Efficiently Computing Tensor Eigenvalues on a GPU

Grey Ballard
UC Berkeley
Computer Science Department
Berkeley, CA
ballard@cs.berkeley.edu

Tamara Kolda and Todd Plantenga
Sandia National Laboratories
Livermore, CA
{*tgkolda,tplante*}*@sandia.gov*

***Abstract*—The tensor eigenproblem has many important applications, generating both mathematical and application-specific interest in the properties of tensor eigenpairs and methods for computing them. A tensor is an m-way array, generalizing the concept of a matrix (a 2-way array). Kolda and Mayo [1] have recently introduced a generalization of the matrix power method for computing real-valued tensor eigenpairs of symmetric tensors. In this work, we present an efficient implementation of their algorithm, exploiting symmetry in order to save storage, data movement, and computation. For an application involving repeatedly solving the tensor eigenproblem for many small tensors, we describe how a GPU can be used to accelerate the computations. On an NVIDIA Tesla C 2050 (Fermi) GPU, we achieve 318 Gflops/s (31% of theoretical peak performance in single precision) on our test data set.**

***Keywords*-tensors; tensor eigenvalues; GPU computing**

I. Introduction

The tensor eigenproblem has many important applications, including blind source separation [2], molecular conformation [3], and magnetic resonance imaging [4]–[6], and both mathematical and application-specific communities have taken recent interest in the properties of tensor eigenpairs as well as methods for computing them. A tensor is an m-way array, generalizing the notion of a matrix (a 2-way array). In this work, we focus on an efficient implementation of the shifted symmetric higher-order power method (SS-HOPM) for computing real-valued eigenpairs of symmetric tensors. This method was introduced by Kolda and Mayo [1] and is a generalization of the matrix power method.

The main motivating application for this work involves detection of nerve fibers in the brain from diffusion-weighted magnetic resonance imaging data. In this application, data is gathered for millions of cubic millimeter-sized voxels. Determining the number and directions of nerve fiber bundles within each voxel requires solving a small tensor eigenvalue problem. Because each voxel can be resolved independently, the computations are amenable to parallelism, and we focus our implementation on a graphics processing unit (GPU) using the Compute Unified Device Architecture (CUDA) programming framework.

We review the definition of the tensor eigenproblem as well as the SS-HOPM algorithm from [1] in Section II. All of the tensors discussed here are symmetric, and exploiting symmetry is the foremost sequential optimization we use to gain performance. Symmetric matrices can be stored in half the space and symmetric matrix computations often require only half the flops of their nonsymmetric counterparts; exploiting symmetry in tensors saves storage and computation by much larger factors. In Section III we discuss a symmetric tensor storage format and how this compressed format is used in the main computational kernels of SS-HOPM.

Instead of attempting to write an algorithm that offers high parallel performance for computing eigenpairs of tensors of general order and dimension, we focus the GPU implementation on small tensors, as in our motivating application. Because of the inherent parallelism in the problem, we can run many independent threads concurrently on the hardware, and we facilitate efficiency of each thread with careful memory management. We describe the motivating application in Section IV and give the details and results of our implementation in Section V.

The main contributions of this work are (1) the introduction of a symmetric storage format that reduces computation, memory use, and data movement by factors of $m!$ for m-way tensors, and (2) a parallel implementation of SS-HOPM that achieves up to 318 Gflops/s in single precision on an NVIDIA Tesla C 2050 (Fermi) GPU (31% of theoretical peak). While the implementation is tailored to a specific application, we believe the approach will be widely applicable to high performance computations with symmetric tensors.

II. Symmetric Tensors and Tensor Eigenpairs

Following the notation and terminology from [7], a tensor of order m is a multiway array whose entries can be indexed by m numbers. Each of the m numbers indexes a mode, and the range of each index is the dimension of the mode. For example, a matrix is an order-2 tensor whose modes are rows and columns. Let $\mathbb{R}^{[m,n]}$ be the set of real-valued order-m tensors where each mode has dimension n; for example, $\mathcal{A} \in \mathbb{R}^{[4,3]}$ means that $\mathcal{A}$ is a $3\times3\times3\times3$ real-valued tensor. We formally introduce the notion of a symmetric tensor which is invariant under any permutation of its indices.

Definition 1 (Symmetric [8]): A tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$ is

symmetric if

$$a_{i_{\pi(1)}\cdots i_{\pi(m)}} = a_{i_1\cdots i_m}$$

for all $i_1, \ldots, i_m \in \{1, \ldots, n\}$ and $\pi \in \Pi_m$ where Π_m is the set of permutations of the set $\{1, \ldots, m\}$.

For example, if $\mathcal{A} \in \mathbb{R}^{[3,2]}$, then entries of $\mathcal{A}$ are indexed by three numbers, each of which are in the set $\{1, 2\}$. If $\mathcal{A}$ is symmetric, then $a_{112} = a_{121} = a_{211}$ and $a_{122} = a_{212} = a_{221}$.

The main computational kernels in SS-HOPM are instances of the following definition of symmetric tensor-vector multiply.

Definition 2 (Symmetric tensor-vector multiply [1]): Let $\mathcal{A} \in \mathbb{R}^{[m,n]}$ be symmetric and $\mathbf{x} \in \mathbb{R}^n$. Then for $0 \leq p \leq m-1$, the $(m-p)$-times product of the tensor $\mathcal{A}$ with the vector $\mathbf{x}$ is denoted by $\mathcal{A}\mathbf{x}^{m-p} \in \mathbb{R}^{[p,n]}$ with entries defined by

$$(\mathcal{A}\mathbf{x}^{m-p})_{i_1\cdots i_p} = \sum_{i_{p+1},\ldots,i_m} a_{i_1\cdots i_m} x_{i_{p+1}} \cdots x_{i_m}. \qquad (1)$$

Note that there is ambiguity in defining a tensor times the same vector in some subset of modes, but due to symmetry the choice of indexing yields the same result as any other valid definition. The result of a symmetric tensor-vector multiply is also a symmetric tensor.[1]

We recall the definition of a tensor eigenpair used in [1], which is relevant to our target application. There are other definitions of eigenvalues and eigenvectors in the literature, but the relationships between the definitions and the many interesting properties of tensor eigenvalues are beyond the scope of this work.

