High Performance Streaming Tensor Decomposition

Yongseok Soh
Computer & Info. Science
University of Oregon
Eugene, OR, USA
ysoh@uoregon.edu

Patrick Flick[†], Xing Liu[†],
Shaden Smith[†]
Google, Facebook, Microsoft
{patrick.flick, xing.research,
shadentsmith}@gmail.com

Fabio Checconi,
Fabrizio Petrini
Parallel Computing Lab, Intel
Santa Clara, CA, USA
{fabio.checconi,
fabrizio.petrini}@intel.com

Jee Choi
Computer & Info. Science
University of Oregon
Eugene, OR, USA
jeec@uoregon.edu

***Abstract*—We present a new algorithm for computing tensor decomposition on streaming data that achieves *up to 102× speedup* over the state-of-the-art *CP-stream* algorithm through *lower computational complexity* and *performance optimization*. For each streaming time slice, our algorithm partitions the factor matrix rows into those *with* and *without updates* and keeps them in *Gram matrix* form to significantly reduce the required computation. We also improve the scalability and performance of the matricized tensor times Khatri-Rao product (*MTTKRP)* kernel, a key performance bottleneck in many tensor decomposition algorithms, by reducing the synchronization overhead through the *combined use of mutex locks* and *thread-local memory*.**

For problems with constraints (e.g., non-negativity), we apply *data blocking* and *operation fusion* to the alternating direction method of multiplier (*ADMM*) kernel in the *constrained CP-stream* algorithm. By combining this *ADMM* optimization with the aforementioned *MTTKRP* optimization, our improved algorithm achieves up to 47× speedup over the original. We evaluate the performance and scalability of our new algorithm and optimization techniques using a 56-core quad-socket Intel Xeon system on four representative real-world tensors.

***Index Terms*—tensor decomposition, streaming, high performance, algorithm**

I. Introduction

Sparse tensor decomposition (TD) is a popular method for analyzing multi-way data in applications such as signal processing, topic monitoring, and trend analysis [1]. In many of these areas, data arrives in a streaming fashion over time (e.g., new updates on social media), and this poses two significant challenges in using traditional TD algorithms to analyze the data — the complete data is not available *a priori*, and the amount of accumulated data grows linearly with time. To address these challenges, a number of *streaming* TD algorithms have been proposed [2]–[5]. Among these, *CP-stream* [2] represents the state-of-the-art in terms of execution time, fitting error, and scalability on parallel systems. As such, we use *CP-stream* as the baseline for comparison against our work presented in this paper.

First, we analyzed the *constrained CP-stream* algorithm and its implementation to identify its key performance bottlenecks. We determined that (i) the alternating direction method of multiplier (*ADMM*) and matricized tensor times Khatri-Rao product (*MTTKRP*) kernels together can make up over 99.9% of the total execution time, and (ii) the time spent in one kernel may dominate the time spent in the other, depending on the number of iterations required for convergence for the *constrained CP-stream* algorithm and the *ADMM* kernel, which are in turn influenced by the property of the tensor being analyzed and the values used to initialize the factor matrices. Therefore, it is critical that we improve the *per iteration performance of both kernels*.

We applied the Roofline model [6] to calculate the arithmetic intensity for the different computational components of the *ADMM* kernel, and determined that every component is highly memory bandwidth-bound. To address this performance bottleneck, we *block the matrices* to increase its reuse in cache, and *fuse the computational components* to reduce the memory traffic that comes from accessing large intermediate data structures and to reduce the total number of operations.

The *MTTKRP* kernel suffers from scalability issues due to *thread contention* when different threads update the same factor matrix row. We use a *combination of mutex locks and thread-local memory* to minimize thread contention. For factor matrices with only a few rows, where the probability of contention is high, we use a thread-local copy of the matrix to make local updates, and then reduce the results at the end. We use mutex locks for the remaining modes.

During our study, we discovered that certain real-world tensors have skewed distribution of non-zero elements across different time slices, resulting in these elements updating only a *small subset* of factor matrix rows. To address this issue, we partition the factor matrices into two subsets — one with rows that are updated during *MTTKRP* and those that are not. This resulted in a *new algorithmic formulation* of *non-constrained CP-stream* that greatly reduces the number of required operations by maintaining the factor matrices in *Gram matrix form*, and computing over these smaller matrices.

This paper makes *two key contributions* to improving the performance of streaming tensor decomposition:

- *Optimized constrained CP-stream*, where we use *Blocked & Fused ADMM* and *Hybrid Lock MTTKRP* kernels to improve the performance of the original *constrained CP-stream* algorithm. We achieve up to 47× speedup on our 56-core testbed system on four real-world tensors.
- *A new algorithm* for *non-constrained CP-stream*, where we partition the factor matrix into two non-overlapping subsets of rows and update them independently. These subsets are stored and computed in Gram matrix form, which greatly reduces the required computation. Our new algorithm achieves up to 102× over the original *non-constrained CP-stream* algorithm.

[†] These authors were employed by Intel when this research was conducted.

II. Related Work

Streaming tensor decomposition is analogous to streaming matrix factorization, which have application in recommender systems [7], [8], dictionary learning [9], [10], and subspace tracking [11]. Some have similar features, such as forgetfulness [7] to control the amount of past information to retain. However, these algorithms are limited to two-way data.

Computing the canonical polyadic decomposition for streaming tensors was first studied in signal processing [12], using the proposed *PARAFAC-SDT* and *PARAFAC-RLST* algorithms. However, these algorithms make inefficient use of memory, limiting their use to small tensors. Mardani et al. proposed a more memory-efficient algorithm called *Online-SGD* [5] which uses the popular stochastic gradient descent (SGD) method to update the non-streaming factor matrices. This algorithm suffers from the same problem that SGD suffers from — finding the optimal learning rate is non-trivial. *Online-CP* [4] is another method for decomposing streaming tensors. However, this algorithm has not been adapted to handle sparse tensors, making it unsuitable for decomposing many real-world tensors, such as those found on the FROSTT tensor repository [13].

This is the first study on systematically analyzing and optimizing the performance bottlenecks of *streaming* tensor decomposition algorithms, to the best of our knowledge. There have been numerous studies [14]–[16] on optimizing sparse tensor decomposition algorithms in recent years. However, these studies focus on improving data access through storage formats and are limited to *non-streaming* algorithms.

III. Background

In this section, we provide a brief overview of *CP-stream* and basic tensor notations. For a more in-depth discussion of tensor algorithms and its applications, we direct the readers to the survey by Kolda and Bader [17].

A. Tensor notations

Tensors are higher-order generalization of matrices. An N-dimensional tensor is referred to as having N *modes* or a *mode-N* tensor. The following notations are used in this paper.

1) *Scalars* are denoted by lower case letters (e. g., a).
2) *Vectors* are denoted by bold lower case letters (e. g., $\mathbf{a}$).
3) *Matrices* are denoted by bold capital letters (e. g., $\mathbf{A}$). A $I_1 \times I_2$ matrix $\mathbf{A}$ can be denoted as $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2}$.
4) *Higher-order tensors* are denoted by bold calligraphic letters (e. g., $\boldsymbol{\mathcal{X}}$). A mode-N tensor $\boldsymbol{\mathcal{X}}$ with dimensions $I_1 \times I_2 \times \cdots \times I_N$ can be denoted as $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$.
5) *Matricization* is the process of reordering the elements of a tensor into a matrix. Mode-n matricization of a tensor $\boldsymbol{\mathcal{X}}$, denoted as $\mathbf{X}_{(n)}$, is a $I_n \times \hat{I}_n$ matrix, where $\hat{I}_n = \prod_{i \neq n} I_n$.
6) *Slices* are sub-tensors that are formed by fixing one particular index. For example, we can form a time slice (i.e., mode-N-1 subtensor) by fixing the time index to a specific value (e.g., 0 for the first time slice) for a mode-N tensor.
7) *Hadamard product*, or the element-wise product between two matrices, is denoted by $\circledast$.
8) *Khatri-Rao product*, or the column-wise Kronecker product between two matrices, is denoted by $\odot$. Khatri-Rao product between two matrices $\mathbf{A}\in \mathbb{R}^{I_1 \times K}$ and $\mathbf{B}\in \mathbb{R}^{I_2 \times K}$ yields the matrix $\mathbf{C}\in \mathbb{R}^{I_1 I_2 \times K}$

B. Canonical Polyadic Decomposition

Canonical polyadic decomposition (CPD) is one of the two most popular TD algorithms in use today, and it *approximates* a mode-N tensor as a summation of K rank-1 tensors, where each rank-1 tensor is formed by the outer product of N vectors. K is referred to as the *rank* of the decomposition, and is typically a small integer value on the order of 10 or 100. TD is a low-rank approximation method analogous to the truncated singular value decomposition (SVD) for matrices.

