Compiling the Einstein summation convention to LIBXSMM library calls

BACHELOR'S THESIS

to attain the academic degree
Bachelor of Science (B.Sc.)
in Computer Science (Informatik)

FRIEDRICH SCHILLER UNIVERSITY JENA
Faculty of Mathematics and Computer Science

submitted by Max Koch
born on 07.09.2001 in Meiningen
advisor: Prof. Dr. A. Breuer
Jena, 17.08.2023

Abstract

Tensors are the backbone of multiple scientific applications. With their rise in popularity, it is important to research high-performance routines for tensor computations. The Einstein summation convention (einsum) is a powerful way to express what should be computed. By assigning a character to every dimension of possibly multiple tensors, it can encode anything from matrix multiplications to tensor network contractions or even more general tensor operations. To minder the computational needs, the contraction of an einsum expression can be performed in binary fashion. During this process, intermediate tensors are stored to keep track of the results of binary contractions. In this thesis, I use LIBXSMM [13], a library targeting small matrix multiplications, to build a library for calculating einsum expressions. I describe a memory layout for intermediate tensors on which matrix multiplications can be performed fast. To achieve this layout, I present an unpacking routine that writes result elements of sub-matrices directly to the respective memory locations in the result tensor of the binary contraction. Furthermore, I show results regarding my implementation and compare them to the einsum implementation of PyTorch [20].

Kurzfassung

Tensoren sind das Rückgrat zahlreicher wissenschaftlicher Anwendungen. Angesichts ihrer zunehmenden Beliebtheit ist es wichtig, leistungsstarke Routinen für Tensoroperationen zu erforschen. Die Einsteinsche Summenkonvention (Einsum) ist eine Notation für Tensoroperationen. Durch die Zuweisung eines Buchstabens zu jeder Dimension von möglicherweise mehreren Tensoren kann sie alles von Matrixmultiplikationen bis hin zu Tensornetzwerkkontraktionen oder noch allgemeineren Kontraktionen zwischen Tensoren ausdrücken. Um den Rechenaufwand zu verringern, kann die Kontraktion eines Einsum-Ausdrucks auf binäre Weise durchgeführt werden. Während dieses Prozesses werden Zwischentensoren gespeichert, um die Ergebnisse der binären Kontraktionen zu sichern. In dieser Arbeit verwende ich LIBXSMM [13], eine Bibliothek für kleine Matrixmultiplikationen, um eine Bibliothek zur Berechnung von Einsum-Ausdrücken zu erstellen. Ich beschreibe ein Speicherlayout für Zwischentensoren, mit dem Matrixmultiplikationen schnell und effektiv durchgeführt werden können. Um dieses Layout zu erreichen, stelle ich eine Routine vor, die Ergebnismatrizen direkt an die entsprechenden Speicheradressen im Ergebnistensor der binären Kontraktion schreibt. Darüber hinaus zeige ich Ergebnisse zu meiner Implementierung und vergleiche sie mit der Einsum-Implementierung von PyTorch [20].

Contents

Acknowledgements

I want to thank everyone who helped me during the process of writing this thesis. Special thanks go to my supervisor Prof. Dr. Alexander Breuer for providing feedback regarding my research and explanations about topics I had not much knowledge about when I first started working on this thesis. I am very grateful for my office mates who motivated me during the writing process and were always helpful by having an open ear for problems regarding implementation details.

This endeavor would not have been possible without the support of my family, especially my parents, who always stood behind me and believed in me.

1 Introduction

Tensors are an essential part of multiple scientific applications. Being the backbone of research in chemistry, physics and quantum physics, math and machine learning, their importance is clearly visible [8, 1, 6, 21]. Tensors are a multidimensional arrangement of numbers, like vectors and matrices. They can be thought of as a way to store information in multiple dimensions or as an extension to transformations, like higher dimensional matrix multiplications [29]. Because of their popularity and importance, research is not only going down the path of possible applications but also high-performance computing. With a growing demand for using tensors, it is crucial to perform calculations and operations with and on tensors fast and efficiently. For example, Nvidia has built-in tensor cores in their recent graphics card A100 series [2], supporting tensor operations on a hardware level. However, there is still a need to write high-performance software routines that can use the given hardware, GPUs and CPUs, to perform several tensor operations for tensors coming in all shapes and sizes. Depending on their function, tensors often do not appear alone. Multiple tensors can be arranged in a tensor network serving as a factorization of a single tensor that can be calculated by contracting the network [11]. To minder the computational load, the networks are usually contracted by contracting two tensors at once until one tensor is left. A binary tensor contraction is essentially encoding multiplications and additions enforced on respective dimensions of the two input tensors.

The Einstein summation convention (einsum) has been established as a powerful way to express operations on one or multiple tensors. An einsum expression is a string referring to every dimension of every input tensor with a single character. With this, it is possible to express numerous tensor operations. They can be calculated in one step using multiple nested loops. But similar to tensor network contractions, a computationally less expensive way of computing a given einsum expression is to compute it in a sequence of binary contractions.

A popular method for calculating tensor contractions and einsum expressions is by using matrix multiplications. One simple approach, known as TTGT, is implemented by multiple tensor libraries like PyTorch [20]. TTGT is applicable for all tensors but needs to perform several memory operations to be able to use matrix multiplication to calculate binary contractions. Finding a way to minimize the memory overhead is crucial for runtime improvements and is part of active research [19, 26]. Binary contracting an einsum expression results in the appearance of *intermediate tensors*. These tensors don't serve as input tensors for the einsum, nor do they represent the final result tensor. They are simply used to store intermediate results. Since they are not needed before and after the calculation of an einsum expression, they can have any memory layout, as long as the contraction routine can process it. In this thesis, I propose a memory layout for intermediate tensors that assists high-performance contractions. I also show how these layouts can be achieved by writing elements of the result tensor to the right memory address immediately after they have been computed, saving memory access costs. As the backbone for matrix multiplication, I use LIBXSMM [13], a library targeting small matrix multiplica-

tions. Generated matrix multiplication kernels by LIBXSMM achieve almost peak floating point performance on a supported CPU.

This thesis has the goal to be self-explanatory. To accomplish that, section 2 provides an introduction to tensor operations and network contractions, as well as the einsum notation. The following section 3 explains approaches on how to calculate tensor network contractions and einsum expressions using matrix multiplications. Additionally, there is a short explanation of the functionality of LIBXSMM. Related work is presented in section 4. Section 5 describes the main contribution of this thesis, a beneficial memory layout for intermediate tensors and a way to immediately write result elements to their respective memory locations. Runtime experiment results are presented in section 6, followed by a conclusion and possible future research.

2 Tensors and tensor operations

This section provides an introduction to the necessary knowledge about tensors and tensor operations to be able to understand key concepts appearing in this thesis. It is important to say that none of these concepts are new. They have been explained and used before in multiple research papers [19, 26] or websites [29].

Tensors can be thought of as a multidimensional arrangement of numbers or multidimensional arrays. The *type* of a tensor describes its number of dimensions. For example, a scalar is a tensor of type 0, a vector is a tensor of type 1 and a matrix is a type 2 tensor. A tensor of type 3 can be seen as a cuboid of numbers. Generally, one scalar in a tensor of type t can be addressed with exactly t indices. Fig. 1a shows a collection of said tensors with indices that address each element.

I will be denoting Tensors as $\mathcal{T}_{d_0,d_1,\ldots,d_{n-1}} \in \mathbb{R}^{s_0 \times s_1 \times \ldots \times s_{n-1}}$, where $\mathcal{T}$ is a tensor of type n with dimensions $d_0, d_1, \ldots, d_{n-1}$ of sizes $s_0, s_1, \ldots, s_{n-1}$. Its set of dimensions is denoted as $D_\mathcal{T} = \{d_0, d_1, \ldots, d_{n-1}\}$, also meaning that $|D_\mathcal{T}|$ describes the type of $\mathcal{T}$. The size of a dimension $d \in D_\mathcal{T}$ is written as $|d|$, so that $\forall d_i \in D_\mathcal{T} : |d_i| = s_i$ holds. Furthermore, $|\mathcal{T}|$ stands for the number of elements in the tensor. To reference an element of $\mathcal{T}$, I use the notation $[\mathcal{T}](i_0, i_1, \ldots, i_{n-1})$ where the ordered indices $i_0, i_1, \ldots, i_{n-1}$ refer to respective dimensions $d_0, d_1, \ldots, d_{n-1}$, encoding the position of the element. To be able to reference elements without a strict order of dimensions, $[\mathcal{T}]\{d_1 = i_1, d_0 = i_0, \ldots, d_{n-1} = i_{n-1}\}$ is an equivalent notation. Additionally, if the assignment of indices to dimensions is clearly given by the context of the expression, the notation $[\mathcal{T}]\{i_1, i_0, \ldots, i_{n-1}\}$ is applicable. To be more comprehensive, I will sometimes write about increasing a dimension by x when the correct phrase would be to increase the index regarding a dimension by x. Sometimes, index names will have the name of the dimensions, mixing the two. Table 1 provides an overview of the above notations.

The vector, matrix, and tensor of type 3 in Fig. 1a could be denoted as $\mathcal{V}_x \in \mathbb{R}^5$, $\mathcal{M}_{m,n} \in \mathbb{R}^{4\times5}$ and $\mathcal{T}_{a,b,c} \in \mathbb{R}^{3\times4\times5}$.

Notation	Description
$\mathcal{T}_{d_0,d_1,\ldots,d_{n-1}} \in \mathbb{R}^{s_0 \times s_1 \times \ldots \times s_{n-1}}$	Tensor $\mathcal{T}$ of type n with dimensions $d_0, d_1, \ldots, d_{n-1}$ of sizes $s_0, s_1, \ldots, s_{n-1}$
$\lvert d \rvert$	Size of the dimension d
$D_{\mathcal{T}}$	Set of dimensions of tensor $\mathcal{T}$
$\lvert D_{\mathcal{T}} \rvert$	Type of tensor $\mathcal{T}$
$\lvert \mathcal{T} \rvert$	Number of elements in tensor $\mathcal{T}$
$[\mathcal{T}](i_0, i_1, \ldots, i_{n-1})$	Element of tensor $\mathcal{T}$ at position $(i_0, i_1, \ldots i_{n-1})$
$[\mathcal{T}]\{d_1 = i_1, \ldots, d_{n-1} = i_{n-1}\}$	Element of tensor $\mathcal{T}$ identified with unordered indices
$[\mathcal{T}]\{i_1, i_0, \ldots, i_{n-1}\}$	Element of tensor $\mathcal{T}$ identified with unordered indices and clear dimension relation

Table 1: Tensor notations

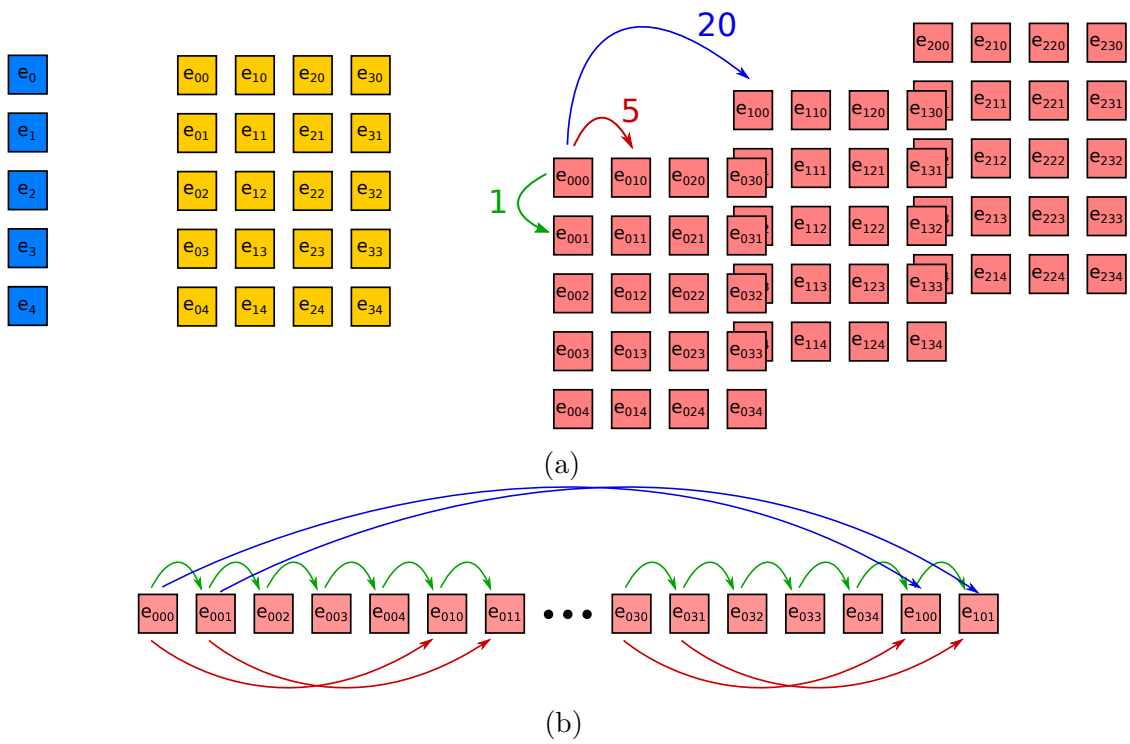


Figure 1: (a) Tensors of different types. (b) Memory layout of the red tensor with strides for each dimension.

2.1 Tensors in memory

To understand how upcoming tensor operations on tensors of arbitrary types and dimension sizes can be calculated, it is important to understand how tensors are represented in memory and what additional information is needed and saved.

Let's first consider an ordinary matrix $\mathcal{M}$. A matrix has two dimensions, m rows and n columns. Every element of the matrix can be addressed with exactly two indices. In a C-style program, one could do so by defining and allocating a two-dimensional array `mat[m][n]`. An element $e_{i,j}$ of the matrix could then be referenced by using a syntax similar to `mat[i][j]`. Note that depending on the allocation process, the elements of the matrix do not have to be contiguous in memory, so going from the last element of a row/column to the first element of the next row/column could result in a memory jump larger than one. For now, assume that all $|\mathcal{M}|$ elements of $\mathcal{M}$ are stored in one contiguous block of memory fitting exactly $|\mathcal{M}|$ elements. $\mathcal{M}$ is considered to be in **row-major** format if the order of elements is similar to reading the matrix from left to right. The order of dimensions would then be $\mathcal{M}_{m,n}$, so increasing the column index (dimension n) results in stepping one element forward in memory. In contrast, $\mathcal{M}$ is in **column-major** format if the elements of the first column are stored first, then the second column, and so on. The matrix is denoted as $\mathcal{M}_{n,m}$ in column-major format.

As an example, the memory layout of the matrix $\mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ in row major format is $\mathcal{A}_{row} = (1, 2, 3, 4, 5, 6)$ and in column major format it is $\mathcal{A}_{col} = (1, 4, 2, 5, 3, 6)$. Note that $\mathcal{A}_{row}$ and $\mathcal{A}_{col}$ can be seen as one-dimensional arrays/vectors, where each of the matrix elements can be addressed by only one index.

2.1.1 Storing tensors as one-dimensional arrays

It is easy to observe that given tensors with arbitrary numbers of dimensions, allocating memory, as well as writing tensor processing code that needs to reference the respective elements using multiple `[]`-operands is tedious and no longer an option. However, similar to the $\mathcal{A}_{row}$ and $\mathcal{A}_{col}$-vectors, a tensor of type t can always be seen as a vector of scalars $vec_{\mathcal{T}}$, where a relation exists between the indices $i_0, i_1, i_2, ..., i_{t-1}$ and the location of the respective element in the vector.

Again, let us consider the matrix $\mathcal{A}_{m,n} \in \mathbb{R}^{2\times 3}$ in row major format with $A_{row} = \{1, 2, 3, 4, 5, 6\}$. Referencing the element $[A](row, col)$ can also be achieved by referencing $[\mathcal{A}_{row}](row \cdot 3 + col)$. Increasing the row index by one results in a memory jump of 3 in A_{row}, since A has three columns. Similarly, assuming $\mathcal{A}_{n,m}$ to be in column major format with $\mathcal{A}_{col} = (1, 4, 2, 5, 3, 6)$, an element $[A](col, row)$ can be found at $[\mathcal{A}_{col}](col \cdot 2 + row)$ since $\mathcal{A}$ has two rows. The index of an element in the vector is called the **offset** of the element in a tensor. The factors multiplied with the indices are called **strides** which describe the sizes of the memory jumps that occur by increasing respective dimensions by one. The concept of strides can be applied to any tensor of any type. A stride $str(d_i)$ of a dimension $d_i \in D_{\mathcal{T}}$ can then be calculated by simply multiplying all the dimension sizes of later dimensions. The last dimension $d_{|D|-1}$ is called the **fast dimension** because its stride $str(|D_{\mathcal{T}}| - 1)$

is simply 1.

As an example for strides, consider the tensor of type four $\mathcal{T}_{d_0,d_1,d_2,d_3} \in \mathbb{R}^{2\times5\times3\times4}$ being stored in a contiguous block of memory. Increasing d_3 results in a memory jump of 1, so $str(d_3) = 1$. By increasing d_2 by one, a whole dimension d_3 is jumped over, meaning that $str(d_2) = |d_3| = 4$. Incrementing d_1 means jumping over two dimensions: d_2 and d_3, resulting in a stride $str(d_1) = |d_2||d_3| = str(d_2)|d_2| = 12$. Similarly, $str(d_0) = |d_1||d_2||d_3| = str(d_1)|d_1| = 60$. The element $[\mathcal{T}](1,3,2,0)$ can be found in memory with an offset of $1str(d_0) + 3str(d_1) + 2str(d_2) + 0str(d_3) = 60 + 36 + 8 + 0 = 104$. Another example of strides is given in Fig. 1b, where strides and memory jumps are shown for the red tensor.

Generally, strides can be calculated by

$$str(d_i) = \prod_{k>i} |d_k| \tag{1}$$

and the offset of a single element with indices $(i_0, i_1, \ldots, i_{n-1})$ is expressed by

$$\text{offset} = \sum_{k=0}^{k=n-1} i_k s(d_i). \tag{2}$$

A tensor memory layout following equation (1) has **unit stride**.

With the above addressing of tensor elements using strides, one simply has to store all the values of the tensor in an array along with the dimension sizes to be able to address every single element. Furthermore, storing the strides helps with faster offset calculation and is essential for tensor representations that have **non-unit stride** (see permutation).

2.2 Operations on tensors

2.2.1 Permutation

A permutation on a tensor is a reordering of its dimensions, where any order can be achieved. One simple example is the permutation of a matrix with $\sigma = (1, 0)$ resulting in the transposed matrix. Fig. 2a shows a tensor $\mathcal{T}_{d_0,d_1,d_2} \in \mathbb{R}^{3\times2\times4}$ being permuted with $\sigma = (1, 2, 0)$, resulting in the permuted blue tensor $\mathcal{P}_{d_0,d_1,d_2} \in \mathbb{R}^{2\times4\times3}$. The figure also shows the new strides and one can see that $\mathcal{P}$, just like $\mathcal{T}$, has unit stride. The second blue tensor is the same as $\mathcal{P}$ while showing the new positions of elements $e_{i,j,k}$ coming from $\mathcal{T}$. With this in mind and by looking at Fig. 2c, one can see that the memory layout of $\mathcal{P}$ has changed.

