Sparsity-Aware Tensor Decomposition

Süreyya Emre Kurt
School of Computing
University of Utah
Salt Lake City, Utah
semre@cs.utah.edu

Saurabh Raje
School of Computing
University of Utah
Salt Lake City, Utah
saurabh.raje@utah.edu

Aravind Sukumaran-Rajam
Department of EECS
Washington State University
Pullman, Washington
a.sukumaranrajam@wsu.edu

P. Sadayappan
School of Computing
University of Utah
Salt Lake City, Utah
saday@cs.utah.edu

***Abstract*—Sparse tensor decomposition, such as Canonical Polyadic Decomposition (CPD), is a key operation for data analytics and machine learning. Its computation is dominated by a set of MTTKRP (Matricized Tensor Times Khatri Rao Product) operations. The collection of required MTTKRP operations for sparse CPD include common sub-computations across them and many approaches exist to factorize and reuse common sub-expressions. Prior work on sparse CPD has focused on minimizing the number of high-level operators. In this paper, we consider a design space that covers whether the partial MTTKRP results should be saved, different mode permutations and model the total volume of data movement to/from memory. Also, we propose a fine-grained load balancing method that supports higher levels of parallelization.**

***Index Terms*—CPD, MTTKRP, sparse tensor factorization**

I. Introduction

Sparse tensor decomposition, such as Canonical Polyadic Decomposition (CPD), is a key operation for data analytics and machine learning. The computation is dominated by a set of MTTKRP (Matricized Tensor Times Khatri Rao Product) operations [1]. Consider the CP decomposition of a 4D tensor $\mathcal{T}(i,j,k,l)$ into a product of four 2D matrices $A(i,r)$, $B(j,r)$, $C(k,r)$, and $D(l,r)$:

$$\mathcal{T}(i,j,k,l) \approx \sum_r A(i,r)B(j,r)C(k,r)D(l,r)$$

The MTTKRP operation computes:

$$\tilde{A}(i,r) = \sum_{j,k,l} \mathcal{T}(i,j,k,l)B(j,r)C(k,r)D(l,r)$$

The iterative algorithm for CPD of an N-dimensional tensor involves a sequence of N MTTKRP operations. For example, CPD of a 4D tensor requires a sequence of 4 MTTKRP operations in a loop until convergence is achieved:

$$\begin{aligned}
&\overline{A}(i,r) = \sum_{j,k,l} \mathcal{T}(i,j,k,l)B(j,r)C(k,r)D(l,r);\ A_n = f(\overline{A},B,C,D)\\
&\overline{B}(j,r) = \sum_{i,k,l} \mathcal{T}(i,j,k,l)A_n(i,r)C(k,r)D(l,r);\ B_n = f(\overline{B},A_n,C,D)\\
&\overline{C}(k,r) = \sum_{i,j,l} \mathcal{T}(i,j,k,l)A_n(i,r)B_n(j,r)D(l,r)\\
&C_n = f(\overline{C},A_n,B_n,D)\\
&\overline{D}(l,r) = \sum_{i,j,k} \mathcal{T}(i,j,k,l)A_n(i,r)B_n(j,r)C_n(k,r)\\
&D_n = f(\overline{D},A_n,B_n,C_n); A = A_n; B = B_n; C = C_n; D = D_n
\end{aligned} \tag{1}$$

The function $f(S,X,Y,Z)$ performs $S{*}(X^\top XY^\top YZ^\top Z)^{-1}$ to generate the new version of the factor matrix. It involves dense matrix operations on small matrices and its cost is low compared to the MTTKRP operations.

Several research efforts have addressed the development of efficient parallel implementations of sparse MTTKRP [2]–[6]. In addition to optimizing individual MTTKRP operations, there is an opportunity for optimization when we consider the collection of MTTKRP operations used in CP Decomposition. It may be seen in this example that the set of 4 MTTKRP operations involves some common sub-expressions and therefore computations can be saved by computing, storing and reusing intermediate tensors corresponding to some common sub-expressions in the collection of MTTKRP computations. For example, the first two MTTKRP operations in the above sequence of 4 MTTKRP operations can be performed using fewer arithmetic operations if the intermediate tensor $\mathcal{I}$ is computed and stored as follows:

$$\begin{aligned}
\mathcal{I}(i,j,r) &= \sum_{k,l} \mathcal{T}(i,j,k,l)C(k,r)D(l,r);\\
\overline{A}(i,r) &= \sum_{j} \mathcal{I}(i,j,r)B(j,r); A_n = f(\overline{A},B,C,D)\\
\overline{B}(j,r) &= \sum_{i} \mathcal{I}(i,j,r)A_n(i,r); B_n = f(\overline{B},A_n,C,D)
\end{aligned}$$

This idea of restructuring the computation, storing and reusing an intermediate result is termed as *memoization*. It has been studied in a few prior studies [4] [3] [7] [8]. A key question in this context is that of determining which of many alternative memoization schemes is the best. The design and implementation of an efficient memoization scheme for sparse tensors is more challenging than the dense case [7] because of the constraints on efficient data access imposed by any compact representation of sparse tensors (such as the CSF) and the resultant irregularity of the data access patterns.

In comparison with prior studies on efficient sparse MTTKRP for tensor factorization, we make the following contributions in this paper:

- We consider a more extensive design space of memoization schemes, and implement a collection of parallel kernels for each scheme.
- We develop a sparsity-aware model of data movement. Such a model allows rapid search of the design space of

memoized configurations. Fast dynamic selection of this configuration for a given sparse tensor is then used to accelerate the decomposition.

- We develop a novel scheme for load-balanced parallel execution of the selected memoization scheme on shared-memory multiprocessors.
- We combine all our optimizations in a STeF (Sparse Tensor Factorization) implementation and we demonstrate superior performance over several prior schemes with publicly available codebases, including AdaTM [4], ALTO [5], SPLATT [2], [9] and TACO [10].

II. Overview

In this section, we provide a high-level overview of the key ideas incorporated in the new memoized MTTKRP approach for CPD presented in this paper. We begin by presenting notations used in describing the sequence of operations invoked for the memoized execution.

A. Notations & Mathematical Preliminaries

Tensor decomposition algorithms use several binary operators that work with combinations of tensors, matrices and vectors. Kolda et al. [1] provides a comprehensive overview of all such operators. We now define the ones used specifically in the context of CP decomposition, i.e., the scope of this work.

$\mathcal{T}$, $A^{(i)}$ and R: We use $\mathcal{T}$ to denote a tensor. Let d be the dimensionality of the tensor. We denote the factor matrices as $A^{(0)}$,$A^{(1)}$,$\ldots$,$A^{(d-1)}$, each of rank R, the number of columns.

Khatri-Rao product (KRP): For matrices $A \in \mathbb{R}^{I\times R}, B \in \mathbb{R}^{J\times R}$, the Khatri-Rao product (KRP) is a matrix $M \in \mathbb{R}^{(IJ)\times R}$; the KRP operator is denoted by the $\odot$ symbol. $M[i\times J + j, r] = A[i,r].B[j,r] \forall i \in [0,I), j\in[0,J), r \in [0,R)$

Tensor Times Matrix (TTM) product: This product "unfolds" a tensor $\mathcal{T} \in \mathbb{R}^{I_0\times\ldots\times I_{d-1}}$ along dimension k into a matrix $T \in \mathbb{R}^{I_i\times(I_0\ldots I_{i-1}I_{i+1}\ldots I_{d-1})}$ and performs the matrix product with another matrix $A \in \mathbb{R}^{I_i\times R}$. The result is a matrix of shape $(I_0\ldots I_{k-1}I_{k+1}\ldots I_{d-1})\times R$. The result is re-shaped back into a tensor $\mathcal{P} \in \mathbb{R}^{I_0\times I_1\times\ldots I_{k-1}\times I_{k+1}\times\ldots I_{d-1}\times R}$, which is called a partially contracted tensor. The TTM operation can be written as $\mathcal{P}[i_0,\ldots,i_{k-1},i_{k+1},\ldots,i_{d-1},r] = \mathcal{T}_k[i_0,\ldots,i_{d-1}]\times_k A[i_k,r]$, where $\times_k$ denotes a contraction (i.e., summation over) of index i_k. In this work, we use the TTM only to contract the last dimension of the tensor, i.e., dimension $d-1$ in the above example.

multi Tensor Times Vector (mTTV) product: For a partially contracted tensor $\mathcal{P} \in \mathbb{R}^{I_0\times\cdots\times I_k\times R}$ (the last dimensions have been contracted out), mTTV with a matrix $A \in \mathbb{R}^{I_k\times R}$ yields another tensor of shape $I_0\times\cdots\times I_{k-1}\times R$. So, mTTV operation can be written as $\overline{\mathcal{P}}[i_0,\ldots,i_{k-1},r] = \mathcal{P}_k[i_0,\ldots,i_k,r]\times_k A[i_k,r]$.

