Journal of Parallel and Distributed Computing 169 (2022) 269–285

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Distributed-memory tensor completion for generalized loss functions in python using new sparse tensor kernels

Navjot Singh *, Zecheng Zhang, Xiaoxiao Wu, Naijing Zhang, Siyuan Zhang, Edgar Solomonik

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA

ARTICLE INFO

Article history:
Received 2 February 2022
Received in revised form 3 July 2022
Accepted 7 July 2022
Available online 21 July 2022

Keywords:
CP decomposition
Tensor completion
Cyclops tensor framework

ABSTRACT

Tensor computations are increasingly prevalent numerical techniques in data science, but pose unique challenges for high-performance implementation. We provide novel algorithms and systems infrastructure which enable efficient parallel implementation of algorithms for tensor completion with generalized loss functions. Specifically, we consider alternating minimization, coordinate minimization, and a quasi-Newton (generalized Gauss-Newton) method. By extending the Cyclops library, we implement all of these methods in high-level Python syntax. To make possible tensor completion for very sparse tensors, we introduce new multi-tensor primitives, for which we provide specialized parallel implementations. We compare these routines to pairwise contraction of sparse tensors by reduction to hypersparse matrix formats, and find that the multi-tensor routines are more efficient in theoretical cost and execution time in experiments. We provide microbenchmarking results on the Stampede2 supercomputer to demonstrate the efficiency of the new primitives and Cyclops functionality. We then study the performance of the tensor completion methods for a synthetic tensor with 10 billion nonzeros and the Netflix dataset, considering both least squares and Poisson loss functions.

© 2022 Elsevier Inc. All rights reserved.

1. Introduction

Emerging sparse tensor methods pose new challenges for high-performance programming languages and libraries. This paper describes new steps in making high-level productive parallel programming for sparse tensor algebra possible. We focus specifically on formulation and implementation of optimization algorithms for tensor completion, which require management of extremely sparse tensors and complicated tensor operations. We provide algorithms and software for tensor completion with generalized loss function [32]. Algorithms for the least-squares loss tensor completion have been a target of recent parallel implementation efforts [35,64]. We provide new formulations of a variety of tensor completion algorithms for the generalized loss function based on a common set of basic kernels. We implement these kernels to provide a new programming abstraction and software infrastructure for distributed-memory sparse tensor optimization algorithms. By extending Cyclops [67], we provide a Python-level interface to these sparse tensor kernels that achieve scalability on high performance distributed-memory architectures.

Tensor completion [50], a generalization of the matrix completion problem, is the task of building a model to approximate a tensor based on a subset of observed entries. The model should accurately represent observed entries, generalize effectively to unobserved entries, have a concise representation, provide efficient prediction of any tensor entry, and be possible to optimize. Low-rank matrix factorizations are a widely used model for matrix completion, while tensor decompositions [43], especially the canonical polyadic (CP) decomposition [31,43], are commonly used for tensor completion [21]. The major computational challenge in tensor completion is the optimization of the model, i.e., computation of a low-rank CP decomposition that effectively approximates the observed entries [35].

We consider four optimization methods for tensor completion with least squares loss, three of which are described in Section 2 and introduce novel algorithms for tensor completion with general loss functions in Section 3. Alternating minimization or alternating least squares for least squares loss (ALS) updates one factor matrix while keeping other factor matrices fixed for each step, yielding a symmetric positive definite (for convex loss functions) linear system of equations to be solved. Coordinate minimization or coordinate descent for least squares updates one column of a

* Corresponding author.
E-mail addresses: navjot2@illinois.edu (N. Singh), solomon2@illinois.edu (E. Solomonik).

https://doi.org/10.1016/j.jpdc.2022.07.005
0743-7315/© 2022 Elsevier Inc. All rights reserved.

factor matrix while keeping others fixed for each step and alternates among factor matrices in a cyclic manner. Compared to ALS, coordinate descent performs updates with less computational cost, but reduces the minimization objective more slowly in each sweep of update. Stochastic gradient descent (SGD) randomly selects samples from the tensor at each iteration and optimizes all the factor matrices based on these entries with a gradient-based update.

Second-order algorithms like Newton's method and Gauss-Newton method for CP decomposition with least-squares loss have been shown to perform better than ALS when the factors are highly correlated and an accurate solution is required [70,2]. However, each iteration of these algorithms is expensive and naive approaches do not scale due to the size of linear system required to solve. For the decomposition problem, the Gauss-Newton method can be implemented efficiently using an implicit form of the Hessian for fast matrix-vector products in the Conjugate gradient algorithm [70,60]. Recently, a second order (Gauss-Newton like) method was proposed for the generalized decomposition problem [77]. In [77], the structure of the Hessian of the generalized decomposition problem is explored and the method is shown to perform better than the gradient-based LBFGS [32] for beta divergence loss functions. In Section 3.3, we use tensor algebra to introduce a new formulation of the Newton's method and consequently a quasi-Newton method for generalized tensor completion which leverages an implicit form of the Hessian arising in the completion problem. Note that if the number of missing entries is set to zero, the problem would become a decomposition problem and the implicit form of the Hessian would correspond to the Hessian constructed in [77]. The implicit form of the Hessian can be leveraged to solve the linear system involving the large Hessian by use of batched conjugate gradient. We show that the implicit matrix-vector products can be efficiently computed with basic sparse tensor algebra operations and the overall method achieves a lower computational cost than a direct solve [77,54].

To achieve high-performance for sparse tensor completion, we extend the functionality for sparse tensor contractions included in Cyclops [66,69,68]. Since tensor completion is often done with extremely sparse tensor datasets [65,50], the use of CSR sparse matrix format for contraction of local tensor blocks becomes inefficient, and hypersparse matrix formats [11] are necessary. We add support for a CSF sparse matrix format to Cyclops (described in Section 4.1), which requires $O(m)$ memory for a tensor with m nonzeros, and provides functionality to support contraction of a sparse and dense tensors (into a sparse output) using the hypersparse format. Support of this format in a distributed memory library imposes new challenges, such as the necessity to perform summation and distributed reduction of blocks in hypersparse format. To the best of our knowledge, Cyclops is the first distributed tensor library to offer this functionality.

We also identify a common generic multi-tensor routine that arises in tensor completion, and is likely to be very useful in generalized CP decomposition of sparse tensors as well as other applications. This routine cannot be executed efficiently by the standard approach of pairwise contraction of tensors which is typically used by Cyclops. Therefore, in Section 4.3, we introduce a programming abstraction for this tensor-times-tensor-product (TTTP) routine that achieves lower cost and memory footprint via a specialized parallel implementation. Specifically, the TTTP routine multiplies entries of a sparse tensor with corresponding multilinear inner products of vectors. Alternating minimization requires computation of tensor contractions with a sparse tensor along with solving systems on the fly to avoid overheads in memory footprint. To address this, we provide a programming abstraction for a sub-iteration of the algorithm involving specialised sparse tensor contractions. We develop library routines that map sparse tensor contractions to sparse matrix products and specialised multilinear operations.

We develop parallel implementations of the tensor completion methods leveraging a new Python interface to Cyclops (described in Section 5). This interface provides routines for Einstein-summation-like contraction of tensors, TTTP, and a multitude of other operations manipulating sparse and dense tensors. The functionality is interfaced via Cython [6] to the C++ core of Cyclops. Cyclops itself uses MPI, OpenMP, and CUDA to perform tensor algebra and data transformations/redistribution. A basic set of parallel dense linear algebra routines are made available by interfacing to ScaLAPACK [9]. The Python interface implements much of the basic functionality of `numpy.ndarray`, making it possible to easily transform sequential Python dense tensor codes to distributed-memory-parallel sparse tensor software.

We provide performance results on the Stampede2 supercomputer for redistribution, tensor contractions, TTTP, and tensor completion algorithms with Cyclops. Our results demonstrate that the new hypersparse representations enable contraction of tensors with extremely low density and that our new specialized TTTP algorithm achieves much better scalability than when done by pairwise contraction. Finally, our tensor completion results show the capability of a high-level Python implementation of tensor completion methods to scale to tens of thousands of cores and 10B nonzeros of a highly sparse (10^{-5} density) tensor. As an example of our algorithmic and software framework for generalized tensor completion, we provide an implementation of tensor completion algorithms for least-squares loss and the first distributed memory implementation of tensor completion algorithms for Poisson loss with logarithm link function [32], and evaluate their performance on the Netflix dataset [7] for tensor completion.

This paper makes the following contributions:

- novel formulation of the second order algorithm for generalized tensor completion that uses implicit conjugate gradient and is easily implementable with tensor algebra kernels,
- novel algorithms for alternating minimization and coordinate minimization for generalized tensor completion,
- novel infrastructure in hypersparse matrix formats for parallel sparse tensor contractions,
- a new programming abstraction for products of sparse tensors and tensor products (TTTP) and solving the linear systems arising in alternating minimization for generalized tensor completion (Solve Factor),
- novel support of distributed-memory sparse tensor algebra operations in Python by a new interface to Cyclops,
- first parallel implementation of generalized tensor completion algorithms that use second order information.

2. Tensor completion background

A tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ has **order** N (i.e. N modes/indices), **dimensions** I_1-by-$\ldots$-by-I_N and **elements** $t_{i_1 \ldots i_N} = t_i$ where $i \in \bigotimes_{i=1}^{N} \{1, \ldots, I_i\}$. Order N tensors can be represented by $N-$dimensional arrays. The algorithms and techniques involved in tensor completion do not differ significantly for tensors of order 3 or larger, and many tensor datasets are order 3, so we focus on this case for simplicity of presentation.

2.1. Tensor completion by generalized CP decomposition

The canonical polyadic (CP) decomposition [31] of an order three tensor $\mathcal{T} \in \mathbb{R}^{I \times J \times K}$ has the form,

$$t_{ijk} = \sum_{r=1}^{R} u_{ir} v_{jr} w_{kr}, \tag{1}$$

Table 1
First and second order derivative information.

Derivatives	General loss function	Least squares loss function (4)
$\nabla f(\boldsymbol{u_i})$	$\sum_{(j,k)\in\Omega_i}(\boldsymbol{v}_j \odot \boldsymbol{w}_k)\phi'_{ijk}$	$\sum_{(j,k)\in\Omega_i}(\boldsymbol{v}_j \odot \boldsymbol{w}_k)\Big(\langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k\rangle - t_{ijk}\Big)$
$\frac{\partial f(u_{ir})}{\partial u_{ir}}$	$\sum_{(j,k)\in\Omega_i} v_{jr}w_{kr}\phi'_{ijk}$	$\sum_{(j,k)\in\Omega_i} v_{jr}w_{kr}\Big(\langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k\rangle - t_{ijk}\Big)$
$\boldsymbol{H}_f(\boldsymbol{u}_i)$	$\sum_{(j,k)\in\Omega_i}(\boldsymbol{v}_j \odot \boldsymbol{w}_k)^T\phi''_{ijk}(\boldsymbol{v}_j \odot \boldsymbol{w}_k)$	$\sum_{(j,k)\in\Omega_i}(\boldsymbol{v}_j \odot \boldsymbol{w}_k)^T(\boldsymbol{v}_j \odot \boldsymbol{w}_k)$
$\frac{\partial^2 f(u_{ir})}{\partial u_{ir}^2}$	$\sum_{(j,k)\in\Omega_i} v_{jr}^2 w_{kr}^2\phi''_{ijk}$	$2\sum_{j,k}\hat{\Omega}_{ijk}v_{jr}^2w_{kr}^2$

where R is referred to as the **rank** of the decomposition and $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$ as **factor matrices**. Letting $\langle\cdot,\cdot,\cdot\rangle$ denote a trilinear inner product, we can rewrite the above in terms of the rows $\boldsymbol{u}_i$, $\boldsymbol{v}_j$, $\boldsymbol{w}_k$ of the factor matrices,

$$t_{ijk} = \langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k\rangle. \tag{2}$$

The set of observed entries of a tensor $\mathcal{T}$ may be represented by index set $\Omega \subset \{1,\dots,I\}\times\{1,\dots,J\}\times\{1,\dots,K\}$, so that for all $(i,j,k)\in\Omega$, t_{ijk} has been observed. The objective function that we seek to minimize is

$$f(\boldsymbol{U},\boldsymbol{V},\boldsymbol{W}) = \underbrace{\sum_{(i,j,k)\in\Omega}\phi(t_{ijk},\langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle)}_{\text{Sum of elementwise loss function defined on observed entries}} + \underbrace{\lambda(\|\boldsymbol{U}\|_F^2+\|\boldsymbol{V}\|_F^2+\|\boldsymbol{W}\|_F^2)}_{\text{regularization to prevent overfitting}}. \tag{3}$$

where $\phi:\mathbb{R}\times\mathbb{R}\to\mathbb{R}$ is an arbitrary convex function that can be chosen according to the inherent data distribution. Various potential choices of these functions have been discussed in [32].

