A Performance Portable Matrix Free Dense MTTKRP in GenTen

Gabriel Kosmacher
Oden Institute
The University of Texas at Austin
Austin, TX USA
gkosmacher@utexas.edu

Eric T. Phipps
Sandia National Laboratories
Albuquerque, NM USA
etphipp@sandia.gov

Sivasankaran Rajamanickam
Sandia National Laboratories
Albuquerque, NM USA
srajama@sandia.gov

***Abstract*—We extend the GenTen tensor decomposition package by introducing an accelerated dense matricized tensor times Khatri-Rao product (MTTKRP), the workhorse kernel for canonical polyadic (CP) tensor decompositions, that is portable and performant on modern CPU and GPU architectures. In contrast to the state-of-the-art matrix multiply based MTTKRP kernels used by Tensor Toolbox, TensorLy, etc., that explicitly form Khatri-Rao matrices, we develop a matrix-free element-wise parallelization approach whose memory cost grows with the rank R like the sum of the tensor shape $\mathcal{O}(R(n+m+k))$, compared to matrix-based methods whose memory cost grows like the product of the tensor shape $\mathcal{O}(R(mnk))$. For the largest problem we study, a rank 2000 MTTKRP, the smaller growth rate yields a matrix-free memory cost of just 2% of the matrix-based methods, a $50\times$ improvement. In practice, the reduced memory impact means our matrix-free MTTKRP can compute a rank 2000 tensor decomposition on a single NVIDIA H100 instead of six H100s using a matrix-based MTTKRP. We also compare our optimized matrix-free MTTKRP to baseline matrix-free implementations on different devices, showing a $2\times$ single-device speedup on an Intel 8480+ CPU, an $11\times$ speedup on an NVIDIA H100 GPU and a $6\times$ speedup on an AMD MI300A GPU. In addition to numerical results, we provide fine grained performance models for an ideal multi-level cache machine, compare analytical performance predictions to empirical results, and provide a motivated heuristic selection for selecting an algorithmic hyperparameter.**

***Index Terms*—tensor decomposition, canonical polyadic, MTTKRP, Kokkos, GPU**

I. Introduction

Dense tensors are relevant in the fields of numerical simulations [1], medical imaging [2], signal processing [3], and color perception [4], among others. Low-rank approximations of dense tensors are powerful tools used to compress such datasets and reveal relationships between modes. A popular decomposition for low-rank approximations is the CANDECOMP/PARAFAC (CP) decomposition, also called canonical polyadic decomposition [5]; the decomposition takes as input a user-supplied rank R and outputs a weighted sum of R rank-1 tensors. The bottleneck kernel for CP algorithms [6] is the matrix action of a mode-k unfolding of a tensor with the Khatri-Rao product of factor matrices and is called the *matricized tensor times Khatri-Rao product* (MTTKRP).

Most open-source tensor decomposition packages (see Section I-A) compute the MTTKRP by explicitly forming the Khatri-Rao product matrix, allowing invocations of BLAS-3 algorithms, but *compromising memory usage.* In contrast, we seek to compute the MTTKRP *without* explicitly forming the Khatri-Rao matrix and *instead* take advantage of the problem structure to develop *matrix-free* (also called *element-based*) algorithms. Our work extends the open-source GenTen[1] tensor decomposition package with an implementation of our dense matrix-free MTTKRP and, *as far as we know, introduces the first GPU algorithm for a matrix-free MTTKRP with dense tensors.*

Contributions: Our work offers the following methodological contributions and experimental results:

- The design and open-source implementation of performance-portable matrix-free algorithms for dense MTTKRPs (see Section III). For a tensor $\mathcal{Y}$ of shape $I_1, \ldots, I_d$ and CP-rank R, our matrix-free algorithms for a mode-k MTTKRP have $\mathcal{O}(R\sum_n I_n)$ memory usage, while standard matrix-based approaches have $\mathcal{O}(R\prod_{n\neq k} I_n)$ memory cost.
- Multiple analytical memory and computational models for our algorithmic variants based on an ideal multi-level cache machine with different caching assumptions (see Section IV-B).
- A heuristic model for an algorithmic hyper-parameter that aligns with measured times on a CPU and GPU (see Section IV-C1).

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. SAND2025-13211O.

[1]https://github.com/sandialabs/GenTen

- CPU and GPU algorithmic evaluations (see Sections IV-D1 and IV-D2). In particular, our matrix-free MTTKRP can perform a tensor decomposition on one GPU that requires six GPUs for a matrix-based MTTKRP. We also compare our optimized, matrix-free MTTKRP to baseline matrix-free implementations, showing a 2× speedup on an Intel 8480+ CPU, an 11× speedup on an NVIDIA H100 GPU, and a 6× speedup on an AMD MI300A GPU.

A. Related work

A variety of open-source packages offer dense CP decompositions. N-Way Toolbox [7], TensorLy [8], and rTensor [9] compute MTTKRPs by explicitly unfolding the tensor and forming the Khatri-Rao product. The MATLAB Tensor Toolbox [6, 10], PyTTB [11] and GenTen [12] use an algorithm introduced by Phan et al. [13], described in Section III-A, that avoids expensive tensor unfolding and reduces memory impact by forming *partial* Khatri-Rao matrices. Vannieuwenhoven et al. [14] build upon [13] by constructing an algorithm, implemented using the Eigen3 C++ library [15], that sequentially stores blocks of the Khatri-Rao matrix in constant memory, hence achieving the desired $\mathcal{O}(R\sum_n I_n)$ storage complexity. *However*, the blocking algorithm is only applicable to the special case where all k-mode MTTKRPs are computed at once, and cannot generalize to compute individual mode-k MTTKRPs needed by, e.g., the CP-ALS algorithm. Table I gives an overview of features for different dense mode-k MTTKRP algorithms and codebases.

There has been considerably more work on optimizing MTTKRPs in the sparse tensor case. DeFacTo [16] stores tensors as collections of sparse matrices and relies on optimized sparse matrix-vector multiply routines to compute the MTTKRP. HiCOO [17] proposes a new storage format that partitions the sparse tensor into sparse blocks to increase memory bandwidth and develops an algorithm that parallelizes over groups of sparse blocks. Both SPLATT and HiCOO were developed primarily for low-thread-count CPUs and do not consider portability to high-thread GPU architectures. Laukemann et al. [18] introduce another sparse tensor format, ALTO, that linearizes the tensor's multi-index set such that tensor entries close in space are close in memory. Entries are then decomposed into *line segments* that encode subspaces, and a parallel algorithm is introduced that assigns threads to chunks of line segments. Though developed for CPUs, increasing the number of chunks per line segment *should* allow for straightforward portability to GPUs. GenTen [19] offers a performance-portable approach that parallelizes over tensor nonzeros and permutes the nonzeros to avoid atomic contention. This algorithm is similar to ours (in the dense case) as it avoids forming the Khatri-Rao product explicitly. However, permuting the tensor nonzeros requires an additional storage cost of $\mathcal{O}(dN)$, where d is the dimension of the tensor and N is the number of nonzeros. As we show in Section IV-B, explicitly permuting tensor entries in such a fashion is *not* necessary for an efficient dense MTTKRP.

II. Background

A. Notation

A tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times\cdots\times I_d}$ is a d-way array of size $N = I_1 \times \cdots \times I_d$ typically stored in a flattened row-major or column-major format. It is standard practice [5] to denote tensors as bold uppercase letters in Euler calligraphic font (e.g., $\boldsymbol{\mathcal{X}}$), matrices by bold uppercase letters (e.g., $\mathbf{A}$), vectors by bold lowercase letters (e.g., $\boldsymbol{a}$), and scalars by lowercase letters (e.g., a). We adopt MATLAB notation for array indexing[2].

The Khatri-Rao product $\odot$ of two matrices $\mathbf{A} \in \mathbb{R}^{m\times p}$ and $\mathbf{B} \in \mathbb{R}^{n\times p}$ is the column-wise Kronecker product of $\mathbf{A}$ and $\mathbf{B}$ defined by

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1 | \ldots | \mathbf{a}_p \otimes \mathbf{b}_p] \in \mathbb{R}^{mn\times p}, \tag{1}$$

where $\otimes$ is the Kronecker product and $\mathbf{a}_j^{(k)} \equiv \mathbf{A}_k(:, j)$ is a column vector.

