Constrained Tensor Factorization with Accelerated AO-ADMM

Shaden Smith*, Alec Beri†, George Karypis*
* *Department of Computer Science and Engineering, University of Minnesota, Minneapolis, USA*
† *Department of Computer Science, University of Maryland, College Park, USA*
{*shaden, karypis*}*@cs.umn.edu, aberi@umd.edu*

***Abstract*—Low-rank sparse tensor factorization is a popular tool for analyzing multi-way data and is used in domains such as recommender systems, precision healthcare, and cybersecurity. Imposing constraints on a factorization, such as non-negativity or sparsity, is a natural way of encoding prior knowledge of the multi-way data. While constrained factorizations are useful for practitioners, they can greatly increase factorization time due to slower convergence and computational overheads. Recently, a hybrid of alternating optimization and alternating direction method of multipliers (AO-ADMM) was shown to have both a high convergence rate and the ability to naturally incorporate a variety of popular constraints. In this work, we present a parallelization strategy and two approaches for accelerating AO-ADMM. By redefining the convergence criteria of the inner ADMM iterations, we are able to split the data in a way that not only accelerates the per-iteration convergence, but also speeds up the execution of the ADMM iterations due to efficient use of cache resources. Secondly, we develop a method of exploiting dynamic sparsity in the factors to speed up tensor-matrix kernels. These combined advancements achieve up to $8\times$ speedup over the state-of-the-art on a variety of real-world sparse tensors.**

I. Introduction

Tensors are the generalization of matrices to higher orders. *Tensor factorization* is a powerful tool for approximating and analyzing multi-way data, and is popular in many domains across machine learning and signal processing, including recommender systems [1], precision healthcare [2], and cybersecurity [3]. These domains produce sparse tensors with millions to billions of non-zeros.

Oftentimes, a domain expert wishes to encode some prior knowledge of the data in order to obtain a more interpretable factorization. Prior knowledge is typically incorporated by either forcing the solution to take some form (i.e., imposing a *constraint*), or penalizing unwanted solutions (i.e., adding a *regularization*). For example, imposing a non-negativity constraint on a factorization allows one to better model data whose values are additive. Similarly, adding a regularization term which encourages sparsity can help model data whose interactions are sparse. While valuable to practitioners, constrained and regularized factorizations change the underlying computations and can significantly increase the computational cost of factorization.

There is a growing body of research dedicated to efficient optimization algorithms for constrained and regularized tensor factorization, especially non-negative factorization [4]–[6]. Huang et al. [7], [8] introduced AO-ADMM, a hybridization of alternating optimization (AO) with the alternating direction method of multipliers (ADMM). The combination of the two frameworks allows AO-ADMM to have both a fast convergence rate and the flexibility to incorporate new constraints and regularizations with minimal effort.

However, alongside the growing body of research is an increasing disparity between efficient optimization algorithms and the available implementations for large-scale tensors. Likewise, there are few available tools which flexibly support a variety of constraints, and to the best of our knowledge none of them are parallel or handle large-scale data. Domain experts must currently go through a major implementation effort to explore the application of a new constraint or regularization, and likely will not easily be able to analyze the full amount of available data due to computational complexity.

To that end, we present a parallelization strategy and high performance implementation of the AO-ADMM framework for shared-memory systems. Our algorithm features two optimizations: (i) a blockwise reformulation of ADMM to improve convergence rate, parallelism, and cache efficiency; and (ii) a method of exploiting the sparsity which dynamically evolves in the factorization. The blockwise reformulation is applicable to any constraint or regularization which is row separable (e.g., non-negativity or row simplex constraints), and factor sparsity naturally occurs in many constraints and regularizations including non-negativity.

In summary, our contributions include:

1) A blockwise reformulation of the AO-ADMM algorithm which improves convergence and execution rate while eliminating parallel synchronization overheads.
2) A method of leveraging sparsity in the factors as they dynamically evolve.
3) An open source, high performance implementation of AO-ADMM which flexibly handles new constraints and regularizations.

The rest of this paper is organized as follows. Section II introduces notation and details the AO-ADMM algorithm. Section III reviews existing work on matrix and tensor factorization. Section IV details the accelerated AO-ADMM algorithm. Section V provides experimental evaluation and

discussion. Lastly, we provide concluding remarks in Section VI.

II. Preliminaries

A. Background and Notation

Tensors are the generalization of matrices to higher dimensions, or *modes*. We denote tensors with bold calligraphic letters ($\boldsymbol{\mathcal{X}}$) and matrices as bold capital letters ($\mathbf{A}$). We focus on three-mode tensors for notational convenience, but emphasize that the algorithms described in this work are equally applicable to both matrices and higher order tensors.

A tensor non-zero with coordinate (i, j, k) is denoted $\boldsymbol{\mathcal{X}}(i, j, k)$, and matrix entries are similarly denoted $\mathbf{A}(i, j)$. A colon in place of an index indicates all non-zero entries. For example, $\mathbf{A}(i, :)$ is the ith row of $\mathbf{A}$.

A tensor can be *matricized* (i.e., flattened) along any of its modes. The matricized tensor along mode m is denoted $\mathbf{X}_{(m)}$. For example, the mode-1 matricization of $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$ is $\mathbf{X}_{(1)} \in \mathbb{R}^{I \times JK}$.

Two important matrix operations are the *Khatri-Rao* and the *Hadamard* products. The Khatri-Rao product, denoted $\odot$, is the columnwise Kronecker product [9]. The Khatri-Rao product of $\mathbf{B} \in \mathbb{R}^{J \times F}$ and $\mathbf{C} \in \mathbb{R}^{K \times F}$ is is $JK \times F$. The Hadamard product, denoted $*$, is the elementwise product of two matrices which must match in dimension.