The K vectors for each mode (that are used to form the rank-1 tensors) make up the columns of the *factor matrices*, with one factor matrix for each mode (i.e., N factor matrices). The nth mode factor matrix is denoted by $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K}$. CPD can be represented by the following optimization problem:

$$\underset{\{\mathbf{A}^{(n)}\}}{\text{minimize}} \; \frac{1}{2} \left\| \mathbf{X}_{(n)} - \mathbf{A}^{(n)} \left(\underset{v \neq n}{\odot} \mathbf{A}^{(v)} \right)^T \right\|^2$$

C. CP-stream

The tensor decomposition problem in a streaming setting can be defined as finding the rank-K CPD of an $(N+1)$-way tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times T}$, in which N-way tensors arrive over time in T batches. The time T can be potentially unbounded (i.e., $T \to \infty$). Finding the solution to this problem can be described by the following optimization problem:

$$\underset{\{\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K}\},\ \mathbf{S} \in \mathbb{R}^{T \times K}}{\text{minimize}} \quad \frac{1}{2} \left\| \boldsymbol{\mathcal{Y}} - [\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}, \mathbf{S}]\!] \right\|^2 \tag{1}$$

where $\{\mathbf{A}^{(n)}\}$ are the factor matrices for the N-way tensors that are streamed in, and $\mathbf{S}$ is the factor matrix that incorporates the temporal information. The term $[\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}, \mathbf{S}]\!]$ is an abbreviation of the outer-product formulation for approximating a tensor using its factor matrices, as described above.

$\boldsymbol{\mathcal{Y}}$ can equivalently be modeled as a *sequence* of N-way tensors, $\boldsymbol{\mathcal{X}}_1, ..., \boldsymbol{\mathcal{X}}_T$, with *each* $\boldsymbol{\mathcal{X}}_t$ modeled by $[\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}; \mathbf{s}_t]\!]$, where $\mathbf{s}_t \in \mathbb{R}^K$. This leads to the following optimization problem:

$$\underset{\{\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K}\},\ \{\mathbf{s}_t \in \mathbb{R}^K\}}{\text{minimize}} \quad \sum_{t=1}^{T} \frac{1}{2} \left\| \boldsymbol{\mathcal{X}}_t - [\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}; \mathbf{s}_t]\!] \right\|^2 \tag{2}$$

With this formulation, an algorithm that estimates the solution by considering *one tensor sample $\boldsymbol{\mathcal{X}}_t$ at a time* can be defined. We first solve for $\mathbf{s}_t$ which has a closed-form solution that depends only on factor matrices from the previous time slice. Once we obtain $\mathbf{s}_t$, we then update the factor matrices for the current time slice. Instead of taking into account all historical data $\boldsymbol{\mathcal{X}}_1, \ldots, \boldsymbol{\mathcal{X}}_t$, *CP-stream* adopts the approach by Vandescapelle et al. [18], and minimizes an *approximate* *loss* by replacing $\boldsymbol{\mathcal{X}}_1, ..., \boldsymbol{\mathcal{X}}_t$ with the existing factorization $[\![\mathbf{A}_{t-1}^{(1)}, \ldots, \mathbf{A}_{t-1}^{(N)}, \mathbf{S}_{t-1}]\!]$, where $\mathbf{S}_{t-1} \in \mathbb{R}^{t-1 \times K}$ is the matrix with rows $s_1, \ldots, s_{t-1}$.

The reformulated objective can be written as follows:

$$\underset{\mathbf{A}^{(n)} \in \mathcal{C}}{\text{minimize}} \quad \frac{1}{2} \left\| \boldsymbol{\mathcal{X}}_t - [\![\mathbf{A}_t^{(1)}, \ldots, \mathbf{A}_t^{(N)}; \mathbf{s}_t]\!] \right\|^2 + \sum_{i=1}^{t-1} \frac{\mu^{t-i}}{2} \left\| [\![\mathbf{A}_{t-1}^{(1)}, \ldots, \mathbf{A}_{t-1}^{(N)}; \mathbf{s}_i]\!] - [\![\mathbf{A}_t^{(1)}, \ldots, \mathbf{A}_t^{(N)}; \mathbf{s}_t]\!] \right\|^2 \tag{3}$$

where $\mu \in [0, 1]$ is the forgetting factor that downweighs the importance of historical data. For problems that require constraints, constraint $\mathcal{C}$ can be applied to the factor matrices $\mathbf{A}^{(n)}$. Since the objective still requires all historical s_i, the *CP-stream* algorithm further relaxes this requirement by introducing a Gram matrix containing all previous temporal information, $\mathbf{G}_{t-1}$. The overall *CP-stream* algorithm is described in Algorithm 1 and we direct interested readers to [2] for more extensive details.

Algorithm 1: CP-stream

```
   Input: $\mathbf{A}, \mathbf{\Phi}, \mathbf{\Psi}$
 1 for $t$ = 1, . . ., $T$ do
 2     $s_t \leftarrow$ least-squares update
 3     repeat
 4         for $n$ = 1, . . ., $N$ do
 5             $\mathbf{\Phi}^{(n)} \leftarrow \left( \underset{v \neq n}{\overset{N}{\circledast}} \mathbf{A}^{(v)\top} \mathbf{A}^{(v)} \right) \circledast \left( \mu \mathbf{G}_{t-1} + s_t s_t^\top \right)$
 6             $\mathbf{\Psi}^{(n)} \leftarrow MTTKRP \left( \boldsymbol{\mathcal{X}}_t, \{\mathbf{A}^{(v)}\}, n \right) +$
                 $\mathbf{A}_{t-1}^{(n)} \left( \left( \underset{v \neq n}{\overset{N}{\circledast}} \mathbf{A}_{t-1}^{(v)\top} \mathbf{A}^{(v)} \right) \circledast \mu \mathbf{G}_{t-1} \right)$
 7             $\mathbf{A}^{(n)} \leftarrow \mathbf{\Psi}^{(n)} \left( \mathbf{\Phi}^{(n)} \right)^{-1}$
 8         end
 9     until convergence
10 end
```

If we wish to apply constraints to Algorithm 1, the ADMM kernel can be used in line 7 instead to apply constraints to the factor matrices, such as non-negativity or sparsity. As this does not have a closed-form solution, ADMM approximates the solution in an iterative manner [19]. The inner workings of ADMM is further explained and analyzed in Section IV-A.

IV. Performance Analysis and Optimization

In this section, we present our performance analysis of the *constrained CP-stream* algorithm, and propose two optimizations that significantly improve the performance of the *ADMM* and *MTTKRP* kernels.

A. Blocked and Fused ADMM

In the original algorithm, the authors use the alternating direction method of multiplier (*ADMM*) [20] to obtain an approximate solution to the constrained least squares problem. The *ADMM* algorithm is shown in Algorithm 2, and the associated computational and memory access cost for each operation are shown in Table I. Notice that the expressions $(\mathbf{\Phi} + \rho \mathbf{I})$ does not change within *ADMM*, and therefore is pre-computed outside and not counted (i.e., only the *Cholesky* operation for the *inverse* is counted). The code is parallelized using *OpenMP* for most matrix operations in a fine-grained manner (i.e., one thread is assigned to each element), and parallelized over the columns for the column-wise norm calculation.

