A Library for Storing and Manipulating Dense Tensors*

Mahmoud Eljammaly
mjammaly@cs.umu.se

Lars Karlsson
larsk@cs.umu.se

Abstract

Aiming to build a layered infrastructure for high-performance dense tensor applications, we present a library, called `dten`, for storing and manipulating dense tensors. The library focuses on storing dense tensors in canonical storage formats and converting between storage formats in parallel. In addition, it supports tensor matricization in different ways. The library is general-purpose and provides a high degree of flexibility.

Keywords: Dense tensors, canonical storage format, tensor matricization, tensor storage format conversion, out-of-place conversion, in-place conversion.

1 Introduction

Tensors or multi-dimensional arrays are used in a diverse set of multi-dimensional data analysis applications. Many software products suitable for tensor computations exist, such as the commercial MATLAB suite enhanced by various open source third party toolboxes. Unlike for computations with matrices where there is a long history of community developed high-performance software libraries being widely used and incorporated into commercial software products, there is yet no analog for computations with tensors. Developing tensor computation algorithms and applications that are open source and independent of large commercial software environments is difficult in large parts due to a lack of open source software support for fundamental tensor operations.

Tensor algorithms and applications tend to either depend on proprietary functions provided by a large software environment or their own application-specific software solutions. Many parallels can be drawn with the early history of the field of matrix computations where every software included its own code for matrix–vector multiplication, scalar products, matrix transposition, and so on. The introduction and widespread adoption of core interfaces such as the BLAS [6, 9, 10, 11, 12, 13, 17, 18, 19] and LAPACK [2] has meant that software reliant on matrix computations have become easier to maintain and now exhibit portable performance. In contrast, the field of tensor computations, especially parallel and high-performance computations, has not yet matured to the point where a standard set of interfaces can be settled. The algorithms used are also much different in nature compared to those used for matrix computations, so it is not clear that the best approach is to mimic what has worked for matrices in the last couple of decades.

Learning from insights made for matrix computations, we nevertheless think that developing a software stack similar to the one depicted in Figure 1 would be a good starting

*Report UMINF 16.22, Dept. of Computing Science, Umeå University, Sweden.

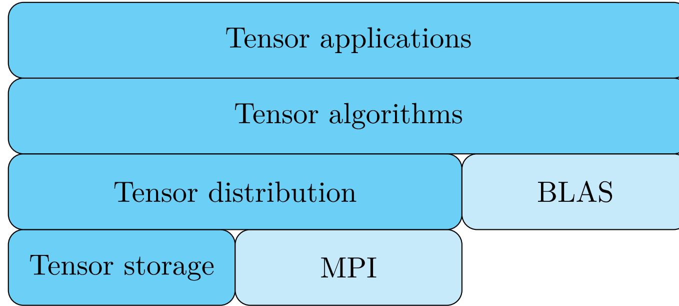


Figure 1: Software stack for tensor computation applications.

point. The proposed stack consists of two established components: the Basic Linear Algebra Subprograms (BLAS) for high-performance fundamental matrix operations and the Message Passing Interface (MPI) for communication between nodes in a distributed memory system. Many fundamental tensor operations can be expressed largely in terms of the BLAS, and MPI is available on virtually any distributed memory system, so there is little doubt that these components will be a part of a future tensor computations software stack. In addition, there are four tensor-specific components in the stack. From the top down: the *tensor applications* component consists of complete applications that use large-scale tensor computations, the *tensor algorithms* component consists of numerical tensor algorithms such as tensor decomposition algorithms and tensor contraction, the *tensor distribution* component consists of such things as communication and (re)distribution of tensors, and finally the *tensor storage* component manages the local storage and manipulation of (sub)tensors on each node in a distributed memory system.

The two components at the top (applications and algorithms) are large and multi-faceted with new algorithms and applications being added as time goes by. But the other two components (distribution and storage) are more fundamental in nature and bounded in scope. We propose to start from the bottom up in an effort to realize a first seed for a tensor computations software stack.

This paper focuses on the *tensor storage* component (see Figure 1) and more specifically on the storage and manipulation of *dense* tensors, i.e., tensors whose elements are mostly non-zero. The sister problem of storing and manipulating *sparse* tensors has been recently addressed by Dahlberg, see [8] and the references within.

Some of the main points of this paper are:

1. Any one-mode or multi-mode tensor matricization is equivalent to converting the storage format of the tensor from one canonical format to another.

2. A tensor stored in a canonical tensor storage format can be interpreted as a matricization of that tensor stored in a canonical matrix storage format.

3. Any tensor storage format conversion can be performed either out-of-place by copying or in-place by in-place permutation.

4. The performance of matricization depends on whether the resulting matrix should be stored in column-major or row-major format.

5. If the ordering of the rows and columns in a matricized tensor is not important, then there exists an efficient way to matricize the tensor.

The rest of the paper is organized as follows. Section 2 gives a mathematical background of tensor storage formats. Tensor storage formats are defined and their relation to matrix storage formats is explained. In Section 3 we present the algorithms used in this paper. Different storage format conversion techniques are discussed in addition to the potential for parallelism. Section 4 provides details about the implementation and introduces the library. Section 5 presents experiments that demonstrate the performance and scalability of the library. Finally, in Section 6 some conclusions and related work are described.

1.1 Notation and terminology

Zero-based indexing is used in order to simplify many of the formulas. We denote by $S(n)$ the set $\{0, 1, \ldots, n-1\}$.

A *sequence* is denoted by angle brackets, e.g., $\langle 0, 1, 2\rangle$, and as a symbol we use a bold lower case letter. The notation $|\cdot|$ denotes the length of a sequence, i.e, the number of elements. The *concatenation* of two sequences $\mathbf{a}$ and $\mathbf{b}$ is denoted by $\mathbf{a} \oplus \mathbf{b}$, e.g., $\langle 0, 1\rangle \oplus \langle 2, 3\rangle = \langle 0, 1, 2, 3\rangle$. A *subsequence* is obtained by deleting zero or more elements from a sequence. An *extraction* of a sequence is denoted by σ and returns a permuted subsequence. An extraction is defined by a sequence of indices, e.g., $\sigma = \langle 2, 1\rangle$, that specify which elements to extract and in which order. The application of σ to a sequence $\mathbf{a}$ is denoted by $\sigma\mathbf{a}$. For example, applying the extraction $\sigma = \langle 2, 1\rangle$ to the sequence $\mathbf{a} = \langle a_0, a_1, a_2, a_3\rangle$ results in the permuted subsequence $\langle a_2, a_1\rangle$. Any extraction that selects the entire sequence degenerates into a permutation and is then denoted by π.

A *tensor* $\mathcal{A}$ of *order* d and *size* $n_0 \times n_1 \times \cdots \times n_{d-1}$ is a d-dimensional array. The size of $\mathcal{A}$ is denoted by the sequence $\mathbf{n} = \langle n_0, n_1, \ldots, n_{d-1}\rangle$. Each *element* of $\mathcal{A}$ is identified by a unique *index sequence* $\mathbf{k} = \langle k_0, k_1, \ldots, k_{d-1}\rangle$ where $k_i \in S(n_i)$ is the *index in mode* i. The $\mathbf{k}$'th element of $\mathcal{A}$ is denoted by $\mathcal{A}(\mathbf{k})$.