Definition 3 (Symmetric tensor eigenpair [1]): Assume that $\mathcal{A}$ is symmetric. Then $\lambda \in \mathbb{C}$ is an *eigenvalue* of $\mathcal{A}$ if there exists $\mathbf{x} \in \mathbb{C}^n$ such that

$$\mathcal{A}\mathbf{x}^{m-1} = \lambda\mathbf{x} \quad \text{and} \quad \|\mathbf{x}\|_2 = 1. \qquad (2)$$

The vector $\mathbf{x}$ is the corresponding *eigenvector*, and $(\lambda, \mathbf{x})$ is called an *eigenpair*.

Unlike in the matrix case, not all eigenpairs of a symmetric tensor are real-valued (i.e., both the eigenvalue and the eigenvector are simultaneously real-valued), though there always exist at least two real-valued eigenpairs for a real-valued symmetric tensor. The eigenvectors of a symmetric tensor are also not orthogonal in general. If the symmetric tensor is of order m and dimension n, then there are $\frac{(m-1)^n-1}{m-2}$ distinct complex eigenpairs [9].

In Figure 1, we present the shifted symmetric higher-order power method (SS-HOPM) [1] for finding real-valued eigenpairs. This algorithm is a generalization of the matrix power method where the operation $\mathcal{A}\mathbf{x}^{m-1}$ generalizes the matrix-vector product and $\mathcal{A}\mathbf{x}^m$ generalizes the Rayleigh quotient for a unit vector. SS-HOPM includes the shift parameter α which is chosen to force the underlying function to be convex ($\alpha \geq 0$) or concave ($\alpha < 0$) in order to ensure convergence. While the matrix power method is guaranteed to converge to the principal eigenpair for (almost) every starting vector, SS-HOPM may converge to different real-valued eigenpairs for different starting vectors.

[1]To see why, permute the indices of the resulting tensor $(i_1 \cdots i_p)$ on the left hand side of Equation 1. This corresponds on the right hand side to a permutation of the first p indices of the tensor entries $a_{i_1\cdots i_m}$ in the summation. Due to symmetry, the values of tensor entries remain invariant.

1: **repeat**
2: **if** $\alpha \geq 0$ **then**
3: $\hat{\mathbf{x}}_{k+1} \leftarrow \mathcal{A}\mathbf{x}_k^{m-1} + \alpha\mathbf{x}_k$
4: **else**
5: $\hat{\mathbf{x}}_{k+1} \leftarrow -(\mathcal{A}\mathbf{x}_k^{m-1} + \alpha\mathbf{x}_k)$
6: **end if**
7: $\mathbf{x}_{k+1} \leftarrow \hat{\mathbf{x}}_{k+1}/\|\hat{\mathbf{x}}_{k+1}\|$
8: $\lambda_{k+1} \leftarrow \mathcal{A}\mathbf{x}_{k+1}^m$
9: **until** λ converges

Figure 1. Pseudocode for SS-HOPM [1]. Here $\mathcal{A} \in \mathbb{R}^{[m,n]}$ is symmetric, $\alpha \in \mathbb{R}$, and $\mathbf{x}_0 \in \mathbb{R}^n$ with $\|\mathbf{x}_0\| = 1$.

The symmetric higher-order power method (with no shift) was introduced in [10], and convergence of the method is proved for certain types of tensors in [2]. While the symmetric higher-order power method does not converge in general, choosing a sufficiently large (in absolute value) shift α guarantees convergence of SS-HOPM. The convergence properties of a given eigenpair are characterized in [1], but there are still many open problems regarding choice of starting vector, choice of shift, and finding eigenpairs with certain properties.

III. Exploiting Symmetry

A. Symmetric Tensor Storage

Let $\mathcal{A} \in \mathbb{R}^{[m,n]}$ be a symmetric tensor. In general, $\mathcal{A}$ has n^m entries, but since it is symmetric, many of the entry values are repeated and need not be stored redundantly. We define an *index* as a number $i \in \{1, \ldots, n\}$, we define a *tensor index* as an array of m indices corresponding to one entry of the tensor, and we define an *index class* as a set of tensor indices such that the corresponding tensor entries all share a value due to symmetry. For example, for $m = 3$ and $n = 2$, the possible indices are 1 and 2, and the tensor indices $[1, 1, 2]$ and $[1, 2, 1]$ are in the same index class since $a_{112} = a_{121}$.

We can find a unique representative of an index class by choosing the tensor index whose indices are in nondecreasing order. We define this nondecreasing tensor index as the *index representation* of the index class.

The index classes of $\mathcal{A}$ can also be characterized by the number of occurrences of each index $i \in \{1, \ldots, n\}$ in the tensor indices of the index class. Thus, we can define the *monomial representation* of an index class as an array of n integers where the i^{th} entry in the array corresponds to the number of occurences of the index i in the index class. Following the example given above, the index class that

includes $[1,1,2]$ and $[1,2,1]$ has monomial representation $[2,1]$ since there are two 1's and one 2 in every tensor index in the class.

In order to avoid redundant storage, we store only the unique values of the tensor (i.e., one value per index class). The following property gives the number of unique values of a dense symmetric tensor.

