Constrained Tensor Factorization with Accelerated AO-ADMM

Shaden Smith\textsuperscript{1}\textsuperscript{*}, Alec Beri\textsuperscript{2}, and George Karypis\textsuperscript{1}

\textsuperscript{1}Department of Computer Science & Engineering, University of Minnesota
\textsuperscript{2}Department of Computer Science, University of Maryland
\textsuperscript{*}shaden@cs.umn.edu
Table of Contents

Introduction

Accelerated AO-ADMM

Experiments

Conclusions
Tensor Introduction

- Tensors are the generalization of matrices to higher dimensions.
- Allow us to represent and analyze multi-dimensional data.
- Applications in precision healthcare, cybersecurity, recommender systems, ...
Canonical polyadic decomposition (CPD)

The CPD models a tensor as the summation of rank-1 tensors.

\[
\begin{align*}
\mathbf{X} & \approx \sum_{f=1}^{F} \mathbf{A}(:, f) \circ \mathbf{B}( :, f) \circ \mathbf{C}( :, f) \\
\end{align*}
\]

minimize \( \mathbf{A}, \mathbf{B}, \mathbf{C} \)

\[
\mathcal{L}(\mathbf{X}, \mathbf{A}, \mathbf{B}, \mathbf{C}) = \left\| \mathbf{X} - \sum_{f=1}^{F} \mathbf{A}(:, f) \circ \mathbf{B}( :, f) \circ \mathbf{C}( :, f) \right\|_F^2
\]

Notation

\( \mathbf{A} \in \mathbb{R}^{I \times F} \), \( \mathbf{B} \in \mathbb{R}^{J \times F} \), and \( \mathbf{C} \in \mathbb{R}^{K \times F} \) denote the factor matrices for a 3D tensor.
Alternating least squares (ALS)

The CPD is most commonly computed with ALS:

**Algorithm 1 CPD-ALS**

1: **while** not converged **do**
2: \[ A^T \leftarrow (C^T C \ast B^T B)^{-1} (X(1)(C \odot B))^T \]
3: \[ B^T \leftarrow (C^T C \ast A^T A)^{-1} (X(2)(C \odot A))^T \]
4: \[ C^T \leftarrow (B^T B \ast A^T A)^{-1} (X(3)(B \odot A))^T \]

5: **end while**

**Notation**

* denotes the Hadamard (elementwise) product.
Constrained factorization

We often want to impose some constraints or regularizations on the factorization:

\[
\text{minimize}_{A,B,C} \quad \underbrace{\mathcal{L}(X, A, B, C)}_{\text{Loss}} + \underbrace{r(A) + r(B) + r(C)}_{\text{Constraints/Regularizations}}
\]

**Example**

Non-negative factorizations use an indicator function for \(\mathbb{R}_+\):

\[
r(A) = \begin{cases} 
0 & \text{if } A \geq 0 \\
\infty & \text{otherwise}
\end{cases}
\]
AO-ADMM [Huang & Sidiropoulos ’15]

AO-ADMM combines alternating optimization (AO) with alternating direction method of multipliers (ADMM).

- **A**, **B**, and **C** are updated in sequence using ADMM.
AO-ADMM [Huang & Sidiropoulos ’15]

AO-ADMM combines alternating optimization (AO) with alternating direction method of multipliers (ADMM).

- A, B, and C are updated in sequence using ADMM.

ADMM formulation for the update of A:

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} \| X^{(1)} - \tilde{A}^T (C \odot B)^T \|_F^2 + r(A) \\
\text{subject to} & \quad A = \tilde{A}^T.
\end{align*}
\]
Alternating optimization step (outer iterations)

