SIAM J. SCI. COMPUT.
Vol. 36, No. 5, pp. C453–C479

EXPLOITING SYMMETRY IN TENSORS FOR HIGH PERFORMANCE: MULTIPLICATION WITH SYMMETRIC TENSORS*

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 a blocked data structure (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 by utilizing a series of temporary tensors to avoid redundant computation. Further, partial symmetry within temporaries is exploited to further avoid redundant storage and redundant computation. A detailed analysis shows that, relative to storing and computing with tensors without taking advantage of symmetry and partial symmetry, storage requirements are reduced by a factor of $O(m!)$ and computational requirements by a factor of $O((m+1)!/2^m)$, where m is the order of the tensor. However, as the analysis shows, care must be taken in choosing the correct block size to ensure these storage and computational benefits are achieved (particularly for low-order tensors). An implementation demonstrates that storage is greatly reduced and the complexity introduced by storing and computing with tensors by blocks is manageable. Preliminary results demonstrate that computational time is also reduced. The paper concludes with a discussion of how insights in this paper point to opportunities for generalizing recent advances in the domain of linear algebra libraries to the field of multilinear computation.

Key words. high performance, multilinear algebra, algorithms

AMS subject classification. 15A69

DOI. 10.1137/130907215

1. Introduction.

A tensor is a multidimensional or m-way array. Tensor computations are increasingly prevalent in a wide variety of applications [22]. Alas, libraries for dense multilinear 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. Specifically, this paper focuses on exploring how exploiting symmetry in matrix computations extends to computations with symmetric tensors, tensors whose entries are invariant under any permutation of indices, and exploring how block structures and algorithms extend to computations with symmetric tensors.

Libraries for dense linear algebra (matrix computations) have long been part of the standard arsenal for computational science, including the basic linear algebra sub-

*Submitted to the journal's Software and High Performance Computing section January 24, 2013; accepted for publication (in revised form) March 11, 2014; published electronically September 30, 2014. This work was partially sponsored by NSF grants ACI-1148125 and CCF-1320112. This work was also supported by the Applied Mathematics program at the U.S. Department of Energy. Sandia National Laboratories is a multiprogram 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).*

http://www.siam.org/journals/sisc/36-5/90721.html

†Department of Computer Science, Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78701 (martin.schatz@utexas.edu, ltm@cs.utexas.edu, rvdg@cs.utexas.edu).

‡Sandia National Laboratories, Livermore, CA 94551-9159 (tgkolda@sandia.gov).

program (BLAS) interface [23, 12, 11, 17, 16], LAPACK [3], and more recent libraries with similar functionality, like the BLAS-like interface software framework (BLIS) [37], and `libflame` [41, 36]. For distributed memory architectures, the ScaLAPACK [10], PLAPACK [35], and Elemental [27] libraries provide most of the functionality of 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 [5, 4] defines many commonly used operations that would be needed by a library for multilinear algebra but does not have any high-performance kernels nor special computations or data structures for symmetric tensors. The PLS Toolbox [13] provides users with operations for analyzing data stored as tensors, but does not expose the underlying system for users to develop their own set of operations. Targeting distributed memory environments, the tensor contraction engine (TCE) project [7] focuses on sequences of tensor contractions and uses compiler techniques to reduce workspace and operation counts. The cyclops tensor framework (CTF) [33] focuses on exploiting symmetry in storage for distributed memory parallel computation with tensors, but at present does not include efforts to optimize computation within each computational node.

In a talk at the Eighteenth Householder Symposium meeting (2011), Charlie Van Loan stated, "In my opinion, blocking will eventually have the same impact in tensor computations as it does in matrix computations." The approach we take in this paper heavily borrows from the FLAME project [36]. We use the *change-of-basis* operation, also known as a symmetric tensor times same matrix (in all modes) (sttsm) operation [5], to motivate the issues and solutions. In the field of computational chemistry, this operation is referred to as an atomic integral transformation [8] when applied to order-4 tensors. This operation appears in other contexts as well, such as computing a low-rank Tucker-type decomposition of a symmetric tensor [21] and blind source separation [32]. We propose algorithms that require significantly less (possibly minimal) computation relative to an approach based on a series of successive matrix-matrix multiply operations by computing and storing temporaries. Additionally, the tensors are stored by blocks, following similar solutions developed for matrices [24, 29]. In addition to many of the projects mentioned previously, other work, such as that by Ragnarsson and Van Loan [30] suggest devising algorithms in terms of tensor blocks to aid in computation with both symmetric tensors and tensors in general.

Given that we store the tensor by blocks, the algorithms must be reformulated to operate with these blocks. Since we need only store the unique blocks of a symmetric tensor, symmetry is exploited at the block level (both for storage and computation) while preserving regularity when computing within blocks. Temporaries are formed to reduce the computational requirements, similar to work in the field of computational chemistry [8]. To further reduce computational and storage requirements, we exploit partial symmetry within temporaries. It should be noted that the symmetry being exploited in this article is different from the symmetry typically observed in chemistry fields. One approach for exploiting symmetry in operations is to store only unique entries and devise algorithms which only use the unique entries of the symmetric operands [40]. By contrast, we exploit symmetry in operands by devising algorithms and storing the objects in such a way that knowledge of the symmetry of the operands is concealed from the implementation (allowing symmetric objects to be treated as nonsymmetric objects).

The contributions of this paper can be summarized as reducing storage and computational requirements of the sttsm operation for symmetric tensors by

- utilizing temporaries to reduce computational costs thereby avoiding redundant computations;
- using blocked algorithms and data structures to improve performance of the given computing environment;
- providing a framework for exploiting symmetry in symmetric tensors (and partial symmetry in temporaries) thereby reducing storage requirements.

The paper analyzes the computational and storage costs demonstrating that the added complexity of exploiting symmetry need not adversely impact the benefits derived from symmetry. An implementation shows that the insights can be made practical. The paper concludes by listing additional opportunities for generalizing advancements in the domain of linear algebra libraries to multilinear computation.

2. Preliminaries. We start with some basic notation, 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 (or m-way) tensors where each mode has dimension n; i.e., a tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$ can be thought of as an m-dimensional cube with n entries in each direction.

An element of $\mathcal{A}$ 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 $(\mathcal{A}, \mathcal{X}, \ldots)$. We denote the ith row of a matrix $\mathbf{A}$ by $\widehat{\mathbf{a}}_i^T$. If we transpose this row, we denote it as $\widehat{\mathbf{a}}_i$.

2.2. Partitioning. For our forthcoming discussions, it is useful to define the notion of a *partitioning* of a set $\mathcal{S}$. We say the sets $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{k-1}$ form a *partitioning* of $\mathcal{S}$ if

$$\mathcal{S}_i \cap \mathcal{S}_j = \emptyset \text{ for any } i, j \in \{0, \ldots, k-1\} \text{ with } i \neq j,$$
$$\mathcal{S}_i \neq \emptyset \text{ for any } i \in \{0, \ldots, k-1\},$$

and

$$\bigcup_{i=0}^{k-1} \mathcal{S}_i = \mathcal{S}.$$

2.3. Partial symmetry. It is possible that a tensor $\mathcal{A}$ may be symmetric in two or more modes, meaning that the entries are invariant to permutations of those modes. For instance, if $\mathcal{A}$ is a 3-way tensor and symmetric in all modes, then

$$\alpha_{i_0 i_1 i_2} = \alpha_{i_0 i_2 i_1} = \alpha_{i_1 i_0 i_2} = \alpha_{i_1 i_2 i_0} = \alpha_{i_2 i_0 i_1} = \alpha_{i_2 i_1 i_0}.$$

It may also be that $\mathcal{A}$ is only symmetric in a subset of the modes. For instance, suppose $\mathcal{A}$ is a 4-way tensor that is symmetric in modes $\mathcal{S} = \{1, 2\}$. Then

$$\alpha_{i_0 i_1 i_2 i_3} = \alpha_{i_0 i_2 i_1 i_3}.$$

We define this formally below.

Let $\mathcal{S}$ be a finite set. Define $\Pi_{\mathcal{S}}$ to be the set of all permutations on the set $\mathcal{S}$ where a permutation is viewed as a bijection from $\mathcal{S}$ to $\mathcal{S}$. Under this interpretation, for any $\pi \in \Pi_{\mathcal{S}}$, $\pi(x)$ is the resulting element of applying π to x.[1]

Let $\mathcal{S} \subseteq \{0, \ldots, m-1\}$, and define Π_S to be the set of all permutations on $\mathcal{S}$ as described above. We say an order-m tensor $\mathcal{A}$ is *symmetric in the modes in* $\mathcal{S}$ if

$$\alpha_{i'_0 i'_1 \cdots i'_{m-1}} = \alpha_{i_0 i_1 \cdots i_{m-1}}$$

for any index vector i' defined by

$$i'_j = \begin{cases} \pi(i_j) & \text{if } j \in \mathcal{S}, \\ i_j & \text{otherwise} \end{cases}$$

for $j = 0, \ldots, m-1$ and $\pi \in \Pi_{\mathcal{S}}$.

