EXPLOITING SYMMETRY IN TENSORS FOR HIGH PERFORMANCE

MARTIN D. SCHATZ[†], TZE MENG LOW[†], ROBERT A. VAN DE GEIJN[‡], AND TAMARA G. KOLDA[§]

Abstract. Symmetric tensor operations arise in a wide variety of computations. However, the benefits of exploiting symmetry in order to reduce storage and computation is in conflict with a desire to simplify memory access patterns. In this paper, we propose Blocked Compact Symmetric Storage wherein we consider the tensor by blocks and store only the unique blocks of a symmetric tensor. We propose an algorithm-by-blocks, already shown of benefit for matrix computations, that exploits this storage format. A detailed analysis shows that, relative to storing and computing with tensors without taking advantage of symmetry, storage requirements are reduced by a factor $O(m!)$ and computational requirements by a factor $O(m)$, where m is the order of the tensor. An implementation demonstrates that the complexity introduced by storing and computing with tensors by blocks is manageable and preliminary results demonstrate that computational time is indeed reduced. The paper concludes with a discussion of how the insights point to opportunities for generalizing recent advances for the domain of linear algebra libraries to the field of multi-linear computation.

1. Introduction.

A tensor is a multi-dimensional or m-array. Tensor computations are increasingly prevalent in a wide variety of applications [15]. Alas, libraries for dense multi-linear algebra (tensor computations) are in their infancy. The aim of this paper is to explore how ideas from matrix computations can be extended to the domain of tensors.

Libraries for dense linear algebra (matrix computations) have long been part of the standard arsenal for computational science, including the Basic Linear Algebra Subprograms interface [16, 10, 9, 12, 11], LAPACK [2], and more recent libraries with similar functionality, like `libflame` [29, 25], and the BLAS-like Interface Software framework (BLIS) [26]. For distributed memory architectures, the ScaLAPACK [8], PLAPACK [24], and (more recently) Elemental [20] libraries provide most of the functionality of the BLAS and LAPACK. High-performance implementations of these libraries are available under open source licenses.

For tensor computations, no high-performance general-purpose libraries exist. The MATLAB Tensor Toolbox [4, 3] defines many of the kernels (i.e., the interface) that would be needed by a library for multilinear algebra, but does not have any high-performance kernels nor special computations or data structure for symmetric tensors. The Tensor Contraction Engine (TCE) project [6] focuses on sequences of tensor contractions and uses compiler techniques to reduce workspace and operation counts. The very recent Cyclops Tensor Framework (CTF) [23] focuses on exploiting symmetry in storage for distributed memory parallel computation with tensors, but does not investigate the sequential libraries that will invariably be needed on the individual nodes.

In a talk at the Eighteenth Householder Symposium meeting in Tahoe City, CA (2011), Prof. Charlie Van Loan stated "In my opinion, blocking will eventually have the same impact in tensor computations as it does in matrix computations." This paper provides early evidence in support of this statement. The approach we take in this

[†] Department of Computer Science, The University of Texas at Austin, Austin, TX. Email: martin.schatz@utexas.edu.

[‡] Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX. Email: rvdg@cs.utexas.edu.

[§] Sandia National Laboratories, Livermore, CA. Email: tgkolda@sandia.gov.

paper heavily borrows from the FLAME project [25]. We use the *change-of-basis* operation, also known as a Symmetric Tensor Times Same Matrix (in all modes) (`sttsm`) operations [4], to motivate the issues and solutions. We propose algorithms that require minimal computation by computing and storing temporaries. Additionally, the tensors are stored by blocks, following similar solutions developed for matrices [17, 22]. The algorithms are reformulated to operate with these blocks. Since we need only store the unique blocks, symmetry is exploited at the block level (both for storage and computation) while preserving regularity when computing within blocks. The paper analyzes the computational and storage costs, demonstrating that the simplicity need not adversely impact the benefits derived from symmetry. An implementation shows that the insights can be made practical. The paper concludes by listing opportunities for future research.

2. Preliminaries.

We start with some basic notation, definitions, and the motivating tensor operation.

2.1. Notation.

In this discussion, we assume all indices and modes are numbered starting at zero.

The *order* of a tensor is the number of ways or modes. In this paper, we deal only with tensors where every mode has the same dimension. Therefore, we define $\mathbb{R}^{[m,n]}$ to be the set of real-valued order-m tensors where each mode has dimension n, i.e., a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[m,n]}$ can be thought of as an m-dimensional cube with n entries in each direction. As the order of a tensor corresponds to the number of ways or modes of a tensors, we sometimes refer to an order-m tensor as an m-way tensor.

An element of $\boldsymbol{\mathcal{A}}$ is denoted as $\alpha_{i_0 \cdots i_{m-1}}$ where $i_k \in \{0, \ldots, n-1\}$ for all $k \in \{0, \ldots, m-1\}$. This also illustrates that, as a general rule, we use lower case Greek letters for scalars $(\alpha, \chi, \ldots)$, bold lower case Roman letters for vectors $(\mathbf{a}, \mathbf{x}, \ldots)$, bold upper case Roman letters for matrices $(\mathbf{A}, \mathbf{X}, \ldots)$, and upper case scripted letters for tensors $(\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{X}}, \ldots)$. We denote the ith row of a matrix $\mathbf{A}$ by $\widehat{\boldsymbol{a}}_i^T$. If we transpose this row, we denote it as $\widehat{\boldsymbol{a}}_i$.

2.2. Symmetric tensors.

A symmetric tensor is a tensor that is invariant under any permutation of indices.

Definition 1 (Symmetric tensor). *A tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[m,n]}$ is symmetric if its entries remain constant under any permutation of the indices, i.e.,*

$$\alpha_{i'_0 \cdots i'_{(m-1)}} = \alpha_{i_0 \cdots i_{(m-1)}} \quad \forall i_j \in \{0, \ldots, n-1\} \text{ for } j = 0, \ldots, m-1, \text{ and } \pi \in \Pi_m,$$

where i' is result of applying the permutation π to the index i, and Π_m is the set of all permutations of the set $\{0, \ldots, m-1\}$.

For example, an order-2 symmetric tensor is nothing more than a symmetric matrix. Similarly, an example of a symmetric order-3 tensor is a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[3,n]}$ with entries that satisfy $\alpha_{i_0 i_1 i_2} = \alpha_{i_0 i_2 i_1} = \alpha_{i_1 i_0 i_2} = \alpha_{i_1 i_2 i_0} = \alpha_{i_2 i_0 i_1} = \alpha_{i_2 i_1 i_0}$ for all $i_0, i_1, i_2 \in \{0, \ldots, n-1\}$.

Since most of the entries in a symmetric tensor are redundant, we need only store a small fraction to fully describe and compute with a given tensor. Specifically, the number of unique entries of a symmetric tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[m,n]}$ is [5]

$$\binom{n+m-1}{m} \approx \frac{n^m}{m!} + O(n^{m-1}).$$

Hence, we can reduce the storage by approximately $m!$ by exploiting symmetry.

2.3. The sttsm operation. The operation used in this paper to illustrate issues related to storage of, and computation with, symmetric tensors is the *change-of-basis* operation

$$\mathcal{C} := [\boldsymbol{\mathcal{A}}; \underbrace{\mathbf{X}, \cdots, \mathbf{X}}_{m \text{ times}}], \tag{2.1}$$

where $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[m,n]}$ is symmetric and $\mathbf{X} \in \mathbb{R}^{p \times n}$ is the change-of-basis matrix. This is equivalent to multiplying the tensor $\boldsymbol{\mathcal{A}}$ by the same matrix $\mathbf{X}$ in every mode. The resulting tensor $\mathcal{C} \in \mathbb{R}^{[m,p]}$ is defined elementwise as

$$\gamma_{j_0 \cdots j_{m-1}} := \sum_{i_0=0}^{n-1} \cdots \sum_{i_{m-1}=0}^{n-1} \alpha_{i_0 \cdots i_{m-1}} \beta_{j_0 i_0} \beta_{j_1 i_1} \cdots \beta_{j_{m-1} i_{m-1}}.$$

where $j_k \in \{0, \ldots, p-1\}$ for all $k \in \{0, \ldots, m-1\}$. It can be observed that the resulting tensor $\mathcal{C}$ is itself symmetric. We refer to this operation (2.1) as the sttsm operation.

3. The Matrix Case.

We will build intuition about the problem and its solutions by first looking at order-2 symmetric tensors.

3.1. The operation. Letting $m = 2$ yields $\mathbf{C} := [\mathbf{A}; \mathbf{X}, \mathbf{X}]$ where $\mathbf{A} \in \mathbb{R}^{[m,n]}$ is an $n \times n$ symmetric matrix, $\mathbf{C} \in \mathbb{R}^{[m,p]}$ is a $p \times p$ symmetric matrix, and $[\mathbf{A}; \mathbf{X}, \mathbf{X}] = \mathbf{X}\mathbf{A}\mathbf{X}^T$. For $m = 2$, (2.1) becomes

$$\begin{aligned}\gamma_{j_0 j_1} = \widehat{\boldsymbol{x}}_{j_0}^T \mathbf{A} \widehat{\boldsymbol{x}}_{j_1} = \sum_{i_0=0}^{n-1} \chi_{j_0 i_0} (\mathbf{A}\widehat{\boldsymbol{x}}_{j_1})_{i_0} &= \sum_{i_0=0}^{n-1} \chi_{j_0 i_0} \sum_{i_1=0}^{n-1} \alpha_{i_0 i_1} \chi_{j_1 i_1} \\ &= \sum_{i_0=0}^{n-1} \sum_{i_1=0}^{n-1} \alpha_{i_0 i_1} \chi_{j_0 i_0} \chi_{j_1 i_1}.\end{aligned} \tag{3.1}$$

3.2. Simple algorithms. Based on (3.1), a naive algorithm that only computes the upper triangular part of symmetric matrix $\mathbf{C} = \mathbf{X}\mathbf{A}\mathbf{X}^T$ is given in Figure 3.1 (left), at a cost of approximately $3p^2n^2$ floating point operations (flops). The algorithm to its right reduces flops by storing intermediate results and taking advantage of symmetry. It is motivated by observing that

$$\begin{aligned}\mathbf{X}\mathbf{A}\mathbf{X}^T &= \mathbf{X} \underbrace{\mathbf{A}\mathbf{X}^T}_{\mathbf{T}} \\ &= \begin{pmatrix} \widehat{\boldsymbol{x}}_0^T \\ \vdots \\ \widehat{\boldsymbol{x}}_{p-1}^T \end{pmatrix} \underbrace{\mathbf{A} \begin{pmatrix} \widehat{\boldsymbol{x}}_0 & \cdots & \widehat{\boldsymbol{x}}_{p-1} \end{pmatrix}}_{\begin{pmatrix} \widehat{\boldsymbol{t}}_0 & \cdots & \widehat{\boldsymbol{t}}_{p-1} \end{pmatrix}} = \begin{pmatrix} \widehat{\boldsymbol{x}}_0^T \\ \vdots \\ \widehat{\boldsymbol{x}}_{p-1}^T \end{pmatrix} \begin{pmatrix} \widehat{\boldsymbol{t}}_0 & \cdots & \widehat{\boldsymbol{t}}_{p-1} \end{pmatrix},\end{aligned} \tag{3.2}$$

where $\widehat{\boldsymbol{t}}_j = \mathbf{A}\widehat{\boldsymbol{x}}_j \in \mathbb{R}^n$ and $\widehat{\boldsymbol{x}}_j \in \mathbb{R}^n$ (recall that $\widehat{\boldsymbol{x}}_j$ denotes the transpose of the jth row of $\mathbf{X}$). This algorithm requires approximately $2pn^2 + p^2n$ flops at the expense of requiring temporary space for a vector $\widehat{\boldsymbol{t}}$.

