EFFICIENTLY COMPUTING TENSOR EIGENVALUES ON A GPU

GREY BALLARD*, TAMARA KOLDA†, AND TODD PLANTENGA‡

Abstract. The tensor eigenproblem has many important applications, and both mathematical and application-specific communities have taken recent interest in the properties of tensor eigenpairs as well as methods for computing them. In particular, Kolda and Mayo [3] present a generalization of the matrix power method for symmetric tensors. We focus in this work on efficient implementation of their algorithm, known as the shifted symmetric higher-order power method, and on how a GPU can be used to accelerate the computation up to 70× over a sequential implementation for an application involving many small tensor eigenproblems.

1. Introduction. The tensor eigenproblem has many important applications, and both mathematical and application-specific communities have taken recent interest in the properties of tensor eigenpairs as well as methods for computing them. In particular, Kolda and Mayo [3] present a generalization of the matrix power method for symmetric tensors. We focus in this work on efficient implementation of their algorithm, known as the shifted symmetric higher-order power method (SS-HOPM).

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 focused 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 [3] in Section 2. 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 can save storage and computation by much larger factors. In Section 3 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 offer an overview of GPU computing in Section 4, describe the motivating application in Section 5, and give the details and results of our implementation in Section 6.

The main contributions of this work are (1) the introduction of a symmetric storage format and means of exploiting symmetry to avoid redundant computation, and (2) a parallel implementation of SS-HOPM. While the implementation is tailored to a specific application, we believe it will be widely applicable to high performance computations with symmetric tensors.

2. Symmetric Tensors and Tensor Eigenpairs. We formally introduce the notion of a symmetric tensor which is invariant under any permutation of its indices. Let $\mathbb{R}^{[m,n]}$ be the set

*UC Berkeley, ballard@cs.berkeley.edu
†Sandia National Laboratories, tgkolda@sandia.gov
‡Sandia National Laboratories, tplante@sandia.gov

of real-valued order-m tensors where each mode has dimension n.

Definition 2.1 (Symmetric tensor [1]). *A tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$ is symmetric if*

$$a_{i_{\pi(1)}\cdots i_{\pi(m)}} = a_{i_1\cdots i_m} \quad \text{for all} \quad i_1,\ldots,i_m \in \{1,\ldots,n\} \text{ and } \pi \in \Pi_m$$

where Π_m is the set of permutations of the set $\{1,\ldots,m\}$.

The main computational kernels in the shifted symmetric higher-order power method will be instances of the following definition of symmetric tensor-vector multiply.

Definition 2.2 (Symmetric tensor-vector multiply [3]). *Let $\mathcal{A} \in \mathbb{R}^{[m,n]}$ be symmetric and $\mathbf{x} \in \mathbb{R}^n$. Then for $0 \le p \le m-1$, the $(m-p)$-times product of the tensor $\mathcal{A}$ with the vector $\mathbf{x}$ is denoted by $\mathcal{A}\mathbf{x}^{m-p} \in \mathbb{R}^{[p,n]}$ and defined by*

$$(\mathcal{A}\mathbf{x}^{m-p})_{i_1\cdots i_p} = \sum_{i_{p+1},\ldots,i_m} a_{i_1\cdots i_m} x_{i_{p+1}} \cdots x_{i_m} \quad \text{for all} \quad 1 \le i_1,\ldots,i_p \le n. \tag{2.1}$$

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 below yields the same result as any other valid definition. Also note that the result of a symmetric tensor-vector multiply is also a symmetric tensor, because any permutation of the indices of the result tensor $(i_1 \cdots i_p)$ on the left hand side of Equation 2.1 corresponds to a permutation of the first p indices of the symmetric input tensor entries in the summation on the right hand side which remains invariant.