Algorithm 2: ADMM Algorithm

```
Input: $\mathbf{A}, \mathbf{\Phi}, \mathbf{\Psi}$
$\mathbf{U} \leftarrow 0$
$\rho = trace(\mathbf{\Phi})/K$
repeat
    $\mathbf{A}_0 \leftarrow \mathbf{A}$                                                    ◁ init
    $\tilde{\mathbf{A}}^T \leftarrow (\mathbf{\Phi} + \rho \mathbf{I})^{-1} (\mathbf{\Psi} + \rho (\mathbf{A} + \mathbf{U}))^T$          ◁ solve
    $\mathbf{A} \leftarrow Proj_C \left( \tilde{\mathbf{A}} - \mathbf{U} \right)$                                  ◁ project
    $\mathbf{U} \leftarrow \mathbf{U} + \mathbf{A} - \tilde{\mathbf{A}}$                                        ◁ update
until $\left\| A - \tilde{A} \right\|_F^2 / \|A\|_F^2 < \epsilon$ and $\|A - A_0\|_F^2 / \|U\|_F^2 < \epsilon$
                                                                       ◁ error
```

As it can be seen from Table I, the arithmetic intensity of most operations in *ADMM* are < 0.125 (assuming double-precision 8-byte word), making them highly memory bandwidth-bound, according to the Roofline model [6]. Additionally, many of the operations in Algorithm 2 are either *row-wise* or *element-wise* independent. The only exception is the *project* operation which involves column-wise norm calculation.

TABLE I: Compute and memory costs for different ADMM operations.

Operation	Compute (flops)	Memory (words) $(Read) + (Write)$
init	0	(IK) + (IK)
solve	$3IK + 2IK^2$	$(4IK + K^2)$ + $(2IK)$
project	$3IK + IK$	$(4IK)$ + $(2IK)$
update	$2IK$	$(3IK)$ + (IK)
error	$10IK$	$(4IK)$ + (0)
Total	$19IK + 2IK^2$	$(16IK + K^2)$ + $(6IK)$

Therefore, we apply the following optimizations to reduce memory traffic: (i) *data blocking*: divide the matrices into *blocks* of rows, and then parallelize over these blocks using *OpenMP* and vectorize the computation within each block, (ii) *operation fusion*: *fuse* matrix operations using registers to eliminate the *load* and *store* of intermediate results, and (iii) use parallel reduction to calculate the column-wise *norm* across the blocks of rows. Our optimized ADMM algorithm is shown in Algorithm 3.

In this optimized version, we use *OpenMP* to parallelize over the row-blocks (i.e., one *OpenMP* thread is assigned to one block) in the *for* loops on line 5 and line 12. Computation inside the *for* loop on line 14 are element-wise independent, so we vectorize the loop to take advantage of data level parallelism (DLP) and to improve the data access efficiency (i.e., fewer *load* instructions). The sequence of computation inside the inner-most loop (i.e., the *init* → *solve* → *project* → *update* in the *repeat* loop in Algorithm 2) was rearranged

Algorithm 3: Optimized ADMM Algorithm

```
   Input: $\mathbf{A}, \mathbf{\Phi}, \mathbf{\Psi}$
1  $\mathbf{U} \leftarrow 0$
2  $\rho = trace(\mathbf{\Phi})/K$
3  $pr, pn, dr, dn \leftarrow 0$                          ◁ used for checking convergence
4  $\mathbf{A}_0 \leftarrow \mathbf{A}$                                                   ◁ init
5  for each row-block B do
6      $\tilde{\mathbf{A}}_B^T \leftarrow (\mathbf{\Phi}_B + \rho \mathbf{I}_B)^{-1} \ (\mathbf{\Psi}_B + \rho\,(\mathbf{A}_{0,B} + \mathbf{U}_B))^T$   ◁ solve
7      $\mathbf{A}_B \leftarrow \tilde{\mathbf{A}}_B - \mathbf{U}_B$
8      $\mathbf{C}_{thr} \leftarrow$ accum_col_norm($\mathbf{A}_B$)                ◁ per-thread
9  end
10 $\mathbf{C}_G \leftarrow$ all-reduce($\mathbf{C}_{thr}$)
11 repeat
12     for each row-block B do
13         $\mathbf{A}_B = Proj_C\,(\mathbf{A}_B, \mathbf{C}_G)$                     ◁ project
14         for each element E in row-block B do
15             register $x \leftarrow \mathbf{A}_B[E]$
16             register $y \leftarrow x - \tilde{\mathbf{A}}_B[E]$
17             register $di \leftarrow \mathbf{U}_B[E] + y$
18             $\mathbf{U}_B[E] \leftarrow di$                                 ◁ update
19             $pr \leftarrow pr + y * y$                     ◁ $\left\|A - \tilde{A}\right\|_F^2$
20             $pn \leftarrow pn + x * x$                          ◁ $\|A\|_F^2$
21             register $p \leftarrow x - \mathbf{A}_{0,B}[E]$
22             $dr \leftarrow dr + p * p$                     ◁ $\|A - A_0\|_F^2$
23             $dn \leftarrow dn + di * di$                        ◁ $\|U\|_F^2$
24             $\mathbf{A}_{0,B}[E] \leftarrow x$                              ◁ init
25             $\tilde{\mathbf{A}}_B[E] \leftarrow (\mathbf{\Psi}_B[E] + \rho\,(x + di))$
26         end
27         $\tilde{\mathbf{A}}_B^T \leftarrow (\mathbf{\Phi}_B + \rho \mathbf{I}_B)^{-1}\, \tilde{\mathbf{A}}_B^T$               ◁ solve
28         $\mathbf{A}_B \leftarrow \tilde{\mathbf{A}}_B - \mathbf{U}_B$
29         $\mathbf{C}_{thr} \leftarrow$ accum_col_norm($\mathbf{A}_B$)
30     end
31     $\mathbf{C}_G \leftarrow$ all-reduce($\mathbf{C}_{thr}$)
32     reduce($pr, pn, dr, dn$)                       ◁ OpenMP reduction
33 until $pr/pn < \epsilon$ and $dr/dn < \epsilon$
```

to bring the *update*, *error*, *init*, and *solve* operations closer together so that registers can be used to hold the intermediate calculations rather than storing them in memory. Note that the *error* operation is interleaved with the other operations inside the inner loop.

This reduces the overall computational cost to $18IK + 2IK^2$ flops (vs. $19IK + 2IK^2$ flops), and data access to $15IK + K^2$ bytes (vs. $22IK + K^2$ bytes), which is more than a 30% reduction in data access. This is **not** counting the reduction in memory traffic due to blocking the matrices, which yields even more benefit.

Note that lines 7 and 8 (and lines 28 and 29) are also fused such that the each element of $\tilde{\mathbf{A}} - \mathbf{U}$ is stored in a register and used to calculate the norm (but we do not show this in Algorithm 3 to keep it concise). Also, we ignore the memory traffic associated with the per-thread array used to store the norms, as they generally tend to be very small in comparison to the matrices.

B. Hybrid Lock Mechanism

The matricized tensor time Khatri-Rao product (*MTTKRP*) is used to calculate the $\mathbf{\Psi}$ used in the ADMM kernel, and is another major performance bottleneck in the *CP-stream* algorithm (both *constrained* and *non-constrained*). In the baseline *CP-stream* implementation, *MTTKRP* is parallelized using *OpenMP* in a fine-grained manner, where each thread is assigned to a set of non-zero elements in the sparse tensor, and each element updates a factor matrix row. While this allows near-perfect load balancing, it introduces a race condition when more than one thread needs to update the same row in the factor matrix. To overcome this issue, the baseline implementation utilizes a pool of mutual exclusion (mutex) locks to serialize the updates.

This method works well when the number of threads is *much* smaller than the number of rows in the matrix (i.e., tensors with large mode lengths), as the probability of two threads updating the same row at the same time is relatively low. However, when one mode has a particularly small length (e.g., < 100), this method leads to extreme performance degradation as threads compete for locks.

One prime example of this is in the *streaming (time)* mode — since the *CP-stream* algorithm processes a single time slice at a time, the factor matrix for the streaming mode always has only a *single* row at any given time. This means that every thread will be attempting to update the same row, causing the updates to be not only completely serialized, but also adds a penalty from using a lock.

To mitigate this bottleneck, we use *thread-local memory* to accumulate the updates, and then reduce them after every thread has completed processing its assigned non-zero elements. This allows every thread to run completely independently until the end, at a small cost of an extra reduction operation at the end.

V. spCP-stream - a New Algorithm for CP-stream

In this section, we provide a detailed description for *Sparse CP-stream* (*spCP-stream*) — our new algorithm for calculating streaming tensor decomposition.

A. Motivation

During our evaluation, we observed that for certain tensors, the non-zero elements were clustered around specific index values for some modes. For example, the *Flickr* dataset [13] has $113M$ non-zero elements and the length of mode 2 is $28M$. This translates to $113M$ *tags* being made on *some* of the $28M$ total available *images*. Due to the streaming nature of this data, only a small subset of images are being tagged at certain times (e.g., images are never tagged again after the initial tag and upload). This means that for every time slice, the mode 2 index values for every non-zero elements can be limited to a small subset of $28M$ possible index values. This is illustrated in Figure 1.

