col}, \texttt{row}]$ in the `bmm` implementation (line 2 of Algorithm 1). Therefore, the estimated relative time to perform the memory rewritings of lines 1 and 3 of Algorithm 1 is simply $\frac{\Delta t - \Delta \tilde{t}}{\Delta t}$.

Results. Figure 4, which is replicated in the left part of Figure 9, shows that the relative time spent doing memory rewritings in `bmm` increases with the ratio $(b+c)/(bc)$, in the *batch-size-first* memory layout. Figure 9 shows that this is similar for both *batch-size-first* and *batch-size-last*.



(a) Batch-size-first (same as Figure 4).



(b) Batch-size-last.

Figure 9: Estimated relative time spent on memory rewritings in `bmm` for the multiplication with $\mathbf{K} \in \Sigma^{\boldsymbol{\pi}}$, for several $\boldsymbol{\pi} = (a, b, c, d)$. We regroup patterns by their value of $(b+c)/(bc)$, and plot a boxplot to summarize the corresponding measurements.

B.3 Details on min time(`kernel`, `bmm`, `bsr`, `einsum`) vs. min time(`dense`, `sparse`) (Section 5)

Figure 10 shows that the speed-up factor of implementations specialized to the Kronecker-sparsity (`kernel`, `bmm`, `bsr`, `einsum`) over the generic `dense` and `sparse` implementations increases with the matrix size $M \times N$. We recall that $M = acd$ and $N = abd$ for a Kronecker-sparse matrix with pattern $\boldsymbol{\pi} = (a, b, c, d)$.

B.4 Details on time(`bmm`) vs. min time(`bsr`, `einsum`) (Section 5)

Figure 11 shows that for a sufficient large matrix size $M \times N$, we always have time(`bmm`) $<$ min time(`bsr`, `einsum`), i.e., the `bmm` implementation is the most efficient among all baseline implementations (`bmm`, `einsum`, `bsr`).

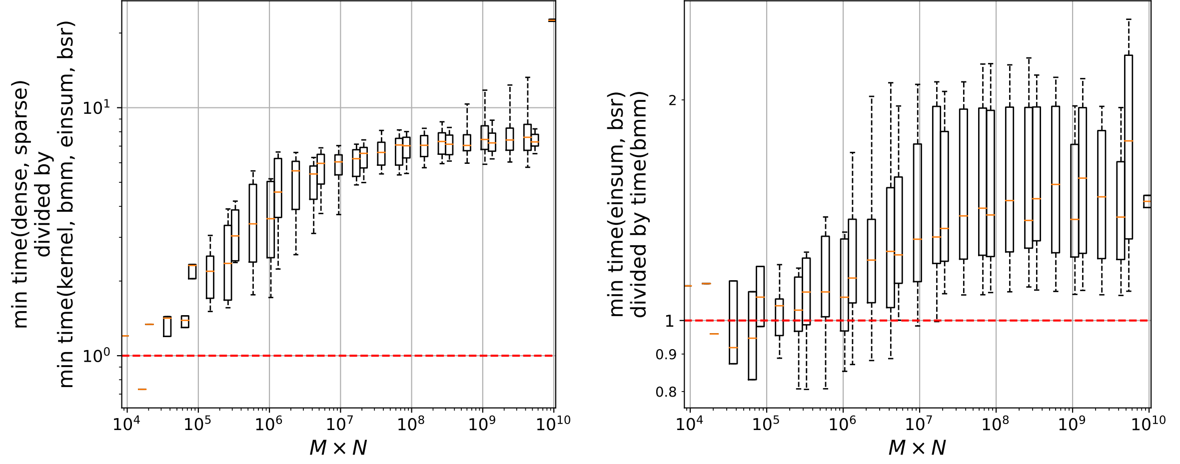


Figure 10: Speed-up factor of min time(`kernel`, `bmm`, `bsr`, `einsum`) compared to min time(`dense`, `sparse`) as a function of the matrix size $M \times N$.

Figure 11: Speed-up factor of time(`bmm`) compared to min time(`einsum`, `bsr`) as a function of the matrix size $M \times N$.