Naive algorithms	Algorithms that reduce computation at the expense of temporary space

$\boldsymbol{\mathcal{A}}$ is a matrix ($m = 2$): $\mathbf{C} := \mathbf{X}\mathbf{A}\mathbf{X}^T = [\mathbf{A}; \mathbf{X}, \mathbf{X}]$

```
for $j_1 = 0, \ldots, p-1$
  for $j_0 = 0, \ldots, j_1$
    $\gamma_{j_0 j_1} := 0$
    for $i_0 = 0, \ldots, n-1$
      for $i_1 = 0, \ldots, n-1$
        $\gamma_{j_0 j_1} := \gamma_{j_0 j_1} + \alpha_{i_0 i_1} \chi_{j_0 i_0} \chi_{j_1 i_1}$
      endfor
    endfor
  endfor
endfor
```

```
for $j_1 = 0, \ldots, p-1$
  $\widehat{\boldsymbol{t}} := \widehat{\boldsymbol{t}}_{j_1} = \mathbf{A}\widehat{\boldsymbol{x}}_{j_1}$
  for  $j_0 = 0, \ldots, j_1$
    $\gamma_{j_0 j_1} := \widehat{\boldsymbol{x}}_{j_0}^T \widehat{\boldsymbol{t}}$
  endfor
endfor
```

$\boldsymbol{\mathcal{A}}$ is a 3-way tensor ($m = 3$): $\boldsymbol{\mathcal{C}} := [\boldsymbol{\mathcal{A}}; \mathbf{X}, \mathbf{X}, \mathbf{X}]$

```
for $j_2 = 0, \ldots, p-1$
  for $j_1 = 0, \ldots, j_2$
    for $j_0 = 0, \ldots, j_1$
      $\gamma_{j_0 j_1 j_2} := 0$
      for $i_2 = 0, \ldots, n-1$
        for $i_1 = 0, \ldots, n-1$
          for $i_0 = 0, \ldots, n-1$
            $\gamma_{j_0 j_1 j_2} +:=$
                $\alpha_{i_0 i_1 i_2} \chi_{j_0 i_0} \chi_{j_1 i_1} \chi_{j_2 i_2}$
          endfor
      $\iddots$
endfor
```

```
for $j_2 = 0, \ldots, p-1$
  $\mathbf{T}^{(2)} := \mathbf{T}_{j_2}^{(2)} = \boldsymbol{\mathcal{A}} \times_2 \widehat{\boldsymbol{x}}_{j_2}^T$
  for $j_1 = 0, \ldots, j_2$
    $\widehat{\boldsymbol{t}}^{(1)} := \widehat{\boldsymbol{t}}_{j_1 j_2}^{(1)} = \mathbf{T}^{(2)} \times_1 \widehat{\boldsymbol{x}}_{j_1}^T$
    for $j_0 = 0, \ldots, j_1$
      $\gamma_{j_0 j_1 j_2} := \widehat{\boldsymbol{t}}^{(1)} \times_0 \widehat{\boldsymbol{x}}_{j_0}^T$
    endfor
  endfor
endfor
```

$\boldsymbol{\mathcal{A}}$ is an m-way tensor: $\boldsymbol{\mathcal{C}} := [\boldsymbol{\mathcal{A}}; \mathbf{X}, \cdots, \mathbf{X}]$

```
for $j_{m-1} = 0, \ldots, p-1$
  $\ddots$
    for $j_0 = 0, \ldots, j_1$
      $\gamma_{j_0 \cdots j_{m-1}} := 0$
      for $i_{m-1} = 0, \ldots, n-1$
        $\ddots$
          for $i_0 = 0, \ldots, n-1$
            $\gamma_{j_0 \cdots j_{m-1}} +:=$
                $\alpha_{i_0 \cdots i_2} \chi_{j_0 i_0} \cdots \chi_{j_{m-1} i_{m-1}}$
          endfor
      $\iddots$
endfor
```

```
for $j_{m-1} = 0, \ldots, p-1$
  $\boldsymbol{\mathcal{T}}^{(m-1)} := \boldsymbol{\mathcal{T}}_{j_{m-1}} = \boldsymbol{\mathcal{A}} \times_{m-1} \widehat{\boldsymbol{x}}_{j_{m-1}}^T$
    $\ddots$
    for $j_1 = 0, \ldots, j_2$
      $\widehat{\boldsymbol{t}}^{(1)} := \widehat{\boldsymbol{t}}_{j_1 \cdots j_{m-1}} = \mathbf{T}^{(2)} \times_1 \widehat{\boldsymbol{x}}_{j_1}^T$
      for $j_0 = 0, \ldots, j_1$
        $\gamma_{j_0 \cdots j_{m-1}} := \widehat{\boldsymbol{t}}^{(1)} \times_0 \widehat{\boldsymbol{x}}_{j_0}^T$
      endfor
    endfor
    $\iddots$
endfor
```

Fig. 3.1. *Algorithms for $\boldsymbol{\mathcal{C}} := [\boldsymbol{\mathcal{A}}; \mathbf{X}, \cdots, \mathbf{X}]$ that compute with scalars. In order to facilitate the comparing and contrasting of algorithms, we present algorithms for the special cases where $m = 2, 3$ (top and middle) as well as the general case (bottom). For each, we give the naive algorithm on the left and the algorithm that reduces computation at the expense of temporary storage on the right.*

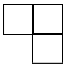
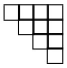
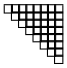
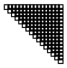
Relative storage of BCSS		$\bar{n} = \lceil n/b_{\mathbf{A}} \rceil$			
		2 	4 	8 	16 
relative to minimal storage	$\dfrac{n(n+1)/2}{n(n+b_{\mathbf{A}})/2}$	0.67	0.80	0.89	0.94
relative to dense storage	$\dfrac{n^2}{n(n+b_{\mathbf{A}})/2}$	1.33	1.60	1.78	1.88

FIG. 3.2. *Storage savings factor of BCSS when $n = 512$.*

3.3. Blocked Compact Symmetric Storage (BCSS).

Since matrices $\mathbf{C}$ and $\mathbf{A}$ are symmetric, it saves space to store only the upper (or lower) triangular part of those matrices. We will consider storing the upper triangular part. While for matrices the savings is modest (and rarely exploited), the savings is more dramatic as the tensor order increases.

To store a symmetric matrix, consider packing the elements of the upper triangle tightly into memory with the following ordering of unique elements:

$$\begin{pmatrix} 0 & 1 & 3 & \cdots \\ & 2 & 4 & \cdots \\ & & 5 & \cdots \\ & & & \ddots \end{pmatrix}.$$

Variants of this theme have been proposed over the course of the last few decades but have never caught on due to the complexity that is introduced when indexing the elements of the matrix [13, 5]. Given that this complexity will only increase with the tensor order, we do not pursue this idea.

Instead, we embrace an idea, storage by blocks, that was introduced into our `libflame` library [17, 29, 25] in order to support algorithms by blocks. Submatrices (blocks) become units of data and operations with those blocks become units of computation. Partition the symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ into submatrices as

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{00} & \mathbf{A}_{01} & \mathbf{A}_{02} & \cdots & \mathbf{A}_{0(\bar{n}-1)} \\ \mathbf{A}_{10} & \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1(\bar{n}-1)} \\ \mathbf{A}_{20} & \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2(\bar{n}-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{(\bar{n}-1)0} & \mathbf{A}_{(\bar{n}-1)1} & \mathbf{A}_{(\bar{n}-1)2} & \cdots & \mathbf{A}_{(\bar{n}-1)(\bar{n}-1)} \end{pmatrix}. \tag{3.3}$$

Here each submatrix $\mathbf{A}_{\bar{\imath}_0\bar{\imath}_1} \in \mathbb{R}^{b_{\mathbf{A}} \times b_{\mathbf{A}}}$. We define $\bar{n} = n/b_{\mathbf{A}}$ where, without loss of generality, we assume $b_{\mathbf{A}}$ evenly divides n. Hence $\mathbf{A}$ is a blocked $\bar{n} \times \bar{n}$ matrix with blocks of size $b_{\mathbf{A}} \times b_{\mathbf{A}}$. The blocks are stored using some conventional method (e.g., each $\mathbf{A}_{\bar{\imath}_0\bar{\imath}_1}$ is stored in column-major order). For symmetric matrices, the blocks below the diagonal are redundant and need not be stored (indicated by gray coloring). Although the diagonal blocks are themselves symmetric, we will not take advantage of this in order to simplify the access pattern for the computation with those blocks. We refer to this storage technique as Blocked Compact Symmetric Storage (BCSS) throughout the rest of this paper.

Storing the upper triangular *individual elements* of the symmetric matrix $\mathbf{A}$ requires storage of

$$n(n+1)/2 = \binom{n+1}{2} \text{ floats.}$$

In constrast, storing the upper triangular *blocks* of the symmetric matrix $\mathbf{A}$ with BCSS requires

$$n(n+b_{\mathbf{A}})/2 = b_{\mathbf{A}}^2 \binom{\bar{n}+1}{2} \text{ floats.}$$

The BCSS scheme requires a small amount of additional storage, depending on $b_{\mathbf{A}}$. Figure 3.2 illustrates how the storage needed when BCSS is used becomes no more expensive than storing only the upper triangular elements (here $n = 512$) as the number of blocks increases.

3.4. Algorithm-by-blocks. Given that $\mathbf{C}$ and $\mathbf{A}$ are stored with BCSS, we now need to discuss how the algorithm computes with these blocks. Partition $\mathbf{A}$ as in (3.3) and $\mathbf{C}$ and $\mathbf{X}$ like

$$\mathbf{C} = \begin{pmatrix} \mathbf{C}_{00} & \cdots & \mathbf{C}_{0(\bar{p}-1)} \\ \vdots & \ddots & \vdots \\ \mathbf{C}_{(\bar{p}-1)0} & \cdots & \mathbf{C}_{(\bar{p}-1)(\bar{p}-1)} \end{pmatrix} \text{ and } \mathbf{X} = \begin{pmatrix} \mathbf{X}_{00} & \cdots & \mathbf{X}_{0(\bar{n}-1)} \\ \vdots & \ddots & \vdots \\ \mathbf{X}_{(\bar{p}-1)0} & \cdots & \mathbf{X}_{(\bar{p}-1)(\bar{n}-1)} \end{pmatrix},$$

where, without loss of generality, $\bar{p} = p/b_{\mathbf{C}}$, and the blocks of $\mathbf{C}$ and $\mathbf{X}$ are of size $b_{\mathbf{C}} \times b_{\mathbf{C}}$ and $b_{\mathbf{C}} \times b_{\mathbf{A}}$, respectively. Then $\mathbf{C} := \mathbf{X}\mathbf{A}\mathbf{X}^T$ means that

$$\begin{aligned} \mathbf{C}_{\bar{\jmath}_0\bar{\jmath}_1} &= \begin{pmatrix} \mathbf{X}_{\bar{\jmath}_0 0} & \cdots & \mathbf{X}_{\bar{\jmath}_0(\bar{n}-1)} \end{pmatrix} \begin{pmatrix} \mathbf{A}_{00} & \cdots & \mathbf{A}_{0(\bar{n}-1)} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{(\bar{n}-1)0} & \cdots & \mathbf{A}_{(\bar{n}-1)(\bar{n}-1)} \end{pmatrix} \begin{pmatrix} \mathbf{X}_{\bar{\jmath}_1 0}^T \\ \vdots \\ \mathbf{X}_{\bar{\jmath}_1(\bar{n}-1)}^T \end{pmatrix} \\ &= \sum_{\bar{\imath}_0=0}^{\bar{n}-1} \sum_{\bar{\imath}_1=0}^{\bar{n}-1} \mathbf{X}_{\bar{\jmath}_0\bar{\imath}_0} \mathbf{A}_{\bar{\imath}_0\bar{\imath}_1} \mathbf{X}_{\bar{\jmath}_1\bar{\imath}_1}^T = \sum_{\bar{\imath}_0=0}^{\bar{n}-1} \sum_{\bar{\imath}_1=0}^{\bar{n}-1} [\mathbf{A}_{\bar{\imath}_0\bar{\imath}_1}; \mathbf{X}_{\bar{\jmath}_0\bar{\imath}_0}, \mathbf{X}_{\bar{\jmath}_1\bar{\imath}_1}] \text{ (in tensor notation).} \end{aligned} \tag{3.4}$$

This yields the algorithm in Figure 3.3, in which an analysis of its cost is also given. This algorithm avoids redundant computation, except within symmetric blocks of $\mathbf{C}$, $\mathbf{C}_{\bar{\jmath}_0\bar{\jmath}_0}$. Comparing (3.4) to (3.1) we see that the only difference lies in replacing scalar terms with their block counterparts. Consequently, comparing this algorithm with the one in Figure 3.1 (top-right), we notice that every scalar has simply been replaced by a block (submatrix). The algorithm now requires $n \times b_{\mathbf{C}}$ extra storage (for $\mathbf{T}$), in addition to the storage for $\mathbf{C}$ and $\mathbf{A}$.

4. The 3-way Case.

We now extend the insight gained in the last section to the case where $\mathcal{C}$ and $\mathcal{A}$ are symmetric order-3 tensors, before moving on to the general case in the next section.