We recall the definition of a tensor eigenpair used in [3]. 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 2.3 (Symmetric tensor eigenpair [3]). *Assume that $\mathcal{A}$ is a symmetric m^{th}-order n-dimensional real-valued tensor. Then $\lambda \in \mathbb{C}$ is an* eigenvalue *of $\mathcal{A}$ if there exists $\mathbf{x} \in \mathbb{C}^n$ such that*

$$\mathcal{A}\mathbf{x}^{m-1} = \lambda\mathbf{x} \quad \text{and} \quad \|\mathbf{x}\|_2 = 1. \tag{2.2}$$

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

Finally, we present the shifted symmetric higher-order power method (SS-HOPM) from [3] as Algorithm 1. 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. Algorithm 1 includes the shift parameter α which is chosen to force the underlying function to be convex ($\alpha \ge 0$) or concave ($\alpha < 0$).

The symmetric higher-order power method (with no shift) was introduced in [2, 4], and convergence of the method was proved for certain types of tensors. 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 [3], but there are still many open problems regarding choice of starting vector, choice of shift, and finding eigenpairs with certain properties.

3. Exploiting Symmetry.

3.1. 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 $a_{112} = a_{121}$.

Algorithm 1 Shifted Symmetric Higher-Order Power Method (SS-HOPM) [3]

Given a tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$.

Require: $\alpha \in \mathbb{R}$, $\mathbf{x}_0 \in \mathbb{R}^n$ with $\|\mathbf{x}_0\| = 1$. Let $\lambda_0 = \mathcal{A}\mathbf{x}_0^m$.

for $k = 0, 1, \ldots$ **do**
  if $\alpha \geq 0$ **then**
    $\hat{\mathbf{x}}_{k+1} \leftarrow \mathcal{A}\mathbf{x}_k^{m-1} + \alpha \mathbf{x}_k$
  else
    $\hat{\mathbf{x}}_{k+1} \leftarrow -(\mathcal{A}\mathbf{x}_k^{m-1} + \alpha \mathbf{x}_k)$
  end if
  $\mathbf{x}_{k+1} \leftarrow \hat{\mathbf{x}}_{k+1} / \|\hat{\mathbf{x}}_{k+1}\|$
  $\lambda_{k+1} \leftarrow \mathcal{A}\mathbf{x}_{k+1}^m$
end for

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 3.1. *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 buckets, where the balls correspond to the indices of the tensor index and the buckets correspond to the possible index values. By a "stars and bars" argument,[1] this number is

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

as claimed. □

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

[1] See Theorem 2 in Section 4.6 of [9], for example.

	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

TABLE 3.1
Set of index classes $\mathfrak{I}^{[3,4]}$ in lexicographic order.

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 3.1 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 will be 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. Note that while the monomial representation will be sparse when $n \gg m$, even a compressed format would require at least m integers.

3.2. Computational Kernels. The two most computationally intensive kernels in Algorithm 1 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 2.2, with $p = 0$ and $p = 1$, respectively.

3.2.1. Tensor times same vector in all modes. Consider the case $p = 0$:

$$\mathcal{A}\mathbf{x}^m = \sum_{i_1=1}^{n} \cdots \sum_{i_m=1}^{n} a_{i_1 \cdots i_m} x_{i_1} \cdots x_{i_m} \tag{3.1}$$

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 $[2, 1]$ will yield 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 3.2. *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 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. One way to solve this problem is to count the number of ways of filling the first bin (given by the binomial coefficient $\binom{m}{k_1}$), followed by the number of ways of filling the second bin (given by $\binom{m-k_1}{k_2}$), and so on. Using the product rule and after much cancellation, we have

$$\binom{m}{k_1} \cdot \binom{m-k_1}{k_2} \cdots \binom{m-(k_1+k_2+\cdots+k_{n-1})}{k_n} = \frac{m!}{k_1!\, k_2!\, \cdots\, k_n!}$$

as claimed. □

We can thus rewrite Equation 3.1 as

$$\mathcal{A}\mathbf{x}^m = \sum_{I \in \mathfrak{I}^{[m,n]}} \binom{m}{k_1, k_2, \ldots, k_n} a_{i_1 \cdots i_m}\, x_1^{k_1} \cdots x_n^{k_n}, \tag{3.2}$$

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 3.2 yields Algorithm 2, which assumes the unique values of $\mathcal{A}$ are stored in lexicographic order. For each unique value, the algorithm computes the monomial coefficient and index array associated with the tensor entry and adds the contribution of that term to the accumulating result.