B. Tensor Factorization

The canonical polyadic decomposition (CPD) is a widely used factorization for large, sparse tensors [10]. The rank-F CPD decomposes a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$ into factors $\mathbf{A} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$, and $\mathbf{C} \in \mathbb{R}^{K \times F}$. The CPD approximates a tensor as the summation of F outer products using the columns of the factors (shown in Figure 1). We are most often interested in a *low-rank* factorization, in which F is a small constant on the order of 10 or 100.

We focus on the least-squares formulation of the CPD, with loss function:

$$\mathrm{LS}(\boldsymbol{\mathcal{X}}, \mathbf{A}, \mathbf{B}, \mathbf{C}) = \left\| \boldsymbol{\mathcal{X}} - \sum_{f=1}^{F} \mathbf{A}(:, f) \circ \mathbf{B}(:, f) \circ \mathbf{C}(:, f) \right\|_F^2 .$$

Computing the CPD results in a non-convex optimization problem:

$$\underset{\mathbf{A},\mathbf{B},\mathbf{C}}{\text{minimize}} \quad \mathrm{LS}(\boldsymbol{\mathcal{X}}, \mathbf{A}, \mathbf{B}, \mathbf{C}) + r(\mathbf{A}) + r(\mathbf{B}) + r(\mathbf{C}), \quad (1)$$

where $r(\cdot)$ is a penalty function. Constraints can be implemented by having $r(\cdot)$ take the value of infinity when the constraint is violated, and regularizations use finite values to penalize unwanted (but valid) solutions. For example, a non-negativity constraint uses the indicator function of $\mathbb{R}_+$ and sparsity-inducing regularization uses $||\cdot||_1$. Due to the similar nature of constraints and regularizations, for the remainder of the paper we will use the terms interchangeably.

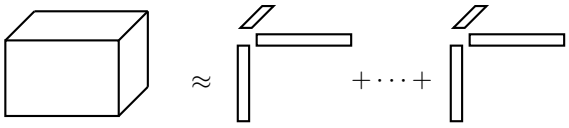


Figure 1: The CPD as the summation of outer products.

For further background on tensors and tensor factorization, we direct the reader to the survey by Kolda and Bader [10].

C. AO-ADMM

Equation (1) is non-convex and commonly solved via AO. When no constraints are enforced, AO becomes the alternating least squares (ALS) algorithm. AO-ADMM [7], [8] combines the AO framework with ADMM, which is a popular framework for constrained optimization problems [11]. AO-ADMM inherits positive qualities from each: a monotonically decreasing objective function from AO and the ability to flexibly incorporate constraints from ADMM.

AO-ADMM proceeds with a sequence of *outer* and *inner* iterations. Each step of an outer iteration optimizes one of the matrix factors by means of ADMM. Internally, ADMM executes a sequence of inner iterations to enforce constraints. Since the ADMM algorithm is the same across all of the tensor modes, we will simplify the discussion and only consider the computations associated with the first mode.

When discussing ADMM, we refer to the primal and dual variables as $\mathbf{H} \in \mathbb{R}^{I \times F}$ and $\mathbf{U} \in \mathbb{R}^{I \times F}$, respectively. An auxiliary variable $\widetilde{\mathbf{H}} \in \mathbb{R}^{F \times I}$ is introduced to arrive at a constrained optimization problem in the form of ADMM:

$$\begin{aligned} \underset{\mathbf{H},\widetilde{\mathbf{H}}}{\text{minimize}} \quad & \frac{1}{2} \left\| \mathbf{X}_{(1)} - \widetilde{\mathbf{H}}^T (\mathbf{C} \odot \mathbf{B})^T \right\|_F^2 + r(\mathbf{H}) \quad (2) \\ \text{subject to} \quad & \mathbf{H} = \widetilde{\mathbf{H}}^T . \end{aligned}$$

Algorithm 1 details the resulting ADMM algorithm. It accepts as input the primal and dual variables and two additional matrices: (i) $\mathbf{K} \in \mathbb{R}^{I \times F}$, the *matricized tensor times Khatri-Rao product* (MTTKRP) which is formed via $\mathbf{K} \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$; and (ii) $\mathbf{G} \in \mathbb{R}^{F \times F}$, the Gram matrix which is formed via $\mathbf{G} \leftarrow (\mathbf{C} \odot \mathbf{B})^T (\mathbf{C} \odot \mathbf{B}) = (\mathbf{B}^T \mathbf{B} * \mathbf{C}^T \mathbf{C})$. Line 6 executes forward- and backward-substitution on an $F \times I$ matrix in $\mathcal{O}(F^2 I)$ time. Line 8 is the *proximity operator* and varies based on $r(\cdot)$. For example, non-negativity constraints project to the non-negative orthant (i.e., "zero out" negative entries). Line 9 updates the dual variable and lastly, lines 10 and 11 compute the relative primal and dual residuals.

Finally, the complete AO-ADMM framework is detailed in Algorithm 2. The factors are cyclically updated using Algorithm 1. Each MTTKRP operation requires $\mathcal{O}(F \operatorname{nnz}(\boldsymbol{\mathcal{X}}))$ operations and is often the most expensive step of the AO-ADMM framework. The relative costs of the factorization steps are further explored in Section V.

