SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication

Shaden Smith*, Niranjay Ravindran†, Nicholas D. Sidiropoulos†, George Karypis*

University of Minnesota, Minneapolis, MN 55455, U.S.A.

*{shaden, karypis}@cs.umn.edu, †{ravi0022,nikos}@umn.edu

Abstract—Multi-dimensional arrays, or *tensors*, are increasingly found in fields such as signal processing and recommender systems. Real-world tensors can be enormous in size and often very sparse. There is a need for efficient, high-performance tools capable of processing the massive sparse tensors of today and the future. This paper introduces SPLATT, a C library with shared-memory parallelism for three-mode tensors. SPLATT contains algorithmic improvements over competing state of the art tools for sparse tensor factorization. SPLATT has a fast, parallel method of multiplying a matricized tensor by a Khatri-Rao product, which is a key kernel in tensor factorization methods. SPLATT uses a novel data structure that exploits the sparsity patterns of tensors. This data structure has a small memory footprint similar to competing methods and allows for the computational improvements featured in our work. We also present a method of finding cache-friendly reorderings and utilizing them with a novel form of cache tiling. To our knowledge, this is the first work to investigate reordering and cache tiling in this context. SPLATT averages almost $30\times$ speedup compared to our baseline when using 16 threads and reaches over $80\times$ speedup on NELL-2.

Keywords-Sparse tensors, PARAFAC, CANDECOMP, CPD, parallel

I. INTRODUCTION

Many application domains give rise to multi-way data that can be naturally represented via tensors. For example, in the context of a user content tagging system, e.g., Flickr[1], Delicious[2], and Mendeley[3] the set of tags that a user in the system provides to a piece of content is naturally represented via a three-mode tensor of *user-item-tag* triplets. Similarly, the three-way *subject-verb-object* relations that are being extracted by the Never Ending Language Learning (NELL) project [1] are represented via a three-mode tensor of *noun-verb-noun* triplets.

This increased applicability of tensors has led to the expanding use of tensor-based analysis techniques. The Canonical Polyadic Decomposition (CPD) is one of the most commonly used factorizations and has seen use in psychometrics [2], signal processing [3], recommender systems [4], and other fields. CPD, described in Section II-B, attempts to decompose a tensor into a set of rank-one tensors. Such a decomposition has numerous applications. For example, in the context of the tagging system, it can be used to recommend a set of tags to a user for a particular item or a

set of items given a user and their previous tags. Similarly in the context of NELL it can be used for noun-phrase concept discovery and to detect contextual synonyms [5].

Though the methods for computing CPD are well understood in the context of dense tensors, most recent applications of tensor decomposition involve tensors that are extremely large and very sparse. For example, NELL has dimensions in the tens of millions and over one-hundred million nonzero entries. Existing approaches for dense tensors cannot be applied to sparse datasets because their memory consumption scales with the tensor dimensions instead of the number of nonzeros entries. To address this need, various approaches that deal with sparse CPD have been proposed in recent years. The Tensor Toolbox [6] is a widely used MATLAB software package and uses an efficient algorithm that is not hindered by extreme sparsity. However, Tensor Toolbox's algorithm cannot easily be parallelized and as such it cannot leverage the multiple cores in today's multiprocessors. On the other hand, GigaTensor [5] uses an algorithm that is explicitly designed for large-scale parallelism but requires more floating-point operations (FLOPs) than other methods.

This paper introduces SPLATT, a C library for operating on three-mode tensors. Our contributions are three-fold:

1) SPLATT contains algorithmic improvements over the state of the art tools for factoring sparse tensors. SPLATT has a fast, parallel method of multiplying a matricized tensor by a Khatri-Rao product, a key kernel in tensor factorizations.
2) SPLATT uses a novel data structure that is able to exploit the sparsity patterns of tensors. This data structure has a small memory footprint and allows for the computational improvements featured in our work.
3) We present a method of finding cache-friendly reorderings and utilize them with a novel form of cache tiling. To our knowledge, this is the first work to investigate reordering and cache tiling in this context.

We evaluate SPLATT across several large datasets of varying properties and demonstrate speedup over other competing methods on each one. We evaluate our method for cache-friendly reordering and tiling by comparing against random orderings of our datasets. Finally, we also demonstrate near-linear scaling of our parallel algorithm.

II. PRELIMINARIES

In this section we provide a brief background on tensors and their notation. We then describe the Canonical Polyadic Decomposition, a widely used tensor factorization. For more information on tensors and their factorizations, we direct the reader to the essential survey by Kolda and Bader [7]. For a thorough discussion on implementation details of tensor computations in MATLAB, see [8].

A. Tensor Notation

Tensors are the generalization of matrices to higher dimensions. The dimensions occupied by the tensor are called *modes*. We can also say a tensor with n modes is of order n. For example, a tensor of order three takes the shape of a box and a tensor of order two would simply be a matrix. In this work we focus on third-order tensors because they are the simplest to reason about and visualize. They also have the added advantage of minimizing clutter due to added indexing. However, we stress that all methods presented in this work are easily extended to work with higher-order tensors. The simplest and perhaps most popular data structure for representing sparse tensors is a list of (i, j, k, v) coordinates.

In this work we denote tensors as $\mathcal{X}$ and matrices as $\mathbf{A}$. We write the element in coordinate (i, j, k) of $\mathcal{X}$ as $\mathcal{X}(i, j, k)$. Unless otherwise stated, the sparse tensor $\mathcal{X}$ is of dimension $I \times J \times K$ and has m nonzeros. We use the colon notation of MATLAB, in which a colon in the place of an index represents all members of that mode. For example, $\mathbf{A}(:, r)$ is column r of the matrix $\mathbf{A}$.

Fibers are a building block of tensors. Fibers are the result of holding all but one index constant. The fibers of a matrix are its rows and columns. In a third-order tensor, its added fibers are referred to as *tubes*. Two possible fibers are $\mathcal{X}(i, j, :)$ and $\mathcal{X}(i, :, k)$. A *slice* of a tensor is the result of holding all but two indices constant. The result is a matrix and two possible slices are $\mathcal{X}(i, :, :)$ and $\mathcal{X}(:, j, :)$.