Property 1: The number of unique values of a symmetric tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$ is given by the binomial coefficient

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

Proof: Each index class corresponds to a unique value. Counting the number of possible monomial representations of length m with n possible values is equivalent to counting the number of ways to distribute m indistinguishable balls into n distinguishable bins, where the balls correspond to the indices of the tensor index and the bins correspond to the possible index values. Solving this standard combinatorial problem yields the result. ■

Assuming $\mathcal{A}$ is dense, we can impose an ordering on the unique entries and avoid storing any index information. We choose to use a lexicographic order of the index classes, increasing with respect to the index representation and decreasing with respect to the monomial representation. That is, the index class with index representation $[i_1, i_2, \ldots, i_m]$ is listed before $[j_1, j_2, \ldots, j_m]$ if $i_1 < j_1$ or if $i_1 = j_1$ and $i_2 < j_2$, and so on. Equivalently, the index class with monomial representation $[k_1, k_2, \ldots, k_n]$ is listed before $[l_1, l_2, \ldots, l_n]$ if $k_1 > l_1$ or if $k_1 = l_1$ and $k_2 > l_2$, and so on. This corresponds to an ordering on monomials in a given polynomial ring (the origin of the terminology). In this case, the index classes correspond to monomials which all have total degree m. See Table I for an example of lexicographic ordering for both representations in the case $m = 3$ and $n = 4$.

While the lexicographic ordering makes storing index information for every unique value unnecessary, it is important to compute index information during computations. Since the index representation requires m integers and the monomial representation requires n integers and we expect $n \gg m$ for most problems, we store the index representation and compute monomial representation values implicitly.

B. Computational Kernels

The two most computationally intensive kernels in SS-HOPM are computing the scalar $\mathcal{A}\mathbf{x}^m$ and the vector $\mathcal{A}\mathbf{x}^{m-1}$, where $\mathcal{A} \in \mathbb{R}^{[m,n]}$ is symmetric and $\mathbf{x} \in \mathbb{R}^n$. Both of these are instances of the symmetric tensor-vector multiply given in Definition 2, with $p = 0$ and $p = 1$, respectively.

Table I
SET OF INDEX CLASSES $\mathcal{I}^{[3,4]}$ IN LEXICOGRAPHIC ORDER.

	index			monomial			
1	1	1	1	3	0	0	0
2	1	1	2	2	1	0	0
3	1	1	3	2	0	1	0
4	1	1	4	2	0	0	1
5	1	2	2	1	2	0	0
6	1	2	3	1	1	1	0
7	1	2	4	1	1	0	1
8	1	3	3	1	0	2	0
9	1	3	4	1	0	1	1
10	1	4	4	1	0	0	2
11	2	2	2	0	3	0	0
12	2	2	3	0	2	1	0
13	2	2	4	0	2	0	1
14	2	3	3	0	1	2	0
15	2	3	4	0	1	1	1
16	2	4	4	0	1	0	2
17	3	3	3	0	0	3	0
18	3	3	4	0	0	2	1
19	3	4	4	0	0	1	2
20	4	4	4	0	0	0	3

1) Tensor times same vector in all modes: Consider the case $p = 0$:

$$\mathcal{A}\mathbf{x}^m = \sum_{i_1=1}^{n} \cdots \sum_{i_m=1}^{n} a_{i_1 \cdots i_m} x_{i_1} \cdots x_{i_m}. \tag{3}$$

For a nonsymmetric tensor, this summation requires at least one multiplication for each term (corresponding to each entry of $\mathcal{A}$), yielding at least n^m flops. However, we can exploit symmetry to reduce the computational complexity. Note that the tensor index matches the indices of the $\mathbf{x}$ vector entries for each term in the summation. Since the product of a set of numbers is also invariant under permutation, all of the terms in the summation corresponding to the same index class will have the same value.

For example, for $m = 3$ and $n = 2$, the term in the summation corresponding to the tensor index $[1,1,2]$ is given by $a_{112} \cdot x_1 \cdot x_1 \cdot x_2 = a_{112} x_1^2 x_2$, and the term in the summation corresponding to the tensor index $[1,2,1]$ is given by $a_{121} \cdot x_1 \cdot x_2 \cdot x_1 = a_{121} x_1^2 x_2$. Any tensor index with monomial representation $[2,1]$ yields this value.

We can avoid recomputing the redundant value by instead computing the number of times each unique term appears in the summation, which is given by the following property.

Property 2: The number of tensor indices of a symmetric tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$ in the index class with monomial representation $[k_1, k_2, \ldots, k_n]$ is given by the multinomial coefficient

$$\binom{m}{k_1, k_2, \ldots, k_n} = \frac{m!}{k_1!\, k_2! \cdots k_n!}.$$

Proof: Consider the monomial representation $[k_1, k_2, \ldots, k_n]$. Counting the number of tensor indices in this class is equivalent to counting the number of ways one can distribute m distinct balls into n distinct bins such

```
function y = SymmTensorVectorMult0(A, x)
    y ← 0
    I ← [1, ..., 1]                                   ▷ length m
    for j = 1 to (m+n−1 choose m) do
        x̂ ← x_{I_1} · x_{I_2} ··· x_{I_m}
        c ← Multinomial0(I)
        y ← y + c · A_j · x̂
        I ← UpdateIndex(I)                             ▷ See Figure 4
    end for
end function

function c = Multinomial0(I)
    div ← 1                                            ▷ divisor of (m choose k_1,...,k_n)
    curr ← −1                                          ▷ current index value
    mult ← −1                                          ▷ multiplicity of curr
    for j = 1 to m do
        if I_j ≠ curr then
            mult ← 1
            curr ← I_j
        else
            mult ← mult + 1
            div ← div · mult
        end if
    end for
    c = m! / div                                       ▷ set c = (m choose k_1,...,k_n)
end function
```