Most of the previous work for CP tensor completion is related to least-squares loss function, i.e., by using the following elementwise function in the equation (3),

$$\phi(t_{ijk},\langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle) = \frac{1}{2}(t_{ijk}-\langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle)^2, \tag{4}$$

which assumes that the error in data is normally distributed. However, many datasets that we encounter do not satisfy this assumption, but may fall in a different category, for example, a dataset of counts might follow Poisson distribution with the elementwise loss function,

$$\phi(t_{ijk},\langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle) = \langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle - t_{ijk}\log\langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle,$$
$$\text{where } \langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle > 0.$$

For further sections, we will use a shorthand notation for the function $\phi^{(t_{ijk})}:\mathbb{R}\to\mathbb{R}$ as ϕ_{ijk} which assumes that the first input to the corresponding binary input function, ϕ, is t_{ijk}. And further, we define

$$\phi'_{ijk} = \frac{\partial\phi(t_{ijk},m_{ijk})}{\partial m_{ijk}}, \quad \text{where } m_{ijk} = \langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle \tag{5}$$

and ϕ''_{ijk} accordingly.

We review tensor completion algorithms for least squares loss in this section, and generalize these algorithms to introduce novel algorithms that use second order information for tensor completion with general loss functions in Section 3. We present a table of derivative information in Table 1, which is used throughout this section and Section 3 to formulate the algorithms. Derivations of these expressions are provided in the Appendix A. Note that $\odot$ is the Hadamard/pointwise product and we omit the regularization term for factors which maybe trivially added.

2.2. Alternating least squares

Alternating least squares (ALS) method is a standard algorithm for CP decomposition with least-squares loss of tensors. Fixing all except one factor matrices results in a quadratic subproblem which can be solved with a single Newton's step. Defining $\hat{\Omega}_{ijk}=1$ if $(i,j,k)\in\Omega$ and 0 otherwise, and taking $t_{ijk}=0$ if $(i,j,k)\notin\Omega$, and using Table 1 for gradient and Hessian values with an initial guess of zeros for each row of the factor matrix, i.e., $\boldsymbol{u}_i=\boldsymbol{0}$, the updated matrix $\boldsymbol{U}^{(\text{new})}$ can be expressed as a system of equations involving sparse tensor contractions,

$$\sum_r u_{ir}^{(\text{new})}(g_{rs}^{(i)}+\lambda\delta_{rs}) = \sum_{j,k} v_{js}w_{ks}t_{ijk},$$
$$\text{where} \quad g_{rs}^{(i)} = \sum_{j,k} v_{jr}w_{kr}\hat{\Omega}_{ijk}v_{js}w_{ks}.$$

Given $m=|\Omega|$ observed values, solving the linear systems has cost $O(IR^3)$, forming the right-hand sides has cost $O(mR)$, and computing the matrices $\boldsymbol{G}^{(i)}$ has cost $O(mR^2)$. Contracting two tensors at a time to form each $\boldsymbol{G}^{(i)}$ all at once incurs additional memory footprint, specifically,

$$O(\min(\text{median}(I,J,K)R^2, mR) + LR^2), \text{ where}$$
$$L = \text{median}(|\{(j,k):(i,j,k)\in\Omega\}|, |\{(i,k):(i,j,k)\in\Omega\}|,$$
$$|\{(i,j):(i,j,k)\in\Omega\}|).$$

2.3. Coordinate descent

Coordinate descent updates a single variable of a factor matrix while keeping all others fixed. Each variable in a column of the factor matrix becomes independent, and therefore the whole column of a factor can be updated simultaneously. An iteration of coordinate descent is analogous to a step of ALS with a rank $R=1$ CP decomposition. The main advantage of coordinate descent over ALS is the lack of a need to solve systems of linear equations. A step of the algorithm can be computed using the values from Table 1,

$$u_{ir}^{(\text{new})} = u_{ir} + \Delta u_{ir},$$
$$\Delta u_{ir} = \left(-\lambda u_{ir} + \sum_{(j,k)\in\Omega_i} v_{jr}w_{kr}\big(t_{ijk}-\langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle\big)\right)\Big/ \left(\lambda + \sum_{(j,k)\in\Omega_i} v_{jr}^2w_{kr}^2\right).$$

Similar to ALS, we can use an initial guess of zeros, i.e., $u_{ir}=0$ and define

$$\rho_{ijk}^{(r)} = \begin{cases} t_{ijk}-\langle \boldsymbol{u}_i,\boldsymbol{v}_j,\boldsymbol{w}_k\rangle + u_{ir}v_{jr}w_{kr}, & \text{if } (i,j,k)\in\Omega \\ 0 & \text{otherwise.}\end{cases}$$

The update can be expressed with sparse tensor contractions,

$$u_{ir}^{(\text{new})} = \Delta u_{ir} = \left(\sum_{j,k} v_{jr} w_{kr} \rho_{ijk}^{(r)}\right) / \left(\lambda + \sum_{j,k} \hat{\Omega}_{ijk} v_{jr}^2 w_{kr}^2\right).$$

These contractions can be performed with $O(m)$ cost to update each u_{ir} for all i, and $\rho_{ijk}^{(r+1)}$ can be obtained from $\rho_{ijk}^{(r)}$ with $O(m)$ cost. Consequently, coordinate descent also requires $O(mR)$ cost to update all factor matrix entries, but has less parallelism and generally makes less progress than ALS since the updates to elements of factor matrix rows are decoupled. Our CCD implementation alternates between factor matrices for each column update, which corresponds to the CCD++ ordering [81].

2.4. Stochastic gradient descent

Instead of solving for a subset of variables at a time, one can solve for all the variables in an iteration using the first order or the gradient information. The simplest algorithm which uses gradient information for all the variables is gradient descent. The values for the derivative with respect to each element can be used from Table 1 yielding the following update,

$$u_{ir}^{(\text{new})} = u_{ir} - \eta \frac{\partial f}{\partial u_{ir}}$$

with a cost of $O(mR)$.

Since more accurate updates with monotonic convergence guarantees can be obtained with similar cost via ALS or coordinate descent, gradient descent is generally less efficient for tensor completion. However, stochastic gradient descent offers a framework in which the initial tensor can be sampled, leading to cost $O(SR + (I + J + K)R)$ (where S is the sample size) for a sweep that updates all factor matrices.

For a general objective function an extra computational cost of $O(m)$ ($O(S)$ in case of stochastic gradient descent) is required to compute ϕ'_{ijk} tensor to compute the gradient for each factor matrix. A distributed memory implementation of SGD for tensor decomposition with generalized loss functions was recently released [18]. Apart from stochastic gradient descent, LBFGS is another gradient based method that has been explored for generalized tensor decomposition in [32]. We do not consider LBFGS in this work.

3. Algorithms using second order information for generalized tensor completion

In prior work, use of second order information for tensor completion with loss functions other than least squares loss is largely unexplored. Alternating minimization and coordinate minimization have been proposed for tensor decomposition with Poisson loss function with identity-link [25]. A parallel implementation of these algorithms has been investigated recently [73]. A quasi-Newton algorithm for generalized tensor decomposition has been formulated recently [77], however the sparsity in Hessian due to missing data in the completion problem has not been discussed. In this subsection, we describe how the formulation of ALS and coordinate descent as Newton's iteration can be used to derive alternating minimization and coordinate minimization algorithms for generalized objective functions. We then introduce the Newton's and quasi-Newton algorithm for generalized tensor completion which uses an implicit Conjugate Gradient (CG) algorithm. We explore the sparse tensor structure of the Hessian matrix, which allows us to formulate matrix vector products in the CG algorithm as sparse tensor contractions that can be efficiently implemented via sparse tensor kernels introduced in this work.

3.1. Alternating minimization

Alternating minimization works by fixing all except one factor matrix at a time and solving the optimization problem with respect to that factor matrix optimally. For solving the resulting subproblem, each row of the factor matrix can be optimized via Newton's method,

$$\boldsymbol{u}_i^{(\text{new})} = \boldsymbol{u}_i + \Delta \boldsymbol{u}_i,$$
$$\text{where} \quad \Delta \boldsymbol{u}_i \boldsymbol{H}_f(\boldsymbol{u}_i) = -\nabla f(\boldsymbol{u}_i).$$

We derive the Hessian of the generalized completion objective (3) with respect to all the factor matrices in Appendix A. The diagonal blocks of this Hessian can be used to perform alternating minimization for general loss functions. A Newton's iteration is equivalent to a sub-iteration of the alternating least squares algorithm in terms of computational cost. Since the objective function is not quadratic, Newton's method may take more than one step unlike least-squares loss. The Hessian $\boldsymbol{H}_f(\boldsymbol{u}_i)$ and gradient $\nabla f(\boldsymbol{u}_i)$ for general loss functions with respect to each row $\boldsymbol{u}_i$ is given in Table 1, and can be used to compute the Newton's step. Note that for a general loss each step costs the same as an ALS sub-iteration except for the fact that ϕ'_{ijk} and ϕ''_{ijk} tensors would need to be computed beforehand, which require $O(mR)$ computational cost and $O(m)$ memory.

3.2. Coordinate minimization

Rather than updating the whole row of a factor matrix as in alternating minimization, coordinate minimization updates a single variable at a time while keeping the others fixed. A Newton's step for a single variable is given as

$$u_{ir}^{(\text{new})} = u_{ir} + \Delta u_{ir},$$
$$\text{where} \quad \Delta u_{ir} = -\frac{\frac{\partial f}{\partial u_{ir}}}{\frac{\partial^2 f(u_{ir})}{\partial u_{ir}^2}}.$$

For coordinate minimization of general loss functions, we can use the diagonal blocks of the Hessian used in alternating minimization and perform the Newton's iteration to solve for a column of the factor matrix. The values of derivatives for each element u_{ir} from Table 1 can be used to compute the Newton's step for the column with the same computational cost as for a CCD++ sub-iteration by using the trilinear product $\langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle$ to compute ϕ'_{ijk} and ϕ''_{ijk} tensors with an additional cost of $O(m)$ for updating both the tensors.

3.3. Newton's and quasi-Newton algorithms

In the regime of optimizing all variables at once, second order information can be used to obtain Newton's and quasi-Newton algorithms for the generalized completion problem. To minimize the objective, each iteration of the algorithm updates all the factor matrices by using the following update,

$$[\boldsymbol{U}^{(\text{new})}, \boldsymbol{V}^{(\text{new})}, \boldsymbol{W}^{(\text{new})}] = [\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}] + [\Delta \boldsymbol{U}, \Delta \boldsymbol{V}, \Delta \boldsymbol{W}],$$
$$\text{where} \quad [\Delta \boldsymbol{U}, \Delta \boldsymbol{V}, \Delta \boldsymbol{W}] = -\boldsymbol{H}_f^{-1}(\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}) \nabla f(\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}),$$

where $\boldsymbol{H}_f(\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W})$ is the Hessian or the approximated Hessian and $\nabla f(\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W})$ is the gradient for the objective function $f(\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W})$ in equation (3) with respect to all the factor matrices.

While the gradient can be computed efficiently, explicitly computing the Hessian or approximated Hessian and storing

it is extremely expensive as it is sparsity unaware requiring $O((I+J+K)^2R^2)$ memory. Moreover, directly inverting the matrix $\boldsymbol{H}_f(\boldsymbol{U},\boldsymbol{V},\boldsymbol{W})$ requires $O((I+J+K)^3R^3)$, which is practically infeasible for large scale tensors. Alternatively, we explore the implicit form of the Hessian for the generalized completion problem to formulate Newton and quasi-Newton algorithm that use CG algorithm with implicit matrix-vector products as applied in the CP decomposition with least-squares loss [60]. The convexity of the generalized tensor completion loss function ensures that the Hessian is positive semi-definite. The Gauss-Newton method approximates the Hessian by excluding the additional term required in the off-diagonal blocks of the Hessian, resulting in a quasi-Newton method for generalized objective function. With the implicit form of Hessian or approximated Hessian, the linear systems arising at each iteration can be solved by performing CG method with implicit matrix-vector products. Tensor contractions for updating the first factor matrix iterate in the implicit matrix-vector product in CG iteration for each Newton's iteration are

$$\sum_{s,l} h^{(1,1)}_{ilrs} x^{(1)}_{ls} = \sum_{s} \sum_{(j,k)\in\Omega_i} v_{jr} w_{kr} \phi''_{ijk} v_{js} w_{ks} x^{(1)}_{is},$$

$$\sum_{s,l} h^{(1,2)}_{ilrs} x^{(2)}_{ls} = \sum_{s} \sum_{(j,k)\in\Omega_i} v_{jr} w_{kr} \phi''_{ijk} u_{is} w_{ks} x^{(2)}_{js} + \sum_{(j,k)\in\Omega_i} w_{kr} \phi'_{ijk} x^{(2)}_{jr},$$

$$\sum_{s,l} h^{(1,3)}_{ilrs} x^{(3)}_{ls} = \sum_{s} \sum_{(j,k)\in\Omega_i} v_{jr} w_{kr} \phi''_{ijk} u_{is} v_{js} x^{(3)}_{ks} + \sum_{(j,k)\in\Omega_i} v_{jr} \phi'_{ijk} x^{(3)}_{kr},$$

where $\sum_{n=1}^{3}\sum_{s,l} h^{(1,n)}_{ilrs} x^{(n)}_{ls}$ corresponds to the first block of the matrix vector product of the Hessian and matrices $\boldsymbol{X}^{(1)}$, $\boldsymbol{X}^{(2)}$, and $\boldsymbol{X}^{(3)}$. The other two blocks of the matrix vector product can be computed similarly.