4.1. The operation for order-3 tensors. Now $\mathcal{C} := [\mathcal{A}; \mathbf{X}, \mathbf{X}, \mathbf{X}]$ where $\mathcal{A} \in \mathbb{R}^{[m,n]}$, $\mathcal{C} \in \mathbb{R}^{[m,p]}$, and $[\mathcal{A}; \mathbf{X}, \mathbf{X}, \mathbf{X}] = \mathcal{A} \times_0 \mathbf{X} \times_1 \mathbf{X} \times_2 \mathbf{X}$. In our discussion, $\mathcal{A}$ is a

Algorithm	Ops (flops)	Total # of times executed	Temp. storage
for $\bar\jmath_1 = 0,\ldots,\bar p-1$			
$\underbrace{\begin{pmatrix}\mathbf{T}_0^T\\ \vdots\\ \mathbf{T}_{\bar n-1}^T\end{pmatrix}}_{\mathbf{T}} := \underbrace{\begin{pmatrix}\mathbf{A}_{00} & \cdots & \mathbf{A}_{0(\bar n-1)}\\ \vdots & \ddots & \vdots\\ \mathbf{A}_{(\bar n-1)0} & \cdots & \mathbf{A}_{(\bar n-1)(\bar n-1)}\end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix}\mathbf{X}_{\bar\jmath_1 0}^T\\ \vdots\\ \mathbf{X}_{\bar\jmath_1(\bar n-1)}^T\end{pmatrix}}_{\mathbf{X}_{\bar\jmath_1}^T}$	$2b_{\mathbf{C}}n^2$	$\bar p$	$b_{\mathbf{C}}n$
for $\bar\jmath_0 = 0,\ldots,\bar\jmath_1$			
$\mathbf{C}_{\bar\jmath_0\bar\jmath_1} := \begin{pmatrix}\mathbf{X}_{\bar\jmath_0 0} & \cdots & \mathbf{X}_{\bar\jmath_0(\bar n-1)}\end{pmatrix}\begin{pmatrix}\mathbf{T}_0^T\\ \vdots\\ \mathbf{T}_{\bar n-1}^T\end{pmatrix}$	$2b_{\mathbf{C}}^2 n$	$\bar p(\bar p+1)/2$	
endfor			
endfor			

Total Cost: $2b_{\mathbf{C}}n^2\bar p + 2b_{\mathbf{C}}^2 n\,(\bar p(\bar p+1)/2) = \sum_{d=0}^{1}\left(2b_{\mathbf{C}}^{d+1}n^{2-d}\binom{\bar p+d}{d+1}\right) \approx 2pn^2 + p^2 n$ flops

Total temporary storage: $b_{\mathbf{C}}n = \sum_{d=0}^{0}\left(b_{\mathbf{C}}^{d+1}n^{1-d}\right)$ entries

FIG. 3.3. *Algorithm-by-blocks for computing* $\mathbf{C} := \mathbf{X}\mathbf{A}\mathbf{X}^T = [\mathbf{A};\mathbf{X},\mathbf{X}]$. *The algorithm assumes that* $\mathbf{C}$ *is partitioned into blocks of size* $b_{\mathbf{C}} \times b_{\mathbf{C}}$, *with* $\bar p = \lceil p/b_{\mathbf{C}}\rceil$.

Algorithm	Ops (flops)	Total # of times executed	Temp. storage
for $\bar\jmath_2 = 0,\ldots,\bar p-1$			
$\begin{pmatrix}\mathcal{T}^{(2)}_{00} & \cdots & \mathcal{T}^{(2)}_{0(\bar n-1)}\\ \vdots & \ddots & \vdots\\ \mathcal{T}^{(2)}_{(\bar n-1)0} & \cdots & \mathcal{T}^{(2)}_{(\bar n-1)(\bar n-1)}\end{pmatrix} :=$ $\mathcal{A}\times_2\begin{pmatrix}\mathbf{X}_{\bar\jmath_2 0} & \cdots & \mathbf{X}_{\bar\jmath_2(\bar n-1)}\end{pmatrix}$	$2b_{\mathcal{C}}n^3$	$\bar p$	$b_{\mathcal{C}}n^2$
for $\bar\jmath_1 = 0,\ldots,\bar\jmath_2$			
$\begin{pmatrix}\mathcal{T}^{(1)}_0\\ \vdots\\ \mathcal{T}^{(1)}_{\bar n-1}\end{pmatrix} := \begin{pmatrix}\mathcal{T}^{(2)}_{00} & \cdots & \mathcal{T}^{(2)}_{0(\bar n-1)}\\ \vdots & \ddots & \vdots\\ \mathcal{T}^{(2)}_{(\bar n-1)0} & \cdots & \mathcal{T}^{(2)}_{(\bar n-1)(\bar n-1)}\end{pmatrix}$ $\times_1\begin{pmatrix}\mathbf{X}_{\bar\jmath_1 0} & \cdots & \mathbf{X}_{\bar\jmath_1(\bar n-1)}\end{pmatrix}$	$2b_{\mathcal{C}}^2 n^2$	$\bar p(\bar p+1)/2$ $=$ $\binom{\bar p+1}{2}$	$b_{\mathcal{C}}^2 n$
for $\bar\jmath_0 = 0,\ldots,\bar\jmath_1$			
$\mathcal{C}_{\bar\jmath_0\bar\jmath_1\bar\jmath_2} :=$ $\begin{pmatrix}\mathcal{T}^{(1)}_0\\ \vdots\\ \mathcal{T}^{(1)}_{\bar n-1}\end{pmatrix}\times_0\begin{pmatrix}\mathbf{X}_{\bar\jmath_0 0} & \cdots & \mathbf{X}_{\bar\jmath_0(\bar n-1)}\end{pmatrix}$	$2b_{\mathcal{C}}^3 n$	$\dfrac{\bar p(\bar p+1)(\bar p+2)}{6} =$ $\binom{\bar p+2}{3}$	
endfor			
endfor			
endfor			

Total Cost: $\sum_{d=0}^{2}\left(2b_{\mathcal{C}}^{d+1}n^{3-d}\binom{\bar p+d}{d+1}\right) \approx 2pn^3 + p^2n^2 + \frac{p^3 n}{3}$ flops

Total temporary storage: $b_{\mathcal{C}}n^2 + b_{\mathcal{C}}^2 n = \sum_{d=0}^{1}\left(b_{\mathcal{C}}^{d+1}n^{2-d}\right)$ entries

FIG. 3.4. *Algorithm-by-blocks for computing* $[\mathcal{A};\mathbf{X},\mathbf{X},\mathbf{X}]$. *The algorithm assumes that* $\mathcal{C}$ *is partitioned into blocks of size* $b_{\mathcal{C}}\times b_{\mathcal{C}}\times b_{\mathcal{C}}$, *with* $\bar p = \lceil p/b_{\mathcal{C}}\rceil$.

symmetric tensor as is $\boldsymbol{\mathcal{C}}$ by virtue of the operation applied to $\boldsymbol{\mathcal{A}}$. Now,

$$\begin{aligned}
\gamma_{j_0 j_1 j_2} &= \boldsymbol{\mathcal{A}} \times_0 \widehat{\boldsymbol{x}}_{j_0}^T \times_1 \widehat{\boldsymbol{x}}_{j_1}^T \times_2 \widehat{\boldsymbol{x}}_{j_2}^T = \sum_{i_0=0}^{n-1} (\boldsymbol{\mathcal{A}} \times_1 \widehat{\boldsymbol{x}}_{j_1}^T \times_2 \widehat{\boldsymbol{x}}_{j_2}^T)_{i_0} \times_0 \chi_{j_0 i_0} \\
&= \sum_{i_0=0}^{n-1} (\sum_{i_1=0}^{n-1} (\boldsymbol{\mathcal{A}} \times_2 \widehat{\boldsymbol{x}}_{j_2}^T)_{i_1} \times_1 \chi_{j_1 i_1})_{i_0} \times_0 \chi_{j_0 i_0} \\
&= \sum_{i_2=0}^{n-1} \sum_{i_1=0}^{n-1} \sum_{i_0=0}^{n-1} \alpha_{i_0 i_1 i_2} \chi_{j_0 i_0} \chi_{j_1 i_1} \chi_{j_2 i_2}.
\end{aligned}$$

4.2. Simple algorithms. A naive algorithm is given in Figure 3.1 (middle-left). The cheaper algorithm to its right is motivated by

$$\begin{aligned}
\boldsymbol{\mathcal{A}} \times_0 \mathbf{X} \times_1 \mathbf{X} \times_2 \mathbf{X} &= \boldsymbol{\mathcal{A}} \times_0 \begin{pmatrix} \widehat{\boldsymbol{x}}_0^T \\ \vdots \\ \widehat{\boldsymbol{x}}_{p-1}^T \end{pmatrix} \times_1 \begin{pmatrix} \widehat{\boldsymbol{x}}_0^T \\ \vdots \\ \widehat{\boldsymbol{x}}_{p-1}^T \end{pmatrix} \times_2 \begin{pmatrix} \widehat{\boldsymbol{x}}_0^T \\ \vdots \\ \widehat{\boldsymbol{x}}_{p-1}^T \end{pmatrix} \\
&= \underbrace{\begin{pmatrix} \mathbf{T}_0^{(2)} & \cdots & \mathbf{T}_{p-1}^{(2)} \end{pmatrix}}_{\boldsymbol{\mathcal{T}}^{(2)}} \times_0 \begin{pmatrix} \widehat{\boldsymbol{x}}_0^T \\ \vdots \\ \widehat{\boldsymbol{x}}_{p-1}^T \end{pmatrix} \times_1 \begin{pmatrix} \widehat{\boldsymbol{x}}_0^T \\ \vdots \\ \widehat{\boldsymbol{x}}_{p-1}^T \end{pmatrix} \\
&= \underbrace{\begin{pmatrix} \mathbf{t}_{00}^{(1)} & \cdots & \mathbf{t}_{0(p-1)}^{(1)} \\ \vdots & \ddots & \vdots \\ \mathbf{t}_{(p-1)0}^{(1)} & \cdots & \mathbf{t}_{(p-1)(p-1)}^{(1)} \end{pmatrix}}_{\boldsymbol{\mathcal{T}}^{(1)}} \times_0 \begin{pmatrix} \widehat{\boldsymbol{x}}_0^T \\ \vdots \\ \widehat{\boldsymbol{x}}_{p-1}^T \end{pmatrix},
\end{aligned}$$

where $\mathbf{T}_{i_2}^{(2)} = \boldsymbol{\mathcal{A}} \times_2 \widehat{\boldsymbol{x}}_{i_2}^T$, $\mathbf{t}_{i_1 i_2}^{(1)} = \mathbf{T}_{i_2}^{(2)} \times_1 \widehat{\boldsymbol{x}}_{i_1}^T = \boldsymbol{\mathcal{A}} \times_2 \widehat{\boldsymbol{x}}_{i_2}^T \times_1 \widehat{\boldsymbol{x}}_{i_1}^T$, and $\mathbf{T}_{i_2}^{(2)} \in \mathbb{R}^{n \times n}$, $\mathbf{t}_{i_1 i_2}^{(1)} \in \mathbb{R}^n$, $\widehat{\boldsymbol{x}}_j^T \in \mathbb{R}^{1 \times n}$.

Although it appears $\boldsymbol{\mathcal{T}}^{(2)}$ is a vector (oriented across the page) of matrices, it is actually a vector (oriented *into* the page) of matrices (each $\mathbf{T}_{i_2}^{(2)}$ should be viewed a matrix oriented first down and then across the page). Similarly, $\boldsymbol{\mathcal{T}}^{(1)}$ should be viewed as a matrix (first oriented *into* the page, then across the page) of vectors (each $\mathbf{t}_{i_1 i_2}^{(1)}$ should be viewed as a vector oriented down the page). This is a result of the mode-multiplication and is an aspect that is difficult to represent on paper.

This algorithm requires $p(2n^3 + p(2n^2 + 2pn)) = 2pn^3 + 2p^2n^2 + 2p^3n$ flops at the expense of requiring workspace for a matrix $\mathbf{T}$ of size $p \times n$ and vector $\mathbf{t}$ of length n.

4.3. Blocked Compact Symmetric Storage (BCSS). In the matrix case (Section 3), we described BCSS, which stores only the blocks in the upper triangular part of the matrix. The storage scheme used in the 3-way case is analogous to the matrix case; the difference is that instead of storing blocks belonging to a 2-way upper triangle, we must store the blocks in the "upper triangular" region of a 3-way tensor. This region is comprised of all indices (i_0, i_1, i_2) where $i_0 \leq i_1 \leq i_2$. For lack of a better term, we refer to this as *upper hypertriangle* of the tensor.