Two essential operations on matrices used in the CPD are the *Hadamard product* and the *Khatri-Rao product*. The Hadamard product, denoted $\mathbf{A} * \mathbf{B}$, is the element-wise multiplication of $\mathbf{A}$ and $\mathbf{B}$. The element (i, j) of $\mathbf{A} * \mathbf{B}$ is $\mathbf{A}(i, j)\mathbf{B}(i, j)$. $\mathbf{A}$ and $\mathbf{B}$ must match in dimension for the Hadamard product to exist. The Khatri-Rao product, denoted $\mathbf{A} \odot \mathbf{B}$, is defined in terms of the Kronecker product

$$\mathbf{A} \odot \mathbf{B} = [a_1 \otimes b_1, a_2 \otimes b_2, \ldots, a_n \otimes b_n].$$

$\mathbf{A}$ and $\mathbf{B}$ must have matching column dimension for their Khatri-Rao product to be defined. If $\mathbf{A}$ is $I \times J$ and $\mathbf{B}$ is $M \times J$, then $\mathbf{A} \odot \mathbf{B}$ is $IM \times J$. Figure 1 illustrates the Khatri-Rao product of two small matrices.

A tensor can be matricized, or *unfolded*, into a matrix along any of its modes. In the mode-n matricization, the mode-n fibers are used to form the columns of the resulting matrix. The mode-n unfolding of $\mathcal{X}$ is denoted as $\mathbf{X}_{(n)}$. If $\mathcal{X}$

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix} \qquad \mathbf{C} \odot \mathbf{B} = \begin{bmatrix} c_{11}b_{11} & c_{12}b_{12} \\ c_{11}b_{21} & c_{12}b_{22} \\ c_{21}b_{11} & c_{22}b_{12} \\ c_{21}b_{21} & c_{22}b_{22} \\ c_{31}b_{11} & c_{32}b_{12} \\ c_{31}b_{21} & c_{32}b_{22} \end{bmatrix}$$

Figure 1: The Khatri-Rao product of two matrices.

is of dimension $I \times J \times K$, then $\mathbf{X}_{(1)}$ is of dimension $I \times JK$. Figure 2 demonstrates the unfolding of a small tensor.

$$\mathcal{X}(:, :, 1) = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \mathcal{X}(:, :, 2) = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

$$\mathbf{X}_{(1)} = \begin{bmatrix} 1 & 2 & 3 & 7 & 8 & 9 \\ 4 & 5 & 6 & 10 & 11 & 12 \end{bmatrix}$$

$$\mathbf{X}_{(2)} = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

$$\mathbf{X}_{(3)} = \begin{bmatrix} 1 & 4 & 2 & 5 & 3 & 6 \\ 7 & 10 & 8 & 11 & 9 & 12 \end{bmatrix}$$

Figure 2: The matricizations of an $(2 \times 3 \times 2)$ tensor.

B. Canonical Polyadic Decomposition

CPD is an extension of the Singular Value Decomposition (SVD) to tensors. In the SVD, a matrix $\mathbf{M}$ is decomposed into the summation F rank-one matrices, where F can either be the rank of $\mathbf{M}$ or some smaller integer if a low-rank approximation is desired. The SVD is most commonly written in terms of three matrices $\mathbf{M} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathsf{T}$, where $\mathbf{U}$ and $\mathbf{V}$ are unitary, $\boldsymbol{\Sigma}$ is a diagonal matrix of scaling factors, and the ith rank-one matrix is the outer product of u_i and v_i. Often, $\boldsymbol{\Sigma}$ is absorbed by scaling $\mathbf{A}$ and $\mathbf{M}$ is instead written as $\mathbf{M} = \mathbf{A}\mathbf{B}^\mathsf{T}$.

CPD extends this concept to factor a tensor into the summation of F rank-one tensors. A rank-one tensor of order n is the outer product of n vectors. Determining the exact rank of a tensor is NP-hard [9] and we are almost always interested in $F \ll \max(I, J, K)$ for sparse tensors. When computing the rank-F CPD of a third-order tensor, we wish to find factor matrices $\mathbf{A} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$, and $\mathbf{C} \in \mathbb{R}^{K \times F}$. $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are typically dense regardless of the sparsity of $\mathcal{X}$. The matricizations of $\mathcal{X}$ can also be defined in terms of its CPD,

$$\mathbf{X}_{(1)} \approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\mathsf{T}, \ \ \mathbf{X}_{(2)} \approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^\mathsf{T}, \ \ \mathbf{X}_{(3)} \approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^\mathsf{T}.$$

The method of *Alternating Least Squares* (ALS) is the most commonly used algorithm for computing the CPD. In

each iteration we first fix $\mathbf{B}$ and $\mathbf{C}$ and solve for $\hat{\mathbf{A}}$ via

$$\hat{\mathbf{A}} = \min_{\hat{\mathbf{A}}} ||\mathbf{X}_{(1)} - \hat{\mathbf{A}}(\mathbf{C} \odot \mathbf{B})^\mathsf{T}||_F^2.$$

The least squares problem is minimized by

$$\hat{\mathbf{A}} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^\mathsf{T}\mathbf{C} * \mathbf{B}^\mathsf{T}\mathbf{B})^\dagger,$$

where $\mathbf{M}^\dagger$ is the pseudo-inverse of $\mathbf{M}$. $(\mathbf{C}^\mathsf{T}\mathbf{C} * \mathbf{B}^\mathsf{T}\mathbf{B})$ is an $F \times F$ matrix, so computing its pseudo-inverse is a minor computation relative to $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$. Once $\hat{\mathbf{A}}$ is computed, $\hat{\mathbf{B}}$ and $\hat{\mathbf{C}}$ are then solved for similarly. The process is repeated until convergence.

C. Matricized Tensor Times Khatri-Rao Product

We denote $\mathbf{M} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ as MTTKRP (matricized tensor times Khatri-Rao product). MTTKRP is executed once per mode per iteration of ALS. For simplicity of notation and space, we only write MTTKRP in terms of operating on the first mode. We define $\mathbf{M}$ to have I rows and $\mathbf{B}$ and $\mathbf{C}$ to have J and K rows, respectively. $\mathbf{M}$, $\mathbf{B}$ and $\mathbf{C}$ all have F columns.

MTTKRP is often the bottleneck of computing the CPD. Even though $\mathbf{M}$ is only an $I \times F$ matrix, $\mathbf{C} \odot \mathbf{B}$ is a dense $JK \times F$ matrix which can occupy significantly more memory than $\mathbf{X}$. The size and the cost of forming $\mathbf{C} \odot \mathbf{B}$ is prohibitive for all but the smallest sparse tensors. An efficient MTTKRP implementation is essential for large-scale tensor operations and $\mathbf{C} \odot \mathbf{B}$ cannot by explicitly formed in practice.

III. RELATED WORK

Over the years a number of approaches have been developed for computing the MTTKRP. The most efficient of these methods operate in $O(m)$ time, but differ in implementation difficulty, cache utilization, and opportunities for parallelism.

A. Sparse Tensor-Vector Products

Each column of $\mathbf{M}$ is a linear combination of the fibers of $\mathbf{X}$ with columns of $\mathbf{B}$ and $\mathbf{C}$. MTTKRP can be formulated as a series of F tensor-vector products [8]. Using tensor-vector products is the chosen method for several major MATLAB implementations such as Tensor Toolbox [6] and Tensorlab [10].