Technically speaking, this definition applies even in the trivial case where $\mathcal{S}$ is a singleton, which is useful for defining multiple symmetries.

2.4. Multiple symmetries. It is possible that a tensor may have symmetry in multiple sets of modes at once. As the tensor is not symmetric in all modes, yet still symmetric in some modes, we say the tensor is *partially* symmetric. For instance, suppose $\mathcal{A}$ is a 4-way tensor that is symmetric in modes $\mathcal{S}_0 = \{1, 2\}$ and also in modes $\mathcal{S}_1 = \{0, 3\}$. Then

$$\alpha_{i_0 i_1 i_2 i_3} = \alpha_{i_3 i_1 i_2 i_0} = \alpha_{i_0 i_2 i_1 i_3} = \alpha_{i_3 i_2 i_1 i_0}.$$

We define this formally below.

Let $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{k-1}$ be a partitioning of $\{0, \ldots, m-1\}$. We say an order-m tensor $\mathcal{A}$ has symmetries defined by the mode partitioning $\{S_i\}_{i=0}^{k-1}$ if

$$\alpha_{i'_0 i'_1 \cdots i'_{m-1}} = \alpha_{i_0 i_1 \cdots i_{m-1}}$$

for any index vector i' defined by

$$i'_j = \begin{cases} \pi_0(i_j) & \text{if } j \in \mathcal{S}_0, \\ \pi_1(i_j) & \text{if } j \in \mathcal{S}_1, \\ \quad \vdots & \\ \pi_{k-1}(i_j) & \text{if } j \in \mathcal{S}_{k-1} \end{cases}$$

for $j = 0, \ldots, m-1$ and $\pi_\ell \in \Pi_{\mathcal{S}_\ell}$ for $\ell = 0, \ldots, k-1$.

Technically, a tensor with no symmetry whatsoever still fits the definition above with $k = m$ and $|\mathcal{S}_i| = 1$ for $i = 0, \ldots, m-1$. If $k = 1$ and $\mathcal{S}_0 = \{0, \ldots, m-1\}$, then the tensor is *symmetric*. If $1 < k < m$, then the tensor is *partially symmetric*. Later, we look at partially symmetric tensors such that $\mathcal{S}_0 = \{0, \ldots, \ell\}$ and $|\mathcal{S}_i| = 1$ for $i = 1, \ldots, k-1$.

2.5. 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

$$\mathcal{C} := [\mathcal{A}; \underbrace{\mathbf{X}, \ldots, \mathbf{X}}_{m \text{ times}}] = \mathcal{A} \times_0 \mathbf{X} \times_1 \cdots \times_{m-1} \mathbf{X}, \tag{2.1}$$

[1] Throughout this paper, π should be interpreted as a permutation, not as a scalar quantity. All other lowercase Greek letters should be interpreted as scalar quantities.

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

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

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.

The sttsm operation is used in computing symmetric versions of Tucker and CP (notably the CP-opt) decompositions for symmetric tensors [22]. In the CP decomposition, the matrix $\mathbf{X}$ of the sttsm operation is a single vector. In the field of computational chemistry, the sttsm operation is used when transforming atomic integrals [8]. Many fields utilize a closely related operation to the sttsm operation, which can be viewed as the multiplication of a symmetric tensor in all modes *but one*. Problems such as calculating Nash equlibria for symmetric games [15] utilize this related operation. We focus on the sttsm operation not only to improve methods that rely on this exact operation, but also to gain insight for tackling related problems of symmetry in related operations.

3. The matrix case. We build intuition about the problem and its solutions by first looking at symmetric matrices (order-2 symmetric tensors).

3.1. The operation for $m = 2$. 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

$$\gamma_{j_0 j_1} = \sum_{i_0=0}^{n-1} \sum_{i_1=0}^{n-1} \alpha_{i_0 i_1} \chi_{j_0 i_0} \chi_{j_1 i_1}. \tag{3.1}$$

3.2. Simple algorithms for $m = 2$. 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 1 (top 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

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

where $\widehat{\mathbf{t}}_j = \mathbf{A}\widehat{\mathbf{x}}_j \in \mathbb{R}^n$ and $\widehat{\mathbf{x}}_j \in \mathbb{R}^n$ (recall that $\widehat{\mathbf{x}}_j$ 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 $\widehat{\mathbf{t}}$.

3.3. Blocked compact symmetric storage (BCSS) $m = 2$. 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 consider storing the upper triangular part. While for

Naive algorithms	Algorithms that reduce computation at the expense of extra workspace
$\mathcal{A}$ 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 $j_1 = 0, \ldots, p-1$
	  $\widehat{\mathbf{t}} := \widehat{\mathbf{t}}_{j_1} = \mathbf{A}\widehat{\mathbf{x}}_{j_1}$
	  for $j_0 = 0, \ldots, j_1$
	  $\gamma_{j_0 j_1} := \widehat{\mathbf{x}}_{j_0}^T \widehat{\mathbf{t}}$
	  endfor
	endfor

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

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

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

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

FIG. 1. *Algorithms for* $\mathcal{C} := [\mathcal{A}; \mathbf{X}, \ldots, \mathbf{X}]$ *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.*

matrices the savings is modest (and rarely exploited), the savings is more dramatic for tensors of higher order.

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 [18, 6]. Given that this complexity only increases with the tensor order, we do not pursue this idea.

Instead, we embrace an idea, storage by blocks, that was introduced into the `libflame` library [24, 41, 36] 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). We do not store the data these blocks represent explicitly; instead, we store information at these locations informing us how to obtain the required data. By doing this, we can retain a simple indexing scheme into $\mathbf{A}$ that avoids the complexity associated with storing only the unique entries. Although the diagonal blocks are themselves symmetric, we do 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 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 contrast, 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 2 illustrates how the storage for BCSS approaches the cost of storing only the upper triangular elements (here $n = 512$) as the number of blocks increases.

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

FIG. 2. *Storage savings factor of BCSS when* $n = 512$.

3.4. Algorithm by blocks for $m = 2$. 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),

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

Without loss of generality, $\bar{p} = p/b_{\mathbf{C}}$ is integral, 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

$$\mathbf{C}_{\bar{\jmath}_0\bar{\jmath}_1} = \begin{pmatrix} \mathbf{X}_{\bar{\jmath}_0 0} & \cdots & \mathbf{X}_{\bar{\jmath}_0(\bar{n}-1)} \end{pmatrix} \begin{pmatrix} \mathbf{A}_{00} & \cdots & \mathbf{A}_{0(\bar{n}-1)} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{(\bar{n}-1)0} & \cdots & \mathbf{A}_{(\bar{n}-1)(\bar{n}-1)} \end{pmatrix} \begin{pmatrix} \mathbf{X}^T_{\bar{\jmath}_1 0} \\ \vdots \\ \mathbf{X}^T_{\bar{\jmath}_1(\bar{n}-1)} \end{pmatrix}$$

(3.4)

$$= \sum_{\bar{\imath}_0=0}^{\bar{n}-1} \sum_{\bar{\imath}_1=0}^{\bar{n}-1} \mathbf{X}_{\bar{\jmath}_0\bar{\imath}_0} \mathbf{A}_{\bar{\imath}_0\bar{\imath}_1} \mathbf{X}^T_{\bar{\jmath}_1\bar{\imath}_1} = \sum_{\bar{\imath}_0=0}^{\bar{n}-1} \sum_{\bar{\imath}_1=0}^{\bar{n}-1} [\mathbf{A}_{\bar{\imath}_0\bar{\imath}_1}; \mathbf{X}_{\bar{\jmath}_0\bar{\imath}_0}, \mathbf{X}_{\bar{\jmath}_1\bar{\imath}_1}] \text{ (in tensor notation).}$$

This yields the algorithm in Figure 3, in which an analysis of its cost is also given. This algorithm avoids redundant computation, except within symmetric blocks on the diagonal. 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 1 (top right), we notice that every scalar has simply been replaced by a block (submatrix). The algorithm now computes a temporary matrix $\mathbf{T} = \mathbf{A}\mathbf{X}^T_{\bar{\jmath}_1:}$ instead of a temporary vector $\widehat{\mathbf{t}} = \mathbf{A}\widehat{\mathbf{x}}_{j_1}$ as in Figure 1 for each index $0 \le \bar{\jmath}_1 < \bar{p}$. It requires $n \times b_{\mathbf{C}}$ extra storage instead of n extra storage in addition to the storage for $\mathbf{C}$ and $\mathbf{A}$.