3.2.2. 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 2.2:

$$\left(\mathcal{A}\mathbf{x}^{m-1}\right)_{i_1} = \sum_{i_2=1}^{n} \cdots \sum_{i_m=1}^{n} a_{i_1 \cdots i_m} x_{i_2} \cdots x_{i_m} \tag{3.3}$$

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 3.3 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 3.3. As before, if a tensor value contributes to the summation for index k of the output vector, its symmetric counterparts will contribute the same value to the sum. Following the example given before, where $m = 3$ and $n = 2$, both a_{112} and a_{121} will contribute to the computation of $\left(\mathcal{A}\mathbf{x}^{m-1}\right)_1$, and each will contribute the value $a_{112} \cdot x_1 \cdot x_2$. Note that a_{211} will not contribute to the summation for $\left(\mathcal{A}\mathbf{x}^{m-1}\right)_1$, because its first index is not 1.

Algorithm 2 Compute $y = \mathcal{A}\mathbf{x}^m$ via Equation 3.2, where $\mathcal{A} \in \mathbb{R}^{[m,n]}$ is symmetric, $\mathbf{x} \in \mathbb{R}^n$, and $y \in \mathbb{R}$

Require: A stores the unique entries of $\mathcal{A}$ in lexicographic order
function y = SymmetricTensorVectorMultiply0(A, $\mathbf{x}$)
  $y = 0$
  $I = [1, \ldots, 1]$ ▷ use index representation (length m)
  for $j = 1$ to $\binom{m+n-1}{m}$ **do** ▷ iterate over unique entries
    $\hat{x} = x_{I_1} \cdot x_{I_2} \cdots x_{I_m}$ ▷ compute monomial value
    C = NumOcc0(I) ▷ compute number of occurrences
    $y = y + C \cdot A_j \cdot \hat{x}$ ▷ accumulate sum
    I = UpdateIndex(I) ▷ See Algorithm 4
  end for
end function

Require: I has length m with entries in nondecreasing order
function C = NumOcc0(I)
  div = 1 ▷ divisor of $\binom{m}{k_1,\ldots,k_n}$
  curr = -1 ▷ current index value
  mult = -1 ▷ multiplicity of current index value
  for $j = 1$ to m **do**
    if $I_j \neq$ curr **then**
      mult = 1
      curr = I_j
    else ▷ repeated index
      mult = mult + 1
      div = div · mult ▷ only update divisor if mult > 1
    end if
  end for
  $C = m!$ / div ▷ set $C = \binom{m}{k_1,\ldots,k_n}$
end function

Computing the number of tensor indices in an index class that will contribute to a given entry of the output vector is a variation on Property 3.2. 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 will be assigned to the j^{th} bin. Using the approach given in the proof of Property 3.2, we see that the number of tensor indices that will contribute the same value to the j^{th} element is given by the multinomial coefficient

$$\binom{m-1}{k_1, \ldots, k_j - 1, \ldots, k_n}.$$

Now we can rewrite Equation 3.3 as

$$\left(\mathcal{A}\mathbf{x}^{m-1}\right)_j = \sum_{\substack{I \in \mathfrak{I}^{[m,n]} \\ k_j > 0}} \binom{m-1}{k_1, \ldots, k_j - 1, \ldots, k_n} a_{i_1 \cdots i_m} x_1^{k_1} \cdots x_j^{k_j - 1} \cdots x_n^{k_n} \tag{3.4}$$

where $\mathcal{I}^{[m,n]}$ 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 3.4 yields Algorithm 3.