Tensors are indexed via multi-indexed tuples $i = (i_1, \ldots, i_d)$ where $i_n \in [I_n]$. The set notation $[a]$ is shorthand for $\{z : z \in \mathbb{Z}, 1 \le z \le a\}$. We call $i \in [N]$ the *linear index* of a tensor and $(i_1, \ldots, i_d)$ the *multi-index*, we assume a bijective mapping between the two index sets, and we refer to a *tensor element* as $x_i \equiv \boldsymbol{\mathcal{X}}(i) \equiv \boldsymbol{\mathcal{X}}(i_1, \ldots, i_d)$. The forward map $i \mapsto (i_1, \ldots, i_d)$ is given by the operator $\textsc{ind2sub}(\boldsymbol{\mathcal{X}}, i)$ and is defined like the MATLAB function of the same name[3], while the backward map $(i_1, \ldots i_d) \mapsto i$ is given by the operator $\textsc{sub2ind}(\boldsymbol{\mathcal{X}}, (i_1, \ldots, i_d))$, again defined like the corresponding MATLAB function[4]. The nth mode-k *subtensor* of a tensor $\boldsymbol{\mathcal{S}}_n^{(k)}$ is a $d-1$ tensor obtained by fixing the kth value of the multi-index to n and allowing the other values to range, i.e., $\boldsymbol{\mathcal{S}}_n^{(1)} = \boldsymbol{\mathcal{X}}(n, :, \ldots, :)$. The mode-$k$ *matricization* of a tensor $\boldsymbol{\mathcal{X}}$ rearranges the tensor elements into a matrix $\mathbf{X}_{(k)} \in \mathbb{R}^{I_k\times N/I_k}$ such that element i maps to (i_k, i'_k) by

$$i'_k = 1 + \sum_{\substack{\ell=1\\ \ell\neq k}}^{d} (i_\ell - 1) \prod_{\substack{m=1\\ m\neq k}}^{\ell-1} I_m. \tag{2}$$

The *reshape* operator $\textsc{reshape}(\boldsymbol{\mathcal{X}}, [s_1, \ldots, s_m])$ *does not permute* the underlying tensor elements (unlike tensor matricization) and is defined like the MATLAB function of the same name[5].

The rank-R *canonical polyadic (CP) decomposition* [20, 21] of a tensor $\boldsymbol{\mathcal{X}}$ is a rank-R tensor $\boldsymbol{\mathcal{M}}$ of the form

$$\boldsymbol{\mathcal{X}} \approx \boldsymbol{\mathcal{M}} = \sum_{j=1}^{R} \lambda_j \mathbf{a}_j^{(1)} \circ \cdots \circ \mathbf{a}_j^{(d)}, \tag{3}$$

[2] https://www.mathworks.com/help/matlab/math/array-indexing.html
[3] https://www.mathworks.com/help/matlab/ref/ind2sub.html
[4] https://www.mathworks.com/help/matlab/ref/sub2ind.html
[5] https://www.mathworks.com/help/matlab/ref/double.reshape.html

TABLE I

METHODS AND CODEBASES FOR DENSE MODE-k MTTKRPS. GIVEN A TENSOR $\boldsymbol{\mathcal{Y}}$ OF SHAPE $I_1, \ldots, I_d$ WITH FACTOR MATRICES $\mathbf{A}_1, \ldots, \mathbf{A}_d$, LET THE KHATRI-RAO MATRIX BE $\mathbf{Z} = \mathbf{A}_d \odot \cdots \odot \mathbf{A}_{k+1} \odot \mathbf{A}_{k-1} \odot \cdots \odot \mathbf{A}_1$ (SEE SECTION II-B FOR DETAILS). WE LIST MEMORY USAGE AND OPEN-SOURCE CODES FOR EACH METHOD, AND WE LIST CPU AND GPU SUPPORT FOR EACH CODEBASE. ALL METHODS HAVE THE SAME ASYMPTOTIC COMPUTATIONAL COST OF $\mathcal{O}(Rd(\prod_n I_n))$.

methods	memory	code	CPU	GPU
full $\mathbf{Z}$ [6]	$\mathcal{O}\left(R \prod_{n \neq k} I_n\right)$	N-way Toolbox [7]	✓	
		TensorLy [8]	✓	✓
		rTensor [9]	✓	
partial $\mathbf{Z}$ [13]	$\mathcal{O}\left(R\left(\prod_{n=1}^{k-1} I_n + \prod_{k+1}^{d} I_n\right)\right)$	TTB [6]	✓	
		PyTTB [11]	✓	
		GenTen [12]	✓	✓
no $\mathbf{Z}$ (our method)	$\mathcal{O}\left(R \sum_{n=1}^{d} I_n\right)$	GenTen	✓	✓

where $\boldsymbol{\lambda} \in \mathbb{R}^R$ is a weight vector and $\circ$ is a d-way outer product. We call $\mathbf{A}_k = [\mathbf{a}_1^{(1)}, \ldots, \mathbf{a}_d^{(k)}] \in \mathbb{R}^{I_k \times R}$ the mode-k *factor matrix* of a tensor $\boldsymbol{\mathcal{X}}$. The form $\boldsymbol{\mathcal{M}} = [\![\mathbf{A}_1, \ldots, \mathbf{A}_d]\!]$ is referred to as a *Kruskal tensor*.

B. The MTTKRP kernel

Algorithms to compute CP decompositions generally fall into two categories: all-at-once [22] or alternating [23, 21]. In both cases, the workhorse kernel is the mode-k *matricized tensor times Khatri-Rao product* (MTTKRP), defined as

$$\mathbf{G}^{(k)} = \mathbf{Y}_{(k)} \mathbf{Z} \operatorname{diag}(\boldsymbol{\lambda}), \tag{4}$$

where $\mathbf{Z} = \mathbf{A}_d \odot \cdots \odot \mathbf{A}_{k+1} \odot \mathbf{A}_{k-1} \odot \cdots \odot \mathbf{A}_1$ for some Kruskal tensor $\boldsymbol{\mathcal{M}} = [\![\mathbf{A}_1, \ldots, \mathbf{A}_d]\!]$, d-way tensor $\boldsymbol{\mathcal{Y}}$, and weight vector $\boldsymbol{\lambda} \in \mathbb{R}_+^R$. We call algorithms that explicitly construct $\mathbf{Z}$ *matrix-based* MTTKRPs.

Matrix-based MTTKRPs have two distinct downsides: (1) the explicit formation of the Khatri-Rao matrix $\mathbf{Z}$, and (2) the explicit matricization $\mathbf{Y}_{(k)}$. For a mode-k MTTKRP with CP-rank R, the Khatri-Rao matrix $\mathbf{Z}$ is size $\prod_{n \neq k} I_n \times R$, which can dwarf the cost $R \sum_{n \neq k} I_n$ of simply storing the factor matrices $\mathbf{A}_n$ individually, leading to $\mathcal{O}(R \prod_{n \neq k} I_n)$ memory usage. As for the matricization $\mathbf{Y}_{(k)}$, tensors are typically stored in memory as a flat array, and in such cases, the mode-k (for $1 < k < d$) matricization requires reordering the non-contiguous tensor entries into contiguous chunks. These reorderings have a memory-bound cost of $\mathcal{O}((d-1)N)$ and impose significant costs on an otherwise computationally bound kernel.

These drawbacks motivate a *matrix-free*, or *element-wise*, definition of the mode-k MTTKRP (common in the *sparse* case [19]), requiring only the explicit storage of $\boldsymbol{\mathcal{Y}}$, $\boldsymbol{\lambda}$, and $\{\mathbf{A}_m\}_{m=1}^d$, given by

$$\mathbf{G}^{(k)}(n, j) = \lambda_j \sum_{\substack{i=1 \\ i_k = n}}^{N} \boldsymbol{\mathcal{Y}}(i) \prod_{\substack{m=1 \\ m \neq k}}^{d} \mathbf{A}_m(i_m, j), \tag{5}$$

for $n \in [I_k]$ and $j \in [R]$. As the CP-rank R grows, the dominant memory cost of this approach becomes the storage of the factor and output matrices, yielding an overall $\mathcal{O}(R \sum_n I_n)$ memory burden.

C. Kokkos programming framework

Kokkos [24] is a C++ framework providing abstractions to write *portable* functions for single-instruction multiple-data (SIMD) parallelism on CPUs and single-instruction multiple-threads (SIMT) parallelism on GPUs. It is the Kokkos convention to discuss levels of parallelism mirroring those of the OpenMP 4.0 specification [25], i.e., *leagues* of *teams* comprising team-threads which may execute *vector-thread* instructions in parallel. Leagues are virtual and correspond to the highest level of parallel space (e.g., a CUDA grid or a collection of OpenMP threads), while a team within a league corresponds to hyperthreads on a CPU and y-dimension threads within a block on a GPU. Vector parallelism corresponds to vector instructions on CPUs and x-dimension threads (i.e., threads within a warp) on GPUs. We use **LeagueRange**, **TeamRange**, and **VectorRange** to denote league, team, and vector parallel ranges, respectively. We use b_x to denote team size and b_y to denote vector size, and we use p_x to denote threads within a team and p_y to denote vector-threads within a thread.