To avoid expensive memory operations, it is possible to simply change the strides and dimension sizes of the dimensions that are permuted. The strides are no longer in descending order, the tensor no longer has unit stride. However, the offset of an element still can be calculated perfectly fine, meaning the values don't have to move in memory and permuting is cheap. Fig. 2 (b) shows this process, again for $\mathcal{T}_{d_0,d_1,d_2} \in \mathbb{R}^{3\times2\times4}$ and $\sigma = (1, 2, 0)$. The dimensions of the pink tensor are simply

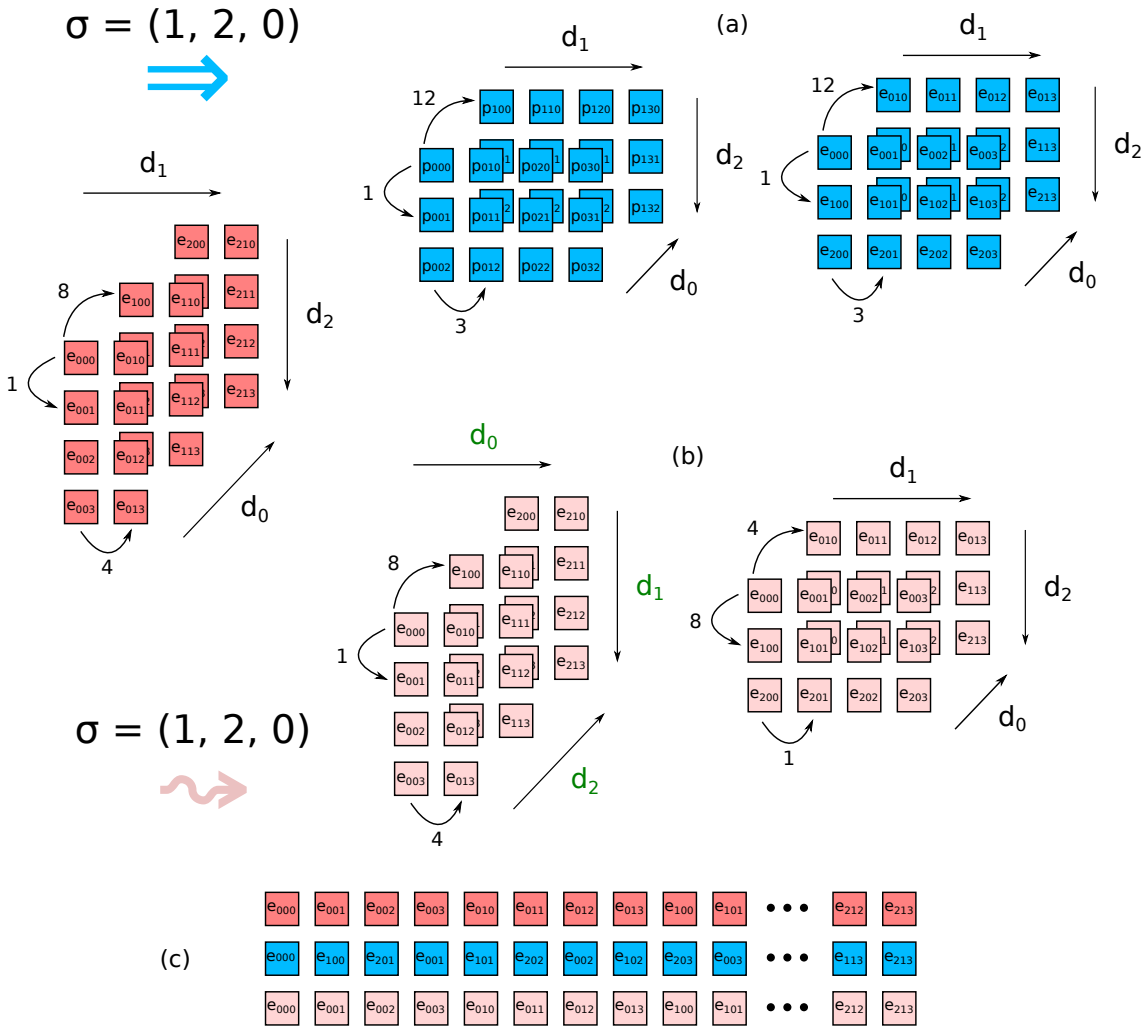


Figure 2: (a) Permutation of a tensor by changing its values in memory. (b) Permutation of a tensor by changing its strides and dimension sizes. (c) Memory layouts for each tensor.

reinterpreted to have different sizes and strides, resulting in no memory operations, as seen in figure 2 (c) I call this type of permutation **soft permutation**.

2.2.2 Reshape

Reshaping means changing the view on a tensor. Its number of elements stays the same, just like the order of its elements in memory. Only the number of dimensions and their sizes possibly change by merging or splitting them. Multiple neighboring dimensions of a tensor can be merged. This means that i adjacent dimensions $d_x, d_{x+1}, \ldots, d_{x+i-1}$ with dimension sizes $|d_x|, |d_{x+1}|, \ldots, |d_{x+i-1}|$ result in one bigger dimension D with dimension size $|D| = \prod_{k=x}^{x+i-1} |d_k|$. Fig. 3 shows how a tensor $\mathcal{T}_{d_0,d_1,d_2} \in \mathbb{R}^{3\times2\times4}$ is reshaped to the yellow tensor $\mathcal{R}_{D_{01},d_2} \in \mathbb{R}^{6\times4}$. The other yellow tensor at the bottom shows the initial position of its elements in $\mathcal{T}$. It is also possible to split one dimension into multiple neighboring dimensions. Splitting dimension D_{01} of $\mathcal{R}$ into two dimensions of sizes $|d_0| = 3$ and $|d_1| = 2$ would result in the tensor

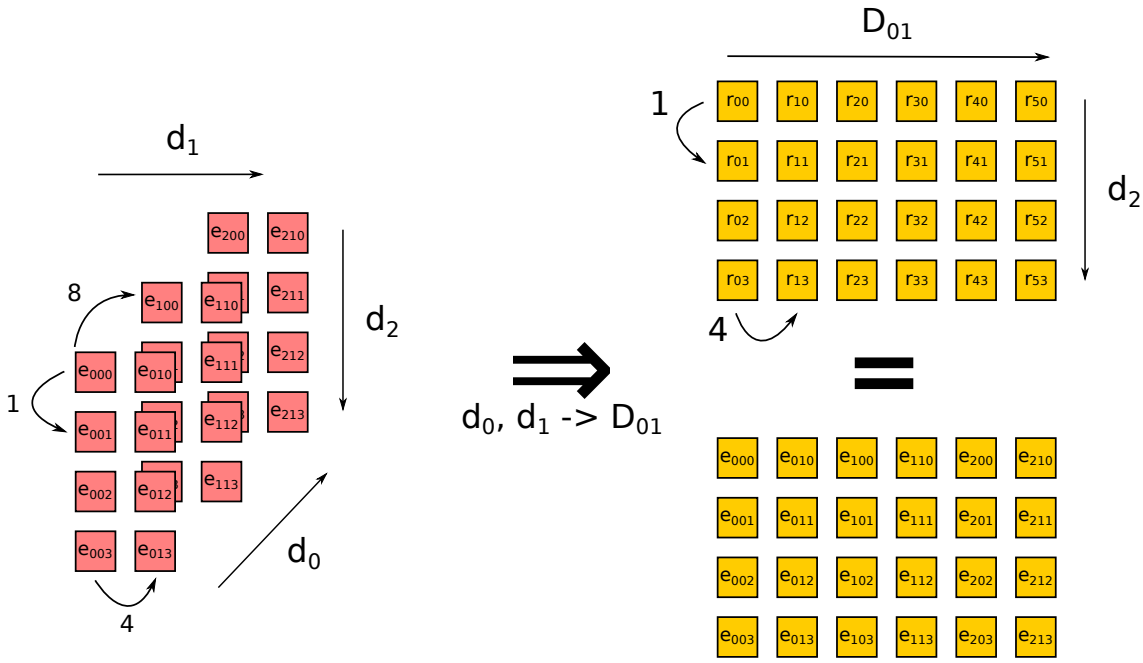


Figure 3: Reshaping a tensor of type 3 to a matrix. The strides are denoted next to the tensors. The bottom matrix shows the origin of the matrix elements.

before the initial merge.

Since reshaping does not touch any of the elements of the tensor in memory, only the dimension sizes and strides have to be changed. This possibly changes if the tensor has been soft-permuted before. The tensor has no unit stride anymore. This means that some neighboring dimensions d_0 and d_1 don't have neighboring strides anymore, so $str(d_0) \neq str(d_1) \cdot |d_1|$. The values of the tensor now have to be permuted in memory before the reshape operation. Otherwise, the resulting reshaped tensor would not have "working" strides, meaning they cannot be expressed with a single natural number. Fig. 4 shows the tensor $\mathcal{P}$ from Fig. 2b that has been soft-permuted. If dimensions d_1 and d_2 are now merged to D_{12} in $\mathcal{R}$, $\mathcal{R}$ will have no valid stride for D_{12}. The merged dimension is first increased with stride 8 two times. After that, the original dimension d_2 has reached its virtual maximum value of 3, and has to be set to zero in the next increment step. This results in a partial stride of $-2 \cdot 8$. Simultaneously, d_1 is increased by one with a stride of 4, resulting in a total memory jump of -12. Since both memory jump sizes 8 and -12 have occurred by increasing the same dimension D_{12}, no valid stride exists for D_{12} and $\mathcal{P}$ has to be permuted in memory before the reshape operation to achieve unit stride.

2.2.3 Summing dimensions

A dimension of a tensor can be summed and thus reducing the number of dimensions and elements of the tensor. Multiple dimensions can be summed at once. Summing all dimensions results in a scalar value that is equal to the sum of all elements in a tensor. Let's consider a tensor $\mathcal{T}_{d_0,d_1,\ldots,d_{n+s-1}}$ with $n+s$ dimensions where s of

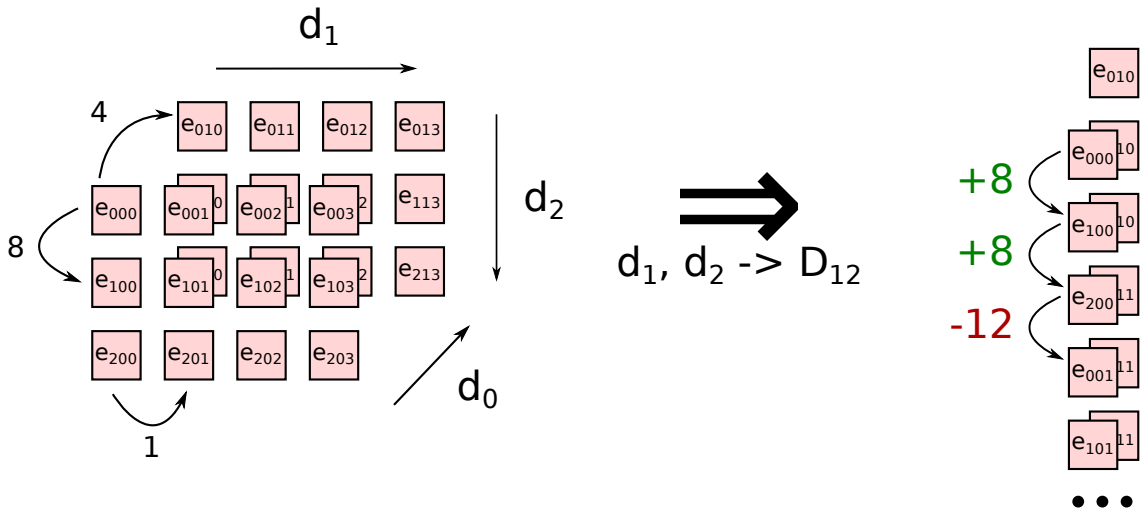


Figure 4: Reshape after soft permutation resulting in non-working strides.

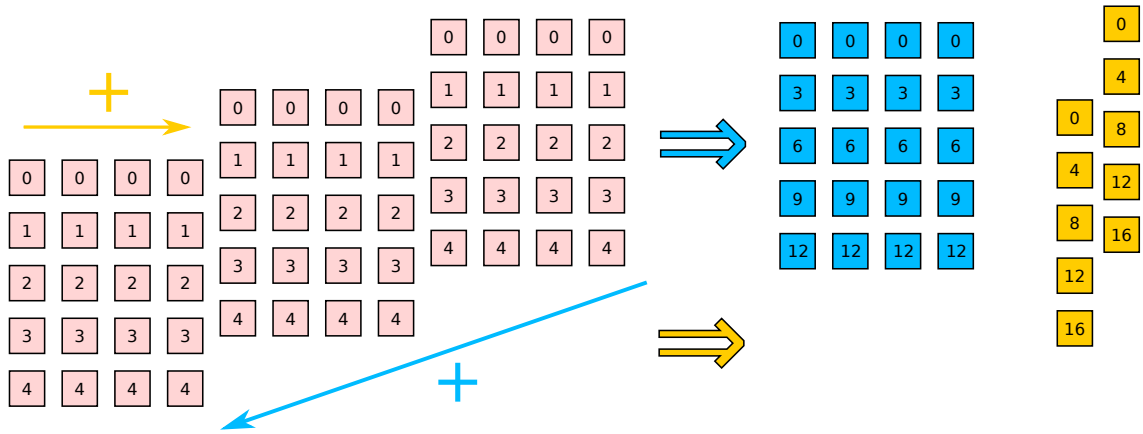


Figure 5: Summing dimension of a tensor. The blue tensor is the result of summing the blue dimension. The yellow tensor is the result of summing the yellow dimension.

them should be summed. The resulting tensor can be denoted as $\mathcal{S}_{d_0,d_1,\ldots,d_{n-1}} = \mathcal{T}_{d_0,\ldots,a_0^+,\ldots a_1^+,\ldots,a_{s-1}^+,d_{n-1}}$, where $a_0^+, a_1^+, \ldots, a_{s-1}^+$ are the summed dimensions. A single element of $\mathcal{S}$ can be calculated using the equation

$$[\mathcal{S}](d_0, d_1, \ldots, d_{n-1}) = \sum_{a_0,a_1,\ldots,a_{s-1}} [\mathcal{T}](d_0, \ldots, a_0, \ldots a_1, \ldots, a_{s-1}, d_{n-1}). \tag{3}$$

Fig. 5 shows a tensor $\mathcal{T}$ of type 3 and two different tensors of type 2, where a dimension of $\mathcal{T}$ has been summed. The blue tensor $\mathcal{B}$ is the result of summing the blue dimension, and the yellow tensor $\mathcal{Y}$ of summing the yellow dimension. An element of $\mathcal{B}$ is calculated by summing the respective elements of each matrix slice. $\mathcal{Y}$ keeps the slices but sums the columns.

2.2.4 Tensor contraction

A tensor contraction is an operation on two tensors resulting in one contracted tensor. I will be calling the two tensors that are contracted **input tensors** and the

contracted tensor **output tensor** or **result tensor**. The contraction can be seen as a product between elements of the two tensors while simultaneously summing dimensions that do not appear in the result. The dimensions that are summed are the so-called **contraction dimensions** and they appear in both input tensors. To contract two tensors $\mathcal{A}$ and $\mathcal{B}$, one first has to identify the dimensions that should be contracted over. The resulting tensor (output tensor) $\mathcal{R}$ holds all the dimensions of the two input tensors that are not contracted over, meaning it has anything between one and $|D_{\mathcal{A}}| + |D_{\mathcal{B}}|$ dimensions. If the $\mathcal{R}$ has exactly $|D_{\mathcal{A}}| + |D_{\mathcal{B}}|$ dimensions, meaning there are no contraction dimensions, then $\mathcal{R}$ is called an **outer product** tensor of $\mathcal{A}$ and $\mathcal{B}$. Let's denote $c_0, c_1, \ldots$ as the dimensions that are contracted and $a_0, a_1, \ldots$, $b_0, b_1, \ldots$ as the dimensions that are not contracted, appearing in $\mathcal{A}$ and $\mathcal{B}$, respectively. $\mathcal{R}$ will then hold the dimensions $a_0, a_1, \ldots, b_0, b_1, \ldots$. An element $[\mathcal{R}](a_0, a_1, \ldots, b_0, b_1, \ldots)$ is calculated by summing the products of all two-element combinations, dependent on $c_0, c_1, \ldots$, where the two elements are $[\mathcal{A}]\{a_0, a_1, \ldots, c_0, c_1, \ldots\}$ and $[\mathcal{B}]\{b_0, b_1, \ldots, c_0, c_1, \ldots\}$. The contraction can be described with the equation

$$[\mathcal{R}](a_0, a_1, \ldots, b_0, b_1, \ldots) = \sum_{c_0, c_1, \ldots} [\mathcal{A}]\{a_0, a_1, \ldots, c_0, c_1, \ldots\} \cdot [\mathcal{B}]\{b_0, b_1, \ldots, c_0, c_1, \ldots\} \tag{4}$$

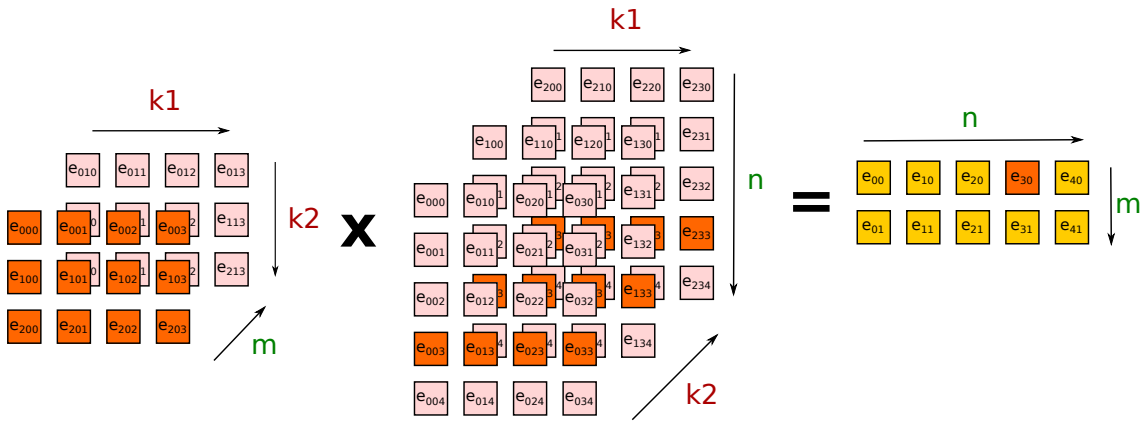
Note that the contraction dimensions have to have the same size in each of the tensors.

One popular tensor contraction is matrix multiplication, where the contraction dimension is the column dimension of the left matrix and the row dimension of the right matrix. The result is again a matrix, keeping the row dimension of the left and the column dimension of the right matrix. Fig. 6 shows a contraction of two tensors of type 3, $\mathcal{A}_{m,k_1,k_2}$ and $\mathcal{B}_{k2,k1,n}$. The contraction dimensions are k_1 and k_2, so that the result tensor $\mathcal{C}_{n,m}$ is a matrix with dimensions n and m. The highlighted elements of $\mathcal{A}$ and $\mathcal{B}$ are the elements involved in the calculation of $[\mathcal{C}](3, 0)$, as seen in the equation at the bottom of the figure.

2.2.5 Tensor network contraction

A tensor network is a network of multiple tensors with a given way to contract this network down to one single tensor. It can be seen as a factorized approximation of a tensor. An example of a tensor network consisting of four tensors is given in Fig. 7 (left). In this graph-like presentation, the nodes are the tensors and the edges are the dimensions of the tensors. Respective dimension sizes are written onto them (a - f). A dimension (edge) connecting two tensors is a contraction dimension between the two, while "free" edges will be kept in the result tensor. If there are f free edges, the result tensor is of type f.

The contraction can be calculated similarly to Eq. (4) by calculating the sum over products between every index combination of contraction dimensions. Let D_i denote the set of dimensions kept in the result tensor coming from tensor $\mathcal{T}^i$ and let C_i be the set of dimensions that $\mathcal{T}^i$ has as a contraction dimension with another tensor.



$$\boxed{c_{3,0} = \sum_{k_1,k_2} a_{0,k_1,k_2} \cdot b_{k_2,k_1,3}}$$

Figure 6: Tensor contraction of two type 3 tensors with two contraction dimensions k_1 and k_2, resulting in a matrix with dimensions n and m. The highlighted elements are involved in calculating the highlighted element of the matrix, as shown in the equation.