Algorithm 3 Compute $\mathbf{y} = \mathcal{A}\mathbf{x}^{m-1}$ via Equation 3.4, where $\mathcal{A} \in \mathbb{R}^{[m,n]}$ is symmetric, and $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

Require: A stores the unique entries of symmetric tensor $\mathcal{A}$ in lexicographic order

function $\mathbf{y}$ = SymmetricTensorVectorMultiply1(A, $\mathbf{x}$)
  $\mathbf{y} = 0$
  $I = [1, \ldots, 1]$ ▷ use index representation (length m)
  for $j = 1$ to $\binom{m+n-1}{m}$ **do** ▷ iterate over unique tensor entries
    for unique $i \in I$ **do** ▷ skip repeated indices in I
      $\hat{x} = x_{I_1} \cdot x_{I_2} \cdots x_{I_m} \;/\; x_i$ ▷ compute monomial value (excluding x_i)
      C = NumOcc1(I, i) ▷ compute number of occurrences
      $y_i = y_i + C \cdot A_j \cdot \hat{x}$ ▷ accumulate sum
    end for
    I = UpdateIndex(I) ▷ See Algorithm 4
  end for
end function

Require: I has length m with entries in nondecreasing order

function C = NumOcc1(I, i)
  div = 1 ▷ divisor of $\binom{m-1}{k_1,\ldots,k_i-1,\ldots,k_n}$
  curr = -1 ▷ current index value
  mult = -1 ▷ multiplicity of current index value
  for $j = 1$ to m **do**
    if $j \neq$ first index of i in I **then** ▷ ignore one occurence of i
      if $I_j \neq$ curr **then**
        mult = 1
        curr = I_j
      else ▷ repeated index
        mult = mult + 1
        div = div · mult ▷ only update divisor if mult > 1
      end if
    end if
  end for
  $C = (m-1)!$ / div ▷ set $C = \binom{m-1}{k_1,\ldots,k_i-1,\ldots,k_n}$
end function

3.2.3. 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 3.1, 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 $[i_1, \ldots, i_k, n, \ldots, n]$[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 3.1, the successor of $[2, 4, 4]$ is $[3, 3, 3]$. See Algorithm 4 for the implementation. 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.

Algorithm 4 Update index representation of unique entry in symmetric tensor $\mathcal{A} \in \mathbb{R}^{[m,n]}$

Require: I has length m with entries in nondecreasing order

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

Ensure: I is the successor in lexicographic ordering (restricted to nondecreasing)

3.2.4. 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 will be 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 will be $1 \cdot 1 \cdot 2 \cdot 1 \cdot 2 \cdot 3 \cdot 4 = 1! \cdot 2! \cdot 4!$. This approach yields the function NumOcc0 in Algorithm 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 NumOcc1 in Algorithm 3.

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

In order to avoid redundant computation (at the expense of extra storage), we can precompute the multinomial coefficient $\binom{m}{k_1,k_2,\ldots,k_n}$ for each index class. This is the coefficient used in the computation of $\mathcal{A}\mathbf{x}^m$, and the coefficients needed in the computation of $\mathcal{A}\mathbf{x}^{m-1}$ can be obtained by dividing the stored value by m and multiplying by k_j for appropriate j.

3.2.5. Computational costs. All the computations in the main loop of Algorithm 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 Algorithm 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 $O\left(m^2 \cdot \frac{n^m}{m!}\right) = O\left(\frac{mn^m}{(m-1)!}\right)$.

4. GPU Computing Overview. Graphics processing units (GPUs) were originally developed and optimized to offload and accelerate graphics rendering computations from the more general purpose microprocessor or "central processing unit" (CPU) on a host computer. Graphics processing consists largely of data parallel computations, and GPU hardware is designed to exploit this data parallelism via single instruction/multiple data (SIMD) instructions. GPUs also exploit instruction level parallelism: instruction streams for several threads of execution are pipelined in order to hide the latency of memory operations for each thread (this requires that the threads be mutually independent).