4. The 3-way case. We 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 order-m case in the next section.

4.1. The operation for $m = 3$. 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 symmetric

Algorithm	Ops (flops)	Total # of times executed	Temp. storage
for $\bar{j}_1 = 0, \ldots, \bar{p}-1$ $\underbrace{\begin{pmatrix} \mathbf{T}_0^T \\ \vdots \\ \mathbf{T}_{\bar{n}-1}^T \end{pmatrix}}_{\mathbf{T}} := \underbrace{\begin{pmatrix} \mathbf{A}_{00} & \cdots & \mathbf{A}_{0(\bar{n}-1)} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{(\bar{n}-1)0} & \cdots & \mathbf{A}_{(\bar{n}-1)(\bar{n}-1)} \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} \mathbf{X}_{\bar{j}_1 0}^T \\ \vdots \\ \mathbf{X}_{\bar{j}_1(\bar{n}-1)}^T \end{pmatrix}}_{\mathbf{X}_{\bar{j}_1}^T}$	$2b_{\mathbf{C}}n^2$	$\bar{p}$	$b_{\mathbf{C}}n$
for $\bar{j}_0 = 0, \ldots, \bar{j}_1$ $\mathbf{C}_{\bar{j}_0\bar{j}_1} := \left(\mathbf{X}_{\bar{j}_0 0} \ \cdots \ \mathbf{X}_{\bar{j}_0(\bar{n}-1)} \right) \begin{pmatrix} \mathbf{T}_0^T \\ \vdots \\ \mathbf{T}_{\bar{n}-1}^T \end{pmatrix}$	$2b_{\mathbf{C}}^2 n$	$\bar{p}(\bar{p}+1)/2$	
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. *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* $\bar{p} = \lceil p/b_{\mathbf{C}} \rceil$. *An expression using summations is given to help in identifying a pattern later on.*

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

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

4.2. Simple algorithms for $m = 3$. A naive algorithm is given in Figure 1 (middle left). The cheaper algorithm to its right is motivated by

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

where

$$\mathbf{T}^{(2)}_{i_2} \in \mathbb{R}^{n\times n} \quad \text{and} \quad \mathbf{T}^{(2)}_{i_2} = \boldsymbol{\mathcal{A}} \times_2 \widehat{\mathbf{x}}^T_{i_2},$$

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

$$\widehat{\mathbf{x}}^T_j \in \mathbb{R}^{1\times n}.$$

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 $n \times n$ and vector $\mathbf{t}$ of length n.

4.3. BCSS for $m = 3$. 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 $i_0 \le i_1 \le i_2$. For lack of a better term, we refer to this as the *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 block being a two-dimensional submatrix, a block for 3-way tensors becomes a 3-way subtensor. Partition tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{[3,n]}$ into cubical blocks $\boldsymbol{\mathcal{A}}_{\bar{\imath}_0\bar{\imath}_1\bar{\imath}_2}$ of size $b_{\boldsymbol{\mathcal{A}}} \times b_{\boldsymbol{\mathcal{A}}} \times b_{\boldsymbol{\mathcal{A}}}$:

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

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

where $\bar{n} = n/b_{\boldsymbol{\mathcal{A}}}$ (without loss of generality, assume $b_{\boldsymbol{\mathcal{A}}}$ divides n). These blocks are stored using some conventional method and the blocks lying outside the upper hypertriangular region are not stored. Once again, we do 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.

As summarized in Table 1, we see that while storing *only* the upper hypertriangular elements of the tensor $\boldsymbol{\mathcal{A}}$ requires $\binom{n+2}{3}$ storage, BCSS requires $b^3_{\boldsymbol{\mathcal{A}}}\binom{\bar{n}+2}{3}$ elements. However, since $\binom{\bar{n}+2}{3} b^3_{\boldsymbol{\mathcal{A}}} \approx \frac{\bar{n}^3}{3!} b^3_{\boldsymbol{\mathcal{A}}} = \frac{n^3}{6}$, we achieve a savings of approximately a factor of 6 if $\bar{n}$ is large enough, relative to storing all elements. Once again, we can apply the same storage method to $\boldsymbol{\mathcal{C}}$ for additional savings.

Blocks such that $\bar{\imath}_0 = \bar{\imath}_1 \neq \bar{\imath}_2$, $\bar{\imath}_0 = \bar{\imath}_2 \neq \bar{\imath}_1$, or $\bar{\imath}_1 = \bar{\imath}_2 \neq \bar{\imath}_0$ still have some symmetry and so are referred to as *partially* symmetric blocks.

4.4. Algorithm by blocks for $m = 3$. We now discuss an algorithm by blocks for the 3-way case. Partition $\boldsymbol{\mathcal{C}}$ and $\boldsymbol{\mathcal{A}}$ into blocks of size $b_{\boldsymbol{\mathcal{C}}} \times b_{\boldsymbol{\mathcal{C}}} \times b_{\boldsymbol{\mathcal{C}}}$ and $b_{\boldsymbol{\mathcal{A}}} \times b_{\boldsymbol{\mathcal{A}}} \times b_{\boldsymbol{\mathcal{A}}}$, respectively, and partition $\mathbf{X}$ into $b_{\boldsymbol{\mathcal{C}}} \times b_{\boldsymbol{\mathcal{A}}}$ blocks. Then, extending the insights we

TABLE 1
Storage requirements for a tensor $\mathcal{A}$ under different storage schemes.

	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

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 4, in which an analysis of its cost is also given. This algorithm avoids redundant computation, except for within blocks of $\mathcal{C}$ that are symmetric or partially symmetric. The algorithm computes temporaries $\mathcal{T}^{(2)} = \mathcal{A} \times_2 \mathbf{X}_{\bar{\jmath}_2:}$ and $\mathcal{T}^{(1)} = \mathcal{T}^{(2)} \times_1 \mathbf{X}_{\bar{\jmath}_1:}$ for each index, where $0 \le \bar{\jmath}_2 < \bar{p}$ and $0 \le \bar{\jmath}_1 \le \bar{\jmath}_2$. The algorithm requires $b_{\mathcal{C}} n^2 + b_{\mathcal{C}}^2 n$ extra storage (for $\mathcal{T}^{(2)}$ and $\mathcal{T}^{(1)}$, respectively), in addition to the storage for $\mathcal{C}$ and $\mathcal{A}$.

5. The m-way case. We now generalize to tensors $\mathcal{C}$ and $\mathcal{A}$ of any order.

5.1. The operation for order-m tensors. For general m, we have $\mathcal{C} := [\mathcal{A}; \mathbf{X}, \mathbf{X}, \ldots, \mathbf{X}]$, where $\mathcal{A} \in \mathbb{R}^{[m,n]}$, $\mathcal{C} \in \mathbb{R}^{[m,p]}$, and $[\mathcal{A}; \mathbf{X}, \mathbf{X}, \ldots \mathbf{X}] = \mathcal{A} \times_0 \mathbf{X} \times_1 \mathbf{X} \cdots \times_{m-1} \mathbf{X}$. In our discussion, $\mathcal{A}$ is a symmetric tensor, as is $\mathcal{C}$ by virtue of the operation applied to $\mathcal{A}$.

Recall that $\gamma_{j_0 j_1 \cdots j_{m-1}}$ denotes the $(j_0, j_1, \ldots, j_{m-1})$ element of the order-m tensor $\mathcal{C}$. Then, by simple extension of our previous derivations, we find that

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

5.2. Simple algorithms for general m. A naive algorithm with a cost of $(m+1)p^m n^m$ flops is given in Figure 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 (in terms of flops) in the m-way case should become obvious. Extending the cheaper algorithm in the 3-way case suggests the algorithm given in Figure 1 (bottom right). This algorithm requires