B.5 Details on the impact of the memory layout (Section 5)

Figure 12 shows the impact of the memory layout on the execution time of each implementation.

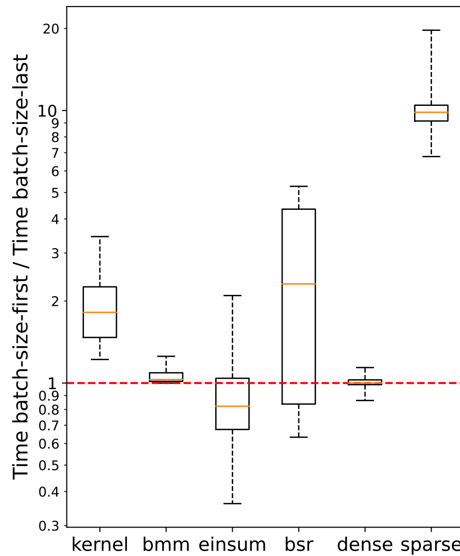


Figure 12: Boxplots of the ratio $\frac{\text{time of } \textit{batch-size-first}}{\text{time of } \textit{batch-size-last}}$.

Table 5 shows the percentage of patterns for which the `kernel` implementation improves over all baseline implementations, either in the *batch-size-first* or the *batch-size-last* memory layout. When restricting all implementations to the *batch-size-first* layout, the `kernel` still improves on 20% of the tested patterns despite non-contiguous memory accesses (Section 4).

Table 5: Percentage out of 600 patterns (a, b, c, d) where `algo1` is *faster* than the `algo2` (denoted by time(`algo1`) < time(`algo2`)), and the median acceleration factor in such cases (that is, the median ratio $\frac{\text{time of algo2}}{\text{time of algo1}}$).

	time(`kernel`) < min time(`bmm`, `einsum`, `bsr`, `dense`, `sparse`)
Batch-size-first	20.0% (×1.28)
Batch-size-last	88.1% (×1.39)

B.6 Time spent in linear layers in vision transformers

This section gives a numerical lower bound estimate on the time spent in fully-connected layers in a Vision Transformer (ViT).

Results. Table 6 shows that, for different ViTs, the fraction of computation time solely dedicated to linear layers in feed-forward network modules varies between 31% and 53% in *half-precision*, and 46% and 61% in *float-precision*. This proportion increases with the size of the architecture. This shows that a non-negligible amount of ViTs inference is dedicated to fully-connected layers. Note that the time for the fully-connected linear layers in the multi-head attention module is not included in our measurements, so our estimate is *only a lower bound* on the time effectively devoted to all fully-connected layers in transformer architectures.

Table 6: Median execution times (ms) of the forward pass in a ViT, and the forward pass in an MLP containing only all the linear layers involved in the feed-forward network modules of the ViT. The latter is reported with its ratio over the first. FP16 is *half-precision*, FP32 is *float-precision*.

Architecture	fp16 (s)		fp32 (s)	
	Complete	Linear in FFNs	Complete	Linear in FFNs
ViT-S/16	0.014	0.0046 (31%)	0.090	0.04 (46%)
ViT-B/16	0.036	0.015 (42%)	0.30	0.16 (54%)
ViT-L/16	0.11	0.050 (46%)	1.0	0.58 (58%)
ViT-H/14	0.31	0.16 (53%)	2.6	1.6 (61%)

Details on the estimation. The transformer architecture is composed of a sequence of transformer blocks, where each block contains a multi-head attention module and a feed-forward network module. The feed-forward network module is an MLP with one hidden layer of neurons, involving two fully-connected linear layers. Table 6 reports the time to perform sequentially all the fully-connected linear layers (without biases) appearing in feed-forward network modules of the considered ViT. This is compared to the total forward time of the transformer network. This is expected to yield a lower bound since we did not measure the time spent in fully-connected linear layers in the multi-head attention module.