A three-mode tensor requires two tensor-vector products per column of $\mathbf{M}$. A temporary array t of size m is used to "stretch" the vectors $\mathbf{B}(:, f)$ and $\mathbf{C}(:, f)$ to map to nonzeros of $\mathbf{X}$. For each column f, the two tensor-vector products are performed at once and stored within t. Once t is filled, we need to "shrink" it to a vector of length I. This essentially sums all of the entries of t that correspond to nonzeros $\mathbf{X}(i,:,:)$ and stores the result as $\mathbf{M}(i, f)$. Algorithm 1 presents pseudocode for computing tensor-vector products.

Using sparse tensor-vector products uses $3mF$ FLOPs ($2mF$ for the initial products and mF for the accumulation steps) and m intermediate memory words for t. Each

---

Algorithm 1 MTTKRP via Sparse Tensor-Vector products.

Input: nonzeros of $\mathbf{X}$ and respective I, J, and K indices
Output: M

  for $f \leftarrow 0$ to F **do**
    for $z \leftarrow 0$ to m **do** $\triangleright$ Vector products
      $t[z] \leftarrow vals[z]\mathbf{B}(indJ[z], f)\mathbf{C}(indK[z], f)$
    end for
    for $z \leftarrow 0$ to m **do** $\triangleright$ Accumulate $\mathbf{M}(:, f)$
      $\mathbf{M}(indI[z], f) \leftarrow \mathbf{M}(indI[z], f) + t[z]$
    end for
  end for

---

nonzero can be processed in parallel during the "stretch" stage because a nonzero will only modify a single element of t. The accumulation step does not have this guarantee, however, and must be executed serially. An advantage of this method is that $\mathbf{X}$ does not require any special data structure and it can be implemented in just a few lines of code in MATLAB.

For a more in-depth overview of various tensor products, we refer the reader to the work of Bader and Kolda [8], [7].

B. GigaTensor

GigaTensor [5] is a parallel CPD-ALS algorithm developed for the MapReduce [11] paradigm. GigaTensor utilizes the massive parallelism of MapReduce by reformulating MTTKRP as a series of Hadamard products. There are no dependencies during a Hadamard product and each element of the output can be computed in parallel.

GigaTensor avoids the construction of $\mathbf{C} \odot \mathbf{B}$ by separately computing the contributions of $\mathbf{B}(:, f)$ and $\mathbf{C}(:, f)$ with $\mathbf{X}_{(1)}$ via two Hadamard products. After computing the separate contributions, the results are combined via a third Hadamard product. The resulting matrix $\mathbf{N}$ has the same sparsity pattern as $\mathbf{X}_{(1)}$ and each nonzero entry $\mathbf{N}(i, y)$ is equal to

$$\mathbf{N}(i, y) = \mathbf{X}_{(1)}(i, y)\mathbf{B}(y\%J, f)\mathbf{C}(y/J, f). \quad (1)$$

After computing the entries of $\mathbf{N}$, the rows of the resulting matrix are summed to form a column of $\mathbf{M}$. The total process requires $5mF$ FLOPs and $m + max(J, K)$ intermediate memory.

C. DFacTo

DFacTo [12] is a recent algorithm designed for distributed tensor factorization. DFacTo uses an efficient MTTKRP algorithm that is posed as a series of sparse matrix-vector multiplications (SpMVs). $\mathbf{M}$ is computed one column at a time and each column is formed by two SpMVs. DFacTo first forms $\mathbf{X}_{(2)}^\mathsf{T}$, an $IK \times J$ sparse matrix whose rows consist of the mode-2 fibers of $\mathbf{X}$. When forming column $\mathbf{M}(:, f)$ we first compute $\mathbf{X}_{(2)}^\mathsf{T}\mathbf{B}(:, f)$ and store the result in the *vals* field of $\mathbf{M}^r$, an $I \times K$ sparse matrix. Finally, we compute

$\mathbf{M}^r\mathbf{C}(:, f)$ and store the result in $\mathbf{M}(:, f)$. The process is repeated for each of the F columns.

DFacTo requires $(2m+P+1)$ memory words to store $\mathcal{X}$, where P is the number of non-empty mode-2 fibers. An additional $(2P+I+1)$ words are required to store $\mathbf{M}^r$ for a total memory footprint of $(2m+I+3P+2)$ words. DFacTo executes MTTKRP in $2F(m+P)$ FLOPs. DFacTo consists entirely of SpMV operations and therefore can take advantage of a wealth of existing research that can be applied to an efficient parallel implementation.

IV. SPLATT

We developed an alternative algorithm for MTTKRP which uses a novel data structure for representing tensors. Our algorithm computes entire rows of $\mathbf{M}$ at a time and as a result only requires a single traversal of the sparse tensor structure. Our work is realized in the form of SPLATT, a C library for three-mode tensors with shared-memory parallelism. In this section we first show a derivation of our algorithm and an analysis of its data structure and computational performance. We discuss its generalization to an arbitrary number of modes and lastly discuss its parallelization.

A. Derivation

Let us briefly assume that $\mathcal{X}$ is dense, and so each row of $\mathbf{X}_{(1)}$ has exactly JK nonzeros. If we start from (1),

$$\mathbf{M}(i, f) = \sum_{z=0}^{JK} \mathbf{X}_{(1)}(i, z)\mathbf{B}(z\%J, f)\mathbf{C}(z/J, f)$$

$$\mathbf{M}(i, :) = \sum_{z=0}^{JK} \mathbf{X}_{(1)}(i, z)(\mathbf{B}(z\%J, :) * \mathbf{C}(z/J, :)) \quad (2)$$

$$= \sum_{k=0}^{K}\sum_{j=0}^{J} \mathcal{X}(i, j, k)(\mathbf{B}(j, :) * \mathbf{C}(k, :)) \quad (3)$$

$$\mathbf{M}(i, :) = \sum_{k=0}^{K} \mathbf{C}(k, :) * \sum_{j=0}^{J} \mathcal{X}(i, j, k)\mathbf{B}(j, :) \quad (4)$$

First, we rewrite (1) to operate on a row of $\mathbf{M}$. Next, we break the columns of $\mathbf{X}_{(1)}$ into J and K components to arrive at (3). Finally, we are able to factor out the inner multiplication of $\mathbf{C}(k, :)$ and reach the more efficient solution (4). The factored $\mathbf{C}$ term saves $F(J-1)$ multiplications per $\mathcal{X}(i, :, k)$ fiber. If $\mathcal{X}$ is sparse and fiber $\mathcal{X}(i, :, k)$ has $\hat{J}$ nonzeros, then $F(\hat{J}-1)$ FLOPs are saved, resulting in a total $2F(m+P)$ FLOPs.