1) SIMD arrays: Compile-time polymorphic SIMD arrays were introduced in [19] as an extension of Kokkos that allow for the allocation of small arrays as register arrays on GPUs or thread-private stack arrays on CPUs. We utilize these arrays to allow for a single vector-thread p_y to calculate many (e.g., 4) updates to the output matrix $\mathbf{G}^{(k)}$ per vector loop, increasing memory-bandwidth efficiency. We denote the SIMD vector size as F and SIMD vectors as Greek letters with the vector arrow, e.g., $\vec{\pi}, \vec{\varphi}$. SIMD vectors are in effect for loops that the compiler is forced to unroll and vectorize in registers.

III. PARALLEL MTTKRP ALGORITHMS

The state-of-the-art dense MTTKRP was introduced in [13] and takes the viewpoint of Eq. (4). The algorithm, called MTTKRP-GEMM (Algorithm 1), uses strategic partitioning of the Khatri-Rao product matrix $\mathbf{Z}^T$ to avoid costly reorders of $\mathcal{Y}$, utilizes temporary memory allocations for a (sometimes) lower storage cost, and gets its name from its use of the BLAS-3 general matrix-matrix multiplies (GEMMs) for highly optimized parallelization.

We introduce element based algorithms all taking the viewpoint of Eq. (5); each of our algorithms utilize the *same* vector parallelization over columns of $\{\mathbf{A}_k\}$ and *differ* in how they assign tensor elements $\mathcal{Y}(i)$ (or groups of $\mathcal{Y}(i)$'s) to Kokkos teams and threads, and are described as follows:

- **Element-ordered** (Algorithm 2): assign each tensor element $\mathcal{Y}(i)$ to a thread; convert linear index to multi-index to read from $\{\mathbf{A}_k\}$; use atomics to resolve *element-level* conflicts in $\mathbf{G}^{(k)}$.
- **Subtensor-ordered** (Algorithm 3 with $N_T = N/I_k$): assign each subtensor $\mathcal{S}_n^{(k)}$ to a team; read from $\{\mathbf{A}_k\}$ with column reuse; compute $\mathbf{G}^{(k)}$ in registers and write to global memory conflict free.
- **Tile-ordered** (Algorithm 3): assign subsets of each subtensor $\mathcal{S}_n^{(k)}((t-1)N_T : tN_T)$, called tiles, to a team; read from $\{\mathbf{A}_k\}$ as in the subtensor-ordering algorithm; write to $\mathbf{G}^{(k)}$ with atomics to resolve *tile-level* conflicts.

Each algorithm (including GEMM) takes as an input the tensor $\mathcal{Y}$, the set of factor matrices $\{\mathbf{A}_k\}_{m=1}^d$, the number of ranks R, the target mode k (for a mode-k MTTKRP); the tile-ordered algorithm takes in a hyperparameter N_T, called the tile volume; each matrix-free algorithms also take in a SIMD size F.

MTTKRP-ELEM, the element-ordered algorithm, is parallel over the tensor elements $\mathcal{Y}(i)$ and is easy to implement, but has no structure when reading $\{\mathbf{A}_k\}$, yielding poor cache management and incurring $\mathcal{O}(NR)$ atomic writes to $\mathbf{G}^{(k)}$.

MTTKRP-SUB, the subtensor-ordered algorithm, assigns each subtensor to a team—taking advantage of the fact that the nth row of $\mathbf{G}^{(k)}$ is determined solely by the tensor elements in $\mathcal{S}_n^{(k)}$—to cache and reuse columns of $\{\mathbf{A}_k\}$ and write to $\mathbf{G}^{(k)}$ conflict free. However, assigning each subtensor to a team reduces parallelism when the subtensor size is larger than the team size as each thread must process many elements.

MTTKRP-TILE, the tile-ordered algorithm, attempts to resolve this issue by assigning teams to constant-sized subsets of subtensors, called tiles, allowing for more parallel threads to launch and ensuring a uniform workload across teams. Columns of $\{\mathbf{A}_k\}$ are reused as in the subtensor ordering, but $\mathbf{G}^{(k)}$ must be written with $\mathcal{O}((N_S/N_T)R)$ atomics—where N_T is the number of elements in a tile and $N_S = N/I_k$ the number of elements in each subtensor—as different tiles in the same subtensor write to the same row; this ordering increases the parallelism compared to MTTKRP-SUB and cache reuse and it decreases atomics compared to MTTKRP-ELEM.

The vector parallelism scheme is the same as [19]: parallelism is introduced over the columns of the factor matrices $\{\mathbf{A}_k\}$ by assigning multiple column indices j in Eq. (5) to a thread vector p_y. Along with the enforcement of a row-wise layout, this allows for coalesced reads of the factor matrices, allowing the compute device to achieve a higher percentage of memory bandwidth. For simplicity, we assume that the CP-rank R divides the product of the vector size and the SIMD array length ($b_y \times F$) for the remainder of this discussion.

A. MTTKRP-GEMM

The key insight of [13] is to avoid the costly matricizations $\mathbf{Y}_{(k)}$ by using the RESHAPE operator and breaking up the *full* Khatri-Rao product $\mathbf{Z}^T$ into *left* and *right* components $\mathbf{Z}^T = [\mathbf{Z}_R^T \in \mathbb{R}^{R \times I_R} | \mathbf{Z}_L^T \in \mathbb{R}^{\mathbb{R} \times I_L}]$, where $I_L = \prod_{m=1}^{k-1} I_m$ and $I_R = \prod_{m=k+1}^{d} I_m$.

We briefly summarize the algorithm given in [13]. For more detail, please refer to the paper. For mode-1, replace $\mathbf{Y}_{(k)}$ with $\text{RESHAPE}(\mathcal{Y}, [I_1, I_R])$ in Eq. (4) to avoid permutations. Similarly, for mode d, replace $\mathbf{Y}_{(k)}$ with $\text{RESHAPE}(\mathcal{Y}, [I_L, I_d])^T$ in Eq. (4) to avoid permutations. The steps for modes $1 < k < d$ are given in Algorithm 1. MTTKRP-GEMM has a storage complexity of $\mathcal{O}(R(I_L + I_R))$ as the left and right factor matrices can be stored in the same temporary memory space. However, for modes-$1, d$, the storage cost of $\mathcal{O}(R(\prod_{m=1}^d I_n / \min\{I_1, I_d\}))$ is the same cost as forming $\mathbf{Z}^T$ explicitly in Eq. (4).

Algorithm 1 MTTKRP-GEMM

Input: $\mathcal{Y}, \{\mathbf{A}^{(m)}\}_{m=1}^d, R, k$
Output: $\mathbf{G}^{(k)}$

```
1: Z_R^T = diag(λ)(A_d ⊙ ··· ⊙ A_{k+1})^T
2: C = RESHAPE(Y, [I_L · I_k, I_R]) Z_R                // GEMM
3: 𝒞 = RESHAPE(C, [I_L, I_k, R])       // tensorize the matrix
4: Z_L^T = diag(λ)(A_{k-1} ⊙ ··· ⊙ A_1)^T
5: G^(k)(ℓ, j) = Σ_{q=1}^{I_L} 𝒞(q, ℓ, j) Z_L(q, j)       // parallelize with einsum, GEMM, etc.
```

1. $\mathbf{Z}_R^T = \text{diag}(\boldsymbol{\lambda})(\mathbf{A}_d \odot \cdots \odot \mathbf{A}_{k+1})^T$
2. $\mathbf{C} = \text{RESHAPE}(\mathcal{Y}, [I_L \cdot I_k, I_R])\mathbf{Z}_R$ // GEMM
3. $\mathcal{C} = \text{RESHAPE}(\mathbf{C}, [I_L, I_k, R])$ // tensorize the matrix
4. $\mathbf{Z}_L^T = \text{diag}(\boldsymbol{\lambda})(\mathbf{A}_{k-1} \odot \cdots \odot \mathbf{A}_1)^T$
5. $\mathbf{G}^{(k)}(\ell, j) = \sum_{q=1}^{I_L} \mathcal{C}(q, \ell, j)\mathbf{Z}_L(q, j)$ // parallelize with einsum, GEMM, etc.

B. MTTKRP-ELEM

The simplest attempt to parallelize Eq. (5) is to assign every thread within a team p_x to a tensor element $\mathcal{Y}(i)$ without enforcing any particular league-wise structure. This algorithm, MTTKRP-ELEM, is described in Algorithm 2. Tensor indices i are given by the team offset plus thread rank in line 3, hence the tensor reads in line 4 are packed on CPUs and coalesced on GPUs. Reading $\{\mathbf{A}_m\}_{m=1}^d$ and writing to $\mathbf{G}^{(k)}$ requires a conversion of the linear index i to a multi-index $(i_1, \ldots, i_d)$ in line 5 with the function $\text{IND2SUB}(\mathcal{Y}, i)$, which uses costly integer divisions. Lines 6-20 are the vector parallelism over columns of the factor matrices $\{\mathbf{A}_m\}_{m=1}^d$. Line 8 ensures that each vector-thread p_y processes *at least* F columns, and the while loop on lines 9-19 increases the number of columns processed by each thread to $R/(b_y \times F)$. Line 18 ensures that the column strides are packed/coalesced for a fixed p_x, but this is negated by line 14 as the multi-index entry i_m is *not coalesced* for neighboring p_x; these *pseudo-random* reads of the factor matrices have a *significant negative effect* on memory bandwidth as $R(d-1)$ factor matrices' columns are read per tensor element $\mathcal{Y}(i)$. Finally, $\mathbf{G}^{(k)}$ is *atomically* updated for each N tensor element R times on line 17.