Each contraction of the type

$$\sum_{s}\sum_{j,k} x_{jr} y_{kr} \hat{t}_{ijk} u_{is} v_{js} w_{ks},$$

where $\hat{\mathcal{T}}$ is a sparse tensor, can be computed in $O(mR)$ cost by breaking it down into two contractions,

$$z_{ijk} = \hat{t}_{ijk} \sum_{s} u_{is} v_{js} w_{ks} \text{ and } a_{ir} = \sum_{j,k} z_{ijk} x_{jr} y_{kr},$$

each of which costs $O(mR)$. Therefore, a CG step for solving a system in the quasi-Newton algorithm costs $O(mR)$.

The computation cost of the quasi-Newton algorithm is dominated by CG iterations. The number of CG iterations can be reduced by using the block diagonal part of the Hessian as a pre-conditioner [60]. However, storing the explicit inverse of the diagonal blocks of $\boldsymbol{H}_f(\boldsymbol{U},\boldsymbol{V},\boldsymbol{W})$ may still be a memory bottleneck for large tensors. Instead, the inverse of a diagonal block of $\boldsymbol{H}_f(\boldsymbol{U},\boldsymbol{V},\boldsymbol{W})$ can be applied with a cost identical to solving for the linear systems in a sub-iteration of alternating minimization algorithm introduced above.

4. New sparse tensor kernels

The aforementioned tensor completion algorithms require sophisticated support for sparse tensor operations. We extend the Cyclops library for tensor computations, which already includes support for sparse tensor contractions, reducing these to matrix multiplication with CSR format locally. Cyclops leverages a cyclic data layout on multidimensional processor grids to achieve good performance and load balance for sparse tensors. However, we observe two major bottlenecks within the sparse tensor algebra operations required in tensor completion that warrant extensions of functionality.

We describe new infrastructure for hypersparse matrix formats, leveraging a doubly-compressed format, which is a special case of the compressed sparse fiber (CSF) layout [62,63]. We apply this infrastructure to obtain TTM and MTTKRP implementations that require a minimal amount of memory and flops [37,46,27,5]. Further, we provide a specialized all-at-once implementation of MTTKRP that is within a factor of four of specialized MTTKRP libraries. Additionally, we introduce a kernel for multiplication of a sparse tensor with multilinear inner products of vectors (TTTP), resulting in an output sparse tensor of the same size. Our parallelization of the kernel leverages batching to achieve lower memory-footprint than previous work [64,35]. TTTP generalizes the sampled dense–dense matrix multiplication (SDDMM) kernel [14,53,40], and is useful also for CP decomposition of sparse tensors.

We also introduce a kernel for solving linear systems on the fly arising in alternating minimization of the generalized objective function for CP completion (3) involving sparse tensors. For the special case of least squares loss, we achieve a comparable performance to the state of the art library for performing ALS completion [64].

4.1. Hypersparse matrix formats

Tensor contractions can be reduced to matrix multiplication with matrices that have the same number of sparse entries. However, while it is uncommon in sparse matrix computations for entire rows or columns of a sparse matrix to be zero, the sparse matrix–matrix products occurring by reduction from tensor computations often have this property [63]. A canonical example is the product of a sparse tensor and a dense matrix, which can be used an initial step for MTTKRP, yielding an intermediate that can typically be reused in multiple MTTKRP operations via dimension trees [38] (also see [58,78,5]). In this tensor times matrix (TTM) operation, given an order three tensor, we seek to compute

$$z_{ijr} = \sum_{k} t_{ijk} w_{kr},$$

where $\mathcal{T}$ is sparse and $\boldsymbol{W}$ is dense. By merging i and j into a single index, TTM reduces to a matrix-matrix product of sparse and a dense matrix. For $\mathcal{T} \in \mathbb{R}^{I\times J\times K}$, if the number of entries in $\mathcal{T}$ is less than IJ, then the above matricization of $\mathcal{T}$ is necessarily hypersparse (contains rows with only zero entries), and $\mathcal{Z}$ is sparse. For many sparse tensor datasets, one of the modes is small, or the number of nonzeros scales with mode size, i.e., $m = O(I+J+K)$. In both cases, we may obtain a matricization that is very hypersparse (most rows are zero), in which case the matricization of $\mathcal{Z}$ cannot be stored in a dense format without increasing memory footprint.

Cyclops represents static sparse tensor data in a COO-like format, storing a single 64-bit integer for each value to encode its global location in the tensor, with index-value pairs sorted locally. When a contraction is executed, the locally stored portion of the tensor is transformed into a CSF [63] sparse matrix format. In this format, we use CSR to encode the nonzero rows and an additional array is stored that maps nonzero rows to the original set of rows. This matrix CSF format also corresponds to the DCSR [11] format without the use of chunks (no AUX array). This layout requires $\Theta(m)$ storage if a tensor has m nonzeros, improving on $\Theta(IJ+m)$ needed for plain CSR for the TTM operation above. Multiplication

of a matrix by a dense matrix is easy, it suffices to multiply the reduced CSR matrix by the dense matrix, then generate a new CSF matrix to represent the sparse output, resulting in $O(mR)$ cost.

Realizing CSF functionality for arbitrary tensor contractions also necessitates implementation of sparse format conversions, summation of CSF blocks, and interprocessor reduction. We provide kernels for each of these steps. When sparsity is involved, Cyclops first ensures that each index arising in the tensor contraction expression occurs in exactly two tensors. If an index occurs in only a single tensor, pre- or post- processing can be performed to reduce or map the input or output, respectively. If an index occurs in all three tensors (specifying a set of independent contractions), Cyclops duplicates the index, converting one of the sparse operands to a tensor of one order higher, placing the original data on the diagonal (e.g. $c_i = v_i w_i$ with sparse $\boldsymbol{v}$ is performed via $\boldsymbol{c} = \hat{\boldsymbol{V}}\boldsymbol{w}$ where $\hat{v}_{ii} = v_i$). By ensuring that each index occurs in exactly two tensors, Cyclops is able to map the local part of the contraction to a matrix–matrix product. Cyclops puts local parts of the tensor into sparse matrix format by first converting to COO then to CSF matrix format (for a standard sparse format, conversion to CSR works similarly).

Local summation of CSF matrices requires identifying which rows are nonzero in both matrices, which is done by comparing the two sets of nonzero row indices. The summation of each row is done by leveraging a dense array. In particular, if each local matrix has K columns, nonzeros in that row are accumulated to the corresponding entries of an array of size K, then the sparse sum is read back and the entries used are zeroed out. The cost of this operation for summing each row scales with the number of nonzeros in the output row, but the buffer must be allocated and cleared, creating a potential bottleneck if the local sparse matrices are very hypersparse in both rows and columns (most rows and most columns are entirely zero). For sparse tensor times matrix contractions arising in the tensor completion kernels, each column contains nonzeros.

Parallel reduction of CSF matrices leverages this summation kernel, using a butterfly collective communication approach (recursive halving followed by recursive doubling [74]) that performs a sparse reduce-scatter followed by a sparse gather. At each step of the sparse reduce-scatter, hypersparse matrices with smaller overall dimensions but higher density are summed by each processor using the sparse summation kernel described above. An example of the reduce-scatter is displayed in Fig. 1. The sparse gather recombines these matrices by concatenation. The partitioning and recombination is done using a k-ary butterfly, where k is a parameter that we chose to be a constant.

4.2. Matricized tensor times Khatri-Rao product

While the use of hypersparse formats enables an implementation of MTTKRP that asymptotically minimizes memory footprint and cost, we also provide a specialized MTTKRP implementation that performs the operation in an all-at-once manner. In particular, the MTTKRP is parallelized by performing smaller local MTTKRPs on each processor, using the sparse tensor data stored on that processor. This parallelization follows SPLATT [62,63] and also uses a reduction to accumulate results. However, the MTTKRP kernel interoperates with other Cyclops functionality, redistributing factor matrices from an arbitrary initial layout, to a partially-replicated distribution necessary to compute the local MTTKRP, and the resulting matrix is put into a layout that is distributed over all processors. Locally, the sparse tensor data is kept in the usual Cyclops COO-like format, as opposed to the specialized CSF format. Partial sums are accumulated for the local part of each tensor fiber along the most quickly changing index. The BLAS `axpy` operation and the MKL vector pointwise product are used to achieve vectorization when the MTTKRP is performed with factor matrices that have more than one column.

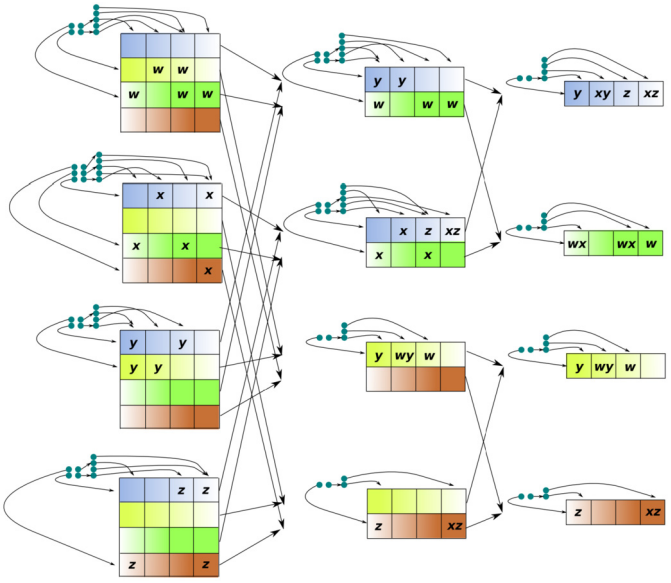


Fig. 1. Depiction of 4 processor reduce-scatter of 4×4 hypersparse matrices stored in doubly compressed (CCSR) format.

4.3. Tensor times tensor product

Efficient support for sparse tensor contractions does not suffice for tensor completion algorithms. Their use entails significant overhead in memory footprint even to just compute the residual,

$$\rho_{ijk} = t_{ijk} - \sum_{r=1}^{R} \hat{\Omega}_{ijk} u_{ir} v_{jr} w_{kr},$$

since forming intermediate or $x_{ijkr} = \hat{\Omega}_{ijk} u_{ir}$ increases memory footprint, while alternatively forming the dense intermediate $y_{ijk} = \sum_{r=1}^{R} u_{ir} v_{jr} w_{kr}$ is suboptimal in both memory footprint and work. Evidently, the most efficient way to perform such operations requires all-at-once contraction of multiple operands. To handle this operation effectively, we introduce the tensor-times-tensor product (TTTP) operation, which takes as input a sparse tensor $\boldsymbol{\mathcal{S}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and a list of up to N matrices $\boldsymbol{A}^{(1)} \in \mathbb{R}^{I_1 \times R}, \ldots, \boldsymbol{A}^{(N)} \in \mathbb{R}^{I_N \times R}$ and computes

$$x_{i_1 \ldots i_N} = s_{i_1 \ldots i_N} \sum_{r=1}^{R} \prod_{j=1}^{N} a_{i_j r}^{(j)}.$$

If fewer then N matrices are specified, the product should iterate only over modes for which an input is provided. By iterating over m nonzero entries in $\boldsymbol{\mathcal{S}}$ and performing the multilinear inner product for each one, TTTP can be performed with cost $O(mR)$ and $O((I_1 + \cdots + I_N)R + m)$ memory footprint. When $N = 2$, TTTP corresponds to the SDDMM operation $\boldsymbol{X} = \boldsymbol{S} \odot (\boldsymbol{U}\boldsymbol{V}^T)$.

TTTP is an integral part of the algorithms for generalized tensor completion as it is used to compute ϕ'_{ijk} (equation (5)) and ϕ''_{ijk}. For the traditional tensor completion with least squares loss, TTTP allows calculation of the residual in tensor completion with CP decomposition, by computing

$$\hat{\Omega}_{ijk} \sum_{r=1}^{R} u_{ir} v_{jr} w_{kr}.$$

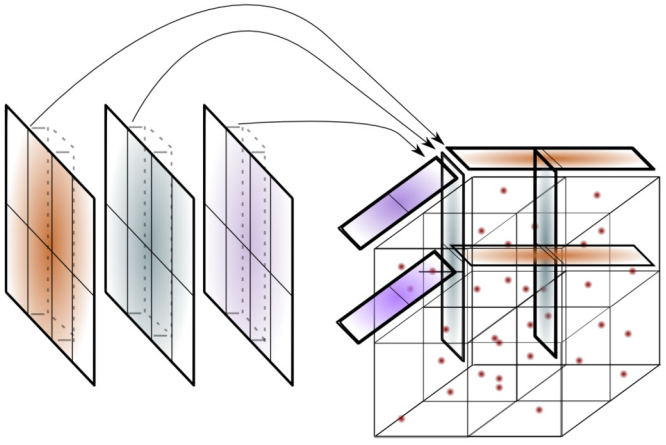


Fig. 2. Depiction of 8 processor parallelization of TTTP computing one of four smaller TTTP substeps.