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

Algorithm	Ops (flops)	Total # of times executed	Temp. storage
for $\bar{j}_2 = 0, \ldots, \bar{p}-1$			
$\begin{pmatrix} \mathcal{T}^{(2)}_{00} & \cdots & \mathcal{T}^{(2)}_{0(\bar{n}-1)} \\ \vdots & \ddots & \vdots \\ \mathcal{T}^{(2)}_{(\bar{n}-1)0} & \cdots & \mathcal{T}^{(2)}_{(\bar{n}-1)(\bar{n}-1)} \end{pmatrix} :=$			
$\mathcal{A} \times_2 \begin{pmatrix} \mathbf{X}_{\bar{j}_2 0} & \cdots & \mathbf{X}_{\bar{j}_2(\bar{n}-1)} \end{pmatrix}$	$2b_{\mathcal{C}}n^3$	$\bar{p}$	$b_{\mathcal{C}}n^2$
for $\bar{j}_1 = 0, \ldots, \bar{j}_2$			
$\begin{pmatrix} \mathcal{T}^{(1)}_0 \\ \vdots \\ \mathcal{T}^{(1)}_{\bar{n}-1} \end{pmatrix} := \begin{pmatrix} \mathcal{T}^{(2)}_{00} & \cdots & \mathcal{T}^{(2)}_{0(\bar{n}-1)} \\ \vdots & \ddots & \vdots \\ \mathcal{T}^{(2)}_{(\bar{n}-1)0} & \cdots & \mathcal{T}^{(2)}_{(\bar{n}-1)(\bar{n}-1)} \end{pmatrix}$	$2b^2_{\mathcal{C}}n^2$	$\bar{p}(\bar{p}+1)/2 = \binom{\bar{p}+1}{2}$	$b^2_{\mathcal{C}}n$
$\times_1 \begin{pmatrix} \mathbf{X}_{\bar{j}_1 0} & \cdots & \mathbf{X}_{\bar{j}_1(\bar{n}-1)} \end{pmatrix}$			
for $\bar{j}_0 = 0, \ldots, \bar{j}_1$			
$\mathcal{C}_{\bar{j}_0\bar{j}_1\bar{j}_2} :=$			
$\begin{pmatrix} \mathcal{T}^{(1)}_0 \\ \vdots \\ \mathcal{T}^{(1)}_{\bar{n}-1} \end{pmatrix} \times_0 \begin{pmatrix} \mathbf{X}_{\bar{j}_0 0} & \cdots & \mathbf{X}_{\bar{j}_0(\bar{n}-1)} \end{pmatrix}$	$2b^3_{\mathcal{C}}n$	$\frac{\bar{p}(\bar{p}+1)(\bar{p}+2)}{6} = \binom{\bar{p}+2}{3}$	
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: $b_{\mathcal{C}}n^2 + b^2_{\mathcal{C}}n = \sum_{d=0}^{1} \left(b_{\mathcal{C}}^{d+1} n^{2-d} \right)$ entries			

FIG. 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* $\bar{p} = \lceil p/b_{\mathcal{C}} \rceil$. *An expression using summations is given to help in identifying a pattern later on.*

at the expense of requiring workspace for temporary tensors of order 1 through $m-1$ with modes of dimension n.

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

As summarized in Table 1, storing *only* the upper hypertriangular elements of the tensor $\mathcal{A}$ requires $\binom{n+m-1}{m}$ storage, and BCSS requires $\binom{\bar{n}+m-1}{m} b^m_{\mathcal{A}}$ elements which achieves a savings factor of $m!$ (if $\bar{n}$ is large enough).

Although the approximation $\binom{\bar{n}+m-1}{m} b^m_{\mathcal{A}} \approx \frac{n^m}{m!}$ is used, the lower-order terms have a significant effect on the actual storage savings factor. In Figure 5, we show the actual storage savings of BCSS (including storage required for metadata entries) over storing all entries of a symmetric tensor. Examining Figure 5, we see that as we increase m, a larger $\bar{n}$ is required to have the actual storage savings factor approach the theoretical factor. While this figure only shows the results for a particular value of $b_{\mathcal{A}}$, the effect applies to all values of $b_{\mathcal{A}}$. This idea of blocking has been used in many projects including the TCE project [7, 31, 14].

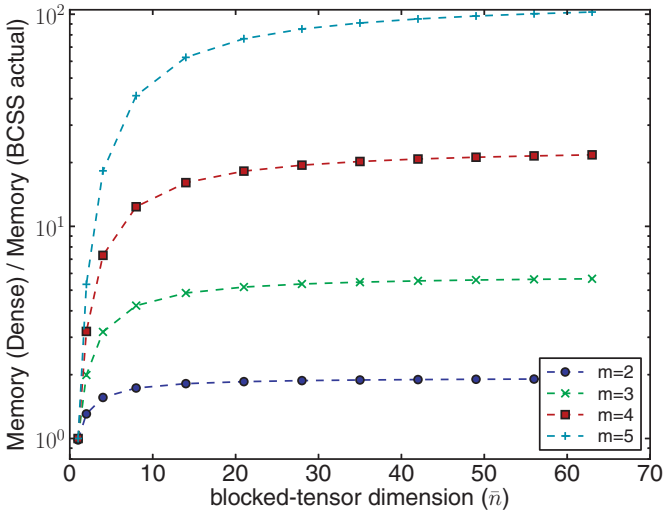

FIG. 5. *Actual storage savings of BCSS on* $\mathcal{A}$ *with block dimension* $b_{\mathcal{A}} = 8$. *This includes the number of entries required for associated metadata.*

5.4. Algorithm by blocks for general m. For $\mathcal{C}$ and $\mathcal{A}$ for general m stored using BCSS, we discuss how to compute with these blocks. Assume the partioning discussed above. Then,

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

This yields the algorithm given in Figure 6, which avoids much redundant computation, except for within blocks of $\mathcal{C}$ that are symmetric or partially symmetric. The algorithm computes temporaries

$$\begin{aligned}\mathcal{T}^{(m-1)} &= \mathcal{A}\times_{m-1}\mathbf{X}_{\bar{\jmath}_{m-1}:},\\ \mathcal{T}^{(m-2)} &= \mathcal{T}^{(m-1)}\times_{m-2}\mathbf{X}_{\bar{\jmath}_{m-2}:},\\ &\vdots\\ \mathcal{T}^{(1)} &= \mathcal{T}^{(2)}\times_1\mathbf{X}_{\bar{\jmath}_1:}\end{aligned}$$

for each index, where $0 \le \bar{\jmath}_1 \le \bar{\jmath}_2 \le \cdots \le \bar{\jmath}_{m-1} < \bar{p}$. This algorithm requires $b_{\mathcal{C}}n^{m-1}+b_{\mathcal{C}}^2n^{m-2}+\cdots+b_{\mathcal{C}}^{m-2}n$ extra storage (for $\mathcal{T}^{(m-1)}$ through $\mathcal{T}^{(1)}$, respectively), in addition to the storage for $\mathcal{C}$ and $\mathcal{A}$.

We realize this approach can result in a small loss of symmetry (due to numerical instability) within blocks. We do not address this effect at this time as the asymmetry only becomes a factor when the resulting tensor is used in subsequent operations. A postprocessing step can be applied to correct any asymmetry in the resulting tensor.

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

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

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

FIG. 6. *Algorithm by blocks for computing* $\mathcal{C} := [\mathcal{A}; \mathbf{X}, \ldots, \mathbf{X}]$. *The algorithm assumes that* $\mathcal{C}$ *is partitioned into blocks of size* $[m, b_{\mathcal{C}}]$ *with* $\bar{p} = \lceil p/b_{\mathcal{C}} \rceil$.

6. Exploiting partial symmetry. We have shown how to reduce the complexity of the sttsm operation by $O(m!)$ in terms of storage. In this section, we describe how to achieve the $O((m+1)!/2^m)$ level of reduction in computation.

6.1. Partial symmetry. Recall that in Figure 6 we utilized a series of temporaries to compute the sttsm operation. To perform the computation, we explicitly formed the temporaries $\mathcal{T}^{(k)}$ and did not take advantage of any symmetry in the objects' entries. Because of this, we were only able to see an $O(m)$ reduction in storage and computation.

However, as we now show, there exists partial symmetry within each temporary that we can exploit to reduce storage and computation as we did for the output tensor $\mathcal{C}$. Exploiting this partial symmetry allows the proposed algorithm to match the theoretical reduction in storage and computation.

THEOREM 6.1. *Given an order-m tensor* $\mathcal{A} \in \mathbb{R}^{I_0 \times I_1 \times \cdots \times I_{m-1}}$ *that has modes* 0 *through* k *symmetric (thus* $I_0 = I_1 = \cdots = I_k$*), then* $\mathcal{C} = \mathcal{A} \times_k \mathbf{X}$ *has modes* 0 *through* $k-1$ *symmetric.*

Proof. We prove this by construction of $\mathcal{C}$.

Since $\mathcal{A}$ has modes 0 through k symmetric, we know (from section 2.4) that

$$\alpha_{i'_0 i'_1 \cdots i'_k i_{k+1} \cdots i_{m-1}} = \alpha_{i_0 i_1 \cdots i_k i_{k+1} \cdots i_{m-1}}$$

under the relevant permutations. We wish to show that

$$\gamma_{i'_0 i'_1 \cdots i'_{k-1} j_k \cdots j_{m-1}} = \gamma_{i_0 i_1 \cdots i_{k-1} j_k \cdots j_{m-1}}$$

for all indices in $\mathcal{C}$.

$$\begin{aligned}\gamma_{i'_0 i'_1 \cdots i'_{k-1} j_k \cdots j_{m-1}} &= \sum_{\ell=0}^{n-1} \alpha_{i'_0 i'_1 \cdots i'_{k-1} \ell j_{k+1} \cdots j_{m-1}} \chi_{j_k \ell} \\ &= \sum_{\ell=0}^{n-1} \alpha_{i_0 i_1 \cdots i_{k-1} \ell j_{k+1} \cdots j_{m-1}} \chi_{j_k \ell} = \gamma_{i_0 i_1 \cdots i_{k-1} j_k \cdots j_{m-1}}.\end{aligned}$$

Since $\gamma_{i'_0 i'_1 \cdots i'_{k-1} j_k \cdots j_{m-1}} = \gamma_{i_0 i_1 \cdots i_{k-1} j_k \cdots j_{m-1}}$ for all indices in $\mathcal{C}$, we can say that $\mathcal{C}$ has modes 0 through $k-1$ symmetric. ∎

By applying Theorem 6.1 to the algorithm in Figure 6, we observe that all temporaries of the form $\mathcal{T}^{(k)}$ formed have modes 0 through $k-1$ symmetric. It is this partial symmetry we exploit to further reduce storage and computational complexity.[2]

6.2. Storage. A generalization of the BCSS scheme can be applied to the partially symmetric temporary tensors as well. To do this, we view each temporary tensor as being comprised of a group of symmetric modes and a group of nonsymmetric modes. There is once again opportunity for storage savings as the symmetric indices have redundancies. As in the BCSS case for symmetric tensors, unique blocks are stored and metadata indicating how to transform stored blocks to the corresponding block are stored for all redundant block entries.