Algorithm 2 MTTKRP-ELEM

Input: $\mathcal{Y}, \{\mathbf{A}_m\}_{m=1}^{d}, R, k, F$
Output: $\mathbf{G}^{(k)}$

1: **LeagueRange** $l \in [N/b_x]$
2:   **TeamRange** $p_x \in b_x$
3:     $i \leftarrow l \times N/b_x + p_x$
4:     $y_i \leftarrow \mathcal{Y}[i]$ // packed/coalesced reads
5:     $(i_1, \ldots, i_d) \leftarrow$ IND2SUB$(\mathcal{Y}, i)$
6:     **VecRange** $p_y \in [b_y]$
7:       $jj \leftarrow 0$
8:       $\vec{\varphi} \leftarrow \mathbf{0}$
9:       **while** $jj < R$ **do** // read column chunks
10:         $j \leftarrow jj + F \times b_y + p_y$
11:         $\vec{\varphi} \leftarrow x_i \times \boldsymbol{\lambda}[j : j + F]$
12:         **for** $m = 1, \ldots, d$ **do**
13:           **if** $m \neq k$ **then**
14:             $\vec{\varphi} \leftarrow \vec{\varphi} \times \mathbf{A}_m[i_m, j : j + F]$ // pseudo-random reads
15:           **end if**
16:         **end for**
17:         AtomicAdd$(\mathbf{G}^{(k)}[i_k, j : j + F], \vec{\varphi})$ // $\mathcal{O}(NR)$ atomic updates
18:         $jj \leftarrow jj + F \times b_y$
19:       **end while**
20:     **End VecRange**
21:   **End TeamRange**
22: **End LeagueRange**

C. MTTKRP-SUB

We would like our element-based parallelization scheme to avoid the atomic updates and pseudo-random reads and writes of the MTTKRP-ELEM algorithm. To avoid atomic operations, Eq. (5) informs us to process each mode-k subtensor (i.e., the $d-1$ subtensor formed by fixing mode k, see Fig. 1) $\mathcal{S}_n^{(k)}$ independently.

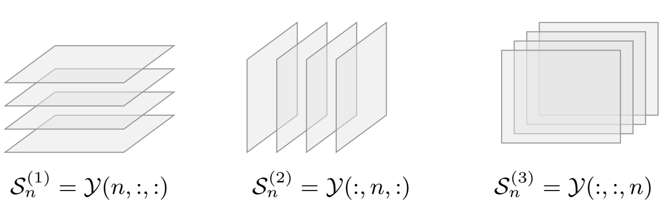

Fig. 1. Subtensors of a 3-way tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ along each mode.

We construct an algorithm MTTKRP-SUB that assigns league and team parallelism to each subtensor $\mathcal{S}_n^{(k)}$ of $\mathcal{Y}$. The MTTKRP-SUB algorithm is described by Algorithm 3 with $N_T = N_S = N/I_k$ (i.e., the number of elements in a given subtensor $\mathcal{S}_n^{(k)}$) and modifying the atomic add on line 25 to be a *conflict-free* global update. The algorithm begins by assigning each team to a subtensor $\mathcal{S}_n^{(k)}$ in line 4. Line 6 then computes an *anchor* multi-index a for the subtensor, in this case given by $a_m = 0$ for $m \neq k$ and $a_k = n$. Lines 8-28 are the vector parallelism over the factor matrices $\{\mathbf{A}_m\}_{m=1}^{d}$ columns and *differ* from the vector parallelism in Algorithm 2 as each vector-thread p_y processes N_T tensor elements y_i (lines 12-24). Line 13 computes the tensor multi-index $(i_1, \ldots i_d)$ as the sum of the anchor multi-index $(a_1, \ldots, a_d)$ and the IND2SUB$(\mathcal{S}_n^{(k)}, ii)$. Note that the resulting multi-index i is length d (i.e., the same length of a multi-index for $\mathcal{Y}$), and the kth entry of the multi-index $i_k = n$. For different subtensors $S_n^{(k)}$ and $S_m^{(k)}$, the multi-indices differ only in their kth element, hence in practice we implement a special IND2SUB$(\mathcal{S}_n^{(k)}, ii)$ function that uses *the same* precomputed offsets for each subtensor, avoiding the need to explicitly form the subtensor and compute costly inner loop integer divisions used in the classic implementation. Line 14 takes the resulting multi-index and calls a function SUB2IND$(\mathcal{Y}, (i_1, \ldots, i_d))$ built on cheap integer additions and multiplications to get the tensor linear index i. Line 2 assigns all I_k subtensors to a Kokkos team, forming a league grid of size (I_k, b_y). The inner loop over tensor elements in a subtensor allows the factor matrices columns to be cached and reused (line 20) in the computation of the *element Hadamard product* $\vec{\varphi} = y_i \prod_{m \neq k} \mathbf{A}_m(i_m, j)$. The element Hadamard product $\vec{\varphi}$ is then added to the *subtensor Hadamard product* $\vec{\pi}$ on line 23. After the inner subtensor loop terminates on line 28, the subtensor Hadamard products $\vec{\pi}$ are written to $\mathbf{G}^{(k)}(i_k, j : j + F)$ *without atomics*.

D. MTTKRP-TILE

The MTTKRP-SUB algorithm has the advantage over MTTKRP-ELEM in that it *avoids atomic conflicts* and has *beneficial cache blocking* for columns of the factor matrices $\{\mathbf{A}_m\}_{m=1}^{d}$. However, MTTKRP-SUB has $\mathcal{O}(N_S)$ more serial operations, which can lead to a *significant loss of parallelism* for MTTKRPs on modes with large subtensors (i.e., a mode-k MTTKRP where I_k is large). We attempt to remedy this loss of parallelism by assigning teams to tiles within a subtensor (see Fig. 2). Given a subtensor $\mathcal{S}_k^{(n)}$, a tile partition, defined by a *tile volume* N_T, portions the subtensor as $\{\mathcal{S}_n^{(k)}((t-1)N_T : tN_T)\}_{t=1}^{N_S/N_T}$, where the set elements $\mathcal{S}_n^{(k)}((t-1)N_T : tN_T)$ are called *tiles*.

We call the resulting algorithm that parallelizes over tiles MTTKRP-TILE and describe it in Algorithm 3. The algorithm is controlled by the tile volume hyperparameter N_T, which controls *both* the length of the serial

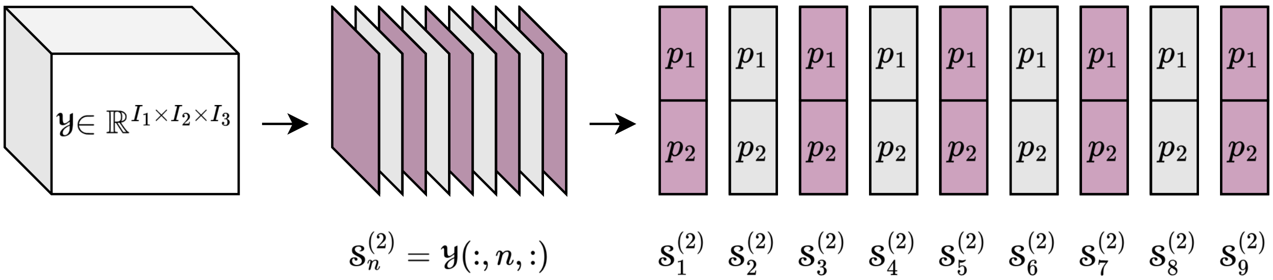

Fig. 2. A tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ decomposed into mode-2 subtensors $\mathcal{S}_n^{(2)}$. Each subtensor is then assigned to a Kokkos team of two threads p_1, p_2, where each team computes a *tile Hadamard product* and atomically reduces their local contributions to $\mathbf{G}^{(k)}(n, j)$. In this instructive example we have one team per subtensor, but in practice we allow for more than one team per subtensor. However, the map between teams and tiles is always bijective.

accumulation of the tile Hadamard product in lines 12-24 and the number of atomic operations in line 25, which is $\mathcal{O}((N_S/N_T)R)$. Parallelism is controlled by the tile volume; line 2 launches N/N_T teams, i.e., a smaller N_T launches more teams. Note that setting a tile size of $N_T = N_S = N/I_k$ and removing the atomic stipulation on line 25 reverts the algorithm to MTTKRP-SUB, while setting a tile size of $N_T = 1$ reverts the algorithm to MTTKRP-ELEM, albeit with more structure in the team assignments. We refer to the discussion in Section III-C for a detailed discussion of the repeated pseudo-code in Algorithm 3.