MTTV: In this paper, we define MTTV as the Matricized Tensor Times Vector operation. For given tensor $\mathcal{P} \in \mathbb{R}^{I_0\times\cdots\times I_k\times R}$, the product with matrix $A \in \mathbb{R}^{(I_0\times\cdots\times I_{k-1})\times R}$ yields a matrix of shape $I_k\times R$, without the first k dimension. Common dimensions $I_0,\ldots,I_{k-1}$ used as contraction index whereas R used as the batch index in the contraction. So, MTTV operation can be written as $\overline{A}[i_k,r] = \mathcal{P}'[i_0\ldots i_{k-1},i_k,r]\times_{0,1,..,k-1} A[i_0i_1\ldots i_{k-1},r]$, where $\mathcal{P}'$ is a tensor where first k-1 modes of $\mathcal{P}$ are combined into a single flattened mode.

$A^{(i)}$ and $a_r^{(i)}$: $A^{(i)}$ denotes the factor matrix for mode i. The number of columns for every factor matrix is the number of ranks, R. Column r of matrix is $A^{(i)}$ is a vector denoted $a_r^{(i)}$.

$K^{(i)}$: We denote as $K^{(i)}$ the Khatri-Rao product of the set of factor matrices $A^{(0)}$,$A^{(1)}$,...,$A^{(i)}$, for $i>0$. $K^{(i)}$ can be represented using the recurrence relation $K^{(i)}$=$K^{(i-1)} \odot A^{(i)}$, with $K^{(0)}$=$A^{(0)}$ as the base case.

$\mathcal{P}^{(i)}$: The partial MTTKRP result after factor matrices $A^{(i+1)}$,$A^{(i+2)}$,...,$A^{(d-1)}$ are contracted with $\mathcal{T}$ is denoted as $\mathcal{P}^{(i)}$. Thus, using a TTM operation we get $\mathcal{P}^{(d-2)} = \mathcal{T}$ $A^{(d-1)}$, a d dimensional tensor. Similarly, using an mTTV operation we get $\mathcal{P}^{(i')}[i_0,...,i_{i'}$,r] = $\mathcal{P}^{(i'+1)}[i_0,...,i_{i'},i_{i'+1}$,r] $\times_{i'+1}$ $A^{(i+1)}[i_{i'+1}$,r], for $i' < d-2$. For convenience, we define $\mathcal{P}^{(d-1)} = \mathcal{T}$.

B. The Compressed Sparse Fiber (CSF) format for storing tensors

Sparse tensors are stored in some compact form that only requires space proportional to the number of non-zero elements. A commonly used format is a tree-based representation called the Compressed Sparse Fiber (CSF) [2] format. For an N dimensional tensor, the CSF tree will have depth N, with each tree level corresponding to one dimension of the tensor. The leaf level stores the non-zero values from the tensor. Each internal node stores the indices of the non-zero fibers of the tensor in the dimension corresponding to the depth of the node. Therefore, the total number of nodes of the tree (and therefore the memory needed to store the sparse tensor) depends on the order of dimensions in the tree. This is also called the "mode-order" of the CSF structure. A common heuristic is to sort the tensor's modes in increasing order of length, and order the CSF tree from root to leaf in that order.

Alg. 1 shows pseudo-code for a 4D MTTKRP operation on a 4D tensor $\mathcal{T}$ using a CSF structure.

C. The space of memoized schemes

Algorithm 2 shows the CPD-ALS algorithm for a 4D tensor. $\mathcal{T}_i$ is unfolding of the tensor $\mathcal{T}$ on i^{th} dimension. In our notation, we will refer to A, B, C and D in this algorithm as $A^{(0)}$, $A^{(1)}$, $A^{(2)}$ and $A^{(3)}$. Fig. 1 illustrates different options for updating $A^{(1)}$. $\times_i$ defines the contraction of i^{th} dimension of the first operand. The factor matrices $A^{(0)}$, $A^{(1)}$, $A^{(2)}$, and $A^{(3)}$ correspond to the modes 0, 1, 2, 3, respectively. Fig. 1(a) shows a sequence of three tensor operations to produce $\bar{A}^{(0)}$: i) A TTM (Tensor Times Matrix) operation $\mathcal{T}[i_0,i_1,i_2,i_3]\times_3 A^{(3)}[i_3,r]$ to produce $\mathcal{P}^{(2)}[i_0,i_1,i_2,r]$; ii) An mTTV (multi Tensor Times Vector) operation $\mathcal{P}^{(2)}[i_0,i_1,i_2,r]\times_2 A^{(2)}[i_2,r]$ to produce $\mathcal{P}^{(1)}[i_0,i_1,r]$; iii) An mTTV operation $\mathcal{P}^{(1)}[i_0,i_1,r]\times_1 A^{(1)}[i_1,r]$ to produce $\bar{A}^{(0)}[i_0,r]$. Fig. 1(b-d) show three options for producing $\bar{A}^{(1)}$.

Algorithm 1: Simple pseudo-code for MTTKRP using a 4D tensor $\mathcal{T}$ using CSF pointers.

```
// $\tilde{A}$ = MTTKRP($\mathcal{T}$,$B$,$C$,$D$)
input : Sparse tensor $\mathcal{T}$ is a 4D tensor. $B$, $C$ and $D$
        represent the factor matrices. $R$ is the rank of
        decomposition.
output: Output $\tilde{A}$ needed to update factorization matrix $A$
$num\_slice \leftarrow Number\ of\ slices\ in\ the\ tensor$
for $i_p$ = 0 to $num\_slice$-1 do in parallel
    $i \leftarrow$ indexI[$i_p$]
    for $j_p$ = ptrI[$i_p$] to ptrI[$i_p$+1]-1 do
        $j \leftarrow$ indexJ[$j_p$]
        for $k_p$ = ptrJ[$j_p$] to ptrJ[$j_p$+1]-1 do
            $k \leftarrow$ indexK[$k_p$]
            for $l_p$ = ptrK[$k_p$] to ptrK[$k_p$+1]-1 do
                $l \leftarrow$ indexL[$l_p$]
                for r in [0:R-1] do
                    $\tilde{A}$[i,r] $\leftarrow$ $\tilde{A}$[i,r] + $\mathcal{T}$[i,j,k,l]$B$[j,r] $C$[k,r]
                      $D$[l,r]
```

Algorithm 2: CPD-ALS code for a 4D tensor

```
input :   $\mathcal{T}$ a 4D tensor
output:   Factorization matrices $A^{(0)}$, $A^{(1)}$, $A^{(2)}$ and $A^{(3)}$
for it = 1 to max iterations do
    $V \leftarrow A^{(1)\top}A^{(1)}A^{(2)\top}A^{(2)}A^{(3)\top}A^{(3)}$
    $A^{(0)} \leftarrow \mathcal{T}_0(A^{(1)} \bigodot A^{(2)} \bigodot A^{(3)})V^{-1}$
    Normalize columns of $A^{(0)}$ and save norms in $\lambda$
    $V \leftarrow A^{(0)\top}A^{(0)}A^{(2)\top}A^{(2)}A^{(3)\top}A^{(3)}$
    $A^{(1)} \leftarrow \mathcal{T}_1(A^{(0)} \bigodot A^{(2)} \bigodot A^{(3)})V^{-1}$
    Normalize columns of $A^{(1)}$ and save norms in $\lambda$
    $V \leftarrow A^{(0)\top}A^{(0)}A^{(1)\top}A^{(1)}A^{(3)\top}A^{(3)}$
    $A^{(2)} \leftarrow \mathcal{T}_2(A^{(0)} \bigodot A^{(1)} \bigodot A^{(3)})V^{-1}$
    Normalize columns of $A^{(2)}$ and save norms in $\lambda$
    $V \leftarrow A^{(0)\top}A^{(0)}A^{(1)\top}A^{(1)}A^{(2)\top}A^{(2)}$
    $A^{(3)} \leftarrow \mathcal{T}_3(A^{(0)} \bigodot A^{(1)} \bigodot A^{(2)})V^{-1}$
    Normalize columns of $A^{(3)}$ and save norms in $\lambda$
    if Convergence condition met then
        exit loop
```

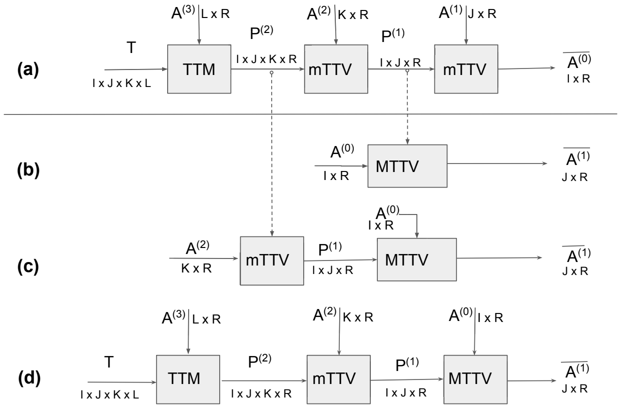


Fig. 1: Computation of the factor matrix corresponding to non-root mode 1. This is done after computing the factor for mode 0. Depending on what intermediates are saved, mode 1 computation can be done in 3 ways as shown in b), c), or d).