2 Canonical tensor storage formats

2.1 Definition

A *storage format* for a dense tensor $\mathcal{A}$ of order d and size $\mathbf{n}$ is a one-to-one mapping of the index sequence $\mathbf{k}$ to the set of integers $S(N)$, where $N = \prod_{i=0}^{d-1} n_i$ denotes the total number of elements. Formally, a tensor storage format is a bijective function parameterized by the size $\mathbf{n}$, i.e.,

$$\phi : S(n_0) \times \cdots \times S(n_{d-1}) \to S(N).$$

The function ϕ maps each index sequence $\mathbf{k}$ to a unique offset in a contiguous memory area of N memory locations.

There are many potential tensor storage formats that fit this definition, but only a few are interesting in practice. A particularly simple and useful tensor storage format is obtained by defining

$$\phi(\mathbf{k}; \mathbf{n}) = \sum_{i=0}^{d-1} k_i \prod_{j=0}^{i-1} n_j. \tag{1}$$

Here, $\mathbf{k}$ is the argument and $\mathbf{n}$ is the parameter to the function. For example, given tensor of size $\mathbf{n} = \langle n_0, n_1, n_2 \rangle$, the function ϕ will map the index sequence $\mathbf{k} = \langle k_0, k_1, k_2 \rangle$ to a continues memory area of size $N = n_0 n_1 n_2$ such that

$$\phi(\mathbf{k}; \mathbf{n}) = k_0 + k_1 n_0 + k_2 n_0 n_1.$$

A more general class of storage formats is obtained by permuting the index sequence $\mathbf{k}$ (and size $\mathbf{n}$) before applying ϕ. Formally, let π be any permutation of a sequence of length d. Then the mapping ϕ_π defined by

$$\phi_\pi(\mathbf{k}; \mathbf{n}) = \phi(\pi\mathbf{k}; \pi\mathbf{n}) \tag{2}$$

is also a valid tensor storage format. Since there are $d!$ permutations of a sequence of length d, there are $d!$ different storage formats of this type. These formats are known as the *canonical (dense) tensor storage formats* and are the focus of this paper.

The concept of a canonical tensor storage format generalizes the row- and column-major storage formats used for matrices. To see this, note that a matrix is a tensor of order $d = 2$ and size $\mathbf{n} = \langle n_0, n_1 \rangle$, where n_0 is the number of rows and n_1 the number of columns. Similarly, an index sequence of the matrix takes the form $\mathbf{k} = \langle k_0, k_1 \rangle$, where k_0 is the row index and k_1 the column index. Choosing the permutation $\pi = \langle 0, 1 \rangle$ in (2) gives the *column-major* matrix storage format, as can be seen by

$$\phi_{\langle 0,1 \rangle}(\mathbf{k}; \mathbf{n}) = \phi(\langle k_0, k_1 \rangle, \langle n_0, n_1 \rangle) = k_0 + k_1 n_0,$$

which we recognize as the column-major ordering. Conversely, choosing $\pi = \langle 1, 0 \rangle$ in (2) gives the *row-major* matrix storage format:

$$\phi_{\langle 1,0 \rangle}(\mathbf{k}; \mathbf{n}) = \phi(\langle k_1, k_0 \rangle, \langle n_1, n_0 \rangle) = k_0 n_1 + k_1.$$

In other words, applying the permutation π to the index sequence $\mathbf{k}$ decides which index of the two will vary the fastest as one scans the memory; $\pi = \langle 0, 1 \rangle$ means that index k_0 will vary faster than index k_1 and $\pi = \langle 1, 0 \rangle$ means that index k_1 will vary faster than index k_0. Generally, the leftmost element in a permutation is the fastest varying one and the rightmost element in a permutation is the most slowly varying one.

2.2 Matricization

A tensor of order d can be reshaped into a matrix, an operation that goes by many names in the literature (e.g., *matricization*, *unfolding*, or *flattening*). We prefer to use the term matricization in this paper. To view a tensor as a matrix, the modes of the tensor need to be partitioned into two disjoint subsets: one subset for the columns of the matrix and another subset for the rows of the matrix. For example, consider a tensor $\mathcal{A}$ of order $d = 4$ and choose the mode subset $\{0, 1\}$ for the rows of the matrix and the complementary subset $\{2, 3\}$ for the columns. The size of the resulting matrix is $\hat{n}_0 \times \hat{n}_1$, where $\hat{n}_0 = n_0 n_1$ and $\hat{n}_1 = n_2 n_3$. To map a tensor index sequence to a matrix index sequence, we need to define two mappings of the form (2) by specifying an extraction σ_{row} for the row dimension and another extraction σ_{col} for the column dimension. For example, we can define $\sigma_{\text{row}} = \langle 0, 1 \rangle$ and $\sigma_{\text{col}} = \langle 2, 3 \rangle$

to obtain the following translation from the tensor index sequence $\mathbf{k} = \langle k_0, k_1, k_2, k_3 \rangle$ to the matrix index sequence $\hat{\mathbf{k}} = \langle \hat{k}_0, \hat{k}_1 \rangle$:

$$\langle k_0, k_1, k_2, k_3 \rangle \mapsto \langle \phi_{\sigma_{\text{row}}}(\mathbf{k}; \mathbf{n}), \phi_{\sigma_{\text{col}}}(\mathbf{k}; \mathbf{n}) \rangle = \langle k_0 + k_1 n_0, k_2 + k_3 n_2 \rangle = \langle \hat{k}_0, \hat{k}_1 \rangle.$$

The matricization defined in this way is denoted by $A_{\sigma_{\text{row}}, \sigma_{\text{col}}}$.

2.3 Matricization and storage format conversion

It turns out that storing a matricized tensor in either the column- or the row-major storage format is equivalent to storing the tensor itself in a canonical tensor storage format of the form (2) for some permutation π. For example, storing the matricization $A_{\langle 0,1 \rangle, \langle 2,3 \rangle}$ in the column-major storage format is equivalent to storing the tensor as in (2) with $\pi = \langle 0, 1, 2, 3 \rangle = \langle 0, 1 \rangle \oplus \langle 2, 3 \rangle$. To see this, note that

$$\phi_{\langle 0,1,2,3 \rangle}(\mathbf{k}; \mathbf{n}) = \underbrace{\phi_{\langle 0,1 \rangle}(\mathbf{k}; \mathbf{n})}_{\hat{k}_0} + \underbrace{\phi_{\langle 2,3 \rangle}(\mathbf{k}; \mathbf{n})}_{\hat{k}_1} \cdot \underbrace{n_0 n_1}_{\hat{n}_0} = \hat{k}_0 + \hat{k}_1 \hat{n}_0,$$

which we recognize as the column-major ordering of $A_{\langle 0,1 \rangle, \langle 2,3 \rangle}$. Similarly, storing the matricization in the row-major storage format is equivalent to choosing $\pi = \langle 2, 3, 0, 1 \rangle = \langle 2, 3 \rangle \oplus \langle 0, 1 \rangle$ in (2).

In general, consider a tensor of order d stored in a canonical format defined by the permutation π and a general matricization of this tensor defined by the extractions σ_{row} for the row dimension and σ_{col} for the column dimension. If $\pi = \sigma_{\text{row}} \oplus \sigma_{\text{col}}$, then the storage mapping (2) for the tensor can be rewritten in the form

$$\phi_\pi(\mathbf{k}; \mathbf{n}) = \phi_{\sigma_{\text{row}}}(\mathbf{k}; \mathbf{n}) + \phi_{\sigma_{\text{col}}}(\mathbf{k}; \mathbf{n}) \cdot \prod_{i \in \sigma_{\text{row}}} n_i,$$

which means that the memory used to store the tensor can be reinterpreted as the matricization $A_{\sigma_{\text{row}}, \sigma_{\text{col}}}$ stored in the column-major format. Similarly, if $\pi = \sigma_{\text{col}} \oplus \sigma_{\text{row}}$, then the same holds but with the matricization stored in the row-major format.