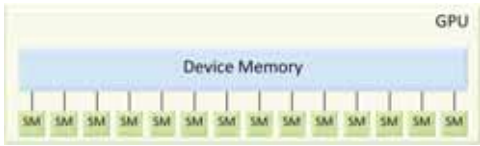
GPU architecture is rapidly developing to meet the demands of new applications and users. Many of these applications require high graphics rendering performance, but a growing number of users are interested in exploiting the computing power of GPUs for many other purposes including scientific computing. To this end, nVidia has invested in the development of Compute Unified Device Architecture (CUDA) which is used for general purpose programming of GPUs. Most programmers use CUDA as an extension of the C language which gives access to a set of virtual instructions for accessing the memory spaces and functional units on a GPU.

Along with making CUDA freely available, nVidia also offers a software development kit including programming guides, example programs, and other documentation for programmers. Much of the information in the following sections is available in more detail in the CUDA documentation, particularly [5, 6].

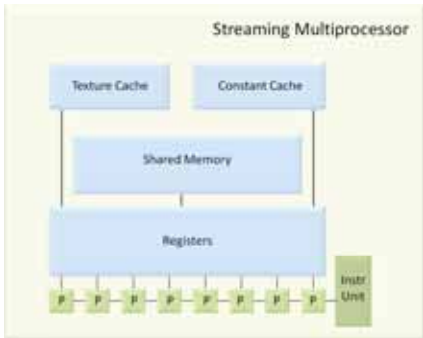
4.1. Physical Hardware Model. Both the computational units and the memory hierarchy on GPUs are fundamentally different from CPU architectures. See Figure 4.1 for a graphical representation of the physical hardware model.

Computational Units. The functional units on a GPU are organized into groups which concurrently execute SIMD instructions. In nVidia terminology, each functional unit is known as a "processor" or "core", and each group of processors resides on a "streaming multiprocessor." On the GeForce 9800 GT used in our experiments, there are 14 multiprocessors (see Figure 4.1(a)), each with 8 processors (see Figure 4.1(b)). Thus eight operations can simultaneously execute the same instruction on different data on a multiprocessor. The GPU we used is capable of only single precision floating point operations, but newer models can execute double precision operations.

Memory Hierarchy. GPUs have a complicated memory hierarchy with several different physical and logical memory spaces. Note that the memory hierarchy discussed here is only representative of nVidia GPUs of Compute Capability 1.x; newer architectures of Compute Capability 2.x have fundamental differences. The largest memory is known as "device memory" and is accessible to all multiprocessors on the GPU (see Figure 4.1(a)). It is also



(a) GPU card with device memory and set of streaming multiprocessors (SM). Memory on each SM shown in (b).



(b) Streaming multiprocessor with on-chip memories and SIMD functional units (P). Each SM has access to device memory as shown in (a).

FIG. 4.1. *GPU Hardware Model*

accessible from the host device (CPU) and is the means through which the CPU and GPU communicate data. Except for "integrated" cards, this memory resides on the graphics card itself. The memory access latency for device memory to one of the GPU's computational units is two orders of magnitude greater than the latency of the on-chip memory.

There are four types of on-chip memory: registers, shared memory, constant cache, and texture cache (see Figure 4.1(b)). The set of registers, or "register file," is relatively large but must be divided up among all threads resident on the multiprocessor; it has the smallest memory access latency (one or two cycles). The shared memory is the next fastest memory. It is smaller than the register file but can be shared among threads in a thread block. Shared memory can be dynamically allocated and can be used as a local store (i.e. there is no hardware-managed caching system).

Some of device memory can be statically allocated as "constant" memory, and accesses to constant memory will be cached by the hardware. Constant memory is read-only for a given GPU kernel function but can be written by the host CPU between kernel calls. A "texture" can be bound to an array in device memory such that the result of a texture "fetch" will be cached. The texture caches on a GPU are shared by two or three multiprocessors. The texture caching system is designed to exploit 2D spatial locality, and texture fetches include other features designed to improve the performance of certain relevant graphics operations. See Table 4.1 for the sizes of the on-chip memory for the GeForce 9800 GT card.