Fig. 1(b) shows how it can be produced with a minimal number of additional arithmetic operations, using saved intermediate tensor $\mathcal{P}^{(1)}[i_0, i_1, r]$ by performing the MTTV operation $A^{(0)}[i_0, r] \times_0 \mathcal{P}^{(1)}[i_0, i_1, r]$ to produce $\bar{A}^{(1)}[i_1, r]$.

Another option is to use the saved intermediate tensor $\mathcal{P}^{(2)}[i_0, i_1, i_2, r]$, by repeating the TTV operation $\mathcal{P}^{(2)}[i_0, i_1, i_2, r] \times_2 A^{(2)}[i_2, r]$ and fuse that computation with a MTTV operation with $A^{(0)}[i_0, r]$, as shown in Fig. 1(c). Finally, it is also possible to produce $\bar{A}^{(1)}[j, r]$ without reusing any of the computation used in producing $\bar{A}^{(0)}$, as shown in Fig. 1(d).

Note that these choices represent different amounts of computation and data movement. While memoization can save arithmetic operations, it introduces additional data movement. This could offset the benefits of memoization, especially since data movement is much more expensive than arithmetic operations. In Sec. IV we develop an analytical model to evaluate the efficacy of alternate memoization schemes.

D. Load-balanced work distribution

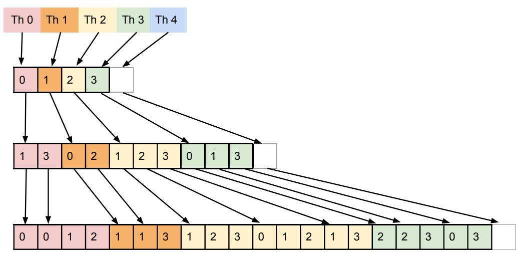
Prior efforts [2], [4] have partitioned the indices at the root mode and assigned slices to threads (Figure 2a). Such an approach can result in load imbalance if the number of slices at the root mode is less than the number of threads and/or the non-zeros are non-uniformly distributed across the slices. For example, in the tensors *vast-2015-mc1-3d* and *vast-2015-mc1-5d*, the number of slices at the root mode (for mode-length ordered CSF representation) is just 2. Therefore, this work division scheme can only utilize 2 threads. Furthermore, a 2-way partitioning for these tensors will also result in 1674% load imbalance, as a virtue of the distribution of non-zeros.

We propose a fine-grained work division strategy where all threads process roughly the same number of non-zeros. Simply dividing the non-zeros across threads may result in write conflict. For example, consider Figure 2b, where both threads 1 and 2 write to the same location at level 0 (similar conflict also occurs at level 1). To ensure correctness we could use use atomic updates; however, the cost of atomic operations will degrade the performance. Another option is to use privatization, keeping a private copy for each thread, but it increases the amount of data movement.

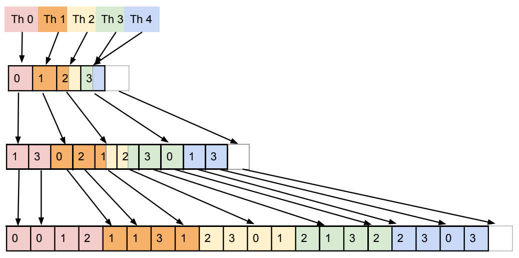
We devise a scheme that efficiently avoids the write conflict without using atomic operations or privatization. Our approach is based on the observation that the write conflicts, if any, will only occur at the boundary elements between the threads. The maximum number of elements that will potentially have write conflict is $\mathbf{T}$, where $\mathbf{T}$ is the number of threads. The write conflicts can be avoided by replicating just the boundary elements, i.e., instead of writing to a tensor of size $N * R$, where N is the number of rows and R is the number of features, we write the to a tensor of size $(N + \mathbf{T}) * R$. The details of this scheme is presented in Section III-A.

E. Switching Mode Order of Last Two Modes

Consider the sequence of steps for computing MTTKRP for a single-mode, e.g., for mode-0 in Fig. 1. It requires one TTM



(a) Slice-based Work Distribution



(b) STeF Work Distribution

Fig. 2: Work distribution for 5 threads. Fig. 2a shows work distribution with outer-mode slice based granularity. Since there are only 4 slices at the root mode, no more than 4 threads get any work. The maximum number of nodes traversed by a thread is 12 (yellow thread) while thread 4 (blue) has no work. Fig. 2b shows the STeF work distribution. Squares with two colors are traversed by 2 threads. The maximum number of nodes traversed by a thread is 9 (orange thread) and the lightest load is 6 nodes (blue thread).

operation using all non-zeros of the tensor $\mathcal{T}$, followed by a series of TTV operations using successively lower dimensional tensors that result from the TTM/TTV operations. At each stage, the number of non-zeros in the output tensor is lower than the input tensor by a factor corresponding to the average number of non-zeros per fiber along the contracted dimension of the input tensor. If the non-zeros in the input tensor $\mathcal{T}$ were uniformly distributed over the index space, the average number of non-zeros per fiber would be proportional to the mode lengths. If so, it would be best to order the modes such that mode lengths monotonically increases so that a maximal compression is achieved. Increasing the compression will result in lower number of non-zeros of the series of lower-dimensional tensors, thereby minimizing the total work in terms of the number of operations.

However, the average fiber length along different dimensions of a tensor is not always directly related to the mode lengths. For example, in *delicious-4d*, the average fiber length for the the longest mode of length 17 million is 1.5 whereas the same stat for the mode of length 2 million is 3. If the average fiber length is not longest along the longest mode, it would be preferable to use one of the other modes as the "fiber" mode in the CSF representation. While it would be very expensive to determine the average fiber length corresponding to all modes of a tensor, we have devised a very efficient algorithm to determine the average fiber length for the last two modes. We therefore consider swapping of the last two modes in the CSF representation. It also turns out that in practice, the longest average fiber length turns out to almost always to be one among the two longest tensor modes.

III. Algorithms & Optimizations

A. Load-balanced work distribution

As mentioned earlier, load balance plays an important role in determining performance. As shown in Figure 2, compared to the default scheme (Figure 2a), where the last thread was idle, our scheme eliminates thread idling and achieves a good load balance. The key to achieving high performance is to deliver good load balance without suffering from the overhead of atomic operations and additional data movement from privatization. First, we divide the non-zeros equally among each thread. Algorithm 3 shows how the start locations for each thread are computed. Each thread is first assigned an equal number of non-zeros. The start position of each thread is found by looking at the parent of non-zeros at the boundaries($find\_parent\_CSF$). Since we distribute *nnz*s across the threads, multiple threads may write to the same memory location in the upper levels of the tree and cause write conflicts. Only the elements at the thread boundary can have write conflicts; hence our scheme replicates these elements.

Algorithm 3: Finding thread start position of CSF representation of $\mathcal{T}$

```
input :  $d$ dimensional sparse tensor $\mathcal{T}$
output:  2D thread_start array showing thread starting
         position for each level of the CSF for each thread
1 for th=0 to number of threads do in parallel
2     thread_start[th][d-1] ← $th * nnz/numberofthreads$
3 for th=0 to number of threads do in parallel
4     for $i$ = $d-2$ to 0 do
5         thread_start[th][i] ←
            find_parent_CSF(thread_start[th][i+1])
```

B. Sparsity Aware MTTKRP

This section explains how all the optimizations proposed in this paper are put together to generate the complete MTTKRP code. For a d dimensional tensor $\mathcal{T}$, STeF chooses two different paths when computing MTTKRP for $\bar{A}^{(0)}$ and $\bar{A}^{(i)}$ where $i > 0$. For computing $\bar{A}^{(i)}$, STeF first performs a TTM operation on $\mathcal{T}$ to produce $\mathcal{P}^{(d-2)}$. Then, series of mTTV operation is used to compute $\mathcal{P}^{(d-3)}$,...,$\mathcal{P}^{(0)}$, where $\mathcal{P}^{(0)}$ becomes $\bar{A}^{(0)}$. If our model decided $\mathcal{P}^{(i)}$ needs to be memoized, that result is saved in memory. When computing MTTKRP for $\bar{A}^{(i)}$ STeF uses $\mathcal{P}^{(i)}$. If $\mathcal{P}^{(i)}$ is not memoized when computing $\bar{A}^{(0)}$, then it is recomputed either from a memoized $\mathcal{P}^{(k)}$ if $\exists k > i$ or from scratch by performing a series of TTM and mTTV on $\mathcal{T}$. MTTKRP for the first mode of the CSF is implemented as a series of dot products, whereas MTTKRP for the last mode of the CSF is a series of Khatri-Rao products. MTTKRP for the internal modes can therefore be written as a combination of dot products and KRPs. Algorithm 4 shows the generic MTTKRP algorithm for any mode, where Algorithm 5 shows the generic loop structure.