Algorithm 3 MTTKRP-TILE

Input: $\mathcal{Y}, \{\mathbf{A}_m\}_{m=1}^d, R, k, F$; tile volume N_T
Output: $\mathbf{G}^{(k)}$

1: $N_S \leftarrow N/I_k$ // slice volume
2: **LeagueRange** $l \in [N/N_T]$ // parallelism over tiles
3: **TeamRange** $p_x \in [N_T]$ // thread reuse $N_T > b_x$
4: $n \leftarrow \text{INTDIV}(l \times b_x + p_x, N_S/N_T)$ // $\mathcal{S}_n^{(k)}$
5: $t \leftarrow \text{INTMOD}(l \times b_x + p_x, N_S/N_T)$
6: $(a_1, \ldots, a_m) \leftarrow \text{IND2SUB}(\mathcal{S}_n^{(k)}[(t-1)N_T], 0)$
7: $jj \leftarrow 0$
8: **VecRange** $p_y \in [b_y]$
9: $\vec{\pi} \leftarrow \mathbf{0}$ // init. tile Hadamard product
10: $\vec{\varphi} \leftarrow \mathbf{0}$ // init. element Hadamard product
11: **while** $jj < R$ **do**
12: **for** $ii \in [N_T]$ **do**
13: $(i_1, \ldots, i_d) \leftarrow (a_1, \ldots, a_d) + \text{IND2SUB}(\mathcal{S}_n^{(k)}[(t-1)N_T], ii)$
14: $i \leftarrow \text{SUB2IND}(\mathcal{Y}, (i_1, \ldots, i_d))$
15: $y_i \leftarrow \mathcal{Y}[i]$ // re-read tensor element
16: $j \leftarrow jj + F \times b_y + p_y$
17: $\vec{\varphi} \leftarrow y_i \times \boldsymbol{\lambda}[j : j + F]$
18: **for** $m = 1, \ldots, d$ **do**
19: **if** $m \neq k$ **then**
20: $\vec{\varphi} \leftarrow \vec{\varphi} \times \mathbf{A}_m[i_m, j : j + F]$ // columns are reused
21: **end if**
22: **end for**
23: $\vec{\pi} \leftarrow \vec{\pi} + \vec{\varphi}$ // update tile product
24: **end for**
25: $\text{ATOMICADD}(\mathbf{G}^{(k)}[n, j : j + F], \vec{\pi})$ // $\mathcal{O}((N_S/N_T)R)$ atomic updates
26: $jj \leftarrow jj + F \times b_y$
27: **end while**
28: **End VecRange**
29: **End TeamRange**
30: **End LeagueRange**

IV. Numerical results

We study the performance and portability of the different MTTKRP algorithms described in Section III. Our numerical experiments are performed on two tensors with randomly generated values based on relevant scientific simulations [1]: the 2D compressible tearing mode tensor [26, 27] $\mathcal{A} \in \mathbb{R}^{401 \times 201 \times 12 \times 501}$, about 3.7 GB in double precision, and the 3D island coalescence tensor [28] $\mathcal{B} \in \mathbb{R}^{129 \times 129 \times 129 \times 12 \times 39}$, about 7.5 GB. Our experiments use CP-ranks $R = 10, 500, 1000, 2000$.

A. Experimental setup

Each algorithmic variant is implemented in the publicly available GenTen library[6] built atop Kokkos for performance portability. Our numerical experiments are performed on cluster nodes of the Hops (OpenMP/CUDA) and El Dorado (HIP) machines at Sandia National Labs. Each Hops node has two 3.8 GHz Intel Xeon Platinum 8480+ CPUs and four Nvidia H100 SXM5 GPUs; each El Dorado node has four 1.8 GHz AMD 4th Gen EPYC CPUs and four AMD Instinct MI300A GPUs. For the CPUs, we set $b_x = b_y = 1$ (i.e., we only use league parallelism), use GenTen heuristics (see [19] Tables 3,4) for selecting F based on R, and compile with the Intel MKL LAPACK and BLAS and architecture specific vectorization flags set by Kokkos. We also set the environment variables `OMP_NUM_THREADS` = 56, `OMP_PROC_BIND` = `close`, and

[6]https://github.com/sandialabs/GenTen

`OMP_PLACES = threads`. On GPUs, we set $b_x = 128/b_y$ and choose b_y and F based on R using the aforementioned GenTen heuristics.

B. Performance analysis

We use a model $\mathbf{f} = NRd$ to measure the flop count of the MTTKRP as for each N tensor elements, we do $R(d-1)$ multiplications and R additions. Memory bandwidth is measured for each element-based algorithmic variant using a 0-cache model $\mathbf{m}_0$, an infinite-cache model $\mathbf{m}_\infty$, and a $0, \mathbf{LM}$-cache model $\mathbf{m}_{0,\mathbf{LM}}$, where $\mathbf{LM}$ is the middle level of cache on a device, i.e., L1 cache on a GPU and L2 cache on a CPU. The $\mathbf{m}_\infty$ model is designed for compute bound kernels where reading data from cache is cheap compared to the computations, the $\mathbf{m}_0$ assumes data movement is expensive (i.e., the cost of reading from global memory is always incurred), and $\mathbf{m}_{0,\mathbf{LM}}$ attempts to split the difference by modeling a cache like $\mathbf{m}_\infty$ but assigning a cost to accessing said cache. For the matrix-free mode-k MTTKRPs, the infinite-cache model is given by $\mathbf{m}_\infty = s_\mathbf{f}(N + R\sum_n I_n)$, where $s_\mathbf{f}$ is the size in bytes of the floating-point type, N is the size of the tensor, $R\sum_{n\neq k} I_n$ is the size of reading all but the kth factor matrix, and RI_k is the size of the output matrix. The 0-cache model is given by $\mathbf{m}_0 = s_\mathbf{f}(N + NR(d-1) + (N_S/N_T)I_kR)$ as the whole tensor must be read in, *every* tensor element N must read *each* factor matrix column R for $d-1$ factor matrices, and each R columns of the output matrix are updated N_S/N_T times for each k subtensors. The $0, \mathbf{LM}$-cache model assumes that the tensor and output matrices cannot be cached as they *are not* read repeatedly, but the factor matrices *are* cached, yielding the model $\mathbf{m}_{0,\mathbf{LM}} = s_\mathbf{f}(N + N_S/N_T(I_kR)) + \frac{1}{l}s_\mathbf{f}(NR(d-1))$, where l is the ratio between $\mathbf{LM}$ bandwidth and device memory bandwidth.

We use these models to predict the total time $T_0 = \mathbf{f}/\tau_\mathbf{f} + \mathbf{m}_0\tau_\mathbf{m}$, $T_\infty = \mathbf{f}/\tau_\mathbf{f} + \mathbf{m}_\infty/\tau_\mathbf{m}$, and $T_{0,\mathbf{LM}} = \mathbf{f}/\tau_\mathbf{f} + \mathbf{m}_{0,\mathbf{LM}}/\tau_\mathbf{m}$, where $\tau_\mathbf{f}$ is the peak machine measured in flops and $\tau_\mathbf{m}$ is the peak machine bandwidth measured in read/write operations per second. We define arithmetic intensity to be $\mathbf{I}_i = \mathbf{f}/\mathbf{m}_i$, $i \in \{0, \infty, (0, \mathbf{LM})\}$, and we say that a model is compute bound if $\mathbf{I}_i > \tau_\mathbf{f}/\tau_\mathbf{m}$ for a chosen i. Given a wall-time t, the throughput of the model is measured as $\mathbf{gflops} = \mathbf{f}/t/1024^3$ and the memory bandwidths for each model are given by $\mathbf{mops}_\infty = \mathbf{m}_\infty/t$, $\mathbf{mops}_{0,\mathbf{LM}} = \mathbf{m}_{0,\mathbf{LM}}/t$, and $\mathbf{mops}_0 = \mathbf{m}_0/t$.

Peak throughput $\tau_\mathbf{f}$, bandwidth $\tau_\mathbf{m}$, and $\mathbf{LM}$ bandwidth ratio l for each device is reported in Table II. For the H100, throughput and bandwidth are taken from NVIDIA's whitepaper[7] and we calculate the peak L1 cache bandwidth as the number of SMs $\times$ the L1 transfer bytes/cycle $\times$ the clock rate, which is $123 \times 128 \times 1.98/10^3 \approx 33$TB/s, where the L1 transfer bytes per clock cycle is taken from NVIDIA's hopper tuning guide[8]. Hence, for the H100, we set $l = 10$. For the MI300A, throughput, bandwidth, L1 transfer bytes per clock cycle, and max clock rate are taken from AMD's whitepaper[9] and we use the same formula to calculate peak L1 cache bandwidth as $228 \times 128 \times 2.1/10^3 \approx 61$TB/s; for the MI300A, we set $l = 12$. Throughput for the 8480+ is taken from Intel's APP metrics sheet[10] while memory bandwidth and L2 cache bandwidth are measured with the STREAM triad benchmark [29], where we use a 2GB array to measure memory bandwidth and a 0.6MB array to measure the L2 cache bandwidth.