Figure 2. Pseudocode for computing $y = \mathcal{A}\mathbf{x}^m$ via Equation 4. We assume the unique entries of the symmetric tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$ are stored in lexicographic order in the array A, $\mathbf{x} \in \mathbb{R}^n$, and $y \in \mathbb{R}$. The helper function Multinomial0 computes the multinomial coefficient (the number of occurrences in the summation) of an index class via the index representation I.

that the i^{th} bin has k_i balls. Here the balls correspond to the (ordered) indices of the tensor index and the bins correspond to the possible index values. Solving this standard combinatorial problem yields the result. ■

We can thus rewrite Equation 3 as

$$\mathcal{A}\mathbf{x}^m = \sum_{I \in \mathfrak{I}^{[m,n]}} \binom{m}{k_1, k_2, \ldots, k_n} a_{i_1 \cdots i_m} \, x_1^{k_1} \cdots x_n^{k_n}, \tag{4}$$

where $\mathfrak{I}^{[m,n]}$ is the set of index classes for a symmetric tensor in $\mathbb{R}^{[m,n]}$, and $[k_1, \ldots, k_n]$ and $[i_1, \ldots, i_m]$ are the monomial and index representations of the index class I, respectively. Equation 4 yields the pseudocode in Figure 2, which assumes the unique values of $\mathcal{A}$ are stored in lexicographic order. For each unique value, the algorithm computes the index array and multinomial coefficient associated with the tensor entry and adds the contribution of that term to the accumulating result.

2) Tensor times same vector in all modes but one: Now consider computing the vector $\mathcal{A}\mathbf{x}^{m-1}$, the case $p = 1$ in Definition 2:

$$\left(\mathcal{A}\mathbf{x}^{m-1}\right)_{i_1} = \sum_{i_2=1}^{n} \cdots \sum_{i_m=1}^{n} a_{i_1 \cdots i_m} x_{i_2} \cdots x_{i_m} \tag{5}$$

Note that the j^{th} component of $\mathcal{A}\mathbf{x}^{m-1}$ does not depend on every tensor entry, only those tensor entries whose index representation starts with index j. Because of symmetry, Equation 5 can be rewritten with i_1 appearing as any index in the tensor index of the tensor value.

As in the case of computing $\mathcal{A}\mathbf{x}^m$, we can exploit symmetry to avoid performing the more than n^m multiplications required to compute all entries of the output vector if we followed Equation 5. As before, if a tensor value contributes to the summation for index k of the output vector, its symmetric counterparts contribute the same value to the sum. Following the example given before, where $m = 3$ and $n = 2$, both a_{112} and a_{121} contribute to the computation of $\left(\mathcal{A}\mathbf{x}^{m-1}\right)_1$, and each contributes the value $a_{112} \cdot x_1 \cdot x_2$. Note that a_{211} does not contribute to the summation for $\left(\mathcal{A}\mathbf{x}^{m-1}\right)_1$, because its first index is not 1.

Computing the number of tensor indices in an index class that contribute to a given entry of the output vector is a variation on Property 2. Consider an index class that contributes to the j^{th} entry of the output vector (i.e., an index class whose index representation includes an index j). Let $[k_1, k_2, \ldots, k_n]$ be the monomial representation, so that $k_j > 0$. In the context of assigning m balls to n bins with appriopriate multiplicities, we can assign the first ball to the j^{th} bin (enforcing that the tensor index starts with j). Then we have $m - 1$ more balls to assign to the n bins, but only $k_j - 1$ more are assigned to the j^{th} bin. Thus, the number of tensor indices that contribute the same value to the j^{th} element is given by the multinomial coefficient

$$\sigma(j) = \binom{m-1}{k_1, \ldots, k_j - 1, \ldots, k_n}.$$

Now we can rewrite Equation 5 as

$$\left(\mathcal{A}\mathbf{x}^{m-1}\right)_j = \sum_{\substack{I \in \mathfrak{I}^{[m,n]} \\ k_j > 0}} \sigma(j) \, a_{i_1 \cdots i_m} x_1^{k_1} \cdots x_j^{k_j - 1} \cdots x_n^{k_n} \tag{6}$$

where $\mathfrak{I}^{[m,n]}$ is the set of index classes for a symmetric tensor in $\mathbb{R}^{[m,n]}$, and $[k_1, \ldots, k_n]$ and $[i_1, \ldots, i_m]$ are the monomial and index representations of the index class I, respectively. Equation 6 yields the pseudocode in Figure 3.

3) Index array calculations: We can compute the index representation of an index class quickly by exploiting the lexicographic ordering and computing each index representation from the previous one. That is, given any index representation, we want to compute the next larger index representation in the lexicographic order, under the conditions that the indices within the index representation are nondecreasing and range between 1 and n.

To find the next representation, we seek to increment the least significant possible index (i.e., the rightmost index not equal to n). In the example given in Table I, the successor of $[1, 1, 1]$ is $[1, 1, 2]$ (the last index is incremented). More generally, suppose the k^{th} index is the least significant index not equal to n, so that the index class is

```
1: function y = SymmTensorVectorMult1(A, x)
2:     y ← 0
3:     I ← [1, ..., 1]                                   ▷ length m
4:     for j = 1 to (m+n−1 choose m) do
5:         for unique i ∈ I do                           ▷ skip repeated indices
6:             x̂ ← x_{I_1} · x_{I_2} ··· x_{I_m} / x_i
7:             c ← Multinomial1(I, i)
8:             y_i ← y_i + c · A_j · x̂
9:         end for
10:        I ← UpdateIndex(I)                            ▷ See Figure 4
11:    end for
12: end function
```

Figure 3. Pseudocode for computing $\mathbf{y} = \mathcal{A}\mathbf{x}^{m-1}$ via Equation 6. We assume the unique entries of the symmetric tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$ are stored in lexicographic order in the array A, and $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. The helper function Multinomial1 (not shown here) is a variant of Multinomial0 and computes the multinomial coefficient (the number of occurrences in the summation) of the index class and index via the index representation I.

```
1: function UpdateIndex(I)
2:     j ← m
3:     while I_j = n do                                  ▷ find least sig. index ≠ n
4:         j ← j − 1
5:     end while
6:     I_j ← I_j + 1                                     ▷ increment least sig. index ≠ n
7:     for k = j + 1 to m do                             ▷ update less sig. indices
8:         I_k ← I_j
9:     end for
10: end function
```

Figure 4. Pseudocode for updating the index representation to its successor in lexicographic order of unique entries in a symmetric tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$.