Then the network contraction can be calculated with

$$[\mathcal{R}]\{D_0, \ldots, D_n\} = \sum_{\{C_0,\ldots,C_n\}} [\mathcal{T}^0]\{D_0, C_0\} \cdot [\mathcal{T}^1]\{D_1, C_1\} \cdot \ldots \cdot [\mathcal{T}^n]\{D_n, C_n\} \quad (5)$$

where $\{D_i, C_i\}$ is a set of sets but should be seen as a simple set of indices that are in $D_i \cup C_i$.

This way of calculating a tensor network contraction is computationally expensive. A cheaper way of contracting would be to contract two tensors at a time until only one output tensor is left. This approach is shown in Fig. 7, where the red and blue tensors are contracted first, then the red/blue tensor with the yellow one, and last but not least the red/blue/yellow tensor with the green tensor, resulting in one matrix. While a contraction strictly following Eq. (5) would need $\mathcal{O}(a \cdot b \cdot c \cdot d \cdot e \cdot f)$ multiplications, contracting the network in binary fashion in the way shown only needs $\mathcal{O}(acdf + adef + abef)$ operations. Binary contracting a network exploits that dimensions are summed in the process and no longer appear in future binary contractions. Finding an optimal contraction sequence is NP-hard and not in the scope of this thesis, but is still part of active research, alongside finding not the optimal, but a good sequence by using heuristics [17, 28, 22].

2.2.6 Einsum summation convention

The einsum summation convention (einsum) is a powerful way to express operations on (multiple) tensors. A simple einsum is the term `abcd, cdefg -> abefg`. The two parts of the einsum are divided by the "`->`" sign. On the left are sub-strings divided by commas that refer to each input tensor. The right side consists of a single

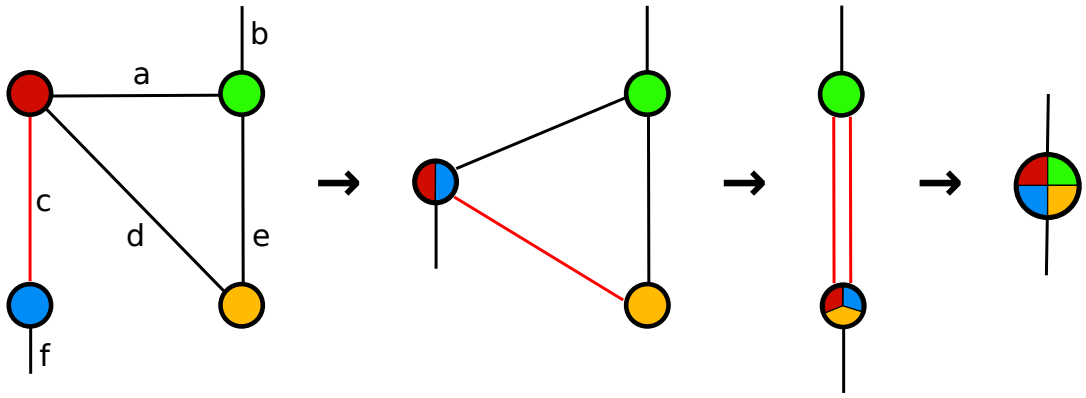


Figure 7: A tensor network in graph notation being contracted with three binary contractions. The nodes represent tensors and the edges are dimensions of size a - e. The resulting tensor is a matrix.

string that represents the output tensor. Each of the sub-strings consists of multiple characters referring to a single dimension of the respective tensor. Changing the arrangement of letters in the output expression means permuting the output tensor. The example einsum represents a binary tensor contraction. The two input tensors both have the contraction dimensions c and d which do not appear in the result tensor. The other dimensions appearing in the left and right input tensor are kept in the output tensor.

Calculating a given einsum is similar to calculating a tensor network contraction. As an example, the einsum expression `abac, ca, def, eg -> gaf` with input tensors $\mathcal{A}_{a_0,a_1,a_2,a_3}$, $\mathcal{B}_{b_0,b_1}$, $\mathcal{C}_{c_0,c_1,c_2}$, $\mathcal{D}_{d_0,d_1}$ and result tensor $\mathcal{R}$ is calculated by

$$[\mathcal{R}](g,a,f) = \sum_{b,c,d,e} [\mathcal{A}](a,b,a,c) \cdot [\mathcal{B}](c,a) \cdot [\mathcal{C}](d,e,f) \cdot [\mathcal{D}](eg). \tag{6}$$

There are a few rules an einsum expression has to follow:

1. Dimensions represented by the same character must have the same size.
2. The number of characters in the string regarding tensor $\mathcal{T}$ must be $|D_{\mathcal{T}}|$.
3. Characters appearing in the output tensor string must appear in at least one of the input strings.
4. Characters in the output tensor string can only appear once.

To get a better understanding of the notation, Table 2 shows some example einsums and the operation that they encode.

Like with the contraction of a tensor network, an einsum can be contracted in binary fashion with a given contraction sequence to reduce the number of needed calculations. Each binary tensor contraction results in a tensor containing all the dimensions that are not contraction dimensions. The dimensions appearing in the result tensor of a binary contraction inside an einsum are determined in a more general way. Here, a dimension is kept if it appears in the final result tensor or in a

Einsum	Description
mk, kn ->mn	matrix multiplication row major
km, nk ->nm	matrix multiplication column major
mn ->nm	matrix transpose
bkm, bnk ->bnm	batched matrix multiplication
ij, ij ->ij	matrix element-wise multiplication
tt ->	trace of matrix
i, j ->ij	vector outer product
i, j ->	vector inner product
abcdef ->befdca	tensor transpose
abcd, cdefg ->abefg	binary tensor contraction
abcd, cdefg, fghij, abexyz ->hijxyz	tensor network contraction
abcdef ->abc	summing dimensions d, e, f

Table 2: Different einsum strings with descriptions. Partially taken from [27].

tensor that is still to be contracted, meaning that dimension is "still needed". The tensor contractions follow the same approach, a non-contraction dimension is either contracted over in a later stage or appears in the output tensor of the network.

Moreover, restrictions of binary tensor contractions in comparison to contractions of einsums are:

1. There is no dimension identifier appearing in both of the input tensors and the output tensor.
2. There is no dimension identifier appearing multiple times in one tensor.
3. The output tensor consists of all the dimensions that appear only in one of the two input tensors.
4. The order of identifiers in the output tensor is given by the order in which they appear in the two input tensors.

3 Calculating binary einsum contractions

This section gives an introduction to popular approaches to compute binary tensor contractions. The aforementioned restrictions of binary tensor contraction in comparison to their einsum counterpart are then lifted. Therefore, binary einsum contractions and with this a complete einsum expression can be calculated.

One could simply follow equation (4) to correctly compute a binary tensor contraction. While the algorithm seems quite simple, the attained performance of this approach is not very high. High-performance implementations of critical building blocks for popular algorithms are using hand-crafted assembly kernels. Matrix multiplication is one of these building blocks, being used in a wide variety of applications. Given the similarity between matrices and tensors, a way to improve the implementation of tensor contractions is by using matrix multiplications. There exist multiple approaches to achieve this. I will be presenting two of them in this

section that have been used in research before as a baseline [19, 26]. First, I will give a short introduction to LIBXSMM, the matrix multiplication backbone of my implementation.

3.1 LIBXSMM

LIBXSMM [13] is a library targeting small matrix multiplications. It serves as a just-in-time code generator by compiling matrix multiplication kernels. The kernel is architecture-specific, harnessing the power of supported vector operations provided by the hardware. **Just-in-time** (JIT) compilation means generating machine code during runtime. The subroutine generating the code returns a pointer to the matrix multiplication program that the main program can then jump to. While this procedure generates an overhead because of the compilation process, the kernel can be reused multiple times. Given the potential performance increase of the matrix multiplication, this overhead is getting less important with every reuse.

The LIBXSMM function generating the general matrix multiplication (GEMM) kernel needs to be provided with m, n, and k parameters that encode the shapes of the matrices. m is equal to the number of rows of the first matrix, n is equal to the number of columns of the second matrix and k encodes the number of columns/rows in the first/second matrix. Both matrices can be optionally transposed. Furthermore, both matrices have to have one dimension of stride 1, the other dimension can have any stride. The stride of one dimension of the output tensor can also be chosen freely. Since a LIBXSMM GEMM kernel needs a stride one dimension to operate, it is possible that elements that should be part of a GEMM need to be packed first. In this thesis, I will refer to **packing** as a process that copies values of a tensor into contiguous memory. **Unpacking**, on the other hand, means writing elements of a matrix to respective memory locations in a result tensor. Packing and unpacking are used in high-performance matrix multiplication implementations [30, 14, 10].
A generated kernel consists of multiple SIMD (single instruction multiple data) operations being performed on matrix elements. At the beginning and the end, stack operations are performed to follow calling conventions. These operations may introduce an additional overhead, especially if the shapes of the matrices are small.

3.2 Calculating contractions with matrix multiplications

3.2.1 The TTGT approach

One approach to calculating a tensor contraction by using matrix multiplications is the transpose-transpose-GEMM-transpose (TTGT) approach. To contract tensors using TTGT, one has to permute them in a way so that they can be reshaped to a matrix. Let's again look at the einsum expression for column-major matrix multiplication: `km, nk -> nm` encodes the contraction dimension k and the dimensions kept in the resulting matrix m and n. Two tensors $\mathcal{A}_{m_0,k_0,m_1,k_1,m_2}$ and $\mathcal{B}_{k_1,n_0,k_0,n_1}$ shall now be contracted by using the TTGT approach. The contraction dimensions are denoted as k_i while the non-contraction dimensions in the left tensor are named m_i and in the right tensor n_i. A scheme applicable to these types of tensor con-

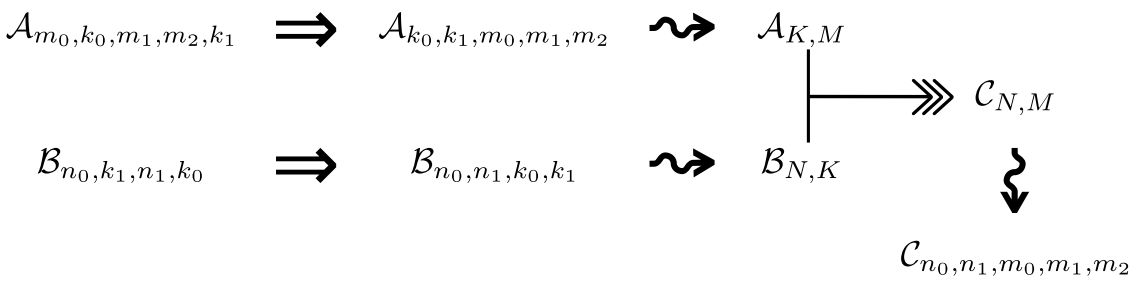


Figure 8: The TTGT approach with permutation(⇒), reshaping (⇝) and matrix multiplication steps (>>>).

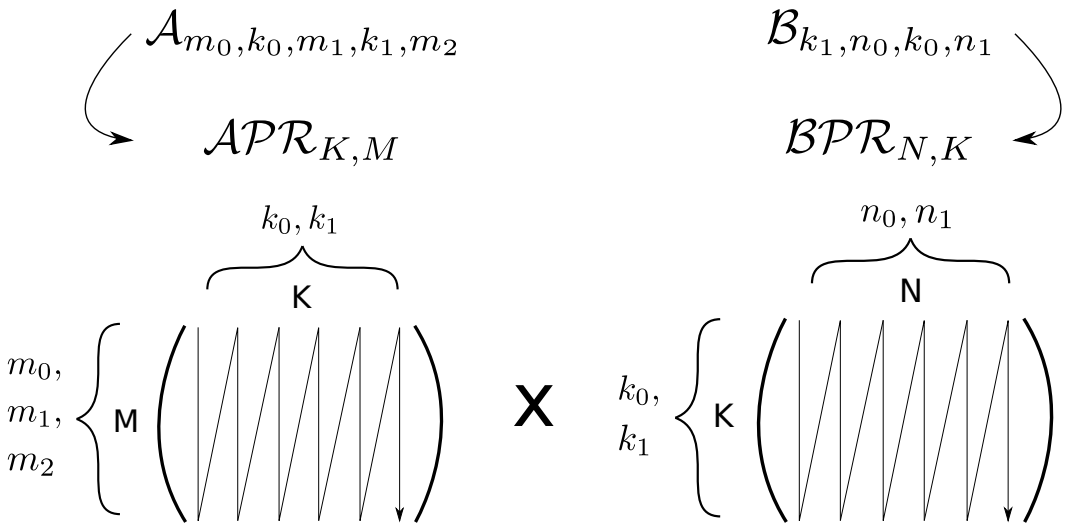


Figure 9: Memory layouts of the matrices $\mathcal{APR}_{K,M}$ and $\mathcal{BPR}_{N,K}$.

tractions is to first permute the tensors in memory, so soft permuting them is not sufficient (reshape after permute, see Fig. 4). In our example, the permuted tensors would have dimension order $\mathcal{AP}_{k_0,k_1,m_0,m_1,m_2}$ and $\mathcal{BP}_{n_0,n_1,k_0,k_1}$. The connection to a matrix multiplication can now be seen by reshaping the two tensors to $\mathcal{APR}_{K,M}$ and $\mathcal{BPR}_{N,K}$. Calculating the output matrix $\mathcal{CM}_{N,M}$ and reshaping it to $\mathcal{C}_{n_0,n_1,m_0,m_1,m_2}$ by splitting up the dimensions N and M will result in the correct result tensor of the tensor contraction. The pipeline of the TTGT approach is shown in Fig. 8 and the new matrix-like memory layout for both $\mathcal{A}$ and $\mathcal{B}$ in Fig. 9.

One upside of TTGT is the attained floating point operations per second (FLOPS) during the matrix multiplication. In contrast to LAGEMM described below, the resulting matrices of TTGT are large and the GEMM subroutine can be applied without any dimension search problems with decently high FLOPS. The big downside of this approach is the permutation step which needs to be performed in memory. Essentially, both input tensors have to be copied which results in a large memory overhead most of the time, especially for smaller tensor contractions. Since the amount of needed memory operations equals $|M| \cdot |K| + |N| \cdot |K|$ and the amount of needed floating point operations is $2 \cdot |M| \cdot |K| \cdot |N|$, very large tensor contractions will not be affected as much by the copy operations. For every other tensor, finding a way to minimize the needed permutations is crucial to increase the performance

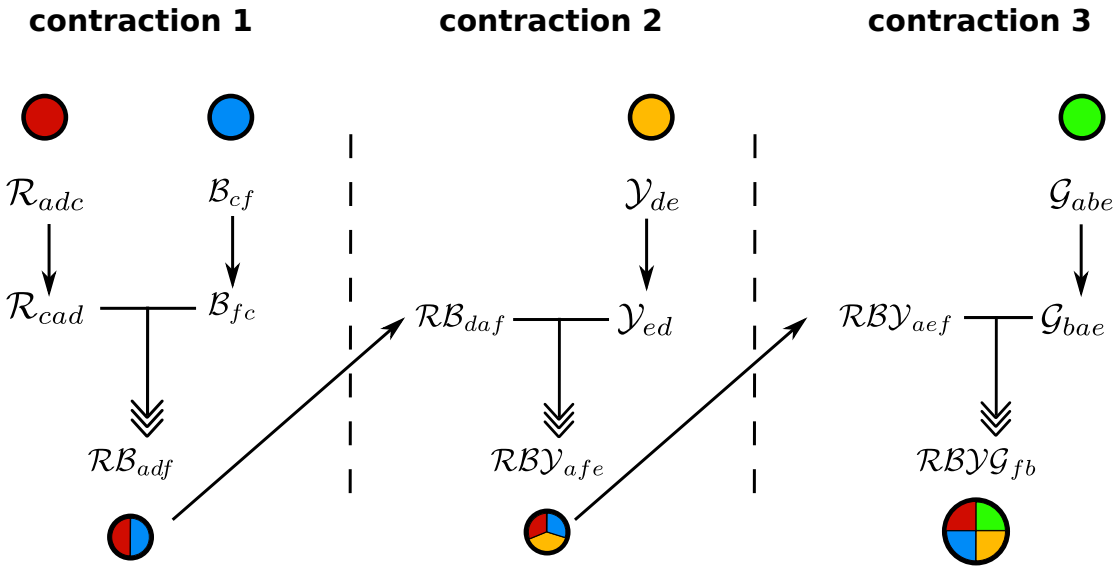


Figure 10: Permutation operations (→) and matrix multiplications (>>>) needed for contracting the tensor network in Figure 7 using TTGT.

of tensor (network) contractions. D.A. Mathews [19] has done some key research in this regard. Fig. 10 shows all the needed permutations (→) and matrix multiplications (>>>) for contracting the tensor network from Fig. 7 using a naive TTGT implementation. One can see that, again, the input tensors of the contraction need to be permuted. The resulting tensor itself is possibly an input tensor at a later stage of contracting the network and needs to be permuted too.

3.2.2 Looping around matrix multiplications (LAGEMM)

Another approach to calculating a tensor contraction by using matrix multiplications is to identify dimensions in the two input tensors that can represent a GEMM and use the other dimensions to loop around it. To give a better perspective, let's look at the two tensors $\mathcal{A}_{a,b,c,d}$ and $\mathcal{B}_{f,b,g,c}$ that are contracted to $\mathcal{C}_{a,d,f,g}$. The Einstein notation for this contraction is `abcd, fbgc -> adfg`. A column major matrix multiplication can be achieved by using the dimensions d, c, g, and use a, b, f to loop around the GEMM represented by `cd, gc -> gd`. Note that all of these matrices are contiguous in memory which is not always the case. The indices regarding dimensions a, b, f provide an offset in $\mathcal{A}$ and $\mathcal{B}$ on where to find respective matrices. The offsets are calculated by simply using indices and strides as described in Eq. (2). In the end, there will be $|a| \cdot |b| \cdot |f|$ matrix multiplications, and the elements are written to their right location in $\mathcal{C}$ during the unpacking routine. However, in this example $\mathcal{A}$ and $\mathcal{B}$ have a good order of dimensions so that matrix multiplication can easily be applied. Depending on if the matrix multiplication subroutine (kernel) allows for transposing matrices or matrices with dimensions with arbitrary strides, this approach will work in more or less cases without permuting the tensors first. For example, if dimensions f and g were to switch positions in $\mathcal{B}$, the einsum would be `abcd, gbfc -> adfg`. The matrix multiplication `cd, gc -> gd` could then still be achieved by simply telling the subroutine the stride of g in $\mathcal{B}$. Given the einsum

`abcd, fbcg -> adfg` where the positions of g and c are switched, the column-major matrix multiplication can still be performed by transposing the second matrix. If the subroutine always needs a dimension of stride 1, then the einsum `xabcd, ycdab -> xy` cannot be computed using this approach without permuting the tensors first. Here, the contraction dimensions are a, b, c, d. There is no way of having only one and the same contraction dimension present in the matrix multiplication.