Algorithms 4 and 5 compute the MTTKRP of mode u, $\bar{A}^{(u)}$, by using $\mathcal{P}^{(u)}$ and $K^{(u-1)}$, for $0 < u < d-1$. For $u = 0$, $\bar{A}^{(0)} = \mathcal{P}^{(0)}$ and for $u = d-1$, $\bar{A}^{(d-1)} = \mathcal{T}_{d-1} * K^{(d-2)} = \mathcal{P}^{(d-1)}_{d-1} * K^{(d-2)}$, which is the original MTTKRP formulation. Algorithm 4 is the high level call handling the MTTKRP computation. Vectors t_0 and k_0 holds a row of the first factorization matrix $A^{(0)}$. At the end of the algorithm if the result $\bar{A}^{(u)}$ is privatized across the threads, it is reduced back to a single matrix. Lines 9-12 handles the the case of $u = 0$. Atomic updates are used in order to avoid race condition between threads. This can only happen if the update is at the thread boundary. If $u > 0$, then for updating $\bar{A}^{(u)}$ either atomic updates are needed, or each thread needs to hold it's own copy, ie. $\bar{A}^{(u)}$ needs to be privatized. If $\bar{A}^{(u)}$ is privatized line 11 of Algorithm 4 is executed to reduce it to a single matrix.

Algorithm 4: MTTKRP for mode u in tensor $\mathcal{T}$

```
input :  Tensor $\mathcal{T}$ in CSF format, factorization matrices $\mathbf{A}$
         = $\{A^{(0)},A^{(1)},...,A^{(d-1)}\}$, MTTKRP mode $u$
output:  Updated factorization matrix $\bar{A}^{(u)}$
1  for th=0 to number of threads - 1 do in parallel
2      for i=thread_start[th][0] to thread_start[th+1][0] do
3          if $u \neq 0$ then
4              $k_0[:] \leftarrow A^{(0)}$[i,:]
5          else
6              $t_0[:] \leftarrow A^{(0)}$[i,:]
7          MTTKRP-LOOP(1,i,$k_0$, $t_0$, $\mathcal{T}$, $\mathbf{A}$,$u$,$th$,$\bar{A}^{(u)}$)
8          if $u$ = 0 then
9              if i=thread_start[th][0] or
                 i=thread_start[th+1][0] then
10                 $\bar{A}^{(0)}$[i,:] $\leftarrow t_0$ with atomic update
11             else
12                 $\bar{A}^{(0)}$[i,:] $\leftarrow t_0$
13 if Is $\bar{A}^{(u)}$ privatized for each thread? then
14     Reduce $\bar{A}^{(u)}$
```

In Algorithm 5, k_i vector holds a row of KRP of the matrices $A^{(0)}$,..,$A^{(i)}$. Similarly, t_i holds a row of partial MTTKRP result after contracting out $A^{(i+1)}$,...,$A^{(d-1)}$. Thus, in order to compute the values $\bar{A}^{(u)}[i_u, r]$, we need to do a Hadamard product of k_{u-1} and t_u. $\mathcal{T}.save$ is an array of boolean values where $\mathcal{T}.save[i]$ denotes whether the partial MTTKRP result $\mathcal{P}^{(i)}$ after contracting out matrices $A^{(i+1)}$,...,$A^{(d-1)}$ is to be stored or not. $\mathcal{T}.ptr[i][idx]$ is a pointer array pointing the start position of children of node idx in mode i to mode $i+1$. $\mathcal{T}.val$ is the array holding values of the tensor.

As per the work distribution in Algorithm 3, two threads

Algorithm 5: MTTKRP-LOOP

```
input :  loop id $i$, pindex, $k_{i-1}$, $t_{i-1}$, Tensor $\mathcal{T}$,
         factorization matrices $\mathbf{A}$ = $\{A^{(0)},A^{(1)},...,A^{(d-1)}\}$,
         MTTKRP mode $u$, thread id $th$, $\bar{A}^{(u)}$
1  start $\leftarrow$ MAX(thread_start[th][i],$\mathcal{T}$.ptr[i-1][pindex] )
2  end $\leftarrow$ MIN(thread_start[th+1][i], $\mathcal{T}$.ptr[i-1][pindex+1])
3  for idx = start to end do
4      if $\mathcal{T}.save[i]$ and ($u = 0$ or $i \geq u$) then
5          $t_i[:] \leftarrow \mathcal{P}^{(i)}[\mathcal{T}.ptr[i] + th$,:]
6      if i<u then
7          $k_i[:] \leftarrow k_{i-1}[:] \odot A^{(i)}$[idx,:]
8      if $\neg\mathcal{T}.save[i]$ or u>i or u = 0 then
9          if $i < d-1$ then
10             MTTKRP-LOOP($i$+1,idx,$k_i$, $t_i$,$\mathcal{T}$, $\mathbf{A}$,$u$,$th$,$\bar{A}^{(u)}$)
11         else
12             $Tval \leftarrow \mathcal{T}.val[idx]$
13             if $u = d-1$ then
14                 $A^{(d-1)}$[idx,:] $\leftarrow A^{(d-1)}$[idx,:] + Tval *
                   $k_{i-1}[:]$
15             else
16                 $t_{i-1}[:] \leftarrow t_{i-1}[:]$ + Tval * $A^{(d-1)}$[idx,:]
17     if $i = u$ then
18         $\bar{A}^{(i)}$[idx,:] $\leftarrow \bar{A}^{(i)}$[idx,:] +$k_{i-1}[:] \odot t_i[:]$
19     else if $i > u$ then
20         $t_i[:] \leftarrow t_{i+1}[:] \odot A^{(i)}$[idx,:]
```

Algorithm 6: STeF MTTKRP to compute $A^{(1)}$ for a 4D tensor $\mathcal{T}$ where $\mathcal{P}^{(1)}$ is stored.

```
input :  Sparse tensor $\mathcal{T}$, factorization matrices
         $A^{(0)}$,$A^{(1)}$,$A^{(2)}$ and $A^{(3)}$, number of factors $R$
output:  Updated factorization matrix $\bar{A}^{(1)}$
1  for $i \in \mathcal{T}$[*,:,:,:] do in parallel
2      $k_0 \leftarrow A^{(0)}$[i,:]
3      for $j \in \mathcal{T}$[i,*,:,:] do
4          $t_1 \leftarrow \mathcal{P}^{(1)}$[i,j]
5          $\bar{A}^{(1)}[j,:] \leftarrow \bar{A}^{(1)}[j,:] + t_1[:] \odot k_0[:]$
6  if Is $\bar{A}^{(1)}$ privatized for each thread? then
7      Reduce $\bar{A}^{(1)}$
```

might want to access the same location for a write operation. In the implementation, each thread uses a localised copy of $\mathcal{P}^{(i)}$ array by shifting its write location by an amount equal to its thread id. More specifically, if thread th and $th+1$ want to write to the same data location s, one of them writes $s+th$ and the other writes $s+(th+1)$, hence avoiding a race condition.

Algorithm 6 implements Figure 1b where partial MTTKRP for $\mathcal{P}^{(1)}[i_0, i_1$,r] is stored. Using the partial results, we can compute MTTKRP for $A^{(1)}$ using a single MTTV operation. Algorithm 7 describes the case in Figure 1c where $A^{(2)}$ needs to be contracted. First, $A^{(2)}$ is contracted with $\mathcal{P}^{(2)}[i_0, i_1, i_2$,r], the partial MTTKRP result, and the MTTV operation is performed to get updated values of $A^{(1)}$. Finally, Algorithm 8 implements Figure 1d in which none of the results are saved; thus, the entire CSF structure is traversed to compute the MTTKRP for $A^{(1)}$.