Although, this residual calculation can be accelerated in ALS, it is explicitly necessary in the coordinate minimization algorithm. Further, for the Newton's and quasi-Newton method with implicit conjugate gradient in Section 3.3, we use TTTP to compute updates via

$$z_{ijk} = \underbrace{\phi''_{ijk} \sum_s v_{js} w_{ks} x_{is}}_{\text{TTTP}}, \quad x^{(new)}_{ir} = \underbrace{\sum_{j,k} v_{jr} w_{kr} z_{ijk}}_{\text{MTTKRP}}.$$

Our parallel implementation of TTTP keeps the sparse tensor input $\mathcal{S}$ and output $\mathcal{X}$ local on whichever processor grid $\mathcal{S}$ was initially distributed on. The matrices $\boldsymbol{A}^{(1)}, \ldots, \boldsymbol{A}^{(N)}$ are input from an arbitrary initial processor grid distribution. Each matrix is sliced into $H \leq R$ pieces by taking H equal-sized subsets of their columns, based on available memory. The computation then proceeds in H steps, each computing a smaller TTTP involving matrices of size $I_j \times (R/H)$. For each step, the corresponding slice of each of the N matrices $\boldsymbol{A}^{(j)}$ is redistributed so that its rows are cyclically distributed over the processor grid dimension along which the jth mode of $\mathcal{S}$ is distributed (if any), and replicated over all others. Each of P processors can then compute a part of the smaller TTTP with the entries of $\mathcal{S}$ (and $\mathcal{X}$) it is assigned locally, performing a total of $O(mR/P)$ work overall.

This parallel TTTP algorithm is depicted in Fig. 2 for scenario with $P = 8$ processors. Assuming a $I = I_1 = \cdots = I_N$ and a processor grid is used of dimensions $P^{1/N} \times \cdots \times P^{1/N}$, using a BSP model of communication [61,75], the latency cost (number of supersteps) is $O(H)$, the interprocessor bandwidth cost is $O(IR/P^{1/N})$, and the memory footprint is $O(m/P + IR/(P^{1/N}H))$. Efficient mechanisms for redistribution of dense matrices between arbitrary processor grids exist in Cyclops [68].

4.4. Solve factor

Alternating minimization for generalized CP tensor completion requires tensor contractions along with a solve which should be done on the fly to avoid a memory bottleneck. To accomplish this, we provide a specialized kernel, which uses a similar parallelization strategy suggested in [64] for ALS completion. Our kernel takes as input a tensor $\mathcal{S} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, a list of up to N matrices $\boldsymbol{A}^{(1)} \in \mathbb{R}^{I_1 \times R}, \ldots, \boldsymbol{A}^{(N)} \in \mathbb{R}^{I_N \times R}$, an integer n, a right hand side matrix $\boldsymbol{M} \in \mathbb{R}^{I_n \times R}$, and solves for the Newton's step with respect to n^{th} factor matrix as described in Section 3.1.

The left hand sides for i_n^{th} row of the factor matrix in the Newton's step, $\boldsymbol{G}^{(i_n)}$, can be computed by the following contractions,

$$g^{(i_n)}_{rs} = \sum_{i_1 \ldots i_{n-1}, i_{n+1} \ldots i_N} \left(\prod_{p=1, p\neq n}^{N} a_{i_p r} \right) s_{i_1 \ldots i_N} \left(\prod_{p=1, p\neq n}^{N} a_{i_p s} \right),$$

which together incur a computational cost of $O(mR^2)$ and a memory footprint of $O(I_n R^2)$.

Our parallel implementation follows the same strategy as in Section 4.3 and keeps the sparse tensor input $\mathcal{S}$ local on whichever processor grid it was initially distributed on. The matrices $\boldsymbol{A}^{(1)}, \ldots, \boldsymbol{A}^{(N)}$ are input from an arbitrary initial processor grid distribution and are redistributed as described in Section 4.3. To make use of BLAS-3 operations for the above mentioned contractions, the input tensor must be sorted with respect to the mode n. We accomplish this in the current format by performing a Counting sort [17] using the indices of the n^{th} mode as keys over the local data with a computational cost of $O(m/P)$.

Forming left hand sides for the normal equations for all the rows can be a memory bottleneck due to a memory footprint of $O(IR^2/P^{1/N})$. We divide the rows into b batches according to the memory available, leading to a memory footprint of $O(IR^2/(bP^{1/N}))$ and then for computing the normal equations for each row, we store the hadamard products of the vectors multiplied with the square root of corresponding tensor entries in a local buffer of size $K \times R$ by using MKL for pointwise vector products. When the buffer is filled up or the number of entries are exhausted, a symmetric rank-k (SYRK) update is performed using BLAS to compute the local left hand sides. A reduce scatter along slice with respect to mode n of the processor grid allows us to scatter the computed left hand sides. The corresponding right hand sides are distributed and a symmetric positive definite solve routine in BLAS (POSV) is used to achieve a parallel solve with $O(IR^3/(bP^{\frac{N+1}{N}}))$ cost for a batch of rows.

5. Python interface and implementation

Cyclops [68] provides extensive support for tensor algebra and tensor data manipulation in C++, leveraging BLAS [44], MPI [23], OpenMP, CUDA, HPTT [71], and ScaLAPACK [9]. The library supports both dense tensor formats [68] as well sparse tensor formats [66], both of which leverage partitioning of the tensor data among all processors. Scaling, summation, and contraction are supported via a succinct programmatic Einstein summation notation. Cyclops also provides general kernels such as tensor transposition, redistribution, slicing, and permutation of tensor indices. Additionally, the library supports user-defined element types and algebraic structures specifying their properties, as well as contractions that operate on tensors of different types, enabling applications such as graph algorithms [69].

Cyclops leverages a runtime-centric execution model, making data distribution and algorithmic scheduling decisions at execution time. This enables performance models to be evaluated for runtime-determined parameters such as problem size and processor count. We leverage this characteristic of the Cyclops system architecture to provide a performance-efficient Python interface to Cyclops. This extension enables productivity for high-performance implementation of tensor computations. By implementing a backend for high-level NumPy-style operations [76], we activate support for sparsity in tensor storage and computations, as well as parallel execution in distributed and shared memory. These capabilities are enabled with minimal overhead to the user. For example, by using Cyclops, sparse storage for a code based on the standard `numpy.ndarray` can be implemented simply by an additional boolean flag in the `ctf.tensor` constructor. By contrast, the standard approach for supporting sparse matrix operations in Python, involves manual handling of the CSR format via SciPy [34].

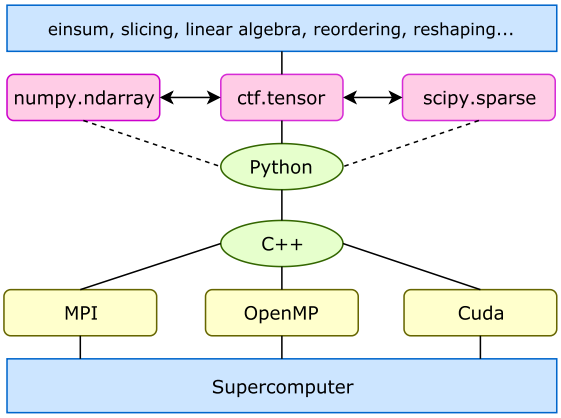


Fig. 3. Overview of Cyclops Python interface organization.

5.1. Cyclops python interface

We utilize Cython [6], which enables interoperability of Python and C++, to encapsulate the main functionalities of Cyclops C++ interface. As shown in Fig. 3, we introduce a Python tensor class that wraps the C++ Cyclops tensor object via Cython and provides the core functionality. Tensor and multidimensional array operations are also built on C++ interface functionalities including `ctf.einsum`, `ctf.tensordot`, `ctf.transpose`, and `ctf.reshape`. Functionality provided by NumPy in `numpy.linalg` is also supported, including QR, Cholesky, SVD, and the symmetric eigensolve. Boolean, integer, and floating point types of a variety of precision are supported, which are specified via `numpy.ndarray.dtype`. The C++ interface of Cyclops uses templating to support arbitrary types and user-defined elementwise operations, so extension of the Python interface to other types is possible. Dense and sparse distributed Cyclops tensors may be defined in a variety of ways.

Listing 1 Example Code: Tensor Initialization.

```
import ctf
U = ctf.tensor([5,7]) # dense zero matrix
M = ctf.random.random((4,4)) # random dense tensor
O = ctf.ones((4,3,5)) # tensor full of ones
I = ctf.eye(9) # dense identity
T = ctf.tensor([5,3,4], sp=True) # sparse tensor
T.fill_sp_random(-1.,1.,.1) # 10% density
S = ctf.speye(9) # sparse identity
```

For both the dense tensor and sparse tensor, NumPy-style indexing/slicing is provided such as `A[0, 1]` to extract a_{01} or `A[3:5, 1:4:2]` to extract a 2-by-2 matrix containing entries at the intersection of rows 3 and 4 and columns 1 and 3. A key difference between the Cyclops Python interface and NumPy functions including slice and (transpose `A.T`) is that Cyclops explicitly creates the new tensor in memory as opposed to providing a logical reference. For example with the Cyclops interface, transposition is done via `B = A.T()`, which returns a new tensor (so modifying elements of B will not change A).

Cyclops supports both NumPy-style Einstein summation, as well as an additional Einstein syntax similar to its C++ interface. For example, the following two lines are equivalent.

Listing 2 Example Code: Einstein Summation.

```
R += T - ctf.einsum("ir,jr,kr->ijk",U,V,W)
R.i("ijk") << T.i("ijk")- U.i("ir")*V.i("jr")*W.i("kr")
```

Expressions such as the above are passed directly to the C++ layer. The C++ layer then makes decisions regarding evaluation ordering and choice of intermediate tensors. Cyclops performs this by considering all possible binary trees for contraction of window of up to 8 tensors (contracting-away one tensor and including the next one given more than 8 operands), based on a heuristic model of computation and memory-bandwidth cost. Intermediate tensors are defined to be sparse if they are a contraction of two sparse operands or if a very sparse tensor is contracted with a dense tensor (contraction corresponds to a matrix–matrix product with a hypersparse matrix that must have fewer than 1 in 3 rows with a nonzero).

5.2. TTTP interface

Cyclops does not automatically determine when to use the multi-tensor TTTP operation. Instead, a simple interface is provided for this operation. For example, the following code computes

$$s_{ijkl} = \sum_r o_{ijkl} u_{ir} v_{jr} w_{kr} z_{lr}, \quad t_{ijkl} = \sum_r o_{ijkl} u_{ir} w_{kr}.$$

Listing 3 Example code: TTTP.

```
O = ctf.tensor((I,J,K,L),sp=True)
U = ctf.tensor((I,R)), V = ctf.tensor((J,R))
W = ctf.tensor((K,R)), Z = ctf.tensor((L,R))
... # fill O,U,V,W,Z
S = ctf.TTTP(Omega,[U,V,W,Z])
T = ctf.TTTP(Omega,[U,None,W,None])
```

The routine alternatively accepts a list of vectors rather than matrices as the second argument. A similar routine is available via the C++ interface to Cyclops.

5.3. Parallel tensor completion in python

Given high-level tensor algebra primitives, we are able to implement the aforementioned tensor completion algorithms without any explicit management of parallelism or data distribution. The problem of parallelization of these algorithms is reduced to expressing them with high-level tensor algebra operations.

5.4. Alternating minimization (alternating least squares) implementation

Alternating minimization algorithm can be implemented entirely using the MTTKRP kernel for the right hand sides and the Solve Factor kernel for the solves. Solving for one factor matrix can be implemented easily via the Cyclops Python interface.

Listing 4 ALS solve for one factor matrix ($\boldsymbol{U}$).

```
rhs = ctf.tensor((I,R))
ctf.MTTKRP(T,[rhs,V,W],0) #compute right hand sides
U = ctf.tensor((I,R))
ctf.Solve_Factor(Omega,[U,V,W],rhs,0,regu) #solve for U
```

5.5. Coordinate minimization (coordinate descent) implementation

The coordinate minimization updates are easy to formulate via Einstein notation contractions and elementwise operations.

For the second expression above, Cyclops finds the right tree of contractions automatically (note that a tree is more efficient

Listing 5 Example code: CCD++ Update Rule.

```
a = ctf.einsum('ijk,j,k->i',R,V[:,r],W[:,r])
b = ctf.einsum('ijk,j,j,k,k->i',
Omega,V[:,r],V[:,r],W[:,r],W[:,r])
U[:,r] = a / (lmbda + b)
```

than contracting left-to-right given any initial order). Slicing permits easy access of columns, although in our final implementation, we split up each factor matrix into column vectors outside of the CCD++ iteration loop to minimize overhead.

We also consider an implementation of CCD++ that is based on the MTTKRP kernel in Cyclops. This approach forgoes the need for tensor contractions with hypersparse matrix representations.

Listing 6 Example code: CCD++ with MTTKRP.

```
ctf.MTTKRP(R,[A,V[:,r],W[:,r]],0)
ctf.MTTKRP(Omega,[B,V[:,r]*V[:,r],W[:,r]*W[:,r]], 0)
```

5.6. Stochastic gradient descent implementation

We leverage a sampling function in the Cyclops Python interface to obtain a random sample of the tensor $\mathcal{T}$ for each SGD sweep (update to each factor matrix).

Listing 7 Example code: SGD Batched Sampling.

```
sampled_T = T.copy()
sampled_T.sample(samprate)
sOmega = getOmega(sampled_T)
R = sampled_T - ctf.TTTP(sOmega,[U,V,W])
ctf.MTTKRP(R,[U,V,W],0)
U+= -2* step* lmbda *samprate*U
```

The bulk of the computation within SGD is then comprised of the above sparse MTTKRP, which calculates a subgradient from $\mathcal{R}$ (the residual for the sampled entries). The `getOmega()` function works by reading the local nonzeros of the tensor, and writing them to a new sparse tensor with unit values. We also consider an implementation of SGD with the all-at-once Cyclops MTTKRP.