The upsides of the approach are that tensors do not have to be permuted in all cases (or never if the kernel supports arbitrary strides and transposition). However, finding dimensions of a good size that the GEMM can be performed on is not always a given. If the dimensions chosen are very small, the overhead of calling the kernel may be larger than the computation time needed. Some kernels only operate on larger matrices, so small ones have to be padded with zeros, wasting computation capabilities. Also, kernels are (way) slower operating on matrices with bigger or arbitrary strides than those that are given matrices contiguous in memory, further limiting the choice of dimensions or even the possibility to use LAGEMM. Furthermore, looping around the GEMM in a cache-friendly way is not as easy as with the TTGT approach. It is favorable to have looping dimensions with small strides so that memory pages containing elements of multiple matrices are already in the cache. However, contraction dimensions are present in both tensors with possibly different strides. If they are not part of the GEMM and are used to loop around it, it is not possible to guarantee a good memory access pattern in both tensors. As an example, consider the tensors $\mathcal{A}_{m_0,m_1,k_0,k_1,m_2}$ and $\mathcal{B}_{k_0,n_0,n_1,n_2,k_1}$ that can be contracted using LAGEMM with the GEMM represented by $\mathtt{k}_1\mathtt{m}_2$, $\mathtt{n}_2\mathtt{k}_1$ `->` $\mathtt{n}_2\mathtt{m}_2$. Looping with k_0 results in a small stride in $\mathcal{A}$ but a large stride in $\mathcal{B}$.

3.2.3 Einsum via matrix multiplication

Given that tensor contractions are possible to calculate using GEMMs, by lifting the constraints 1. - 4. described in Sec. 2.2.6, it is also possible to calculate any einsum expression consisting of more than one input tensor with matrix multiplications.

1. There can be dimension identifiers appearing in both input tensors and the output tensor: These dimensions are called batch dimensions and can be interpreted as a loop around the complete GEMM.

2. Dimension appearing multiple times in one input tensor: A dimension identifier appearing multiple times in one tensor increases the stride of this dimension in the tensor. This also means that some of its elements are disregarded. Elements can be accessed by using the larger stride or a copy of the tensor can be made, where the identifier only appears once and where all elements are used in the contraction.

3. The output tensor can hold more or fewer dimension identifiers than those only appearing in one of the two input tensors: If the output tensor consists of more dimensions, then those dimensions are batch dimensions (see 1.). If it has fewer dimensions, then the respective dimensions of the input tensors can be summed before contraction.

4. There is no fixed arrangement of dimensions in the output tensor: The output tensor can be permuted and thus achieve any ordering of its dimensions.

4 Related work

There has been a vast amount of research in the last decade on the implementation of tensor operations. Whether it being about the use of tensors in specific domains [6, 21, 9], building on top of the TTGT or LAGEMM routines [19, 26], or general tensor libraries being able to target arbitrary tensor layouts during runtime. Popular choices for the latter with Python interfaces are NumPy [12], TensorFlow [18] and PyTorch [20] which also provides a C++ interface. There are libraries for finding the optimal combination of different approaches [26] or focussing on ease of usability by providing interfaces on tensor diagrams [7]. opt_einsum [25] provides a Python interface to find a good or even optimal contraction sequence for a given einsum expression.

High-performance matrix multiplication is the backbone of tensor calculations and has been researched over multiple decades and implemented in a variety of libraries targeting basic linear algebra operations. To name a few, the basic linear algebra subroutines (BLAS) [16, 5, 4] have been around since the 1970s. Intel built their own routines named Intel MKL [15]. BLIS [30] is a newer library improving BLAS and giving it more flexibility. Here, two (large) matrices are recursively blocked by packing sub-matrices for the various cache levels. The packing for different cache levels was shown to be not affecting the performance too much [10]. The computation happens by essentially looping around an optimized assembly kernel for a given architecture. This implementation achieves almost peak floating point performance on a given CPU.

Matthews showed a way to contract two tensors without permuting them first, while still using high-performance matrix multiplication kernels [19]. In his research, he interprets the two tensors as matrices that are later multiplied. Because there are no permutation operations first, the matrices are not contiguous in memory. They are in a "scatter-matrix-layout" which means that the memory strides for going one element further to the next row/column differ. Matthews gives multiple options on how to handle this layout. The first option is to pack (copy) the currently needed elements of the matrix in the scatter layout into contiguous memory. However, this is not always needed, since some parts of the scatter matrix still have strides that a GEMM using a BLIS kernel [30] can be performed on. By exploiting this and the fact that permutations are not needed, this approach results in almost the same performance as standard matrix multiplications where the matrices are of the same size as the tensors. This means that the matrix multiplication they compare themselves to is essentially the calculation step of the TTGT approach, where the permutation has been performed already.

Springer and Bientinesi [26] had the same goal as Matthews; contracting two tensors without any costly permutation operations. Their approach is to find the best contraction candidate between TTGT, LAGEMM (or loop over GEMM in their

case), or their newly introduced GETT approach. GETT stands for "GEMM-like tensor-tensor multiplication". Essentially, it searches potential dimension reshape operations to then perform a GEMM on a sub-tensor. The reshapes consider the size of the result dimensions and the various cache-level sizes of the machine. The sub-tensors are chosen so that they fit into the cache levels, similar to blocking the matrices in the BLIS matrix multiplication routine.
Multiple GETT candidates are then timed, and the best one is chosen to be compared to TTGT and LAGEMM candidates. It was also shown that there is no need to time more than 16 GETT candidates since timing more does not result in further noticeable performance benefits. However, the presented work is only comparable to the work from Matthews in the sense that it contracts tensors in a high-performance fashion. Former tries to find a way to generate contraction code by first simulating multiple contraction approaches. The latter uses its presented approach at all times without any searching process.

I will be comparing my work to the einsum implementation of PyTorch [20] in section 6 of this thesis. PyTorch implements the vanilla TTGT approach for einsum expressions. It permutes tensors that should be contracted to two matrices and uses a batch matrix multiplication subroutine to do the calculation. The order of dimensions in the binary result tensor is then given by the order of dimensions in the input tensors. If the result tensor needs to be contracted itself, it has to be permuted too. There is no use of the knowledge about future contractions the result tensor has to undergo and its potential memory layout.

My work differs from the three examples described above. I do not only consider binary tensor contractions but look at the complete contraction sequence. By doing this, I try to exploit knowledge about future contractions tensors have to undergo. There has been research and implementations regarding tensor networks and their contraction routines [24], but to my knowledge no research about considering contractions of tensor networks or einsum expressions by having future contractions in mind.

5 Memory layouts for intermediate tensors

This section describes a way to analyze an einsum expression and its sequence of binary contractions to be able to optimize the LAGEMM approach. By identifying input, output and *intermediate* tensors, one can find a memory layout for intermediate tensors that assists a fast contraction routine. To achieve this, good memory layouts and an unpacking process that writes the calculated parts of the kernel output to the right locations in memory are needed.

5.1 Intermediate tensors

Intermediate tensors are tensors that result from a binary contraction in the contraction sequence of an einsum expression and are contracted themselves at a later stage. This means that they are neither input tensors provided up front, nor the result tensor at the end of the contraction sequence, since the last result tensor is

not contracted again. The result tensor of the einsum has to have a specific order of dimensions. The input tensors are provided with a specific memory layout too. Intermediate tensors, however, are used to save intermediate results and are generally of no importance after the contraction sequence has been processed. Because of this, they can have any memory layout, as long as the contraction routine can process it. A more sophisticated unpacking routine can write result matrices of the GEMM kernel back to any memory location and thus providing the desired memory layout for the intermediate tensors.

Reducing memory operations By immediately writing to any memory location during the unpacking process, one essentially gets a "free" permutation operation for any intermediate tensor. Let's assume an intermediate tensor $\mathcal{I}$ is at some point a result tensor of a binary contraction of the tensors $\mathcal{A}$ and $\mathcal{B}$. Since this is done with the help of GEMMs, $\mathcal{A}$ and $\mathcal{B}$ are represented as matrices. During this, $\mathcal{A}$ and $\mathcal{B}$ are blocked and sub-blocks of $\mathcal{I}$ are calculated which have to be written to the right position in memory. These memory operations can be assumed to operate on main memory if $|\mathcal{I}|$ is large enough. The total number of memory operations is then $|\mathcal{I}|$. $\mathcal{I}$ is possibly contracted itself at a later stage. Assuming one wants to permute $\mathcal{I}$ again before contracting it to achieve a new memory layout, it will take another $|\mathcal{I}|$ memory operations. In total, $2|\mathcal{I}|$ memory operations are performed because of the intermediate tensor $\mathcal{I}$. By knowing about the future memory layout that $\mathcal{I}$ should be in, it is possible to merge both sets of memory operations into one, which is performed during the unpacking process. The total amount of memory operations would then only be $|\mathcal{I}|$. An idea that is not further investigated in this thesis of how to utilize this for the TTGT approach is shown in Fig. 11. Here, the same contraction is performed as in Fig. 10. However, the intermediate tensors created from contractions 1 and 2 are not stored in a memory layout that the standard TTGT approach would normally have. Instead, the intermediate tensors $\mathcal{RB}$ and $\mathcal{RBY}$ are stored to be in the correct matrix layout that they need to be in for their own contraction step.

5.2 Intermediate tensors during LAGEMM

With the described possibility to achieve any memory layout for intermediate tensors by gaining a free memory permutation, most of the drawbacks of the LAGEMM approach can be eliminated. To do so, one can find a beneficial memory layout so that

1. a matrix multiplication using LIBXSMM is possible.
2. the GEMM is calculated on matrices that are contiguous in memory, guaranteeing good memory access patterns.
3. dimensions not part of the GEMM can be positioned so that looping around the GEMM is done in a cache-friendlier way.
4. the matrices part of the GEMM can have any size, meaning the kernel can have any shape. This ensures that the kernel performs well.

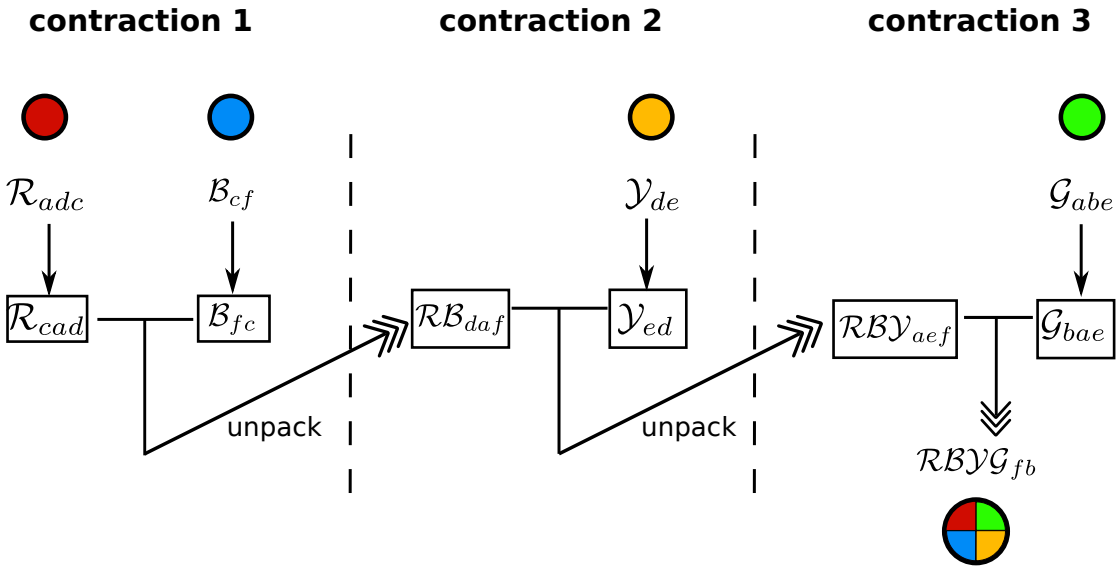


Figure 11: Contraction of a tensor network using TTGT while intermediate tensors are in a correct memory layout for their contraction.

Finding a memory layout supporting characteristics 1. - 4. will improve the stability of the LAGEMM approach, since performance is no longer dependent on possibly poor memory access patterns or tensor layouts.

Identifying different dimension types To find a beneficial memory layout for two intermediate tensors $\mathcal{I}$ and $\mathcal{J}$, one has to know the dimensions that are part of the contraction the tensors have to undergo. This includes dimensions of the result tensor $\mathcal{C}$ and both input tensors $\mathcal{I}$ and $\mathcal{J}$. This is because different types of dimensions have to be identified to then be able to specify the memory layout attaining characteristics 1. - 4. For a binary contraction between tensors $\mathcal{I}$ and $\mathcal{J}$ resulting in tensor $\mathcal{C}$, the possible types of dimensions are:

(a) Dimensions appearing in $\mathcal{I}$ and $\mathcal{J}$, but not in $\mathcal{C}$ (K dimensions).

(b) Dimensions appearing in $\mathcal{I}$ and $\mathcal{C}$, but not in $\mathcal{J}$ (M dimensions).

(c) Dimensions appearing in $\mathcal{J}$ and $\mathcal{C}$, but not in $\mathcal{I}$ (N dimensions).

(d) Dimensions appearing in $\mathcal{I}$, $\mathcal{J}$ and $\mathcal{C}$ (B or batch dimensions).

(e) Dimensions appearing only in $\mathcal{I}$.

(f) Dimensions appearing only in $\mathcal{J}$.

Note that dimensions appearing only in C are not possible under the constraints of a valid einsum expression. For a tensor contraction, only options (a) - (c) will appear. Options (e) and (f) are of no concern because those dimensions can be summed. They are not in the result tensor $\mathcal{C}$, so they are not important for future contractions or the final output tensor. They are not contracted over, so they would only increase the computational load by a factor of their size. Dimensions that can be summed will never appear in an intermediate tensor because they would have

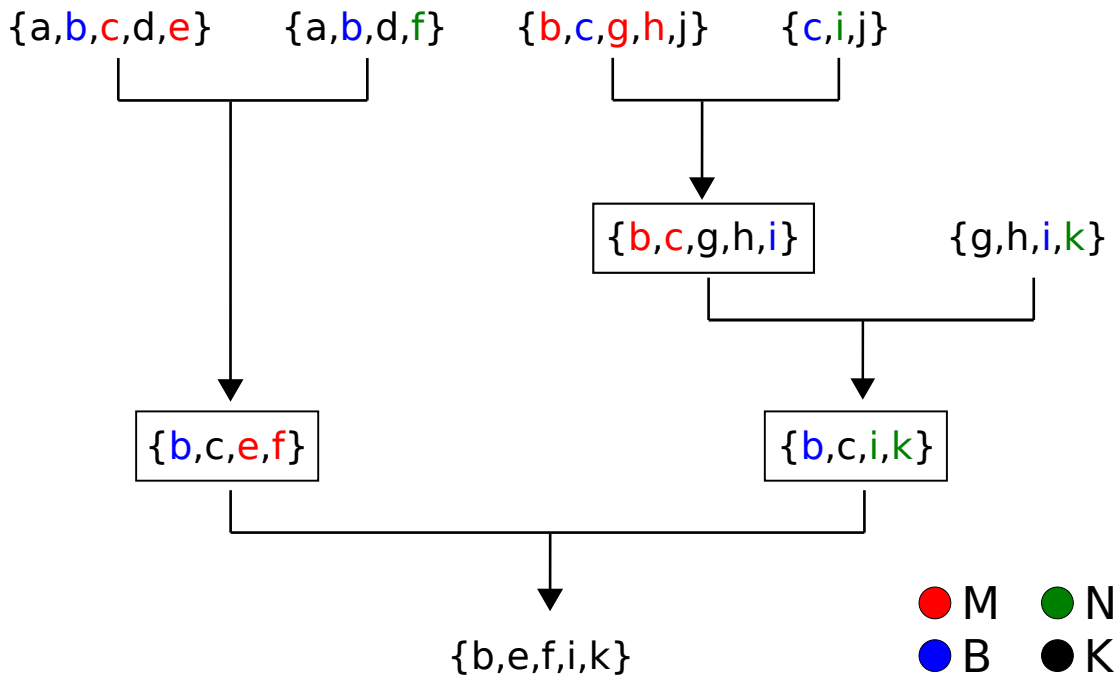


Figure 12: Sets of dimensions for a given contraction sequence. Sets of intermediate tensors have boxes around them. The sets hold respective batch dimensions (blue), contraction dimensions (black), M dimensions (red) and N dimensions (green).

been already summed in one of the input tensors of the einsum. $|K|, |M|, |N|, |B|$ denote the products of dimension sizes for dimensions that are part of K, M, N, B. So if there are three K dimensions k_0, k_1 and k_2, then $|K| = |k_0||k_1||k_2|$.
Fig. 12 shows how dimensions are categorized in a contraction of an einsum expression. The sets of dimension identifiers for respective tensors are colored to distinguish between the different types of dimensions. Blue is a batch dimension, black is a contraction dimension (K), red is an M dimension and green is an N dimension. M dimensions only appear in the left input sets and green ones only appear in the right ones. Note that a dimension of some type in one of the input tensors is not necessarily of the same type in the output tensor $\mathcal{C}$ of the binary contraction. The category of a dimension $d \in D_{\mathcal{C}}$ is chosen w.r.t. the contraction where $\mathcal{C}$ serves as an input tensor.

It is possible to color the complete contraction graph at the start. Given a contraction sequence, the sets of dimensions for all the tensors (including intermediate tensors) are known. This can be done by going through the contraction sequence and looking at future contractions. If a dimension in one of the input tensors for a binary contraction is still needed in a later binary contraction, it will be in the respective binary output tensor. It is clear which tensors are contracted together. With index sets for both input tensors and the output tensor, the types of dimensions can be set. Note that the types are unambiguous because every tensor is only contracted once.

5.2.1 Finding a memory layout

With the types of dimensions for a binary contraction, it is now possible to describe a tensor layout for intermediate tensors that will be contracted using LAGEMM. For now, assume that both input tensors of a binary contraction $\mathcal{I}$ and $\mathcal{J}$ are intermediate tensors and their memory layout can be chosen freely. With the categorization of dimensions in $D_{\mathcal{I}}$ and $D_{\mathcal{J}}$, it is now possible to further specify a layout that has the characteristics 1. - 4. described above.

Characteristics 1. and 2. are easy to achieve by identifying dimensions in $\mathcal{I}$ and $\mathcal{J}$ that a matrix multiplication can be performed on. It is sufficient to identify one M dimension m, one N dimension n and one K dimension k, and arrange them accordingly for a column-major GEMM. The last two dimensions with the smallest strides in $\mathcal{I}_{\ldots,k,m}$ should be k and m, or n and k in $\mathcal{J}_{\ldots,n,k}$. If there are no M and/or N dimensions, a GEMM is still possible by interpreting one/both of the matrices as a vector. The same applies if there are no K dimensions. Then, an outer product can be calculated between two vectors.

Characteristic 3. demands thinking about in what order dimensions that are not part of the GEMM are used to loop around the GEMM. As an example, consider the two index sets $D_{\mathcal{I}} = \{b_0, m_0, m_1, m_2, k_0, k_1, k_2\}$ and $D_{\mathcal{J}} = \{b_0, n_0, n_1, n_2, k_0, k_1, k_2\}$, where b_0 is a batch dimension and m_i, n_i, k_i are M, N and K dimensions, respectively. The GEMM's dimensions can be chosen to be m_0, n_0, k_0, leaving $b_0, m_1, m_2, n_1, n_2, k_1$, k_2 for looping around the GEMM. Note that this selection is not the only option since there are multiple M, N and K dimensions. The question is which dimension index should be increased for looping first, which one afterward, and so on. To find a loop order heuristic, consider that all K dimensions are not part of the result tensor $\mathcal{C}$. Changing their indices will not change the memory position that elements of the result matrices are written to. By increasing the K dimensions first, respective elements of $\mathcal{C}$ can be kept in registers. They need to be written to memory once after the K-loops are finished. Choosing the loop order to start with the K dimensions means that those dimensions should have the smallest possible strides in both $\mathcal{I}$ and $\mathcal{J}$. This ensures good memory access because input matrices along the K dimension are stored one after another. The known memory layouts can be expanded to $\mathcal{I}_{\ldots,k_1,k_2,k_0,m_0}$ and $\mathcal{J}_{\ldots,k_1,k_2,n_0,k_0}$. While k_1 and k_2 must have the same order in $\mathcal{I}$ and $\mathcal{J}$, there is still a degree of freedom by choosing the order of k_1 and k_2. The batch and M/N dimensions are still left to be positioned in $\mathcal{I}$ and $\mathcal{J}$. A batch dimension is present in all the three tensors $\mathcal{I}$, $\mathcal{J}$ and $\mathcal{C}$. Increasing its index results in memory jumps in all of them. Batch dimensions should be looped over last, which means that the final memory layouts are $\mathcal{I}_{b_0,m_1,m_2,k_1,k_2,k_0,m_0}$ and $\mathcal{J}_{b_0,n_1,n_2,k_1,k_2,n_0,k_0}$. Again, the order of m_1, m_2 and n_1, n_2 are not fixed. The resulting memory layout for $\mathcal{I}$ can be seen in Fig. 13a and for $\mathcal{J}$ in Fig. 13b, where column-major matrices are stored along the K dimension first, then M dimensions, then batch dimensions.