6.3. Computation. Recall that each temporary is computed via $\mathcal{T}^{(k)} = \mathcal{T}^{(k+1)} \times_{k+1} \mathbf{B}$, where $\mathcal{T}^{(k)}$ and $\mathcal{T}^{(k+1)}$ have associated symmetries ($\mathcal{T}^{(m)} = \mathcal{A}$ when computing $\mathcal{T}^{(m-1)}$), and $\mathbf{B}$ is some matrix. We can rewrite this operation as

$$\mathcal{T}^{(k)} = \mathcal{T}^{(k+1)} \times_{k+1} \mathbf{B} = \mathcal{T}^{(k+1)} \times_0 \mathbf{I} \times_1 \cdots \times_k \mathbf{I} \times_{k+1} \mathbf{B} \times_{k+2} \mathbf{I} \times_{k+3} \cdots \times_{m-1} \mathbf{I},$$

where $\mathbf{I}$ is the first p rows of the $n \times n$ identity matrix. An algorithm akin to that of Figure 6 can be created (care is taken to only update unique output blocks) to perform the necessary computation. Of course, computing with the identity matrix is wasteful and, therefore, we only implicitly compute with the identity matrix to save on computation.

6.4. Analysis. Utilizing this optimization allows us to arrive at the final cost functions for storage and computation shown in Table 2.

Taking, for example, the computational cost (assuming $n = p$ and $b_{\mathcal{A}} = b_{\mathcal{C}}$), we have the following expression for the cost of the BCSS algorithm:

$$\begin{aligned}&2\bar{n} b_{\mathcal{C}} b_{\mathcal{A}}^m \sum_{d=0}^{m-1} \binom{\bar{p}+d}{d+1} \binom{\bar{n}+m-d-2}{m-d-1} \left(\frac{b_{\mathcal{C}}}{b_{\mathcal{A}}}\right)^d \\ &= 2n b_{\mathcal{A}}^m \sum_{d=0}^{m-1} \binom{\bar{n}+d}{d+1} \binom{\bar{n}+m-(d+1)-1}{m-(d+1)} (1)^d \\ &\approx 2n b_{\mathcal{A}}^m \binom{2\bar{n}+m-2}{m} \approx 2n b_{\mathcal{A}}^m \frac{(2\bar{n})^m}{m!} = \frac{(2n)^{m+1}}{m!}.\end{aligned}$$

To achieve this approximation, the Vandermonde identity, which states that

$$\binom{m+n}{r} = \sum_{k=0}^{r} \binom{m}{k} \binom{n}{r-k},$$

was employed.

[2] It is true that the other modes *may* have symmetry as well, however, in general this is not the case, and therefore we do not explore exploiting this symmetry.

TABLE 2

Costs associated with different algorithms for computing $\mathcal{C} = [\mathcal{A}; \mathbf{X}, \ldots, \mathbf{X}]$. *The BCSS column takes advantage of partial symmetry within the temporaries. The* $\bar{n}^m$, $\bar{p}^m$, *and* $\bar{n}^{d+1}$ *terms correspond to the number of metadata elements associated with our choice of storage scheme. The term memops refers to memory operations.*

	BCSS	Dense
$\mathcal{A}$ (elements)	$b_{\mathcal{A}}^m \binom{\bar{n}+m-1}{m} + \bar{n}^m$	n^m
$\mathcal{C}$ (elements)	$b_{\mathcal{C}}^m \binom{\bar{p}+m-1}{m} + \bar{p}^m$	p^m
$\mathbf{X}$ (elements)	pn	pn
All temporaries (elements)	$b_{\mathcal{C}} b_{\mathcal{A}}^{m-1} \sum_{d=0}^{m-2} \left(\binom{\bar{n}+m-d-2}{m-d-1} \left(\frac{b_{\mathcal{C}}}{b_{\mathcal{A}}}\right)^d + \bar{n}^{d+1} \right)$	$pn^{m-1} \sum_{d=0}^{m-2} \left(\frac{p}{n}\right)^d$
Computation (flops)	$2\bar{n} b_{\mathcal{C}} b_{\mathcal{A}}^m \sum_{d=0}^{m-1} \binom{\bar{p}+d}{d+1} \binom{\bar{n}+m-d-2}{m-d-1} \left(\frac{b_{\mathcal{C}}}{b_{\mathcal{A}}}\right)^d$	$2pn^m \sum_{d=0}^{m-1} \left(\frac{p}{n}\right)^d$
Permutation (memops)	$\left(\bar{n} + 2\frac{b_{\mathcal{C}}}{b_{\mathcal{A}}}\right) b_{\mathcal{A}}^m \sum_{d=0}^{m-1} \binom{\bar{p}+d}{d+1} \binom{\bar{n}+m-d-2}{m-d-1} \left(\frac{b_{\mathcal{C}}}{b_{\mathcal{A}}}\right)^d$	$\left(1 + 2\frac{p}{n}\right) n^m \sum_{d=0}^{m-1} \left(\frac{p}{n}\right)^d$

TABLE 3

Approximate costs associated with different algorithms for computing $\mathcal{C} = [\mathcal{A}; \mathbf{X}, \ldots, \mathbf{X}]$. *The BCSS column takes advantage of partial symmetry within the temporaries. In the above costs, it is assumed that the tensor dimensions and block dimensions of* $\mathcal{A}$ *and* $\mathcal{C}$ *are equal, i.e.,* $n = p$ *and* $b_{\mathcal{A}} = b_{\mathcal{C}}$. *We assume* $n^m \gg \bar{n}^m$.

	BCSS	Dense
$\mathcal{A}$, $\mathcal{C}$ (elements)	$b_{\mathcal{A}}^m \binom{\bar{n}+m-1}{m}$	n^m
$\mathbf{X}$ (elements)	n^2	n^2
All temporaries (elements)	$\frac{n^m}{m!}$	$(m-1)n^m$
Computation (flops)	$\frac{(2n)^{m+1}}{m!}$	$2mn^{m+1}$
Permutation (memops)	$(\bar{n}+2)\frac{(2n)^m}{m!}$	$3mn^m$

Using similar approximations, we arrive at the estimates summarized in Table 3.

Comparing this computational cost to that of the dense algorithm (as given in Table 3), we see that the BCSS algorithm achieves a reduction of

$$\frac{\text{dense cost}}{\text{BCSS cost}} = \frac{2mn^{m+1}}{\left(\frac{(2n)^{m+1}}{m!}\right)} \approx \frac{(m+1)!}{2^m}$$

in terms of computation.

6.5. Analysis relative to minimum. As we are storing some elements redundantly, it is important to compare how the algorithm performs compared to the case where we store no extra elements, that is, we only compute the unique entries of $\mathcal{C}$. Assuming $\mathcal{A} \in \mathbb{R}^{[m,n]}$, $\mathcal{C} \in \mathbb{R}^{[m,p]}$, $p = n$, and $b_{\mathcal{A}} = b_{\mathcal{C}} = 1$, the cost of computing the sttsm operation is

$$2n \sum_{d=0}^{m-1} \binom{n+d}{d+1} \binom{n+m-d-2}{m-d-1} \approx 2n \binom{2n+m-2}{m} \approx 2n \frac{(2n)^m}{m!} = \frac{(2n)^{m+1}}{m!},$$

which is of the same order as our blocked algorithm.