3 Tensor storage format conversion

Given a tensor stored in the format defined by π_{in} and a target format defined by π_{out}, the problem of tensor storage format conversion consists of permuting the tensor elements in memory such that the storage format changes from π_{in} to π_{out}. There are two main types of conversions: out-of-place (OOP) conversion involves the explicit copying of the tensor elements to a separate memory area, but in-place (IP) conversion changes the format by overwriting the old memory area and uses only a small constant amount of additional memory.

The mapping from input memory location ℓ_{in} to output memory location ℓ_{out} is a bijective function $f : S(N) \to S(N)$ and is defined by

$$f(\ell_{\text{in}}; \mathbf{n}) = \phi_{\pi_{\text{out}}}(\phi^{-1}_{\pi_{\text{in}}}(\ell_{\text{in}}; \mathbf{n}); \mathbf{n}) = \ell_{\text{out}}. \tag{3}$$

The mapping consist of two steps: first $\phi^{-1}_{\pi_{in}}$ maps the input memory location ℓ_{in} to the corresponding index sequence $\mathbf{k}$ and then $\phi_{\pi_{out}}$ maps this index sequence to the output memory location ℓ_{out}. The function f determines the memory transfer pattern and is completely determined by the size $\mathbf{n}$ and the input and output formats π_{in} and π_{out}.

3.1 Efficient conversion by moving blocks of memory

For many pairs of formats, the memory transfers implied by f can be arranged into a set of efficient copies of contiguous blocks of memory. Exploiting this feature of the problem whenever possible is important to obtain high performance in the conversion process, since it will benefit from the memory hierarchy and require fewer evaluations of f. In addition, the block transfers is a source of parallelism since different blocks can be transferred at the same time to speed up the process.

To see where the blocks come from and how big they are, suppose that the formats π_{in} and π_{out} have a common prefix σ_{pre} of length m. In other words, it is possible to write $\pi_{\text{in}} = \sigma_{\text{pre}} \oplus \sigma_{\text{in}}$ and $\pi_{\text{out}} = \sigma_{\text{pre}} \oplus \sigma_{\text{out}}$ for some σ_{in} and σ_{out}. Consider the set of elements in the input tensor for which $\sigma_{\text{in}}\mathbf{k}$ are the same but the indices in $\sigma_{\text{pre}}\mathbf{k}$ vary. This set consists of $\prod_{i \in \sigma_{\text{pre}}} n_i$ elements and is stored *contiguously in memory* due to the structure of (2) since the indices in σ_{pre} vary faster than the fastest varying index in σ_{in}. The same holds for the output tensor, and the relative order of the elements in each such block is preserved. Hence, the storage format conversion can be carried out using $\prod_{i \notin \sigma_{\text{pre}}} n_i$ block memory transfers of size $\prod_{i \in \sigma_{\text{pre}}} n_i$.

3.2 Out-of-place versus in-place conversion

The OOP conversion technique involves creating a second tensor and copying each block to its new location in the output tensor. In contrast, the IP conversion technique involves permuting the blocks inside the original memory area, thereby using roughly half of the memory required by the OOP conversion technique. The block transfers form cycles (see Section 3.3) where the blocks in a cycle are shifted within the cycle, see [14] and the references within.

Figure 2 illustrates both the OOP and the IP conversion techniques. The figure shows a tensor of order $d = 4$ and size $\mathbf{n} = \langle 5, 3, 2, 4 \rangle$. The conversion changes the tensor storage format from $\pi_{\text{in}} = \langle 0, 1, 2, 3 \rangle$ to $\pi_{\text{out}} = \langle 0, 3, 2, 1 \rangle$. The in-place conversion consist of six cycles of which two are singleton cycles (i.e., containing only one block). Each cycle is shown in the figure with a unique color, while the singleton cycles share the same color (red).

3.3 In-place conversion techniques

The in-place conversion technique moves tensor blocks one by one from its initial position to its final position within the same memory area. To avoid overwriting the already occupied destination, that block must in turn first be moved to its final position. This continues until a block is encountered whose final position is the initial position of the first block moved. This completes a cycle and the whole permutation consists of one or more such cycles of potentially diverse lengths. Some cycles involve moving/shifting only one block, which actually keeps the block in the same position and involves no data movement at all. Such degenerate cycles are called *singleton cycles* and for any permutation resulting from the mapping function f with at least two blocks there are at least two singletons. See Figure 2 for an example.

3.3.1 Forward versus backward cycle shifting

Different techniques can be used in in-place conversion to make it more efficient [14]. Specifically, a cycle can be shifted in one of two ways: forward or backward.

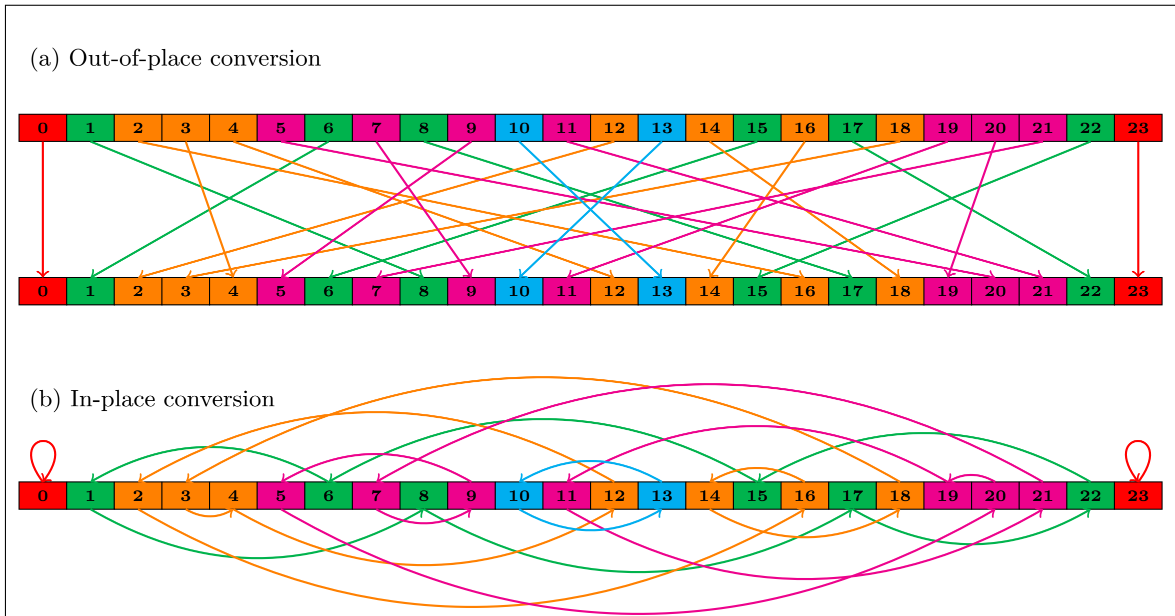
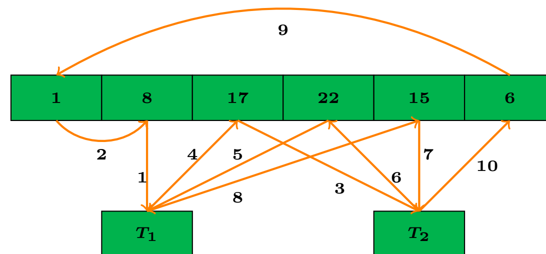


Figure 2: Illustration of the out-of-place and in-place tensor storage format conversion techniques for a tensor of size $5 \times 3 \times 2 \times 4$.