Similar to how we extended the notion of the upper triangular region of a 3-way tensor, we must extend the notion of a block to three dimensions. Instead of a

	Compact (Minimum)	Blocked Compact (BCSS)	Dense
$m = 2$	$\frac{(n+1)n}{2} = \binom{n+1}{2}$	$b_{\mathcal{A}}^2 \binom{\bar{n}+1}{2}$	n^2
$m = 3$	$\binom{n+2}{3}$	$b_{\mathcal{A}}^3 \binom{\bar{n}+2}{3}$	n^3
$m = d$	$\binom{n+d-1}{d}$	$b_{\mathcal{A}}^d \binom{\bar{n}+d-1}{d}$	n^d

FIG. 4.1. *Storage requirements for a tensor $\mathcal{A}$ under different storage schemes.*

block being a two-dimensional submatrix, a block for 3-way tensors becomes a 3-way subtensor. Partition tensor $\mathcal{A} \in \mathbb{R}^{[3,n]}$ into cubical blocks of size $b_{\mathcal{A}} \times b_{\mathcal{A}} \times b_{\mathcal{A}}$:

$$\mathcal{A}_{::0} = \begin{pmatrix} \mathcal{A}_{000} & \mathcal{A}_{010} & \cdots & \mathcal{A}_{0(\bar{n}-1)0} \\ \mathcal{A}_{100} & \mathcal{A}_{110} & \cdots & \mathcal{A}_{1(\bar{n}-1)0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{A}_{(\bar{n}-1)00} & \mathcal{A}_{(\bar{n}-1)10} & \cdots & \mathcal{A}_{(\bar{n}-1)(\bar{n}-1)0} \end{pmatrix}, \cdots,$$

$$\mathcal{A}_{::(\bar{n}-1)} = \begin{pmatrix} \mathcal{A}_{00(\bar{n}-1)} & \mathcal{A}_{01(\bar{n}-1)} & \cdots & \mathcal{A}_{0(\bar{n}-1)(\bar{n}-1)} \\ \mathcal{A}_{10(\bar{n}-1)} & \mathcal{A}_{11(\bar{n}-1)} & \cdots & \mathcal{A}_{1(\bar{n}-1)(\bar{n}-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{A}_{(\bar{n}-1)0(\bar{n}-1)} & \mathcal{A}_{(\bar{n}-1)1(\bar{n}-1)} & \cdots & \mathcal{A}_{(\bar{n}-1)(\bar{n}-1)(\bar{n}-1)} \end{pmatrix},$$

where $\mathcal{A}_{\bar{\imath}_0\bar{\imath}_1\bar{\imath}_2}$ is $b_{\mathcal{A}} \times b_{\mathcal{A}} \times b_{\mathcal{A}}$ (except, possibly, the fringe blocks) and $\bar{n} = \lceil n/b_{\mathcal{A}} \rceil$. These blocks are stored using some conventional method and the blocks lying outside the upper hypertriangular region are not stored. Once again, we will not take advantage of any symmetry within blocks (blocks with $\bar{\imath}_0 = \bar{\imath}_1$, $\bar{\imath}_0 = \bar{\imath}_2$, or $\bar{\imath}_1 = \bar{\imath}_2$) to simplify the access pattern when computing with these blocks.

Summarized in Figure 4.1, we see that while storing *only* the upper hypertriangular elements of the tensor $\mathcal{A}$ requires $\binom{n+2}{3}$ storage, one has to now store $b_{\mathcal{A}}^3 \binom{\bar{n}+2}{3}$ elements. However, since $\binom{\bar{n}+2}{3} b_{\mathcal{A}}^3 \approx \frac{\bar{n}^3}{3!} b_{\mathcal{A}}^3 = \frac{n^3}{6}$, we achieve a savings of approximately a factor 6 (if $\bar{n}$ is large enough) relative to storing all elements. Once again, we can apply the same storage method to $\mathcal{C}$ for additional savings. What we have described is the obvious extension to BCSS, which is what we will call it for higher order tensors as well.

Blocks for which the index criteria $\bar{\imath}_0 = \bar{\imath}_1 = \bar{\imath}_2$ is *NOT* met yet still contain *some* symmetry are referred to as *partially*-symmetric blocks. Taking advantage of partial-symmetry in blocks is left for future research.

4.4. Algorithm-by-blocks. We now discuss an algorithm-by-blocks for the 3-way case. Partition $\mathcal{C}$ and $\mathcal{A}$ into $b_{\mathcal{C}}^3$ and $b_{\mathcal{A}}^3$ blocks, respectively, and partition $\mathbf{X}$ into $b_{\mathcal{C}} \times b_{\mathcal{A}}$ blocks. Then, extending the insights we gained from the matrix case, $\mathcal{C} := [\mathcal{A}; \mathbf{X}, \mathbf{X}, \mathbf{X}]$ means that

$$\begin{aligned} \mathcal{C}_{\bar{\jmath}_0\bar{\jmath}_1\bar{\jmath}_2} &= \sum_{\bar{\imath}_0=0}^{\bar{n}-1} \sum_{\bar{\imath}_1=0}^{\bar{n}-1} \sum_{\bar{\imath}_2=0}^{\bar{n}-1} \mathcal{A}_{\bar{\imath}_0\bar{\imath}_1\bar{\imath}_2} \times_0 \mathbf{X}_{\bar{\jmath}_0\bar{\imath}_0} \times_1 \mathbf{X}_{\bar{\jmath}_1\bar{\imath}_1} \times_2 \mathbf{X}_{\bar{\jmath}_2\bar{\imath}_2} \\ &= \sum_{\bar{\imath}_0=0}^{\bar{n}-1} \sum_{\bar{\imath}_1=0}^{\bar{n}-1} \sum_{\bar{\imath}_2=0}^{\bar{n}-1} [\mathcal{A}_{\bar{\imath}_0\bar{\imath}_1\bar{\imath}_2}; \mathbf{X}_{\bar{\jmath}_0\bar{\imath}_0}, \mathbf{X}_{\bar{\jmath}_1\bar{\imath}_1}, \mathbf{X}_{\bar{\jmath}_2\bar{\imath}_2}]. \end{aligned}$$

This yields the algorithm in Figure 3.4, in which an analysis of its cost is also given. This algorithm avoids redundant computation, except for within blocks of $\mathfrak{C}$ which are symmetric or partially-symmetric ($\bar{\jmath}_0 = \bar{\jmath}_1$ or $\bar{\jmath}_1 = \bar{\jmath}_2$ or $\bar{\jmath}_0 = \bar{\jmath}_2$). The algorithm now requires $b_{\mathfrak{C}} n^2 + b_{\mathfrak{C}}^2 n$ extra storage (for $\mathfrak{T}^{(2)}$ and $\mathfrak{T}^{(1)}$, respectively), in addition to the storage for $\mathfrak{C}$ and $\boldsymbol{\mathcal{A}}$.

5. The m-way Case.

We now generalize to tensors $\mathfrak{C}$ and $\boldsymbol{\mathcal{A}}$ of any order.

5.1. The operation for order-m tensors.

Letting m be any non-negative integer value yields $\mathfrak{C} := [\boldsymbol{\mathcal{A}}; \mathbf{X}, \mathbf{X}, \cdots, \mathbf{X}]$ where $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[m,n]}$, $\mathfrak{C} \in \mathbb{R}^{[m,p]}$, and $[\boldsymbol{\mathcal{A}}; \mathbf{X}, \mathbf{X}, \cdots \mathbf{X}] = \boldsymbol{\mathcal{A}} \times_0 \mathbf{X} \times_1 \mathbf{X} \times_2 \cdots \times_{m-1} \mathbf{X}$. In our discussion, $\boldsymbol{\mathcal{A}}$ is a symmetric tensor as is $\mathfrak{C}$ by virtue of the operation applied to $\boldsymbol{\mathcal{A}}$.

Let $\gamma_{j_0 j_1 \cdots j_{m-1}}$ denote the $(j_0, j_1, \ldots, j_{m-1})$ element of the order-m tensor $\mathfrak{C}$. Then, by simple extension, we find that

$$\begin{aligned} \gamma_{j_0 j_1 \cdots j_{m-1}} &= \boldsymbol{\mathcal{A}} \times_0 \widehat{\boldsymbol{x}}_{j_0}^T \times_1 \widehat{\boldsymbol{x}}_{j_1}^T \times_2 \cdots \times_{m-1} \widehat{\boldsymbol{x}}_{j_{m-1}}^T \\ &= \sum_{i_{m-1}=0}^{n-1} \cdots \sum_{i_0=0}^{n-1} \alpha_{i_0 i_1 \cdots i_{m-1}} \chi_{j_0 i_0} \chi_{j_1 i_1} \cdots \chi_{j_{m-1} i_{m-1}}. \end{aligned}$$

5.2. Simple algorithms.

A naive algorithm with a cost of $(m+1)p^m n^m$ flops is now given in Figure 3.1 (bottom-left). By comparing the loop structure of the naive algorithms in the 2-way and 3-way cases, the pattern for a cheaper algorithm (one which reduces flops) in the m-way case should become obvious. Extending the cheaper algorithm in the 3-way case suggests the algorithm given in Figure 3.1 (bottom-right), This algorithm requires

$$p(2n^3 + p(2n^2 + 2pn)) = 2pn^m + 2p^2 n^{m-1} + \cdots 2p^{m-1} n = 2 \sum_{i=0}^{m-1} p^{i+1} n^{m-i} \text{ flops}$$

at the expense of requiring workspace for tensors of order 1 through $m-1$.

5.3. Blocked Compact Symmetric Storage (BCSS).

We now further extend BCSS for the general m-way case. The upper hypertriangular region now contains all indices $(i_0, i_1, \ldots, i_{m-1})$ where $i_0 \leq i_1 \leq \ldots \leq i_{m-1}$. Using the 3-way case as a guide, one can envision by extrapolation how a block partitioned order-m tensor looks. The tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[m,n]}$ is partitioned into hyper-cubical blocks of size $b_{\boldsymbol{\mathcal{A}}}^m$. The blocks lying outside the upper hypertriangular region are not stored. Once again, we will not take advantage of symmetry within blocks.

Summarized in Figure 4.1 while storing *only* the upper hypertriangular elements of the tensor $\boldsymbol{\mathcal{A}}$ requires $\binom{n+m-1}{m}$ storage, one has to now store $\binom{\bar{n}+m-1}{m} b_{\boldsymbol{\mathcal{A}}}^m$ elements which potentially achieves a savings factor of $m!$ (if $\bar{n}$ is large enough).

Although the approximation $\binom{\bar{n}+m-1}{m} b_{\boldsymbol{\mathcal{A}}}^m \approx \dfrac{n^m}{m!}$ is used, the lower-order terms have a significant effect on the actual storage savings factor. Examining Figure 5.1, we see that as we increase m, we require a significantly larger $\bar{n}$ to have the actual storage savings factor approach the theoretical factor. While this figure only shows the results for a particular value of $b_{\boldsymbol{\mathcal{A}}}$, the effect applies to all values of $b_{\boldsymbol{\mathcal{A}}}$.

5.4. Algorithm-by-blocks.

Given that $\mathfrak{C}$ and $\boldsymbol{\mathcal{A}}$ are going to be stored using BCSS, we now need to discuss how to compute with these blocks. Assume the

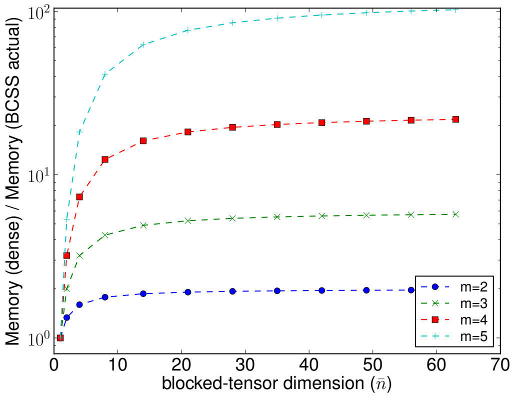


FIG. 5.1. *Actual storage savings of BCSS on $\mathcal{A}$*

Algorithm	Ops (flops)	Total # of times executed	Temp. storage
for $\bar{\jmath}_{m-1} = 0, \ldots, \bar{p}-1$			
$\mathcal{T}^{(m-1)} := \mathcal{A} \times_{m-1} \left(\mathbf{X}_{\bar{\jmath}_{m-1}0} \;\cdots\; \mathbf{X}_{\bar{\jmath}_{m-1}(\bar{n}-1)} \right)$	$2b_{\mathcal{C}}n^m$	$\binom{\bar{p}}{1}$	$b_{\mathcal{C}}n^{m-1}$
$\ddots$	$\vdots$	$\vdots$	$\vdots$
for $\bar{\jmath}_1 = 0, \ldots, \bar{\jmath}_2$			
$\mathcal{T}^{(1)} := \mathcal{T}^{(2)} \times_1 \left(\mathbf{X}_{\bar{\jmath}_1 0} \;\cdots\; \mathbf{X}_{\bar{\jmath}_1(\bar{n}-1)} \right)$	$2b_{\mathcal{C}}^{m-1}n^2$	$\binom{\bar{p}+m-2}{m-1}$	$b_{\mathcal{C}}^{m-1}n$
for $\bar{\jmath}_0 = 0, \ldots, \bar{\jmath}_1$			
$\mathcal{C}_{\bar{\jmath}_0\bar{\jmath}_1\cdots\bar{\jmath}_{m-1}} :=$ $\mathcal{T}^{(1)} \times_0 \left(\mathbf{X}_{\bar{\jmath}_0 0} \;\cdots\; \mathbf{X}_{\bar{\jmath}_0(\bar{n}-1)} \right) =$ $\begin{pmatrix} \mathcal{T}^{(1)}_0 \\ \vdots \\ \mathcal{T}^{(1)}_{\bar{n}-1} \end{pmatrix} \times_0 \left(\mathbf{X}_{\bar{\jmath}_0 0} \;\cdots\; \mathbf{X}_{\bar{\jmath}_0(\bar{n}-1)} \right)$	$2b_{\mathcal{C}}^{m}n$	$\binom{\bar{p}+m-1}{m}$	
endfor			
endfor			
$\iddots$			
endfor			

$$\text{Total Cost: } \sum_{d=0}^{m-1} \left(2b_{\mathcal{C}}^{d+1} n^{m-d} \binom{\bar{p}+d}{d+1} \right) \text{ flops}$$

$$\text{Total additional storage: } \sum_{d=0}^{m-2} \left(b_{\mathcal{C}}^{d+1} n^{m-1-d} \right) \text{ floats}$$