To make data access more efficient, we created a new data structure for storing the factor matrices, where we divide the factor matrices into two non-overlapping subsets of rows — one subset with rows that are updated, and the other with rows that are never updated during *MTTKRP* — and update the two subsets individually as required. Propagating this data structure

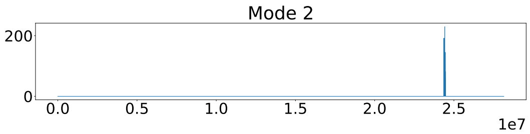


Fig. 1: Histogram of non-zero elements for mode-2 of the *Flickr* dataset at time slice 500. All of the non-zero elements occupy a small subset of index values. In contrast, the elements are more evenly distributed among all index values for all other modes.

to the rest of the algorithm led to a new algorithm that greatly reduces the computational complexity of the *overall algorithm*.

B. spCP-stream Notations

Here are additional notations specific to *spCP-stream*:

1) For a given time slice $\boldsymbol{\mathcal{X}}_t$, the set of mode n index values for every non-zero element is denoted by $\mathbf{nz}^{(n)}$.
2) We use $\boldsymbol{A}_{nz}^{(n)}$ to denote the matrix formed by "gathering" the rows of the factor matrix $\boldsymbol{A}^{(n)}$ if the row index is in $\mathbf{nz}^{(n)}$. We also denote this by $\boldsymbol{A}_{nz}^{(n)} \leftarrow \boldsymbol{A}^{(n)}\left[\mathbf{nz}^{(n)}\right]$.
3) We use $\boldsymbol{A}_{z}^{(n)}$ to denote the matrix formed by taking all remaining rows of $\boldsymbol{A}^{(n)}$. That is, $\boldsymbol{A}_{z}^{(n)} \leftarrow \boldsymbol{A}^{(n)}\left[\mathbf{z}^{(n)}\right]$, where $Z \leftarrow \{1, \ldots, I_n\}$, and $\mathbf{z}^{(n)} \leftarrow Z \setminus \mathbf{nz}^{(n)}$.
4) We use $\oplus$ to denote forming a new matrix by "gathering" the rows from $\boldsymbol{A}_{z}^{(n)}$ and $\boldsymbol{A}_{nz}^{(n)}$. That is, $\boldsymbol{A}_{nz}^{(n)} \oplus \boldsymbol{A}_{z}^{(n)} \equiv \boldsymbol{A}^{(n)}\left[\mathbf{nz}^{(n)} \cup \mathbf{z}^{(n)}\right]$.
5) *Sparse MTTKRP* (*spMTTKRP*) is our new *MTTKRP* implementation that operates *only* over the non-zero rows (i.e., over $\boldsymbol{A}_{nz}^{(n)}$)

C. spCP-stream Algorithm

To fully utilize the benefit of separating the factor matrix into two subsets of non-zero rows (i.e., $\boldsymbol{A}_{z}^{(n)}$ and $\boldsymbol{A}_{nz}^{(n)}$), we need to formulate an *end-to-end computation scheme* that operates over these matrices.

If we collapse the inner-most loop in Algorithm 1 (i.e., combine lines 5 through 7), then we have

$$\begin{aligned}\mathbf{A}^{(n)} = \Big(&MTTKRP\left(\boldsymbol{\mathcal{X}}_t, \{\mathbf{A}^{(v)}\}\right) + \\ &\mathbf{A}_{t-1}^{(n)}\left(\left(\underset{v\neq n}{\circledast}\mathbf{A}_{t-1}^{(v)T}\mathbf{A}^{(v)}\right)\circledast \mu\mathbf{G}_{t-1}\right)\Big)\cdot \\ &\left(\left(\underset{v\neq n}{\overset{N}{\circledast}}\mathbf{A}^{(v)T}\mathbf{A}^{(v)}\right)\circledast\left(\mu\mathbf{G}_{t-1}+s_t s_t^T\right)\right)^{-1}\end{aligned} \tag{4}$$

We now separate this out to two parts — one for computing $\boldsymbol{A}_{nz}^{(n)}$ and the other for computing $\boldsymbol{A}_{z}^{(n)}$. For $\boldsymbol{A}_{z}^{(n)}$, *MTTKRP* makes no contribution, and thus can be eliminated:

$$\begin{aligned}\mathbf{A}_z^{(n)} = \mathbf{A}_{z,t-1}^{(n)}&\underbrace{\left(\left(\underset{v\neq n}{\circledast}\mathbf{A}_{t-1}^{(v)T}\mathbf{A}^{(v)}\right)\circledast\mu\mathbf{G}_{t-1}\right)}_{\mathbf{Q}^{(n)}}\cdot \\ &\underbrace{\left(\left(\underset{v\neq n}{\overset{N}{\circledast}}\mathbf{A}^{(v)T}\mathbf{A}^{(v)}\right)\circledast\left(\mu\mathbf{G}_{t-1}+s_t s_t^T\right)\right)^{-1}}_{\left(\mathbf{\Phi}^{(n)}\right)^{-1}}\end{aligned} \tag{5}$$

Simplifying this equation, we end up with:

$$\mathbf{A}_z^{(n)} = \mathbf{A}_{z,t-1}^{(n)}\mathbf{Q}^{(n)}\left(\mathbf{\Phi}^{(n)}\right)^{-1} \tag{6}$$

As for $\boldsymbol{A}_{nz}^{(n)}$, we can equivalently simplify the equation as follows:

$$\mathbf{A}_{nz}^{(n)} = \left(spMTTKRP\left(\boldsymbol{\mathcal{X}}_t, \{\mathbf{A}_{nz}^{(v)}\}\right) + \mathbf{A}_{nz,t-1}^{(n)}\mathbf{Q}^{(n)}\right)\left(\mathbf{\Phi}^{(n)}\right)^{-1} \tag{7}$$

where *spMTTKRP* is our new $MTTKRP$ implementation that operates over only the non-zero rows. Notice that both $\mathbf{Q}^{(n)}$ and $\mathbf{\Phi}^{(n)}$ are small $K \times K$ Gram matrices, and $\mathbf{\Phi}^{(n)}$ is positive definite.

Also, we define:

$$\mathbf{C}^{(n)} = \mathbf{A}^{(n)T}\mathbf{A}^{(n)} \tag{8}$$

$$\mathbf{H}^{(n)} = \mathbf{A}_{t-1}^{(n)T}\mathbf{A}^{(n)} \tag{9}$$

$\mathbf{C}^{(n)}$ and $\mathbf{H}^{(n)}$ are also $K \times K$ Gram matrices, and can be written as the sum of the Gram matrices of $\boldsymbol{A}_{z}^{(n)}$ and $\boldsymbol{A}_{nz}^{(n)}$. For example,

$$\begin{aligned}\mathbf{C}^{(n)} &= \mathbf{A}^{(n)T}\mathbf{A}^{(n)} \\ &= \underbrace{\mathbf{A}_z^{(n)T}\mathbf{A}_z^{(n)}}_{\mathbf{C}_z^{(n)}} + \underbrace{\mathbf{A}_{nz}^{(n)T}\mathbf{A}_{nz}^{(n)}}_{\mathbf{C}_{nz}^{(n)}}\end{aligned} \tag{10}$$

$H^{(n)}$ decomposes in the same way.

Notice that from Equation 6, we have

$$\begin{aligned}\mathbf{A}_z^{(n)T}\mathbf{A}_z^{(n)} &= \left(\mathbf{A}_{z,t-1}^{(n)}\mathbf{Q}^{(n)}\left(\mathbf{\Phi}^{(n)}\right)^{-1}\right)^T\mathbf{A}_{z,t-1}^{(n)}\mathbf{Q}^{(n)}\left(\mathbf{\Phi}^{(n)}\right)^{-1} \\ &= \left(\mathbf{\Phi}^{(n)}\right)^{-T}\mathbf{Q}^{(n)T}\mathbf{A}_{z,t-1}^{(n)T}\mathbf{A}_{z,t-1}^{(n)}\mathbf{Q}^{(n)}\left(\mathbf{\Phi}^{(n)}\right)^{-1} \\ &= \left(\mathbf{\Phi}^{(n)}\right)^{-T}\mathbf{Q}^{(n)T}\mathbf{C}_{z,t-1}^{(n)}\mathbf{Q}^{(n)}\left(\mathbf{\Phi}^{(n)}\right)^{-1}\end{aligned} \tag{11}$$

Putting it all together, we can calculate $\mathbf{C}^{(n)}$ (Equation 10) by adding the results of Equation 11 to the Gram matrix of the result of Equation 7 (i.e., $\mathbf{A}_{nz}^{(n)T}\mathbf{A}_{nz}^{(n)}$).