Figure 3 illustrates SPLATT when operating with a single column. The algorithm that follows from (4) is straightforward. Algorithm 2 details the work for a single slice of $\mathcal{X}$.

Algorithm 2 SPLATT

Input: Slice $\mathcal{X}(i, :, :)$
Output: Row $\mathbf{M}(i, :)$

  $\mathbf{M}(i, :) \leftarrow 0$
  for all unique $k \in \mathcal{X}(i, :, :)$ **do**   ▷ Each $\mathcal{X}(i, :, k)$ fiber
    $accum(:) \leftarrow 0$ ▷ $F \times 1$ vector for accumulation
    for all $j \in \mathcal{X}(i, :, k)$ **do**
      $accum(:) \leftarrow accum(:) + \mathcal{X}(i, j, k)\mathbf{B}(j, :)$
    end for
    $\mathbf{M}(i, :) \leftarrow \mathbf{M}(i, f) + \mathbf{C}(k, f)accum(:)$
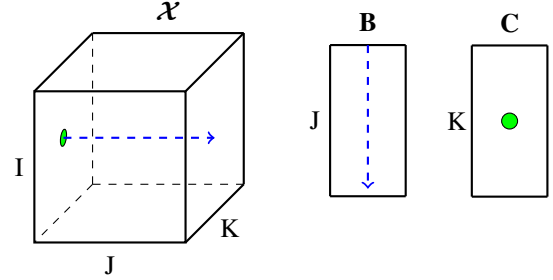 **end for**

Figure 3: SPLATT: The dashed blue line shows the fiber of $\mathcal{X}$ and its inner product with a column of $\mathbf{B}$. The inner product is then scaled by the circled value of $\mathbf{C}$.

B. Storage Scheme and Computational Complexity

SPLATT represents sparse tensors in a hierarchical, fiber-centric fashion. Each mode is stored as a list of slices. Each slice in turn contains a list of fibers represented as sparse vectors. Slices are stored in a structure that is very similar to a compressed sparse row (CSR) matrix. However, since slices are often extremely sparse, empty fibers should consume no storage overhead. Each fiber is also accompanied by a fiber id (*fid*) which specifies its K coordinate index.

Let $\mathcal{X}$ have P mode-2 fibers. SPLATT uses m floating-point numbers and m integers to store each nonzero value and its j coordinate. Analogous to rows of a CSR matrix, $(P+1)$ integers are required to store the *start* indices for the fibers and one integer is used for each *fid*. Finally, $(I+1)$ integers are used to mark the *start* indices of each slice. The total memory footprint of $\mathcal{X}$ is $(2m+I+P+2)$ words. The only additional memory is used to store the F inner products that are used to update $\mathbf{M}(i, :)$. SPLATT uses $2F(m+P)$ FLOPs which is identical to DFacTo.

We note that storage requirements for SPLATT and DFacTo are similar. Both are ultimately focused on a representation of the sparse fibers in $\mathcal{X}$. However, SPLATT only needs to store F additional floating-point numbers during computation and does not need the $2P$ memory words used to store $\mathbf{M}^r$ that DFacTo requires. Additionally, SPLATT

exhibits memory access patterns that have better spatial locality because the sparse structure of $\mathcal{X}$ is traversed only once and each nonzero value is used in F multiplications after being fetched from memory. This is a result of the row-oriented approach taken by SPLATT, which to our knowledge is the first of its kind in the sparse tensor community.

The decision to factor out $\mathbf{C}$ instead of $\mathbf{B}$ in (4) was arbitrary. Deciding which term to factor can greatly impact storage and computational performance. The decision is most relevant when the dimensions of the $\mathcal{X}$ are not equal. By storing fibers along the longer mode, we are able to minimize the number of stored fibers and increase the average fiber length. The benefit of this scheme is twofold: we reduce the amount of memory required to store the tensor and reduce the number of FLOPs due to a larger number of factored multiplications. Section VII-A demonstrates the benefits of selecting the best mode to factor.

C. Extensions to Higher-Order Tensors

While the body of this work is dedicated to three-mode tensors, our algorithm is easily extended to operate on tensors with four or more modes. If $\mathcal{X}$ is an n-mode tensor, then MTTKRP in the first mode becomes

$$\mathbf{M} = \mathbf{X}_{(1)}(\mathbf{A}^{(n)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(2)}).$$

The block structure in the Khatri-Rao product becomes more pronounced as n increases. SPLATT is able to exploit this block structure by factoring out a new set of multiplications per mode. Let $\mathcal{X}$ be an n-mode tensor with dimensions $I_1 \times I_2 \times \ldots \times I_n$. Our algorithm for MTTKRP in the first mode is formulated as

$$\mathbf{M}(i_1,:) = \sum_{i_n=0}^{I_n} \mathbf{A}^{(n)}(i_n,:) * \sum_{i_{n-1}=0}^{I_{n-1}} \mathbf{A}^{(n-1)}(i_{n-1},:)$$
$$\cdots \sum_{i_3=0}^{I_3} \mathbf{A}^{(3)}(i_3,:) * \left(\sum_{i_2=0}^{I_2} \mathcal{X}(i_1, i_2, \ldots, i_{n-1}, i_n) \mathbf{A}^{(2)}(i_2,:) \right).$$

The Khatri-Rao product operates on $n-1$ modes, requiring $F(n-2)$ words of intermediate memory. The last mode does not need intermediate memory because it writes to M directly. Like before, fibers of $\mathcal{X}$ are used for inner products with $\mathbf{A}^{(2)}$, which are then scaled by the corresponding row of $\mathbf{A}^{(3)}$ and so on.

When forming each of the n representations of $\mathcal{X}$, we must choose an ordering of the remaining $n-1$ modes. As discussed in Section IV-B, arranging the modes to minimize the number of fibers (and maximize the average fiber length) can have a significant impact on the storage and computation required. This is achieved by sorting modes by their dimension such that the shortest modes correspond to outer loops and the longest mode corresponds to the direction that $\mathcal{X}$ stores its fibers.

D. Parallelization

The parallel version of SPLATT uses a task decomposition on the rows of $\mathbf{M}$. Since the computation of $\mathbf{M}(i,:)$ requires only the nonzeros in slice $\mathcal{X}(i,:,:)$, the mode-1 slices of $\mathcal{X}$ can be distributed among processes. All process write to distinct rows of $\mathbf{M}$ and thus parallel execution requires no locks or synchronization. Each process requires only F words of additional storage to accumulate inner products. Since $F \ll m$ this method is memory scalable.