4.2. CUDA Programming Model. The simplest CUDA programming model treats the GPU as a coprocessor to the host CPU. That is, a single thread of execution works on the CPU sequentially until it calls a "kernel" function on the GPU which is run by many CUDA threads in parallel, and after the kernel returns, the single CPU thread resumes execution until it calls another kernel or terminates. Multiple CPU threads can be used in order to overlap CPU

Register file	8192 registers
Shared memory	16 KB
Texture cache	6-8 KB
Constant cache	8 KB

TABLE 4.1
On-chip memory sizes per multiprocessor for GeForce 9800 GT (Compute Capability 1.1)

and GPU computation, but we only consider one CPU thread in this work. Kernel functions may call other functions to be run on the GPU (which will also run in parallel); these other functions cannot be called from host code. When a kernel function is launched from the host code, the host specifies the number of thread blocks, the number of threads per block, and optionally the amount of shared memory to allocate to each thread block (all of which can be determined at run time).

Thread blocks are groups of threads which are all run on the same multiprocessor. They have a common memory space residing in the physical shared memory through which the threads can communicate and synchronize. Thread blocks are logical entities and the number of threads per block is unrestricted up to a certain maximum; however, threads are physically grouped into warps (the physical unit of SIMD instructions) during execution, so the number of threads per block should be a multiple of the warp size (typically 32).

The logical memory hierarchy is tightly coupled to the physical memory. Registers are local to threads, shared memory is restricted to threads within a thread block, and global memory (which resides in "device" memory) is accessible by all threads and by the host code. Communication between thread blocks using global memory is possible but rare because thread blocks may be scheduled on any multiprocessor in any order. Textures and constant memory are also globally accessible and are read-only; textures are accessed via special texture fetches. Another memory space known as "local" memory is logically local to each thread, but the name is misleading because local memory physically resides in device memory. In general, local memory is used to handle register spilling.

5. 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 on-chip memories of the 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) [10].

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 [7, 8, 10], 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 2^{nd} 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 ($m = 8$ requires 120 measurements). The correspondence between coefficients of spherical harmonic functions with the entries in the associated symmetric tensor are given in [10].

As described in [3], 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 2.2). Thus, in order to determine the principal fiber orientations in a given voxel, we can compute the principal eigenvectors of the associated tensor.

Note that specific instances of Properties 3.1 and 3.2 for $n = 3$ appear in the DW-MRI literature. See Equations 17 and 19 in [7], for example.

6. Implementation Details. The computation 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, Algorithm 1 must be executed with different starting vectors. Because there is not much theory to direct the choice of starting vectors to find all eigenpairs corresponding to local maxima, 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 Algorithm 1 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. In order to cover the sphere, we use somewhere between 32 and 128 starting vectors for each tensor. With this much inherent parallelism in the problem, we can easily saturate the computational units on a GPU. The main data structures involved in the computation include the unique entries of each tensor, an array of randomly generated starting vectors, an array of output eigenvectors, and an array of output eigenvalues.

6.1. 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, 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. Note that $\alpha = 0$ implies that SS-HOPM is the same algorithm as the one given in [2, 4]. 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.

6.2. 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 will utilize all the processors on its multiprocessor. Similarly, as long as the number of tensors is at least 50 or so, all of the multiprocessors will be utilized with three or four thread blocks each (multiple thread blocks are necessary to fill the instruction pipelines).

6.3. 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 on-chip memory 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 Algorithms 2 and 3. The index information is stored as an array of size $m \times U$ and can be shared by all threads. 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 in Algorithm 2 is just a look-up, and computing the related multinomial coefficients used in Algorithm 3, 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] Thus the array of multinomial coefficients is of size U. All threads can share this information.

6.4. Memory Management. We use both the shared memory and constant cache to minimize the memory accesses to device memory. Because the index array and multinomial coefficients are read only and can be shared by all the threads in the computation, we designate them as constant memory which resides in global (device) memory. However, because that information can fit into the constant cache of each multiprocessor, they will be read from device memory to the cache only once per multiprocessor for the entire computation. Because the tensor entries can be shared by the threads within one thread block, we store them in the shared memory. In this way, the tensor entries are read from device memory to the on-chip shared memory only once per thread block.