Similarly, we can calculate $\mathbf{H}^{(n)}$ by calculating

$$\mathbf{H}_{nz}^{(n)} = \underbrace{\mathbf{A}_{nz,t-1}^{(n)T}}_{\text{from previous time slice}}\underbrace{\mathbf{A}_{nz}^{(n)}}_{\text{Equation 7}} \tag{12}$$

$$
\begin{aligned}
\mathbf{H}_z^{(n)} &= \mathbf{A}_{z,t-1}^{(n)T}\mathbf{A}_z^{(n)} \\
&= \mathbf{A}_{z,t-1}^{(n)T}\underbrace{\mathbf{A}_{z,t-1}^{(n)}\mathbf{Q}^{(n)}\left(\mathbf{\Phi}^{(n)}\right)^{-1}}_{\text{Equation } 6} \\
&= \mathbf{C}_{z,t-1}^{(n)}\mathbf{Q}^{(n)}\left(\mathbf{\Phi}^{(n)}\right)^{-1}
\end{aligned} \tag{13}
$$

Finally, the calculation for $\mathbf{Q}^{(n)}$ and $\mathbf{\Phi}^{(n)}$ reduces to:

$$
\begin{aligned}
\mathbf{Q}^{(n)} &= \left(\underset{v\neq n}{\circledast}\mathbf{A}_{t-1}^{(v)T}\mathbf{A}^{(v)}\right)\circledast\mu\mathbf{G}_{t-1} \\
&= \left(\underset{v\neq n}{\circledast}\mathbf{H}^{(v)}\right)\circledast\mu\mathbf{G}_{t-1} \\
\mathbf{\Phi}^{(n)} &= \left(\overset{N}{\underset{v\neq n}{\circledast}}\mathbf{A}^{(v)T}\mathbf{A}^{(v)}\right)\circledast\left(\mu\mathbf{G}_{t-1}+s_t s_t^T\right) \\
&= \left(\overset{N}{\underset{v\neq n}{\circledast}}\mathbf{C}^{(v)}\right)\circledast\left(\mu\mathbf{G}_{t-1}+s_t s_t^T\right)
\end{aligned} \tag{14}
$$

In conclusion, by maintaining and updating the small $K\times K$ Gram matrices $\mathbf{C}^{(n)}$ and $\mathbf{H}^{(n)}$ in the inner loop, we can reduce the amount of computation required for calculating $\mathbf{Q}^{(n)}$, $\mathbf{\Phi}^{(n)}$. The overall algorithm is shown in Algorithm 4.

D. Pre- and Post- Operations

To take advantage of the new formulation described in the previous subsection, some additional *housekeeping* operations are required. For instance, when a new time slice arrives, it is necessary to identify the non-zero rows for every mode in advance. Also, because the inner iteration relies on the previous time slice's factor matrices, $\mathbf{C}_{z,t-1}^{(n)}$ needs to be computed prior to the inner iteration.

Once the inner iteration reaches convergence for all modes, we also need to update the full factor matrices $\boldsymbol{A}^{(n)}$ based on the final $\boldsymbol{A}_{nz}^{(n)}$ and $\boldsymbol{A}_{z}^{(n)}$. However, these newly introduced operations have minimal impact on the overall execution time, as their overhead can be amortized by the inner iterations.

E. Convergence Check

At the end of each iteration, the normalized absolute difference between $\mathbf{A}^{(n)}$ and $\mathbf{A}_{t-1}^{(n)}$ needs to be calculated to check for convergence. Since *spCP-stream* eliminates the need to construct full factor matrices, we also propose a way to check for convergence using only the Gram matrices $\mathbf{C}^{(n)}$ and $\mathbf{H}^{(n)}$.

The termination criteria at time step t is determined by calculating

$$
\delta_t = \sum_{n=1}^{N}\frac{\left\|\mathbf{A}^{(n)}-\mathbf{A}_{t-1}^{(n)}\right\|_F}{\left\|\mathbf{A}^{(n)}\right\|_F} \tag{15}
$$

and terminating if $|\delta_t-\delta_{t-1}|<\epsilon$ where ϵ is some tolerance.

Using basic linear algebra properties, we can compute the denominator of Equation 15 using $\mathbf{C}^{(n)}$ as follows:

$$
\left\|\mathbf{A}^{(n)}\right\|_F^2 = tr\left(\mathbf{A}^{(n)T}\mathbf{A}^{(n)}\right) = tr(\mathbf{C}^{(n)}) \tag{16}
$$

Similarly, the numerator of Equation 15 can be computed as follows:

$$
\begin{aligned}
\left\|\mathbf{A}^{(n)}-\mathbf{A}_{t-1}^{(n)}\right\|_F^2 &= tr\left(\left(\mathbf{A}^{(n)}-\mathbf{A}_{t-1}^{(n)}\right)^T\left(\mathbf{A}^{(n)}-\mathbf{A}_{t-1}^{(n)}\right)\right) \\
&= tr\left(\mathbf{A}^{(n)T}\mathbf{A}^{(n)}-\mathbf{A}^{(n)T}\mathbf{A}_{t-1}^{(n)}\right. \\
&\qquad \left. -\mathbf{A}_{t-1}^{(n)T}\mathbf{A}^{(n)}+\mathbf{A}_{t-1}^{(n)T}\mathbf{A}_{t-1}^{(n)}\right) \\
&= tr\left(\mathbf{C}^{(n)}-\mathbf{H}^{(n)T}-\mathbf{H}^{(n)}+\mathbf{C}_{t-1}^{(n)}\right) \\
&= tr\left(\mathbf{C}^{(n)}\right)+tr\left(\mathbf{C}_{t-1}^{(n)}\right)-2\,tr\left(\mathbf{H}^{(n)}\right)
\end{aligned} \tag{17}
$$

This yields an equivalent convergence check rule using only the diagonal elements (i.e., *trace*) of $\mathbf{C}^{(n)}$ and $\mathbf{H}^{(n)}$.

VI. Performance Evaluation

A. System

For our evaluation, we use a large-memory node from the University of Oregon's *Talapas* high-performance computing (HPC) cluster. This is a quad-socket Intel E7-4830v4 system with a total of 56 cores and 512GB of main memory.

B. Datasets and Execution Parameters

Datasets used for our evaluation are shown in Table II. We selected these datasets from the FROSTT tensor repository [13] to have varying number non-zero elements, number of modes, and dimension sizes.

For each dataset, we used a forgetting factor of 0.99 and convergence tolerances of 10^{-5} (*Patents* and *Flickr*) and 10^{-6} (*Uber* and *NIPS*). Additionally, we used a Frobenius norm regularization of 10^{-2} on the streaming mode for stability. We used tensor decomposition ranks of {16, 32, 64, 128} and number of *OpenMP* threads of {1, 7, 14, 28, 56}. Lastly, for each decomposition run, random values were used to initialize the factor matrices.

C. Execution Time, Fit Error, and Convergence Property

The execution times presented in the rest of this section are *per iteration time*, as the number of iterations to convergence for each experimental run vary due to random initialization. We also use the *minimum* per iteration execution time from 10 trials, as the *thread contention* in *MTTKRP* for the baseline implementation causes a large standard deviation in its execution time. However, the minimum is a more *conservative* measure of the speedup achieved by our optimized kernels and algorithm — if we use the average or the median, the speedup achieved by our work is *significantly* higher across all datasets, as our work shows more consistent execution times.

Our work demonstrates similar fit error and convergence properties as the original *CP-stream* algorithm. We omit them here due to page limit.