Algorithm 1 ADMM to solve Equation (2)

Input: $\mathbf{H}$, $\mathbf{U}$, $\mathbf{K}$, $\mathbf{G}$
Output: $\mathbf{H}$, $\mathbf{U}$
$\rho \leftarrow \text{trace}(\mathbf{G})/F$
$\mathbf{L} \leftarrow \text{Cholesky}(\mathbf{G} + \rho\mathbf{I})$
repeat ▷ Inner iterations
  $\widetilde{\mathbf{H}} \leftarrow \mathbf{L}^{-T}\mathbf{L}^{-1}\left(\mathbf{K} + \rho(\mathbf{H} + \mathbf{U})\right)^T$
  $\mathbf{H}_0 \leftarrow \mathbf{H}$
  $\mathbf{H} \leftarrow \text{argmin}_{\mathbf{H}}\, r(\mathbf{H}) + \frac{\rho}{2}||\mathbf{H} - \widetilde{\mathbf{H}}^T + \mathbf{U}||_F^2$
  $\mathbf{U} \leftarrow \mathbf{U} + \mathbf{H} - \widetilde{\mathbf{H}}^T$
  $r \leftarrow ||\mathbf{H} - \widetilde{\mathbf{H}}^T||_F^2 / ||\mathbf{H}||_F^2$
  $s \leftarrow ||\mathbf{H} - \mathbf{H}_0||_F^2 / ||\mathbf{U}||_F^2$
until $r < \epsilon$ and $s < \epsilon$

Algorithm 2 AO-ADMM

Initialize primal variables $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ randomly.
Initialize dual variables $\hat{\mathbf{A}}$, $\hat{\mathbf{B}}$, and $\hat{\mathbf{C}}$ with $\mathbf{0}$.
repeat ▷ Outer iterations
  $\mathbf{G} \leftarrow \mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C}$
  $\mathbf{K} \leftarrow \mathbf{X}_{(1)}\left(\mathbf{C} \odot \mathbf{B}\right)$ ▷ MTTKRP
  $\mathbf{A}, \hat{\mathbf{A}} \leftarrow \text{ADMM}(\mathbf{A}, \hat{\mathbf{A}}, \mathbf{K}, \mathbf{G})$ ▷ Algorithm 1

  $\mathbf{G} \leftarrow \mathbf{A}^T\mathbf{A} * \mathbf{C}^T\mathbf{C}$
  $\mathbf{K} \leftarrow \mathbf{X}_{(2)}\left(\mathbf{C} \odot \mathbf{A}\right)$ ▷ MTTKRP
  $\mathbf{B}, \hat{\mathbf{B}} \leftarrow \text{ADMM}(\mathbf{B}, \hat{\mathbf{B}}, \mathbf{K}, \mathbf{G})$ ▷ Algorithm 1

  $\mathbf{G} \leftarrow \mathbf{A}^T\mathbf{A} * \mathbf{B}^T\mathbf{B}$
  $\mathbf{K} \leftarrow \mathbf{X}_{(3)}\left(\mathbf{B} \odot \mathbf{A}\right)$ ▷ MTTKRP
  $\mathbf{C}, \hat{\mathbf{C}} \leftarrow \text{ADMM}(\mathbf{C}, \hat{\mathbf{C}}, \mathbf{K}, \mathbf{G})$ ▷ Algorithm 1
until $\text{LS}(\mathcal{X}, \mathbf{A}, \mathbf{B}, \mathbf{C})$ ceases to improve.

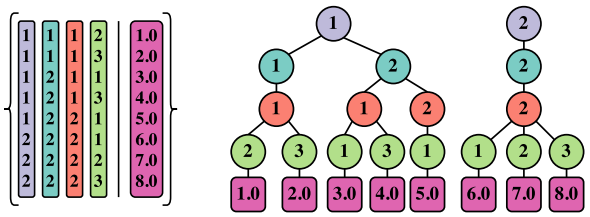
III. Related Work

A. Constrained Factorization

There is a large body of research dedicated to optimization algorithms for constrained tensor factorization, especially in the context of non-negativity constraints. Zhang et al. [4] presented a parallel algorithm for dense non-negative tensor factorization using projected gradient descent. Non-negative tensor factorization was formulated for the ADMM framework by Liavas and Sidiropoulos [12]. Recently, Kannan et al. [13] developed a parallel algorithm for dense and sparse non-negative matrix factorization using non-negative least squares. For additional background on non-negative factorizations, we direct the reader to the survey by Zhou et al. [6] and the book by Cichocki et al. [14].

B. MTTKRP with a Sparse Tensor

AO-ADMM relies on computational kernels which are also present while computing the unconstrained CPD. Specifically, MTTKRP is also the most expensive computational kernel in the unconstrained CPD and has received attention by the high performance computing community [15]–[20]. However, while many works have considered the case when the tensor is sparse, no works to our knowledge have addressed the case when both the tensor and factor matrices are sparse.



(a) Coordinate (b) CSF

Figure 2: Encodings of a four-mode tensor with 5 non-zeros.

Algorithm 3 MTTKRP with a three-mode CSF tensor

Input: Tensor $\mathcal{X}$ and factor matrices $\mathbf{B}$ and $\mathbf{C}$
Output: $\mathbf{K} \leftarrow \mathbf{X}_{(1)}\left(\mathbf{C} \odot \mathbf{B}\right)$

for $i \in \{1, \ldots, I\}$ in parallel **do**
  $\mathbf{K}(i,:) \leftarrow \mathbf{0}$
  for $j \in \mathcal{X}(i,:,:)$ **do**
    $\mathbf{z} \leftarrow \mathbf{0}^{F \times 1}$ ▷ Buffer for accumulation.
    for $k \in \mathcal{X}(i,j,:)$ **do**
      $\mathbf{z} \leftarrow \mathbf{z} + \mathcal{X}(i,j,k)\mathbf{C}(k,:)$
    end for
    $\mathbf{K}(i,:) \leftarrow \mathbf{K}(i,:) + \mathbf{z} * \mathbf{B}(j,:)$
  end for
end for

In prior work, we presented the *compressed sparse fiber* (CSF) data structure for sparse tensors [21]. CSF, shown in Figure 2, can be viewed as a higher-order generalization of the compressed sparse row (CSR) storage format for sparse matrices. CSF compresses the modes of a sparse tensor recursively such that each path from a root to a leaf node encodes the coordinates of a non-zero. The non-zero values are stored at the lowest level of the trees.