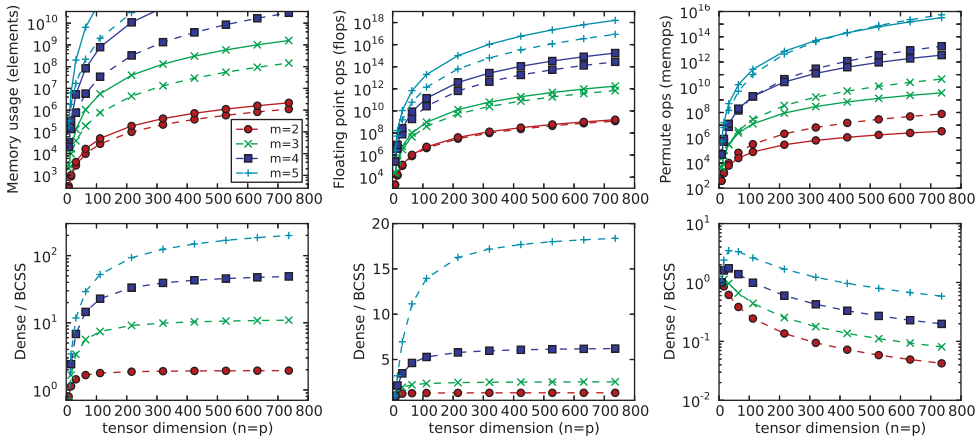

FIG. 7. *Comparison of dense to BCSS algorithms for fixed block size. Solid and dashed lines correspond to dense and BCSS, respectively. From left to right: storage requirements, cost from computation (flops), cost from permutations (memops). For these graphs* $b_{\mathcal{A}} = b_{\mathfrak{C}} = 8$.

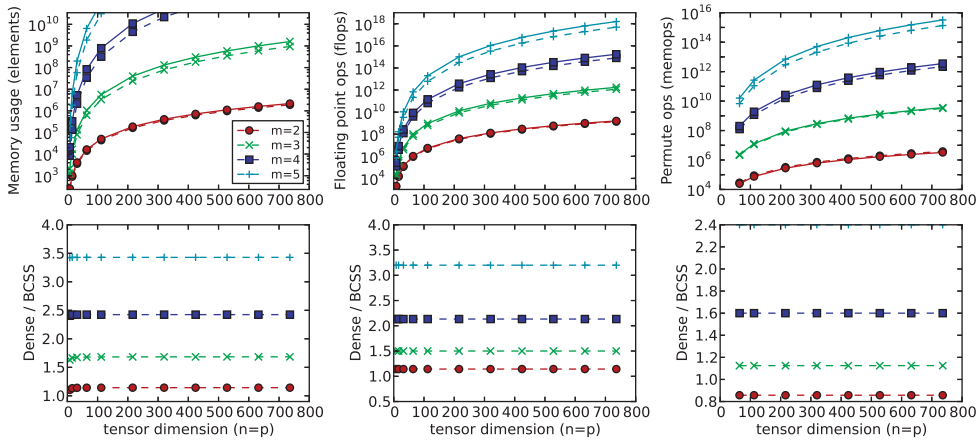

FIG. 8. *Comparison of dense to BCSS algorithms for fixed number of blocks. Solid and dashed lines correspond to dense and BCSS, respectively. From left to right: storage requirements, cost from computation (flops), cost from permutations (memops). Here* $\bar{n} = \bar{p} = 2$.

6.6. Summary. Figures 7–8 illustrate the insights discussed in this section. The (exact) formulas developed for storage, flops, and memops are used to compare and contrast dense storage with BCSS.

In Figure 7, 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 ($b_{\mathcal{A}} = b_{\mathfrak{C}} = 8$). 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 8, 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 issue 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.

7. 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. With that said, another primary concern is ensuring the overall time of computation is reduced. To achieve this, a reduction in the number of flops as well as an implementation that computes the necessary operations efficiently are both desired. Second to that is the desire to reduce the number of flops to the minimum required. The provided analysis shows that our algorithms perform the minimum number of flops (under approximation). Although our algorithms do not yet perform these operations efficiently, our results show that we are still able to reduce the computation time (in some cases significantly).

7.1. 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 and tensors. The implementations were linked to the OpenBLAS 0.1.1 library [1, 39], a fork of the GotoBLAS2 implementation of the BLAS [17, 16]. As noted, most of the 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. For instance, as argued previously, we know that tensor permutations can dominate the time spent computing the sttsm operation. These experiments make no attempt to reduce the number of tensor permutations required when computing a single block of the output. Algorithms reducing the effect of tensor permutations have been specialized for certain tensor operations and have been shown to greatly increase the performance of routines using them [26, 34]. Much room for improvement remains.

7.2. Implementation. The implementation was coded in a style inspired by the `libflame` library [41, 36] and can be found in the Google Code `tlash` project (http://code.google.com/p/tlash). An API similar to the FLASH API [24] 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 those that store and compute with the tensor densely, ignoring symmetry.

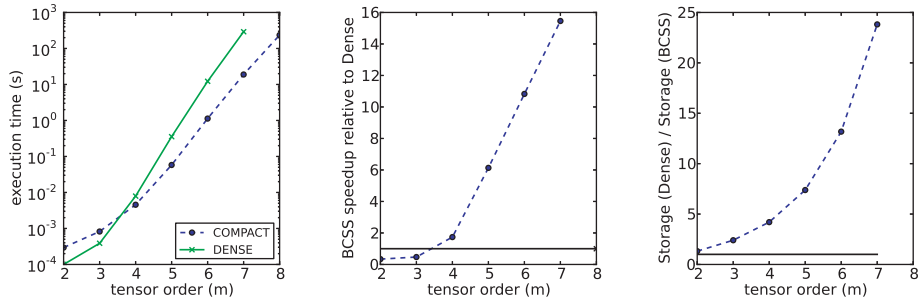

FIG. 9. *Experimental results when $n = p = 16$, $b_{\mathcal{A}} = b_{\mathcal{C}} = 8$, and the tensor order, m, is varied. For $m = 8$, storing $\mathcal{A}$ and $\mathcal{C}$ without taking advantage of symmetry requires too much memory. The solid black line is used to indicate a unit ratio.*

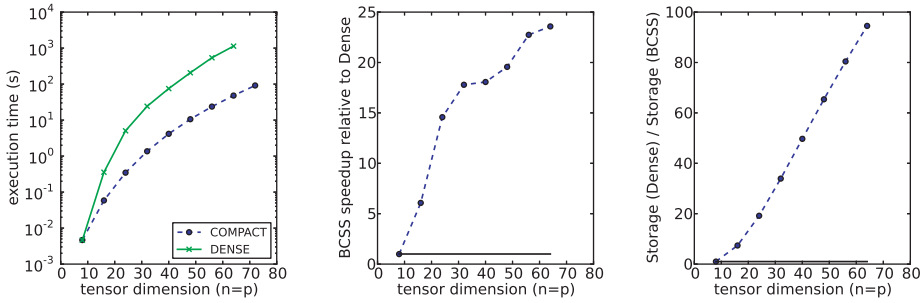

FIG. 10. *Experimental results when the order $m = 5$, $b_{\mathcal{A}} = b_{\mathcal{C}} = 8$, and the tensor dimensions $n = p$ are varied. For $n = p = 72$, storing $\mathcal{A}$ and $\mathcal{C}$ without taking advantage of symmetry requires too much memory. The solid black line is used to indicate a unit ratio.*

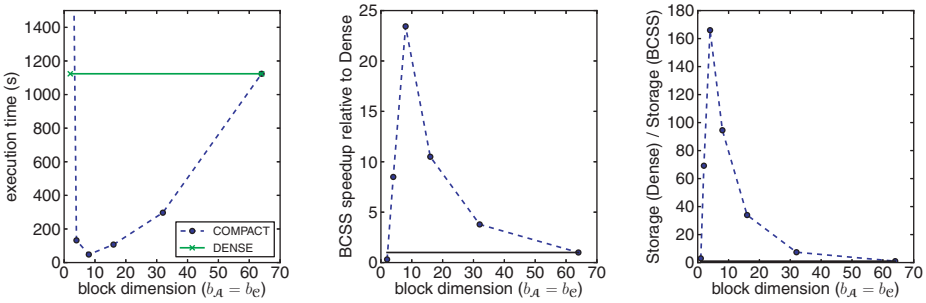

FIG. 11. *Experimental results when the order $m = 5$, $n = p = 64$, and the block dimensions $b_{\mathcal{A}} = b_{\mathcal{C}}$ are varied. The solid black line is used to indicate a unit ratio.*

7.3. Results. Figures 9–12 show results from executing our implementation on the target architecture. The dense algorithm does not take advantage of symmetry nor blocking of the data objects, whereas the BCSS algorithm takes advantage of both. 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 not including storage required for metadata.