The unstructured sparsity pattern of $\mathcal{X}$ poses the issue of potential load imbalance. The nonzeros of $\mathcal{X}$ are rarely distributed in a uniform fashion. For example, the number of nonzeros across the slices in NELL can vary by several orders of magnitude. A static decomposition of rows can assign hugely disproportionate amounts of work to the processes, resulting in severe load imbalance and reduced scalability. Therefore, SPLATT uses dynamic load balancing when distributing tasks to processes.

V. OPTIMIZING FOR CACHE PERFORMANCE

In addition to the algorithmic improvements used by SPLATT, we present a method of achieving further speedup by efficiently utilizing the CPU memory hierarchy through means of reordering and cache blocking.

A. Tensor Reordering

Permuting the indices within one or more modes, or *reordering*, can lead to significant performance gains as it can potentially improve cache utilization by exploiting spatial and temporal locality. Figure 4 illustrates a tensor before and after reordering.

The goal of reordering a sparse tensor is to group nonzeros into semi-dense regions. Nonzeros form a sequence of semi-dense cuboids along the super-diagonal after an ideal reordering. Dense regions are attractive because they offer increased cache performance while accessing $\mathbf{B}$ and $\mathbf{C}$. Consider the execution of SPLATT along the first mode. The mode-2 indices in a fiber determine which rows of $\mathbf{B}$ are accessed and the constant mode-3 index determines the accessed row of $\mathbf{C}$. Consecutive mode-2 indices result in an unstrided access pattern that offers spatial locality in memory and can effectively utilize hardware prefetching mechanisms. If the accessed portion of $\mathbf{B}$ is sufficiently small and there are shared mode-2 indices in nearby fibers, the required portions of $\mathbf{B}$ will still reside in cache. Additionally, as other slices are processed we can also see the same reuse in $\mathbf{C}$ due to repeated mode-3 indices.

In this work we identify two methods of reordering sparse tensors. The first is based on the partitioning of a graph that models the interactions between slices of each mode of $\mathcal{X}$. This method is *mode-independent* because a single reordering is used for each mode of computation. The second method is based on the partitioning of a hypergraph that models the memory accesses to $\mathbf{M}$, $\mathbf{B}$, and $\mathbf{C}$. Unlike the

$$\begin{bmatrix} 0 & 3 & 0 & 3 & | & 0 & 0 & 0 & 0 & | & 2 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & | & 1 & 0 & 1 & 0 & | & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & | & 1 & 0 & 1 & 0 & | & 2 & 0 & 0 & 2 \\ 0 & 3 & 0 & 3 & | & 0 & 0 & 0 & 0 & | & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 3 & 3 & 0 & 0 & | & 0 & 0 & 0 & 0 & | & 0 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 & | & 0 & 2 & 2 & 0 & | & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & | & 0 & 2 & 2 & 0 & | & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & | & 0 & 0 & 0 & 0 & | & 0 & 0 & 1 & 1 \end{bmatrix}$$

(b)

Figure 4: A tensor (a) before reordering and (b) after reordering.

Figure 5: (a) $\mathcal{X}$, a $2 \times 2 \times 2$ tensor. (b) $\mathcal{X}$ mapped to a mode-1 hypergraph whose nodes are the $\mathcal{X}(i, :, k)$ fibers. Filled nodes are hyperedges. (c) $\mathcal{X}$ mapped to a tripartite graph with unlisted weights assumed to be unit.

tripartite graph model, the hypergraphs are specific to a given mode (*mode-dependent*) and thus multiple reorderings are needed.
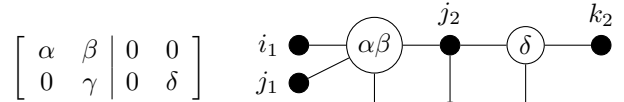
1) Mode-Independent Reorderings: The objective of a mode-independent reordering is to find a single tensor permutation that results in improved execution time regardless of which mode MTTKRP is being performed on. We achieve this goal by permuting modes of the tensor such that indices with high levels of similarity are adjacent.

We draw from the *bipartite graph model* for reordering sparse matrices [13]. Suppose $\mathcal{X}$ is an n-mode tensor with dimensions $I_1 \times I_2 \times \ldots \times I_n$. We construct an n-partite graph whose vertex sets are of cardinalities $I_1 \times I_2 \times \ldots \times I_n$. Nonzero $\mathcal{X}(i_1, i_2, \ldots, i_n)$ generates a clique that connects nodes $i_1, i_2, \ldots, i_n$. Using this scheme, edge (i_a, i_b) will be created every time a nonzero is processed with indices i_a and i_b appearing together. To account for this we weight edges based on the number of times they are generated. For example, when $\mathcal{X}$ has three modes the resulting graph is tripartite and edge (i, k) has weight equal to the number of nonzeros in the $\mathcal{X}(i, :, k)$ fiber. Figure 5 shows a small tensor and its corresponding graph.
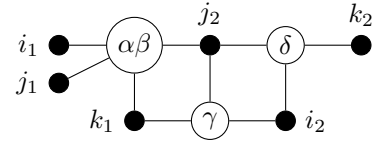
After generating a tensor's graph, a graph partitioner is used to create a partitioning. The graph is next relabeled such that vertices in the same partition are given consecutive labels. Finally, we generate a reordered tensor from the relabeled graph.

2) Mode-Dependent Reorderings: Mode-dependent reorderings offer further opportunities for optimization at the cost of additional work during the reordering stage. When operating within a certain mode we know precisely which memory accesses will result from the tensor's sparsity pattern.
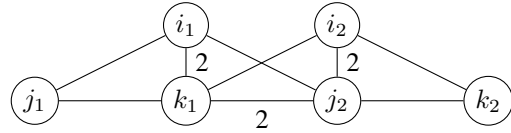
Our hypergraph model is an extension of the *column-net model* originally used for parallel sparse matrix-vector multiplication [14]. Fibers are our unit of work and are analogous to rows in a sparse matrix. Fibers are mapped to vertices in the hypergraph. Each mode emits as many hyperedges as its own dimension. A three-mode tensor will have $I+J+K$ hyperedges. Each hyperedge connects all fibers that its corresponding index can be found within. For example, if fiber $\mathcal{X}(i, :, k)$ has three nonzeros, then that vertex will be connected by five hyperedges. Two connections will come from the i and k indices and the final three will come from each nonzero mode-2 index found in the fiber.

Our goal is to model memory accesses as hyperedges. The number of partitions in which a hyperedge is found (or, its *connectivity*) exactly models the number of times that its corresponding row in **M**, **B**, or **C** must be fetched from memory. Thus, by minimizing the connectivity of all hyperedges (known as the *sum of external degrees*), we minimize the number of total memory accesses.