Shape of matrix multiplications Choosing the dimensions that the GEMM should be performed on is not a trivial task. As described earlier, the routine that generates LIBXSMM matrix kernels is provided with matrix shapes that encode the sizes of the matrices that are multiplied. Looping around GEMMs of different

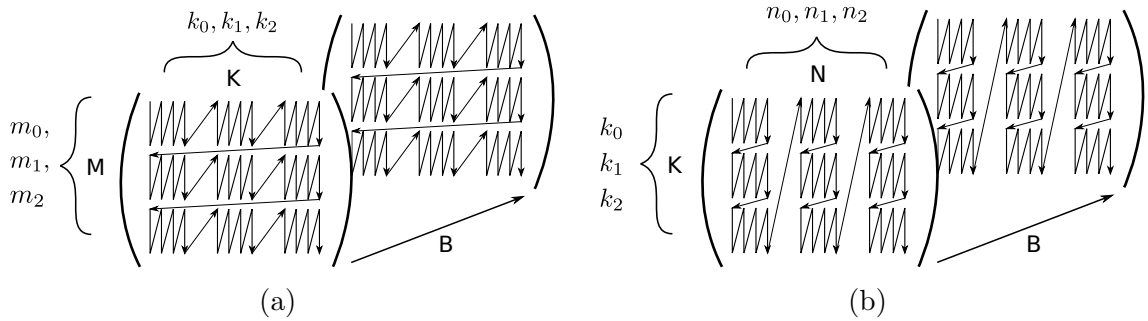


Figure 13: (a) Memory layout for a left input tensor. (b) Memory layout for a right input tensor.

shapes results in varying performance. Fig. 14 shows the attained performance of different kernel shapes on an Intel Xeon Platinum 8360Y processor. The sizes of the matrices matter and so does the choice of dimensions the GEMM is performed on. There are multiple ways to select dimensions. First, one could pick dimensions where their sizes are close to a matrix shape that performs well. This, however, limits the possible matrix shapes drastically. Another observation is that multiple dimensions of the same type can be merged into a bigger dimension and one dimension can be split up into multiple smaller dimensions. This rapidly increases the number of possible matrix shapes. At the smallest level, one can divide a dimension d into multiple dimensions $d_0, \ldots, d_n$ where $|d_0|, \ldots, |d_n|$ are the prime factors of $|d|$. The shape of one matrix is now only limited by a product combination of all the prime factors regarding one dimension type. The research of Springer and Bientinesi about the GETT approach [26] essentially targets this problem by finding ways to reshape the tensors to get new possible matrix shapes. However, it still does not guarantee a well-shaped matrix. For example, in a tensor consisting of one M dimension m_0 with $|m_0|$ being a large prime number, the M dimension can't be split in any way. One would be stuck with the given matrix shape with $|m_0|$ rows.

Counting up indices Until here, there only have been considerations about how to permute dimensions of an intermediate tensor. While characteristics 1. - 3. can be met with this approach, characteristic 4. is still dependent on the tensor provided. Standard permutation ensures that tensors have strides for each dimension and elements can be accessed using Eq. (2). This means that intermediate tensors could be stored and used independently of the einsum they originate from. Since intermediate tensors are generally not used outside the einsum, accessing elements can be made more complicated in order to guarantee matrix multiplications of a desired shape.

Let's consider a binary contraction with three K dimensions k_0, k_1 and k_2 with dimension sizes $|k_0| = 2$, $|k_1| = 2$ and $|k_2| = 3$. In addition, the kernel shape should be $kernel_k = 4$. To achieve this, it is easy to see that k_0 and k_1 can be merged to K_{01} with $|K_{01}| = 4$. k_2 could then be used to loop around the GEMM. The looping stride is equal to the stride of k_2 in $\mathcal{I}_{k_2,K_{01}}$. In this case, it is $|k_0||k_1| = 4$ and is equal to $kernel_k$. The left side of the table in Fig. 15 shows what indices of the K

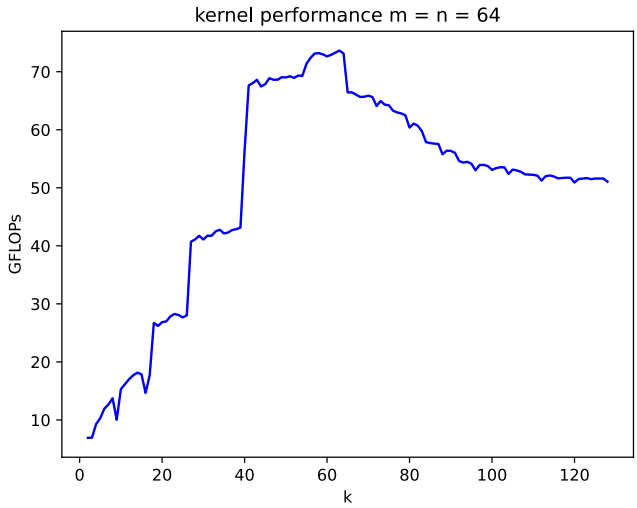


Figure 14: Kernel performance in GFLOPs for different matrix shapes using double precision floating point numbers. m = n = 64 are constant and k varies from 2 to 128.

dimensions are part of what matrix in the K direction. There are a total of three matrices divided by a horizontal line. By going down rows the indices are "counted up" from right to left. By going down four elements, only k_2 will be increased by one while k_0 and k_1 remain constant. This is because $|K_{01}| = 4$ divides the stride 4 for jumping a matrix. An algorithm that increases indices $i_0, \ldots, i_{n-1}$ by one is shown in Alg. 1. "Counting up" refers to increasing indices multiple times.

Let's now look at a different approach. As already stated, finding a dimension combination so that a specific kernel size is accomplished is not always possible. The search for a combination can have a large search space if tensors have lots of dimensions or dimensions that have a long prime factorization. To work around this, we can look at what happens if we take the order of the K dimensions as granted and fetch $kernel_k$ elements for each matrix. For our example, the indices resulting from this procedure are shown in the right column of the table in Fig. 15. Jumping one matrix still results in a matrix stride of 4. This pattern still provides matrices in a great memory layout. No exhaustive search of potential reshape operations is needed. Also, any kernel shape can be chosen, since fetching $kernel_k$ elements is always possible until there possibly needs to be a remainder kernel. However, the relation between the respective indices of different matrices is complicated. Rows 1, 5 and 9 of the table in Fig. 15 show indices of the first elements in each of the three matrices. By going from row 1 to 5 and 5 to 9, k_2 gets increased by one in the left column. There is no clear pattern for doing the same in the right column by going from indices $(0, 0, 0)$ to $(0, 1, 1)$ to $(1, 0, 2)$. First, k_0 stays the same while k_1 and k_2 get increased by one. The second jump results in k_0 and k_2 being increased by one while k_1 gets decreased by one. This makes computing indices of dimensions harder than before.

If we now introduce the same concept to the M dimensions, we get a tensor layout

order of increasing the indices

	(k_2, k_0, k_1)	(k_0, k_1, k_2)
Matrix 1	$(0,0,0)$	$(0,0,0)$
	$(0,0,1)$	$(0,0,1)$
	$(0,1,0)$	$(0,0,2)$
	$(0,1,1)$	$(0,1,0)$
Matrix 2	$(1,0,0)$	$(0,1,1)$
	$(1,0,1)$	$(0,1,2)$
	$(1,1,0)$	$(1,0,0)$
	$(1,1,1)$	$(1,0,1)$
Matrix 3	$(2,0,0)$	$(1,0,2)$
	$(2,0,1)$	$(1,1,0)$
	$(2,1,0)$	$(1,1,1)$
	$(2,1,1)$	$(1,1,2)$

$$|k_0| = 2$$
$$|k_1| = 2$$
$$|k_2| = 3$$

Figure 15: Indices part of three matrices being increased in different order.

similar to the one shown in Fig. 13a. However, the shape of matrices can be chosen freely and characteristic 4. is finally met. Going one matrix further into the K direction means getting $kernel_k$ new index tuples $(k_0, \ldots, k_n)$ by counting up from the rightmost index tuple in the current matrix. I call this memory layout the **fixed block layout** (FBL). The LAGEMM is now implementable by having four loops (batch, M, N, K) around a GEMM of any desired shape defined by $kernel_m$, $kernel_n$ and $kernel_k$. Assuming the kernel sizes divide the respective sizes $|M|$, $|N|$ and $|K|$[1], the number of loop iterations will be $kloop = |K| \div kernel_k$, $mloop = |M| \div kernel_m$ and $nloop = |N| \div kernel_n$. Note that the number of loop iterations $bloop = |B|$ is not divided by any kernel size, since batch dimensions are not part of the GEMM.

With intermediate tensors in FBL, a LIBXSMM kernel can now be used to efficiently calculate a binary einsum contraction between them. The remaining problem is to think of an unpacking routine that writes elements of kernel result matrices to the correct position in the respective output tensor following the FBL.

5.2.2 Unpacking routine for the FBL

The goal of this section is to describe an unpacking routine that can write elements of result matrices of a LIBXSMM kernel to their correct positions in memory. Considering that the result tensor is in FBL, this is more complicated than an unpacking routine for the standard LAGEMM approach.

Let's look at a contraction between the tensors $\mathcal{A}$ and $\mathcal{B}$ resulting in the intermediate tensor $\mathcal{I}$ that is in FBL. The contraction between $\mathcal{A}$ and $\mathcal{B}$ is denoted as ab. Since $\mathcal{I}$ is an intermediate tensor it will be contracted at a later stage with another tensor. This tensor is named $\mathcal{J}$ and their contraction ij. For describing the unpacking

[1]This is still a restriction of the current implementation.

Algorithm 1 increase indices

Input: indices $i_0, i_1, \ldots i_{n-1}$, dimension sizes $s_0, \ldots, s_{n-1}$
Output: increased dimensions $i_0, i_1, \ldots i_{n-1}$

```
for j ← n − 1 to 0 do
    i_j ← i_j + 1
    if i_j < s_j then
        break
    end if
    i_j ← 0
end for
return i_0, ..., i_{n-1}
```

routine, $\mathcal{A}$ and $\mathcal{B}$, as well as $\mathcal{J}$ don't have to be in FBL, but they may be. For the contraction ab, the usual types of dimensions exist, resulting in the sets of dimension types B_{ab}, M_{ab}, N_{ab} and K_{ab}. The respective sets are present for the contraction ij, namely B_{ij}, M_{ij}, N_{ij} and K_{ij}. If we assume that $\mathcal{I}$ is a left input tensor, then N_{ij} is of no importance going further. It is important to understand the connection (or no connection) between the sets of each contraction. Any dimension appearing in K_{ab} will not be in any of the sets for ij. All the dimensions in $B_{ab} \cup M_{ab} \cup N_{ab}$ are also in $B_{ij} \cup M_{ij} \cup K_{ij}$ and with this present in $\mathcal{I}$. Note that these are also the only dimensions in $\mathcal{I}$, so given their indices it should be possible to calculate the respective offset in $\mathcal{I}$. There are no further dependencies between the dimensions of both contractions. A dimension d can be part of B_{ab} and M_{ij}, meaning d is a batch dimension in ab, but an M dimension in ij. All the other combinations are also possible. Fig. 16 shows an example contraction and the respective dimension type sets. The sets have to be taken into consideration while calculating an offset for an element that should be stored in $\mathcal{I}$.

To do so, one needs to know where the elements of a result matrix $\mathcal{M}$, coming out of the LIBXSMM kernel, need to be unpacked to $\mathcal{I}$. This boils down to two things:

1. Find out the dimension indices for every element in $\mathcal{M}$.
2. Calculate the respective offset for the element in $\mathcal{I}$.

Retrieving indices The matrix $\mathcal{M}$ holds indices for all dimensions being part of $B_{ab} \cup M_{ab} \cup N_{ab}$ since those are all the dimensions of $\mathcal{I}$. Indices that are part of B_{ab} are batch dimensions and constant throughout all elements of $\mathcal{M}$. Retrieving them is simply done by knowing about the order in which the batch dimensions are increased. So for every batch loop iteration, they get increased by one. To know about the indices that are part of M_{ab} or N_{ab}, a starting point of indices, as well as the order in which they are increased, are needed. A starting point refers to the indices for the top left element of $\mathcal{M}$. $\mathcal{M}$ can now be traversed and indices increased until the indices for every element are known. Alg. 2 shows how the indices of matrix elements are retrieved.

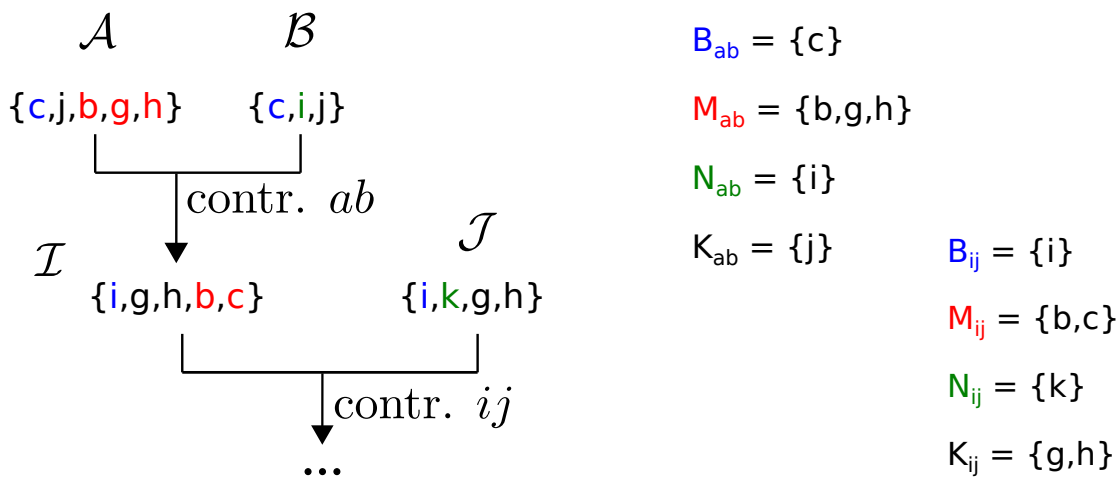


Figure 16: Sets of dimension types for contractions ab and ij.

Algorithm 2 retrieve indices

Input: $kernel_m$, $kernel_k$, start values $start_{m0}, start_{m1}, \ldots$ and $start_{n0}, start_{n1}, \ldots$
Output: indices for each element

```
m_0, m_1, ... ← start_m1, start_m2, ...
n_0, n_1, ... ← start_n1, start_n2, ...
for i ← 0 to kernel_n − 1 do
    m_0, m_1, ... ← start_m1, start_m2, ...
    for j ← 0 to kernel_m − 1 do
        //current indices can be retrieved here
        increaseIndices(m_0, m_1, ...)
    end for
    increaseIndices(n_0, n_1, ...)
end for
```

Calculating offsets Given an index for every dimension in $B_{ab} \cup M_{ab} \cup N_{ab} = B_{ij} \cup M_{ij} \cup K_{ij}$, the goal now is to calculate the corresponding offset in $\mathcal{I}$. Normally, one would multiply every index with the stride of the respective dimension in $\mathcal{I}$. Since $\mathcal{I}$ is in FBL, this is not possible for every dimension. First, let's match the dimensions in $B_{ab} \cup M_{ab} \cup N_{ab}$ to one of the sets B_{ij}, M_{ij} or K_{ij} they appear in. Indices for dimensions in B_{ij} can still be multiplied with their dimension strides in $\mathcal{I}$. This is possible since batch dimensions of $\mathcal{I}$ only loop around matrix slices that are in FBL. All the slices are of the same size which provides a proper stride. This gives the part of the offset that dimensions in B_{ij} are responsible for. For dimension in M_{ij} or K_{ij}, no such stride exists because of the FBL.

Let's consider dimensions in M_{ij} first. In $\mathcal{I}$, for every new matrix in M direction, the indices for dimensions in M_{ij} will be increased $kernel_m$ times. The dimensions still have an order in which they are increased. By knowing the indices of the dimensions, one knows exactly how often the indices have been increased. Similar to strides, each dimension has an increase-stride $iStr$. Assume that $m_0, m_1, \ldots, m_{|M_{ij}|-1}$ are counted up from right to left, then their increase-stride is calculated by

$$iStr(m_i) = \prod_{k=i+1}^{|M_{ij}|-1} |m_k|. \tag{7}$$

The number of m-increases is then calculated with indices $i_0, \ldots, i_{|M_{ij}|-1}$ by using the equation

$$\text{m-increases} = \sum_{k=0}^{|M_{ij}|-1} i_k \cdot iStr(m_i). \tag{8}$$

The way these equations function can be described with number systems. The indices regarding dimensions $m_0, m_1, \ldots, m_{|M_{ij}|-1}$ form a number N that is equal to the m-increases. This number can be interpreted with a custom number system, where each digit has its own maximum value. If all the dimension sizes were 10, the m-increases are equal to N interpreted in the decimal system. The respective $iStr$-values would then be $\ldots, 100, 10, 1$, corresponding to how much the number increases by increasing the respective digit by one.

With the number of m-increases, it is possible to find the index of the current matrix in the M direction by calculating $iMat_m = \lfloor \text{m-increases} \div kernel_m \rfloor$. The remainder $rMat_m = \text{m-increases} \mod kernel_m$ is the offset inside the current matrix in the M direction. $iMat_m$ still needs to be multiplied with a stride of a matrix jump in the M direction. This stride is possible to compute by knowing how many matrices are stored along the K dimension since matrices are stored along the K dimension first. The number of matrices along the K dimension is calculated by $nK = |K| \div kernel_k = kloop$. With nK, the part of the offset that dimensions in M_{ij} are responsible for is $iMat_m \cdot nK \cdot kernel_m \cdot kernel_k + rMat_m$, where $kernel_m \cdot kernel_k$ is the size of a matrix. $nK \cdot kernel_m \cdot kernel_k = str(Mat_m)$ can be seen as a stride for going one matrix into M direction. The partial offsets of dimensions in K_{ij} can be calculated similarly, as well as N_{ij} if $\mathcal{I}$ would be a right input tensor. Respective numbers of increases would be calculated alongside matrix indices, remainders,

matrix strides, and remainder strides. Adding all the partial offsets results in one final offset of an element in $\mathcal{I}$. An example calculation of an offset for one element coming can be seen in Fig. 17. The figure also provides an overview of all needed strides and indices.

Doing all the above calculations for every element is not practical and too computationally expensive. The strides and increase-strides could be calculated once and saved. The number of increases, indices of matrices, the remainders, and the final product with respective strides still have to be computed. Doing so for every element is still not acceptable.