1: Initialize primal variables $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ randomly.
2: Initialize dual variables $\hat{\mathbf{A}}$, $\hat{\mathbf{B}}$, and $\hat{\mathbf{C}}$ with $\mathbf{0}$.
3: repeat
4: $\mathbf{G} \leftarrow \mathbf{B}^T \mathbf{B} \ast \mathbf{C}^T \mathbf{C}$
5: $\mathbf{K} \leftarrow \mathbf{X}_{(1)} (\mathbf{C} \odot \mathbf{B})$
6: $\mathbf{A}, \hat{\mathbf{A}} \leftarrow \text{ADMM}(\mathbf{A}, \hat{\mathbf{A}}, \mathbf{K}, \mathbf{G})$
7: $\mathbf{G} \leftarrow \mathbf{A}^T \mathbf{A} \ast \mathbf{C}^T \mathbf{C}$
8: $\mathbf{K} \leftarrow \mathbf{X}_{(2)} (\mathbf{C} \odot \mathbf{A})$
9: $\mathbf{B}, \hat{\mathbf{B}} \leftarrow \text{ADMM}(\mathbf{B}, \hat{\mathbf{B}}, \mathbf{K}, \mathbf{G})$
10: $\mathbf{G} \leftarrow \mathbf{A}^T \mathbf{A} \ast \mathbf{B}^T \mathbf{B}$
11: $\mathbf{K} \leftarrow \mathbf{X}_{(3)} (\mathbf{B} \odot \mathbf{A})$
12: $\mathbf{C}, \hat{\mathbf{C}} \leftarrow \text{ADMM}(\mathbf{C}, \hat{\mathbf{C}}, \mathbf{K}, \mathbf{G})$
13: until $\mathcal{L} (\mathbf{X}, \mathbf{A}, \mathbf{B}, \mathbf{C})$ ceases to improve.
ADMM step (inner iterations)

ADMM to update one factor matrix:

1. **Input:** $H$, $U$, $K$, $G$
2. **Output:** $H$, $U$
3. $\rho \leftarrow \text{trace}(G)/F$
4. $L \leftarrow \text{Cholesky}(G + \rho I)$
5. repeat
6. $H_0 \leftarrow H$
7. $\tilde{H} \leftarrow L^{-T} L^{-1} (K + \rho(H + U))^{T}$
8. $H \leftarrow \text{argmin}_{H} r(H) + \frac{\rho}{2}||H - \tilde{H}^{T} + U||^2_F$
9. $U \leftarrow U + H - \tilde{H}^{T}$
10. $r \leftarrow ||H - \tilde{H}^{T}||^2_F / ||H||^2_F$
11. $s \leftarrow ||H - H_0||^2_F / ||U||^2_F$
12. until $r < \epsilon$ and $s < \epsilon$
Table of Contents

Introduction

Accelerated AO-ADMM

Experiments

Conclusions
Parallelization opportunities

All steps but Line 8 are either element-wise or row-wise independent.

1: **Input:** $H, U, K, G$
2: **Output:** $H, U$
3: $\rho \leftarrow \text{trace}(G)/F$
4: $L \leftarrow \text{Cholesky}(G + \rho I)$
5: **repeat**
6: $H_0 \leftarrow H$
7: $\tilde{H} \leftarrow L^{-T}L^{-1}(K + \rho(H + U))^T$
8: $H \leftarrow \text{argmin}_H r(H) + \frac{\rho}{2}\|H - \tilde{H}^T + U\|_F^2$
9: $U \leftarrow U + H - \tilde{H}^T$
10: $r \leftarrow \|H - \tilde{H}^T\|_F^2/\|H\|_F^2$
11: $s \leftarrow \|H - H_0\|_F^2/\|U\|_F^2$
12: **until** $r < \epsilon$ and $s < \epsilon$
Performance opportunities

1. The factor matrices are tall-skinny (e.g., $10^6 \times 50$).
   - The ADMM step will be bound by memory bandwidth.
2. Real-world tensors have non-uniform distributions of non-zeros.
   - This may lead to non-uniform convergence of the factor rows during ADMM.
3. Many constraints and regularizations naturally invoke sparsity in the factors.
   - We can exploit this sparsity during MTTKRP (*in paper*).
**Blocked ADMM**

If the proximity operator coming from $r(\cdot)$ is row-separable, reformulate the ADMM problem to work on $B$ blocks of rows:

\[
\begin{align*}
\text{minimize} & \quad \sum_{b=1}^{B} \frac{1}{2} \left\| \left( X^{(1)} \right)_b - \tilde{A}_b^T (C \odot B)_b^T \right\|_F^2 + r(A_b) \\
\text{subject to} & \quad A_1 = \tilde{A}_1, \ldots, A_B = \tilde{A}_B.
\end{align*}
\]

Optimizing each block separately allows for them to converge at different rates, while acting as a form of cache tiling.
 Blocked ADMM

More simply:
Effects of block size

The block size affects both convergence rate and computational efficiency:

- A block size of 1 optimizes each row of $H$ independently.
- Larger block sizes better utilize hardware resources, but should be chosen to fit in cache.