$[i_1, \ldots, i_k, n, \ldots, n]$.[2] Thus, this is the largest representation with prefix $[i_1, \ldots, i_k, \ldots]$, so the successor must have prefix $[i_1, \ldots, i_k + 1, \ldots]$. The smallest such representation that satisfies the nondecreasing condition is

$$[i_1, \ldots, i_k + 1, i_k + 1, \ldots, i_k + 1].$$

For example, again from Table I, the successor of $[2, 4, 4]$ is $[3, 3, 3]$. See Figure 4 for the pseudocode. In this way, we can store index information in an array of m integers, and under the lexicographic ordering, and updating the index information for each term in the summation requires $O(m)$ operations.

4) Computing number of occurrences: The number of occurrences of each index class is given by a multinomial coefficient in terms of the monomial representation of the index class. Since we store the index representation and not the monomial representation, we compute the multinomial coefficient implicitly. We can do this by computing the denominator with one pass over the array storing the index representation. The numerator is constant over all index classes and can be precomputed (either $m!$ or $(m-1)!$ for the two computational kernels).

In the case of computing $\mathcal{A}\mathbf{x}^m$, the task is to compute for each index class the product $k_1! \cdots k_n!$, where $[k_1, \ldots, k_n]$ is the monomial representation which is not stored explicitly. Note that k_i is the number of occurrences of index i in the index representation which is stored in memory. Since the index representation is nondecreasing, repeated occurrences of an index are contiguous. Thus, as we pass over the index array, we can multiply the accumulated product by 1 for the first occurrence of an index, by 2 for the second occurrence, and so on. For example, given the index representation $[1, 2, 2, 5, 5, 5, 5]$, the accumulated product is $1 \cdot 1 \cdot 2 \cdot 1 \cdot 2 \cdot 3 \cdot 4 = 1! \cdot 2! \cdot 4!$. This approach yields the function Multinomial0 in Figure 2.

In the case of computing $\mathcal{A}\mathbf{x}^{m-1}$, we take the same approach to compute the denominator, but we ignore one occurrence of the index corresponding to the entry of the output vector being computed. Following the preceding example, in the case of computing the 5^{th} element of $\mathcal{A}\mathbf{x}^{m-1}$, the index representation $[1, 2, 2, 5, 5, 5, 5]$ would yield to the accumulated product $1 \cdot 1 \cdot 2 \cdot 1 \cdot 2 \cdot 3 = 1! \cdot 2! \cdot 3!$. This approach yields the function Multinomial1 in Figure 3 (pseudocode not shown).

5) Computational costs: All the computations in the main loop of Figure 2 are done in $O(m)$ operations (floating point and otherwise). Thus, the computational complexity of computing $\mathcal{A}\mathbf{x}^m$ is $O\left(m \cdot \frac{n^m}{m!}\right) = O\left(\frac{n^m}{(m-1)!}\right)$.

There are nested loops in Figure 3, and the inner loop requires m iterations in the worst case. All the computations in the inner loop are done in $O(m)$ operations (floating point and otherwise), so the computational complexity of computing $\mathcal{A}\mathbf{x}^{m-1}$ is $O\left(m^2 \cdot \frac{n^m}{m!}\right) = O\left(\frac{mn^m}{(m-1)!}\right)$.

These computational costs (as well as the storage requirements) are summarized in Table II, which also provides a comparison to the costs of general nonsymmetric tensors. In the general case, both $\mathcal{A}\mathbf{x}^m$ and $\mathcal{A}\mathbf{x}^{m-1}$ can be computed by a sequence of matrix-vector products with the proper matricization of $\mathcal{A}$ and reshaping of results. The cost is dominated by the first matrix-vector product in which the matrix has size $n^{m-1} \times n$.

In the symmetric case, we can also trade off computation and storage. The results of Table II assume that the multinomial coefficients and index array updates are computed at every iteration. Alternatively, we can pre-compute and store both multinomial coefficient information and index arrays for each unique entry. This reduces the computational complexities of both kernels to $\frac{n^m}{(m-1)!} + O(n^{m-2})$ (this count does not include the pre-computation) but increases the storage requirements by a factor of $(m+2)$. Note that the extra storage is all integer information, and for most values of n and m, this data could be compressed into only a few bits per integer. This information can also be shared by different tensors of the same order and dimension; we use this in our application of many small tensor eigenproblems which all have the same size (see Section V).

[2]Note that there may be no instances of index n in the index class, in which case $k = m$, the index class is $[i_1, \ldots, i_k]$, and the successor is $[i_1, \ldots, i_k + 1]$.