Algorithm 3 presents a parallel algorithm for performing MTTKRP for the first mode, using a three-mode CSF tensor. The algorithm's derivation and generalization to higher modes can be found in prior work [16], [21]. The computation takes the form of three nested loops, each traversing a mode of the tensor. The innermost operations scales rows of $\mathbf{C}$ by the tensor non-zeros.

IV. Accelerated AO-ADMM

We now present techniques for parallelizing and accelerating AO-ADMM (Algorithm 2). We begin with a parallelization strategy for ADMM (Algorithm 1) and then discuss a reformulation which improves convergence and computational

efficiency. Next, we introduce a strategy for accelerating MTTKRP by exploiting the sparsity that naturally occurs in the factor matrices.

A. Parallelized ADMM

There is a wealth of research that focuses on accelerating the individual dense matrix kernels that constitute ADMM. These include matrix multiplication, Cholesky factorization, and forward/backward substitution [22]. Since the matrices of interest are tall and skinny, the kernels will ultimately be parallelized over the matrix rows while carefully optimizing for cache and other hardware features. Additionally, many popular constraints have proximity operators which are row separable. These include l_1 regularization for sparsity, non-negativity, and row simplex constraints.

A consequence of row separable computations is that Lines 6 through 9 in Algorithm 1, which comprise the bulk of the ADMM computation, can all be parallelized by distributing rows of the tall and skinny matrices to threads. Lastly, convergence can be computed in parallel using any decomposition of the primal and dual variables. Each thread computes thread-local primal and dual norms (Lines 10 and 11) which are then aggregated.

B. Blocked ADMM

While the previous parallelization strategy offers a large amount of parallelism by focusing on the individual kernels, it does not take into account the iterative nature of the ADMM as a whole. We consider two challenges that emerge when viewing the ADMM algorithm beyond just a sequence of optimized kernels:

Non-uniform convergence: Real-world datasets often exhibit non-zeros which follow a power-law distribution. For example, a product rating tensor used by a recommender system will have some popular items and prolific users, while on average each item and user only have a few submitted ratings. It is natural to expect the rows corresponding to prolific users and items to carry much of the factorization's information. A consequence is that these "high-signal" rows may require many more iterations to converge than the average row. Since convergence criteria is an aggregation of all rows, this disparity not only decreases factorization quality by performing too few iterations on the high-signal rows, but also increases factorization time by performing additional iterations on the low-signal rows.

Memory bandwidth: Each step in Algorithm 1 involves a linear pass over the primal and dual matrices. Note that even the computationally intensive step, the forward/backward substitutions requiring $\mathcal{O}(F^2 I)$ operations, is linear in the large row dimension. If the size of the matrices exceed the size of the CPU cache, then we will access the matrices entirely from main memory instead of cache. Thus, the performance of Algorithm 1 will be determined by a machine's memory bandwidth instead of its compute capabilities.

We address both limitations by developing a blockwise reformulation of Equation 2. If the proximity operator is row separable, we split the problem into B blocks of rows:

$$\begin{aligned} \underset{\{(\mathbf{H}_i, \widetilde{\mathbf{H}}_i)\}}{\text{minimize}} \quad & \sum_{b=1}^{B} \frac{1}{2} \left\| (\mathbf{X}_{(1)})_b - \widetilde{\mathbf{H}}_b^T (\mathbf{C} \odot \mathbf{B})_b^T \right\|_F^2 + r(\mathbf{H}_b) \\ \text{subject to} \quad & \mathbf{H}_1 = \widetilde{\mathbf{H}}_1, \ldots, \mathbf{H}_B = \widetilde{\mathbf{H}}_B. \end{aligned}$$

Each of the blocks can then be optimized independently using Algorithm 1.

The blockwise reformulation accelerates convergence by allowing each block of rows to converge using different numbers of iterations. At one extreme, if $B = I$ then we separately optimize the solution for each row. In effect, the amount of work performed on each row is independent of all others. Therefore, the rows which take many iterations to converge will not be affected by those which converge early. Similarly, computation will not be wasted on rows which have already converged.

Likewise, the blockwise reformulation improves computational performance by creating temporal locality in the matrices. Since a block is processed until it has converged, a sufficiently small block size will allow of the necessary matrix data to be cache resident throughout the optimization procedure. Hence, we can expect to achieve better performance than the previously memory-bound formulation.

From a parallelization standpoint, this blockwise reformulation naturally provides an alternative decomposition of the computations. Instead of parallelizing the individual steps within Algorithm 1, we can simply distribute blocks to threads. This has the benefit of eliminating all synchronization overheads, as each block can be optimized totally in parallel. Even though blocks are equal in size, they may require different numbers of iterations. Thus, we cannot statically distribute blocks and instead dynamically load balance the optimization at block-level granularity. The work distribution is a simple loop over the B blocks and can be managed by the dynamic looping mechanism provided by many parallel frameworks such as OpenMP.

Selecting the number of blocks affects both convergence rate and execution performance. A natural first choice is to use $B{=}I$ and optimize over each row individually. In effect, the convergence benefits are maximized while guaranteeing cache residency for all but very high-rank factorizations. Unfortunately, other overheads such as function calls and instruction cache misses are exaggerated when such a small amount of work is performed in each step. We empirically found that blocks of 50 rows offered a good trade-off between convergence and execution.