Experimental settings. The architecture ViT-S/16 corresponds to the one in [24], while the architecture ViT-B/16, ViT-L/16 and ViT-H/14 correspond to those in [6]. Input images are of size 224×224. In *float-precision*, the PyTorch implementation of ViT architecture are taken from [23]. In *half-precision*, the considered implementation of the transformer architecture uses FlashAttention [5] to compute the scaled dot product attention, like in [22]. The MLP containing only the linear layers of the feed-forward modules in the transformer architecture is implemented using `torch.nn.Sequential` and `torch.nn.Linear`. Experiments are done on a single A100-40GB GPU on AMD EPYC 7742 64-Core Processor. Measurements are done using the PyTorch tool `torch.utils.benchmark.Timer` for benchmarking. The image batch size is set at 128.

B.7 Details on the acceleration of the inference of a ViT (Section 6)

Chosen Kronecker-sparse matrices. The dense weight matrices are replaced by products of two Kronecker-sparse matrices $\mathbf{K}_1\mathbf{K}_2$ (Definition 2.1) with respective sparsity patterns $\boldsymbol{\pi}_1, \boldsymbol{\pi}_2$ given by: $(1, 192, 48, 2), (2, 48, 192, 1)$ for the size $N \times N$, $(1, 768, 192, 2), (6, 64, 64, 1)$ for the size $4N \times N$, $(1, 768, 192, 2), (6, 64, 64, 1)$ for the size $4N \times N$.

Additional results. Table 7 provides additional results to Table 4 on linear submodules of a ViT-S/16.

Table 7: Acceleration of submodules of a ViT-S/16 using Kronecker-sparse matrices.

	$\frac{\text{time}(\texttt{bmm})}{\text{time}(\texttt{fully-connected})}$	$\frac{\text{time}(\texttt{kernel})}{\text{time}(\texttt{fully-connected})}$
Linear $N \times N$	0.82	**0.50**
Linear $N \times N$ + bias	0.97	**0.66**
Linear $4N \times N$	0.80	**0.78**
Linear $4N \times N$ + bias	0.93	**0.90**
Linear $N \times 4N$	0.91	**0.58**
Linear $N \times 4N$ + bias	0.94	**0.61**

B.8 Additional results in half-precision

For the sake of completeness we perform the benchmark described in Section 5 in *half-precision*. The equivalent of Table 3, Figure 6, Figures 9 to 12 in *half-precision* are Table 8, Figure 13, Figures 14 to 17, respectively. Note that just as Figure 6, the Figure 13 only considers sparsity patterns for which min time(`kernel`, `bmm`, `bsr`, `einsum`) < min time(`dense`, `sparse`). This corresponds to 87% of the tested patterns in *half-precision*, cf. Table 8.

Table 8: Percentage out of 600 patterns (a, b, c, d) where `algo1` is *faster* than the `algo2` in *half-precision* (denoted by time(`algo1`) < time(`algo2`)), and the median acceleration factor in such cases (that is, the median ratio $\frac{\text{time of } \texttt{algo2}}{\text{time of } \texttt{algo1}}$). For each implementation, we take the minimum time between the *batch-size-first* and the *batch-size-last* memory layout. Experiments are carried in *half-precision*.

$\min \text{time}\begin{pmatrix}\texttt{kernel}\\ \texttt{bmm}\\ \texttt{einsum}\\ \texttt{bsr}\end{pmatrix} < \min \text{time}\begin{pmatrix}\texttt{dense}\\ \texttt{sparse}\end{pmatrix}$	$\text{time}(\texttt{bmm}) < \min \text{time}\begin{pmatrix}\texttt{einsum}\\ \texttt{bsr}\\ \texttt{dense}\\ \texttt{sparse}\end{pmatrix}$	$\text{time}(\texttt{kernel}) < \min \text{time}\begin{pmatrix}\texttt{bmm}\\ \texttt{einsum}\\ \texttt{bsr}\\ \texttt{dense}\\ \texttt{sparse}\end{pmatrix}$
86.95% (×8.45)	83.22% (×1.83)	36.69% (×1.46)