Table II
COMPARISON OF STORAGE AND COMPUTATIONAL COSTS BETWEEN GENERAL AND SYMMETRIC TENSORS IN $\mathbb{R}^{[m,n]}$.

	general	symmetric
storage	n^m	$\frac{n^m}{m!} + O\left(n^{m-1}\right)$
computation ($\mathcal{A}\mathbf{x}^m$)	$2n^m + O(n^{m-1})$	$O\left(\frac{n^m}{(m-1)!}\right)$
computation ($\mathcal{A}\mathbf{x}^{m-1}$)	$2n^m + O(n^{m-1})$	$O\left(\frac{mn^m}{(m-1)!}\right)$

IV. DETECTING NERVE FIBER DIRECTION IN THE BRAIN

We next discuss an application well-suited for computation on a GPU. It involves many independent problems that can be solved in parallel, and each problem involves an amount of data that is small enough to reside in the memories on the streaming multiprocessors.

Diffusion-weighted magnetic resonance imaging (DW-MRI) is a tool used to detect nerve fibers in the brain. It is a non-invasive procedure that uses magnetic resonance to measure how quickly water diffuses in a certain direction. Water diffuses more quickly along the longitudinal axis of nerve fiber bundles than in any transverse or axial direction. DW-MRI measurements are taken from many different orientations for a discrete set of voxels in the brain. For each voxel, a diffusion function $D : \Sigma \to \mathbb{R}$ which maps an orientation to its rate of diffusion (here Σ denotes the unit sphere in $\mathbb{R}^3$) is approximated using the measurement data. For a unit vector $\mathbf{g}$, $D(\mathbf{g})$ is known as the "apparent diffusion coefficient" (ADC) [6].

When a voxel includes only one fiber orientation, the longitudinal direction should (globally) maximize D (it will exhibit the largest ADC). When a voxel includes more than one fiber orientation (in the case of crossing fibers), each fiber orientation should correspond to a local maximum of D.

According to [4]–[6], a common way to approximate the diffusion function is as a finite sum of spherical harmonic functions (which form a basis for complex functions on the unit sphere). The 2nd order series (with 6 terms) corresponds to a quadratic form

$$D(\mathbf{g}) \approx \mathbf{g}^T \mathbf{M} \mathbf{g}$$

where $\mathbf{M}$ is a symmetric positive definite 3×3 matrix. In this case, at least six measurements are required to determine the unique entries in the matrix $\mathbf{M}$ (or the six coefficients of the first spherical harmonic functions). In the case of a voxel with one principal fiber orientation, this approach is usually sufficient for resolving the correct orientation. However, in the case of fiber crossings or other complications such as bending or fanning fiber bundles, the approximation is often unable to resolve the fiber directions.

In order to handle such cases, more accurate measurements and approximations are necessary. The approach is to use higher order spherical harmonic series approximations which can be represented not as quadratic forms, but more generally as homogeneous forms. The homogeneous forms correspond to higher order tensors:

$$D(\mathbf{g}) \approx \mathcal{A}\mathbf{g}^m$$

for some symmetric tensor $\mathcal{A} \in \mathbb{R}^{[m,3]}$. Note that m must be even since $D(\mathbf{g})$ is a positive physical quantity for all $\mathbf{g}$ (if m is odd then $\mathcal{A}(-\mathbf{g})^m = -\mathcal{A}\mathbf{g}^m$). More DW-MRI measurements are required to determine the greater degrees of freedom in tensors of order $m > 2$, and the higher order polynomial can better approximate the true diffusion function. Orders $m = 4$ and $m = 6$ are most commonly used ($m = 4$, $m = 6$, and $m = 8$ require at least 15, 28, and 45 measurements respectively). The correspondence between coefficients of spherical harmonic functions with the entries in the associated symmetric tensor are given in [6].

As described in [1], the critical points of the function $f(\mathbf{x}) = \mathcal{A}\mathbf{x}^m$ and their function values are exactly the eigenpairs of the tensor $\mathcal{A}$ (satisfying Equation 2). Thus, in order to determine the principal fiber orientations in a given voxel, we can compute the principal eigenvectors of the associated tensor.

V. IMPLEMENTATION DETAILS

The computational problem for the nerve fiber data is to take as input a three-dimensional array of symmetric tensors and output one or more eigenpairs for each tensor. The three-dimensional array corresponds to the set of voxels which discretize the volume of a brain. The entries of each tensor correspond to the coefficients of the homogeneous polynomial which approximates the diffusion function for a given voxel. The eigenpairs which define local maxima of the approximate diffusion function correspond to principal nerve fiber directions within the voxel.

In order to find multiple eigenpairs, SS-HOPM must be executed with different starting vectors. We use many randomly chosen starting vectors in order to get reasonable coverage of the unit sphere. We choose random vectors by independently selecting each vector entry uniformly from $[-1, 1]$ and then normalizing. Alternatively, one could use a deterministic approach and pick starting vectors evenly spaced about the sphere.

The computational problem consists of executing SS-HOPM with many different tensors and many different starting vectors each. Since the voxel size for DW-MRI is on the order of one cubic millimeter, the number of voxels in a data set for a human brain can be in the millions.

We discuss two different implementations in this section. The first implementation works for sets of tensors of general order and dimension (though each tensor in the set must be the same size). The second implementation includes an optimization of complete loop unrolling (see Section V-D)

that is tailored to tensors of order $m = 4$ and dimension $n = 3$ which occur in our application.

A. Synthetic Test Set

We experimented with a synthetic test set provided by the Scientific Computing and Imaging Institute at the University of Utah. It consists of 1024 tensors corresponding to a 2D array of voxels which includes some with one and some with two principal fiber directions. Each tensor is 4^{th} order and has dimension $n = 3$, so each has 81 total entries with 15 unique values. We chose to use 128 starting vectors for each tensor in the hope of reasonably covering the sphere in $\mathbb{R}^3$ and also because it is a multiple of 32, the physical warp size on the GPU. We used a shift of $\alpha = 0$ as it yielded correct results for the tensors in this synthetic set (though we have no proof of convergence for $\alpha = 0$ and this set of tensors). Note that $\alpha = 0$ implies that SS-HOPM is the same algorithm as the one given in [2], [10]. Although the performance of the implementation will not vary much with α, choosing an appropriate shift for real data will balance a tradeoff between guarantees of convergence and time-to-completion. To find local maxima, a nonnegative shift must be used.