[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.

```
y1 =  Avals[0]      * x1 * x1 * x1 + \
      Avals[1]  * 3 * x1 * x1 * x2 + \
      Avals[2]  * 3 * x1 * x1 * x3 + \
      Avals[3]  * 3 * x1 * x2 * x2 + \
      Avals[4]  * 6 * x1 * x2 * x3 + \
      Avals[5]  * 3 * x1 * x3 * x3 + \
      Avals[6]      * x2 * x2 * x2 + \
      Avals[7]  * 3 * x2 * x2 * x3 + \
      Avals[8]  * 3 * x2 * x3 * x3 + \
      Avals[9]      * x3 * x3 * x3;
```

FIG. 6.1. *Unrolled computation of the first entry of the vector* $\mathcal{A}\mathbf{x}^{m-1}$*, for* $\mathcal{A} \in \mathbb{R}^{[4,3]}$*. The variables* `x1`*,* `x2`*,* `x3` *are register variables which store the input vector and the* `Avals` *array is in shared memory.*

Finally, we store the input and output vectors, which are private to each thread, in shared memory. Although this data will not be shared with other threads in the thread block, we use the shared memory because it is the only on-chip memory that can be dynamically allocated and overwritten. There are two main drawbacks from using shared memory this way. First, allocating $2n$ words of shared memory per thread requires a lot of memory per thread block, and since the physical shared memory is shared by all thread blocks on a multiprocessor, fewer thread blocks can be scheduled simultaneously on each multiprocessor. The amount of oversubscription (known as "occupancy" in nVidia's terminology) allows for pipelining instruction streams and hiding memory latency. Second, the register file is faster to access than shared memory. Since the number of thread blocks per multiprocessor is limited by the shared memory requirements, the size of the register file is not being exploited.

6.5. 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. This is possible for small problems, but to scale to larger problems we would need a blocked approach. See Figure 6.1 for an example of an unrolled loop in the case $m = 4$ and $n = 3$. The `Avals` array stores the unique tensor entries in lexicographic order, the input vector entries are stored in static variables, and the multinomial coefficients are stored as constants in the instruction stream. In this case, 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.

Without unrolling the loops, each access to an input or output vector requires two memory operations. For example, if the index information is stored in an array called `index`, then accessing entries to the input vector `x` take the form `x[index[k]]`. This indirection prevents the compiler from pipelining instructions within one thread and degrades performance even if `index` and `x` are both on-chip (see Section 6.6).

Note that the arithmetic intensity (ratio of flops to bytes involved in the computations) is high for both kernels. In the case $m = 4$ and $n = 3$, there are 15 unique tensor entries and two vectors each with three entries, so the number of bytes in single precision is 84 while the number of flops in computing $\mathcal{A}\mathbf{x}^{m-1}$ is 140. Another possible optimization would be to use common subexpression elimination on the unrolled summations. For example, the code shown in Figure 6.1 computes x_1^2 three times.

6.6. Results. The processor used for these results is a quad-core Intel Bloomfield (Core i7). The GPU used is an nVidia GeForce 9800 GT which nVidia classifies as Compute Capa-

(a) Flop rates in Gflops/s and speedup of loop unrolling

	General	Unrolled	Unrolled speedup
CPU seq	0.24	1.86	7.86
CPU par	0.92	6.85	7.41
GPU	5.95	131.73	22.15

(b) Relative performance, normalized to sequential implementations

	General	Unrolled
CPU seq	1.00	1.00
CPU par	3.90	3.67
GPU	25.08	70.66

TABLE 6.1
Performance results for six different implementations on all 1024 tensors

bility 1.1. The parallel CPU code was run with four threads using OpenMP. All computations were done in single precision (the only precision available on GPUs of Compute Capability 1.1), and we use 128 starting vectors in all cases.

We report on six different implementations. We benchmarked a completely sequential implementation, using one core on the quad-core CPU; a parallel CPU implementation, using all four cores of the processor; and our GPU implementation. In each case, we benchmarked both the general version of the code and the loop-unrolled version which is specialized to tensors of order 4 and dimension 3. Note that no memory hierarchy optimizations were used in the CPU implementations.