D. Constrained CP-stream

In this subsection, we illustrate and analyze the performance of our Blocked & Fused *ADMM* and Hybrid Lock *MTTKRP* kernels, and our optimized *CP-stream* implementations.

```
Algorithm 4: New Algorithmic Formulation
Input: $\boldsymbol{\mathcal{X}}_1, ..., \boldsymbol{\mathcal{X}}_T$; forgetting factor $\mu$
initialize $\mathbf{A}_0^{(1)}, ..., \mathbf{A}_0^{(N)}$
$\mathbf{G}_0 \leftarrow 0$
for $t$ = 1, ..., $T$ do
    $s_t \leftarrow$ closed form update
    /* Extract rows of $\mathbf{C}_{t-1}$ that matches current tensor $\boldsymbol{\mathcal{X}}_t$'s
      nonzero rows */
    for $n$ = 1, ..., $N$ do
        if $t > 1$ then
            $\mathbf{nz}^{(n)} \leftarrow nonzero\_slices(\boldsymbol{\mathcal{X}}_t, n)$
            $\mathbf{C}_{old} \leftarrow \mathbf{A}^{(n)}[\mathbf{nz}_{t-1}^{(n)} \backslash \mathbf{nz}^{(n)}]^T \mathbf{A}^{(n)}[\mathbf{nz}_{t-1}^{(n)} \backslash \mathbf{nz}^{(n)}]$
            $\mathbf{C}_{new} \leftarrow \mathbf{A}^{(n)}[\mathbf{nz}^{(n)} \backslash \mathbf{nz}_{t-1}^{(n)}]^T \mathbf{A}^{(n)}[\mathbf{nz}^{(n)} \backslash \mathbf{nz}_{t-1}^{(n)}]$
            $\mathbf{C}_{z,t-1}^{(n)} + = \mathbf{C}_{old} - \mathbf{C}_{new}$
        end
    end
    repeat
        $\delta \leftarrow 0$
        for $n$ = 1, ..., $N$ do
            $\mathbf{Q}^{(n)} \leftarrow \left( \underset{v \neq n}{\circledast} \mathbf{H}^{(v)} \right) \circledast \mu \mathbf{G}_{t-1}$
            $\mathbf{\Phi}^{(n)} \leftarrow \left( \overset{N}{\underset{v \neq n}{\circledast}} \mathbf{C}^{(v)} \right) \circledast \left( \mu \mathbf{G}_{t-1} + s_t s_t^T \right)$
            /* Solve for non-zero slices */
            $\mathbf{A}_{nz}^{(n)} \leftarrow$
              $\left( spMTTKRP\left( \boldsymbol{\mathcal{X}}_t, \{\mathbf{A}_{nz}^{(v)}\} \right) + \mathbf{A}_{nz,t-1}^{(n)} \mathbf{Q}^{(n)} \right) \cdot$
              $\left( \mathbf{\Phi}^{(n)} \right)^{-1}$
            $\mathbf{C}_{nz}^{(n)} \leftarrow \mathbf{A}_{nz}^{(n)T} \mathbf{A}_{nz}^{(n)}$
            $\mathbf{H}_{nz}^{(n)} \leftarrow \mathbf{A}_{nz,t-1}^{(n)T} \mathbf{A}_{nz}^{(n)}$
            /* Solve for zero slices */
            $\mathbf{H}_z^{(n)} \leftarrow \mathbf{C}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} \left( \mathbf{\Phi}^{(n)} \right)^{-1}$
            $\mathbf{C}_z^{(n)} \leftarrow \left( \mathbf{\Phi}^{(n)} \right)^{-T} \mathbf{Q}^{(n)T} \mathbf{C}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} \left( \mathbf{\Phi}^{(n)} \right)^{-1}$
            /* Update C and H */
            $\mathbf{C}^{(n)} \leftarrow \mathbf{C}_z^{(n)} + \mathbf{C}_{nz}^{(n)}$
            $\mathbf{H}^{(n)} \leftarrow \mathbf{H}_z^{(n)} + \mathbf{H}_{nz}^{(n)}$
            /* Normalize */
            $normalize\,(\mathbf{C}, \mathbf{H})$
            $\delta_t = \sum_{n=1}^{N} \sqrt{\frac{tr\left( \mathbf{C}^{(n)} - \mathbf{H}^{(n)T} - \mathbf{H}^{(n)} + \mathbf{C}_{t-1}^{(n)} \right)}{tr\left( \mathbf{C}^{(n)} \right)}}$
        end
    until convergence;
    $\mathbf{A}^{(n)} \leftarrow \mathbf{A}_z^{(n)} \oplus \mathbf{A}_{nz}^{(n)} \; \forall \; n \in \{1, \dots, N\}$
    $\mathbf{G}_t = \mu \mathbf{G}_{t-1} + s_t s_t^T$
end
```

TABLE II: Data sets used in our evaluation. The streaming mode is in bold for each data set.

Data	Dimensions	# non-zeros
Patents	**year** × terms × terms **46** × 239k × 239k	3.5B
Flickr	user × image × tag × **date** 320K × 28M × 1.6M × **731**	113M
Uber	date × hour × lat. × long. **183** × 24 × 1.1K × 1.7K	3.3M
NIPS	paper × author × word × **year** 2.5K × 2.9K × 14K × **7**	3.1M

*1) **Blocked & Fused ADMM**:* Figure 2 compares the performance of our Blocked & Fused (**BF**) *ADMM* to the **baseline** *ADMM* implementation. As previously determined by our Roofline analysis (Section IV-A), the *ADMM* kernel is memory bandwidth-bound. This is why the execution time stops scaling for both implementations when the number of threads exceeds 14 for the *NIPS* dataset — there is already enough memory requests to saturate the memory bandwidth. Our *cache blocking* and *operation fusion* optimizations, *while still memory bandwidth-bound*, reduces the overall memory traffic, leading to as much as 8.1× speedup with 56 threads at rank-16. We omit the results for rank-64 and rank-128 as they show a similar trend as rank-16, with speedup peaking at 7.7× and 4.2×, respectively.

The only anomaly occurs with 56 threads at rank-32, where we see a significantly higher speedup of 12.3×. This is likely due to our using the *minimum* execution time combined with particular poor execution for all 10 trials for the **baseline**, which also explains why the execution time *goes up* from 28 threads to 56 threads.

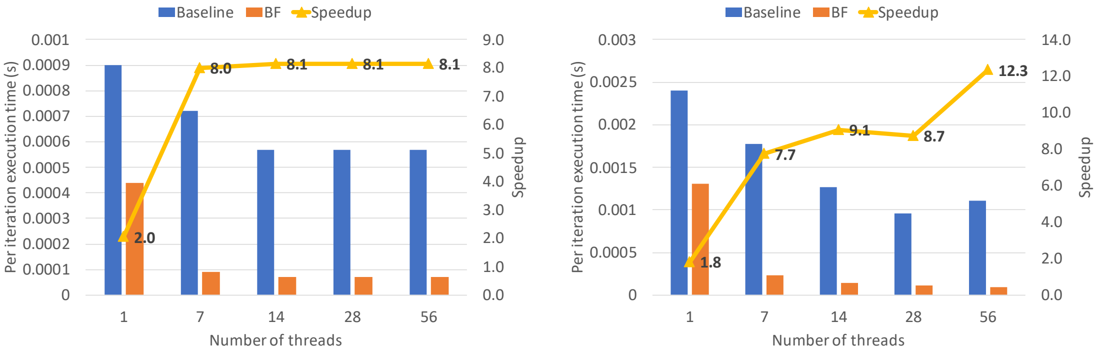
Figure 3 shows the speedup achieved by our Blocked & Fused *ADMM* for three different datasets using 56 threads. We see a similar trend as before, except with *Uber* showing lower speedup — this is due to the *factor matrices fitting in cache* (i.e., small dimension sizes for all modes), and as a result, the *data blocking* part of the optimization has no impact. The observed speedup can be mostly attributed to *operation fusion*.

*2) **Hybrid Lock MTTKRP**:* Figure 4 compares the performance our Hybrid Lock (**HL**) *MTTKRP* kernel to the **baseline** *MTTKRP* implementation. We observe both significant speedup and better scalability with increasing number of threads, since our implementation reduces thread contention. In contrast, the baseline performs *worse* with more threads due to contention. Using 56 threads, we achieve 30.6× and 24.1× speedup for rank-16 and rank-128 decomposition, respectively. We omit the remaining ranks (i.e., rank-32 and rank-64), as they show a similar trend, with speedup peaking at 26.8× and 24.1×, respectively.