Algorithm 7: STeF MTTKRP to compute $A^{(1)}$ for a 4D tensor $\mathcal{T}$ where $\mathcal{P}^{(2)}$ is stored.

```
input :   Sparse tensor $\mathcal{T}$, factorization matrices
          $A^{(0)}$,$A^{(1)}$,$A^{(2)}$ and $A^{(3)}$, number of factors $R$
output:   Updated factorization matrix $\bar{A}^{(1)}$
1  for $i \in \mathcal{T}[*,:,:,:]$ do in parallel
2      $k_0 \leftarrow A^{(0)}$[i,:]
3      for $j \in \mathcal{T}[i,*,:,:]$ do
4          $t_1$[:] $\leftarrow$ zero vector of length R
5          for $k \in \mathcal{T}[i,j,*,:]$ do
6              $t_2$[:] $\leftarrow$ T2[i,j,k]
7              $t_1[:] \leftarrow t_1[:] + t_2[:] \bigodot A^{(2)}[k,:]$
8          $\bar{A}^{(1)}[j,:] \leftarrow \bar{A}^{(1)}[j,:] + t_1[:] \bigodot k_0[:]$
9  if Is $\bar{A}^{(1)}$ privatized for each thread? then
10     Reduce $\bar{A}^{(1)}$
```

Algorithm 8: STeF MTTKRP to compute $A^{(1)}$ for 4D tensor $\mathcal{T}$ where no partial results are stored.

```
input :   Sparse tensor $\mathcal{T}$, factorization matrices
          $A^{(0)}$,$A^{(1)}$,$A^{(2)}$ and $A^{(3)}$, number of factors $R$
output:   Updated factorization matrix $\bar{A}^{(1)}$
1  for $i \in \mathcal{T}[*,:,:,:]$ do in parallel
2      $k_0 \leftarrow A^{(0)}$[i,:]
3      for $j \in \mathcal{T}[i,*,:,:]$ do
4          $t_1$[:] $\leftarrow$ zero vector of length R
5          for $k \in \mathcal{T}[i,j,*,:]$ do
6              $t_2$[:] $\leftarrow$ zero vector of length R
7              for $l \in \mathcal{T}[i,j,k,*]$ do
8                  $Tval \leftarrow \mathcal{T}[i,j,k,l]$
9                  $t_2[:] \leftarrow t_2[:] + Tval * A^{(3)}[l,:]$
10             $t_1[:] \leftarrow t_1[:] + t_2[:] \bigodot A^{(2)}[k,:]$
11         $\bar{A}^{(1)}[j,:] \leftarrow \bar{A}^{(1)}[j,:] + t_1[:] \bigodot k_0[:]$
12 if Is $\bar{A}^{(1)}$ privatized for each thread? then
13     Reduce $\bar{A}^{(1)}$
```

IV. Models

This section motivates the need to model the impact of memoization and mode order switching on data movement. We also develop an analytical model to quantify the expected data movement and, thereby, the performance.

A. Saving Partial Results

As shown in Section I, a single CPD iteration involves multiple MTTKRPs, some of which share the same intermediate results. Rather than recomputing the intermediate results for each MTTKRP, they could be computed once and reused thereafter; this strategy is called memoization. While memoization helps to reduce the total number of computations, it introduces additional memory write and read operations to save and load the intermediate results. Depending on the operand sizes, this additional data movement can overwhelm the savings from the reduced number of operations. Consider for example, the *uber* tensor described in Table I. Saving all the intermediate results for the decomposition of this tensor requires 62M reads and 22M writes. However, not saving the biggest partial result will result in 24M reads and 238K writes, which makes the decomposition faster.

In contrast for vast-2015-mc1-3d tensor, saving the intermediate tensor results in 1.7B reads and 833M writes. Not saving them on the other hand, will result in a total read and write count of 2.6B and 833M, respectively. In this case therefore, saving is beneficial.

B. Switching Order of Last Two Modes

Algorithm 9: Finding number of fibers if last two modes of the CSF representation of $\mathcal{T}$ are swapped

```
input :   Sparse tensor $\mathcal{T}$
output:   Number of fibers in 1-2-4-3 order
1  for th=0 to number of thread - 1 do in parallel
2      for row = 0 to $n_3$-1 do
3          observed[th][row] $\leftarrow$0
4      num_fibers[th] $\leftarrow$0
5  for $i \in \mathcal{T}[*,:,:,:]$ do in parallel
6      th $\leftarrow$ thread id
7      for $j \in \mathcal{T}[i,*,:,:]$ do
8          for $k \in \mathcal{T}[i,j,*,:]$ do
9              for $l \in \mathcal{T}[i,j,k,*]$ do
10                 if $observed[th][l] \neq (i,j)$ then
11                     $observed[th][l] \leftarrow (i,j)$
12                     Increment num_fibers[th] by 1
13 number_fibers $\leftarrow$ 0
14 for th=0 to number of thread - 1 do
15     number_fibers $\leftarrow$ number_fibers + num_fibers[th]
```

As mentioned in Section II-E, switching the order of the last two modes impacts the data movement and hence the performance. To determine whether to switch or not, we use a data movement model that requires the total number of fibers at each level. The CSF format inherently contains the total number of fibers for the original (non switched order). However, this count is not available for the switched order. Creating a new CSF format to count the fiber information for the switched-mode will increase the overhead of decision-making.

Algorithm 9 efficiently computes the fiber information for the switched order. Note that, since $d-2$ modes are the same for both original and switched order, the number of fibers for them is the same. Hence, we only need to compute the number of non-zeros after the first contraction of MTTKRP in the switched order. In Algorithm 9, we use a buffer of size $num\_threads * n_3$, where n_3 is the length of the third dimension of the tensor. Each thread initializes its portion of the buffer to 0 (Line 4). Line 5 distributes the root mode of the tensor across multiple threads. In Line 10, each thread checks whether it has previously seen the $< i, j, l >$ pair. If an pair was seen for the first time, we increment the fiber count (Line 12). Line 15 aggregates the fiber count across multiple threads.

C. Data Movement Model

Let x be the total number of read access to the i^{th} factorization matrix of size $N_i * R$. Each such access reads R elements (ie. one row) of the i^{th} factorization matrix. If the data footprint of the matrix is less than or equal to the cache size, each element will be loaded only once to the cache (cold miss) and reused thereafter. Otherwise, each x_i access will read R elements from the memory without reuse. Hence the data movement associated with i^{th} factorization matrix ($DM\_factor_i(x)$) can be computed as follows:

$$DM\_factor_i(x) = \begin{cases} x * R & N_i * R > cachesize \\ min(N_i * R, x * R) & otherwise \end{cases}$$

Let m_i be the number of fibers at level i. Assuming that there is no memoization, for all modes, the read cost can be expressed as:

$$DM\_no\_mem_{read}(i) = \sum_{i=0}^{d-1}(2 * m_i + DM\_factor_i(m_i))$$

If memoization is enabled, we can avoid the reads required to compute the intermediate tensor after the initial MTTKRP to compute them, but we must read the partial results. Let k be the first mode that has been memoized where $0 < i \leq k$. The corresponding read cost can be expressed as:

$$DM\_mem\_k_{read}(i) = \sum_{i=0}^{k-1}(2*m_i+DM\_factor_i(m_i)+m_i*R)$$

For mode 0 we may need additional writes if some partial MTTKRP results have been memoized. The volume of data movement for mode 0 is modeled as:

$$DM_{read}(0) = DM\_no\_mem_{read} + \sum_{i \in \mathbf{M}} m_i * R$$

where $\mathbf{M}$ is the set of modes that have been memoized after MTTKRP of mode 0.

The write cost corresponding to mode $0 < i \leq d-1$ can be expressed as:

$$DM_{write}(i) = DM\_factor_i(m_i)$$

The write cost for mode 0 can be written as:

$$DM_{write}(0) = n_0 * R + \sum_{i \in \mathbf{M}} m_i * R$$

Combining these we get the following for $i > 0$:

$$DM_{read}(i) = \begin{cases} DM\_mem\_k_{read}(i) & \exists k \ s.t. \ k \in \mathbf{M} \ and \ k \geq i \\ DM\_no\_mem_{read}(i) & otherwise \end{cases}$$

Adding $DM_{read}(i)$ and $DM_{write}(i)$, we can estimate the total data movement for the MTTKRP operation for updating $A^{(i)}$. Hence, the total data movement estimation for a single iteration of CPD will be:

$$Total \ Data \ Movement = \sum_{i=0}^{d-1} DM_{read}(i) + DM_{write}(i)$$

Note that if the order of the modes and/or the list of memoized partial MTTKRP results $\mathbf{M}$ is changed, DM_{read} and DM_{write} will change and the total data movement estimation will change. Our model exhaustively checks every configuration to select the one with the lowest data movement estimate.