While we focus on shared-memory parallelism in this work, we note that the blockwise formulation also affords

opportunities for distributed-memory parallelism. Since each block is processed independently, no communication needs to occur beyond the MTTKRP operation, which has efficient distributed-memory algorithms [17], [23].

C. MTTKRP with Sparse Factors

Sparsity in the CPD factors is an attractive characteristic to practitioners. Intuitively, a sparse solution is a more simple one, and is thus easier to interpret and to gain insight from. Factor sparsity is also attractive from a performance perspective, as it affords opportunities for computational savings by avoiding operations with zero elements.

We focus our sparsity optimizations on the MTTKRP operation. Each tensor non-zero results in an access to the matrix factors, and thus MTTKRP can benefit greatly from sparsity. MTTKRP is primarily bound by memory bandwidth due to accesses to the factor matrices [16], [19], and thus optimizations should reduce the volume of data fetched from the factors in order to achieve speedups. This is especially challenging when computing with a small rank, as each row has relatively few elements regardless, and thus memory savings are limited to a fraction of a cache line.

A first solution is to store a copy of the sparse factors in CSR format. CSR allows the factors to be randomly accessed by rows, which is required during MTTKRP (Algorithm 3). Since only the non-zero values and their indices are represented, the amount of data fetched from main memory scales with the matrix sparsity. Fortunately, the switch from a dense to compressed matrix format only requires minor changes to the MTTKRP algorithm. Whole rows of the factor are accessed at a time, and thus we only need to account for the difference between a dense and sparse row representation. In practice, the cost of accessing $\mathbf{C}$ dominates accesses to the other factor matrices due to it being accessed by every non-zero instead of fiber or slice. Therefore, we only represent $\mathbf{C}$ in CSR form and only need to modify Line 9 of Algorithm 3.

While a CSR representation *decreases* the effects of limited memory bandwidth, it *increases* the effects of memory latency. Consider the difference in implementation of dense and CSR matrices. A dense matrix will incur one latency cost when initially accessing a row of $\mathbf{C}$, but the remaining entries exhibit spatial locality and will be fetched from main memory within the same cache line. Adjacent cache lines should be fetched by the hardware prefetching mechanisms. A CSR matrix, however, is implemented with three structures which encode the row length, the non-zero indices, and the non-zero values. The row length is required to index into the indices and values, and thus multiple latency costs will be incurred.

We address the challenge of memory latency costs by considering a hybrid combination of the dense and sparse matrix structures. Much like the distribution of tensor non-zeros, the sparsity patterns of the matrix factors are non-uniform. Importantly, $\mathbf{C}$ may have a few mostly-dense columns, with the remaining ones containing only a few non-zeros. We call a column "dense" when it contains more non-zeros than the average column density. When constructing the hybrid structure, we first sort the columns based on the number of non-zeros and place the dense columns first. The dense columns are represented with a simple dense matrix, and the remaining sparse columns are stored in CSR format. When a row of $\mathbf{C}$ is accessed during MTTKRP, we use software prefetching to begin fetching the CSR structure. During the data movement, we compute with the dense entries of the row. Finally, the row of the CSR structure is processed after the dense component.

Unlike the tensor which has a static sparsity pattern throughout the factorization, the sparsity patterns of the factors are dynamic. Therefore, techniques to exploit sparsity must be carefully vetted for efficiency, because their overheads are not amortized over multiple iterations. The potential gains that can be achieved by using a CSR representation to accelerate MTTKRP need to be balanced with the cost of constructing this CSR representation. Constructing the CSR matrix is an $\mathcal{O}(IF)$ operation due to it requiring a pass over the dense matrix to determine the sparsity pattern. Fortunately, this overhead is negligible when multiple ADMM iterations are performed, each taking $\mathcal{O}(F^2I)$ time.

Table I: Summary of datasets.

Dataset	NNZ	I	J	K
`Reddit` [24]	95M	310K	6K	510K
`NELL` [25]	143M	3M	2M	25M
`Amazon` [26]	1.7B	5M	18M	2M
`Patents` [27]	3.5B	46	240K	240K

NNZ is the number of nonzero entries in the dataset. **I**, **J**, and **K** are the dimensions of the datasets. *K*, *M*, and *B* stand for thousand, million, and billion, respectively.

V. Results

A. Experimental Methodology

Datasets: Table I summarizes the tensors used in our evaluation. We selected four real-world tensors from various domains which are publicly available as part of the FROSTT collection [27]. `NELL` is a *noun-verb-noun* tensor from the Never Ending Language Learning project [25]. `Reddit` is a *user-community-word* tensor encoding a subset of comments on Reddit from 2007 to 2010 [24]. `Amazon` is a *user-item-word* tensor of product reviews [26]. `Patents` is a *year-word-word* tensor of pairwise co-occurrence probabilities from United States utilities patents.

Machine & Software Configuration: Experiments were performed on a workstation with 396GB of main memory and two ten-core Intel Xeon E5-2650v3 processors with 25MB of last-level cache. Our source code is modified from SPLATT version 1.1.1, a C library for high performance sparse tensor factorization [28]. Our source code is to be made part of the next SPLATT release. We use the Intel

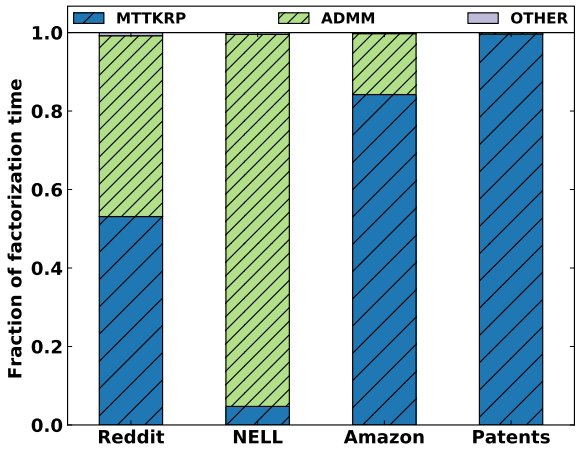


Figure 3: Fraction of time spent in MTTKRP and ADMM during rank-50 non-negative factorization.