TABLE II
PEAK COMPUTE THROUGHPUT $\tau_\mathbf{f}$, MEMORY BANDWIDTH $\tau_\mathbf{m}$, AND CACHE BANDWIDTH RATIO l FOR EACH DEVICE.

Device	$10^{12}\tau_\mathbf{f}$	$1024^4\tau_\mathbf{m}$	l
8480+	2.28	0.12	8
H100	34	3.35	10
MI300A	61.3	5.3	12

We analyze the efficacy of our performance models in Table III. Our results show that the cache effects play a significant role for each algorithmic variant: the T_∞ prediction is much too optimistic for each variant. The MTTKRP-ELEM algorithm fails to surpass even the 0-cache predicted time T_0 for either device. As discussed in Section III-B, this can be attributed to the uncoalesced repeated reads of the factor-matrices. The MTTKRP-SUB algorithm also fails to surpass T_0 on the GPU, though it does so on the CPU. We hypothesize that this discrepancy is due to the much larger cache size of the CPU compared to the GPU. For MTTKRP-SUB on the CPU and MTTKRP-TILE on both devices, the 0-cache model is far too pessimistic while the ∞-cache model is too optimistic, indicating that the algorithms achieve (to some extent) the desired caching effects on the repeated reads of the factor matrices. Our empirical results for these methods align most with the $0, \mathbf{LM}$-cache model, achieving at least 90% of the predicted time on the CPU, 40% of the predicted time on the H100 GPU, and 10% of the predicted time on the MI300A GPU. We surmise that the higher accuracy of our models on the CPU is due to the fact that we use empirically measured peak bandwidth, while on the GPU we use the vendor reported ideal world numbers.

C. Experiments

1) Selecting the tile size: Our first goal is to select the heuristic tile volume parameter N_T in the

[7]https://www.NVIDIA.com/en-us/data-center/h100/

[8]https://docs.NVIDIA.com/CUDA/hopper-tuning-guide/index.html

[9]https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/data-sheets/amd-instinct-mi300a-data-sheet.pdf

[10]https://www.intel.com/content/www/us/en/support/articles/000005755/processors.html

TABLE III

EXECUTION TIME T IN SECONDS MEASURED AGAINST ANALYTICAL PERFORMANCE MODEL TIMES $T_0, T_{0,\mathbf{LM}}, T_\infty$ FOR THE TENSORS $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}$ ON THE OPENMP, CUDA AND HIP EXECUTION SPACES. THE TILE SIZE N_T FOR THE *MTTKRP-TILE* ALGORITHM IS SELECTED VIA THE HEURISTIC IN EQ. (6). THE CP-RANK IS FIXED TO $R = 32$.

		OpenMP				CUDA				HIP			
		T	T_0	$T_{0,\mathbf{LM}}$	T_∞	T	T_0	$T_{0,\mathbf{LM}}$	T_∞	T	T_0	$T_{0,\mathbf{LM}}$	T_∞
$\boldsymbol{\mathcal{A}}$	ELEM	10.394	3.817	1.349	0.056	1.823	0.138	0.047	0.003	2.668	0.087	0.028	0.002
	SUB	0.543	2.877	0.409	0.056	8.528	0.104	0.013	0.003	16.267	0.066	0.007	0.002
	TILE	0.526	2.955	0.487	0.056	0.043	0.110	0.019	0.003	0.063	0.069	0.011	0.002
$\boldsymbol{\mathcal{B}}$	ELEM	25.167	9.876	3.054	0.130	3.924	0.356	0.105	0.007	7.995	0.225	0.063	0.004
	SUB	1.137	7.927	1.105	0.130	23.032	0.286	0.035	0.007	47.401	0.181	0.019	0.004
	TILE	1.517	8.414	1.592	0.130	0.120	0.304	0.052	0.007	0.202	0.188	0.026	0.004

MTTKRP-TILE algorithm. Table IV presents the numerical performance of a sweep over *tile-widths* $\sqrt[d-1]{N_T} \in \{2, 4, 6, 8, 10, 12\}$ for the tearing-mode and island coalescence tensors for a fixed $R = 32$. On the GPU, we find that for the 4-way tensor $\boldsymbol{\mathcal{A}}$, a tile size of 216 is optimal, while for the 5-way tensor $\boldsymbol{\mathcal{B}}$, a tile size of 256 is optimal.

On an H100, our maximum occupancy with our standard block configuration of $(4, 32)$ is 64 warps per SM. Nsight Compute reports our achieved occupancy to be 60%, but even if we assume a more conservative occupancy of 50%, we have 32 tiles per SM, with each tile rereading 8×256 bytes of factor matrices entries, or 64KB of information per SM, 25% of the total L1 cache, a good target given that the L1 cache handles read, write, and atomic instructions on a GPU.

On the 8480+, we find that a tile width of 12, i.e., the size of the minimum dimension, is optimal. Our parallel CPU implementation assigns one tile per core, and given that each core has 2MB of L2 cache, this means that the factor matrices take up at most $\sim 0.5\%$ of the L2 cache. However, our implementation assumes that the tile size is *regular*, i.e., that $\sqrt[d-1]{N_T} \in \mathbb{Z}$. As such, performance decreases when $\sqrt[d-1]{N_T} > \min_{n=1,\ldots,d} I_n$.

To impose tile regularity and cache awareness, we use the simple heuristic

$$N_T = \min \left\{ \left(\sqrt[d-1]{\frac{s_{\mathbf{LM}}/4}{s_{\mathbf{f}}(c/2)}} \right)^{d-1}, \min_{n=1,\ldots,d} \{I_n\}^{d-1} \right\}, \quad (6)$$

where $s_{\mathbf{LM}}$ is the size in bytes of the middle level of cache (i.e., L1 cache (per SM) on a GPU, L2 cache (per core) on a CPU), and c is the maximum number of tiles per cache unit (i.e., SM/core). We find that this heuristic is valid for most R.

D. *Memory impact of MTTKRP variants*

As discussed in Section III-A, forming the left and right Khatri-Rao matrices $\mathbf{Z}_L^T$ and $\mathbf{Z}_R^T$ can incur a large storage cost. This cost dominates the total memory required for the mode-k MTTKRP-GEMM algorithm, which is given by $\mathbf{m}_\infty^{\text{GEMM}} = N + R(I_L + I_R + I_k)$. It is important to note that the storage cost of the MTTKRP-GEMM algorithm relies heavily on the *initial shape* of a given tensor. For example, the maximum memory usage over all modes k for the island coalescence tensor $\boldsymbol{\mathcal{B}}$ for $R = 2000$ is 391GB, but if the initial tensor shape was permuted to be $129 \times 12 \times 129 \times 39 \times 129$ (best case), the memory usage would be 123GB, while if the tensor was permuted to be $129 \times 129 \times 129 \times 39 \times 12$ (worst case), the memory usage would be 1255GB. However, especially in the context of in situ decomposition of scientific simulation data [1, 30], reshaping tensors can eliminate multidimensional relationships between tensor entries while tensor transpositions can be expensive and impractical. Fig. 3 shows the maximum memory usage over all modes k for the MTTKRP-GEMM and MTTKRP-TILE algorithms for the tearing mode tensor $\boldsymbol{\mathcal{A}}$ and the island coalescence tensor $\boldsymbol{\mathcal{B}}$. The y axis of the plot denotes the *minimum* number of 80GB H100 GPUs needed for storage; however, the distribution protocol in Section IV-D2 does not evenly assign memory to MPI ranks, hence *in practice* more GPUs may be required for a given problem size.

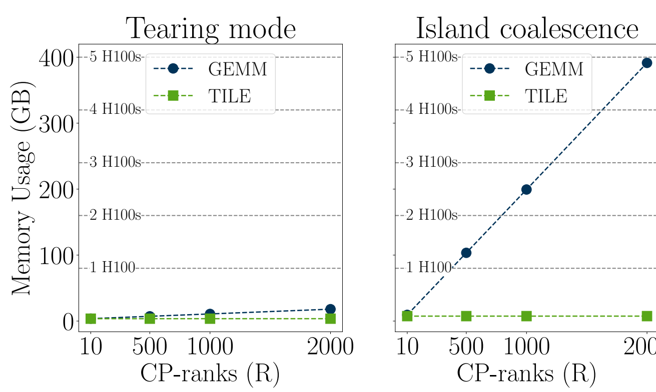

Fig. 3. Worst case memory usage for the *GEMM* and *TILE* MTTKRP variants for the tearing mode tensor $\boldsymbol{\mathcal{A}}$ and the island coalescence tensor $\boldsymbol{\mathcal{B}}$ for different CP-ranks. The plot is annotated with lines showing the minimum number of NVIDIA H100 GPUs required to incur the storage costs.