V. Related Work

Smith et al. [2] proposed SPLATT which can compute CPD for tensor with using only a single CSF representation. This has also been extended to accelerate distributed tucker decomposition in [11]. Baskaran et al. [8] propose a split-reorder of the mode-contraction loop in the Tucker decomposition algorithm, such that partially contracted tensors can be re-used in an optimal manner. The trade-off between storing partially contracted sparse tensors vs recomputing them, seems to have originated in Kolda et al [12], again in the context of Tucker decomposition. Kaya et al [3] proposed a novel sparse tensor representation for the CP decomposition algorithm. The Balanced Dimension Tree recursively partitions the set of modes into two halves. Each leaf hence contains a single mode. Each of the n mode contractions therefore involves a walk from the corresponding leaf to all internal nodes, via the root. Since several internal nodes are common among consecutive walks, the partially decomposed tensors corresponding to these nodes can be stored in memory. The only restriction for reuse is that the factor matrices are updated in sequential order. The partially decomposed tensors that used these factor matrices hence have to be recomputed. While this work focuses on the distributed memory setting, it also exploits intra-node shared memory parallelism. The corresponding *HyperTensor* [13] library implementation has not yet been released to open-source, making an empirical comparison impossible for this work.

Li et al. [4] proposed a similar approach to choose different memoizations based on a model. They introduce a new storage format *vCSF* to store the sparse tensor. For the n MTTKRP operations in total, they propose to store and reuse $\theta(\sqrt{N})$ partially contracted tensors. Coupled with mode reordering, this scheme essentially uses a CSF forest with the lower half of each CSF tree stored in memory for reuse. The implementation is named $AdaTM$, and has been used as one of the baselines for performance comparison.

More recently, Helal et al. [5] proposed a new compressed format called ALTO for storing tensors. This approach offers better load balancing and locality while performing a single MTTKRP. Furthermore, it also avoids the need to change the tensor representation for performing the different MTTKRPs in a CPD iteration. The storage structure uses a novel indexing representation for the nonzero elements of a sparse tensor that is formed by permuting a compacted binary representation of concatenated multi-dimensional index coordinates of tensor elements. The work currently computes all mode contractions from scratch, and hence has a significantly higher FLOP count. We also compare performance with ALTO in Section VI.

Li et al. have also proposed mode reordering algorithms $BFS-MCS$ and $Lexi-Order$ [6]. The $Lexi-Order$ seems to consistently outperform the hypergraph partitioning-based $BFS-MCS$. As per $Lexi-Order$, the lower-stride modes of the given ordering are sorted in increasing order of co-ordinates. It can easily be applied to COO, HiCOO or CSF representations in memory, and seems to improve speedup

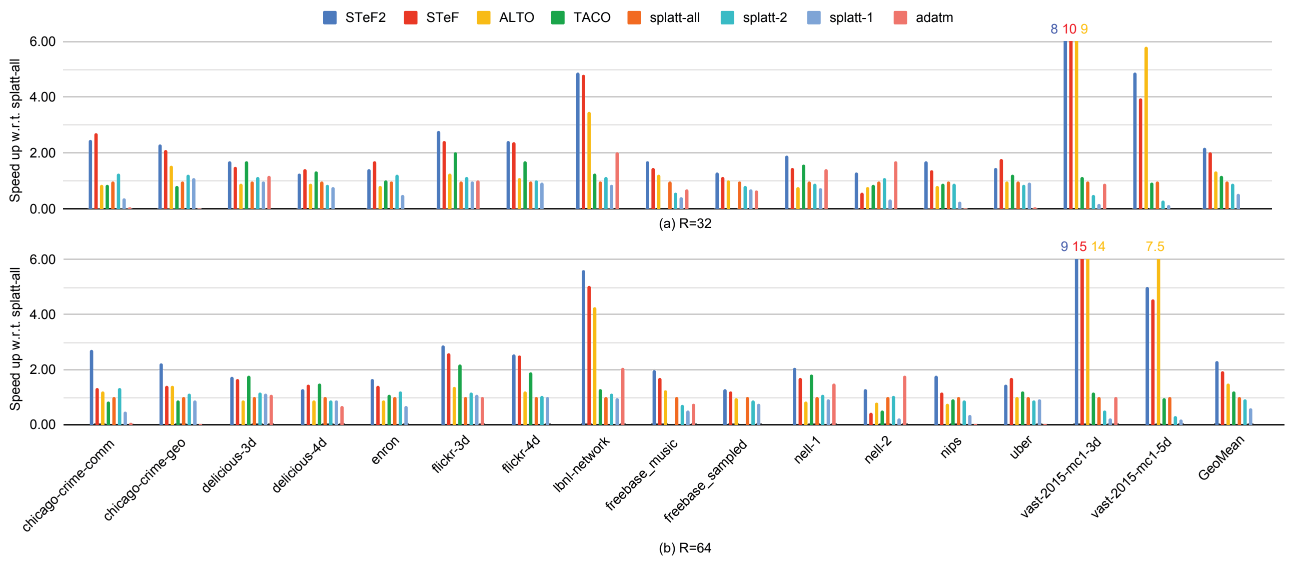


Fig. 3: Results for number of factors = 32 and 64 for 18 core Intel Cascade Lake Processor (higher is better).

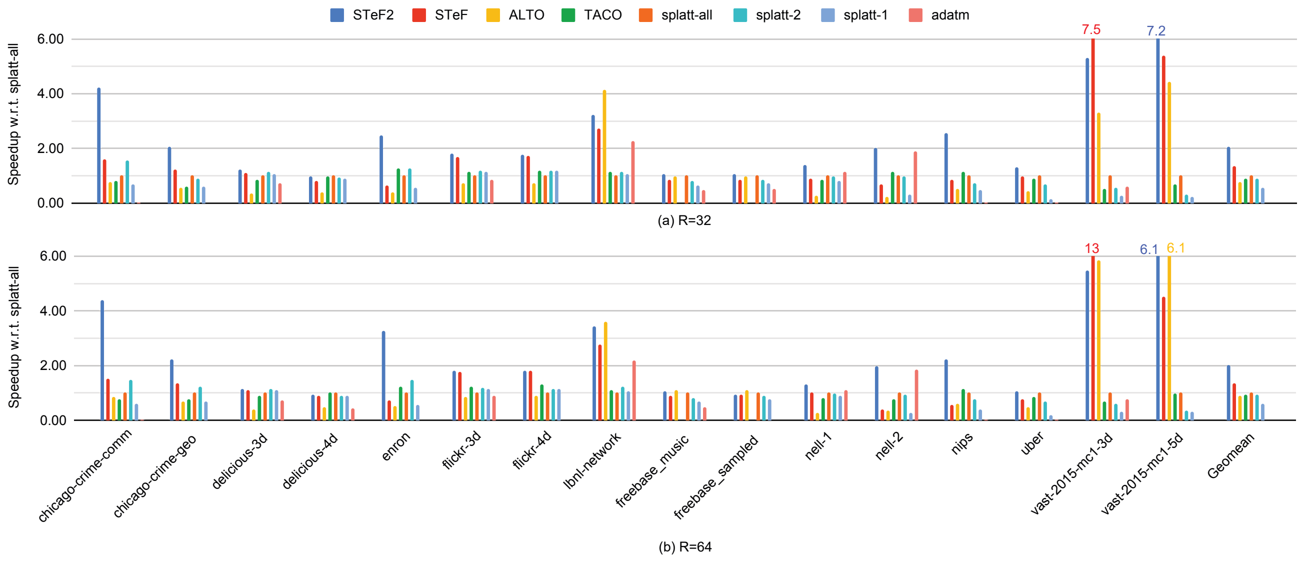


Fig. 4: Results for number of factors = 32 and 64 for 64 core AMD Threadripper Processor (higher is better)

significantly in each case. For the CSF-based implementation, this seems to be complimentary to our contributions viz model-based storage of intermediate results.

Shivakumar et al. [14] propose a new compressed sparse symmetric CSS structure specifically for sparse symmetric tucker decomposition. This format exploits the constraint of a single factor matrix, and a core tensor with all dimensions of equal length. While this pushes the pareto front of runtime vs memory, this method currently is restricted to symmetric decomposition only.

The TACO compiler has integrated several optimizations for tensor decomposition and hence simplified the search across the entire optimization space, including the choice of mode orderings [10] [15] [16]. We consider the latest scheduling TACO implementation as one of our baselines for performance comparison.