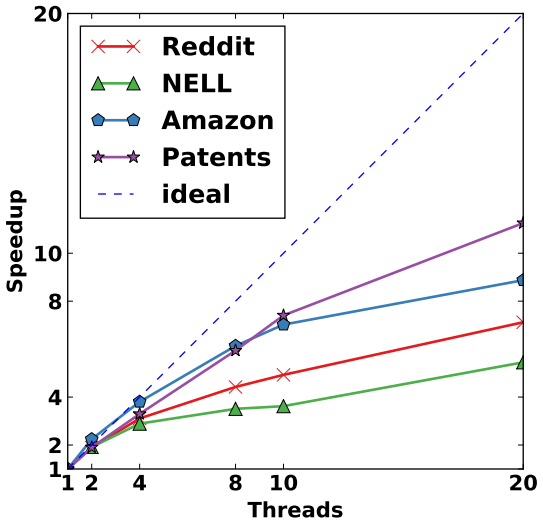


Figure 4: Speedup on baseline rank-50 non-negative CPD.

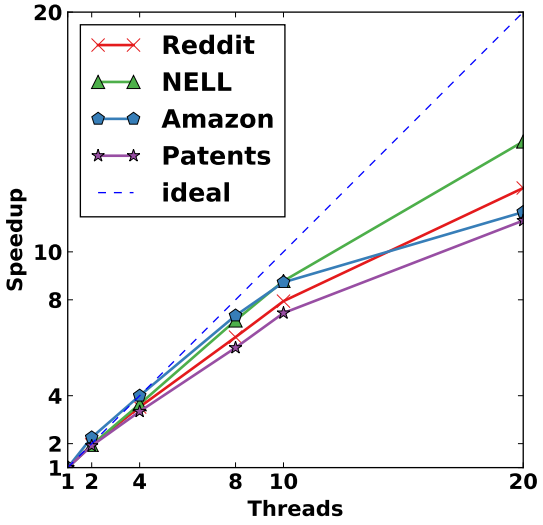


Figure 5: Speedup on blocked rank-50 non-negative CPD.

compiler version 17.0.1 and Intel MKL used for Cholesky factorization and forward/backward substitution. OpenMP is used for parallelism. Unless otherwise specified, we run with one twenty OpenMP threads.

Convergence Criteria: We follow the factorization community and use a normalized value of $\text{LS}(\cdot)$ to measure the quality of a factorization. Specifically, we measure the relative error between $\mathcal{X}$ and its factored form:

$$\text{relative error} = \frac{\left\| \mathcal{X} - \sum_{f=1}^{F} \mathbf{A}(:,f) \circ \mathbf{B}(:,f) \circ \mathbf{C}(:,f) \right\|_F^2}{\|\mathcal{X}\|_F^2}.$$

Convergence is detected when the relative error improves less than 10^{-6} or if we exceed 200 outer iterations.

B. Relative Factorization Costs

We first investigate the performance characteristics of a parallel implementation of AO-ADMM without blocking or sparsity optimizations. Figure 3 shows the fraction of factorization time spent in the main computational kernels of Algorithm 2 (i.e., MTTKRP and ADMM) during a rank-50 non-negative factorization.

Neither of the kernels consistently dominate the computation. `NELL` has both the longest modes of the datasets and is also the most sparse, and therefore spends most of the runtime in ADMM updating the factors. `Amazon` and `Patents`, on the other hand, are dominated by MTTKRP. These tensors have more non-zeros and are both more dense than `NELL`, which emphasizes the cost of MTTKRP. These results indicate that in order to achieve high performance, both the computations performed during ADMM and MTTKRP need to be optimized and parallelized effectively.

C. Parallel Scalability

We examine the parallel scalability of the baseline AO-ADMM algorithm in Figure 4. The baselines achieve speedups ranging from $5.4\times$ on `NELL` to $12.7\times$ on `Patents`. Note that the amount of achieved speedup is related to the amount of time spent on MTTKRP in Figure 3. The datasets which are dominated by the cost of MTTKRP exhibit the best scalability due to the already-optimized kernels provided by SPLATT.

We now observe the parallel scalability of the blocked AO-ADMM algorithm in Figure 5. Speedups range from $12.7\times$ on `Patents` to $14.6\times$ on `NELL`. The trend observed on the baseline scalability is reversed: datasets which are dominated by ADMM runtime now achieve the best scalability. This is expected, as blocked ADMM features high temporal locality and minimal synchronization costs compared to its baseline counterpart.

Table II: Effects of sparse matrix data structures on CPD runtime.

	Reddit			Amazon		
	$F = 50$ 3% dense	$F = 100$ 1% dense	$F = 200$ 2% dense	$F = 50$ 3% dense	$F = 100$ 3% dense	$F = 200$ 5% dense
DENSE	227.8	430.7	1774.9	305.9	715.3	18120.0
CSR	212.7	231.6	1000.5	**272.6**	**539.2**	**12535.6**
CSR-H	**199.4**	**186.3**	**903.5**	320.6	588.5	13476.5

Values are the total time in seconds to compute the CPD. We impose a $10^{-1}||\cdot||_1$ regularization on all factors to promote sparsity. The density of each rank indicates the density of the longest factor matrix (i.e., the matrix that is stored in a sparse representation during MTTKRP). Density is computed via $\text{nnz}(\mathbf{C})/KF$. DENSE uses a baseline MTTKRP implementation with a dense matrix. CSR uses the compressed sparse row (CSR) format during MTTKRP. CSR-H uses the hybrid dense and CSR format.