We partition the hypergraph to induce a reordering of the tensor. Fibers (vertices) are relabeled such that fibers in the same partition are given consecutive labels. Relabeling a fiber means to relabel all indices found in its nonzero entries. Indices are not unique to fibers and so we ensure that we only label an index the first time it is encountered. Consider fibers stored along the second mode. Mode-1 and mode-3 indices determine the order in which fibers are processed. This affects temporal locality because it allows fibers with similar sparsity pattern to be processed nearby in time. Relabeled mode-2 indices affect spatial locality and allow a fiber and its neighbors to access consecutive rows of **B**.

A clear drawback of a mode-dependent reordering is the need to construct and partition a hypergraph for each mode. Fortunately, much of this cost is mitigated due to the ordering of modes done by SPLATT. Recall that SPLATT stores fibers along the mode with the largest dimension. Consider a tensor of dimensions $I \times J \times K$ and $I < J < K$. SPLATT will store fibers along the third mode for the first

two modes of computation. During the third mode, fibers will be stored along the second mode because it has the next largest dimension. The only difference in execution between the first and second modes is the order in which fibers are processed. Thus, the hypergraphs of the first and second modes will be identical except for the labels of mode-1 and mode-2 hyperedges. A consequence is that a partitioning of one hypergraph will be equally suited for the other. Therefore, only partitionings of the mode-1 and mode-3 hypergraphs are needed for a complete reordering. This observation extends to tensors of higher modes as well. Irrespective of the number of modes, only two partitionings are needed: one generated by the longest mode and one generated by any other mode.

B. Cache Blocking over Sparse Tensors

The extremely large dimensions that sparse tensors often exhibit are prohibitive to memory performance, even with a good reordering. Assume that fibers run along the second mode and are defined by a unique (i, k) pair. Long fibers will fetch enough of the rows of $\mathbf{B}$ to evict cache lines that would otherwise be reused in other nearby fibers. In order to maximize data reuse, we used cache blocking.

Our method of blocking over a sparse tensor during MTTKRP is a generalization of the blocking used for matrix-vector multiplication. We seek to define three-dimensional tiles over the sparsity pattern of $\mathcal{X}$. If a tile has dimension $I' \times J' \times K'$ then accesses to $\mathbf{M}$, $\mathbf{B}$, and $\mathbf{C}$ are limited to a maximum of I', J', and K' rows, respectively. Thus, by carefully selecting tile dimensions such that the entire working set fits in CPU cache we can increase reuse of $\mathbf{M}$, $\mathbf{B}$, and $\mathbf{C}$.

Tiling over a sparse tensor is not a trivial task. An implementation that statically assigns nonzeros to tiles based on their coordinates and the tile dimensions will result in mostly empty or near-empty tiles due to the high levels of sparsity present. Additionally, most datasets feature unstructured sparsity patterns that can result in tiles of wildly varying density. We propose a method of *growing* tiles to adapt to the sparsity pattern of a given tensor.

First, we divide the mode-1 slices into sets of size I'. We call a set of slices a *layer*. Since empty slices can be trivially removed from the dataset, we assume that they are either not present or are rare enough such that we may statically assign slice i to layer i/I'. The sparsity pattern of each layer may differ dramatically and thus each layer is given an independent tiling.

We proceed one layer at a time. Within each layer we first construct the set of mode-3 indices present. We divide the indices into sets of size K' and arrange the $\mathcal{X}(i, :, k)$ fibers into *tubes*, each with a maximum of I' mode-1 indices and K' mode-3 indices. Each tube must be tiled independently due to their varying sparsity patterns.

Finally, within each tube we construct the set of mode-2 indices that are present. This set is used to divide the tube into tiles with $I' + J' + K'$ unique indices. If we choose dimensions so that $F(I' + J' + K')$ floating point numbers can comfortably fit in cache, and the ordering of $\mathcal{X}$ provides regions which are relatively dense, then we have effectively increased reuse in $\mathbf{M}$, $\mathbf{B}$, and $\mathbf{C}$.

All of the fibers within a slice are no longer adjacent in memory after tiling. Consequently, parallel execution within a layer is difficult because writes to the same row of $\mathbf{M}$ can occur at any time. We identify two methods of modifying SPLATT to execute over a tiled tensor. The first method is to distribute the tiled layers among threads and prevent race conditions while avoiding synchronization or atomics. The drawback of distributing entire layers is that the working set of each tile is now local to individual threads. The data reused between threads will be limited to similarities in sparsity pattern between layers. The second method of tiling is a cooperative scheme. Each thread uses its own $I' \times F$ matrix of scratch space to accumulate writes to $\mathbf{M}$. All threads execute concurrently within a tile but must synchronize at the end of each layer. After the synchronization, threads cooperate to do a summation of the scratch matrices. Since we operate in a shared address space we are able to evenly distribute the I' rows of scratch space among threads and do a reduction with only a synchronization at the end.

VI. EXPERIMENTAL METHODOLOGY

A. Experimental Setup

SPLATT was implemented in C with double precision floating-point numbers and 64-bit integers. SPLATT uses OpenMP for shared memory parallelism. All source code is available for download[4]. Load balance is achieved by OpenMP's dynamic scheduling with a chunksize of 16. Experiments were carried out on an HP ProLiant BL280c G6 blade server with dual 8-core E5-2670 Xeon processors running at 2.6 GHz. Source code was compiled with GCC 4.8.0 using optimization level O2. For all experiments we used $F = 10$.

B. Datasets

We evaluated our method across several datasets of varying properties. Table I is a summary of the mentioned datasets.

The Netflix dataset is taken from the Netflix Prize competition [15]. The dataset forms a *user-item-time* ratings tensor. Two datasets come from the Never Ending Language-Learning (NELL) project [1] which is freely available. Both tensors represent *noun-verb-noun* triplets. NELL-1 is the complete, extremely sparse dataset and NELL-2 is a smaller, more dense version in which the infrequent items have been pruned. BrainQ [16] is derived from fMRI measurements of

---

[4]*http://cs.umn.edu/~shaden/software/*

Table I: Summary of datasets.

Dataset	I	J	K	nnz	density
Netflix	480K	18K	2K	100M	5.4e-06
NELL-1	4M	4M	25M	144M	3.1e-13
NELL-2	15K	15K	30K	77M	1.3e-05
BrainQ	60	70K	9	11M	2.9e-01
Delicious	532K	17M	2.5M	140M	6.1e-12

nnz is the number of nonzero entries in the dataset. **density** is defined by $nnz/(I{\times}J{\times}K)$.

Table II: Difference in storage requirements and runtime for the mode-1 slices of each dataset.