Figure 3 shows the speedup achieved by our Hybrid Lock *MTTKRP* for three different datasets using 56 threads. We observe lower speedups at higher ranks — this is due to the lower probability of contention at higher ranks, since the same number of contentions will be spread over a longer execution time. *Uber* again shows a significantly lower level of speedup due to factor matrices fitting in cache — updates occur more quickly in cache, leading to lower wait time during contention.

*3) **Optimized Constrained CP-stream**:* Figure 5 shows the speedup comparison between our **optimized** *constrained CP-stream* implementation (i.e., using both BF *ADMM* and HL *MTTKRP*) against the baseline using 56 threads. The overall speedup trend mimics those of *ADMM* and *MTTKRP* kernels, and we achieve significant speedup on datasets with moderate (e.g., *NIPS*) to large (e.g., *Patents*) dimension sizes. Even on extremely small tensors whose factor matrices fit in cache (e.g., *Uber*), we achieve at least 3× to 5× speedup.

Note that, depending on the tensor property and initial values, the number of iterations to convergence for *ADMM* will



(a) Rank = 16

(b) Rank = 32

Fig. 2: Comparison of Blocked & Fused *ADMM* kernel to the baseline on the NIPS dataset.

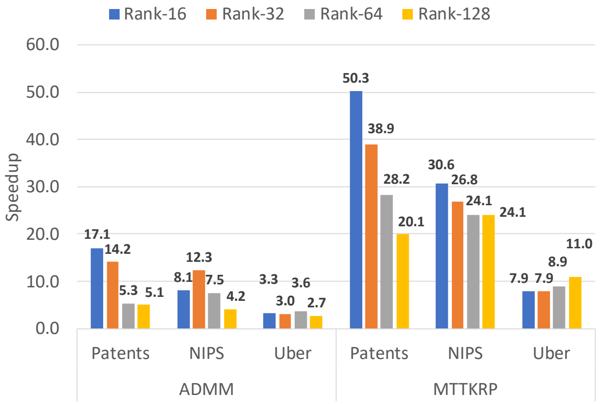


Fig. 3: Speedup achieved by our Blocked & Fused *ADMM* and Hybrid Lock *MTTKRP* over the baseline for three datasets using 56 threads.

vary. This may lead to *ADMM* making up a smaller percentage of the overall execution time compared to *MTTKRP*, and the overall speedup may be closer to that of *MTTKRP* (e.g., 47.0× for *Patents* at rank-16), or vice versa.

E. Non-constrained CP-stream

In many real-world problems, applying constraints to the factor matrices is unnecessary, and the *non-constrained CP-stream* can be used (i.e., *ADMM* is replaced with a direct solver such as *Cholesky*), as it performs faster than its *constrained* counterpart. In this subsection, we present the performance of our new *spCP-stream* algorithm, and compare it against the **baseline** *non-constrained CP-stream*, and our **optimized** implementation that uses our Hybrid Lock *MTTKRP*.

1) Sparse MTTKRP (spMTTKRP): First, we demonstrate the effectiveness of dividing the factor matrices into two subsets of non-zero rows (i.e., $\boldsymbol{A}_{nz}^{(n)}$) and zero-rows (i.e., $\boldsymbol{A}_{z}^{(n)}$) on the *MTTKRP* kernel. Our *spMTTKRP* kernel achieves as much as 121.1× speedup against the baseline *MTTKRP* kernel, and as much as 6.4× speedup against our own Hybrid Lock *MTTKRP* kernel.

Much of this speedup comes from more efficient memory access, as the accesses are spread over a much smaller memory footprint (e.g., accessing 10 rows in a 10-row matrix vs. accessing 10 rows in a 1000-row matrix), leading to fewer TLB misses and better pre-fetching performance.

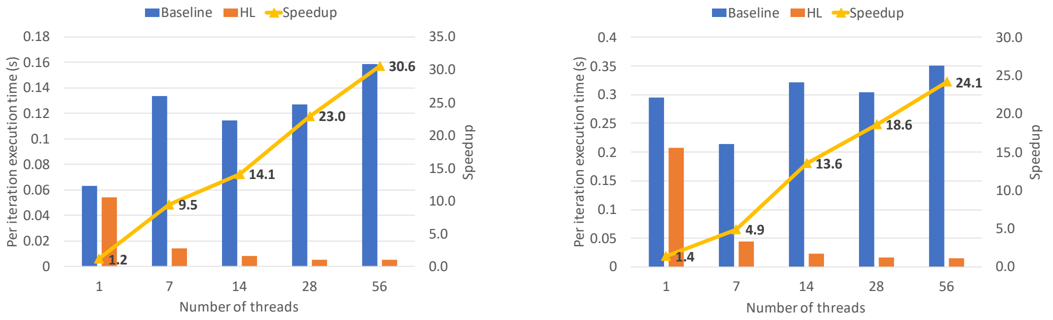
However, it should be noted that identifying the non-zero rows and creating the required data structures do not come for free. They add an extra overhead to our new formulation, and needs to be amortized effectively. Therefore, we do not present any direct comparison between our *Sparse MTTKRP* kernel and others. Instead, we present results that compare the performance of the overall algorithm, including the overhead.

2) spCP-Stream: In this subsection, we present the performance of our new *spCP-stream* algorithm and compare it against the **baseline** and **optimized** *non-constrained CP-stream* implementations. Our optimized implementation uses our Hybrid Lock *MTTKRP*. Figure 6 shows the per iteration execution time for the NIPS dataset.

For rank-16 decomposition, our **optimized** implementation achieves 18.8× speedup on 56 threads, whereas our new algorithmic formulation achieves 31.9× speedup. For rank-128 decomposition, the **optimized** implementation and *spCP-stream* achieved 10.4× and 12.0×, respectively. The smaller *gap* between the two speedups at higher ranks is due to *spCP-stream* using Gram matrices — as the rank goes up, the computation scales exponentially (i.e., $K \times K$), whereas the original algorithm scales linearly with rank (i.e., $I_n \times K$). We omit the remaining ranks, as they show a similar trend to the rank-16 decomposition, with their speedup sitting between those of rank-16 and rank-128.

Figure 7 shows how *spCP-stream* and our **optimized** implementation compare to the baseline on the remaining datasets. First, for the *Patents* dataset, our algorithms scale well up to 28 threads. However, the **baseline** execution time *increases* with the number of threads due to thread contention in *MTTKRP*, leading to 102.2× and 54.2× speedup at 56 threads for *spCP-stream* and **optimized**, respectively.

For the *Uber* dataset, in contrast, the execution time actually goes *up* slightly for all three implementations. This is because the number of non-zero elements and the dimension sizes are



(a) Rank-16 decomposition

(b) Rank-128 decomposition

Fig. 4: Comparison of Hybrid Lock (HL) *MTTKRP* kernel over the baseline on the NIPS dataset.

both extremely small, and there is not enough work to keep the system fully occupied.

The *Flickr* dataset shows a *very* different trend. With the other two datasets, *spCP-stream* achieves from $1.5\times$ to $2.7\times$ speedup over our **optimized** implementation. However, with the *Flickr* dataset, this goes up to $5.6\times$ to $17.9\times$. This is due to the extremely sparse nature of the second mode of the *Flickr* dataset (Section V-A), where approximately 99% of the rows are zero rows. This causes our row-sparse *MTTKRP* to perform significantly better than our Hybrid Lock *MTTKRP*, leading to such high speedups for the overall algorithm.

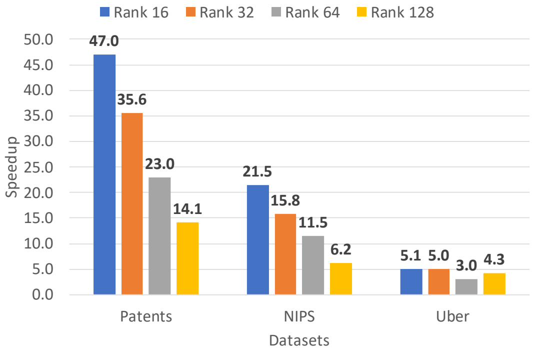


Fig. 5: *Overall* speedup for our **optimized** *constrained CP-stream* compared to the baseline using 56 threads.