D. Convergence Rate

We now evaluate the benefits of the blockwise formulation on the convergence of AO-ADMM. It is important to separate the speedups achieved by accelerated convergence and by faster execution rate. Figure 6 shows convergence on all datasets as both a function of time and the number of outer iterations. Including convergence as a function of outer iteration allows us to observe the effects of blocked ADMM without considering machine effects such as cache locality. When a solution of higher quality (i.e., lower error) is reached, or fewer outer iterations are performed, then we know that convergence has been improved.

Blocking improves the per-iteration convergence on every evaluated dataset. The positive benefits of blocking are observed in two forms: (i) reaching a higher-quality solution in the same or less time, or (ii) converging to a comparable solution in fewer iterations. For example, `NELL` and `Amazon` both converge to lower errors than the baselines. This is most exemplified with `NELL`, which converges $3.7\times$ faster and reaches a 3% lower error. The success of `NELL` is a result of the combination of both faster ADMM iterations and additional ADMM iterations being performed on the "high-signal" blocks. `Reddit` and `Patents`, on the other hand, converge in fewer iterations due to blocking and reach errors that are less than 1% higher than the baseline.

E. Accelerating MTTKRP with Factor Sparsity

We now evaluate the benefits of exploiting factor sparsity during MTTKRP. We compute l_1-regularized factorizations in order find to sparse solutions. For each factor, we set $r(\cdot) = 10^{-1}||\cdot||_1$. We omit `NELL` and `Patents` from this evaluation because they did not tend to exhibit sparsity and instead converged to either mostly dense or totally zero solutions as the regularization parameter was introduced.

Table II shows the time-to-solution for `Reddit` and `Amazon` on a variety of ranks. For each configuration, we include times for the baseline dense computation, CSR, and the hybrid dense and CSR computation. Notably, the complete factorization time is presented despite the evaluated algorithms only benefiting the MTTKRP portion of the factorization. The time-to-solution more accurately portrays the benefits of sparsity as it accounts for conversion overheads and the early iterations in which the factors are not yet sparse. For our evaluation, we empirically determined that a factor can be gainfully treated as sparse when its density falls below 20%.

Exploiting sparsity outperforms the baseline dense computation in all cases. Speedups from sparse MTTKRP range from $1.1\times$ to $2.3\times$. Interestingly, the hybrid CSR data structure is beneficial for `Reddit` but not `Amazon`. While the two datasets feature similar levels of sparsity in their largest factor matrix, the longest mode of `Amazon` is over thirty times longer than `Reddit`. Further investigation is needed in order to predict the best choice of data structure based on the tensor properties, and is left to future work.

VI. Conclusions & Future Work

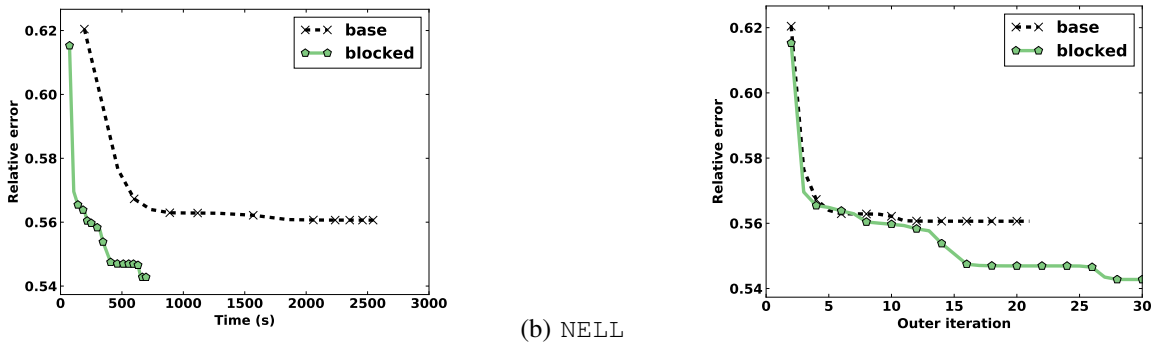
We studied the acceleration and high performance implementation of AO-ADMM, a recent framework for constrained tensor factorization. We presented a parallelization and two optimizations which together accelerate the complete factorization process. First, a blockwise reformulation improves performance up to $4\times$ by creating temporal cache locality, eliminating parallel synchronization costs, and accelerating convergence. Second, we exploit the sparsity that dynamically emerges in the factorization output to reduce operations and memory bandwidth in the primary tensor kernel, achieving up to $2.3\times$ additional speedup.

There are several items of future work. First, further investigation is required in order to automatically select the best data structure for the sparse matrix factors during MTTKRP. Second, an analytical model of the ADMM algorithm could provide a method of choosing block sizes.