Offset vectors To help with expensive offset calculation, I use vectors to store values that are part of an offset for an element. A similar idea is used in [19] to assist offset calculations for their scatter-matrix layout. Let's look at a result matrix $\mathcal{M}$ coming out of a LIBXSMM kernel. Using the notation of the previous paragraph, this matrix is calculated at some point in the contraction ab. Again, the goal is to retrieve an offset for each of the elements in $\mathcal{M}$ to be able to store them in the intermediate tensor $\mathcal{I}$, which is in FBL. $\mathcal{M}$ holds dimensions from M_{ab} and N_{ab}. Dimensions in M_{ab} are increased by going down the rows of the matrix. Dimensions in N_{ab} are increased by going columns to the right. The indices of dimensions in B_{ab} are constant for every element in $\mathcal{M}$. The matrix can be seen as a two-dimensional coordinate system. One axis is regarding the rows and the other axis is regarding the columns. The objective is to find vectors for each axis that should encode parts of the offset for an element in $\mathcal{I}$. They can be calculated once at the beginning of a binary contraction and reduce index calculation overhead during the unpacking process drastically. There is no need to have extra vectors for each kernel result matrix. Only each of the dimension sets B_{ab}, M_{ab}, N_{ab} need one vector denoted as v_B, v_M and v_N. Every index i of the vector v_M encodes indices for every dimension of M_{ab}, where the indices have been increased i times. The unpacking routine has to know about the starting indices of the three vectors for a given result matrix. By increasing indices of v_M and v_N in the order the matrix is traversed, respective vector indices can be derived for every element.

A simple offset calculation routine would allow the entries of the three vectors to be added to calculate the offset. This is not the case with offset calculation for the FBL. To view the problem, recall how offsets of elements are calculated. Each dimension of M_{ab} can be part of B_{ij}, M_{ij} or K_{ij} (if the result tensor of ab is a left input tensor at a later stage). The same applies to dimensions in N_{ab} and B_{ab}. Before an offset can be calculated, all dimensions are first assigned to the dimension sets in which they appear in $\mathcal{I}$. Those are the sets B_{ij}, M_{ij}, N_{ij} and K_{ij}. Only after that, partial offsets are calculated for each of the sets and later added. The problem is that the different vectors v_B, v_M, v_N must be calculated independently of one another. This means that entries in v_M only know about dimensions in M_{ab} and their indices. The assignment of dimensions to the dimension sets of ij is incomplete from the viewpoint of v_M. By doing the offset calculation on this view, a problem occurs. Let's consider the partial offset gained from indices regarding dimensions in K_{ij}. The offset depends on the values $iMat_k$ and $rMat_k$ that denote the index of

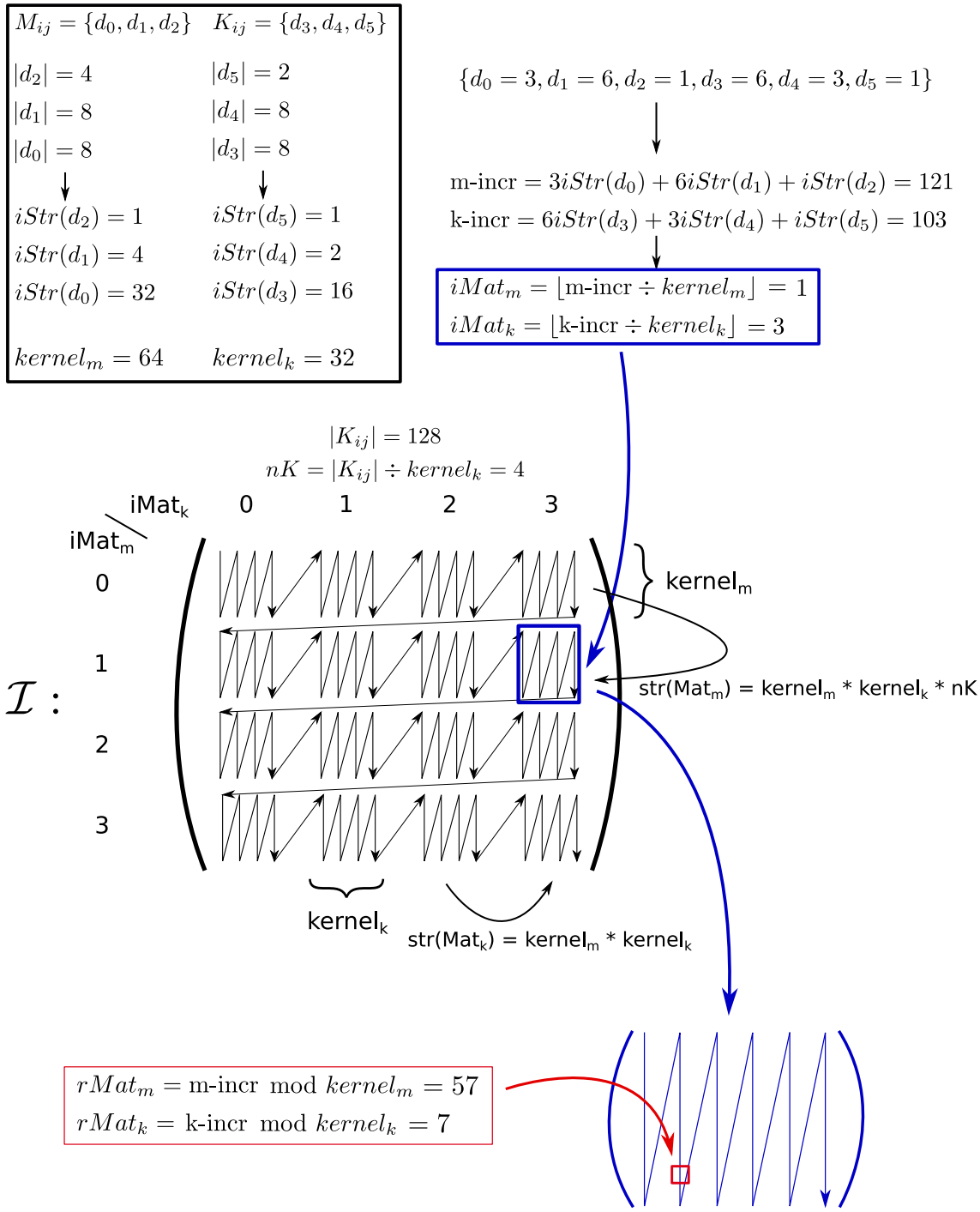


Figure 17: Calculation of the offset for an example element.

the current matrix and the remainder index in the current matrix. The remainder index is an issue. Normally, multiple remainders don't exist. Their values would all be part of the number of m-increases from which $iMat_m$ and $rMat_m$ are derived. Now, each of the vectors v_B, v_M, v_N will calculate an offset based on their view of the dimension sets. This results in multiple remainder indices and a possibly wrong offset calculation by simply adding the partial offsets. The reason is that it is important to make a distinction between $iMat_m$ being increased by one and $rMat_m$ being increased by $kernel_m$. This is because the stride for increasing $iMat_m$ is not equal to $rMat_m \cdot kernel_m$.

Simply adding calculated offsets from v_B, v_M, v_N does not work. However, there is a way to work around this. Each vector entry has to hold multiple values[2]. First, it has to hold the offset that can be calculated from the partial view the vector has. This is the part of the offset that is independent of the remainders. It serves as a summand to later calculate the final offset. I call this offset the **applicable offset**. Secondly, the vector has to have the remainders stored for each of the partial offsets resulting from the index sets M_{ij} and K_{ij} (or N_{ij} for a right input tensor). Again, dimensions in B_{ij} don't have a remainder since they serve as an outer loop around matrix slices of a fixed size. With the remainders stored in v_B, v_M and v_N, it is now possible to calculate the offset for an element in $\mathcal{I}$. To do so, one can add respective remainders that reference the same dimension set, let's say K_{ij}, of the three vectors. The gained value can then be used to calculate how much further $iMat_k$ has to be increased and what the real remainder $rMat_k$ is. Now, both values can be multiplied with matching strides. The result is the last summand needed to finish the partial offset calculation of indices regarding dimensions in K_{ij}. By doing the same for the remainder regarding M_{ij}, the final offset is calculated by adding all summands. The whole process is shown in Fig. 18, where the offset of an element in the result matrix is calculated using v_M, v_N, v_B. The indices of the vectors are dependent on respective start values. The index of v_B is constant for the complete result matrix.

5.3 Current implementation

Input tensors of the einsum expression There is still the problem that the input tensors of the einsum expression are not in FBL, so the described LAGEMM approach can't be applied to them. The current solution is to simply change the memory layout of all these tensors so that they are in FBL. Let's assume we have i input tensors for an einsum. There will be $i-1$ binary contractions until the final result tensor is calculated. This means that there will be $i-2$ intermediate tensors since the final output tensor is not one of them. The ratio of tensors that do not need to be permuted to those that need to be permuted is $\frac{i-2}{2i-2} \approx \frac{1}{2}$. It is possible to contract tensors that are not in FBL using different techniques. This would result in research like [26] or [19] that looks at traditional tensor contractions and their high-performance implementations. It is not part of this thesis and my current implementation. Nevertheless, there are einsum expressions with input tensors that

[2]This is a matrix, of course. However, I still call them offset vectors.

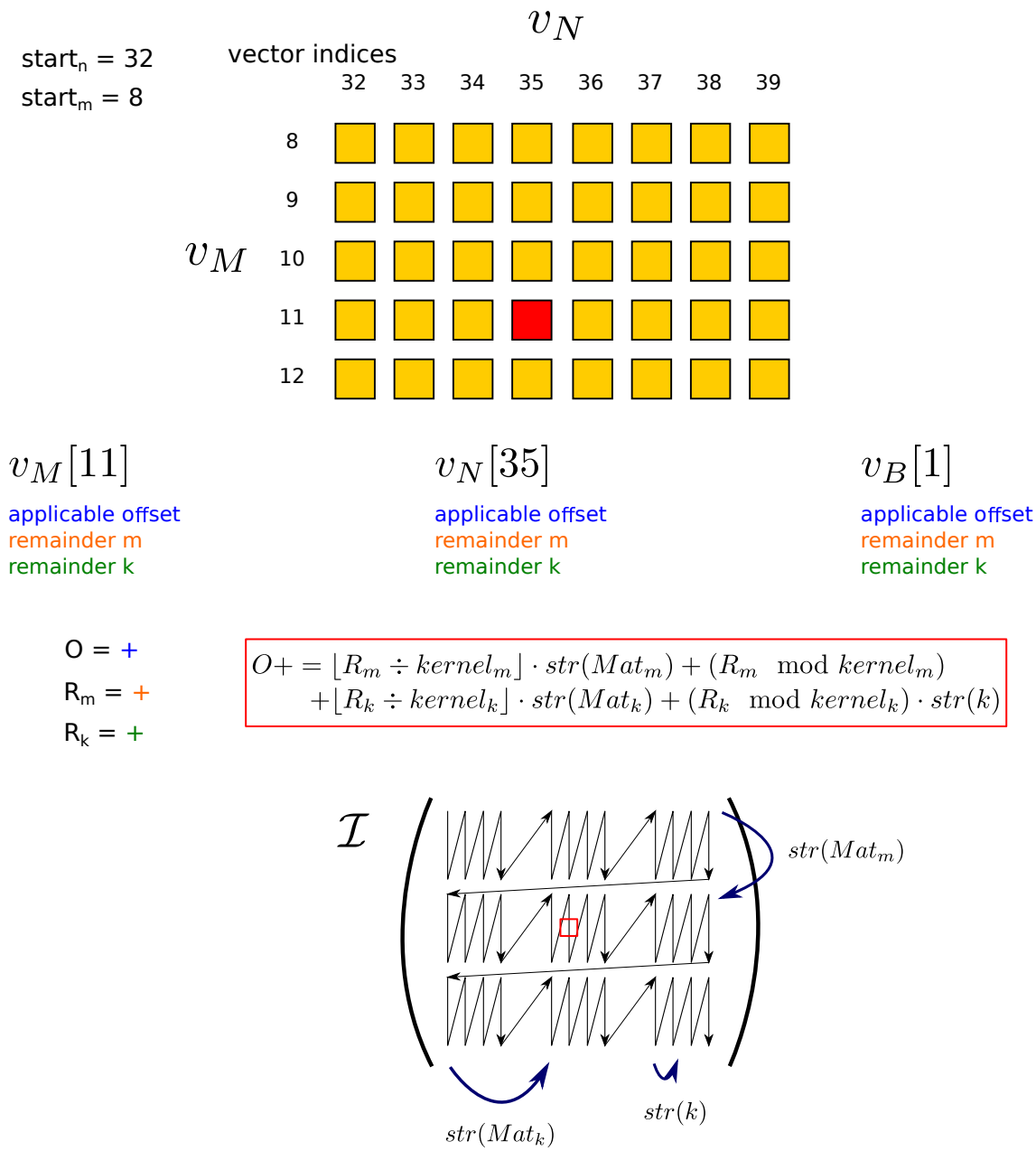


Figure 18: Offset calculation using offset vectors.

do not need to be permuted to be in FBL. They already are. The einsum `km, nk -> nm` encodes a column-major matrix multiplication. Given that the dimensions behind m, n, k are less or equal to $kernel_m, kernel_n, kernel_k$, both tensors are in FBL, to begin with. Some einsums that refer to operations on small tensors can be contracted without any permutation. Additionally, the current implementation makes use of the possibility to transpose matrices. The einsum `abxy, xycd -> abcd` could then be contracted the same way as `xyab, cdxy -> abcd` without permuting the input tensors. The unpacking routine can be redundant too if the result tensor has a suitable memory layout. In those cases, the matrix multiplication kernel writes its results to the tensor immediately, without a cached matrix that needs to be unpacked.

Kernel shapes A desired kernel shape can be defined for each of the binary contractions. The actual shape is then given by $kernel_m = \min(desired_m, |M|)$, $kernel_n = \min(desired_n, |N|)$, $kernel_k = \min(desired_k, |K|)$. If a desired dimension is not chosen because it is too big, then there is spare kernel space. If $desired_m = 64$ but $|M| = 16$, then $kernel_m = 16$ and $kernel_n$ or $kernel_m$ can be doubled twice. The kernel shapes are allowed to get larger than the respective desired values if the permutation of inputs can be skipped as a consequence. With this heuristic, kernel performance is decreased a little to reduce the number of memory operations.

Parallelization The current implementation is multithreaded by using OpenMP [3]. There are four loops around the GEMM kernel with the K-loop being the inner one. Each of the B, M or N-loops can be parallelized because they have an impact on the address where elements are written into the result tensor. Running one of the three loops in parallel results in no write dependencies since all result elements are unpacked to different memory locations. Because the three loop candidates are neighboring, I use the statement `#pragma omp parallel for collapse(3)` to parallelize all of them.

Supported einsum expressions The current implementation does not support all possible einsum expressions yet. There are three major limitations. First, only einsum expressions with multiple input tensors are supported. Since the calculation backbone of my implementation is LIBXSMM, a library targeting matrix multiplications, it is clear that operations on a single tensor are not computable this way. Every other einsum that operates on multiple tensors can be interpreted as a product between matrices or vectors and a LIBXSMM kernel can be applied. The second limitation is that there is currently no support for summing dimensions. If there is an input tensor of an einsum expression that has a dimension identifier which does not appear in any other input tensor, nor the final output tensor, then this dimension can be summed. Summing the dimension is again an operation on a single tensor which is not supported. One could not sum the dimension and consider it as an extra loop, but then the contraction time would not be competitive anymore. The number of arithmetic operations would be larger by a factor of the dimension size of the dimension that could be summed. Currently, for every einsum expression, one has to make sure that every dimension identifier appears in at least two tensors.

The last (and most impactful) limitation is about the possible dimension sizes of the input tensors. The matrices that the GEMM is performed on have a fixed size given by a desired kernel shape. This kernel shape has to divide the respective dimension sizes without any remainder. This means that $|M| \mod kernel_m = 0$, $|N| \mod kernel_n = 0$ and $|K| \mod kernel_k = 0$ must hold. If this is not the case, my implementation does not work yet. A workaround is to choose every dimension size and desired kernel shape so that they are equal to a power of two. This ensures that $|M|, |N|, |K|$ are also powers of two. The kernel shapes then divide $|M|, |N|, |K|$ if they are smaller, or they are set to be equal to $|M|, |N|, |K|$ if they are larger and thus dividing them too.

There are multiple solutions to make it possible to input any dimension sizes. First, one could search for a kernel shape that does divide respective sizes, but then the characteristic that any desired kernel shape can be chosen does not exist anymore. Another possible way is to zero pad the tensors so that the remainder kernel is called on matrices where parts are set to zero. Then the unpacking routine has to know that some elements of the kernel result matrix are not needed and don't need to be written to the result tensor. The approach that fits best to the aspect that LIBXSMM is a library that compiles kernels just in time is to compile kernels that operate on the remainder part. Assume that $|M| = 130$ and $kernel_m = 64$, then the kernel could be called two times into M direction until there is a remainder of 2. It is possible to generate a new kernel of a new shape where $kernel_m$ is two. The complexity is that each of $|M|$, $|N|$ and $|K|$ are possibly not properly divided by the desired kernel shape. This means that $2^3 = 8$ kernels have to be generated, one for each combination where $kernel_m$, $kernel_n$ and $kernel_k$ have their normal size or their remainder size. This adds additional overhead because more kernels have to be compiled. It is not easy to write an unpacking routine that ensures that the output tensor is in FBL while also considering multiple possible kernel shapes within one binary contraction.

6 Results

Throughout this section, I will be comparing the attained GFLOPs numbers (billion floating point operations per second) of my implementation to the einsum implementation of PyTorch [20]. More precisely, I will use the ATen C++ interface. ATen serves as the backend of PyTorch, providing much of its core functionality. Going forward, I will use the terms PyTorch and ATen synonymously. PyTorch's einsum implementation follows the TTGT approach. It permutes input tensors to two large matrices that a batch matrix multiplication is then performed on. Doing so in binary fashion leaves a single result tensor at the end of the contraction. PyTorch's implementation of the einsum routine can be found under [23].

I ran performance tests for einsum expressions containing different numbers of input tensors. The tensors are ranging from small to large, meaning their number of elements varies. There are both tests for single and double-precision floating point numbers. I will present results regarding the multithreaded variant. To analyze some runtimes of my approach in more detail later in this section, I show measurements

on a single core. The CPUs I used to run tests on were

- Intel Xeon Platinum 8360Y CPU @ 48 cores with 252GB RAM
- and Intel Core i5 13600K @ 14 cores (6 performance, 8 efficiency) with 32GB RAM.

My implementation can analyze an einsum expression once and call it multiple times on different input tensors. This analysis is not just a string analysis, it also contains all the offset vector initializations and just-in-time compilations of the GEMM kernels. In the following, I will analyze the einsum once for tests regarding my implementation, and then call the calculation routine multiple times without the need to examine the einsum string again. I will provide GFLOPs numbers regarding the einsum routine for both including and excluding the analysis. PyTorch analyzes the einsum expression every time the `torch::einsum()` function is called, resulting in an overhead. This overhead is minuscule for einsums with lots of calculations to be done. However, it can be a bottleneck of einsum expressions regarding small input tensors, for example small matrix multiplications. Since my implementation delivers a final output tensor that has unit stride, the measurements for PyTorch will include the statement `out_tensor = torch::contiguous(out_tensor)` at the end to ensure the same.

The desired kernel shapes have been chosen by using a grid search over the variables $kernel_m$, $kernel_n$ and $kernel_k$. For example, the kernel shapes for the Xeon Platinum processor are $kernel_m = kernel_n = 32$, $kernel_k = 128$ for double-precision, and $kernel_m = kernel_n = 64$, $kernel_k = 128$ for single-precision. If there are not enough elements in the tensor to use this kernel shape, then the shape of a respective binary contraction gets smaller, as described in section 5.3.