(a) Forward shifting



(b) Backward shifting

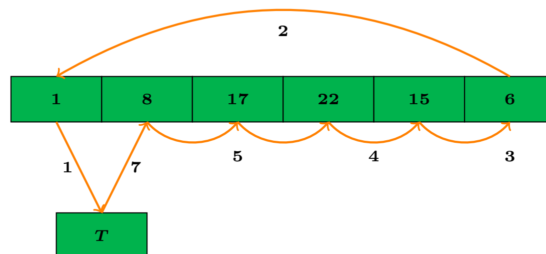


Figure 3: Illustration of forward and backward shifting for a cycle with six blocks. Numbers inside the blocks represent the block's position in the tensor (T is a temporary workspace block) while numbers on arrows represent the ordering of the steps.

In the forward shifting technique, the destination block is first moved to a temporary memory area and then the source block is moved to the now vacant destination block. If the number of blocks in the cycle is at least four, then we need two temporary storage areas, each the size of one block. Figure 3(a) shows the forward shifting technique applied to a cycle consisting of six blocks. We start with block number 1 and its destination is block number 8, so we move block number 8 to the temporary storage area T_1. Then we move block number 1 to block number 8. Now to move block number 8 to its destination block number 17, we need to first save block number 17 to the second temporary area T_2. This process continues until we reach the block whose destination is the first block in the cycle (block number 1). For non-singleton cycles with $b > 2$ blocks, the forward shifting technique requires $2(b-1)$ block memory transfers. In our case, the cycle has $b = 6$ blocks and we need $2(6-1) = 10$ steps.

In the backward shifting technique, we begin by moving the first block to a temporary storage area T. Then we loop backward in the cycle to the block whose destination is the block we just copied. We move that block (in this case block number 6) and repeat the procedure until we have traversed the entire cycle. We end by transferring the initial block from the temporary storage area to its now vacant destination. The number of steps required by the backward shifting technique is only $b + 1$. In our case, we need $b + 1 = 7$ steps.

In conclusion we prefer the backward shifting technique not only because it uses fewer steps to shift a cycle, but also because it uses the source block in one step as the destination block in the next. If the block fits into the cache, then the data will be reused.

3.3.2 Sub-blocking

If the blocks become too large to fit in the lowest level cache, then the nice cache effects inherent in the backward shifting technique do not apply. There is a simple scheme called *sub-blocking* that can be used to overcome this issue and retain the beneficial cache behavior. The sub-blocking scheme works by partitioning each block into smaller sub-blocks that fit inside some desired level of the cache hierarchy. The backward cycle shifting is replaced by several rounds of backward cycle shifting: one round for each sub-block. Figure 4 illustrates the sub-blocking scheme with three sub-blocks per block (and hence three rounds of cycle shifting per cycle). The figure shows the sub-blocking of a cycle consisting of six blocks into three sub-blocks per block. The first round moves the dark green blocks (subscript “1”). The second round moves the green blocks (subscript “2”). Finally, the third round moves the light green blocks (subscript “3”).

3.4 Parallel conversion: Sources of concurrency

Both out-of-place and in-place conversion allow for parallel processing, but the former has a higher degree of (inherent) parallelism. In out-of-place conversion, parallelism is available when moving blocks since every block can be simultaneously copied to their respective destinations. In in-place conversion, the situation is quite different since the same memory area is used for the input and the output and the blocks can therefore not be moved simultaneously. But there is still potential for parallelism. Dependencies between blocks exist only within a given cycle. Two different cycles can be shifted at the same time and hence the cycles are a source of concurrency. To exploit the available parallelism efficiently, we need to take care of the load balance since the cycles do not all have the same length in general. Distributing

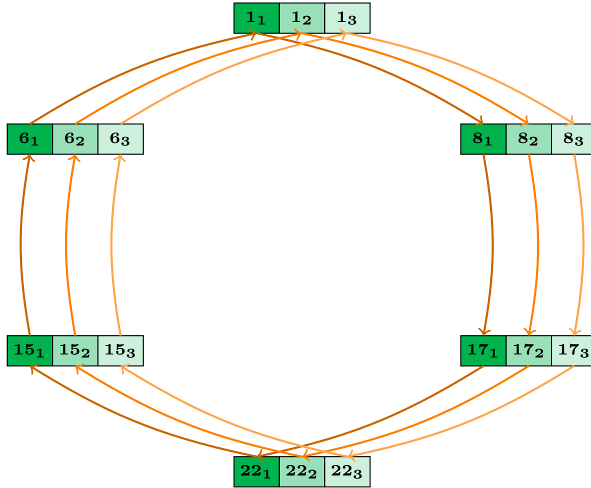


Figure 4: Illustration of the sub-blocking scheme to improve the cache behavior of the backward cycle shifting technique in in-place tensor storage format conversion. Each of the six blocks in the cycle have been partitioned into three sub-blocks. The numbers inside the blocks represent the block number in the tensor while the subscript numbers represent the sub-block number of that block.

the blocks evenly over the processors increases the scalability of the out-of-place conversion. In the in-place case, the aim is to distribute the total work evenly over the processors. Such a balanced load can in many cases be well approximated by using a dynamic load balancing scheme.

3.5 Matricization by storage format conversion

As described in Section 2.3, obtaining an explicit matricization is equivalent to converting the tensor storage format. The input is a tensor stored in a specific format defined by the permutation π_{in} and a subset $M \subseteq S(d)$ of the modes to associate with the columns of the resulting matrix. The output is the same tensor but stored in a format defined by some permutation π_{out} such that the stored tensor can be reinterpreted as a matricization of the tensor in either the row-major or the column-major format with the modes in M associated with the columns of the matrix. Figure 5 illustrates for a third-order tensor that the choice of target matrix format can drastically change the cost of the resulting tensor storage format conversion. In this example, choosing the row-major storage format will result in no change of the input tensor and is therefore entirely free. On the other hand, choosing the column-major storage format will result in a costly conversion with blocks of size one, which is the worst possible case.

Since the precise ordering of the rows and columns of the resulting matrix is seldom important in applications, there are many candidate formats π_{out}. Specifically, the matrix may be stored in either the row-major or the column-major format and the modes associated with the rows and columns may be arbitrarily ordered. The choice of output format affects the performance of the conversion process primarily because it determines the block size of the conversion as described in Section 3.1.

For the column dimension of the matrix, we need to choose an extraction σ_{col} of $S(d)$ of

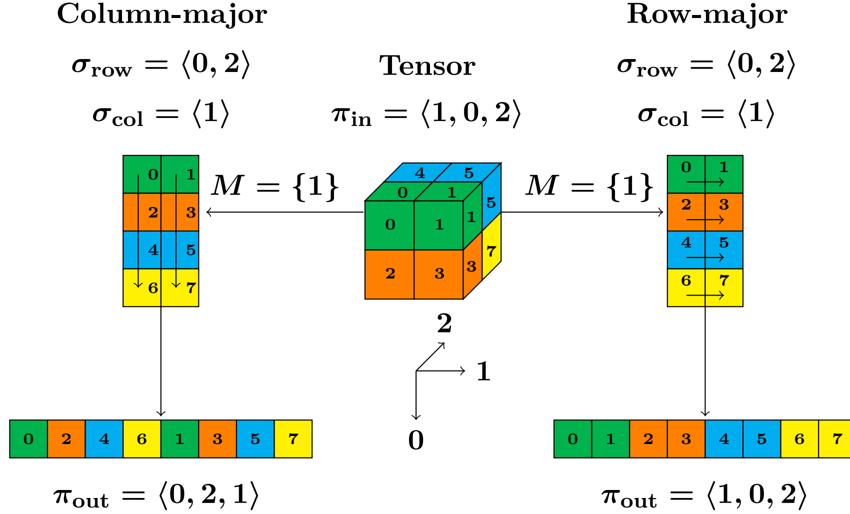


Figure 5: Tensor matricization can be performed in different ways depending on the choice of target matrix format. On the left, the matricization results in a matrix in column-major storage format but requires a very expensive tensor storage format conversion with blocks of size one. On the right, the same matricization results in a matrix in row-major storage format and requires no memory transfers at all.

length $|M|$, and for the row dimension we need a complementary extraction σ_{row} of $S(d)$ of length $|S(d) \setminus M|$. In addition, we need to choose the target matrix format: row- or column-major. The output format π_{out} is determined by these three choices as follows: if the target is the column-major format, then $\pi_{\text{out}} = \sigma_{\text{row}} \oplus \sigma_{\text{col}}$, otherwise $\pi_{\text{out}} = \sigma_{\text{col}} \oplus \sigma_{\text{row}}$.