FIG. 5.2. *Algorithm-by-blocks for computing $\mathcal{C} := [\mathcal{A}; \mathbf{X}, \cdots, \mathbf{X}]$. The algorithm assumes that $\mathcal{C}$ is partitioned into blocks of size $b_{\mathcal{C}}^m$, with $\bar{p} = \lceil p/b_{\mathcal{C}} \rceil$.*

partioning discussed above, with an obvious indexing into the blocks. Now,

$$\begin{aligned}
\mathcal{C}_{\bar{\jmath}_0\bar{\jmath}_1\cdots\bar{\jmath}_{m-1}} &= \sum_{\bar{\imath}_0=0}^{\bar{n}-1} \cdots \sum_{\bar{\imath}_{m-1}=0}^{\bar{n}-1} \mathcal{A}_{\bar{\imath}_0\bar{\imath}_1\cdots\bar{\imath}_{m-1}} \times_0 \mathbf{X}_{\bar{\jmath}_0\bar{\imath}_0} \times_1 \mathbf{X}_{\bar{\jmath}_1\bar{\imath}_1} \times_2 \cdots \times_{m-1} \mathbf{X}_{\bar{\jmath}_{m-1}\bar{\imath}_{m-1}} \\
&= \sum_{\bar{\imath}_0=0}^{\bar{n}-1} \cdots \sum_{\bar{\imath}_{m-1}=0}^{\bar{n}-1} [\mathcal{A}_{\bar{\imath}_0\bar{\imath}_1\cdots\bar{\imath}_{m-1}}; \mathbf{X}_{\bar{\jmath}_0\bar{\imath}_0}, \mathbf{X}_{\bar{\jmath}_1\bar{\imath}_1}, \cdots, \mathbf{X}_{\bar{\jmath}_{m-1}\bar{\imath}_{m-1}}].
\end{aligned}$$

	$\boldsymbol{\mathcal{C}}$	$\boldsymbol{\mathcal{A}}$	$\boldsymbol{\mathcal{T}}^{(m-1)}$
Memory (in floats)	$\binom{\bar{p}+m-1}{m} b_{\boldsymbol{\mathcal{C}}}^m \approx \frac{\bar{p}^m}{m!} b_{\boldsymbol{\mathcal{C}}}^m = \frac{p^m}{m!}$	$\binom{\bar{n}+m-1}{m} b_{\boldsymbol{\mathcal{A}}}^m \approx \frac{\bar{n}^m}{m!} b_{\boldsymbol{\mathcal{A}}}^m = \frac{n^m}{m!}$	$n^{m-1} b_{\boldsymbol{\mathcal{C}}}$

FIG. 5.3. *Memory requirements for various objects used.*

This yields the algorithm given in Figure 5.2, which avoids much redundant computation, except for within blocks of $\boldsymbol{\mathcal{C}}$ which are symmetric or partially-symmetric ($\bar{\jmath}_i = \bar{\jmath}_k$ and $i \neq k$).

5.5. Memory requirements. The algorithm described in Figure 5.2 utilizes a set of temporaries. At its worst, the `sttsm` operation acting on a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[m,n]}$ forms a temporary $\boldsymbol{\mathcal{T}}^{(m-1)}$ with $n^{(m-1)} \times b_{\boldsymbol{\mathcal{C}}}$ entries [*] for a chosen block dimension $b_{\boldsymbol{\mathcal{C}}}$. The algorithm requires additional temporaries $\boldsymbol{\mathcal{T}}^{(m-2)}, \cdots, \boldsymbol{\mathcal{T}}^{(1)}$; however, these are significantly smaller than $\boldsymbol{\mathcal{T}}^{(m-1)}$ and therefore we ignore their space requirements in this analysis. We store $\boldsymbol{\mathcal{A}}$ with BCSS with block size $b_{\boldsymbol{\mathcal{A}}}$ and we store all elements of $\boldsymbol{\mathcal{T}}^{(m-1)}, \cdots, \boldsymbol{\mathcal{T}}^{(1)}$. There may be partial symmetries in $\boldsymbol{\mathcal{T}}^{(m-1)}, \cdots, \boldsymbol{\mathcal{T}}^{(1)}$, but we do not take advantage of these here. The storage requirements are then given in Figure 5.3 where we use the approximation $\binom{n+m-1}{m} \approx \frac{n^m}{m!}$ when $m \ll n$.

It is interesting to ask the question how parameters have to change with m, as m increases, in order to maintain a constant fraction of space required for the temporaries. For example, if we wish to keep memory required for $\boldsymbol{\mathcal{T}}^{(m-1)}$ constant relative to the memory required for $\boldsymbol{\mathcal{A}}$, then the fraction

$$\frac{n^{m-1} b_{\boldsymbol{\mathcal{C}}}}{\binom{\bar{n}+m-1}{m} b_{\boldsymbol{\mathcal{A}}}^m} \approx \frac{n^{m-1} b_{\boldsymbol{\mathcal{C}}}}{\frac{n^m}{m!}} = \frac{b_{\boldsymbol{\mathcal{C}}} m!}{n}$$

must be kept approximately constant. Hence, if n is constant and m increases, $b_{\boldsymbol{\mathcal{C}}}$ must be chosen inversely proportional to $m!$. This is disconcerting, because the larger $b_{\boldsymbol{\mathcal{C}}}$, the less benefit one derives from BCSS. On the other hand, if we look at this more optimistically, and allow for a fraction of the *dense* storage for the workspace, then

$$\frac{n^{m-1} b_{\boldsymbol{\mathcal{C}}}}{n^m} = \frac{b_{\boldsymbol{\mathcal{C}}}}{n}.$$

and $b_{\boldsymbol{\mathcal{C}}}$ can be kept constant as m increases.

Because we are storing symmetric tensors hierarchically, in practice, we require additional storage for meta-data (implementation-specific data that is unrelated to underlying mathematics) associated with each block of the tensors we store with BCSS. Although this does impact the overall storage requirement, as this is an implementation detail we do not thoroughly discuss it here. The impact of meta-data storage on the proposed algorithm is discussed in Appendix B and is shown to be minimal.

5.6. Computational cost. In Figure 5.2, an analysis of the computational cost of computing intermediate results is given, as well as the overall cost. Notice that

[*] $\boldsymbol{\mathcal{T}}^{(m-1)}$ is an m-way tensor where each mode is of dimension n except the last which is of dimension $b_{\boldsymbol{\mathcal{C}}}$

if the algorithm did *not* take advantage of symmetry, the range for each nested loop becomes $0, \ldots, \bar{p}-1$, meaning that each nested loop is executed $\bar{p}$ times, and the total cost becomes

$$\sum_{d=0}^{m-1} \left(2 b_{\mathfrak{C}}^{d+1} n^{m-d} \bar{p}^{d+1}\right) = \sum_{d=0}^{m-1} \left(2 n^{m-d} p^{d+1}\right) \text{ flops.}$$

A simple formula for the speedup achieved by taking advantage of symmetry is nontrivial to establish in general from the formula in Figure 5.2 and the above formula. However, when $d \ll \bar{p}$ we find that

$$\begin{aligned} \frac{\sum_{d=0}^{m-1} 2n^{m-d}p^{d+1}}{\sum_{d=0}^{m-1} 2b_{\mathfrak{C}}^{d+1} n^{m-d} \binom{\bar{p}+d}{d+1}} &\approx \frac{\sum_{d=0}^{m-1} n^{m-d}p^{d+1}}{\sum_{d=0}^{m-1} b_{\mathfrak{C}}^{d+1} n^{m-d} \frac{\bar{p}^{d+1}}{(d+1)!}} = \frac{\sum_{d=0}^{m-1} n^{m-d}p^{d+1}}{\sum_{d=0}^{m-1} n^{m-d} \frac{p^{d+1}}{(d+1)!}} \\ \text{(and, if } n = p\text{,)} \qquad &= \frac{\sum_{d=0}^{m-1} n^{m+1}}{\sum_{d=0}^{m-1} \frac{n^{m+1}}{(d+1)!}} = \frac{m}{\sum_{d=0}^{m-1} \frac{1}{(d+1)!}}. \end{aligned}$$

Since $1 \leq \sum_{d=0}^{m-1} \frac{1}{(d+1)!} < \sum_{d=0}^{\infty} \frac{1}{(d+1)!} = e - 1$ we establish that

$$0.58m < \frac{m}{e-1} < \underbrace{\frac{m}{\sum_{d=0}^{m-1} \frac{1}{(d+1)!}}}_{\substack{\text{approximate speedup} \\ \text{from exploiting symmetry} \\ \text{when } n = p \text{ and } m \ll \bar{p}}} \leq m.$$

As the above shows, if we take advantage of symmetry when applying the `sttsm` operation to an order-m tensor, we can reduce the number of required flops by a factor between $0.58m$ and m over the approach examined which does *not* take advantage of symmetry.

5.7. Permutation overhead. Tensor computations are generally implemented by a sequence of permutations of the data in the input tensor $\boldsymbol{\mathcal{A}}$, interleaved with calls to a matrix-matrix multiplication implemented by a call to the BLAS routine `dgemm`, details of which can be found in Appendix A. These permutations constitute a nontrivial overhead, as we will discuss now.

If one does NOT take advantage of symmetry, then a series of exactly m tensor-matrix-multiply operations are required to be performed on $\boldsymbol{\mathcal{A}}$, each resizing a single mode at a time. This means the cost associated with permutations for the dense case is

$$\left(1 + 2\frac{p}{n}\right) \sum_{d=0}^{m-1} p^d n^{m-d} \text{ memory operations (memops).}$$

When BCSS is used, the permutations and multiplications happen for each tensor computation with a block. Figure 5.4 presents an analysis of the cost of permuting data for each tensor-matrix-multiply operation, as well as the total cost of performing permutations of the data. We see that the proposed algorithm-by-blocks requires significantly more memops to perform the permutations since each block of $\boldsymbol{\mathcal{A}}$ is

Algorithm	Permute ops (in mem ops)	Total # of times executed
for $\bar{\jmath}_{m-1} = 0, \ldots, \bar{p} - 1$		
$\mathfrak{T}^{(m-1)} :=$		
$\boldsymbol{\mathcal{A}} \times_{m-1} \left(\mathbf{X}_{\bar{\jmath}_{m-1}0} \; \cdots \; \mathbf{X}_{\bar{\jmath}_{m-1}(\bar{n}-1)} \right)$	$n^m + 2b_{\mathfrak{C}}n^{m-1}$	$\binom{\bar{p}}{1}$
$\ddots$	$\vdots$	$\vdots$
for $\bar{\jmath}_1 = 0, \ldots, \bar{\jmath}_2$		
$\mathfrak{T}^{(1)} := \mathfrak{T}^{(2)} \times_1 \left(\mathbf{X}_{\bar{\jmath}_1 0} \; \cdots \; \mathbf{X}_{\bar{\jmath}_1(\bar{n}-1)} \right)$	$b_{\mathfrak{C}}^{m-2}n^2 + 2b_{\mathfrak{C}}^{m-1}n$	$\binom{\bar{p}+m-2}{m-1}$
for $\bar{\jmath}_0 = 0, \ldots, \bar{\jmath}_1$		
$\mathfrak{C}_{\bar{\jmath}_0\bar{\jmath}_1\cdots\bar{\jmath}_{m-1}} :=$		
$\mathfrak{T}^{(1)} \times_0 \left(\mathbf{X}_{\bar{\jmath}_0 0} \; \cdots \; \mathbf{X}_{\bar{\jmath}_0(\bar{n}-1)} \right)$	$b_{\mathfrak{C}}^{m-1}n + 2b_{\mathfrak{C}}^{m}$	$\binom{\bar{p}+m-1}{m}$
endfor		
endfor		
$\cdot^{\cdot^{\cdot}}$		
endfor		

Total Cost: $\displaystyle \left(1 + 2\frac{b_{\mathfrak{C}}}{n}\right) \sum_{d=0}^{m-1} b_{\mathfrak{C}}^{d} n^{m-d} \binom{\bar{p}+d}{d+1}$ mem ops

FIG. 5.4. *Algorithm-by-blocks for computing* $\mathfrak{C} := [\boldsymbol{\mathcal{A}}; \mathbf{X}, \cdots, \mathbf{X}]$. *The algorithm assumes that* $\mathfrak{C}$ *is partitioned into blocks of size* $b_{\mathfrak{C}}^{m}$, *with* $\bar{p} = \lceil p/b_{\mathfrak{C}} \rceil$.

permuted $\binom{\bar{p}+d}{d+1}$ times. The required cost becomes

$$\frac{\left(1 + 2\frac{p}{n}\right)\sum_{d=0}^{m-1} p^d n^{m-d}}{\left(1 + 2\frac{b_{\mathfrak{C}}}{n}\right)\sum_{d=0}^{m-1} b_{\mathfrak{C}}^{d} n^{m-d}\binom{\bar{p}+d}{d+1}} \approx \frac{\left(1 + 2\frac{p}{n}\right)\sum_{d=0}^{m-1} p^d n^{m-d}}{\left(1 + 2\frac{b_{\mathfrak{C}}}{n}\right)\sum_{d=0}^{m-1} b_{\mathfrak{C}}^{d} n^{m-d}\frac{\bar{p}^{d+1}}{(d+1)!}}$$

$$\text{(if, further, } n = p\text{)} \qquad = \frac{3\sum_{d=0}^{m-1} n^m}{\left(1 + 2\frac{b_{\mathfrak{C}}}{n}\right)\sum_{d=0}^{m-1}\frac{n^m \bar{p}}{(d+1)!}} = \frac{3m}{(\bar{p}+2)\sum_{d=0}^{m-1}\frac{1}{(d+1)!}} \text{ memops.}$$

This yields

$$\frac{1.74m}{(\bar{p}+2)} < \frac{3m}{(\bar{p}+2)(e-1)} < \underbrace{\frac{3m}{(\bar{p}+2)\sum_{d=0}^{m-1}\frac{1}{(d+1)!}}}_{\substack{\text{approximate speedup} \\ \text{from exploiting symmetry} \\ \text{when } n = p \text{ and } m \ll \bar{p}}} \leq \frac{3m}{(\bar{p}+2)}.$$

As the above shows, if $\bar{p}$ is chosen too large, the algorithm will require more memops in the form of permutations, unless m is scaled accordingly. However, m is typically defined by the problem and therefore is not adjustable.