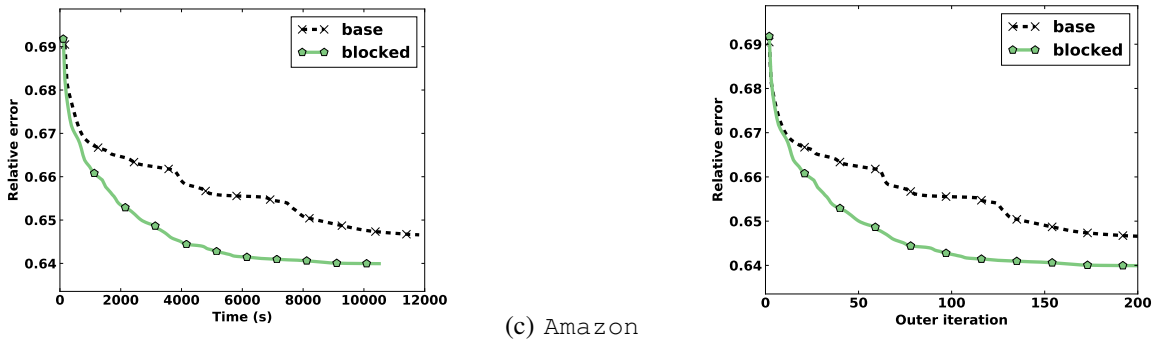
Acknowledgments

We thank the anonymous reviewers for insightful comments and suggestions for future work. This work was supported in part by NSF (IIS-1460620, IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), a University of Minnesota Doctoral Dissertation Fellowship, Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing

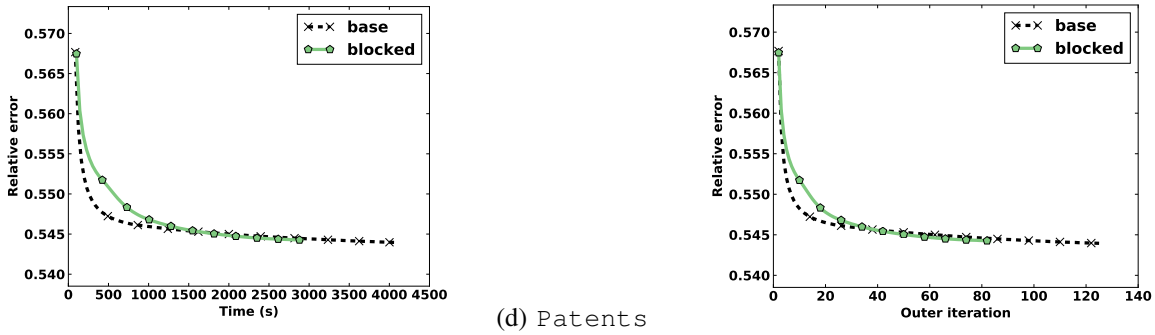
(a) Reddit



(b) NELL



(c) Amazon



(d) Patents

Figure 6: Effects of blockwise ADMM on rank-50 non-negative factorization. **base** and **blocked** are the unblocked and blocked algorithms, respectively.

Institute. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] 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.

[2] Y. Wang, R. Chen, J. Ghosh, J. C. Denny, A. Kho, Y. Chen, B. A. Malin, and J. Sun, "Rubik: Knowledge guided tensor factorization and completion for health data analytics," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1265–1274.

[3] H. Fanaee-T and J. Gama, "Tensor-based anomaly detection: An interdisciplinary survey," *Knowledge-Based Systems*, vol. 98, pp. 130–147, 2016.

[4] Q. Zhang, M. W. Berry, B. T. Lamb, and T. Samuel, "A parallel nonnegative tensor factorization algorithm for mining global climate data," in *Computational Science–ICCS 2009*. Springer, 2009, pp. 405–415.

[5] A. Cichocki and P. Anh-Huy, "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 92, no. 3, pp. 708–721, 2009.

[6] G. Zhou, A. Cichocki, Q. Zhao, and S. Xie, "Nonnegative matrix and tensor factorizations: An algorithmic perspective," *IEEE Signal Processing Magazine*, vol. 31, no. 3, pp. 54–65, 2014.

[7] K. Huang, N. D. Sidiropoulos, and A. P. Liavas, "A flexible and efficient algorithmic framework for constrained matrix and tensor factorization," *IEEE Transactions on Signal Processing*, vol. 64, no. 19, pp. 5052–5065, 2016.

[8] ——, "Efficient algorithms for 'universally' constrained matrix and tensor factorization," in *Signal Processing Conference (EUSIPCO), 2015 23rd European*. IEEE, 2015, pp. 2521–2525.

[9] T. G. Kolda and B. Bader, "The TOPHITS model for higher-order web link analysis," in *Proceedings of Link Analysis, Counterterrorism and Security 2006*, 2006.

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

[11] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[12] A. P. Liavas and N. D. Sidiropoulos, "Parallel algorithms for constrained tensor factorization via the alternating direction method of multipliers," *arXiv preprint arXiv:1409.2383*, 2014.

[13] R. Kannan, G. Ballard, and H. Park, "A high-performance parallel algorithm for nonnegative matrix factorization," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, p. 9.

[14] A. Cichocki, R. Zdunek, A. H. Phan, and S.-i. Amari, *Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation*. John Wiley & Sons, 2009.

[15] M. Baskaran, B. Meister, and R. Lethin, "Low-overhead load-balanced scheduling for sparse tensor computations," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.

[16] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *International Parallel & Distributed Processing Symposium (IPDPS'15)*, 2015.

[17] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 77.

[18] O. Kaya and B. Uçar, "Parallel CP decomposition of sparse tensors using dimension trees," Inria - Research Centre Grenoble – Rhône-Alpes, Research Report RR-8976, Nov. 2016.

[19] S. Smith, J. Park, and G. Karypis, "Sparse tensor factorization on many-core processors with high-bandwidth memory," *31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)*, 2017.

[20] J. Li, J. W. Choi, I. Perros, J. Sun, and R. Vuduc, "Model-driven sparse CP decomposition for higher-order tensors," *31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)*, 2017.

[21] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 7.

[22] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.

[23] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," in *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS'16)*, 2016.

[24] J. Baumgartner, "Reddit comment dataset," https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/, 2015.

[25] 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.

[26] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 2013, pp. 165–172.

[27] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[28] S. Smith and G. Karypis, "SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit," http://cs.umn.edu/~splatt/.