Ballard and Rouse et al. [17] establishes lower bound for communication in MTTKRP operation and provide an algorithm achieving the bound for dense case. Nisa et al. [18] focuses on load balancing the MTTKRP operation on massively parallel systems such as GPUs by creating Hybrid Balanced CSF format (HB-CSF). Nguyen et al [19] proposes

BLCO format to tackle sparse MTTKRP on GPUs as a continuation from ALTO [5]. Li et al. [20] proposes BDS-MCS reordering algortihm to improve locality of the MTTKRP operation.

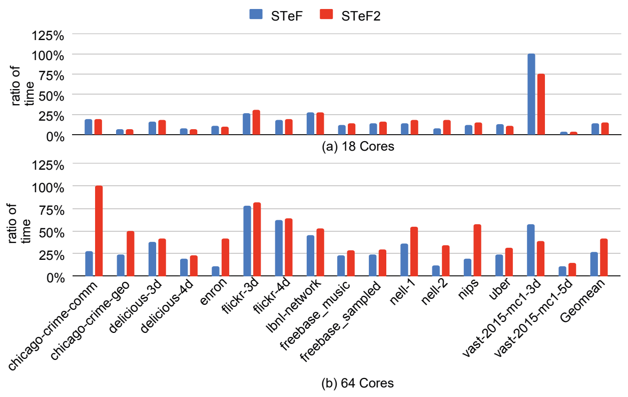


Fig. 5: Preprocessing overhead for calculating whether to switch orders of the last two modes or not for 18 Core Intel and 64 Core AMD machines. Each bar represents preprocessing overhead to one set of MTTKRP operations.

VI. Experimental Evaluation

A. Experimental Setup

The experimental evaluation was performed on two systems: an 18 core Intel i9-10980XE (Cascade Lake) and a 64 core AMD 3990X (Threadripper) processor, using GCC v9.3.0. Both systems have 128 GB memory and used Ubuntu 20.04.2 LTS. For benchmarking, we used the same set of tensors that have been used in previous studies on sparse tensor factorization, from the FROSTT [21] and HaTen2 [22] datasets. The properties of tensors used in the experiments are summarized in Table I. Experiments were performed for two values for the rank: R=32 and R=64.

B. Performance Comparison

We compared STeF against multiple state-of-the-art algorithms – AdaTM [4], ALTO [5], SPLATT 2.0.0 [9] and TACO [10]. SPLATT has multiple versions for a single CPD iteration, where the number of distinct copies of the tensor is one, two, or as many as the number of dimensions of the tensors; we called these variants splatt-1, splatt-2, and splatt-all. In addition to STeF, which only requires a single CSF representation of the sparse tensor, we also report performance for a variant of STeF called STeF2 that uses an additional copy of the tensor with a different CSF layout. The additional CSF representation uses a layout where the root mode is the leaf mode in the base CSF representation used with STeF. The rationale for this choice is that the MTTV operation needed to perform the MTTKRP operation corresponding to the leaf mode of the base CSF representation can be quite expensive and using a second CSF representation with that mode as the root allows a more efficient implementation using TTM and mTTV operations. ALTO has 2 different implementations using either 64 or 128 bits indices. We report performance data for the faster version of two for each tensor.

TABLE I: List of tensors used in the experiments

Tensor	Dimensions	NNZ
chicago-crime-comm	6Kx24x77x32	5M
chicago-crime-geo	6Kx24x380x395x32	6M
delicious-3d	533Kx17Mx2M	140M
delicious-4d	533Kx17Mx2Mx1K	140M
enron	6Kx6Kx244Kx1K	54M
flickr-3d	320Kx28Mx2M	113M
flickr-4d	320Kx28Mx2Mx731	113M
freebase_music	23Mx23Mx166	100M
freebase_sampled	38Mx38Mx533	100M
lbnl-network	2Kx4Kx2Kx4Kx868K	2M
nell-1	3Mx2Mx25M	144M
nell-2	12Kx9Kx29K	77M
nips	2Kx3Kx14Kx17	3M
uber	183x24x1Kx2K	3M
vast-2015-mc1-3d	165Kx11Kx2	26M
vast-2015-mc1-5d	165Kx11Kx2x100x89	26M

Figures 3 and 4 compare the performance of all the algorithms relative to splatt-all. STeF achieves 437%, 50%, 180%, 77%, 59% and 55% geometric mean speed-up over AdaTM, ALTO, splatt-1, splatt-2, splatt-all and TACO respectively on average, across both machines and the two R values. STeF2 achieves a higher average speedup of 603%, 93%, 270%, 137%, 114% and 103%, respectively, over AdaTM, ALTO, splatt-1, splatt-2, splatt-all and TACO, respectively, on the two machines and the two R values.

STeF and STeF2 show consistent and often significant speedup over all the methods (except ALTO, for vast-2015-mc1-3d and vast-2015-mc1-5d tensors) on both machines due to its very work distribution. splatt, AdaTM and TACO perform work distribution by assigning each thread a slice of contiguous root-mode indices, whereas STeF and STeF2 performs load balancing across the leaf non-zeros. On tensors with adequate slices for load-balanced work distribution, such as flickr-3d and flickr-4d, ALTO and splatt achieve lower performance STeF due to not memoizing and saving operations. Even though AdaTM uses memoizing, it fails to select an optimal mode order or memoizing decisions.

STeF performance is lower than AdaTM, splatt-2, splatt-all and TACO for the nell-2 tensor on both machines. This is mainly attributable to a slow MTTV kernel for the leaf mode. STeF2 closes this gap because it uses a second CSF representation, avoiding the need to use the MTTV kernel. TACO performs better in delicious-3d and nell-1, where the length of the modes are too different and benefits from the saving the intermediate results are not significant. Overall, TACO is better than splatt-all even though they are very similar methods. The main reason is that TACO uses auto-tuning across various chunk sizes and selects the best, paying a small preprocessing overhead for faster run time. For chicago-crime-comm and chicago-crime-geo, the longest mode has length of 6K. For $R = 32$, the factor matrix can be cached in the Intel machine but not for $R = 64$. Therefore, there is a sharp slow down for STeF in these tensors in Figure 3. A similar pattern occurs with the nips and nell-2 tensors on the

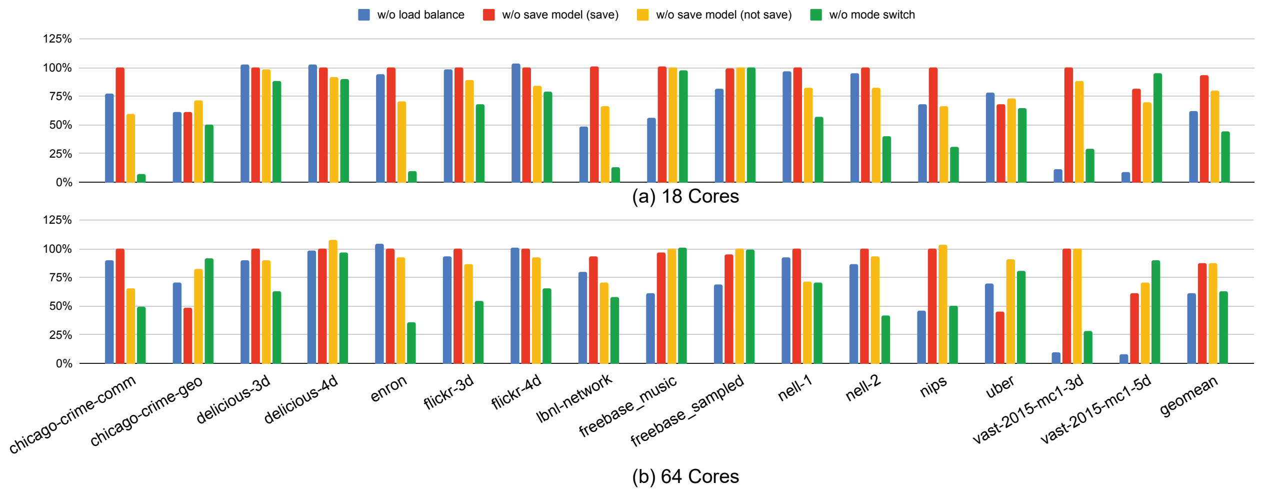


Fig. 6: Ablation study for different optimizations presented in this paper on an 18 core Intel Cascade Lake and 64 Core AMD Threadripper processor, for R=32. Performance is normalized with respect to the model-chosen configuration. 100% means same performance as the model-chosen configuration - below 100% means slower without that optimization.