To maximize the block size in the conversion process, we need to maximize the length of the common prefix of the given π_{in} and the chosen π_{out} subject to its constraints. The choice of target matrix format is governed only by the first component of π_{in}. If that component is in M and hence associated with the columns and a member of σ_{col}, the only way to get a block size greater than one is to choose the row-major format since that places σ_{col} first in π_{out}. Conversely, if the component is in $S(d) \setminus M$, the only reasonable choice is the column-major format. With the target matrix format fixed, the remaining components of σ_{row} and σ_{col} need to be chosen such that the length of the common prefix of π_{in} and π_{out} is maximized. For example, if $\pi_{\text{in}} = \langle 0, 1, 2, 3 \rangle$ and $M = \{1, 3\}$, then by the reasoning above we should choose the column-major format (since $0 \notin M$) and place 0 first in σ_{row}. Since the next component is in M and hence not in σ_{row}, there is no way to create a match between the second components of π_{in} and π_{out}. There are in this particular case two solutions with the same block size n_0: $\pi_{\text{out}} = \langle 0, 2, 1, 3 \rangle$ and $\pi_{\text{out}} = \langle 0, 2, 3, 1 \rangle$.

To help visualize the matricization process, we represented π_{in} as a number of O's and $\times$'s, Figure 6 (a). The O denotes a component in M and $\times$ denotes a component in $S(d) \setminus M$. The subscript numbers represent the order of the components. For example, $\times_2$ is the second component in π_{in} from $S(d) \setminus M$ and O_1 is the first component in π_{in} from M. π_{out} is also represented as a number of O's and $\times$'s but initially without subscript numbers, Figure 6 (a). The goal is to map π_{in} $\times$'s to π_{out} $\times$'s and π_{in} O's to π_{out} O's such that the block size is maximized. The mapping of a component from π_{in} to π_{out} is represented as an arrow. The arrow points to the component in π_{out} that will be numbered, Figure 6 (b,c). The length of

the common prefix is defined by the number of leading contiguous vertical arrows between π_{in} and π_{out}. In other words, the prefix size is captured by the components coming before the first tilted arrow.

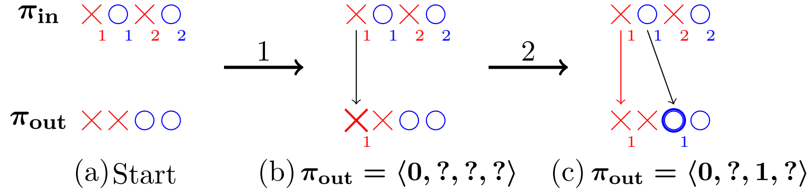


Figure 6: Matricization algorithm: block size fixed after mapping 2 components.

Using this notation, Figure 6 represents the matricization in the previous example. Since the target format is column-major format, M components come last in π_{out}. In the first step, we map the first component in π_{in}, which is $\times_1$, 0 in the example. In this case the arrow is vertical. In the next step, we map the next component in π_{in}, which is $\circ_1$. The mapping arrow is tilted in this case, which means that the block size of this matricization is now known to be n_0.

Algorithm 1 formalizes the steps required to build π_{out} in a manner that maximizes the block size.

We start with an empty $\sigma_\times$ and $\sigma_\circ$. We check the first component in π_{in}, if it is from M we append it to $\sigma_\circ$ otherwise we append it to $\sigma_\times$. Then, we move to the next component in π_{in}. We keep doing that until we have mapped all the components in π_{in}.

After the mapping, we decide the order of $\sigma_\times$ and $\sigma_\circ$ in π_{out} based on $\pi_{\text{in}}(0)$: if it's from M then $\pi_{\text{out}} = \sigma_\circ \oplus \sigma_\times$ otherwise $\pi_{\text{out}} = \sigma_\times \oplus \sigma_\circ$.

3.6 Matricizing two tensors

A common operation in tensor computation is the process of combining a subset of indices from one tensor with a subset of indices from another tensor. This is called *tensor contraction*. A contraction can be performed using matrix-matrix multiplication. In this case, the two tensors need to be converted to matrices. We matricize each tensor over its contraction subset then multiply the two matrices.

Given a tensor $\mathcal{A}$ of size $\mathbf{n}_A$ stored in the format defined by π^A_{in}, a tensor $\mathcal{B}$ of size $\mathbf{n}_B$ stored in the format defined by π^B_{in}, a subset M_A of the modes of $\mathcal{A}$, and a corresponding subset M_B of the modes of $\mathcal{B}$. We must decide on a format π^A_{out} for $\mathcal{A}$ and a *compatible* format π^B_{out} for $\mathcal{B}$. The formats are compatible if and only if the modes in M_A occur in π^A_{out} in the exact same order as their corresponding elements in M_B occur in π^B_{out}.

The constraint on the ordering of the elements of M_A and M_B in the chosen formats implies that, at least in general, we cannot find optimal formats for $\mathcal{A}$ and $\mathcal{B}$ independently. In other words, we seek an algorithm for finding an optimal pair of formats.

Given two sets of compatible formats, which is better? When matricizing a single tensor, we assumed that a bigger block size implies a faster conversion, see Section 3.5. When matricizing a pair of tensors, we have two block sizes. Let $h(\alpha, \beta)$ be some measure of the execution rate of converting $\mathcal{A}$ with block size α and $\mathcal{B}$ with block size β. We can reasonably assume that h is non-decreasing in each parameter, i.e., that $h(\alpha + \Delta, \beta) \geq h(\alpha, \beta)$ and

```
Input  : π_in // Input storage format.
       : M // Set of matricization modes.
Output : π_out // Output storage format.
       : γ // Block size.
       : σ_∘ // Extraction specifying the order of the matricization modes.
1  begin
2  |  σ_× ← ⟨⟩
3  |  σ_∘ ← ⟨⟩
4  |  for i = 0 to d − 1 do
5  |  |  if π_in(i) ∈ M then
6  |  |  |  σ_∘ ← σ_∘ ⊕ π_in(i)
7  |  |  else
8  |  |  |  σ_× ← σ_× ⊕ π_in(i)
9  |  |  end
10 |  end
11 |  if π_in(0) ∈ M then
12 |  |  π_out ← σ_∘ ⊕ σ_×
13 |  else
14 |  |  π_out ← σ_× ⊕ σ_∘
15 |  end
16 |  Compute the block size γ
17 |  return π_out, γ, σ_∘
18 end
```

Algorithm 1: Matricization algorithm.