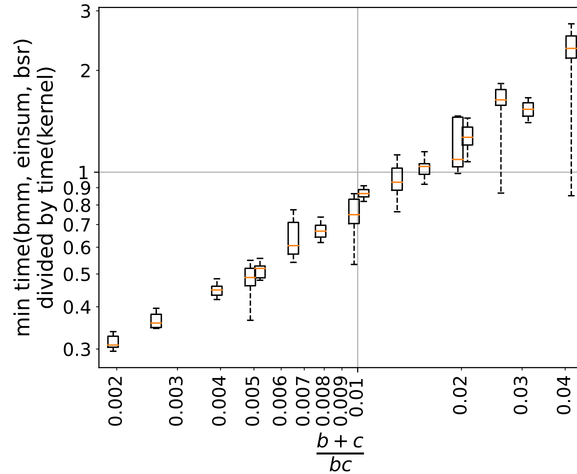
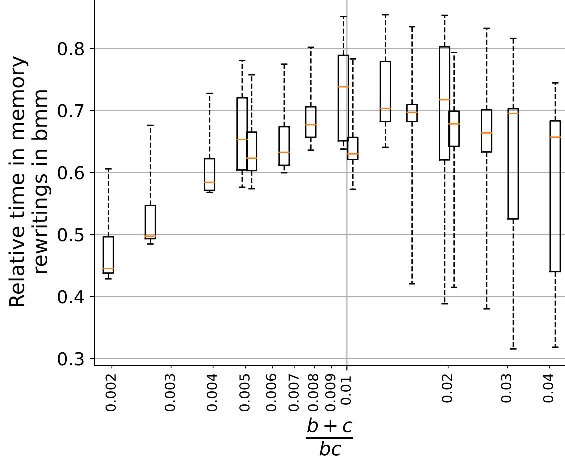
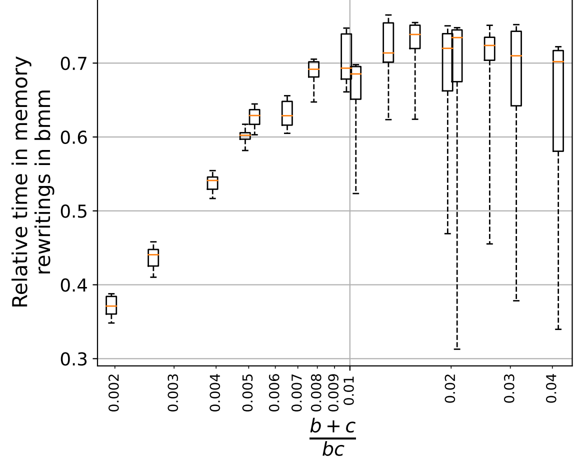


Figure 13: Speedup factor of `kernel` compared to min(`bmm`, `einsum`, `bsr`) in *half-precision*. For each implementation, we take the minimum time between the *batch-size-first* and the *batch-size-last* memory layout. We regroup the (a, b, c, d) patterns by their value of $(b + c)/(bc)$, and use a boxplot to summarize the corresponding measurements. Experiments are carried in *half-precision*.



(a) Batch-size-first.



(b) Batch-size-last.

Figure 14: Estimated relative time spent on memory rewritings in `bmm` for the multiplication with $\mathbf{K} \in \Sigma^{\boldsymbol{\pi}}$, for several $\boldsymbol{\pi} = (a, b, c, d)$. We regroup patterns by their value of $(b + c)/(bc)$, and plot a boxplot to summarize the corresponding measurements. Experiments are carried in *half-precision*.