AMD machine. Since the cache sizes and cache structures are different this phenomenon happens with different tensors on different machines. For the majority of the tensors, STeF and STeF2 perform better than the state-of-the-art solutions and STeF2 achieves over 25% speedup on average with respect to the best among all the state-of-the-art approaches.

C. Preprocessing Overhead

Figure 5 shows the preprocessing overhead associated with finding the number of fibers in different modes and determining whether we should use switch order or not (Algorithm 9). On average, the preprocessing overhead is 19% and 25% of the parallel execution time for a set of MTTKRP operations for a single CPD iteration, for $R = 32$, for Intel and AMD machines, respectively. For $R = 64$, the average overhead is 10% and 14% of the time for a single CPD iteration, respectively, for the Intel and AMD machines. The maximum ratio is below 100% for all benchmarks. Typically CPD algorithms require many iterations to converge; Since preprocessing overhead is less than a single CPD iteration, preprocessing for finding the fiber counts to determine the order of the last two modes is essentially negligible.

D. Additional Space Requirement

Table II shows the additional space requirement to use the memoization technique of STeF, for $R = 32$ and $R = 64$. For a given R value space requirement is the same for any machine. Overall, additional space requirement for ranks 32 and 64 for the partial MTTKRP results is on average 35% and 45% of the space requirement for the CSF structure and the factorization matrices and this ratio is at most 2.34. This low space requirement for STeF is partly due to selective memoization of the partial MTTKRP results using the model. The model tends to not memoize if memoizing a partial result would increase the total data movement volume. This also helps avoid large penalties for storing partial MTTKRPs. For example, in chicago-crime-comm the ratio of partial MTTKRP compared CSF and factorization matrices would have been 5.43 if all the partial MTTKRP have been saved. As R doubles, the size of partial MTTKRP and factorization matrices doubles as well; however, the size of the CSF structure stays constant. Therefore, the overhead ratio increases slightly with increase in R.

TABLE II: Space requirement for storing the partial MTTKRP results.

	Size of stored partial MTTKRPs (GB)		Size of Tensor and Factorization Matrices (GB)		Ratio	
Tensor	R=32	R=64	R=32	R=64	R=32	R=64
chicago-crime-comm	0.01	0.02	0.08	0.08	0.13	0.25
chicago-crime-geo	0.04	0.08	0.15	0.15	0.29	0.57
delicious-3d	8.92	17.85	7.49	12.32	1.19	1.45
delicious-4d	14.84	29.68	7.85	12.68	1.89	2.34
enron	0.22	0.44	0.88	0.94	0.25	0.47
flickr-3d	3.18	6.36	9.06	16.23	0.35	0.39
flickr-4d	6.30	12.59	9.25	16.42	0.68	0.77
freebase_music	0.00	0.00	13.51	24.64	0.00	0.00
freebase_sampled	0.00	0.00	21.94	40.52	0.00	0.00
lbnl-network	0.01	0.02	0.24	0.45	0.05	0.05
nell-1	4.14	8.28	9.71	16.99	0.43	0.49
nell-2	0.08	0.16	1.16	1.17	0.07	0.14
nips	0.00	0.00	0.05	0.06	0.04	0.07
uber	0.00	0.00	0.06	0.06	0.02	0.03
vast-2015-mc1-3d	0.06	0.13	0.43	0.48	0.14	0.26
vast-2015-mc1-5d	0.00	0.00	0.64	0.68	0.00	0.00
Average	2.36	4.73	5.16	8.99	0.35	0.45

E. Ablation Study

Figure 6 show the effectiveness of the optimization presented in this paper. We chose the model selected configuration as the baseline configuration. We only present data for $R = 32$ since results for $R = 64$ were very similar.

1) Work Distribution: To compare the effectiveness of our work distribution scheme, we compare the performance of our model selection configuration with and without our load balancing. Turning off our work distribution results in an average slow down of 39% on MTTKRP performance for both Intel and AMD machines. There are five cases across the two machines where our work distribution results in slightly lower performance than use of slice-based parallelism. While our work distribution ensures each thread processes an equal number nodes at the leaf level, it does not guarantee perfect load balancing at the non-leaf levels. Usually the amount of work per-level rapidly decreases as we move up the tree and any load-imbalance at the higher levels does not have much of an impact. But for a few tensors, where slice-based parallelization has sufficient parallelism at the root mode for effective load-balancing, it performs better because it performs load-balancing based on the whole forest and not just the work at the leaf nodes.

2) Saving Partial MTTKRPs: To measure the impact of our model, which predicts whether to save the intermediate results or not, we compare our results against two extreme choices (i) save all intermediate results, and (ii) don't save any intermediate results. As shown in Figure 6, our model has a significant impact on 3 of the tensors, and on average, it speeds up MTTKRP by 12% and 13% for the Intel and AMD machine, respectively. Turning off this optimization did not give more than 5% speed up on any case.

3) Switching Mode Order: Our model also predicts whether we should switch the order of the last two modes or not. In order to assess this impact, we compare our approach against an approach that chooses the opposite choice as that of our model. As shown in Figure 6 this choice is significant for many tensors, and the average slowdown for the opposite choice configuration is 55% and 37%, respectively, for the Intel and AMD machines. The model for choosing mode order selected the worse order in only one case, with a performance difference of 1%.

VII. Conclusion

In this paper, we develop a data movement-aware MTTKRP algorithm targeted at applications like CPD. Our proposed solution achieves good load balancing and reduced data movement using a single CSF representation of the tensor. We explore the design space of possible configurations for MTTKRP and develop an efficient model to select a good configuration. Our experimental section demonstrates the superiority of our approach compared to the state-of-the-art methods.

Acknowledgments

We thank the reviewers for their numerous suggestions to improve the paper. This work was supported in part by the by the U.S.National Science Foundation through award 2009007.

References

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

[2] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 61–70.

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

[4] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, "Model-driven sparse CP decomposition for higher-order tensors," in *2017 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2017, pp. 1048–1057.

[5] A. E. Helal, J. Laukemann, F. Checconi, J. J. Tithi, T. Ranadive, F. Petrini, and J. Choi, "ALTO: adaptive linearized storage of sparse tensors," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 404–416.

[6] J. Li, B. Uçar, U. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, "Efficient and effective sparse tensor reordering," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 227–237.

[7] S. Eswar, K. Hayashi, G. Ballard, R. Kannan, M. A. Matheson, and H. Park, "PLANC: Parallel low-rank approximation with nonnegativity constraints," *ACM Trans. Math. Softw.*, vol. 47, no. 3, Jun. 2021.

[8] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin, "Efficient and scalable computations with sparse tensors," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–6.

[9] S. Smith and G. Karypis, "SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit," http://cs.umn.edu/ splatt/, 2016.

[10] R. Senanayake, C. Hong, Z. Wang, A. Wilson, S. Chou, S. Kamil, S. Amarasinghe, and F. Kjolstad, "A sparse iteration space transformation framework for sparse tensor algebra," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[11] V. T. Chakaravarthy, S. S. Pandian, S. Raje, and Y. Sabharwal, "On optimizing distributed non-negative tucker decomposition," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 238–249.

[12] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *ICDM 2008: Proceedings of the 8th IEEE International Conference on Data Mining*, 2008, pp. 363–372.

[13] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–11.

[14] S. Shivakumar, J. Li, R. Kannan, and S. Aluru, "Efficient parallel sparse symmetric tucker decomposition for high-order tensors," in *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 2021, pp. 193–204.

[15] S. Chou, F. Kjolstad, and S. Amarasinghe, "Automatic generation of efficient sparse tensor format conversion routines," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 823–838.

[16] S. Mueller, P. Ahrens, S. Chou, F. Kjolstad, and S. Amarasinghe, "Sparse tensor transpositions," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 559–561.

[17] G. Ballard, N. Knight, and K. Rouse, "Communication lower bounds for matricized tensor times khatri-rao product," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 557–567.

[18] I. Nisa, J. Li, A. Sukumaran-Rajam, R. Vuduc, and P. Sadayappan, "Load-balanced sparse mttkrp on gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 123–133.

[19] A. Nguyen, A. E. Helal, F. Checconi, J. Laukemann, J. J. Tithi, Y. Soh, T. Ranadive, F. Petrini, and J. W. Choi, "Efficient, out-of-memory sparse mttkrp on massively parallel architectures," *arXiv preprint arXiv:2201.12523*, 2022.

[20] J. Li, B. Uçar, Ü. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, "Efficient and effective sparse tensor reordering," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 227–237.

[21] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis, "FROSTT: The formidable repository of open sparse tensors and tools," *Chicago Univ., Chicago, IL, USA, Tech. Rep*, 2017.

[22] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 1047–1058.