Dataset	Storage (Improvement)		Time (Speedup)	
	Short	Long	Short	Long
Netflix	7.75	5.02 (1.54×)	8.77	6.02 (1.45×)
NELL-1	11.91	8.88 (1.34×)	25.74	19.83 (1.29×)
NELL-2	4.32	3.69 (1.17×)	3.18	2.78 (1.14×)
BrainQ	0.54	0.50 (1.08×)	0.28	0.31 (0.90×)
Delicious	9.28	8.23 (1.12×)	17.66	15.61 (1.13×)

Short and **Long** refer to SPLATT using fibers along the short or the long mode, respectively. Storage is measured in gigabytes. Runtime is the average time in seconds to perform an execution of SPLATT in all three modes. Storage and runtime for Short is measured relative to Long. No cache tiling is used. × denotes improvement over Short.

Table III: Effects of Tensor Reordering.

Dataset	Time (Speedup)		
	Random	Mode-Independent	Mode-Dependent
Netflix	6.02	5.26 (1.14×)	5.43 (1.10×)
NELL-1	19.83	17.83 (1.11×)	17.55 (1.12×)
NELL-2	2.78	2.61 (1.06×)	2.60 (1.06×)
Delicious	15.61	13.10 (1.19×)	12.51 (1.24×)

Runtime is the average time to perform a serial execution of SPLATT across all three modes. When reordering, the number of partitions was scaled from 32 to 1024 and the best result used. Time is measured in seconds. × denotes speedup over a random ordering.

Table IV: Effects of Cache Tiling.

Thds	Time (Speedup)			
	SPLATT	tiled	MI+tiled	MD+tiled
1	8.14 (1.0×)	8.90 (0.9×)	8.70 (1.0×)	9.18 (0.9×)
2	4.73 (1.7×)	4.88 (1.7×)	4.37 (1.9×)	4.52 (1.8×)
4	2.54 (3.2×)	2.58 (3.2×)	2.29 (3.6×)	2.35 (3.5×)
8	1.42 (5.7×)	1.41 (5.8×)	1.26 (6.5×)	1.26 (6.4×)
16	0.90 (9.0×)	0.85 (9.5×)	0.74 (11.0×)	0.75 (10.8×)

Time is measured in seconds and averaged across all datasets. **Thds** is the number of threads used. **MI** and **MD** are mode-independent and mode-dependent reorderings, respectively. When reordering, the number of partitions was scaled from 32 to 1024 and the best result used. × denotes speedup over a random ordering without tiling.

brain activity. Its three modes are *noun-voxel-human subject*. BrainQ is an interesting dataset because its dimensions are relatively small, resulting in a tensor several orders of magnitudes more dense than the other tensors studied in this work. Delicious is a *user-item-tag* dataset originally crawled by Görlitz et al. [17] and is also available for download.

VII. RESULTS

A. Effects of Fiber Direction Selection

SPLATT chooses at runtime which direction to store fibers in each of its modes. For example, the slices of the first mode can either have fibers that run along the second or third mode and the slices of the second mode will follow either the first or third mode. This is analogous to determining whether the sparse matrix representing each slice is stored in a row or column major format. Each fiber comes with some storage overhead and the number of saved FLOPs is dependent on the number of nonzeros per fiber. When there is a large disparity between the dimensions of $\mathcal{X}$, choosing to have fewer, longer fibers is beneficial.

We evaluated this optimization on each of our datasets and present results in Table II. SPLATT requires less memory when storing fibers along the longer dimensions for all tested datasets. Additionally, faster runtimes are exhibited on all datasets except BrainQ, in which shorter fibers had a $1.10\times$ speedup. Speedup peaked at $1.45\times$ on Netflix.

B. Effects of Tensor Reordering and Cache Tiling

To evaluate our methods of improving cache performance we measured runtime of SPLATT across orderings and tile sizes. The baseline is a randomly permuted tensor without tiling. Since reordering will only offer speedup on very sparse tensors, we omitted BrainQ from the reordering experiments. Times reported are the average time of executing SPLATT with one thread across all three modes. Results are shown in Table III. Delicious saw the largest benefit and reached $1.24\times$ speedup after a mode-dependent reordering.

We found that reordering alone is not sufficient for maximizing performance. On all datasets, the best parallel speedups were found using a combination of reordering and cache tiling. The best results that we achieved using 16 threads are shown in Table IV. Note that these configurations are the most *scalable* configurations and not necessarily the fastest at small numbers of threads. This is because tiling increases arithmetic operations and the memory footprint of the tensor due to fibers being split across boundaries. After tiling we found the runtimes of mode independent and mode dependent reorderings to be similar, with mode-independent reorderings slightly faster.

Datasets with modes of relatively small dimension (BrainQ, Netflix, and NELL-2) saw benefit from cooperative tiling with up to a $1.22\times$ speedup on BrainQ compared to traditional tiling. The number of synchronizations and reductions scale with the mode dimensions and thus large datasets such as NELL-1 and Delicious saw impaired scalability when using cooperative tiling. We experimentally found that tiles of dimension $2048{\times}2048{\times}4096$ gave the best performance when executing cooperatively and tiles of dimension $32{\times}1024{\times}1024$ gave the best performance when distributing entire layers to threads.

C. Parallel Scaling Results

We evaluated SPLATT against three competitor algorithms: sparse tensor-vector products ('TVec'), GigaTensor, and DFacTo. TVec and GigaTensor were implemented in C and optimized to the best of our ability. DFacTo is written in C++ and has been made freely available by the authors [18]. DFacTo is a distributed code and uses MPI for parallelism. During our scalability study we ran with all MPI ranks on a single machine and omitted communication costs from our timings. Each of the competitor methods are column-oriented approaches and thus we used a column-major layout for the factor matrices when evaluating competitors. SPLATT uses a row-major layout for the factor matrices. Tensors start from a random ordering and SPLATT does not have cache tiling enabled. Speedup reported is based off the average time to execute MTTKRP across all three modes, which simulates an iteration of ALS. We scale from one to sixteen processors and measure speedup relative to the serial runtime of TVec.

Figure 6 shows the mean speedup across all datasets. On average, SPLATT is $3.7\times$ faster than our baseline and scales to be $29.8\times$ faster with 16 threads. SPLATT exhibits the best performance on NELL-2, reaching $81\times$ speedup. Figure 7 illustrates scaling on NELL-2. Near-linear scaling is achieved on all but one dataset, BrainQ. BrainQ's low dimension, especially in the third mode, limits the parallelism that SPLATT can exploit (Figure 8). SPLATT extracts parallelism from the slices of each mode and thus any processes beyond the number of slices are necessarily idle during computation. Despite the limited scalability on BrainQ, SPLATT is still able to significantly outperform competitor methods across a wide range of nonzero densities.