3) Execution Time Breakdown: Another reason why *spCP-stream* performs so well on *Flickr* lies in how the *other* computations are also reduced. Figure 8 shows a breakdown of where time is spent in each iteration of the three implementations. As it can be seen, our Hybrid Lock *MTTKRP* significantly reduces the time spent in *MTTKRP* (i.e., **baseline** vs. **Optimized**). However, calculating the *Historical* information (Line 5 in Algorithm 1) still takes up a significant amount of time, as it involves multiple matrix-matrix multiplications between extremely large factor matrices (e.g., $28M \times K$ for mode 2). Since we replace this with *Hadamard product between Gram matrices* (i.e., $K \times K$), the computational complexity is greatly reduced, and *spCP-stream* achieves significant speedup against even our own **optimized** implementation. This behavior should occur in any tensors with very large dimension sizes.

VII. Conclusion and Future Work

In this work, we propose *spCP-stream*, a novel streaming TD algorithm which lowers the overall computational complexity and yields significant performance speedup against state-of-the-art *CP-stream* for non-constrained problems. We demonstrate its effectiveness on four real-world tensors, where it outperforms *CP-stream* by a factor of up to *102*×. For constrained problems, we present *optimized CP-stream*, which improves *ADMM* and *MTTKRP* to achieve up to $47\times$ speedup.

Despite the better performance, our current *spCP-stream* algorithm can only be applied to *non-constrained* problems due to its incompatibility with *ADMM*. In future work, we will focus on integrating *ADMM* into *spCP-stream* to further improve its performance.

References

[1] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, “Tensor decomposition for signal processing and machine learning,” *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[2] S. Smith, K. Huang, N. D. Sidiropoulos, and G. Karypis, “Streaming tensor factorization for infinite data sources,” *Proceedings of the 2018 SIAM International Conference on Data Mining (SDM’18)*, 2018.

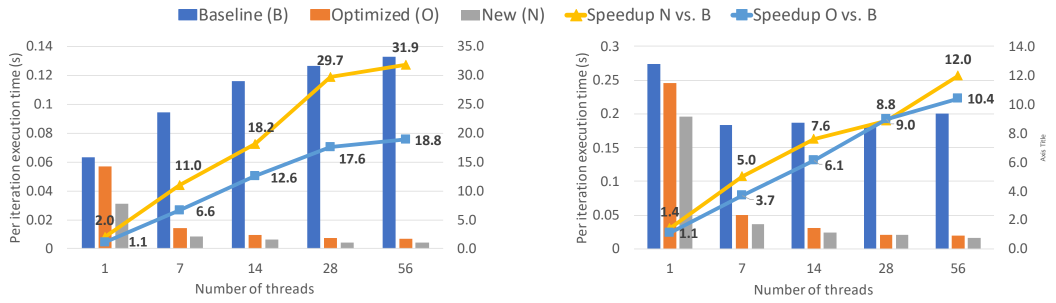
[3] H. Kasai, “Online low-rank tensor subspace tracking from incomplete data by cp decomposition using recursive least squares,” in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016, pp. 2519–2523.

[4] S. Zhou, N. X. Vinh, J. Bailey, Y. Jia, and I. Davidson, “Accelerating online cp decompositions for higher order tensors,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1375–1384. [Online]. Available: https://doi.org/10.1145/2939672.2939763

[5] M. Mardani, G. Mateos, and G. B. Giannakis, “Subspace learning and imputation for streaming big data matrices and tensors,” *IEEE Transactions on Signal Processing*, vol. 63, no. 10, pp. 2663–2677, 2015.

[6] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

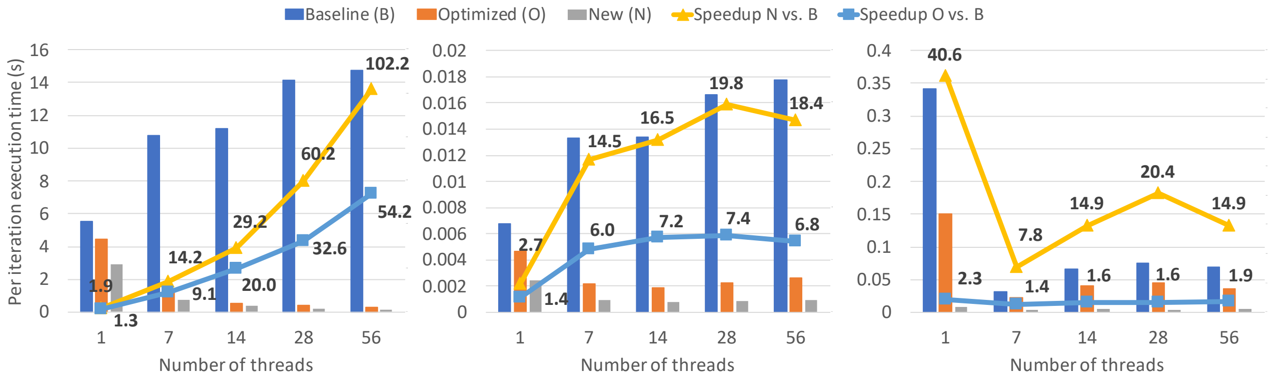
[7] P. Matuszyk and M. Spiliopoulou, “Selective forgetting for incremental matrix factorization in recommender systems,” in *Discovery Science*, S. Džeroski, P. Panov, D. Kocev, and L. Todorovski, Eds. Cham: Springer International Publishing, 2014, pp. 204–215.



(a) Rank-16 decomposition

(b) Rank-128 decomposition

Fig. 6: Comparison of *spCP-stream* and **optimized** *non-constrained CP-stream* to the baseline on the *NIPS* dataset.



(a) Patents

(b) Uber

(c) Flickr

Fig. 7: Comparison of *spCP-stream* and **optimized** *non-constrained CP-stream* to the baseline on the *Patents*, *Uber*, and *Flickr* datasets for rank-16 decomposition.

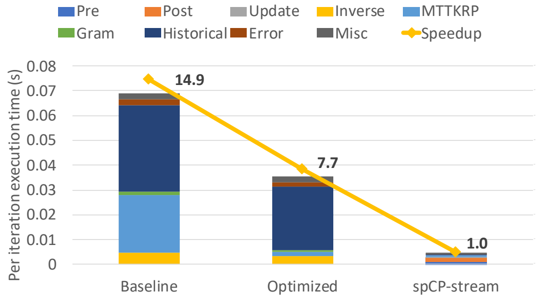


Fig. 8: Execution time breakdown for the three implementations for the *Flickr* dataset (rank-16 using 56 threads).

[8] W. Wang, H. Yin, Z. Huang, Q. Wang, X. Du, and Q. V. H. Nguyen, "Streaming ranking based recommender systems," in *The 41st International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 525–534. [Online]. Available: https://doi.org/10.1145/3209978.3210016

[9] A. Mensch, J. Mairal, B. Thirion, and G. Varoquaux, "Dictionary learning for massive matrix factorization," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, p. 1737–1746.

[10] J. Mairal, F. Bach, J. Ponce, and G. Sapiro, "Online learning for matrix factorization and sparse coding," *J. Mach. Learn. Res.*, vol. 11, p. 19–60, Mar. 2010.

[11] Bin Yang, "Projection approximation subspace tracking," *IEEE Transactions on Signal Processing*, vol. 43, no. 1, pp. 95–107, 1995.

[12] D. Nion and N. D. Sidiropoulos, "Adaptive algorithms to track the parafac decomposition of a third-order tensor," *IEEE Transactions on Signal Processing*, vol. 57, no. 6, pp. 2299–2310, 2009.

[13] 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/

[14] J. Li, J. Sun, and R. Vuduc, "Hicoo: Hierarchical storage of sparse tensors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[15] S. Smith and G. Karypis, "Accelerating the tucker decomposition with compressed sparse tensors," in *European Conference on Parallel Processing*. Springer, 2017.

[16] J. W. Choi, X. Liu, S. Smith, and T. Simon, "Blocking optimization techniques for sparse tensor computation," in *32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*, 2018.

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

[18] M. Vandecappelle, N. Vervliet, and L. De Lathauwer, "Nonlinear least squares updating of the canonical polyadic decomposition," in *2017 25th European Signal Processing Conference (EUSIPCO)*, 2017, pp. 663–667.

[19] 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, pp. 1–122, 01 2011.

[20] S. Smith, A. Beri, and G. Karypis, "Constrained tensor factorization with accelerated ao-admm," in *46th International Conference on Parallel Processing (ICPP '17)*. IEEE, 2017.