B. Thread Organization

Because of the number of independent problems, we are able to map the computation to the GPU in a straightforward way with minimal synchronization. We organize the CUDA threads in the following way: assign a thread block to each tensor and assign each thread in a thread block to a different starting vector. Since the number of starting vectors is greater than the warp size, each thread block utilizes all the processors on its multiprocessor. Similarly, as long as the number of tensors is at least 50 or so, all of the multiprocessors are utilized with three or four thread blocks each (multiple thread blocks are necessary to fill the instruction pipelines). We note that for larger numbers of tensors, this approach generalizes to a system with multiple GPUs.

C. Data Structures

Because of the small size of the tensors and vectors in this problem, we can fit all the data for each thread block in the memory on the multiprocessor and minimize the accesses to device memory. Let T be the number of tensors, U be the number of unique entries in each tensor, and V be the number of starting vectors. Recall that for this problem, $m = 4$, $n = 3$, $T = 1024$, $U = 15$, and $V = 128$. For real data, we expect T to grow into the millions but the rest of the parameters will remain constant, though V could be varied experimentally. The tensor data is of size $T \cdot U$, the array of starting vectors is $n \times V$, the array of output eigenvectors is $n \times (T \cdot V)$, and the array of output eigenvalues is of size $T \cdot V$. Note that every thread block can use the same set of starting vectors, but each has its own set of output vectors.

In addition to the main data structures, we pre-compute and store the index and multinomial coefficient information required in Figures 2 and 3. In the general implementation, the index information is stored as an array of size $m \times U$ and can be shared by *all* threads since all tensors are of the same order and dimension. We store the multinomial coefficient $\binom{m}{k_1,\ldots,k_n}$ for each unique tensor value, where $[k_1, \ldots, k_n]$ is the monomial representation of the index class of the unique entry. In this way, finding the number of occurrences of an entry for $\mathcal{A}\mathbf{x}^m$ is just a look-up, and computing the related multinomial coefficients used in $\mathcal{A}\mathbf{x}^{m-1}$, which are of the form $\binom{m-1}{k_1,\ldots,k_i-1,\ldots,k_n}$ for some i, can be done by reading the stored value, multiplying by k_i and dividing by m.[3]

D. Loop Unrolling

For a given order and dimension, we can unroll the loops within the two main computational kernels. This enables us to exploit the register file for storing the input and output vectors by statically allocating register variables corresponding to input and output vector entries. Not only does this expose instruction-level parallelism to the compiler, it also removes the indirection in accessing input and output vector entries. Further, it obviates the need to store index information and multinomial coefficients explicitly; these can be computed during the code generation at compile-time. This is possible for small problems, but to scale to larger problems we need a blocked approach. Handling the different cases that arise when blocking a symmetric tensor is future work. In the case $m = 4$ and $n = 3$, the number of terms in the summation for $\mathcal{A}\mathbf{x}^m$ is 15, and each of the three summations for the entries of the output vector $\mathcal{A}\mathbf{x}^{m-1}$ have 10 terms.

Another possible optimization would be to use common subexpression elimination on the unrolled summations. This optimization would reduce the flop count but also introduce dependencies in the unrolled instructions.

E. Results

The GPU used for these results is an NVIDIA Tesla C 2050 (Fermi) which has a single precision peak performance of 1030 GFLOPS. The CPU used is a dual-socket quad-core Intel Nehalem. Each core has a theoretical single precision peak performance of 22.4 GFLOPS. The parallel CPU code uses OpenMP. All computations are done in single precision, and we use 128 starting vectors in all cases.

We report on eight different implementations. We benchmark a completely sequential implementation, using one core of the CPU; parallel CPU implementations, using four cores (one socket) and eight cores (both sockets); and our

[3]One might consider storing the "coefficient" $\binom{m-1}{k_1,\ldots,k_n}$ so that only one multiply is needed to update the stored value for each kernel, but note that this value is not an integer in general.

(a) Flop rates in GFLOPS

	General	Unrolled	Unrolled speedup
CPU - 1 core	0.24	2.05 (9% peak)	8.47
CPU - 4 cores	0.86	7.07 (8% peak)	8.23
CPU - 8 cores	1.73	9.67 (5% peak)	5.60
GPU	17.00	317.83 (31% peak)	18.70

(b) Run times in milliseconds

	General	Unrolled
CPU - 1 core	2451	289
CPU - 4 cores	691	84
CPU - 8 cores	344	61
GPU	35	1.9

(c) Relative performance, normalized to sequential implementations

	General	Unrolled
CPU - 1 core	1.00	1.00
CPU - 4 cores	3.55	3.45
CPU - 8 cores	7.14	4.72
GPU	70.23	155.07

Table III
PERFORMANCE RESULTS FOR SIX DIFFERENT IMPLEMENTATIONS ON ALL 1024 TENSORS

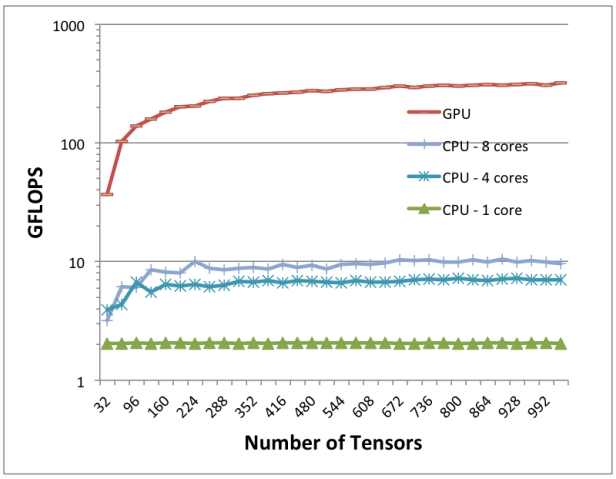

Figure 5. Performance results for running SS-HOPM on sets of 4^{th} order 3-dimensional tensors (with unrolled loops) with 128 starting vectors each. Note the y-axis is a log scale.