$h(\alpha, \beta + \Delta) \geq h(\alpha, \beta)$ for every $\Delta > 0$ because increasing only one of the block sizes improves the execution rate of that conversion while the other is unaffected. See Section 4.4 for more details about h.

Consider how Algorithm 1 builds an optimal output format. It starts off with an empty output and a block size of 1. In each iteration, one additional element of the output format is determined. The block size will increase in the first few iterations until the maximum block size is reached. After that point, the block size will remain constant as the missing elements of the output format are determined. Crucially, the leading elements (which determine the block size) are uniquely determined. This means that even though there could be several output formats that have the same optimal block size, they will all have the same *prefix* that determines the optimal block size.

Now consider running Algorithm 1 on both $\mathcal{A}$ and $\mathcal{B}$ at the same time. In general, the two outputs will not be compatible. This happens if the two executions produce a different ordering for the elements in M_A and M_B. Instead, start the two executions and run them until the first iteration that wants to map an element in M_A respectively M_B. Pause both executions. At this point, the two partially defined formats are compatible. From this point forward there are two cases, Figure 7. In the best case, the two elements that are about to be mapped are in correspondence with one another, Figure 7(a). In this case, we simply advance both executions to the next iteration; both block sizes will be increased and the formats will remain compatible. In the worst case, the two elements are not in correspondence with one another, Figure 7(b). In that case, we must accept that one of the block sizes need to be fixed in order to preserve compatibility. There are again two cases: we fix the block size of either $\mathcal{A}$ or $\mathcal{B}$. Let α and β denote the block sizes obtained thus far by Algorithm 1. If we fix the block size α, then we can proceed with the execution on $\mathcal{B}$ and obtain the block size $\beta + \Delta_B$ for $\mathcal{B}$. On the other hand, if we fix the block size β, then we can proceed with the execution on $\mathcal{A}$ and obtain the block size $\alpha + \Delta_A$ for $\mathcal{A}$. Which is better depends on h: if $h(\alpha + \Delta_A, \beta) < h(\alpha, \beta + \Delta_B)$, then we should fix α and otherwise we should fix β.

In conclusion, to compute an optimal pair of formats we use the following procedure:

1. Run Algorithm 1 on $\mathcal{A}$. Let α_1 denote the resulting block size.

2. Run Algorithm 2, a modified version of Algorithm 1, on $\mathcal{B}$ that produces a format compatible with that produced in Step 1. Let β_1 denote the resulting block size.

3. Run Algorithm 1 on $\mathcal{B}$. Let β_2 denote the resulting block size.

4. Run Algorithm 2 on $\mathcal{A}$ to ensure compatibility with the format produced in Step 3. Let α_2 denote the resulting block size.

5. If $h(\alpha_1, \beta_1) \geq h(\alpha_2, \beta_2)$ then use the formats from Steps 1 and 2 and otherwise use the formats from Steps 3 and 4.

Algorithm 3 formalizes the steps required to build π^A_{out} and π^B_{out} in a manner that maximizes the execution rate.

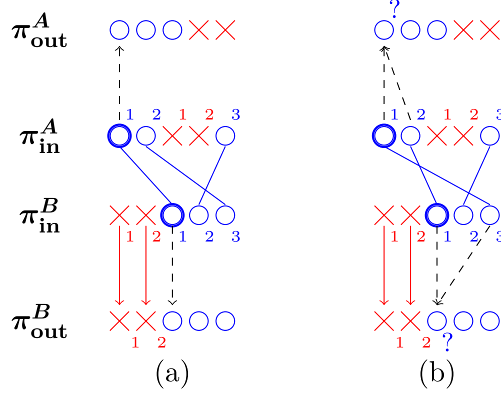


Figure 7: Different scenarios for mapping an element from M.

Input : π_{in} `// Input storage format for tensor 2.`
: M_1 `// Set of matricization modes of tensor 1.`
: M_2 `// Set of matricization modes of tensor 2.`
: $\sigma^1_\circ$ `// Extraction specifying the order of tensor 1 matricization modes.`
Output: π_{out} `// Output storage format for tensor 2.`
: γ `// Block Size for tensor 2.`

1 **begin**
2 $\quad \sigma_\times \leftarrow \langle\rangle$
3 $\quad$ Using $\sigma^1_\circ$ define an extraction $\sigma_\circ$ that orders the elements of M_2 which keeps the correspondence between M_1 and M_2
4 $\quad$ **for** $i = 0$ **to** $d - 1$ **do**
5 $\quad\quad$ **if** $\pi_{in}(i) \notin M_2$ **then**
6 $\quad\quad\quad \sigma_\times \leftarrow \sigma_\times \oplus \pi_{in}(i)$
7 $\quad\quad$ **end**
8 $\quad$ **end**
9 $\quad$ **if** $\pi_{in}(0) \in M_2$ **then**
10 $\quad\quad \pi_{out} \leftarrow \sigma_\circ \oplus \sigma_\times$
11 $\quad$ **else**
12 $\quad\quad \pi_{out} \leftarrow \sigma_\times \oplus \sigma_\circ$
13 $\quad$ **end**
14 $\quad$ Compute the block size γ
15 $\quad$ **return** π_{out}, γ
16 **end**

Algorithm 2: Modified version of Matricization algorithm.

Input : π_{in}^{A} `// Input storage format of tensor A.`
: π_{in}^{B} `// Input storage format of tensor B.`
: M_A `// Set of matricization modes of tensor A.`
: M_B `// Set of matricization modes of tensor B.`
Output: π_{out}^{A} `// Output storage format of tensor A.`
: π_{out}^{B} `// Output storage format of tensor B.`

1 **begin**
2 $(\pi_{out}^{A|A}, \gamma_{A|A}, \sigma_{\circ}^{A|A}) \leftarrow$ `Algorithm 1`(π_{in}^{A}, M_A)
3 $(\pi_{out}^{B|A}, \gamma_{B|A}) \leftarrow$ `Algorithm 2`$(\pi_{in}^{B}, \sigma_{\circ}^{A|A}, M_B, M_B)$
4 $(\pi_{out}^{B|B}, \gamma_{B|B}, \sigma_{\circ}^{B|B}) \leftarrow$ `Algorithm 1`(π_{in}^{B}, M_B)
5 $(\pi_{out}^{A|B}, \gamma_{A|B}) \leftarrow$ `Algorithm 2`$(\pi_{in}^{A}, \sigma_{\circ}^{B|B}, M_B, M_A)$
6 **if** $h(\gamma_{A|A}, \gamma_{B|A}) < h(\gamma_{A|B}, \gamma_{B|B})$ **then**
7 $\pi_{out}^{A} \leftarrow \pi_{out}^{A|B}$
8 $\pi_{out}^{B} \leftarrow \pi_{out}^{B|B}$
9 **else**
10 $\pi_{out}^{A} \leftarrow \pi_{out}^{A|A}$
11 $\pi_{out}^{B} \leftarrow \pi_{out}^{B|A}$
12 **end**
13 **return** $\pi_{out}^{A}, \pi_{out}^{B}$
14 **end**

Algorithm 3: Matricize-pair algorithm.

4 Software

A software library called `dten` has been written in the C programming language with the OpenMP extension used to parallelize the core functions.

4.1 Features

Presently, `dten` contains two levels of functionality: basic and advanced. The basic functionality consists of essential functions for allocating, deallocating, initializing, printing, copying, and saving/loading to/from an HDF5-based file format [24]. In addition, primitive functions for obtaining information about the tensor such as its order and size are provided. The advanced functionality consists of parallel tensor storage format conversion and wrappers for efficient matricization. Both out-of-place (OOP) and in-place (IP) conversion functions are provided. In the out-of-place function, the user can choose which tensor (input or output) to traverse contiguously. The allocation and initialization of the output tensor in the out-of-place conversion is the responsibility of the user to potentially save the allocation and deallocation time, which may be a factor affecting the conversion performance specially for large tensors.