Our evaluation uses $F=50$, and we experimentally found a block size of 50 rows to be a good balance between convergence rate and performance.
Table of Contents

Introduction

Accelerated AO-ADMM

Experiments

Conclusions
Experimental Setup

Source code:

- Modified from SPLATT
- Written in C and parallelized with OpenMP
- Compiled with icc v17.0.1 and linked with Intel MKL

Machine specifications:

- 2× 10-core Intel Xeon E5-2650v3 (Haswell)
- 396GB RAM

\(^1\)https://github.com/ShadenSmith/splatt
Convergence measurement

We measure convergence based on the relative reconstruction error:

\[
\text{relative error} = \frac{\mathcal{L}(X, A, B, C)}{\|X\|_F^2}.
\]

Termination:

- Convergence is detected when the relative error improves less than $10^{-6}$ or if we exceed 200 outer iterations.
- ADMM is limited to 50 iterations and $\epsilon = 10^{-2}$. 

Datasets

We selected four tensors from the FROSTT\(^2\) collection based on non-negative factorization performance:

- require a non-trivial number of iterations
- have a factorization quality that suggests a non-negative CPD is appropriate

<table>
<thead>
<tr>
<th>Dataset</th>
<th>NNZ</th>
<th>I</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reddit</td>
<td>95M</td>
<td>310K</td>
<td>6K</td>
<td>510K</td>
</tr>
<tr>
<td>NELL</td>
<td>143M</td>
<td>3M</td>
<td>2M</td>
<td>25M</td>
</tr>
<tr>
<td>Amazon</td>
<td>1.7B</td>
<td>5M</td>
<td>18M</td>
<td>2M</td>
</tr>
<tr>
<td>Patents</td>
<td>3.5B</td>
<td>46</td>
<td>240K</td>
<td>240K</td>
</tr>
</tbody>
</table>

\(^2\)http://frostit.io/
Relative Factorization Costs

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

![Bar chart showing fraction of factorization time spent on different datasets.](chart.png)
Parallel Scalability

Blocked ADMM improves speedup when the factorization is dominated by ADMM:

Baseline

Blocked
Convergence: Reddit

Blocking results in faster per-iteration runtimes and also converges in fewer iterations.
Convergence: NELL

Convergence is $3.7 \times$ faster with blocking, despite using additional iterations to achieve a lower error.
Convergence: Amazon

Both formulations exceed the maximum of 200 outer iterations, but the blocked formulation achieves a lower error in less time.
Convergence: Patents

Per-iteration runtimes are largely unaffected, as Patents is dominated by MTTKRP time. However, fewer iterations are required.
Table of Contents

Introduction

Accelerated AO-ADMM

Experiments

Conclusions
Wrapping up

Blocked ADMM accelerates constrained tensor factorization in two ways:

▶ Optimizing blocks independently saves computation on the “simple” rows and better optimizes “hard” rows.
▶ Blocks can be kept in cache during ADMM, saving memory bandwidth.

Also in the paper:

▶ MTTKRP can be accelerated by exploiting the sparsity that dynamically evolves in the factors.
▶ An additional $\sim 2\times$ speedup is achieved.

Future work:

▶ Analytical model for selecting block sizes.
▶ Automatic runtime selection of data structure for sparse factors.
Reproducibility

All of our work is open source (in the wip/ao-admm branch for now):

https://github.com/ShadenSmith/splatt

Datasets are freely available:

http://frostt.io/
Backup Slides
MTTKRP is a key kernel for computing the CPD:

\[ K = X_{(1)} (C \odot B) \]

### Notation
- \( X_{(1)} \) unfolds a tensor.
- \((C \odot B)\) is the Khatri-Rao (columnwise Kronecker) product.
Sparse MTTKRP

Convergence on Reddit with $F = 100$ and $r(\cdot) = 10^{-1}||\cdot||_1$. 

![Convergence graph](image_url)
Compressed sparse fiber (CSF)

- Modes are recursively compressed.
- Paths from roots to leaves encode non-zeros.
- The tree structure encodes opportunities for savings.
/* foreach outer slice */
for(int i=0; i < I; ++i) {
    /* foreach fiber in slice */
    for(int s = s_ptr[i]; s < s_ptr[i+1]; ++s) {
        accum[0:r] = 0;

        /* foreach nnz in fiber */
        for(int nnz = f_ptr[s]; nnz < f_ptr[s+1]; ++nnz) {
            int k = f_ids[nnz];
            accum[0:r] += vals[nnz] * C[k][0:r];
        }

        int j = s_ids[s];
        A[i][0:r] += accum[0:r] * B[s][0:r];
    }
}