We can mitigate the permutation overhead by adjusting $b_{\mathfrak{C}}$. For example, by letting $\bar{p} = 2$ ($b_{\mathfrak{C}} = \lceil p/2 \rceil$), we are able to take advantage of symmetry in $\mathfrak{C}$ for

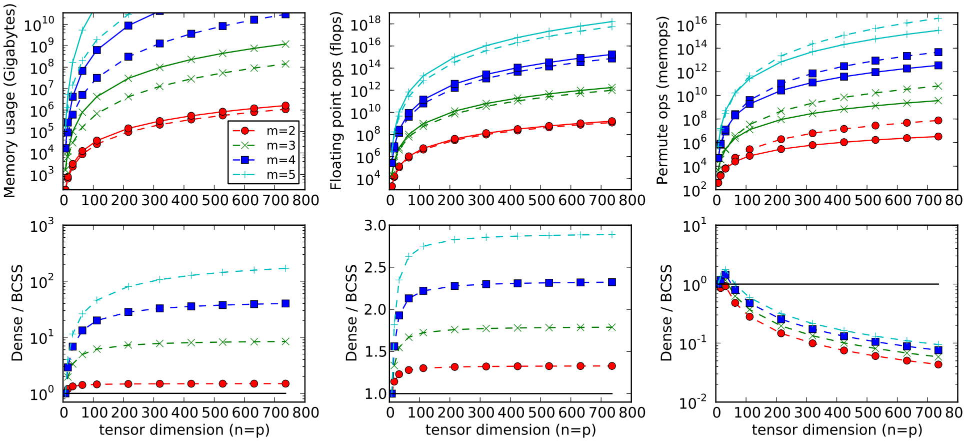


FIG. 5.5. *Comparison of dense to BCSS algorithms. Solid and dashed lines correspond to dense and BCSS, respectively. From left to right: storage requirements; cost from computation (flops); cost from permutations (memops). For these graphs $b_{\mathbf{A}} = b_{\mathbf{C}} = 8$.*

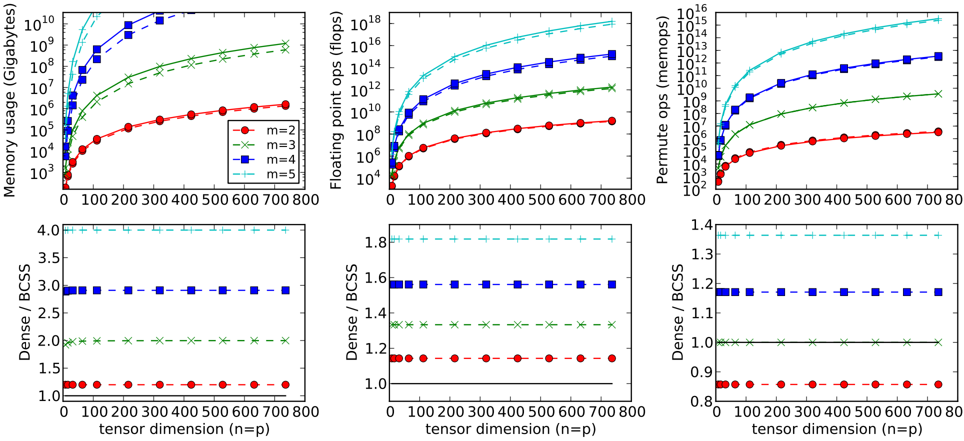


FIG. 5.6. *Comparison of dense to BCSS algorithms. Solid and dashed lines correspond to dense and BCSS, respectively. From left to right: storage requirements; cost from computation (flops); cost from permutations (memops). Here $\bar{n} = \bar{p} = 2$.*

storage reasons (as $b_{\mathbf{C}} \neq 1$) while limiting the increase in memops required (relative to that encounted when dense storage is used) to

$$0.43m < \frac{3m}{4(e-1)} < \frac{3m}{4\sum_{d=0}^{m-1}\frac{1}{(d+1)!}} \leq 0.75m.$$

It is tempting to discuss an optimal choice for $\mathbf{C}$. However, it would be better to focus on greatly reducing this overhead, as we will discuss in the conclusion.

5.8. Summary. Figures 5.5–5.6 illustrate the insights discussed in this section. The (exact) formulae developed for storage, flops, and memops are used to compare and contrast dense storage with BCSS.

In Figure 5.5, the top graphs report storage, flops, and memops (due to permutations) as a function of tensor dimension (n), for different tensor orders (m), for the case where the storage block size is relatively small ($b_{\mathcal{A}} = b_{\mathcal{C}} = 8$). The bottom graphs report the same information, but as a ratio. The graphs illustrate that BCSS dramatically reduces the required storage and the proposed algorithms reduce the flops requirements for the `sttsm` operation, at the expense of additional memops due to the encountered permutations.

In Figure 5.6 a similar analysis is given, but for the case where the block size is half the tensor dimension (i.e., $\bar{n} = \bar{p} = 2$). It shows that the memops can be greatly reduced by increasing the storage block dimensions, but this then adversely affects the storage and computational benefits.

It would be tempting to discuss how to choose an optimal block dimension. However, the real issues is that the overhead of permutation should be reduced and/or eliminated. Once that is achieved, in future research, the question of how to choose the block size becomes relevant.

6. Experimental Results. In this section, we report on the performance attained by an implementation of the discussed approach. It is important to keep in mind that the current state of the art of tensor computations is first and foremost concerned with reducing memory requirements so that reasonably large problems can be executed. This is where taking advantage of symmetry is important. Second to that is the desire to reduce the number of floating point operations to the minimum required. Our algorithms perform the minimum number of floating point operation (except for a lower order term that would result from taking advantage of partial symmetry in the temporary results). Actually attaining high performance, like is attained for dense matrix computations, is still relatively low priority compared to these issues.

6.1. Target architecture. We report on experiments on a single core of a Dell PowerEdge R900 server consisting of four six-core Intel Xeon 7400 processors and 96 GBytes of memory. Performance experiments were gathered under the GNU/Linux 2.6.18 operating system. Source code was compiled by the GNU C compiler, version 4.1.2. All experiments were performed in double-precision floating-point arithmetic on randomized real domain matrices. The implementations were linked to the OpenBLAS 0.1.1 library [19, 28], a fork of the GotoBLAS2 implementation of the BLAS [12, 11]. As noted, most time is spent in the permutations necessary to cast computation in terms of the BLAS matrix-matrix multiplication routine `dgemm`. Thus, the peak performance of the processor and the details of the performance attained by the BLAS library are mostly irrelevant at this stage. The experiments merely show that the new approach to storing matrices as well as the algorithm that takes advantage of symmetry has promise, rather than making a statement about optimality of the implementation. Much room for improvement remains.

6.2. Implementation. The implementation was coded in a style inspired by the `libflame` library [29, 25]. An API similar to the FLASH API [17] for storing matrices as matrices of blocks and implementing algorithm-by-blocks was defined and implemented. Computations with the (tensor and matrix) blocks were implemented as the discussed sequence of permutations interleaved with calls to the `dgemm` BLAS kernel. No attempt was yet made to optimize these permutations. However, an apples-to-apples comparison resulted from using the same sequence of permutations and calls to `dgemm` for both the experiments that take advantage of symmetry and

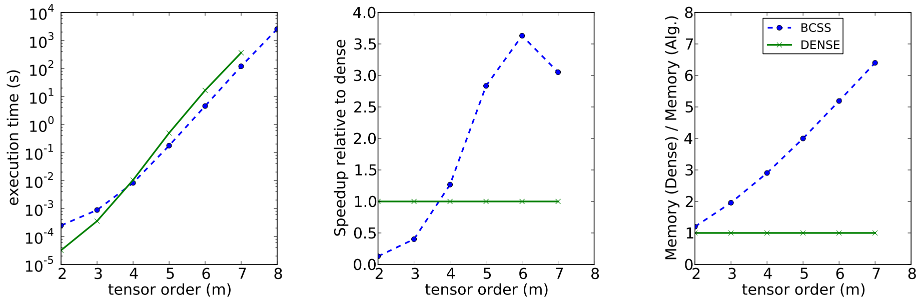


FIG. 6.1. *Experimental results when $n = p = 16$, $b_{\mathcal{A}} = b_{\mathcal{C}} = 8$ and the tensor order, m, is varied. For $m = 8$, storing $\mathcal{A}$ and $\mathcal{C}$ without taking advantage of symmetry requires too much memory.*

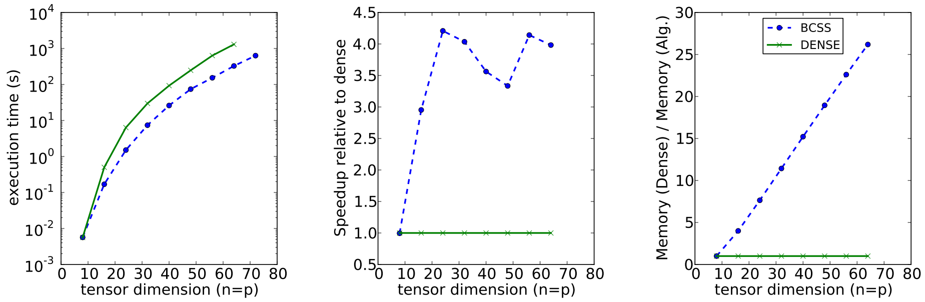


FIG. 6.2. *Experimental results when the order $m = 5$, $b_{\mathcal{A}} = b_{\mathcal{C}} = 8$ and the tensor dimensions $n = p$ are varied. For $n = p = 72$, storing $\mathcal{A}$ and $\mathcal{C}$ without taking advantage of symmetry requires too much memory.*

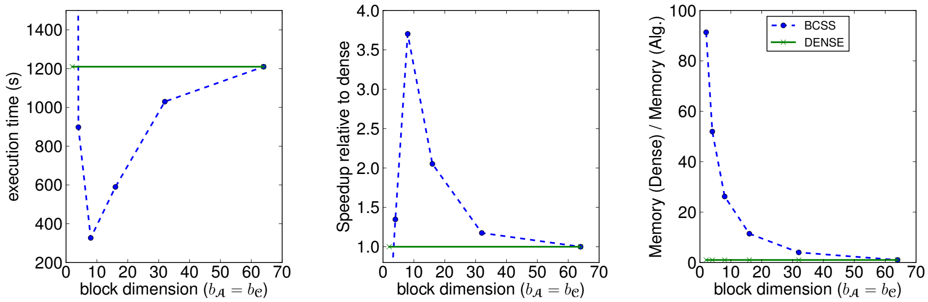


FIG. 6.3. *Experimental results when the order $m = 5$, $n = p = 64$ and the block dimensions $b_{\mathcal{A}} = b_{\mathcal{C}}$ are varied.*

those that store and compute with the tensor densely, ignoring symmetry.