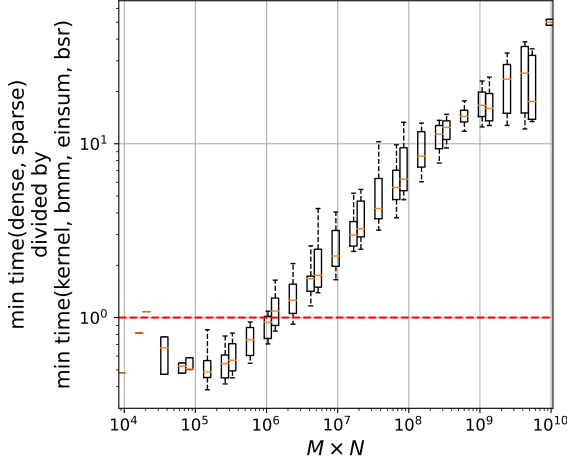


Figure 15: Speed-up factor of min time(`kernel`, `bmm`, `bsr`, `einsum`) compared to min time(`dense`, `sparse`) vs. the matrix size $M \times N$. Experiments are carried in *half-precision*.

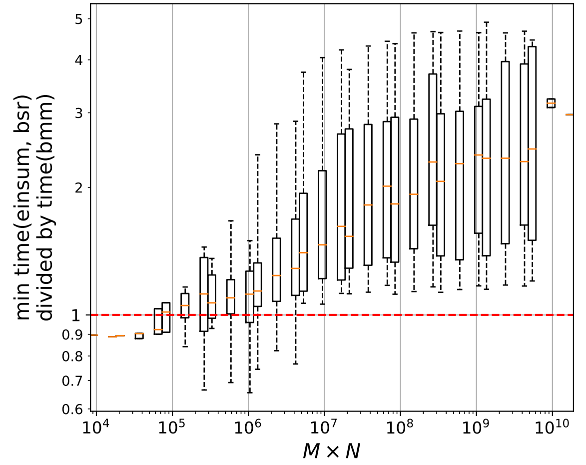


Figure 16: Speed-up factor of time(`bmm`) compared to min time(`einsum`, `bsr`) vs. the matrix size $M \times N$. Experiments are carried in *half-precision*.

C Details on perfect shuffle permutations

The goal is to prove Equation (1), which we recall here for convenience:

$$\mathbf{S}_{\boldsymbol{\pi}} = \underbrace{(\mathbf{I}_a \otimes \mathbf{P}_{b,d})}_{:=\mathbf{P}} \underbrace{(\mathbf{I}_{ad} \otimes \mathbf{1}_{b \times c})}_{=\mathbf{S}_{\tilde{\boldsymbol{\pi}}}} \underbrace{(\mathbf{I}_a \otimes \mathbf{P}_{c,d})^\top}_{:=\mathbf{Q}} = \mathbf{P}\mathbf{S}_{\tilde{\boldsymbol{\pi}}}\mathbf{Q},$$

where the matrix $\mathbf{P}_{p,q}$ is the so-called (p, q) perfect shuffle permutation introduced below. To prove this formula, we will use the next lemma.

Lemma C.1. *For any positive integers b, c, d:*

$$\mathbf{P}_{b,d}{}^\top (\mathbf{1}_{b \times c} \otimes \mathbf{I}_d) \mathbf{P}_{c,d} = \mathbf{I}_d \otimes \mathbf{1}_{b \times c},$$

where $\mathbf{P}_{p,q}$ denotes the (p, q) perfect shuffle *of $r := pq$ [21], which is the permutation matrix of size $r \times r$ defined as:*

$$\mathbf{P}_{p,q} := \begin{pmatrix} \mathbf{I}_r[R_0, :] \\ \mathbf{I}_r[R_1, :] \\ \vdots \\ \mathbf{I}_r[R_{q-1}, :] \end{pmatrix}, \tag{3}$$

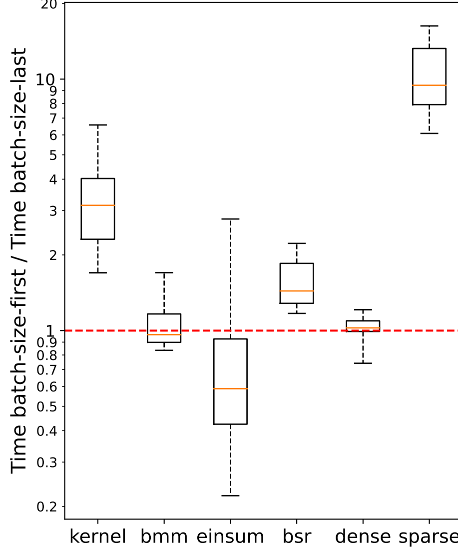


Figure 17: Boxplots of the ratio $\frac{\text{time of } \textit{batch-size-first}}{\text{time of } \textit{batch-size-last}}$ in *half-precision*.

where $R_i := \{i + qj \mid j \in [\![0, p-1]\!]\}$ *for* $i \in [\![0, q-1]\!]$.