GPU implementation. In each case, we benchmark both the general version of the code and the loop-unrolled version which is specialized to tensors of order $m = 4$ and dimension $n = 3$. The CPU implementation executes the same algorithm as the GPU implementation, and the CPU parallelization is achieved with an `omp for` pragma on the loop over the input tensors. Note that no memory hierarchy optimizations are used in the CPU implementations, and we do not exploit the SIMD parallelism available on the Nehalem.

Table III shows the performance results for all eight implementations computing the eigenpairs for all 1024 tensors. Table III(a) shows the absolute performance and gives the speedup observed for each implementation by unrolling the loops. Comparing the unrolled code to the general implementations, we see that unrolling yields over $8\times$ speedup for the sequential implementation and a $19\times$ speedup for the GPU implementation. Table III(b) shows the run times, and in Table III(c), the relative performance values are normalized to the sequential CPU implementations to show parallel speedups. The CPU implementation (sequential or parallel) is not optimized for the memory hierarchy and we do not exploit SSE intrinsics. We chose the GPU as the platform for our primary implementation because of the large number of small, independent problems in this application. Future research will explore which architecture is better suited for computing eigenpairs of larger tensors or tensor applications with less inherent parallelism. In either case, developing high performing code for general orders and dimensions will require an efficient blocking strategy to allow for loop unrolling and the use of register variables.

Figure 5 shows performance results for the four different implementations for subsets of the 1024 tensors in our test set. Each implementation exploits loop unrolling. Because of the independence of the tensor eigenproblems, the parallel CPU implementation achieves nearly perfect parallel speedup over four threads with only a slight modification of the sequential code using OpenMP pragmas. We did not observe the same scaling using 8 threads (all cores available on the Nehalem) and we believe this is due to inefficient use of the memory hierarchy across both sockets.

Our GPU implementation does not perform as well for larger problems because it uses the register file to store both input and output vectors for each thread and shared memory to store a tensor for each thread block. As the tensor size grows, the per-thread and per-thread-block memory requirements also grow, resulting in decreased occupancy on the GPU (fewer threads and thread blocks scheduled simultaneously on each multiprocessor). We observe decreased performance for tensor sizes past a threshold of around order 4 and dimension 5. We also note that the code does not exploit features specific to the Fermi architecture. We obtained similar performance (relative to peak) for tensors of order 4 and dimension 3 on two other NVIDIA GPUs.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we present an implementation of SS-HOPM targeted for a GPU. We describe how to exploit symmetry to save both storage and computation in the two main computational kernels of the algorithm, and for the case of solving many small tensor eigenproblems we show how to map the computation onto a GPU. For our experimental data set, we achieved large parallel speedups over a sequential code using the same low-level optimizations (but no memory hierarchy optimizations).

We believe that the techniques for exploiting symmetry

may be extended to other computations involving symmetric tensors, but many open questions remain about how to write sequential or parallel implementations of the computational kernels that scale to higher order and higher dimension tensors. For the computations $\mathcal{A}\mathbf{x}^m$ and $\mathcal{A}\mathbf{x}^{m-1}$, we hope to be able to attain the same performance reported here for tensors of general size using register blocking and loop unrolling. The main implementation challenges will be to classify the various shapes of register blocks that arise (for each order m) so that each shape may be handled separately, and also to determine an implicit ordering on the unique tensor entries that attains the best cache behavior.

ACKNOWLEDGMENT

We would like to thank Fangxiang Jiao, Yaniv Gur, and Chris Johnson of the Scientific Computing and Imaging Institute at the University of Utah for the motivating application and for providing the sample data.

This work was funded by the applied mathematics program at the U.S. Department of Energy and by the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] T. G. Kolda and J. R. Mayo, "Shifted power method for computing tensor eigenpairs," Jul. 2010, arXiv:1007.1267v1.

[2] E. Kofidis and P. A. Regalia, "On the best rank-1 approximation of higher-order supersymmetric tensors," *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 3, pp. 863–884, 2002.

[3] R. Diamond, "A note on the rotational superposition problem," *Acta Crystallographica Section A*, vol. 44, no. 2, pp. 211–216, Mar 1988.

[4] E. Özarslan and T. H. Mareci, "Generalized diffusion tensor imaging and analytical relationships between diffusion tensor imaging and high angular resolution diffusion imaging," *Magnetic Resonance in Medicine*, vol. 50, pp. 955–965, 2003.

[5] ——, "Generalized scalar measures for diffusion MRI using trace, variance, and entropy," *Magnetic Resonance in Medicine*, vol. 53, pp. 866–876, 2005.

[6] T. Schultz and H.-P. Seidel, "Estimating crossing fibers: A tensor decomposition approach," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, pp. 1635–1642, 2008.

[7] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[8] P. Comon, G. Golub, L.-H. Lim, and B. Mourrain, "Symmetric tensors and symmetric tensor rank," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 3, pp. 1254–1279, 2008.

[9] D. Cartwright and B. Sturmfels, "The number of eigenvalues of a tensor," Apr. 2010, arXiv:1004.4953v1.

[10] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors," *SIAM Journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1324–1342, 2000.