The matricization functionality, as described in Section 3.5, is split into three functions. The first matricizes over a single mode, $|M| = 1$. The second matricizes over a subset of the modes, $|M| > 1$. The third matricizes a pair of tensors such that a contraction over a subset of the modes can be performed afterwards using standard matrix–matrix multiplication routines. For each matricization function, the user can specify the target matrix format as

well as the ordering of the modes associated with the rows and/or the columns or choose to leave one or more of these choices to the library.

In addition, if no initial permutation of a tensor is supplied to the allocation, then the library tries to maximize the potential block size by ordering the indices based on their size in a descending order.

4.2 Tunable parameters

The library contains a few parameters that affect the performance of some of the functions and can be tuned to give the best performance on a particular machine. As mentioned in Section 3.4, the parallelism in the in-place conversion function is done by shifting multiple cycles in parallel. The cycles need to be identified first and since the number of cycles is potentially very large, the number of cycles to generate before shifting them in parallel is bounded by a tunable parameter. The cycle cache should be large enough to enable effective load balancing (i.e., much larger than the number of threads), but not too big as to waste a lot of memory. The effect on performance is negligible unless the cache is very small in which case there will not be enough cycles to parallelize, leading to idle threads.

Another parameter is the size of the sub-blocks as described in Section 3.3.2. This parameter affects the core of the conversion function and has a large impact on the performance. The optimal choice depends on the size and characteristics of the memory hierarchy.

4.3 Cycle shifting strategy

As mentioned in Section 3.3.1, there are two ways of shifting a cycle in the in-place technique. The backward shifting technique is used due to the advantages explained in Section 3.3.1.

4.4 Matricize-pair heuristic function

The matricize-pair algorithm described in Section 3.6 uses a heuristic function h to find which tensor to optimize. The heuristic function takes as an input the block sizes, α and β, and returns an estimate of the execution rate when using these two block sizes. For a given set of formats and corresponding block sizes, we assume that the execution rates of the two conversions are limited by the smallest block size among the two; so the heuristic function returns the smallest block size, i.e., $h(\alpha, \beta) = \min\{\alpha, \beta\}$, as an estimate of the execution rate. In Algorithm 3 we get two sets of compatible formats to matricize two tensors together. We evaluate the two sets and we choose the one with the highest execution rate, i.e., the one that gives the highest value for the heuristic function. In case the execution rates are equal, we use the following tie-breaker:

1. Choose the set which leads to applying the smallest block size on the shorter tensor.

2. If both tensors are equally large, maximize the largest block size.

5 Performance

This section presents the performance and the scalability of the conversion function. The experiments were performed on one node of the high performance computer Abisko, which is operated by High Performance Computing Center North (HPC2N) at Umeå University.

One node consists of four AMD Opteron 6238 processors clocked at 2.6 GHz. Each processor contains two chips with six cores each for a total of 48 cores per node. Each chip has its own memory controller, which leads to eight NUMA domains per node. Each group of six cores share a memory bus on Abisko, for that reason the number of cores in our tests are multiple of six.

The main factor affecting the conversion performance is the block size. In addition, the performance of the OOP conversion is affected by which tensor will be accessed contiguously. While the performance of the IP conversion is affected by whether the sub-blocking is used or not.

Figure 8 shows the change in memory bandwidth with respect to changing the block size. The figure contains different plots for four different cases, OOP with contiguous access to the output tensor, OOP with contiguous access to the input tensor, IP without sub-blocking and IP with sub-blocking. The four cases were tested on 48 cores. The tensor chosen as a study case is of order 6 with size $\mathbf{n} = \langle x, 8, 4, 4, 5, 2 \rangle$, where x was changed to give different block sizes. The initial storage format was defined by $\phi_{\pi_{in}}(\mathbf{k}; \mathbf{n})$ where $\pi_{in} = \langle 0, 1, 2, 3, 4, 5 \rangle$ and the target storage format was defined by $\phi_{\pi_{out}}(\mathbf{k}; \mathbf{n})$ where $\pi_{out} = \langle 0, 3, 2, 1, 4, 5 \rangle$. The conversion contains 200 cycles, most of them involve moving 7 blocks while the rest are singleton cycles. The sub-blocking size used for the IP conversion is 8KB. The tested block sizes were 8KB, 16KB, 32KB, ..., 3.2MB.

The aggregate cache for the 48 cores on one node of Abisko is 96MB. If the tensor size is more than that it will not fit into the cache. This explains why there is a drop in the memory bandwidth for tensors of sizes larger than the aggregate cache size. Recall that OOP conversion doubles the memory used for tensor storage, see Section 3.2.

It is clear from Figure 8 that accessing either tensor contiguously is not affecting the performance dramatically for the OOP conversion. On the other hand, the IP conversion with sub-blocking improves the performance drastically compared to omitting the sub-blocking for large size tensors. These results influenced us to use OOP conversion with input tensor accessed contiguously and IP conversion with sub-blocking for the rest of the conducted experiments.

To study the scalability of the conversion functions, different tensors are tested. The tensor sizes were taken from the previous described experiment. We chose the case where the aggregate cache is almost full, which is 40MB tensor for the OOP conversion and 80MB tensor for the IP conversion. We will call this case the cache fit case. In addition, 10MB, 640MB and 4GB tensors were tested for both OOP and IP conversion. The real factor that affects the scalability is the memory buses.

Figures 9 and 10 present the scalability of the OOP conversion and the IP conversion functions, respectively. The figures show the number of cores versus the efficiency E given by

$$E = \frac{t_s}{p \times t_p}, \tag{4}$$

where t_s is the sequential execution time, t_p is the parallel execution time and p is number of cores. Both figures show acceptable efficiency for large tensor sizes. The cache fit case gives slightly better performance due to efficient use of cache memory while the small tensor case behave wildly because it is too small to be parallelized.

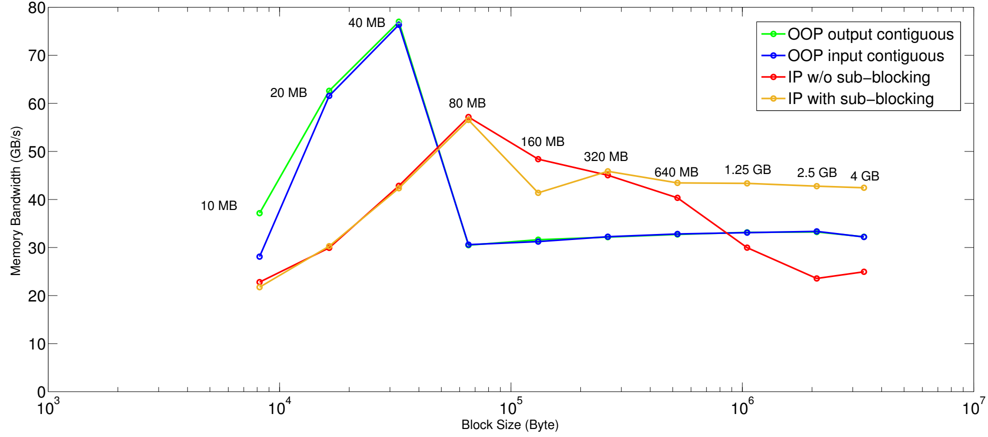


Figure 8: The effect of block size on performance.