Proof of Lemma C.1. This is a direct consequence of a more general result claiming that the Kronecker product commutes up to some perfect shuffle permutation matrices [21, Section 1]. ☐

We now turn to the proof of Equation (1).

Proof of Equation (1). By definition, $\mathbf{S}_{\boldsymbol{\pi}} = \mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d$ when $\boldsymbol{\pi} = (a, b, c, d)$. By Lemma C.1,

$$\mathbf{S}_{\boldsymbol{\pi}} = \mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d = \mathbf{I}_a \otimes \left(\mathbf{P}_{b,d}(\mathbf{I}_d \otimes \mathbf{1}_{b\times c})\mathbf{P}_{c,d}{}^\top\right).$$

By the equality $(\mathbf{AB}) \otimes (\mathbf{CD}) = (\mathbf{A} \otimes \mathbf{C})(\mathbf{B} \otimes \mathbf{D})$ for any matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ of compatible sizes, we get the result:

$$\begin{aligned}
\mathbf{S}_{\boldsymbol{\pi}} &= \mathbf{I}_a \otimes \left(\mathbf{P}_{b,d}(\mathbf{I}_d \otimes \mathbf{1}_{b\times c})\mathbf{P}_{c,d}{}^\top\right) \\
&= (\mathbf{I}_a \otimes \mathbf{P}_{b,d}) \left(\mathbf{I}_a \otimes \left((\mathbf{I}_d \otimes \mathbf{1}_{b\times c})\mathbf{P}_{c,d}{}^\top\right)\right) \\
&= (\mathbf{I}_a \otimes \mathbf{P}_{b,d})(\mathbf{I}_a \otimes \mathbf{I}_d \otimes \mathbf{1}_{b\times c})(\mathbf{I}_a \otimes \mathbf{P}_{c,d}{}^\top) \\
&= (\mathbf{I}_a \otimes \mathbf{P}_{b,d})(\mathbf{I}_{ad} \otimes \mathbf{1}_{b\times c})(\mathbf{I}_a \otimes \mathbf{P}_{c,d}{}^\top).
\end{aligned}$$

☐

D Implementations

D.1 Details on baseline GPU implementations

To keep it short, we only give the code in the case of the *batch-size-first* memory layout (except for `dense` and `sparse` where the codes are small). The case of *batch-size-last* can simply be obtained by inverting the first and last positions in all tensor reshapings.

`einsum` implementation. This implementation uses tensor contractions with the high-performance `einops` library. The *abcd* nonzero entries of the Kronecker-sparse matrix $\mathbf{K}$ (Figure 2) are stored in a `PyTorch` 4D-tensor `K_einsum` of shape (a, b, c, d). The implementation uses Einstein notations.

```
1 def kronecker_einsum(X_bsf, K_einsum):
2     X_perm = einops.rearrange(X_bsf, "... (a c d) -> ... a c d", a=a, c=c, d=d)
3     Y_perm = einops.einsum(X_perm, K_einsum, "... a c d, a b c d -> ... a b d")
4     Y_bsf = einops.rearrange(Y_perm, "... a b d-> ... (a b d)")
5     return Y_bsf
```

The second line of this code does at the same time all the matrix multiplications $\mathbf{Y}[:, \texttt{row}] \leftarrow \mathbf{X}[:, \texttt{col}]\mathbf{K}^\top[\texttt{col}, \texttt{row}]$ for all the pairs $(\texttt{row}, \texttt{col})$ in Algorithm 2.