1) *Single-node performance:* The linear-in-rank memory scaling of the MTTKRP-GEMM algorithm described in Section III-A motivates the use of the tearing-mode

TABLE IV

SWEEP OVER TILE WIDTHS $\sqrt[d-1]{N_T}$ FOR THE *MTTKRP-TILE* ALGORITHMIC VARIANT WITH $R = 32$ FOR EACH TENSOR $\mathcal{A}, \mathcal{B}$ ON BOTH EXECUTION SPACES, WITH PERFORMANCE IN **gflops** (HIGHER IS BETTER) AVERAGED OVER ALL MODES. THESE RESULTS MOTIVATE A HEURISTIC TO CHOOSE N_T GIVEN BY EQ. (6).

	MTTKRP-TILE **gflops**					
$\sqrt[d-1]{N_T}$	Tearing mode tensor $\mathcal{A}$			Island coalescence tensor $\mathcal{B}$		
	OpenMP	CUDA	HIP	OpenMP	CUDA	HIP
2	32.9	253.3	168.0	51.4	585.1	296.6
4	89.9	1313.6	801.8	99.0	**1232.2**	**836.3**
6	98.9	**1328.3**	**910.1**	106.8	1173.4	743.1
8	98.1	1103.7	789.2	104.9	944.1	580.8
10	96.5	947.3	674.2	103.3	884.4	477.7
12	**99.6**	1230.0	808.3	**110.8**	948.8	475.9

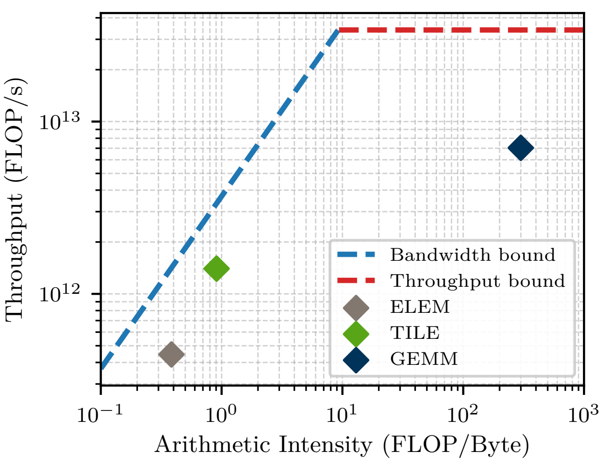

Fig. 4. Roofline plot for the tearing mode tensor $\mathcal{A}$ with rank $R = 2000$ on the NVIDIA H100. The $\mathbf{m}_\infty$ model is used for the *GEMM* MTTKRP variant and the $\mathbf{m}_{0,\mathbf{LM}}$ model for the *ELEM* and *TILE* MTTKRP variants.

tensor $\mathcal{A}$ to study single-node performance of the different MTTKRP variants. Fig. 4 shows a roofline analysis the ELEM, TILE, and GEMM kernels on $\mathcal{A}$ with $R = 2000$ on the H100 GPU; the GEMM kernel is sufficiently compute bound while the matrix-free kernels are memory bound and close to the roofline. Fig. 5 reports the average MTTKRP performance, measured in **gflops**, over all modes, for the OpenMP, CUDA, and HIP execution spaces. We compare the SUB, TILE, and GEMM variants on the OpenMP space and the ELEM, TILE, and GEMM variants on the CUDA and HIP spaces, where the tile size N_T is chosen by Eq. (6). We omit the ELEM variant for OpenMP and the SUB variant for CUDA/HIP given their poor performance shown in Table III. For $R > 10$, MTTKRP-TILE achieves at least 20% of MTTKRP-GEMM's performance, a favorable result given MTTKRP-TILE's memory footprint. Additionally, for $R > 10$, MTTKRP-TILE performs at least 2× faster than the SUB variant on the OpenMP space, at least 3× faster than ELEM on the CUDA space, with a peak speedup of 11× for $R = 500$, and at least 2× faster that ELEM on the HIP space, with a peak speedup of 6× for $R = 500$. These speedups reinforce the necessity for data reuse and reduced atomic contention for element-based parallelization approaches.

2) Comparison against distributed MTTKRP-GEMM: The disparities of memory costs between the MTTKRP-TILE and MTTKRP-GEMM algorithms for the island coalescence tensor $\mathcal{B}$ (see Fig. 3) motivate us to compare running the MTTKRP-TILE algorithm on a single GPU against running MTTKRP-GEMM algorithm on (the required) multiple GPUs. Distributed MTTKRPs use the communication pattern introduced in [31] whose communication overhead consists of an initial tensor redistribution and all-reduce communication to combine contributions across processors.

Fig. 6 compares the execution time of the CP-ALS algorithm on $\mathcal{B}$ with tolerance 10^{-4} over ranks $R = 10, 500, 1000, 2000$ on a single device when using the MTTKRP-TILE algorithm and on multiple devices when using the MTTKRP-GEMM algorithm. While tensor and factor matrices fit snugly into memory for the TILE variant, MTTKRP-GEMM requires at least six H100s for rank 2000, three H100s for rank 1000, and two H100s for rank 500 (findings in line with our lower bound memory model). Note that the Hops machine requires two nodes to distribute across six GPUs, with the multi-node communication bandwidth drastically increasing tensor redistribution and MTTKRP communication time. When $R = 2000$, MTTKRP-TILE-based CP-ALS on a single GPU is able to achieve 67% of the performance of MTTKRP-GEMM-based CP-ALS run on the minimum six H100s. For $R = 1000$, four H100s are required to beat the MTTKRP-TILE-based CP-ALS and *at least* five are required to beat the MTTKRP-TILE-based CP-ALS for $R = 500$. For all tested ranks, tensor redistribution is the most expensive overhead incurred in the distributed CP-ALS, dwarfing the all-reduce communication time.

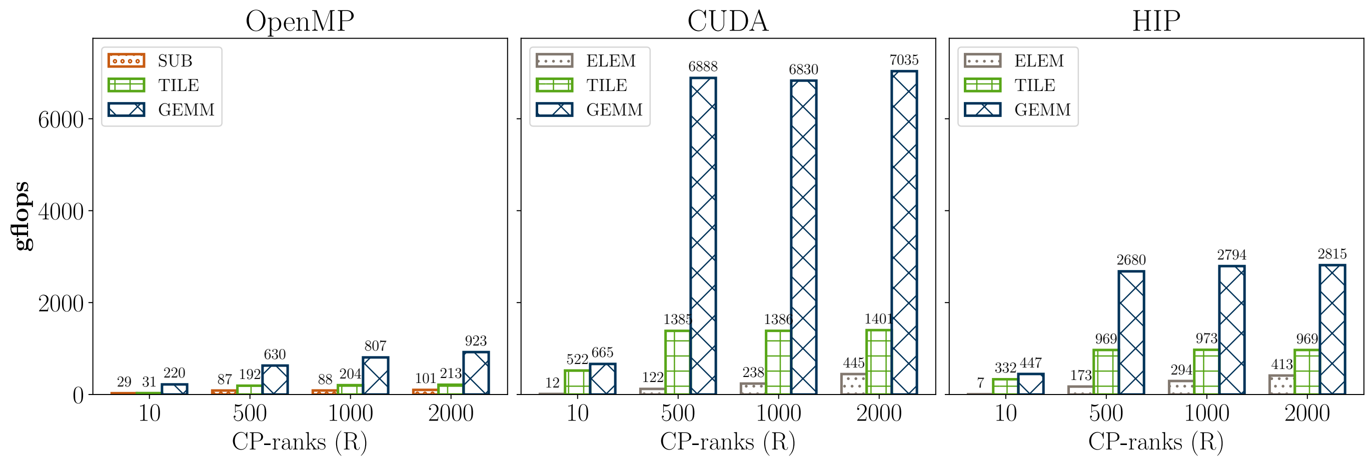

Fig. 5. Tearing mode tensor $\mathcal{A}$ average (over all modes) MTTKRP performance (higher is better) for different algorithmic variants on different execution spaces.