5.6.1. Quasi-Newton (Gauss-Newton) implementation

For implementing the quasi-Newton or Newton's algorithm, right hand sides in each iteration can be easily computed as these are negative of gradient with respect to each factor matrix. For solving the linear system in each iteration, CG iterations require matrix vector products with the implicit form of the Hessian. Each block of contraction required for the method as described in Section 3.3 can be implemented using TTTP and the MTTKRP kernel. The output of TTTP is fed into the MTTKRP kernel with the desired output index. The expression for $(1,2)$ Hessian contractions is as follows

Listing 8 GN implicit block (1,2) contraction.

```
A[0] += ctf.MTTKRP(ctf.TTTP(Omega,[U,Delta[1],W]),
        [None,V,W], 0)
```

Preconditioning can also be easily incorporated using the Solve Factor kernel used in Section 5.4.

6. Experimental evaluation

We provide performance results for a range of kernels and for tensor completion algorithms overall.[1] All benchmarks and application code are written purely in Python using Cyclops without any explicit distributed data management/communication. We study the scalability of redistribution routines within Cyclops for sparse and dense tensors by benchmarking tensor transposition and reshaping routines. We then consider performance of the new hypersparse contraction and TTTP kernels by benchmarking TTM, MTTKRP, TTTP, and Solve Factor. Finally, we provide a comparative study of the performance of all the algorithms introduced in Section 2 for tensor completion on a model low-rank dataset and on a realistic large tensor (Netflix dataset [7]) with two different loss functions.

6.1. Benchmarking configuration

All results are collected on the Stampede2 supercomputer at Texas Advanced Computing Center (TACC) via XSEDE. Stampede2 consists of 4200 Intel Knights Landing (KNL) compute nodes (each capable of a performance rate over 3 Teraflops/s) connected by an Intel Omni-Path (OPA) network with a fat-tree topology (achieving an injection bandwidth of 12.5 GB/sec). We use Cyclops v1.5.5 built with Intel ICC compiler v18.0.2 with MKL and ScaLAPACK, Intel MPI, HPTT v1.0.5, and -O1 level of optimization. We benchmark the MTTKRP in SPLATT v1.1.1 and use the 'sc16' branch to benchmark tensor completion [64], using distributed MPI variants of both. All experiments use 64 MPI processes per node, with 1 thread per process. For all benchmarks except tensor completion, we quantify noise by displaying estimated 95% confidence intervals. These are centered at the arithmetic mean and have a width of four standard deviations in the observed data (first/warm-up trial ignored).

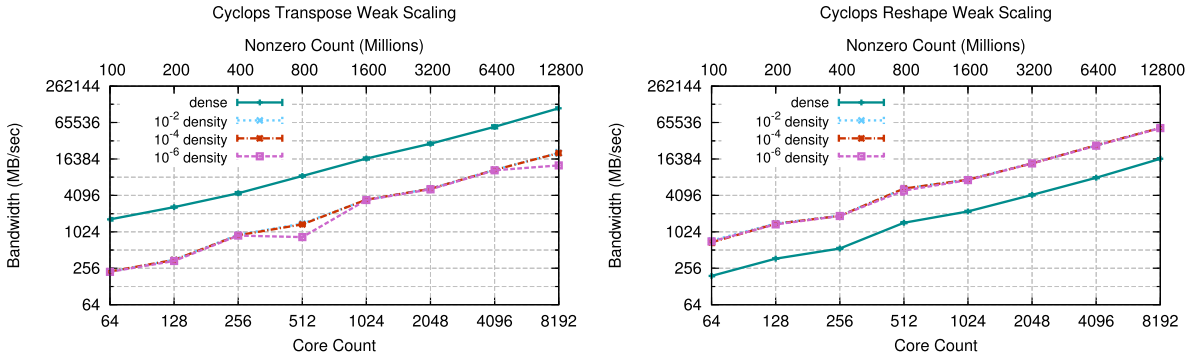
6.2. Redistribution performance

Fig. 4(a) and Fig. 4(b) consider the weak scalability of tensor transposition and reshaping. These are commonly used as multidimensional array operations in NumPy Python code, so their performance is important for a range of applications. Redistributions are substantially more costly in a distributed environment and are often the main bottleneck in Cyclops tensor contractions due to the necessity of communicating data between processes to a new processor grid mapping. The number of nonzero elements is kept fixed across variants, but increased in proportion to the number of nodes used. Overall, we observe good scalability in end-to-end bandwidth (computed as the number of bytes necessary to store the tensor divided by execution time) of the two operations. The reshape performance for dense tensors can be improved, as it converts to sparse format, leveraging preservation of global ordering. The performance is generally independent of tensor order or of the particular type of transpose/reshape.

6.3. Hypersparse representation performance

Fig. 5(a) compares variants of Cyclops tensor times matrix (TTM) kernels using 64 nodes of Stampede2 for various density of nonzeros (for a fixed nonzero count). The performance of each variant is plotted for problem sizes for which it does not run out of memory. We observe that the dense variant performs relatively well, but quickly runs out of memory. Using a sparse tensor representation and a dense output representation achieves the best

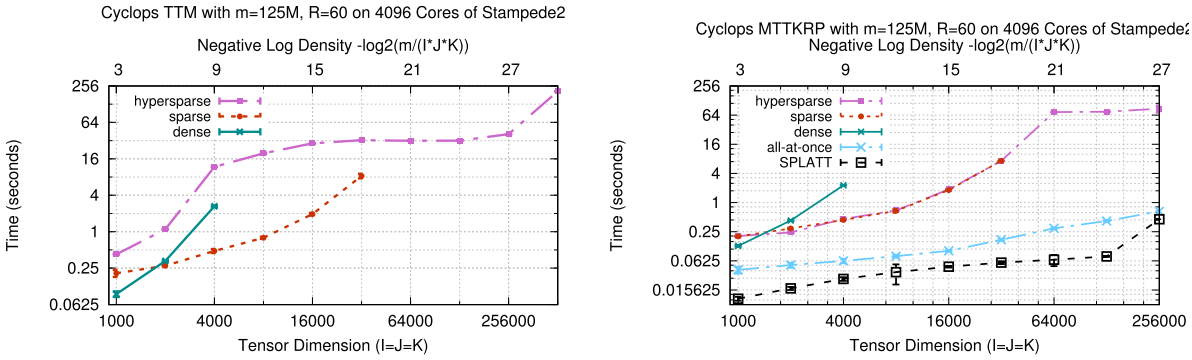
[1] The tensor completion codes are available via https://github.com/cyclops-community/Tensor_completion.



(a) Tensor Transposition with Cyclops

(b) Tensor Reshape with Cyclops

Fig. 4. Achieved bandwidth/throughout of transpose and reshape Python functions with Cyclops (16 bytes assumed for each nonzero in a sparse tensor and 8 bytes for each value in a dense tensor).



(a) Tensor-Times-Matrix (TTM) with Cyclops

(b) MTTKRP with Cyclops

Fig. 5. Execution time of TTM and MTTKRP for order 3 tensors, both averaged over three possible variants (choices of contracted and uncontracted modes, respectively).

performance, but as the number of nonzeros grows, the output becomes sparse and representing it in a dense format incurs an unmanageable memory footprint. Finally, the hypersparse variant, which leverages a sparse output tensor, incurs significant overhead with respect to using a dense output, but is able to scale to substantially more sparse tensors. Overall, we conclude that the hypersparse implementation achieves the desired memory scaling, but at a significant constant factor overhead, due to the need for more sparse format conversions, indirect accesses, and sparse reduction.

Fig. 5(b) demonstrates the performance of MTTKRP using Cyclops, comparing also to the highly-optimized SPLATT implementation [62,63] (by profiling the MTTKRP within its parallel CP decomposition). In the MTTKRP kernel, a third order tensor is contracted with matrices along two modes, e.g., $\sum_{i,k} t_{ijk} u_{ir} w_{kr}$. The given performance results are the average over the three choices of uncontracted modes. There are two choices for performing this operation via pairwise tensor contractions, either to first contract $\mathcal{T}$ and $\boldsymbol{U}$ or to first contract $\boldsymbol{U}$ and $\boldsymbol{W}$. The latter can be faster if $\mathcal{T}$ is relatively dense, but is slower if $\mathcal{T}$ is sufficiently sparse. When the intermediate output tensor is sufficiently sparse and contracting with $\mathcal{T}$ first is estimated to take less time, Cyclops automatically leverages the hypersparse representation. Use thereof permits scalability to much sparser tensors. However, we observe that all-at-once computation of MTTKRP is much faster than pairwise tensor contraction.

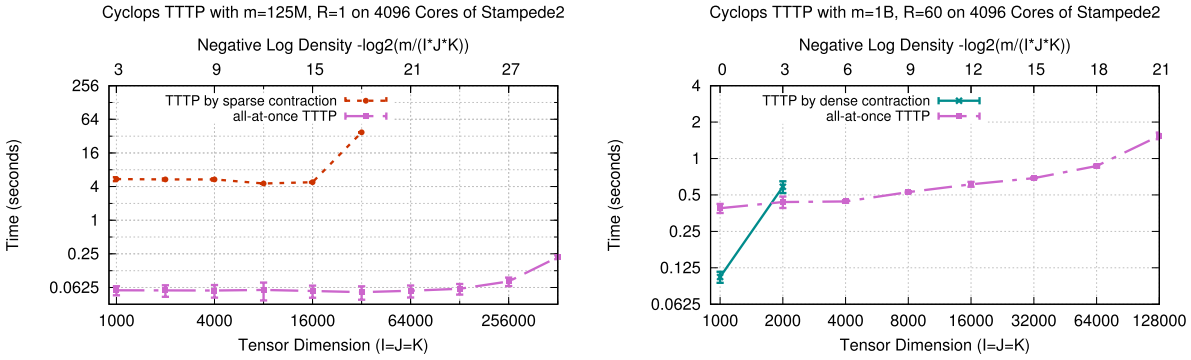
SPLATT outperforms the Cyclops all-at-once implementation as the latter requires redistribution of factor matrices and does not use the CSF format. However, generally Cyclops is within a factor of four or less in performance with respect to SPLATT. Further, the approach used in Cyclops permits easier combination with other tensor operations, since the input distribution of the factor matrices is not specialized for the kernel.

6.4. TTTP performance

Fig. 6(a) and Fig. 6(b) compare the performance of the new TTTP kernel to alternatives based on pairwise tensor contraction, including with the use of hypersparsity. However, even with hypersparsity, the intermediates which must be formed in any pairwise contraction tree increase the memory usage, whenever $R > 1$. We observe that the TTTP kernel is always significantly faster and can scale to extremely low density. By comparison, pairwise tensor contraction approaches are slower even when $R = 1$ and are less memory scalable. Overall, the benefit of performing TTTP all-at-once as opposed to via pairwise contractions is clearly evident.

6.5. Solve factor and alternating least squares performance

We compare our implementation of alternating minimization for least squares loss (ALS) using the Solve Factor and MTTKRP kernels introduced above to the state of the art implementation of ALS in SPLATT [64] in Fig. 7. The SPLATT approach forms the left and right hand sides in a single pass over the tensor nonzero entries, thereby reusing Hadamard products of the rows for each nonzero entry. In contrast, our implementation does two passes over the tensor nonzeros. In Fig. 7(a), we compare the performance of one ALS iteration on a tensor with fixed number of observed entries while increasing the dimensions of the tensor dataset. SPLATT outperforms our ALS implementation by a speed of about 2× for most



(a) TTTP with $R = 1$ (b) TTTP with $R = 60$

Fig. 6. Execution time of the described TTTP kernel (all-at-once TTTP) and implementations based on pairwise tensor contraction, with $R = 1$ and $R = 60$ tensor products.