For the experiments reported in Figure 9 we fix the dimensions n and p, and the block sizes $b_{\mathcal{A}}$ and $b_{\mathcal{C}}$, and vary the tensor order m. Based on experiment, the BCSS algorithm begins to outperform the dense algorithm after the tensor order is

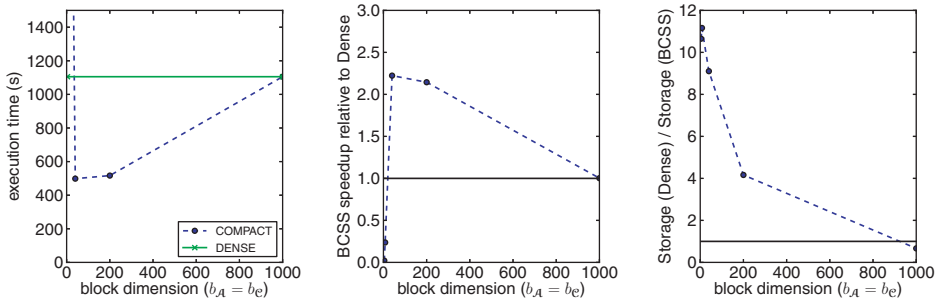

FIG. 12. *Experimental results when the order $m = 3$, $n = p = 1000$, and the block dimensions $b_{\mathcal{A}} = b_{\mathcal{C}}$ are varied. The solid black line is used to indicate a unit ratio.*

greater than or equal to 4. This effect for small m should be understood in context of the experiments performed. As $n = p = 16$, the problem size is quite small (the problems for $m = 2$ and $m = 3$ are equivalent to matrices of size 16×16 and 64×64, respectively) reducing the benefit of storage by blocks (as storing the entire matrix contiguously requires minor space overhead but benefits greatly from more regular data access). Since such small problems do not provide useful comparisons for the reader, the results of using the BCSS algorithm with problem parameters $m = 3$, $n = p = 1000$, and varied block dimensions are given in Figure 12. Figure 12 shows that the BCSS algorithm is able to outperform the dense algorithm given a large enough problem size and an appropriate block size. Additionally, notice that BCSS allows larger problems to be solved; the dense algorithm was unable to compute the result when an order-8 tensor was given as input due to an inability to store the problem in memory.

Our model predicts that the BCSS algorithm should achieve an $O\left((m+1)!/2^m\right)$ speedup over the dense algorithm. Although it appears that our experiments are only achieving a linear speedup over the dense algorithm, this is because the values of m are so small that the predicted speedup factor is approximately linear with respect to m. In terms of storage savings, we would expect the BCSS algorithm to have an $O\left(m!\right)$ reduction in space over the dense algorithm. The fact that we are not seeing this in the experiments is because the block dimensions $b_{\mathcal{A}}$ and $b_{\mathcal{C}}$ are relatively large when compared to the tensor dimensions n and p, meaning the BCSS algorithm does not have as great an opportunity to reduce storage requirements.

In Figure 10 we fix the order m, the block sizes $b_{\mathcal{A}}$, 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 experiments show a roughly linear speedup when viewed relative to n with perhaps a slight leveling off effect towards larger problem dimensions. We would expect the BCSS algorithm to approach a maximum speedup relative to the dense algorithm. According to Figure 7, we would expect the speedup of the BCSS algorithm over the dense algorithm to level off completely when our problem dimensions (n and p) are on the order of 400 to 600. Unfortunately, due to space limitations we were unable to test beyond the $n = p = 64$ problem dimension and therefore were unable to completely observe the leveling off effect in the speedup.

In Figure 11 we fix m, n, and p, and vary the block sizes $b_{\mathcal{A}}$ and $b_{\mathcal{C}}$. The rightmost point on the axis corresponds to the dense case (as $b_{\mathcal{A}} = n = b_{\mathcal{C}} = p$) and the leftmost 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. Further, the BCSS algorithm performs as well or worse than

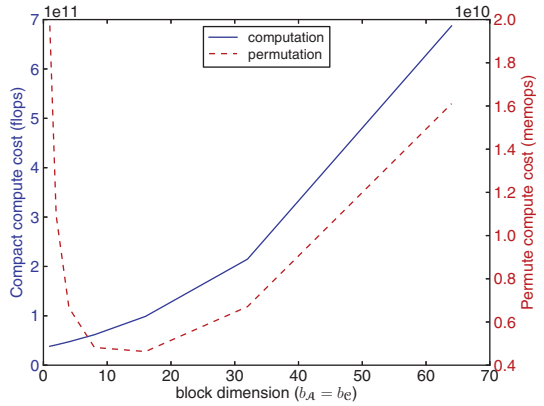

FIG. 13. *Comparison of computational cost to permutation cost where $m = 5$, $n = p = 64$, and tensor block dimension ($b_{\mathcal{A}}$, $b_{\mathcal{C}}$) is varied. The solid line represents the number of flops (due to computation) required for a given problem (left axis), and the dashed line represents the number of memops (due to permutation) required for a given problem (right axis).*

the dense counterpart at the two endpoints in the graph. This is expected toward the right of the figure as the BCSS algorithm reduces to the dense algorithm, however, the left of the figure requires a different explanation.

In Figure 13, we illustrate (with predicted flop and memop counts) that there exists a point where smaller block dimensions *dramatically* increases the number of memops required to compute the operation. Although a smaller block dimension results in fewer flops required for computing, the number of memops required increases significantly more. As memops are typically significantly more expensive than flops, we can expect that picking too small a block dimension can be expected to drastically degrade overall performance.

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

This initial study exposes many new research opportunities for extending insights from the field of high-performance linear algebra to multilinear computation, which we believe to be the real contribution of this paper. We finish by discussing some of these opportunities.

Optimizing tensor permutations. In our work, we made absolutely no attempt to optimize the tensor permutation operation. Without doubt, a careful study of how to organize these tensor 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 misses [17], which cause the core to stall for a hundred or more cycles.

Optimized kernels/avoiding tensor permutations. A better way to mitigate the tensor 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 [37], 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 creating a BLAS-like library for blocked tensor operations. One alternative for this is to apply the techniques developed as part of the PHiPAC [9], TCE, SPIRAL [28], or ATLAS [38] 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:

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

Then

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

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 multilinear 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.

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 the SuperMatrix [29] runtime does for the `libflame` library, or PLASMA [2] does for its tiled algorithms.

We did not yet pursue this because at the moment the permutations contribute a significant overhead to the overall computation which we speculate consumes signifi-

cant bandwidth. As a result, parallelization does not make sense until the cost of the permutations is mitigated.

Exploiting accelerators. In a large number of papers [19, 20, 25, 2], 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 CTF [33], or that we build on our own experience with distributed memory libraries for dense matrix computations, the PLAPACK [35] and Elemental [27] libraries, to develop a new distributed memory tensor library.

General multilinear 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.

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

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

This operation is typically computed by casting it as a matrix-matrix multiplication for which high-performance implementations are available as part of the 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-order/higher-order tensor.

As an example, consider an order-4 tensor $\mathcal{C} \in \mathbb{R}^{I_0 \times I_1 \times I_2 \times I_3}$. We can view $\mathcal{C}$ 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 $\mathcal{C}$ are both mapped

to a one-dimensional vector using column-major order and its higher-dimensional extension (which we call dimensional order), both are stored identically.

The need for permutation. If we instead view our example $\mathcal{C} \in \mathbb{R}^{I_0 \times I_1 \times I_2 \times I_3}$ 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 a rearrangement of the data is required. This is because, in general, mapping $\mathbf{C}$ and $\mathcal{C}$ to memory using dimensional order will not, in general, 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 $\mathcal{A} \in \mathbb{R}^{I_0 \times \cdots \times I_{m-1}}$ and the desired permutation of indices of $\mathcal{A}$, π, we define the transformation $\mathcal{C} = \text{permute}(\mathcal{A}, \pi)$ that yields $\mathcal{C} \in \mathbb{R}^{I_{\pi_0} \times I_{\pi_1} \times \cdots \times I_{\pi_{m-1}}}$ so that $\mathcal{C}_{i'_0 \cdots i'_{m-1}} = \mathcal{A}_{i_0 \cdots i_{m-1}}$, where i' corresponds to the result of applying the permutation π to i. The related operation ipermute inverts this transformation when supplied π so that $\mathcal{C} = \mathit{ipermute}(\mathcal{A}, \pi)$ yields $\mathcal{C} \in \mathbb{R}^{I_{\pi_0^{-1}} \times I_{\pi_1^{-1}} \times \cdots \times I_{\pi_{m-1}^{-1}}}$, where $\mathcal{C}_{i'_0 \cdots i'_{m-1}} = \mathcal{A}_{i_0 \cdots i_{m-1}}$, where i' corresponds to the result of applying the permutation π^{-1} to i.

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 (metadata) for a block in each element of the top-level tensor. These metadata add to memory requirements. In our current implementation, the top-level tensor of metadata is itself a dense tensor. The metadata in the upper hypertriangular tensor describe stored blocks. The metadata 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 show that although the metadata can potentially require considerable space, this can be easily mitigated. We use $\mathcal{A}$ as an example.