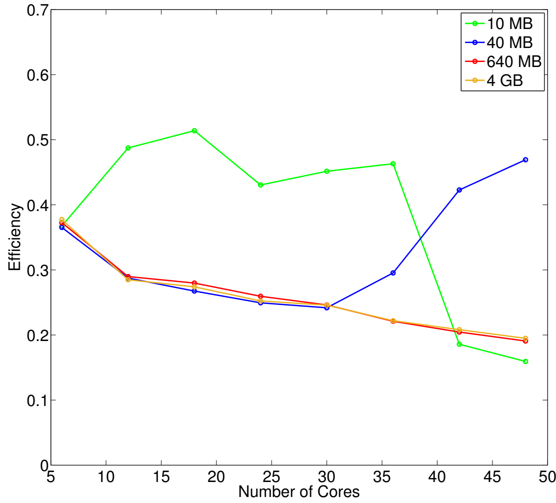


Figure 9: The scalability of OOP conversion.

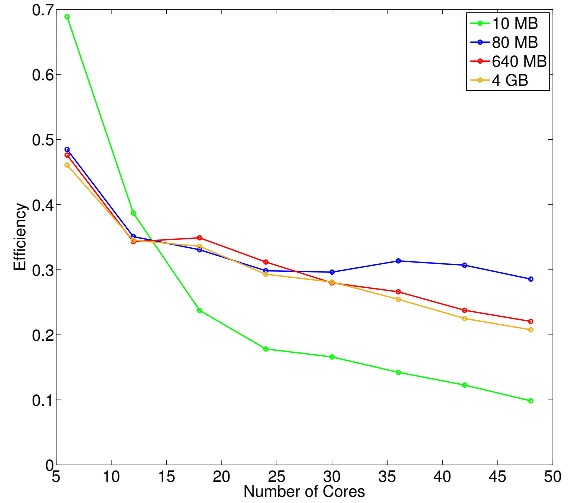


Figure 10: The scalability of IP conversion.

6 Conclusion and related work

An n-dimensional tensor has $n!$ canonical tensor storage formats. Converting a tensor from one format to another can, in many cases, be done efficiently by transferring memory blocks. But sometimes the blocks can degenerate and consist of a single element. The maximum block size is determined by the pair of formats and the size of the tensor. Putting the tensor dimensions in a descending order of size can maximize the potential block size.

Converting a tensor format can be done using an out-of-place technique or an in-place technique. The former uses another memory location to perform the conversion where the latter shifts the tensor blocks within the same memory in cycles. Also the in-place technique requires almost half of the memory but at the expense of exploiting lower degree of inherent parallelism.

We showed that shifting cycles in the in-place technique can be done in two ways, from which the backward shifting is chosen because, compared to the forward shifting, it is more cache efficient and requires less steps. In addition, to benefit from the memory hierarchy and be more cache efficient, the blocks could be divided into sub-blocks in the in-place conversion.

Furthermore, tensor matricization can always be performed using non-degenerate blocks if the output matrix format (row- or column-major) can be chosen freely.

6.1 Related work

Some work has been done in tensor computation and some tools are presented which target storage format of dense tensors. The MATLAB Tensor Toolbox [4] provides a set of tensor related functions such as tensor multiplication, matricization and various tensor decompositions. Another MATLAB toolbox is Tensorlab [25] which supports complex optimization, tensor factorization and tensor optimization. While the MATLAB toolbox TT-Toolbox [22] provides basic tensor operations for tensors stored in tensor-train format. The python library Scikit-tensor [21] supports basic tensor operations and factorization. A famous software for symbolic computation, Wolfram Mathematica [1], represents tensors as a list of lists. The software supports many tensor operations and tensor related algorithms. In Torch7 [7], a scientific computing framework for machine learning on GPU, a tensor represents a view to a storage. The data in the tensor may not be contiguous in memory. Yet, the framework provides tools for manipulating and rearranging the data in storage. In [15] Albert Hartono et al. present a way for permuting the indices of a tensor by generating multiple code versions optimized during the library installation stage. Another automatic code generator widely used is So Hirata's Tensor Contraction Engine (TCE) [16]. While the main focus is to generate optimized code for tensor contraction, Hirata proposed a way for tensor permutation where the tensor is divided into tiles and the position of the tile for each required permutation during a contraction is precomputed and stored in the memory. Ballared G. et al. in [5] address the problem of symmetric tensors storage format in their proposal for computing tensor eigenvalues on GPU. While Beverly A. Sanders et al. in [23] are focusing more on distributed memory. Also related to distributed memory, Austin W. et al. in [3] propose to distribute the tensor among a processor grid and perform the matricization logically without data movement. While Dmitry I. Lyakh in [20] is considering the case where the block size is one.

Acknowledgments

The work was supported by eSSENCE, a strategic collaborative e-Science program funded by the Swedish Government via the Swedish Research Council (VR). The authors thank the High Performance Computing Center North (HPC2N) for providing computational resources and valuable support during test and performance runs.

References

[1] Wolfram Mathematica: symbolic computation software. https://www.wolfram.com/mathematica/.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] W. Austin, G. Ballard, and T. Kolda. Parallel tensor compression for large-scale scientific data. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 912–922. IEEE, 2016.

[4] B. W. Bader and T. G. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, December 2006.

[5] G. Ballard, T. Kolda, and T. Plantenga. Efficiently computing tensor eigenvalues on a GPU. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium*, pages 1340–1348. IEEE, 2011.

[6] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[7] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[8] T. Dahlberg. *Compact Representation and Efficient Manipulation of Sparse Multidimensional Arrays*. Bachelor thesis, Umeå University, Department of Computing Science, 2014.

[9] J. Dongarra. Basic linear algebra subprograms technical (blast) forum standard. *International Journal of High Performance Computing Applications*, 16(1,2):1–111,115–199, 2002.

[10] J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[11] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: An extended set of basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Softw.*, 14(1):18–32, March 1988.

[12] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, March 1988.

[13] J. J. Dongarra, J. Du Cruz, S. Hammerling, and I. S. Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Softw.*, 16(1):18–28, March 1990.

[14] F. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software (TOMS)*, 38(3):17, 2012.

[15] A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D.E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *The Journal of Physical Chemistry A*, 113(45):12715–12723, 2009.

[16] S. Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[17] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. *ACM Trans. Math. Software*, 24(3):268–302, 1998.

[18] B. Kågström, P. Ling, and C. Van Loan. Algorithm 784: GEMM-Based Level 3 BLAS: Portability and Optimization Issues. *ACM Trans. Math. Software*, 24(3):303–316, 1998.

[19] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[20] D. I. Lyakh. An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. *Computer Physics Communications*, 189:84 – 91, 2015.

[21] M. Nickel. Scikit-tensor: Python library for multilinear algebra and tensor factorizations. https://github.com/mnick/scikit-tensor.

[22] I. Oseledets, S. Dolgov, V. Kazeev, O. Lebedeva, and T. Mach. TT-Toolbox 2.2, 2012. http://spring.inm.ras.ru/osel.

[23] B. A. Sanders, R. Bartlett, E. Deumens, V. Lotrich, and M. Ponton. A block-oriented language and runtime system for tensor algebra with very large arrays. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010.

[24] The HDF Group. Hierarchical Data Format, version 5, 1997-2017. http://www.hdfgroup.org/HDF5/.

[25] N. Vervliet, O. Debals, and L. De Lathauwer. Tensorlab 3.0—numerical optimization strategies for large-scale constrained and coupled matrix/tensor factorization. In *2016 Conference Record of the 50th Asilomar Conference on Signals, Systems and Computers. IEEE*, 2016.