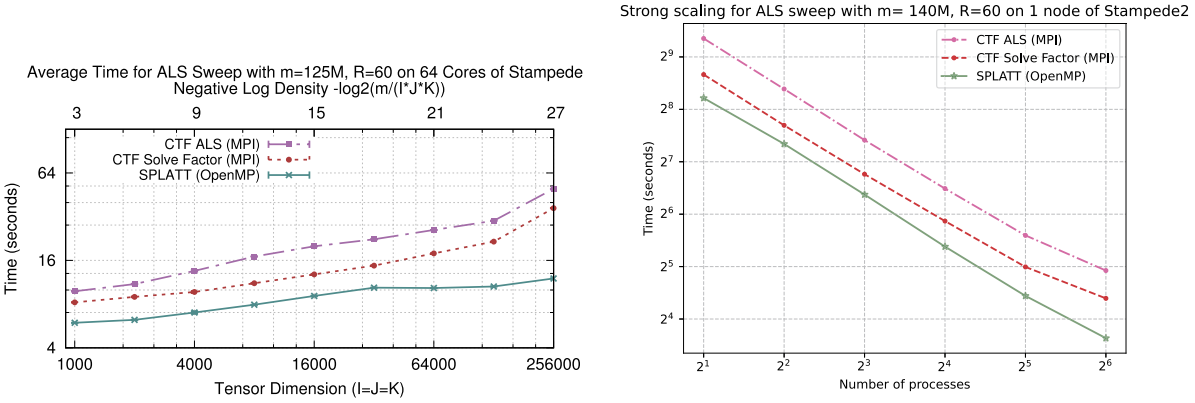


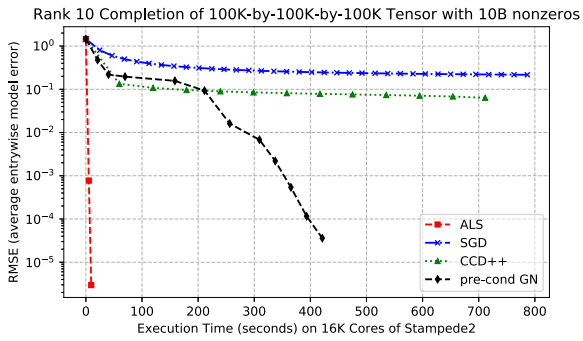
Fig. 7. Comparison of performance of ALS approaches for a single ALS sweep for the random processes using 64 MPI processes for CTF and 64 OPENMP threads with 1 MPI process on a single KNL node of Stampede2.

of the cases. The speed up becomes 4.1× for the largest dimension because of communication among the cores in the MPI implementation, which is not needed in SPLATT's threaded implementation. Moreover, we perform a redistribution of factor matrices for each kernel call, which also involves an overhead. In Fig. 7(b), we compare the strong scaling of our implementation and SPLATT on a single KNL node of Stampede2 for a synthetic equidimensional tensor with dimension 64,000 and approximately 140M nonzeros. We observe that both the implementations strong scale well and SPLATT ALS is approximately 2× faster than our ALS iteration for each configuration. While somewhat slower than SPLATT, our implementation has lower memory footprint. SPLATT is unable to perform completion with MPI parallelization on one node for larger dimensions due to the memory bottleneck of forming left hand sides described in Section 4.4. Our implementation alleviates this memory overhead by using batched computation of the rows of the required factor matrices. SPLATT stores multiple compressed sparse fibre (CSF) representations of the tensor, which eliminates the need for sorting tensor nonzeros on the fly. CSF is faster as compared to Cyclops COO-like format which requires extra computation to determine the indices for each nonzero. However, the Cyclops COO-like format is more memory efficient than SPLATT, as it uses only one rather than three copies of the input tensor. Further, the replicated CSF approach would entail additional overheads for generalized loss functions as the input to the kernel changes for other loss functions at each sub-iteration, necessitating construction of three copies of the data at each sub-iteration. Our Solve Factor kernel is only about 1.7× slower than SPLATT for most of the cases, suggesting the overheads of using general kernels is not too high.

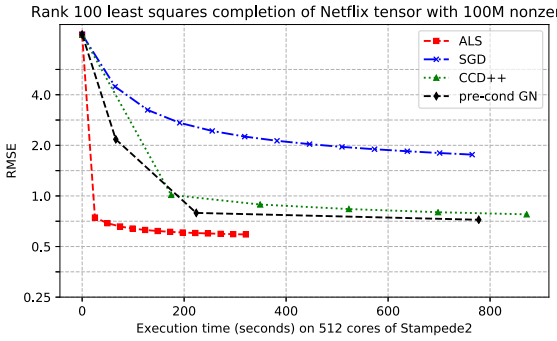
6.6. Tensor completion with least squares loss

Fig. 8(a) studies the performance of tensor completion algorithms with Cyclops on a model problem constructed from a sampled function as described in Karlsson et al. [35]. The sampled tensor has low CP rank (we pick $R = 10$) and a good CP decomposition is easily found by quadratic approximation. We observe that ALS requires only a few iterations to achieve full accuracy (RMSE proportional to the regularization used, $\lambda = 10^{-5}$). CCD++ is executed with a regularization of $\lambda = 10^{-5}$ and SGD is executed with sampling and learning rate of $5 \cdot 10^{-3}$ and regularization of 10^{-7}. The CCD++ and SGD approaches achieve comparable performance, requiring less time per iteration, but making progress at a slower rate overall (RMSE plotted after every 20 iterations). Preconditioned Gauss-Newton method also converges to full accuracy in a few iterations (regularization used $\lambda = 10^{-3}$) but is considerably slower than ALS execution time. Using 256 nodes of Stampede2, this experiment demonstrates the scalability of our Python-based tensor completion implementations, as they are executed on a problem containing 10 billion observed entries (nonzeros) with a density of 10^{-5}.

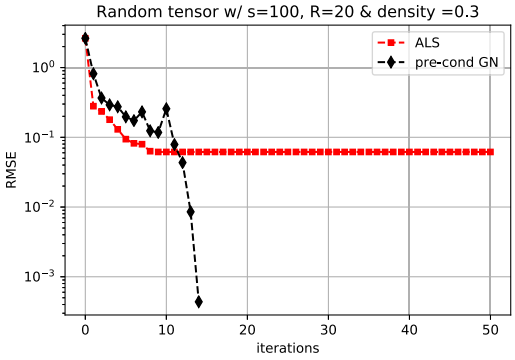
In Fig. 8(b), we consider performance for the Netflix movie rating dataset on 4 nodes of Stampede2 with a rank 100 CP representation. This tensor is $480,189 \times 17,770 \times 2,182$ and contains $m = 100,477,727$ nonzeros. While ALS achieves the lowest RMSE, the three methods that use second order information are relatively competitive for this tensor. ALS iterations take the least time followed by the CCD++ iterations, For CCD++, we traverse the tensor nonzeros $2R$ times for each CCD++ iteration as compared to 2 times for each ALS iteration. Both algorithms use a



(a) Tensor Completion for Function Tensor Model Problem



(b) Tensor completion for Netflix Dataset



(c) Tensor completion for tensor constructed from positive random matrices

Fig. 8. Performance results of tensor completion methods.

regularization parameter of $\lambda = 10^{-5}$. Gauss-Newton with implicit pre-conditioned CG uses a relative tolerance of $5 \cdot 10^{-3}$ and max iterations of at most 30 for CG and a regularization parameter $\lambda = 10^{-3}$. We use the Solve Factor kernel to implement block diagonal pre-conditioning which is essential for faster convergence and stability of CG iterations. The algorithm starts to take more time as CG iterations start to increase due to the fact that gradient norm decreases. Unlike the function tensor model problem, SGD requires fine-tuning of parameters, diverging when the learning rate is set to be too high. We show performance with a learning and sampling rate of $3 \cdot 10^{-3}$ with $\lambda = 10^{-5}$, which resulted in cheap iterations and steady but slow convergence (RMSE plotted after every 20 iterations). The progress made by the SGD steps can likely be improved by strategies that vary the learning and sample rate, a consideration which we leave for future work.

In Fig. 8(c), we consider performance for pre-conditioned Gauss-Newton method and ALS for a synthetic tensor. This tensor is constructed with random matrices with entries sampled uniformly from $[0, 1]$ with dimension $s = 100$ and CP rank $R = 20$ with 30% observed entries, i.e., the tensor has $3 \cdot 10^5$ observed entries and is relative dense. We observe that for this type of problem, pre-conditioned Gauss-Newton converges to the solution in a few iterations whereas ALS seems to make very little progress after 10 iterations. This corroborates the claim in the previous work [49] that ALS does not perform well for relatively dense tensors and methods like Gauss-Newton may be preferable when an exact solution exists.

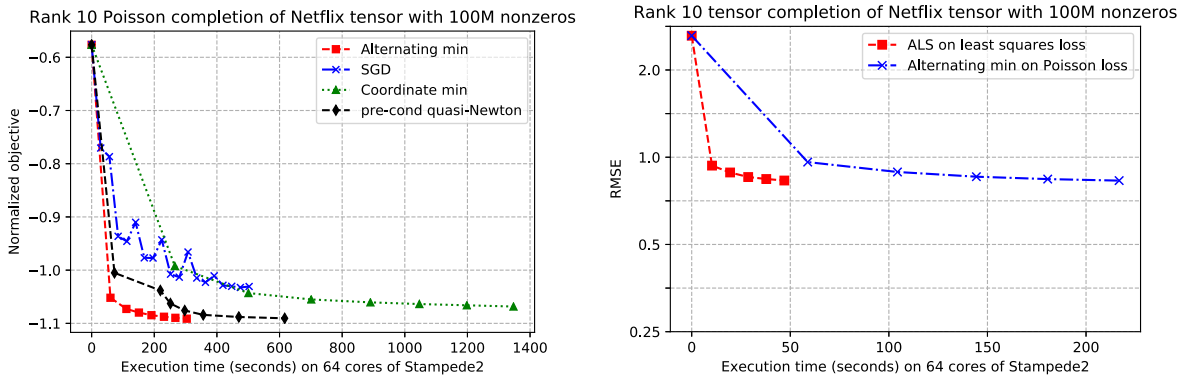
6.7. Tensor completion with Poisson loss

To demonstrate our algorithmic and software framework for generalized CP completion, we implement the above described algorithms for tensor completion with Poisson loss with the logarithm link function (log-link) for the Netflix tensor. Poisson loss for decomposing tensors with entries in the set of natural numbers has several qualitative advantages that have been explored in the previous literature [25,15]. We explore the quantitative performance and scalability of various algorithms in a distributed setting.

Poisson loss with log-link was introduced in [32] for tensors with entries in the set of natural numbers. The advantage of using log-link is that it relaxes the nonnegativity constraints required with the identity-link and hence, we can use our framework to implement all the algorithms without having to account for any constraints. The loss function minimized here is described by setting the elementwise function ϕ introduced in equation (3) to

$$\phi(t_{ijk}, \langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle) = \exp(\langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle) - t_{ijk} \langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle.$$

With the elementwise function defined as above, we use values from Table 1 to implement all the completion algorithms described in Section 3 with this loss for the Netflix tensor with rank $R = 10$. We plot the normalised loss, i.e., $\frac{1}{|\Omega|} \sum_{i,j,k} \phi(t_{ijk}, \langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle)$ versus time for each algorithm in Fig. 9(a). Each point in the Fig. 9(a) represents an iteration, except for SGD, for which each point is plotted after every 20 iterations. Both the alternating minimization and coordinate minimization inner iterations are performed until a relative step tolerance of 10^{-3} or a maximum count of 5 is reached. Pre-conditioned quasi-Newton has a relative tolerance of $5 \cdot 10^{-3}$ or a maximum iteration count of R for CG iterations for each system solve. Also, the regularization parameter plays a pivotal role for this objective function as algorithms diverge easily due to the exponentiation. Compared to least-squares loss, we employ higher values of regularization for all the algorithms to ensure that



(a) Comparison of performance of different Poisson tensor completion algorithms

(b) Comparison of performance of alternating minimization on different loss functions

Fig. 9. Tensor completion with Poisson and least squares loss function on Netflix tensor on 64 cores of Stampede2.

they do not diverge. We use a value of $\lambda = 1$ for coordinate minimization, and $\lambda = 0.1$ for alternating minimization, quasi-Newton, and SGD.

We observe that alternating minimization is the fastest algorithm to reach the least value of the objective followed by preconditioned quasi-Newton method and then coordinate minimization. While implementation of all these algorithms can be fine-tuned to further run faster for the particular loss functions, we observe that SGD implementation is outperformed by other algorithms indicating that the benefit of using second order information. Our formulation of the quasi-Newton with implicit preconditioned CG method not only makes the implementation feasible via tensor algebra kernels, but is also competitive with other algorithms in practical scenarios.

For a Poisson loss objective, we plot the Frobenius norm of the subtraction of input tensor and exponentiated reconstructed tensor at each iteration and compare it with the ALS iterations. We observe that these values are equal up to 2 digits suggesting that the Poisson loss also minimizes the least squares loss, however, vice versa is not true as there may be negative values making the Poisson objective infeasible to calculate. Note that the Poisson loss completion comes at the cost of performing inner iterations, which results in longer running time, as observed in Fig. 9(b).

7. Related work

We review related work on parallel tensor abstractions and on previous parallel implementations of tensor completion. We also review work on sparse tensor kernels for tensor decompositions.

7.1. Parallel tensor completion

The tensor completion algorithms presented in this paper have commonly-used analogous in matrix completion (ALS [33], SGD [39], CCD [81]). These approaches, especially SGD, have been optimized extensively for the matrix case, which may be viewed as a simple two-layer neural network. In shared memory, SGD is widely used, as it can be made efficient by asynchronous execution [59]. ALS, CCD, and SGD for matrix completion have all been target of efficient distributed-memory implementations [72,81,26,21].

Tensor completion via the CP tensor representation [21] has been a target of recent distributed-memory implementation efforts. Karlsson et al. [35] implement ALS and CCD by replicating the factor matrices on each process and distributing observed entries. While efficient, this approach is not scalable to very large factor matrices. Smith et al. [64] improve upon this method by distributing both the factor matrix and tensor in coherent formats, similar to our parallel method for TTTP when it is done with a single parallel step. For generalized tensor decomposition, a distributed memory implementation is available [18,45] that uses stochastic gradient descent algorithm and uses permutation arrays in COO format to represent sparse tensors. Our work is the first to implement distributed tensor completion using high-level tensor operations for general tensor contractions. We reproduce previous work [35,64] in the observation that ALS is generally most efficient for distributed tensor completion.