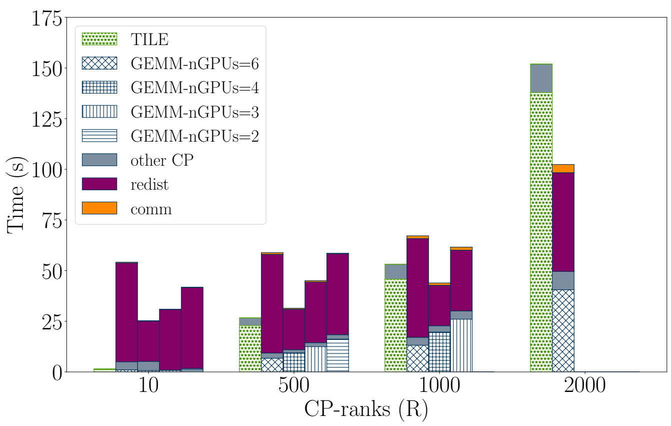

Fig. 6. Time (lower is better) for CP-ALS on the island coalescence tensor $\mathfrak{B}$ with *MTTKRP-TILE* algorithm on a single H100 compared with *MTTKRP-GEMM* on one H100 per process. MTTKRP time is denoted with hatches, other CP-ALS operations (i.e., inner-product, Cholesky solve, etc.) by a grey fill, tensor redistribution time with a purple fill, and MTTKRP communication time with an orange fill.

V. Conclusion

We introduce fast and memory-efficient algorithms for matrix-free dense mode-k MTTKRPs together with detailed performance analysis and evaluations on CPUs and GPUs. We extend the open-source GenTen package with our methods and demonstrate that state-of-the-art matrix-based MTTKRPs need many GPUs to compute the same tensor decomposition that, when replaced with our MTTKRP, can be computed on a single device. As a general rule, *we recommend that a practitioner use the GEMM algorithm when product matrices fit into memory and highly optimized BLAS-3 rutines are available; otherwise, use the TILE algorithm with our heuristic for selecting N_T.*

Our methods have limitations. Fig. 5 shows that we have yet to achieve the efficiency of the matrix-based MTTKRP-GEMM algorithm when the Khatri-Rao product matrix fits in device memory. Table III motivates us to further improve our caching strategies to achieve a higher percentage of the predicted infinite-cache time T_∞. Our roofline analysis (Fig. 4) shows that our TILE algorithm needs to move into the compute-bound regime to achieve higher throughput. Strategies to increase the arithmetic intensity include GEMM-like tilings of the output matrix: a 2D block tiling over the output matrix with shared memory caching, thread and warp tiling, and league-size/team-size auto-tuning—in short, mimicking GEMM without forming the product matrices. Additional optimizations include a Hilbert curve or ALTO-like [18] ordering of the input tensor to increase cache locality beyond the standard row-major layout. The optimizations stated above are future work.

Acknowledgment

This material is based upon work partially supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract number DE-NA0003525.

References

[1] D. M. Dunlavy, E. T. Phipps, H. Kolla, J. N. Shadid, and E. Phillips, "Goal-oriented low-rank tensor decompositions for numerical simulation data," 2025. [Online]. Available: https://arxiv.org/abs/2508.11139

[2] D. Van Essen, K. Ugurbil, E. Auerbach, D. Barch, T. Behrens, R. Bucholz, A. Chang, L. Chen, M. Corbetta, S. Curtiss, S. Della Penna, D. Feinberg, M. Glasser, N. Harel, A. Heath, L. Larson-Prior, D. Marcus, G. Michalareas, S. Moeller, R. Oostenveld, S. Petersen, F. Prior, B. Schlaggar, S. Smith, A. Snyder, J. Xu, and E. Yacoub, "The human connectome project: A data acquisition perspective," *NeuroImage*,

vol. 62, no. 4, pp. 2222–2231, 2012, connectivity. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1053811912001954

[3] 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.

[4] S. M. Nascimento, K. Amano, and D. H. Foster, "Spatial distributions of local illumination color in natural scenes," *Vision Research*, vol. 120, pp. 39–44, 2016, vision and the Statistics of the Natural Environment. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0042698915002709

[5] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009. [Online]. Available: https://doi.org/10.1137/07070111X

[6] B. W. Bader and T. G. Kolda, "Efficient matlab computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, 2008. [Online]. Available: https://doi.org/10.1137/060676489

[7] C. A. Andersson and R. Bro, "The n-way toolbox for matlab," *Chemometrics and Intelligent Laboratory Systems*, vol. 52, no. 1, pp. 1–4, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016974390000071X

[8] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, "Tensorly: Tensor learning in python," *Journal of Machine Learning Research*, vol. 20, no. 26, pp. 1–6, 2019. [Online]. Available: http://jmlr.org/papers/v20/18-277.html

[9] J. Li, J. Bien, and M. T. Wells, "rTensor: An R package for multidimensional array (tensor) unfolding, multiplication, and decomposition," *Journal of Statistical Software*, vol. 87, no. 10, pp. 1–31, 2018.

[10] B. W. Bader and T. G. Kolda, "Algorithm 862: Matlab tensor classes for fast algorithm prototyping," *ACM Trans. Math. Softw.*, vol. 32, no. 4, p. 635–653, Dec. 2006. [Online]. Available: https://doi.org/10.1145/1186785.1186794

[11] D. M. Dunlavy, N. T. Johnson *et al.*, "pyttb: Python Tensor Toolbox, v1.8.3," Aug. 2025. [Online]. Available: https://github.com/sandialabs/pyttb

[12] E. Phipps, T. Kolda, D. Dunlavy, G. Ballard, and T. Plantenga, "Genten: Software for generalized tensor decompositions v. 1.0.0," 06 2017. [Online]. Available: https://www.osti.gov/biblio/code-45463

[13] A.-H. Phan, P. Tichavský, and A. Cichocki, "Fast alternating ls algorithms for high order candecomp/parafac tensor factorizations," *IEEE Transactions on Signal Processing*, vol. 61, no. 19, pp. 4834–4846, 2013.

[14] N. Vannieuwenhoven, K. Meerbergen, and R. Vandebril, "Computing the gradient in optimization algorithms for the cp decomposition in constant memory through tensor blocking," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. C415–C438, 2015. [Online]. Available: https://doi.org/10.1137/14097968X

[15] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[16] J. H. Choi and S. V. N. Vishwanathan, "Dfacto: distributed factorization of tensors," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 1296–1304.

[17] J. Li, J. Sun, and R. Vuduc, "Hicoo: Hierarchical storage of sparse tensors," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 238–252.

[18] J. Laukemann, A. E. Helal, S. I. G. Anderson, F. Checconi, Y. Soh, J. J. Tithi, T. Ranadive, B. J. Gravelle, F. Petrini, and J. Choi, "Accelerating sparse tensor decomposition using adaptive linearized representation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 5, pp. 1025–1041, 2025.

[19] E. T. Phipps and T. G. Kolda, "Software for sparse tensor decomposition on emerging computing architectures," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C269–C290, 2019. [Online]. Available: https://doi.org/10.1137/18M1210691

[20] J. D. Carroll and J. J. Chang, "Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition," *Psychometrika*, vol. 35, pp. 283–319, 1970.

[21] R. A. Harshman, "Foundations of the parafac procedure: Models and conditions for an "explanatory" multi-model factor analysis," in *Working Papers in Phonetics*, 1970. [Online]. Available: https://api.semanticscholar.org/CorpusID:6816804

[22] E. Acar, D. M. Dunlavy, and T. G. Kolda, "A scalable optimization approach for fitting canonical tensor decompositions," *J. Chemom.*, vol. 25, no. 2, pp. 67–86, Feb. 2011.

[23] J. Douglas Carroll, S. Pruzansky, and J. B. Kruskal, "Candelinc: A general approach to multidimensional analysis of many-way arrays with linear constraints on parameters," *Psychometrika*, vol. 45, no. 1, pp. 3–24, Mar 1980. [Online]. Available: https://doi.org/10.1007/BF02293596

[24] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[25] OpenMP Architecture Review Board, "OpenMP ap-

plication program interface version 4.0," 2013.
[26] J. Bonilla, J. Shadid, X.-Z. Tang, M. Crockatt, P. Ohm, E. Phillips, R. Pawlowski, S. Conde, and O. Beznosov, "On a fully-implicit vms-stabilized fe formulation for low mach number compressible resistive mhd with application to mcf," *Computer Methods in Applied Mechanics and Engineering*, vol. 417, p. 116359, 2023, a Special Issue in Honor of the Lifetime Achievements of T. J. R. Hughes. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782523004838
[27] L. Chacón, "An optimal, parallel, fully implicit newton–krylov solver for three-dimensional viscoresistive magnetohydrodynamicsa)," *Physics of Plasmas*, vol. 15, no. 5, p. 056103, 02 2008. [Online]. Available: https://doi.org/10.1063/1.2838244
[28] J. Shadid, R. Pawlowski, E. Cyr, R. Tuminaro, L. Chacón, and P. Weber, "Scalable implicit incompressible resistive mhd with stabilized fe and fully-coupled newton–krylov-amg," *Computer Methods in Applied Mechanics and Engineering*, vol. 304, pp. 1–25, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782516300184
[29] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.
[30] G. Ballard, K. Hayashi, and K. Ramakrishnan, "Parallel nonnegative cp decomposition of dense tensors," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, 2018, pp. 22–31.
[31] C. Lewis and E. Phipps, "Low-communication asynchronous distributed generalized canonical polyadic tensor decomposition," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–5.