Table 6.1 shows the performance results for all six implementations computing the eigenpairs for all 1024 tensors. Table 6.1(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 7× speedup for both CPU implementations and a 22× speedup for the GPU implementation. In Table 6.1(b) the relative performance values are normalized to the sequential CPU implementations to show parallel speedups. We observe that the GPU implementation achieves a speedup of 20× over the parallelized CPU implementation. Although the CPU implementation was not optimized for the memory hierarchy, we believe that because of the large number of independent problems in this application, the GPU implementation will outperform the best multi-core implementation for this test set. 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 6.2 shows performance results for four different implementations for subsets of the 1024 tensors in our test set. Note that the loop unrolling makes a significant difference in the GPU performance for all problem sizes. Because of the independence of the tensor eigenproblems, the parallel CPU implementation requires only a slight modification of the sequential code using OpenMP pragmas and we observe close to perfect parallel scaling for sufficiently large problems.

7. Conclusions. In this paper we present an implementation of SS-HOPM targeted for a GPU. We describe how 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

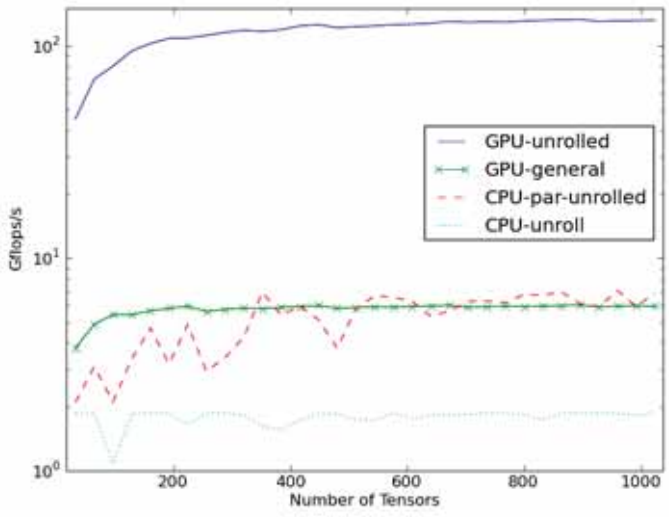


FIG. 6.2. *Performance results for running SS-HOPM on sets of* 4^{th} *order* 3*-dimensional tensors with* 128 *starting vectors each. Note the y-axis is a log scale.*

parallel speedups of up to 70× 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. We are also interested in how to map these computations onto different computing platforms, including more recent GPUs which offer fundamentally different hardware features.

Acknowledgments. 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.

REFERENCES

[1] P. COMON, G. GOLUB, L.-H. LIM, AND B. MOURRAIN, *Symmetric tensors and symmetric tensor rank*, SCCM Technical Report 06-02, Stanford University, 2006.

[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, 23 (2002), pp. 863–884.

[3] T. G. KOLDA AND J. R. MAYO, *Shifted power method for computing tensor eigenpairs*. arXiv:1007.1267v1 [math.NA], July 2010.

[4] L. D. LATHAUWER, B. D. MOOR, AND J. VANDEWALLE, *On the best rank-1 and rank-(r[sub 1],r[sub 2],. . .,r[sub n]) approximation of higher-order tensors*, SIAM Journal on Matrix Analysis and Applications, 21 (2000), pp. 1324–1342.

[5] NVIDIA, *NVIDIA CUDA programming guide version 3.0*.

[6] ———, *PTX: Parallel thread execution ISA version 2.0*.

[7] 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, 50 (2003), pp. 955–965.

[8] ———, *Generalized scalar measures for diffusion mri using trace, variance, and entropy*, Magnetic Resonance in Medicine, 53 (2005), pp. 866–876.

[9] K. H. Rosen, *Discrete mathematics and its applications (2nd ed.)*, McGraw-Hill, Inc., New York, NY, USA, 1991.

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