To obtain the number of GFLOPs for both PyTorch and my implementation, I first computed the einsum a few times on the same input data to warm up the caches. Then, I measured the time of a single execution to approximate how often the einsum implementations have to be executed so that they run for about five seconds. Note that this repeat value possibly varies between PyTorch and my implementation. Last but not least, I ran both implementations their respective number of repetitions, timed them, and calculated the number of GFLOPs by using the average runtimes.

6.1 Ordinary einsum expressions

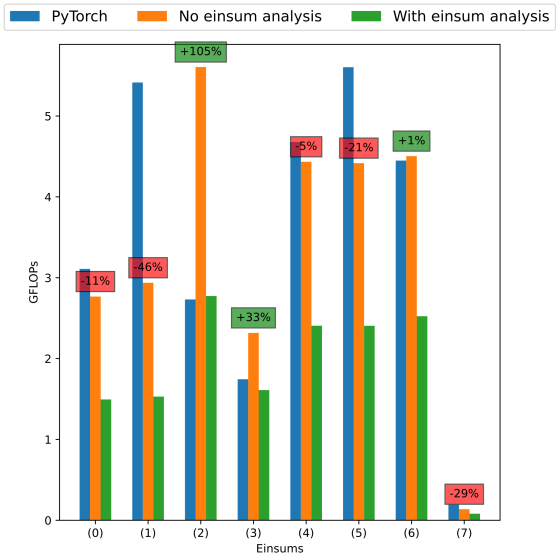
This section shows tests for ordinary einsum expressions. These expressions denote basic linear algebra operations, like (batched) matrix multiplications, chain matrix multiplications, or vector-matrix products. Where multiple input tensors appear, they are contracted from left to right. This will be the case for all upcoming tests. Some expressions only have two input tensors, meaning there will be no intermediate tensors. For those einsums, the number of necessary memory permutation operations for my implementation is comparable to that of PyTorch. Both inputs are permuted first, then GEMMs can be performed. Depending on the memory layout of the final output tensor, PyTorch possibly permutes it too, whereas my implementation writes the elements to the correct position in memory during the unpacking step.

For these simple einsum expressions that operate on small input tensors, it is expected that my implementation profits from not needing to analyze the einsum multiple times. Additionally, it is possible to skip the permutation of the input tensors, as well as the unpacking routine most of the time. The runtime is then simply about calling the kernel and its calculation time, which is short given the small matrix sizes. The results regarding the ordinary einsum expressions on small tensors are shown in Fig. 19 for the Xeon (top) and i5 processor (bottom). All the dimension sizes were set to 32, except for the two batch matrix multiplications, where the dimensions sizes were chosen to be 16. Note that my implementation does support different-sized dimensions too. The left figures show results for single-precision numbers, the right side for double-precision numbers. Blue bars show the GFLOPs attained by PyTorch, orange bars the GFLOPs of my implementation, and in contrast to the green bars without the einsum analysis (including offset vector initializations and kernel compilations). The percentage numbers on top of the orange bars show the relative increase/decrease in performance compared to PyTorch. This is of course an optimistic value since it excludes the analysis.

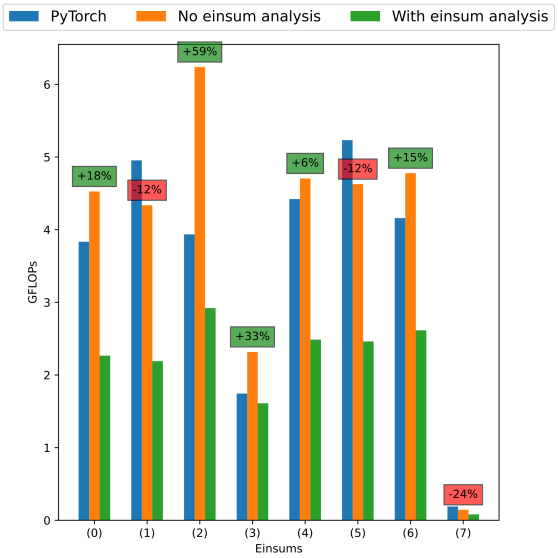
The expected outcome that my implementation profits from the small size of the tensors can be seen in the results regarding the i5, but not necessarily on the Xeon. On the i5, my implementation performs better in almost all cases, while on the Xeon it performs worse in five of the eight single-precision tests. The reason for that is not clear, since further runtime analysis shows that calculating the einsum `km, nk -> nm` is indeed only about calling the kernel with some additional overhead coming from boilerplate code. It is interesting to see that the implementations perform differently relative to each other on both processors.

The relations between the runtimes of PyTorch and my implementation are similar for using single or double-precision floating point numbers. All upcoming diagrams will only be showing single-precision numbers. The results for running the same ordinary tests for larger-sized tensors are shown in Fig. 20. Here, all the dimensions are of size 2048, except dimensions of tensors in the batched multiplications, where dimensions are of size 512. The einsum analysis step is of no importance since the calculation and memory operations take most of the time. The attained GFLOPs are larger than before because there is more data that matrix multiplication kernels can be called on in parallel. Consecutive kernel calls can be executed without any time between them. In comparison to the smaller tensors, PyTorch has better results since the costs of permute operations are less decisive on larger tensors than on smaller ones. The amount of necessary calculation grows faster with increasing tensor sizes than the number of memory operations. After permutation, the used batched matrix multiplication routine can show its full potential. The matrix-vector multiplication of my implementation performs poor. Further runtime analysis has shown that a large amount of time (over 90% of total runtime!) has been spent on permuting the inputs. Permutation is more expensive for matrix-vector products since the number of necessary FLOPs is less compared to a matrix multiplication with comparable dimension sizes. It is not worth permuting input tensors to achieve faster computation because there is not much calculation to be done.

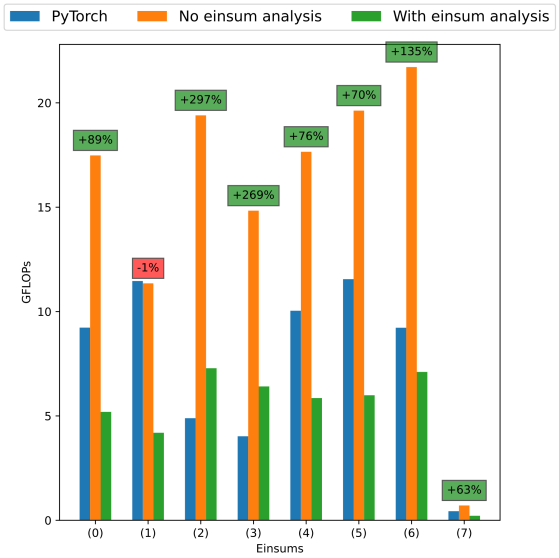
It is noteworthy that my approach using LIBXSMM outperforms a normal PyTorch



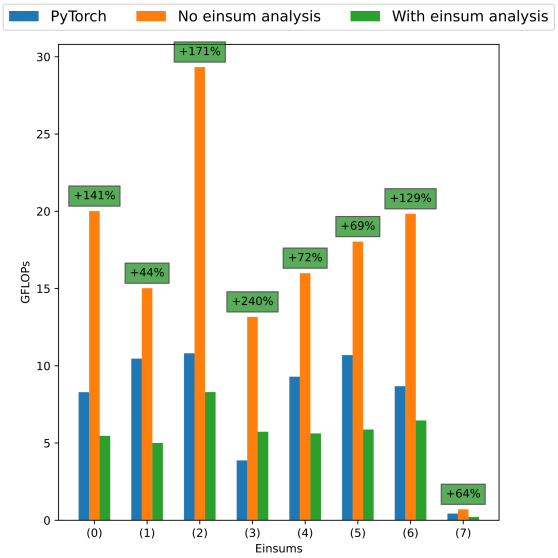
(a) Xeon single-precision



(b) Xeon double-precision

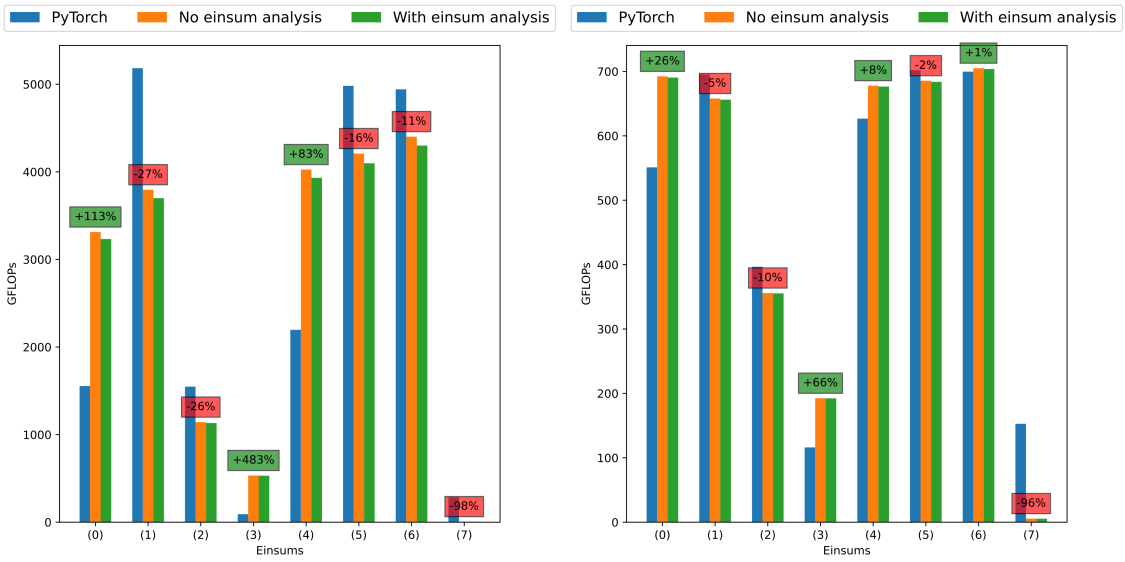


(c) i5 13600k single-precision



(d) i5 13600k double-precision

Figure 19: Results for einsums encoding simple linear algebra operations on small tensors.
(0) `km,nk->nm`, (1) `mk,kn->mn`, (2) `mkb,nbk->bmn`, (3) `mkb,nbk->bmn`, (4) `ab,ca,dc->db`, (5) `ba,ac,cd->bd`, (6) `ba,bc,cd,ed,fe,fh->ah`, (7) `mk,k->m`



(a) Xeon single-precision

(b) i5 16600k single-precision

Figure 20: Results for einsums encoding simple linear algebra operations on large tensors.
(0) `km,nk->nm`, (1) `mk,kn->mn`, (2) `mkb,nbk->bmn`, (3) `mkb,nbk->bmn`, (4) `ab,ca,dc->db`, (5) `ba,ac,cd->bd`, (6) `ba,bc,cd,ed,fe,fh->ah`, (7) `mk,k->m`

in a column-major GEMM with $m = n = k = 2048$ by a factor of 2 on the Xeon processor. My routine is not optimized for large matrix multiplications, while the PyTorch implementation uses batched matrix multiplication routines as its calculation backbone. PyTorch is, however, faster for calculating a row-major matrix. This is because its implementation permutes the matrices to row-major matrices first. By increasing the dimension sizes even further, the expected outcome appears: PyTorch surpasses my implementation with dimension sizes $m = n = k = 8192$ for the column-major matrix multiplication on the Xeon.

6.2 Tensor network contractions

The following einsum expressions encode tensor network contractions. Here, the number of input tensors is larger than in the section before. There will be a considerable amount of intermediate tensors. The results for small tensors with single-precision data are shown in Fig. 21 (top), where all dimension sizes are set to 2. My implementation on the i5 is faster than PyTorch in all test cases where the einsum analysis is not measured and in most of them on the Xeon. Again, the analysis takes a relatively large amount of time because there is not much calculation to be done. Note that einsum (0) and (3), as well as (1) and (4) denote the same network contraction. Only the orders of dimensions in the input tensors are different. It can be seen that the permutations have almost no effect on performance. PyTorch permutes the tensors in all cases. My implementation does so too for the input tensors. After that, the contraction routines on differently permuted tensors are the same. The overall achieved GFLOPs are rather poor for both my implementation and PyTorch. Even if parallelization can't be utilized efficiently on these small ten-

sors, interpreting the numbers as single-core performances does not yield any better outcome.

The results regarding the same contractions for larger tensors are shown in Fig. 21 (bottom). Here, the dimensions sizes are chosen to be 8 for the einsums $(0), (2), (3)$ and 4 for the einsums $(1), (4)$. Note that this implies that the contraction of the einsum still happens rather quickly, explaining the difference in obtained GFLOPs between the measures with/without the einsum analysis. The analysis takes a considerable amount of time compared to the complete contraction routine. The reason why the dimension sizes are not chosen to be bigger is because some tensors that appear during the contraction would get too large. This is especially the case for einsums (1) and (4) because many dimensions of a binary contraction are kept in the respective binary output tensors, thus reducing the dimension sizes is necessary. The contractions are still far from being calculation bound but perform with higher GFLOPs than their smaller counterparts.

6.3 General einsum expressions

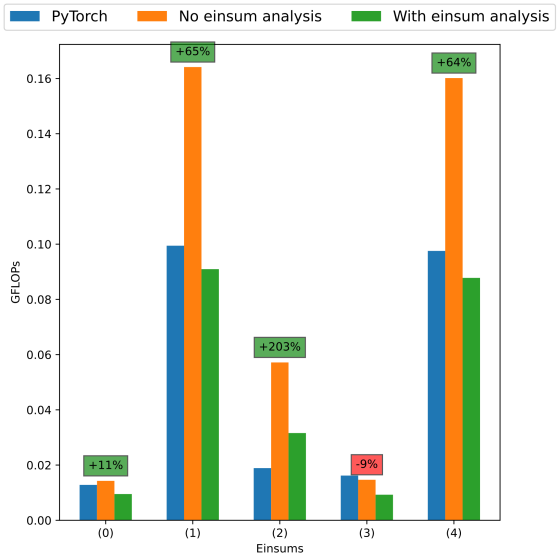
The following results are about more general einsum expressions that don't fall under the category of tensor network contractions. There can be batch dimensions, and an identifier can appear multiple times in the einsum expression. Summing dimensions is not supported, as already stated. So every dimension identifier appears in at least two tensor sub-strings. The results can be seen in Fig. 22 (top) for small input tensors with dimension sizes of 2. My implementation is faster than PyTorch on both processors for the small variants where the einsum analysis is not included in the time measurements. The performances including the analysis are still comparable to those of PyTorch, while sometimes being higher, and sometimes lower. Again, just like with the network contractions on small input tensors, the overall numbers of GFLOPs are very small. There is much room to improve.

Results for the same einsum expressions on larger input tensors are shown in Fig. 22 (bottom). The dimension sizes were chosen to be 8 for einsums (0) - (3) and 4 for einsums $(4), (5)$. The same pattern can be seen as before with network contractions, the numbers of GFLOPs are larger. My implementation now performs a bit worse than PyTorch on the i5, especially if the analysis is considered too.

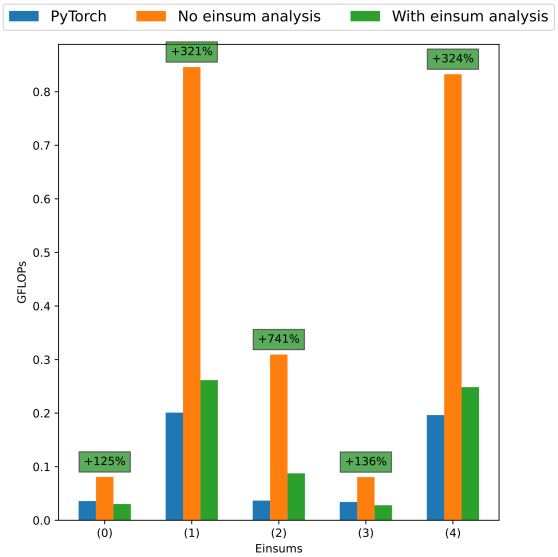
6.4 Detailed runtime analysis

The upcoming results show an in-depth view of where time was spent during different contractions. Multiple subroutines were inspected. The subroutines are:

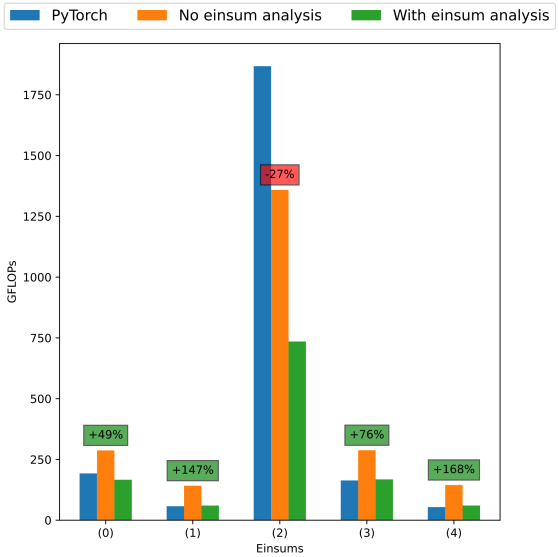
- Kernel: The accumulative time the kernel has taken to calculate all matrix multiplications.
- Permute inputs: The time taken to permute all input tensors so that they are in FBL.
- Unpack calculate offset: The time needed to calculate the offsets for elements in the result matrices. This means doing the computation with the index



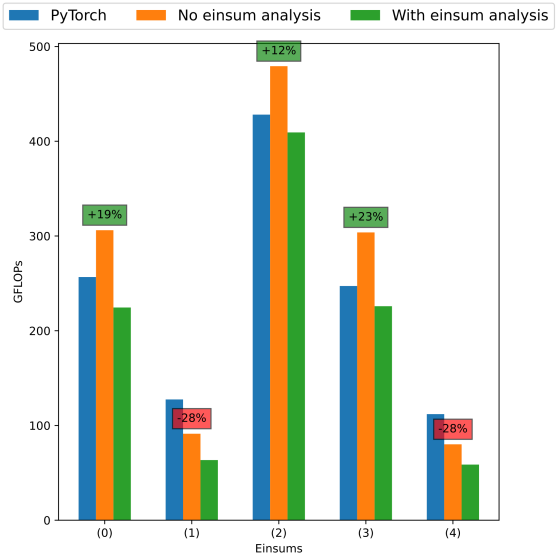
(a) Xeon small tensors



(b) i5 13600k small tensors

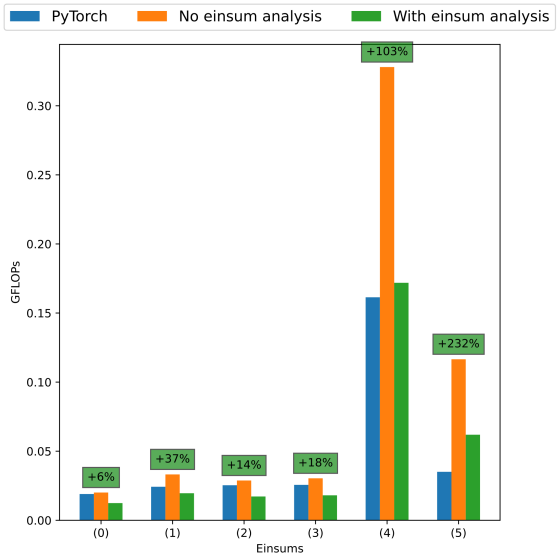


(c) Xeon larger tensors

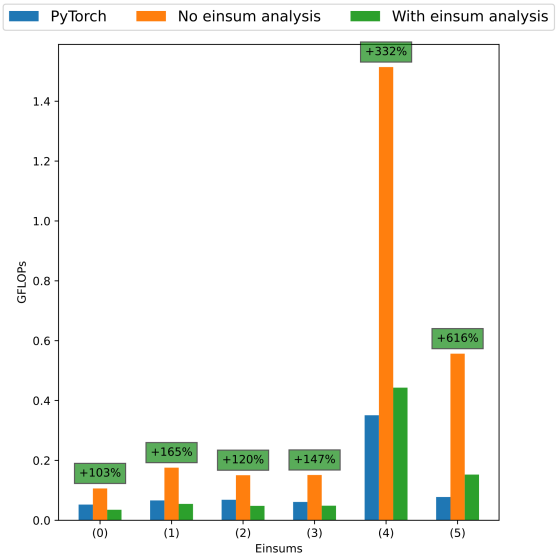


(d) i5 13600k larger tensors

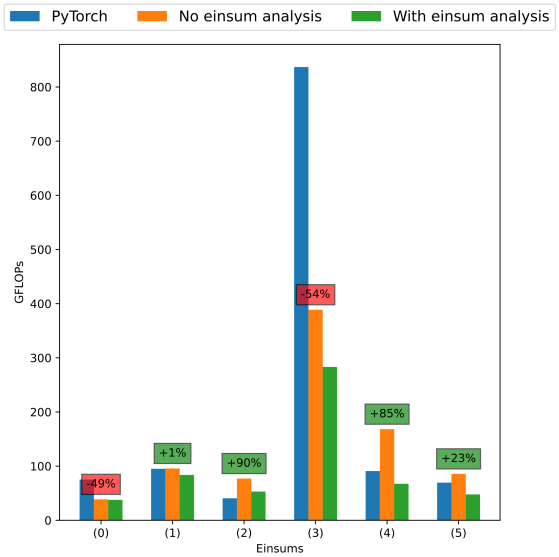
Figure 21: Results for einsums encoding tensor network contractions in single-precision.
(0) `abcd,befg,cdjl,fhij,gikl->aehk`
(1) `abrs,bct,cdefu,defgv,ghwx,hijk,ijlos,lmuv,mnprw,nqtx->akopq`
(2) `atuv,abuvL,bcwx,cdyzA,dewABC,efC,fgBD,ghD,hiE,ij,jtk,klxEFM,lmyGM,mouzN,ouHN,opG,pqI,qrJK,rs,sIK->LFHJ`
(3) `cadb,fbge,cdlj,hifj,lkig->eakh`
(4) `sbra,ctb,ufedc,evdgf,xghw,hjki,olsji,lmvu,pmrwn,txqn->oakpq`