`bsr` **implementation.** This is an implementation of Algorithm 1 using the high-performance Block compressed Sparse Row (BSR) `PyTorch` library. The matrix $\tilde{\mathbf{K}}$ is stored as a tensor `K_bsr` stored in the BSR format.

```
def kronecker_bsr(X_bsf, K_bsr):
    batch_size = X_bsf.shape[0]
    X_perm = (
        X_bsf.view(batch_size, a, c, d)
        .transpose(-1, -2)
        .reshape(batch_size, a * c * d)
    )
    Y_perm = torch.nn.functional.linear(
        X_perm, K_bsr
    )
    Y_bsf = (
        Y_perm.view(batch_size, a, d, b)
        .transpose(-1, -2)
        .reshape(batch_size, a * b * d)
    )
    return Y_bsf
```

`bmm` **implementation.** This is an implementation of Algorithm 1 using the high-performance Block compressed Sparse Row (BSR) `PyTorch` library. The matrix $\tilde{\mathbf{K}}$ is stored as a tensor `K_bsr` stored in the BSR format. This implementation using `torch.bmm`, which is based on high-performance batched matrix multiplication NVIDIA routines. The non-zero entries of $\tilde{\mathbf{K}}$ are stored in a four-dimensional `PyTorch` tensor `K_bmm` of shape $(a * d, b, c)$.

```
def kronecker_bmm(X_bsf, K_bmm):
    batch_size = X_bsf.shape[0]
    X_perm = (
        X_bsf.view(batch_size, a, c, d)
        .transpose(-1, -2)
        .reshape(batch_size, a * d, c).
        contiguous().
        transpose(0, 1)
    )
    Y_perm = torch.empty(batch_size, a * d, b, device=x.device, dtype=x.dtype).
       transpose(0, 1)
    Y_perm = torch.bmm(X_perm, K_bmm.transpose(-1, -2))
    Y_bsf = (
        Y_perm.transpose(0, 1)
        .reshape(batch_size, a, d, b)
        .transpose(-1, -2)
        .reshape(batch_size, a * b * d)
    )
    return
```

`dense` **implementation.** This ignores the sparsity of the Kronecker-sparse matrix $\mathbf{K}$, that is stored as a dense matrix in a 2d-tensor `K_dense`.

batch-size-first: `torch.nn.functional.linear(X_bsf,K_dense)`

batch-size-last: `torch.matmul(K_dense,X_bsl)`

The implementation in *batch-size-first* is the default PyTorch implementation of a forward pass of a linear layer. For *batch-size-last*, we had to choose an implementation since Pytorch uses *batch-size-first* by default. We made our choice based on a small benchmark of different alternatives.

`sparse` **implementation.** This exploits the sparsity of the Kronecker-sparse matrix $\mathbf{K}$ but not its structure (recall that the support are not arbitrary, they are structured since they must be expressed as Kronecker products, see Definition 2.1).

batch-size-first: `torch.nn.functional.linear(X_bsf,K_csr)`

batch-size-last: `torch.matmul(K_csr,X_bsl)`

D.2 Details on the kernel implementation

Classical optimizations that we build upon. The proposed implementation use *vectorization* as soon as an operation can be vectorized. Concretely, the float4 and half2 vector types are used to mutualize read/write operations [1, 16, 17, 18]. An *epilogue* [16] is also implemented to avoid writing in global memory in a disorganized way. Indeed, after having accumulated the output in registers, each thread has specific rows and columns of the output to write to global memory, and may finish its computation before the others. To avoid that, the epilogue starts to write in the shared memory, in a disorganized way, and then organize the writing from shared to global memory. Another implemented optimization is *double buffering* [1, 14, 16, 17]: a thread block is always both computing the output of a tile, and loading the next tile from global to shared memory. This allows us to hide some latency that arises when loading from the global memory.

Note that as with any CUDA kernel, the constants (such as the number of threads) need to be tailored to each specific case of use —here, each Kronecker-sparsity pattern $\boldsymbol{\pi} = (a, b, c, d)$— and to each GPU.