7.2. Sparse tensor kernels

Parallel sparse matrix multiplication algorithms comprise an active area of research [69,42,3,4,12,24,55]. Multiplication of hypersparse matrices has seen considerably less study [11]. An optimized doubly compressed CSR/CSC layout similar to the CSF matrix layout used in this paper is the standard sequential approach to hypersparse matrix–matrix products [11].

Effective sparse tensor layouts have been designed for TTM and MTTKRP operations in shared memory and distributed memory. The compressed sparse fiber (CSF) layout serves as an extension of hypersparse matrix representations and achieves efficient storage and TTM operations [63,62]. The hierarchical coordinate (HiCOO) layout is designed to further improve efficiency for TTM and MTTKRP [47]. The Adaptive Linearized Tensor Order (ALTO) [28] is another proposed sparse tensor format for shared memory architecture which is an improvement over the HiCOO and CSF. ALTO uses an adaptive recursive partitioning of the high dimensional space of the sparse tensor to map the nonzeros onto a compact line so that the neighbouring nonzeros are close to each other. The tensor algebra compiler (TACO) supports hierarchical layouts with compressed or uncompressed modes [40] as well as other optimized sparse formats [16]. These layouts can be interchanged and may improve upon the CSF matrix layout used in our work. However, our design is the first to enable arbitrary tensor contractions to be reduced to a storage-efficient layout, and to support distributed-memory tensor operations with hypersparse representations.

TTM and MTTKRP are standard benchmark tensor kernels [43,48]. MTTKRP has been the target of optimization for distributed-memory architectures with both MPI [63,37] and MapReduce [56,10]. While TTM is a special case of a tensor contraction, MTTKRP involves contraction of multiple tensors and consequently presents

potential for further performance optimization over pairwise contraction by all-at-once contraction [27]. The TTTP operation introduced in this paper differs significantly from MTTKRP and can be specially optimized via all-at-once contraction.

7.3. Tensor frameworks

Tensors and multidimensional arrays are a prevalent programming abstraction that encapsulates data parallelism. Many tensor libraries are designed for methods in quantum chemistry. The Tensor Contraction Engine (TCE) [30] provides factorization of multi-tensor expressions into pairwise contractions. TCE generates parallel tensor contraction code based on a partitioned global address-space (PGAS) [80] language, Global Arrays [52]. Global Arrays and other PGAS languages such as UPC [19] provide multidimensional array abstractions that enable tensor programming, but generally do not support high-level tensor algebra operations. The Libtensor library [20] provides efficient shared-memory tensor contractions, targeted at quantum chemistry applications. Libtensor and other libraries [51] support block-sparse tensors. The TiledArray [57,13] library provides distributed-memory support for block-sparse tensor contractions. Outside of Cyclops, to the best of our knowledge, tensor contractions with arbitrary elementwise sparsity are only supported for single-node execution [36]. Currently, the Python interface of Cyclops does not support arbitrary ring operations, however there is recent work [29] for sparse array programming on single-node machines. All the efforts above theimmediate one leverage an Einstein notation syntax for contractions and aim at efficient execution of tensor contractions arising in quantum chemistry.

The Tensor Algebra Compiler (TACO) [40] provides support for sequential sparse tensor contractions and more general multi-tensor expressions. In recent work, TACO has been improved to factorize longer tensor algebra expressions and their subcomponents into subsequences [41], the former being a user-guided version of the automated factorization in Cyclops. Tensor libraries have also been designed for machine learning workloads, e.g., TensorFlow by Google [1] and Tensor Comprehensions by Facebook [79]. Both focus on task-level parallelism and GPU acceleration as opposed to distributed-memory data parallelism.

8. Conclusion and future work

We present new advances in parallel sparse tensor computations infrastructure and methodology, driven by its application to tensor completion. Specifically, we propose a new tensor algebra routine, TTTP, which consists of tensor contractions that may be significantly accelerated by an all-at-once contraction algorithm. Further, we provide the first distributed general sparse tensor contraction infrastructure that can leverage hypersparse matrix representations, achieving scalability to massively sparse tensors.

For tensor completion, we propose a novel Newton-method-based algorithmic framework for generalized tensor completion. In this framework, we introduce alternating minimization, coordinate minimization and quasi-Newton algorithms which encompass the ALS, CCD++ and Gauss-Newton algorithm for least squares loss and generalize easily for other objective functions. Our results demonstrate that these algorithms are more accurate than the SGD algorithm for generalized completion. By providing a high-level Python interface to the tensor algebra operations, we are able to develop very concise, but massively-parallel implementations of these algorithms for generalized tensor completion via CP decomposition. Moreover, we show that our distributed memory implementation of alternating minimization for least squares loss which uses general sparse tensor kernels is within a factor of four of the state of the art distributed implementation of ALS. Our experimental results demonstrate that hypersparsity, all-at-once kernels for MTTKRP, and the new TTTP algorithm enable generalized tensor completion algorithms to be executed on much larger and sparser tensors than possible with previously available libraries.

For the generalized objective functions, some of the link functions use nonnegativity constraints [32], which are not incorporated in our current framework. While all the link functions can be modified to remove these constraints, the interpretation of the factors might change. These constraints can be incorporated with use of projected Newton's algorithm [8] or using a barrier formulation. All the kernels introduced in Section 4 can be optimized further by using specialised tensor formats like CSF coupled with an optimal threaded implementation for best performance. However, it is non-trivial to construct these formats optimally for each sub-iteration for a generalized loss functions.

CRediT authorship contribution statement

Navjot Singh: Conceptualization, Methodology, Data curation, Software, Writing, Validation, Visualization, Investigation. **Zecheng Zhang:** Software, Data Curation, Investigation, Writing. **Xiaoxiao Wu:** Software, Resources, Data curation, Writing. **Naijing Zhang:** Resources, Data curation, Writing. **Siyuan Zhang:** Resources, Data curation, Writing. **Edgar Solomonik:** Conceptualization, Methodology, Software, Writing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Via XSEDE, the authors made use of the TACC Stampede2 supercomputer. The research was supported by the US NSF OAC via award No. 1942995.

Appendix A. Generalized CP decomposition

All the algorithms for generalized CP completion introduced in Section 2 are based on elementwise derivatives of the generalized objective function [32] with respect to each variable in the factor matrix. In this section, we use tensor calculus to derive the necessary expressions for an N^{th} order input tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. We assume that we have an index set $\Omega \subset \{1, \ldots, I_1\} \times \cdots \times \{1, \ldots I_N\}$, which represents the set of observed entries of the input tensor. If Ω consists of all the elements then the objective function would correspond to a decomposition problem. The objective function is

$$f(\mathbf{A}^{(1)} \ldots \mathbf{A}^{(N)}) = \sum_{i_1,\ldots,i_N \in \Omega} \phi(x_{i_1 \ldots i_N}, m_{i_1 \ldots i_N}),$$

$$\text{where} \quad m_{i_1 \ldots i_N} = \sum_{r=1}^{R} \prod_{n=1}^{N} a^{(n)}_{i_n r}.$$

The elementwise expression for the gradient of $f(\mathbf{A}^{(1)} \ldots \mathbf{A}^{(N)})$ with respect to d^{th} factor matrix is

$$\frac{\partial f}{\partial a^{(d)}_{kr}} = \sum_{i_1,\ldots,i_N \in \Omega} \frac{\partial \phi(x_{i_1 \ldots i_N}, m_{i_1 \ldots i_N})}{\partial m_{i_1 \ldots i_N}} \delta_{i_d k} \prod_{n=1, n \neq d}^{N} a^{(n)}_{i_n r}.$$

Computing the gradient corresponds to an MTTKRP operation with the derivative tensor $\phi'_{i_1 \ldots i_N}$ (5). We can differentiate the above expression for gradient further to arrive at an elementwise form of the Hessian matrix. This form is useful for writing algorithms that use second order information as these use a part of the Hessian matrix and/or use the implicit form of the Hessian. The derivative of the gradient with respect to p^{th} factor matrix can be calculated by applying chain rule inductively

$$h^{(d,p)}_{krlz} = \frac{\partial f^2}{\partial a^{(d)}_{kr} \partial a^{(p)}_{lz}}$$
$$= \sum_{i_1,\ldots,i_N \in \Omega} \phi''_{i_1 \ldots i_N} \delta_{i_p l} \Big(\prod_{n=1, n \neq p}^{N} a^{(n)}_{i_n z} \Big) \delta_{i_d k} \Big(\prod_{n=1, n \neq d}^{N} a^{(n)}_{i_n r} \Big)$$
$$+ (1 - \delta_{dp}) \sum_{i_1,\ldots,i_N \in \Omega} \phi'_{i_1 \ldots i_N} \delta_{i_d k} \Big(\prod_{n=1, n \neq d, p}^{N} a^{(n)}_{i_n r} \Big) \delta_{i_p l} \delta_{rz},$$

where δ_{ij} is the Kronecker-Delta function. Newton or quasi-Newton method requires solution to linear systems involving the Hessian at each iteration. Conjugate gradient method can be used to solve these systems of equations by making use of the implicit form of the Hessian. Given current factor matrix updates $\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(N)}$, the matrix-vector product with the Hessian can be computed by the following tensor contractions,

$$w^{(d)(new)}_{kr} = \sum_{p} \sum_{l,z} h^{(d,p)}_{krlz} w^{(p)}_{lz}, \quad d \in \{1, \ldots, N\}, p \in \{1, \ldots, N\},$$

where $\boldsymbol{W}^{(d)(\text{new})}$ is the updated matrix corresponding to the d^{th} factor matrix. These contractions reduce to simpler contractions as mentioned in Section 3.3. The above form of Hessian can be used to derive all the methods described in Section 3.

Alternating minimization described in Section 3.1 is equivalent to a block non-linear Gauss-Siedel method [22] to minimize the above objective function. Alternating minimization subiteration uses a diagonal block of the above described Hessian for optimizing a factor matrix given by

$$h^{(d,d)}_{krlz} = \sum_{i_1,\ldots,i_N \in \Omega} \Big(\prod_{n=1, n \neq d}^{N} a^{(n)}_{i_n z} \Big) \delta_{i_d k} \phi''_{i_1 \ldots i_N} \delta_{i_d l} \Big(\prod_{n=1, n \neq d}^{N} a^{(n)}_{i_n r} \Big).$$

Coordinate minimization subiteration described in Section 3.2 is equivalent to a non-linear Gauss-Seidel method, as in each subiteration, the method minimizes only one variable (in parallel) at a time. It uses the diagonal of the diagonal block of the above described Hessian given by

$$h^{(d,d)}_{kr} = \sum_{i_1,\ldots,i_N \in \Omega} \Big(\prod_{n=1, n \neq d}^{N} a^{(n)}_{i_n r} \Big)^2 \delta_{i_d k} \phi''_{i_1 \ldots i_N}.$$

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, Tensorflow: a system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA, 2016, pp. 265–283.

[2] E. Acar, D.M. Dunlavy, T.G. Kolda, A scalable optimization approach for fitting canonical tensor decompositions, J. Chemom. 25 (2) (2011) 67–86.

[3] G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, S. Toledo, Communication optimal parallel multiplication of sparse random matrices, in: Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, ACM, New York, NY, USA, 2013, pp. 222–231.

[4] G. Ballard, A. Druinsky, N. Knight, O. Schwartz, Brief announcement: hypergraph partitioning for parallel sparse matrix-matrix multiplication, in: Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2015, pp. 86–88.

[5] G. Ballard, N. Knight, 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.

[6] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, K. Smith, Cython: the best of both worlds, Comput. Sci. Eng. 13 (2) (2011) 31–39.

[7] J. Bennett, S. Lanning, et al., The Netflix prize, in: Proceedings of KDD Cup and Workshop, New York, NY, USA, vol. 2007, 2007, p. 35.

[8] D.P. Bertsekas, Projected Newton methods for optimization problems with simple constraints, SIAM J. Control Optim. 20 (2) (1982) 221–246.

[9] L.S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[10] Z. Blanco, B. Liu, M.M. Dehnavi, CSTF: large-scale sparse tensor factorizations on distributed platforms, in: Proceedings of the 47th International Conference on Parallel Processing, ACM, 2018, p. 21.

[11] A. Buluc, J.R. Gilbert, On the representation and multiplication of hypersparse matrices, in: 2008 IEEE International Symposium on Parallel and Distributed Processing, IEEE, 2008, pp. 1–11.

[12] A. Buluç, J.R. Gilbert, Parallel sparse matrix-matrix multiplication and indexing: implementation and experiments, SIAM J. Sci. Comput. 34 (4) (2012) C170–C191.

[13] J.A. Calvin, C.A. Lewis, E.F. Valeev, Scalable task-based algorithm for multiplication of block-rank-sparse matrices, in: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, ACM, 2015, p. 4.

[14] J. Canny, H. Zhao, Big data analytics with small footprint: squaring the cloud, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2013, pp. 95–103.

[15] E.C. Chi, T.G. Kolda, On tensors, sparsity, and nonnegative factorizations, SIAM J. Matrix Anal. Appl. 33 (4) (2012) 1272–1299.

[16] S. Chou, F. Kjolstad, S. Amarasinghe, Format abstraction for sparse tensor algebra compilers, Proceedings of the ACM on Programming Languages 2 (October 2018).