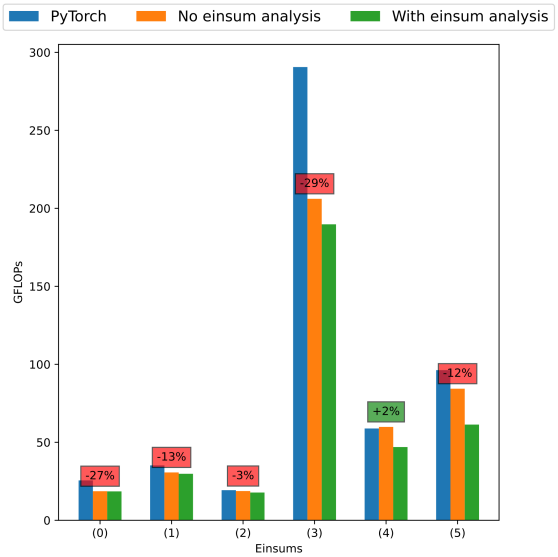
(a) Xeon small tensors



(b) i5 13600k small tensors



(c) Xeon larger tensors



(d) i5 13600k larger tensors

Figure 22: Results for more general einsum expressions in single-precision.
(0) `aabcda,grwas,fdwsar,dgf->abgc`
(1) `abcef,ubhwic,fpewaib,begwi,beop->boauhg`
(2) `abczef,reabq,oza,abqsa,feabo->asrbc`
(3) `abxcd,befg,cdxjl,fhij,giklk->aexhk`
(4) `abrbs,bct,cdefu,deXfgv,ghwhx,hijk,ijlos,lmluv,mnpCrw,nXqCtx->akXopq`
(5) `atXuv,abuvL,bcwx,cdyXzA,dewABC,efXC,fgBD,ghXD,hiE,iXj,jtk,klxEFM,lmyGM,mouzN,ouHYN,oYpG,pqXYI,qrJK,rYs,sIXK->LFYXHJ`

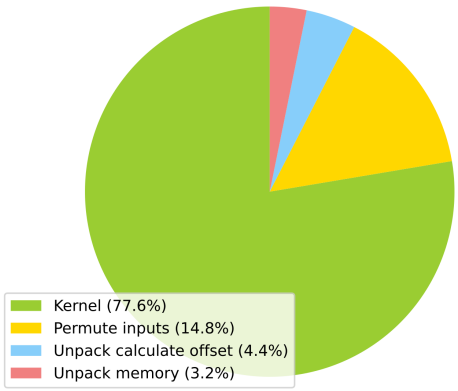


Figure 23: Detailed runtime analysis for the einsum `km,nk->nm` with dimension sizes = 2048.

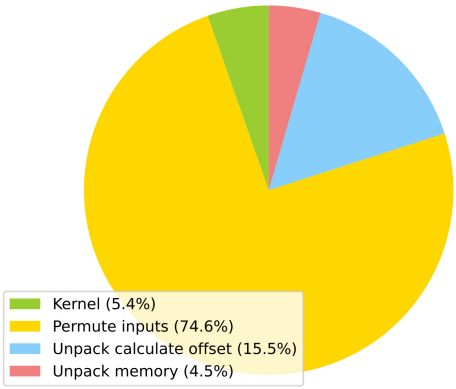
vectors as described earlier.

- Unpack memory: The time taken to write the elements of the result matrices coming from the kernel to their correct memory position in the result tensor.

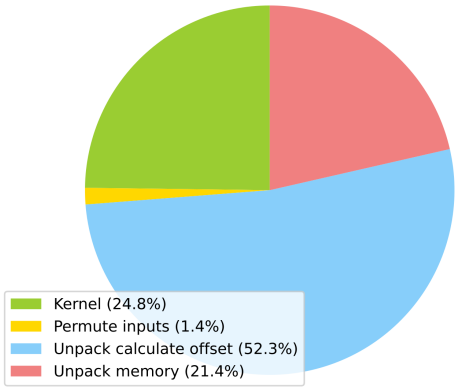
Note that the einsum analysis subroutine is not included. The percentages of runtimes were collected by benchmarking my implementation using a single thread on the Xeon processor. Fig. 23 shows percentages for a large column-major matrix multiplication. As expected, the kernel needs the most time, followed by permuting the inputs. The unpacking routine takes in total about half the time of the permutation, because one large tensor has to be unpacked, but two have to be permuted. One can see that the offset calculation during the unpacking routine takes more time than the memory writing step. This is even more extreme in later tests, so offset calculation using offset vectors still has room to improve.

The percentages of time spent on the different routines differ from einsum expression to einsum expression. The only clear indicator is if tensors are large enough so that the kernel takes the most amount of time. Fig. 24 shows four contractions that were part of previous tests. On the top, the same tensor network contraction is analyzed for differently sized input tensors, so all the dimensions are of size 2 (top left) or 8 (top right). One can see that permuting the inputs takes the most amount of time for contracting the network of small tensors. In contrast, the permutations have almost no effect on the runtime of the contraction of the network consisting of larger tensors. Here, the offset calculation during the unpacking routine takes the most amount of time.

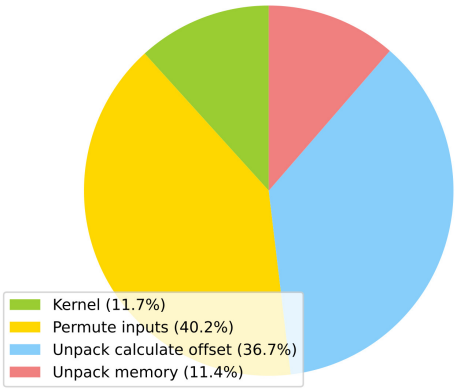
The analysis of a more general einsum contraction is shown on the bottom, also for input tensors with dimension sizes of 2 (bottom left) and 8 (bottom right). The same pattern regarding the time taken for permuting the inputs can be seen as before. For the contraction of small tensors, permuting the inputs takes the most amount of time. It is of no importance for the contraction of the larger tensors. Here, the time taken by the unpacking routine is almost at 90% of the total contraction time. Both the offset calculation and memory writes take a long time to execute. Memory



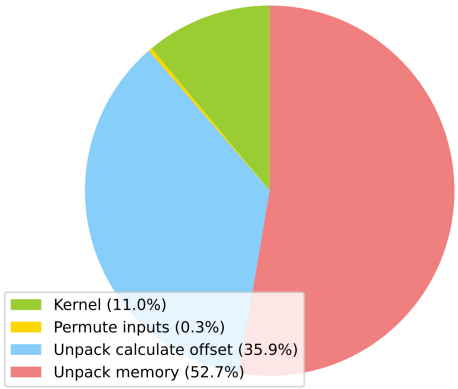
(a) `abcd,befg,cdjl,fhij,gikl->aehk`
dimension sizes = 2



(b) `abcd,befg,cdjl,fhij,gikl->aehk`
dimension sizes = 8



(c) `abczef,reabq,oza,abqsa,feabo ->asrbc`
dimension sizes = 2



(d) `abczef,reabq,oza,abqsa,feabo ->asrbc`
dimension sizes = 8

Figure 24: Detailed runtime analyses for different einsum expressions.

writes take over 50% of the total contraction time. The memory layouts of the tensors are not easy to handle and neighboring elements of a result matrix have to be written to memory locations that are distant from each other.

By looking at the four figures, no clear bottleneck can be found. Depending on the einsum expression, each of the subroutines can take the longest. One noteworthy thing is the time the step for calculating the offset takes. In all four tests it is larger than 15%, in three cases it is larger than 33% and in one case it is more than 50% of the total contraction time. It takes longer than the memory writing step in 3 of the four tests. Further improvement of the offset calculation is necessary to improve these runtimes. The current implementation using the offset vectors is not sufficient and too much calculation needs to be done to find out the offset for a single element.

6.5 Discussion

The obtained performance numbers are comparable to those of PyTorch and are of the same magnitude most of the time. The option to call the contraction routine on an already analyzed einsum expression leads to increased performance for contractions operating on small tensors. In cases where an einsum expression needs to be calculated multiple times on different input tensors, this effect can be exploited. By including the analysis step into the timing of the contraction, the performance gets worse. While it makes no difference for contractions that are computation bound, it decreases the number of GFLOPs for contractions on small tensors. It is important to say that I did not try to optimize the analysis step. The generation of the LIBXSMM kernels is not the bottleneck of the analysis. The computing of all necessary strides and offsets is. However, performance numbers where the timing of the analysis was included are generally still comparable to those of PyTorch. They are sometimes better but worse most of the time.

The performance of my implementation in comparison to PyTorch does not follow a clear pattern, being faster or slower for different einsum expressions. I know that the presented results do not provide a clear conclusion on which implementation is the better one. One is not able to see what type of einsums can be calculated faster by which implementation. It strongly depends on the einsum that should be calculated, and the dimension sizes of the input tensors. What can be said is that using the FBL brings no guarantee for better performance. Finding a good way to benchmark einsum implementations could be part of future work. The overall attained GFLOPs are rather low for most of the calculations. As seen in the benchmark for larger linear algebra operations, the hardware is capable of way more operations per second than achieved by many of the contractions. However, this is true for both my implementation and PyTorch's. It would be interesting to see one of the approaches presented in [19] or [26] in an einsum or tensor network setting and compare them to my implementation. Unfortunately, I am not able to provide a comparison at this point in time.

The unpacking routine of my implementation gained complexity to ensure that the intermediate tensors are in FBL. There are the benefits described in earlier sections on the one hand, on the other hand, the additional difficulty can be seen in the

runtime measurements. The calculation of offsets takes a large amount of time. It is crucial to further reduce the complexity of this step. Furthermore, writing result elements back to memory can't be done by using faster vector stores, because there are possibly weird memory access patterns while writing elements of the kernel result matrix to their memory positions in the output tensor. However, this drawback would still appear if intermediate tensors were stored in a more natural memory layout and permuted later, like TTGT does. The permutation results in suboptimal access patterns too. One way to further improve the runtimes is to think of an unpacking routine that does ensure intermediate tensors in FBL, while also being more memory friendly. This could be done by unpacking elements of the kernel output matrix in a different order than before.

7 Conclusion and future work

In this thesis, I implemented a library using LIBXSMM for calculating einsum expressions. It is currently possible to compute a fair amount of einsums under the constraint that dimension sizes have to be a power of two. The implementation builds on top of the described LAGEMM approach, trying to minimize expensive memory operations by finding a good memory layout for intermediate tensors. Results show a mixed view of runtime performances. In comparison to PyTorch, my implementation shows performance numbers in the same order of magnitude. The largest gain in performance is the possibility to analyze einsum expressions once and compute them multiple times afterward, without the need to analyze them again. This means that the presented fixed block layout is not a guaranteed source of higher performance. However, there is room to improve the more complicated unpacking routine to make it more competitive.

In the future, a goal would be to widen the range of einsum expressions my implementation can be used for. This means the possibility to use tensors where the kernel shape does not divide given dimension sizes properly, and a remainder kernel is needed. Another path of research is to further improve the current implementation. Calculating element offsets by using the presented offset vectors is not sufficient and needs to be improved. Unpacking the elements of a kernel result matrix in a different order could result in better memory access patterns in the result tensor to further minder the time taken by the unpacking routine. It is possible to improve the FBL by introducing cache blocking to it. Then, there would be more loops around the GEMM kernel, ensuring that sub-matrices exploit the cache sizes of the machine. Different intermediate tensor layouts have not been examined and could be part of future research. Other layouts could be both more or less complicated to achieve better performances.

References

[1] Rodney J. Bartlett and Monika Musiał. "Coupled-cluster theory in quantum chemistry". In: *Rev. Mod. Phys.* 79 (1 Feb. 2007), pp. 291–352. DOI: 10.1103/RevModPhys.79.291. URL: https://link.aps.org/doi/10.1103/RevModPhys.79.291.

[2] Jack Choquette et al. "NVIDIA A100 Tensor Core GPU: Performance and Innovation". In: *IEEE Micro* 41.2 (2021), pp. 29–35. DOI: 10.1109/MM.2021.3061394.

[3] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313. URL: https://doi.org/10.1109/99.660313.

[4] J. J. Dongarra et al. "A Set of Level 3 Basic Linear Algebra Subprograms". In: *ACM Trans. Math. Softw.* 16.1 (Mar. 1990), pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/77626.79170. URL: https://doi.org/10.1145/77626.79170.

[5] Jack J. Dongarra et al. "An Extended Set of FORTRAN Basic Linear Algebra Subprograms". In: *ACM Trans. Math. Softw.* 14.1 (Mar. 1988), pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/42288.42291. URL: https://doi.org/10.1145/42288.42291.

[6] G. Evenbly and G. Vidal. "Algorithms for entanglement renormalization". In: *Phys. Rev. B* 79 (14 Apr. 2009), p. 144108. DOI: 10.1103/PhysRevB.79.144108. URL: https://link.aps.org/doi/10.1103/PhysRevB.79.144108.

[7] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. "The ITensor Software Library for Tensor Network Calculations". In: *SciPost Phys. Codebases* (2022), p. 4. DOI: 10.21468/SciPostPhysCodeb.4. URL: https://scipost.org/10.21468/SciPostPhysCodeb.4.

[8] "Front Matter". In: *Multi-Way Analysis with Applications in the Chemical Sciences*. John Wiley & Sons, Ltd, 2004. ISBN: 9780470012116.

[9] Martin Ganahl et al. *TensorNetwork on TensorFlow: Entanglement Renormalization for quantum critical lattice models*. 2019. arXiv: 1906.12030 [physics.comp-ph].

[10] Kazushige Goto and Robert A. van de Geijn. "Anatomy of High-Performance Matrix Multiplication". In: *ACM Trans. Math. Softw.* 34.3 (May 2008). ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. URL: https://doi.org/10.1145/1356052.1356053.

[11] Johnnie Gray and Stefanos Kourtis. "Hyper-optimized tensor network contraction". In: *Quantum* 5 (Mar. 2021), p. 410. ISSN: 2521-327X. DOI: 10.22331/q-2021-03-15-410. URL: https://doi.org/10.22331/q-2021-03-15-410.

[12] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[13] Alexander Heinecke et al. "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation". In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 981–991. DOI: 10.1109/SC.2016.83.

[14] Jianyu Huang and Robert A. van de Geijn. "BLISlab: A Sandbox for Optimizing GEMM". In: *CoRR* abs/1609.00076 (2016). arXiv: 1609.00076. URL: http://arxiv.org/abs/1609.00076.

[15] *Intel MKL*. https://www.intel.com/content/www/us/en/docs/onemkl/get-started-guide/2023-0/overview.html.

[16] C. L. Lawson et al. "Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Trans. Math. Softw.* 5.3 (Sept. 1979), pp. 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847. URL: https://doi.org/10.1145/355841.355847.

[17] Ling Liang et al. "Fast Search of the Optimal Contraction Sequence in Tensor Networks". In: *IEEE Journal of Selected Topics in Signal Processing* 15.3 (2021), pp. 574–586. DOI: 10.1109/JSTSP.2021.3051231.

[18] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[19] Devin A. Matthews. "High-Performance Tensor Contraction without Transposition". In: *SIAM Journal on Scientific Computing* 40.1 (2018), pp. C1–C24. DOI: 10.1137/16M108968X. eprint: https://doi.org/10.1137/16M108968X. URL: https://doi.org/10.1137/16M108968X.

[20] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[21] Robert N. C. Pfeifer, Glen Evenbly, and Guifré Vidal. "Entanglement renormalization, scale invariance, and quantum criticality". In: *Phys. Rev. A* 79 (4 Apr. 2009), p. 040301. DOI: 10.1103/PhysRevA.79.040301. URL: https://link.aps.org/doi/10.1103/PhysRevA.79.040301.

[22] Robert N. C. Pfeifer, Jutho Haegeman, and Frank Verstraete. "Faster identification of optimal contraction sequences for tensor networks". In: *Phys. Rev. E* 90 (3 Sept. 2014), p. 033315. DOI: 10.1103/PhysRevE.90.033315. URL: https://link.aps.org/doi/10.1103/PhysRevE.90.033315.

[23] *PyTorch einsum implementation*. https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/Linear.cpp.

[24] Chase Roberts et al. *TensorNetwork: A Library for Physics and Machine Learning*. 2019. arXiv: 1905.01330 [physics.comp-ph].

[25] Daniel G. a. Smith and Johnnie Gray. "opt_einsum - A Python package for optimizing contraction order for einsum-like expressions". In: *Journal of Open Source Software* 3.26 (2018), p. 753. DOI: 10.21105/joss.00753. URL: https://doi.org/10.21105/joss.00753.

[26] Paul Springer and Paolo Bientinesi. "Design of a High-Performance GEMM-like Tensor–Tensor Multiplication". In: *ACM Trans. Math. Softw.* 44.3 (Jan. 2018). ISSN: 0098-3500. DOI: 10.1145/3157733. URL: https://doi.org/10.1145/3157733.

[27] *Stack Overflow einsum explanation*. https://stackoverflow.com/questions/55894693/understanding-pytorch-einsum.

[28] Mihail Stoian. *On the Optimal Linear Contraction Order for Tree Tensor Networks*. 2023. arXiv: 2209.12332 [quant-ph].

[29] *Tensor Network*. https://tensornetwork.org/.
[30] Field G. Van Zee and Robert A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Trans. Math. Softw.* 41.3 (June 2015). ISSN: 0098-3500. DOI: 10.1145/2764454. URL: https://doi.org/10.1145/2764454.

Declaration of independence

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Seitens des Verfassers bestehen keine Einwände die vorliegende Bachelorarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Jena, den 17.08.2023