VIII. Conclusions

Tensors are becoming increasingly important in today's data analysis and there is a real need for highly optimized sparse tensor tools. In this work we introduced SPLATT, a C library for parallel sparse tensor computations with a highly optimized method of computing MTTKRP. We presented a method of reordering sparse tensors and cache tiling to improve data locality and by using a novel data structure were able to improve cache reuse and achieve serial and parallel speedup across a variety of datasets. On average, SPLATT is over $3\times$ faster than our baseline using one thread and scales to average $29\times$ faster at 16 threads.

A challenging characteristic of many tensor computations is the modal nature of the problem. SPLATT, DFacTo, and GigaTensor all require a separate representation of the tensor for each mode. Ravindran et al. [19] recently introduced efficient algorithms for performing MTTKRP in all three modes that collectively use only a single representation of the tensor. Our future work includes adapting SPLATT to this model investigating memory-scalable algorithms for tensor factorization in the context of distributed systems.
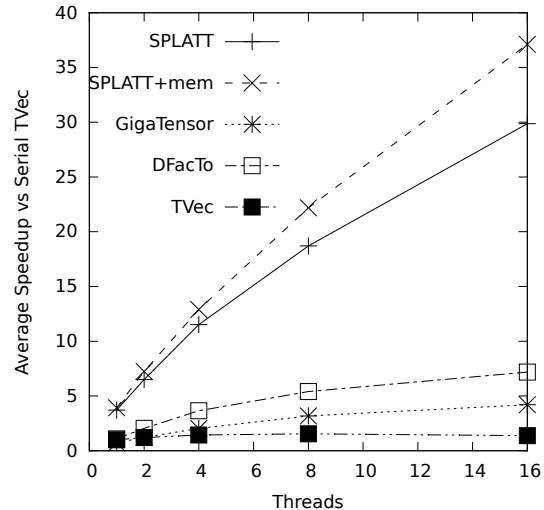

Figure 6: Average speedup over serial Tensor-Vector Products ('TVec'). SPLATT+mem indicates that reordering and tiling are used.
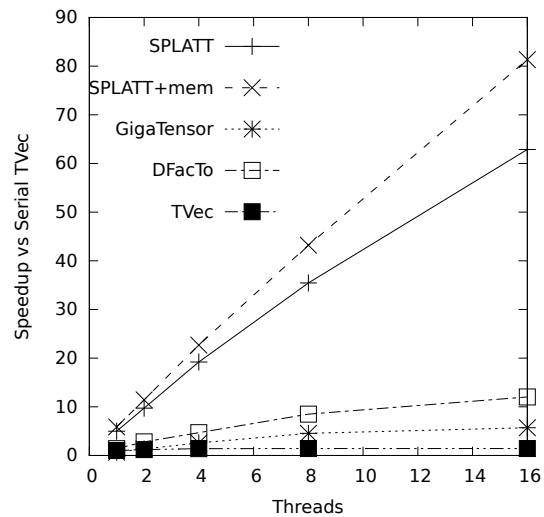

Figure 7: Speedup on NELL-2 over serial Tensor-Vector Products ('TVec'). SPLATT+mem indicates that reordering and tiling are used.
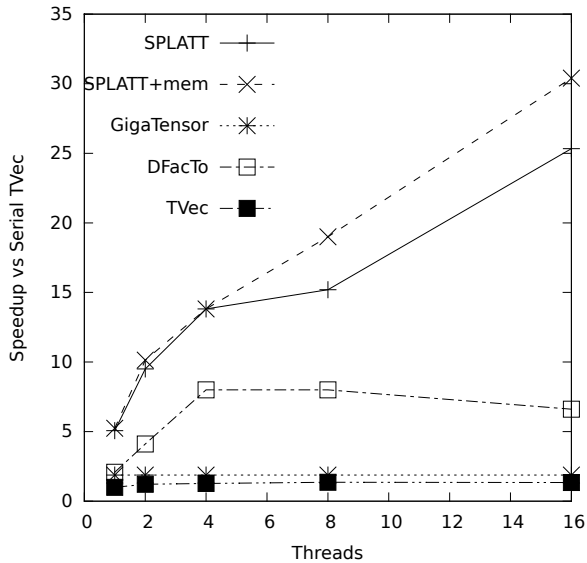
Figure 8: Speedup on BrainQ over serial Tensor-Vector Products ('TVec'). SPLATT+mem indicates that reordering and tiling are used.

REFERENCES

[1] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka, and T. M. Mitchell, "Toward an architecture for never-ending language learning," in *In AAAI*, 2010.

[2] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of eckart-young decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[3] D. Nion and N. D. Sidiropoulos, "Tensor algebra and multidimensional harmonic retrieval in signal processing for mimo radar," *Signal Processing, IEEE Transactions on*, vol. 58, no. 11, pp. 5693–5705, 2010.

[4] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, A. Hanjalic, and N. Oliver, "Tfmap: optimizing map for top-n context-aware recommendation," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2012, pp. 155–164.

[5] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 316–324.

[6] B. W. Bader, T. G. Kolda *et al.* (2012, Jan.) Matlab tensor toolbox version 2.5. Http://www.sandia.gov/~tgkolda/TensorToolbox/.

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

[8] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, December 2007.

[9] J. Håstad, "Tensor rank is np-complete," *Journal of Algorithms*, vol. 11, no. 4, pp. 644–654, 1990.

[10] L. Sorber, M. Van Barel, and L. De Lathauwer. (2014, Jan.) Tensorlab v2.0. [Online]. Available: http://www.tensorlab.net/

[11] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] J. H. Choi and S. Vishwanathan, "Dfacto: Distributed factorization of tensors," in *Advances in Neural Information Processing Systems*, 2014, pp. 1296–1304.

[13] B. Hendrickson and T. G. Kolda, "Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing," *SIAM Journal on Scientific Computing*, vol. 21, no. 6, pp. 2048–2072, 2000.

[14] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparsemmatrix vector multiplication," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 7, pp. 673–693, 1999.

[15] J. Bennett and S. Lanning, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, 2007, p. 35.

[16] T. M. Mitchell, S. V. Shinkareva, A. Carlson, K.-M. Chang, V. L. Malave, R. A. Mason, and M. A. Just, "Predicting human brain activity associated with the meanings of nouns," *science*, vol. 320, no. 5880, pp. 1191–1195, 2008.

[17] O. Görlitz, S. Sizov, and S. Staab, "Pints: peer-to-peer infrastructure for tagging systems." in *IPTPS*, 2008, p. 19.

[18] J. H. Choi and S. Vishwanathan. (2015, Jan.) Dfacto source code. [Online]. Available: http://web.ics.purdue.edu/~choi240/index.html

[19] N. Ravindran, N. D. Sidiropoulos, S. Smith, and G. Karypis, "Memory-efficient parallel computation of tensor and matrix products for big tensor decomposition," in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 2014.