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

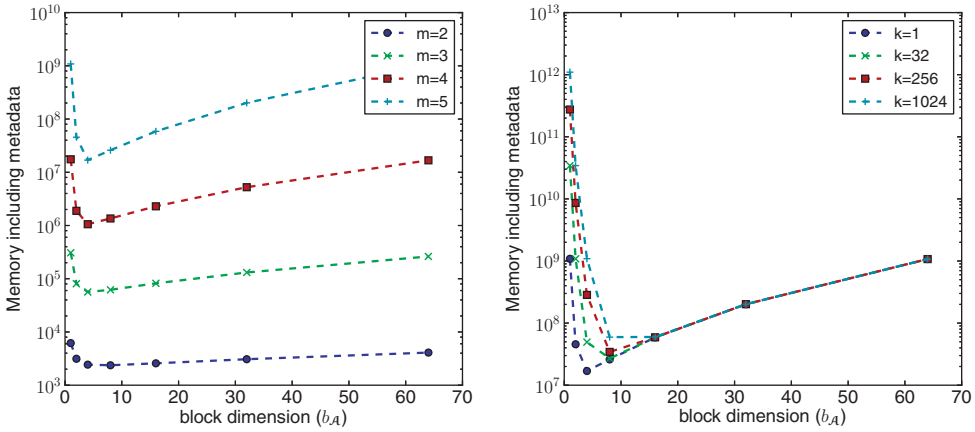

FIG. 14. *Left: storage requirements of* $\mathcal{A} \in \mathbb{R}^{[m,64]}$ *as block dimension changes. Right: storage requirements of* $\mathcal{A} \in \mathbb{R}^{[5,64]}$ *for different choices for the metadata stored per block, measured by* k*, as block dimension changes.*

$\mathcal{A}$ with BCSS is

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

k is a constant representing the amount of storage required for the metadata associated with one block, in floats. Obviously, these metadata are of a different data type, but floats will be our unit.

There is a trade-off between the cost for storing the metadata and the actual entries of $\mathcal{A}$, parameterized by the block size $b_{\mathcal{A}}$:

- if $b_{\mathcal{A}} = n$, then we only require a minimal amount of memory for metadata, k floats, but must store all entries of $\mathcal{A}$ 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 $b_{\mathcal{A}} = 1$, then $\bar{n} = n$ and we must store metadata 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 $b_{\mathcal{A}} = \sqrt{n}$, then $\bar{n} = \sqrt{n}$ and the total storage required to store $\mathcal{A}$ with BCSS is

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

which, provided that $k \ll \frac{n^{\frac{m}{2}}}{2}$, 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 14, we illustrate that as long as we pick a block dimension that is greater than 4, we avoid incurring extra costs due to metadata 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 metadata slowly shifts the optimal block dimension choice towards the dense storage case.

Acknowledgment. We would like to thank Grey Ballard for his insights into restructuring many parts of this paper.

REFERENCES

[1] *OpenBLAS*, http://xianyi.github.com/OpenBLAS/ (2012).

[2] 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*, J. Phys. Conf. Ser., 180 (2009), 012037.

[3] 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*, 3rd ed., Software Environ. Tools 9, SIAM, Philadelphia, 1999.

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

[5] B. W. Bader, T. G. Kolda, et al., *Matlab Tensor Toolbox version* 2.5, http://www.sandia.gov/∼tgkolda/TensorToolbox/ (2012).

[6] G. Ballard, T. G. Kolda, and T. Plantenga, *Efficiently computing tensor eigenvalues on a GPU*, in Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Ph.D. Forum, IEEE, Piscataway, NJ, 2011, pp. 1340–1348.

[7] 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 Proc. IEEE, 93, 2005, pp. 276–292.

[8] C.F Bender, *Integral transformations. A bottleneck in molecular quantum mechanical calculations*, J. Comput. Phys., 9 (1972), pp. 547–554.

[9] 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 11th International Conference on Supercomputing, ACM, New York, 1997, pp. 340–347.

[10] 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 Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 120–127.

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

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

[13] Eigenvector Research, Inc., *PLS Toolbox*, http://www.eigenvector.com/software/pls_toolbox.htm (2012).

[14] E. Epifanovsky, M. Wormit, T. Kus, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, and A. I. Krylov, *New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations*, J. Comput. Chem., 34 (2013), pp. 2293–2309.

[15] S. F. Cheng, D. M. Reeves, Y. Vorobeychik, and M. P. Wellman, *Notes on equilibria in symmetric games*, in Proceedings of the 6th International Workshop On Game Theoretic And Decision Theoretic Agents (GTDT), 2004, pp. 71–78.

[16] K. Goto and R. van de Geijn, *High-performance implementation of the level-*3 *BLAS*, ACM Trans. Math. Software, 35 (2008), 4.

[17] K. Goto and R. A. van de Geijn, *Anatomy of high-performance matrix multiplication*, ACM Trans. Math. Software, 34 (2008), 12.

[18] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, *FLAME: Formal linear algebra methods environment*, ACM Trans. Math. Software, 27 (2001), pp. 422–455.

[19] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, and F. G. Van Zee, *The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations*, J. Parallel Distrib. Comput., 72 (2012), pp. 1134 – 1143.

[20] F. D. Igual, G. Quintana-Ortí, and R. van de Geijn, *Level-3 BLAS on a GPU: Picking the Low Hanging Fruit, FLAME Working Note #37*, Technical report DICC 2009-04-01, Universidad Jaume I, Depto. de Ingenieria y Ciencia de Computadores., Castellón, Spain, 2009.

[21] M. Ishteva, P. A. Absil, and P. Van Dooren, *Jacobi algorithm for the best low multilinear rank approximation of symmetric tensors*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 651–672.

[22] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Rev., 51 (2009), pp. 455–500.

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

[24] T. M. Low and R. 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, Austin, TX, May 2004.

[25] M. Marqués, G. Quintana-Ortí, E. S. Quintana-Ortí, and R. van de Geijn, *Solving "large" dense matrix problems on multi-core processors and GPUs*, in Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'09), Rome, Italy, 2009, IEEE, Piscataway, NJ, 2009, pp. 123–132.

[26] A.-H. Phan, P. Tichavsky, and A. Cichocki, *Fast alternating algorithms for high order CANDECOMP/PARAFAC tensor factorizations*, IEEE Trans. Signal Process., 61 (2013), pp. 4834–4846.

[27] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, *Elemental: A new framework for distributed memory dense matrix computations*, ACM Trans. Math. Software, 39 (2013), 13.

[28] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, *SPIRAL: Code generation for DSP transforms*, Proc. IEEE, 93 (2005), pp. 232–275.

[29] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan, *Programming matrix algorithms-by-blocks for thread-level parallelism*, ACM Trans. Math. Software, 36 (2009), 14.

[30] S. Ragnarsson and C. F. Van Loan, *Block tensors and symmetric embeddings*, Linear Algebra Appl., 438 (2013), pp. 853–874.

[31] S. Ragnarsson and C. F. Van Loan, *Block tensor unfoldings*, SIAM J. Matrix Anal. Appl., 33 (2012), pp. 149–169.

[32] P. A. Regalia, *Monotonically convergent algorithms for symmetric tensor approximation*, Linear Algebra Appl., 438 (2013), pp. 875–890.

[33] E. Solomonik, J. Hammond, and J. Demmel, *A Preliminary Analysis of Cyclops Tensor Framework*, Technical report UCB/EECS-2012-29, EECS Department, University of California, Berkeley, CA, 2012.

[34] G. Tomasi and R. Bro, *Multilinear models: Iterative methods*, in Comprehensive Chemometrics, S. Brown, R. Tauler, and R. Walczak, eds., Vol. 2, Elsevier, Oxford, 2009, pp. 411–451.

[35] R. A. van de Geijn, *Using PLAPACK: Parallel Linear Algebra Package*, The MIT Press, Cambridge, MA, 1997.

[36] F. G. Van Zee, E. Chan, R. van de Geijn, E. S. Quintana-Ortí, and G. Quintana-Ortí, *The libflame library for dense matrix computations*, Comput. Sci. Eng., 11 (2009), pp. 56–62.

[37] F. G. Van Zee and R. A. van de Geijn, *BLIS: A Framework for Generating BLAS-like Libraries*, FLAME Working Note #66, Technical report TR-12-30, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2012.

[38] R. C. Whaley and J. J. Dongarra, *Automatically tuned linear algebra software*, in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, ACM, New York, 1998, pp. 1–27.

[39] Z. Xianyi, W. Qian, and Z. Yunquan, *Model-driven level 3 BLAS performance optimization on Loongson 3A processor*, in Proceedings of the IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, Piscataway, NJ, 2012, pp. 684–691.

[40] S. Yamamoto and U. Nagashima, *Four-index integral transformation exploiting symmetry*, Comput. Phys. Comm., 166 (2005), pp. 58–65.

[41] F. G. Van Zee, libflame: *The Complete Reference*, Field G. Van Zee, 2009.