[17] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, 8.2 counting sort, in: Introduction to Algorithms, Mit Press and Mcgraw-Hill, 2001, pp. 636–640, ISBN 262032937.

[18] K.D. Devine, G. Ballard, Gentenmpi: Distributed memory sparse tensor decomposition, https://doi.org/10.2172/1656940.

[19] T. El-Ghazawi, W. Carlson, T. Sterling, K. Yelick, UPC: Distributed Shared Memory Programming, vol. 40, John Wiley & Sons, 2005.

[20] E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, A.I. Krylov, New implementation of high-level correlated methods using a general block-tensor library for high-performance electronic structure calculations, J. Comput. Chem. (2013).

[21] R. Gemulla, E. Nijkamp, P.J. Haas, Y. Sismanis, Large-scale matrix factorization with distributed stochastic gradient descent, in: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2011, pp. 69–77.

[22] L. Grippo, M. Sciandrone, On the convergence of the block nonlinear Gauss–Seidel method under convex constraints, Oper. Res. Lett. 26 (3) (2000) 127–136.

[23] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, MA, USA, 1994.

[24] F.G. Gustavson, Two fast algorithms for sparse matrices: multiplication and permuted transposition, ACM Trans. Math. Softw. 4 (3) (1978) 250–269.

[25] S. Hansen, T. Plantenga, T.G. Kolda, Newton-based optimization for Kullback–Leibler nonnegative tensor factorizations, Optim. Methods Softw. 30 (5) (2015) 1002–1029.

[26] T. Hastie, R. Mazumder, J.D. Lee, R. Zadeh, Matrix completion and low-rank SVD via fast alternating least squares, J. Mach. Learn. Res. 16 (1) (2015) 3367–3402.

[27] K. Hayashi, G. Ballard, Y. Jiang, M.J. Tobia, Shared-Memory Parallelization of MTTKRP for Dense Tensors, ACM SIGPLAN Notices, vol. 53, ACM, 2018, pp. 393–394.

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

[29] R. Henry, O. Hsu, R. Yadav, S. Chou, K. Olukotun, S. Amarasinghe, F. Kjolstad, Compilation of sparse array programming models, Proceedings of the ACM on Programming Languages 5 (OOPSLA) (2021) 1–29.

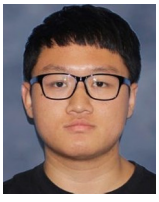
[30] S. Hirata, Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories, J. Phys. Chem. A 107 (46) (2003) 9887–9897.

[31] F.L. Hitchcock, The expression of a tensor or a polyadic as a sum of products, Stud. Appl. Math. 6 (1–4) (1927) 164–189.

[32] D. Hong, T.G. Kolda, J.A. Duersch, Generalized canonical polyadic tensor decomposition, SIAM Rev. 62 (1) (2020) 133–163.

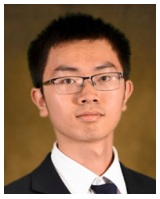
[33] P. Jain, P. Netrapalli, S. Sanghavi, Low-rank matrix completion using alternating minimization, in: Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing, ACM, 2013, pp. 665–674.
[34] E. Jones, T. Oliphant, P. Peterson, SciPy: Open source scientific tools for Python, 2014.
[35] L. Karlsson, D. Kressner, A. Uschmajew, Parallel algorithms for tensor completion in the CP format, Parallel Comput. 57 (2016) 222–234.
[36] D. Kats, F.R. Manby, Sparse tensor framework for implementation of general local correlation methods, J. Chem. Phys. 138 (14) (2013).
[37] O. Kaya, 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.
[38] O. Kaya, B. Uçar, Parallel CANDECOMP/PARAFAC decomposition of sparse tensors using dimension trees, SIAM J. Sci. Comput. 40 (1) (2018) C99–C130.
[39] R.H. Keshavan, A. Montanari, S. Oh, Matrix completion from noisy entries, J. Mach. Learn. Res. 11 (Jul 2010) 2057–2078.
[40] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, S. Amarasinghe, The tensor algebra compiler, Proceedings of the ACM on Programming Languages 1 (OOPSLA) (2017) 77.
[41] F. Kjolstad, P. Ahrens, S. Kamil, S. Amarasinghe, Tensor algebra compilation with workspaces, International Symposium on Code Generation and Optimization, February 2019.
[42] P. Koanantakool, A. Azad, A. Buluç, D. Morozov, S.-Y. Oh, L. Oliker, K. Yelick, Communication-avoiding parallel sparse-dense matrix-matrix multiplication, in: Parallel and Distributed Processing Symposium, 2016 IEEE International, IEEE, 2016, pp. 842–853.
[43] T. Kolda, B. Bader, Tensor decompositions and applications, SIAM Rev. 51 (3) (2009) 455–500.
[44] C.L. Lawson, R.J. Hanson, D.R. Kincaid, F.T. Krogh, Basic linear algebra subprograms for fortran usage, ACM Trans. Math. Softw. 5 (3) (1979) 308–323.
[45] C. Lewis, E.T. Phipps, T.G. Kolda, Distributed generalized canonical polyadic decomposition, Tech. Rep., Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), 2021.
[46] J. Li, J. Choi, I. Perros, J. Sun, 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.
[47] J. Li, J. Sun, R. Vuduc, HiCOO: hierarchical storage of sparse tensors, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2018, pp. 238–252.
[48] J. Li, Y. Ma, X. Wu, A. Li, K. Barker, PASTA: a parallel sparse tensor algorithm benchmark suite, preprint, arXiv:1902.03317, 2019.
[49] A. Liu, A. Moitra, Tensor completion made practical, preprint, arXiv:2006.03134, 2020.
[50] J. Liu, P. Musialski, P. Wonka, J. Ye, Tensor completion for estimating missing values in visual data, IEEE Trans. Pattern Anal. Mach. Intell. 35 (1) (2012) 208–220.
[51] E. Mutlu, K. Kowalski, S. Krishnamoorthy, Toward generalized tensor algebra for ab initio quantum chemistry methods, in: Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ACM, 2019, pp. 46–56.
[52] J. Nieplocha, R.J. Harrison, R.J. Littlefield, Global arrays: a nonuniform memory access programming model for high-performance computers, J. Supercomput. 10 (1996) 169–189.
[53] I. Nisa, A. Sukumaran-Rajam, S.E. Kurt, C. Hong, P. Sadayappan, Sampled dense matrix multiplication for high-performance machine learning, in: 2018 IEEE 25th International Conference on High Performance Computing (HiPC), IEEE, 2018, pp. 32–41.
[54] P. Paatero, A weighted non-negative least squares algorithm for three-way 'parafac' factor analysis, Chemom. Intell. Lab. Syst. 38 (2) (1997) 223–242.
[55] R. Pagh, M. Stöckel, The input/output complexity of sparse matrix multiplication, in: A.S. Schulz, D. Wagner (Eds.), Algorithms - ESA 2014, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 750–761.
[56] N. Park, B. Jeon, J. Lee, U. Kang, Bigtensor: mining billion-scale tensor made easy, in: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, ACM, 2016, pp. 2457–2460.
[57] C. Peng, J.A. Calvin, F. Pavosevic, J. Zhang, E.F. Valeev, Massively parallel implementation of explicitly correlated coupled-cluster singles and doubles using TiledArray framework, J. Phys. Chem. A 120 (51) (2016) 10231–10244.
[58] A.-H. Phan, P. Tichavskỳ, A. Cichocki, Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations, IEEE Trans. Signal Process. 61 (19) (2013) 4834–4846.
[59] B. Recht, C. Re, S. Wright, F. Niu, Hogwild: a lock-free approach to parallelizing stochastic gradient descent, in: Advances in Neural Information Processing Systems, 2011, pp. 693–701.
[60] N. Singh, L. Ma, H. Yang, E. Solomonik, Comparison of accuracy and scalability of Gauss-Newton and alternating least squares for cp decomposition, preprint, arXiv:1910.12331, 2019.
[61] D.B. Skillicorn, J. Hill, W.F. McColl, Questions and answers about BSP, Sci. Program. 6 (3) (1997) 249–274.
[62] S. Smith, G. Karypis, Tensor-matrix products with a compressed sparse tensor, in: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, ACM, 2015, p. 5.
[63] S. Smith, N. Ravindran, N.D. Sidiropoulos, G. Karypis, SPLATT: efficient and parallel sparse tensor-matrix multiplication, in: 2015 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2015, pp. 61–70.
[64] S. Smith, J. Park, G. Karypis, An exploration of optimization algorithms for high performance tensor completion, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 31:1–31:13.
[65] S. Smith, J.W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, G. Karypis, FROSTT: the formidable repository of open sparse tensors and tools, 2017.
[66] E. Solomonik, T. Hoefler, Sparse tensor algebra as a parallel programming model, preprint, arXiv:1512.00066, 2015.
[67] E. Solomonik, D. Matthews, J. Hammond, J. Demmel, Cyclops tensor framework: reducing communication and eliminating load imbalance in massively parallel contractions, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IEEE, 2013, pp. 813–824.
[68] E. Solomonik, D. Matthews, J.R. Hammond, J.F. Stanton, J. Demmel, A massively parallel tensor contraction framework for coupled-cluster computations, J. Parallel Distrib. Comput. 74 (12) (2014) 3176–3190.
[69] E. Solomonik, M. Besta, F. Vella, T. Hoefler, Scaling betweenness centrality using communication-efficient sparse matrix multiplication, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, New York, NY, USA, 2017, pp. 47:1–47:14.
[70] L. Sorber, M. Van Barel, L. De Lathauwer, Optimization-based algorithms for tensor decompositions: canonical polyadic decomposition, decomposition in rank-(l_r,l_r, 1) terms, and a new generalization, SIAM J. Optim. 23 (2) (2013) 695–720.
[71] P. Springer, T. Su, P. Bientinesi, HPTT: a high-performance tensor transposition C++ library, in: Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ACM, 2017, pp. 56–62.
[72] C. Teflioudi, F. Makari, R. Gemulla, Distributed matrix completion, in: 2012 Ieee 12th International Conference on Data Mining, IEEE, 2012, pp. 655–664.
[73] K. Teranishi, D.M. Dunlavy, J.M. Myers, R.F. Barrett, Sparten: leveraging Kokkos for on-node parallelism in a second-order method for fitting canonical polyadic tensor models to Poisson data, in: 2020 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2020, pp. 1–7.
[74] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in mpich, Int. J. High Perform. Comput. Appl. 19 (1) (2005) 49–66.
[75] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[76] S. Van Der Walt, S.C. Colbert, G. Varoquaux, The NumPy array: a structure for efficient numerical computation, Comput. Sci. Eng. 13 (2) (2011) 22.
[77] M. Vandecappelle, N. Vervliet, L.D. Lathauwer, A second-order method for fitting the canonical polyadic decomposition with non-least-squares cost, IEEE Trans. Signal Process. 68 (2020) 4454–4465.
[78] N. Vannieuwenhoven, K. Meerbergen, R. Vandebril, Computing the gradient in optimization algorithms for the CP decomposition in constant memory through tensor blocking, SIAM J. Sci. Comput. 37 (3) (2015) C415–C438.
[79] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W.S. Moses, S. Verdoolaege, A. Adams, A. Cohen, Tensor comprehensions: framework-agnostic high-performance machine learning abstractions, preprint, arXiv:1802.04730, 2018.
[80] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S.L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, et al., Productivity and performance using partitioned global address space languages, in: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, ACM, 2007, pp. 24–32.
[81] H.-F. Yu, C.-J. Hsieh, S. Si, I. Dhillon, Scalable coordinate descent approaches to parallel matrix factorization for recommender systems, in: 2012 IEEE 12th International Conference on Data Mining, IEEE, 2012, pp. 765–774.

Navjot Singh received his M.S. in Applied Mathematics in 2020 from University of Illinois at Urbana-Champaign and is currently a Ph.D. student in the Computer Science department at University of Illinois at Urbana-Champaign. His current research interests include numerical linear algebra, high performance computing, and tensor decompositions.



Zecheng Zhang is a founding engineer at Kumo.ai Inc. He received his M.S. in Computer Science at Stanford University in 2021 and B.S. in Computer Science at University of Illinois at Urbana-Champaign in 2019. He has research interests in machine learning with graphs and data mining.

Xiaoxiao Wu received her Bachelors degree in Mathematics and Computer Science from University of Illinois at Urbana-Champaign in 2019. She received her M.S. in Computational Finance from Carnegie Mellon University in 2021.



Naijing Zhang received his B.S. in Computer Science from University of Illinois at Urbana-Champaign in 2019. He received his M.Eng. in Computer Science from University of California, Berkley in 2020.

Siyuan Zhang received his B.S. in Computer Science from University of Illinois at Urbana-Champaign in 2019. He received his M.S. in Computer Science from University of Illinois at Urbana-Champaign in 2021.

Edgar Solomonik is an Assistant Professor in the Computer Science Department at University of Illinois at Urbana-Champaign. He received his Ph.D. from University of California, Berkeley in 2014. His research interests include numerical linear algebra, parallel algorithms, tensor networks, tensor decompositions, high performance computing. He received the Alston S. Householder Prize XX for the best dissertation in numerical linear algebra, SIAM Activity Group on Supercomputing Early Career Prize, and the NSF CAREER Award.