6.3. Results. Figures 6.1–6.3 show results from executing our implementation on the target architecture of the algorithms which do and do not take advantage of

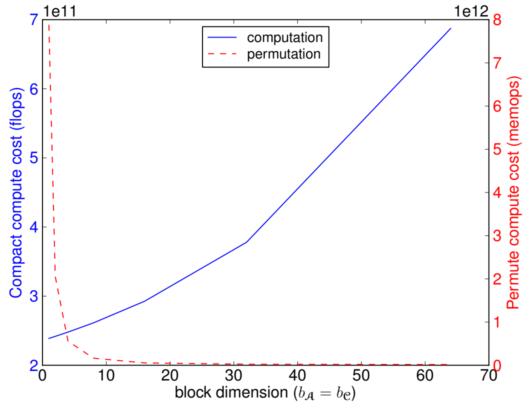


FIG. 6.4. *Comparison of computational cost to permutation cost where $m = 5$, $n = p = 64$ and tensor block dimension ($b_{\mathcal{A}}$, $b_{\mathcal{C}}$) is varied. The solid line represents the number of flops (due to computation) required for a given problem (left axis), and the dashed line represents the number of memops (due to permutation) required for a given problem (right axis).*

symmetry. The first is denoted a BCSS algorithm while the latter a dense algorithm. All figures show comparisons of the execution time of each algorithm, the associated speedup of the BCSS algorithm over the dense algorithm, and the estimated storage savings factor of the BCSS algorithm.

For the experiments reported in Figure 6.1 we fix n, p, $b_{\mathcal{A}}$, $b_{\mathcal{C}}$, and vary the tensor order m. The BCSS algorithm begins to outperform the dense algorithm after the tensor order is equal to or greater than 4. This is due to the fact that at lower tensor orders, the speedup of the BCSS algorithm is lower, and therefore any computational overhead introduced due to our implementation of the BCSS algorithm is more pronounced (increasing the execution time of the BCSS algorithm). Additionally, notice that that BCSS allows larger problems to be solved; the dense algorithm was unable to store the entire symmetric tensor when $m = 8$. We see that once the problem-size becomes larger ($m = 7$), the speedup of the BCSS algorithm over the dense algorithm decreases. This is surprising since, according to the mathematics defining this operation, we would expect a continual increase in speedup (computational savings of the BCSS algorithm is proportional to the order of the tensor). We believe this effect to be due to an implementation detail that we plan to investigate in future work.

In Figure 6.2 we fix m, $b_{\mathcal{A}}$, and $b_{\mathcal{C}}$ and vary the tensor dimensions n and p. We see that the BCSS algorithm outperforms the dense algorithm and attains a noticeable speedup. The speedup appears to level out as the tensor dimension increases which is unexpected. We will discuss this effect later as it appears in other experiments.

In Figure 6.3 we fix m, n, and p, and vary the block sizes $b_{\mathcal{A}}$ and $b_{\mathcal{C}}$. The right-most point on the axis corresponds to the dense case (as $b_{\mathcal{A}} = n = b_{\mathcal{C}} = p$) and the left-most point corresponds to the fully-compact case (where only unique entries are stored). There now is a range of block dimensions for which the BCSS algorithm outperforms the dense algorithm. Similar to previous result, as we increase the block size, our relative speedup decreases which we now discuss.

In Figures 6.2–6.3, we saw that as tensor dimension or block dimension increased the relative speedup of the BCSS algorithm over the dense algorithm lessened/leveled out. We believe this is due to a balance between the cost of permutations and the cost of computation. In Figure 6.4 we illustrate (with predicted flop and memop counts) that if we pick a small block dimension, we dramatically increase the memops due to permutations while dramatically decreasing the computational cost. If instead we pick a large block dimension, we dramatically lower the permutation cost, while increasing the computational cost. Although this plot only shows one case, in general this tension between memops due to permutations and flops due to computation exists. In the large block dimension regime we are performing fewer permutations than computations which is beneficial since memops are significantly more expensive than flops.

We believe that the effect of the small block dimension region (which dramatically increases memops) explains the dropping/leveling off of relative speedup of Figure 6.2 where the tensor dimension becomes large while the block dimension remains constant and thus the relative block dimension becomes small. Further, we believe that the effect of the large block dimension region which dramatically increases flops explains the dropping/leveling off of relative speedup of Figure 6.3 where the block dimension increases while the tensor dimension remains constant (the block dimension becomes relatively large). This, and other effects related to block size, will be studied further in future research.

7. Conclusion and Future Work.

We have presented storage by blocks, BCSS, for tensors and shown how this can be used to compactly store symmetric tensors. The benefits were demonstrated with an implementation of a new algorithm for the change-of-basis (`sttsm`) operation. Theoretical and practical results showed that both the storage and computational requirements were reduced relative to storing the tensors densely and computing without taking advantage of symmetry.

This initial study has exposed many new research opportunities, which we believe to be the real contribution of this paper. We finish by discussing some of these opportunities.

Optimizing permutations. In our work, we made absolutely no attempt to optimize the permutation operations. Without doubt, a careful study of how to organize these permutations will greatly benefit performance. It is likely that the current implementation not only causes unnecessary cache misses, but also a great number of Translation Lookaside Buffer (TLB) misses [12], which cause the core to stall for a hundred or more cycles.

Optimized kernels/avoiding permutation. A better way to mitigate the permutations is to avoid them as much as possible. If $n = p$, the `sttsm` operation performs $O(n^{m+1})$ operations on $O(n^m)$ data. This exposes plenty of opportunity to optimize this kernel much like `dgemm`, which performs $O(n^3)$ computation on $O(n^2)$ data, is optimized. For other tensor operations, the ratio is even more favorable.

We are developing a BLAS-like library, BLIS [26], that allows matrix operations with matrices that have both a row and a column stride, as opposed to the traditional column-major order supported by the BLAS. This means that computation with a planar slice in a tensor can be passed into the BLIS matrix-matrix multiplication routine, avoiding the explicit permutations that must now be performed before calling `dgemm`. How to rewrite the computations with blocks in terms of BLIS, and studying the performance benefits, is a future topic of research.

One can envision instead creating a BLAS-like library for the tensor operations we perform with blocks. One alternative for this is to apply the techniques developed as part of the PHiPAC [7], TCE, SPIRAL [21], or ATLAS [27] projects to the problem of how to optimize computations with blocks. This should be a simpler problem than optimizing the complete tensor contraction problems that, for example, TCE targets now, since the sizes of the operands are restricted. The alternative is to create microkernels for tensor computations, similar to the microkernels that BLIS defines and exploits for matrix computations, and to use these to build a high-performance tensor library that in turn can then be used for the computations with tensor blocks.

Algorithmic variants for the `sttsm` *operation.* For matrices, there is a second algorithmic variant for computing $\mathbf{C} := \mathbf{X}\mathbf{A}\mathbf{X}^T$. Partition $\mathbf{A}$ by rows and $\mathbf{X}$ by columns:

$$\mathbf{A} = \begin{pmatrix} \widehat{\boldsymbol{a}}_0^T \\ \vdots \\ \widehat{\boldsymbol{a}}_{n-1}^T \end{pmatrix} \quad \text{and} \quad \mathbf{X} = \begin{pmatrix} \mathbf{x}_0 & \cdots & \mathbf{x}_{n-1} \end{pmatrix}.$$

Then

$$\mathbf{C} = \mathbf{X}\mathbf{A}\mathbf{X}^T = \begin{pmatrix} \mathbf{x}_0 & \cdots & \mathbf{x}_{n-1} \end{pmatrix} \begin{pmatrix} \widehat{\boldsymbol{a}}_0^T \\ \vdots \\ \widehat{\boldsymbol{a}}_{n-1}^T \end{pmatrix} \mathbf{X}^T = \mathbf{x}_0(\widehat{\boldsymbol{a}}_0^T\mathbf{X}) + \cdots \mathbf{x}_{n-1}(\widehat{\boldsymbol{a}}_{n-1}^T\mathbf{X}).$$

We suspect that this insight can be extended to the `sttsm` operation, yielding a new set of algorithm-by-blocks that will have different storage and computational characteristics.

Extending the FLAME methodology to multi-linear operations. In this paper, we took an algorithm that was systematically derived with the FLAME methodology for the matrix case and then extended it to the equivalent tensor computation. Ideally, we would derive algorithms directly from the specification of the tensor computation, using a similar methodology. This requires a careful consideration of how to extend the FLAME notation for expressing matrix algorithms, as well as how to then use that notation to systematically derive algorithms.

Partial symmetry: storage and computation. The presented algorithm, which takes advantage of symmetry for computational purposes only, does so with respect to the final output $\mathfrak{C}$. However, the described algorithm forms temporaries which contain partial-symmetries of which the proposed algorithm does not take advantage. Therefore, unnecessary storage is required and unnecessary computation is being performed when forming temporaries. Moreover, as numerous temporaries are formed, one can argue that taking advantage of partial symmetries in temporaries is almost as important as taking advantage of symmetry in the input and output tensors.

So far, we have only spoken about exploiting partial symmetries at the block level. However, each block may contain partial-symmetries which can be exploited for computational or storage reasons. It is difficult to say whether taking advantage of partial-symmetries within each block is advantageous, but it should be examined.

Multithreaded parallel implementation. Multithreaded parallelism can be accomplished in a number of ways.

- The code can be linked to a multithreaded implementation of the BLAS, thus attaining parallelism within the `dgemm` call. This would require one to hand-parallelize the permutations.
- Parallelism can be achieved by scheduling the operations with blocks to threads much like our SuperMatrix [22] runtime does for the `libflame` library, or PLASMA [1] does for its tiled algorithms.

We did not yet pursue this because at the moment the permutations are a bottleneck that, likely, consumes all available memory bandwidth. As a result, parallelization does not make sense until the cost of the permutations is mitigated.

Exploiting accelerators. In a number of papers [18, 14], we and others have shown how the algorithm-by-blocks (tiled algorithm) approach, when combined with a runtime system, can exploit (multiple) GPUs and other accelerators. These techniques can be naturally extended to accommodate the algorithm-by-blocks for tensor computations.

Distributed parallel implementation. Once we understand how to derive sequential algorithms, it becomes possible to consider distributed memory parallel implementation. It may be that our insights can be incorporated into the Cyclops Tensor Framework [23], or that we build on our own experience with distributed memory libraries for dense matrix computations, the PLAPACK [24] and Elemental [20] libraries, to develop a new distributed memory tensor library.

General multi-linear library. The ultimate question is, of course, how the insights in this paper and future ones can be extended to a general, high-performance multilinear library, for all platforms.

Acknowledgments. This work was partially sponsored by NSF grant OCI-1148125. This work was supported by the Applied Mathematics program at the U.S. Department of Energy. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. volume 180, 2009.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.).* SIAM, 1999.

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

[4] Brett W. Bader, Tamara G. Kolda, et al. Matlab tensor toolbox version 2.5. Available online, January 2012.

[5] Grey Ballard, Tamara G. Kolda, and Todd Plantenga. Efficiently computing tensor eigenvalues on a GPU. In *IPDPSW'11: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1340–1348. IEEE Computer Society, May 2011.

[6] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. In *Proceedings of the IEEE*, volume 93, pages 276–292, 2005.

[7] J. Bilmes, K. Asanovic, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing.* ACM SIGARC, July 1997.

[8] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Sympo-*

sium on the Frontiers of Massively Parallel Computation, pages 120–127. IEEE Comput. Soc. Press, 1992.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[11] Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Soft.*, 35(1):1–14, 2008.

[12] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3):12, May 2008. Article 12, 25 pages.

[13] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, Dec. 2001.

[14] Francisco D. Igual, Ernie Chan, Enrique S. Quintana-Ort, Gregorio Quintana-Ort, Robert A. van de Geijn, and Field G. Van Zee. The flame approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing*, 72(9):1134 – 1143, 2012. Accelerators for High-Performance Computing.

[15] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51:455–500, Jan. 2009.

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

[17] Tze Meng Low and Robert van de Geijn. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-2004-15, The University of Texas at Austin, Department of Computer Sciences, May 2004.

[18] Mercedes Marqués, Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving "large" dense matrix problems on multi-core processors and gpus. In *10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing - PDSEC'09. Roma (Italia)*, 2009.

[19] `http://xianyi.github.com/OpenBLAS/`, 2012.

[20] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 39(2).

[21] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005. special issue on Program Generation, Optimization, and Adaptation.

[22] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3):14:1–14:26, July 2009.

[23] Edgar Solomonik, Jeff Hammond, and James Demmel. A preliminary analysis of Cyclops Tensor Framework. Technical Report UCB/EECS-2012-29, EECS Department, University of California, Berkeley, Mar 2012.

[24] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[25] Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering*, 11(6):56–62, 2009.

[26] Field G. Van Zee and Robert A. van de Geijn. FLAME Working Note #66. BLIS: A framework for generating BLAS-like libraries. Technical Report TR-12-30, The University of Texas at Austin, Department of Computer Sciences, Nov. 2012. submitted to ACM TOMS.

[27] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[28] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.

[29] Field G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.

Appendix A. Casting Tensor-Matrix Multiplication to BLAS. Given a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_0 \times \cdots \times I_{m-1}}$, a mode k, and a matrix $\mathbf{B} \in \mathbb{R}^{J \times I_k}$, the result of multiplying $\mathbf{B}$ along the k-th mode of $\boldsymbol{\mathcal{A}}$ is denoted by $\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}} \times_k \mathbf{B}$, where $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{I_0 \times \cdots \times I_{k-1} \times J \times I_{k+1} \times \cdots \times I_{m-1}}$ and each element of $\boldsymbol{\mathcal{C}}$ is defined as

$$\boldsymbol{\mathcal{C}}_{i_0 \cdots i_{k-1} j_0 i_{k+1} \cdots i_{m-1}} = \sum_{i_k=0}^{I_k} \alpha_{i_0 \cdots i_{m-1}} \beta_{j_0 i_k}.$$

This operation is typically computed by casting it as a matrix-matrix multiplication for which high-performance implementations are available as part of the Basic Linear Algebra Subprograms (BLAS) routine `dgemm`.

The problem viewing a higher-order tensor as a matrix is analogous to the problem of viewing a matrix as a vector. We first describe this simpler problem and show how it generalizes to objects of higher-dimension.

Matrices as vectors (and vice-versa). A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ can be viewed as a vector $\mathbf{a} \in \mathbb{R}^{M}$ where $M = mn$ by assigning $\mathbf{a}_{i_0 + i_1 m} = \mathbf{A}_{i_0 i_1}$. (This is analogous to column-major order assignment of a matrix to memory.) This alternative view does not change the relative order of the elements in the matrix, since it just logically views them in a different way. We say that the two dimensions of $\mathbf{A}$ are merged or "grouped" to form the single index of $\mathbf{a}$.

Using the same approach, we can view $\mathbf{a}$ as $\mathbf{A}$ by assigning the elements of $\mathbf{A}$ according to the mentioned equivalence. In this case, we are in effect viewing the single index of $\mathbf{a}$ as two separate indices. We refer to this effect as a "splitting" of the index of $\mathbf{a}$.

Tensors as matrices (and vice-versa). A straightforward extension of grouping of indices allows us to view higher-order tensors as matrices and (inversely) matrices as higher-order tensors. The difference lies with the calculation used to assign elements of the lower/higher-order tensor.

As an example, consider an order-4 tensor $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{I_0 \times I_1 \times I_2 \times I_3}$. We can view $\boldsymbol{\mathcal{C}}$ as a matrix $\mathbf{C} \in \mathbb{R}^{J_0 \times J_1}$ where $J_0 = I_0 \times I_1$ and $J_1 = I_2 \times I_3$. Because of this particular grouping of indices, the elements as laid out in memory need not be rearranged (relative order of each element remains the same). This follows from the observation that memory itself is a linear array (vector) and realizing that if $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ are both mapped to a 1-dimensional vector using column-major order and its higher dimensional extension (which we will call dimensional order), both will be stored identically.

The need for permutation. If we wished to instead view our example $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{I_0 \times I_1 \times I_2 \times I_3}$ as a matrix $\mathbf{C} \in \mathbb{R}^{J_0 \times J_1}$ where, for instance, $J_0 = I_1$ and $J_1 = I_0 \times I_2 \times I_3$, then this would require a rearrangement of the data since mapping $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ to memory using dimensional order will not generally produce the same result for both. This is a consequence of changing the relative order of indices in our mappings.

This rearrangement of data is what is referred to as a *permutation* of data. By specifying an input tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_0 \times \cdots \times I_{m-1}}$ and the desired permutation of indices of $\boldsymbol{\mathcal{A}}$, π, we define the transformation $\boldsymbol{\mathcal{C}} = \text{permute}(\boldsymbol{\mathcal{A}}, \pi)$ that yields $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{I_{\pi_0} \times I_{\pi_1} \times \cdots \times I_{\pi_{m-1}}}$ so that $\boldsymbol{\mathcal{C}}_{i'_0 \cdots i'_{m-1}} = \boldsymbol{\mathcal{A}}_{i_0 \cdots i_{m-1}}$ where i' corresponds to the result of applying the permutation π to i. The related operation ipermute inverts this transfomation when supplied π so that $\boldsymbol{\mathcal{C}} = \mathit{ipermute}(\boldsymbol{\mathcal{A}}, \pi)$ yields $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{I_{\pi_0^{-1}} \times I_{\pi_1^{-1}} \times \cdots \times I_{\pi_{m-1}^{-1}}}$ where $\boldsymbol{\mathcal{C}}_{i'_0 \cdots i'_{m-1}} = \boldsymbol{\mathcal{A}}_{i_0 \cdots i_{m-1}}$ where i' corresponds to the result of applying the permutation π^{-1} to i.

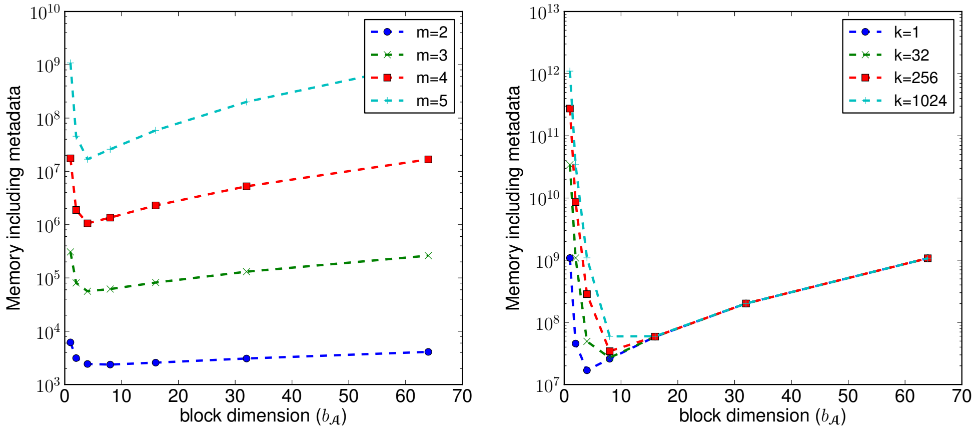


Fig. B.1. *Left: Storage requirements of $\mathcal{A} \in \mathbb{R}^{[m,64]}$ as block dimension changes. Right: Storage requirements of $\mathcal{A} \in \mathbb{R}^{[5,64]}$ for different choices for the meta-data stored per block, measured by k, as block dimension changes.*

Casting a tensor computation in terms of a matrix-matrix multiplication. We can now show how the operation $\mathcal{C} = \mathcal{A} \times_k \mathbf{B}$, where $\mathcal{A} \in \mathbb{R}^{I_0 \times \cdots \times I_{m-1}}$, $\mathbf{B} \in \mathbb{R}^{J \times I_k}$, and $\mathcal{C} \in \mathbb{R}^{I_0 \times \cdots \times I_{k-1} \times J \times I_{k+1} \times \cdots \times I_{m-1}}$, can be cast as a matrix-matrix multiplication if the tensors are appropriately permuted. The following describes the algorithm:

1. Permute: $\mathcal{P}_{\mathcal{A}} \leftarrow \text{permute}(\mathcal{A}, \{k, 0, \ldots, k-1, k+1, \ldots, m-1\})$.
2. Permute: $\mathcal{P}_{\mathcal{C}} \leftarrow \text{permute}(\mathcal{C}, \{k, 0, \ldots, k-1, k+1, \ldots, m-1\})$.
3. View tensor $\mathcal{P}_{\mathcal{A}}$ as matrix $\mathbf{A}$: $\mathbf{A} \leftarrow \mathcal{P}_{\mathcal{A}}$, where $\mathbf{A} \in \mathbb{R}^{I_k \times J_1}$ and $J_1 = I_0 \cdots I_{k-1} I_{k+1} \cdots I_{m-1}$.
4. View tensor $\mathcal{P}_{\mathcal{C}}$ as matrix $\mathbf{C}$: $\mathbf{C} \leftarrow \mathcal{P}_{\mathcal{C}}$, where $\mathbf{C} \in \mathbb{R}^{J \times J_1}$ and $J_1 = I_0 \cdots I_{k-1} I_{k+1} \cdots I_{m-1}$.
5. Compute matrix-matrix product: $\mathbf{C} := \mathbf{BA}$.
6. View matrix $\mathbf{C}$ as tensor $\mathcal{P}_{\mathcal{C}}$: $\mathcal{P}_{\mathcal{C}} \leftarrow \mathbf{C}$,
   where $\mathcal{P}_{\mathcal{C}} \in \mathbb{R}^{J \times I_0 \times \cdots \times I_{k-1} \times I_{k+1} \times \cdots \times I_{m-1}}$.
7. "Unpermute": $\mathcal{C} \leftarrow \text{ipermute}(\mathcal{P}_{\mathcal{C}}, \{k, 0, \ldots, k-1, k+1, \ldots, m-1\})$.

Step 5. can be implemented by a call to the BLAS routine `dgemm`, which is typically highly optimized.

Appendix B. Design Details.

We now give a few details about the particular implementation of BCSS, and how this impacts storage requirements. Notice that this is one choice for implementing this storage scheme in practice. One can envision other options that, at the expense of added complexity in the code, reduce the memory footprint.

BCSS views tensors hierarchically. At the top level, there is a tensor where each element of that tensor is itself a tensor (block). Our way of implementing this stores a description (meta-data) for a block in each element of the top-level tensor. This meta-data adds to memory requirements. In our current implementation, the top-level tensor of meta-data is itself a dense tensor. The meta-data in the upper hypertriangular tensor describes stored blocks. The meta-data in the rest of the top-level tensor reference the blocks that correspond to those in the upper hypertriangular tensor (thus requiring an awareness of the permutation needed to take a stored block and transform it). This design choice greatly simplifies our implementation (which we hope to describe in a future paper). We will show that although the meta-data can potentially require considerable space, this can be easily mitigated. We will use $\boldsymbol{\mathcal{A}}$ for example purposes.

Given $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[m,n]}$ stored with BCSS with block dimension $b_{\boldsymbol{\mathcal{A}}}$, we must store meta-data for $\bar{n}^m$ blocks where $\bar{n} = \lceil n/b_{\boldsymbol{\mathcal{A}}} \rceil$. This means that the total cost of storing $\boldsymbol{\mathcal{A}}$ with BCSS is

$$C_{\text{storage}}(\boldsymbol{\mathcal{A}}) = k\bar{n}^m + b_{\boldsymbol{\mathcal{A}}}^m \binom{\bar{n}+m-1}{m} \text{ floats,}$$

k is a constant representing the amount of storage required for the meta-data associated with one block, in floats. Obviously, this meta-data is of a different datatype, but floats will be our unit.

There is a tradeoff between the cost for storing the meta-data and the actually entries of $\boldsymbol{\mathcal{A}}$, parameterized by the blocksize $b_{\boldsymbol{\mathcal{A}}}$:

- If $b_{\boldsymbol{\mathcal{A}}} = n$, then we only require a minimal amount of memory for meta-data, k floats, but must store all entries of $\boldsymbol{\mathcal{A}}$ since there now is only one block, and that block uses dense storage. We thus store slightly more than we would if we stored the tensor without symmetry.
- If $b_{\boldsymbol{\mathcal{A}}} = 1$, then $\bar{n} = n$ and we must store meta-data for each element, meaning we store much more data than if we just used a dense storage scheme.

Picking a block dimension somewhere between these two extremes results in a smaller footprint overall. For example, if we choose a block dimension $b_{\boldsymbol{\mathcal{A}}} = \sqrt{n}$, then $\bar{n} = \sqrt{n}$ and the total storage required to store $\boldsymbol{\mathcal{A}}$ with BCSS is

$$\begin{aligned} C_{\text{storage}(\boldsymbol{\mathcal{A}})} &= k\bar{n}^m + b_{\boldsymbol{\mathcal{A}}}^m \binom{\bar{n}+m-1}{m} = kn^{\frac{m}{2}} + n^{\frac{m}{2}} \binom{n^{\frac{m}{2}}+m-1}{m} \\ &\approx kn^{\frac{m}{2}} + n^{\frac{m}{2}} \left(\frac{n^{\frac{m}{2}}}{m!}\right) = n^{\frac{m}{2}} \left(k + \frac{n^{\frac{m}{2}}}{m!}\right) \text{ floats,} \end{aligned}$$

which, provided that $k \ll \dfrac{n^{\frac{m}{2}}}{2}$, is significantly smaller than the storage required for the dense case (n^m). This discussion suggests that a point exists that requires less storage than the dense case (showing that BCSS is a feasible solution).

In Figure B.1, we illustrate that as long as we pick a block dimension that is greater than 4, we avoid incurring extra costs due to meta-data storage. It should be noted that changing the dimension of the tensors also has no effect on the minimum, however if they are too small, then the dense storage scheme may be the minimal storage scheme. Additionally, adjusting the order of tensors has no real effect on the block dimension associated with minimal storage. However increasing the amount of storage allotted for meta-data slowly shifts the optimal block dimension choice towards the dense storage case.