Concurrent Alternating Least Squares for multiple simultaneous Canonical Polyadic Decompositions

CHRISTOS PSARRAS, RWTH Aachen University, Germany
LARS KARLSSON, Umeå Universitet, Sweden
PAOLO BIENTINESI, Umeå Universitet, Sweden

Tensor decompositions, such as CANDECOMP/PARAFAC (CP), are widely used in a variety of applications, such as chemometrics, signal processing, and machine learning. A broadly used method for computing such decompositions relies on the Alternating Least Squares (ALS) algorithm. When the number of components is small, regardless of its implementation, ALS exhibits low arithmetic intensity, which severely hinders its performance and makes GPU offloading ineffective. We observe that, in practice, experts often have to compute multiple decompositions of the same tensor, each with a small number of components (typically fewer than 20), to ultimately find the best ones to use for the application at hand. In this paper, we illustrate how multiple decompositions of the same tensor can be fused together at the algorithmic level to increase the arithmetic intensity. Therefore, it becomes possible to make efficient use of GPUs for further speedups; at the same time the technique is compatible with many enhancements typically used in ALS, such as line search, extrapolation, and non-negativity constraints. We introduce the Concurrent ALS algorithm and library, which offers an interface to Matlab, and a mechanism to effectively deal with the issue that decompositions complete at different times. Experimental results on artificial and real datasets demonstrate a shorter time to completion due to increased arithmetic intensity.

CCS Concepts: • **Mathematics of computing → Mathematical software performance**; • **Software and its engineering → Software performance**.

Additional Key Words and Phrases: Tensor, decomposition, high-performance

1 INTRODUCTION

The Canonical Polyadic Decomposition (CPD or CP), also known as PARAllel FACtor analysis (PARAFAC), is a tensor decomposition or "tensor model" that has found applications in several domains, including chemometrics [2], signal processing and machine learning [27]. A target tensor is (often approximately) decomposed into a sum of rank-1 tensors, commonly referred to as "components". A CP decomposition can be computed by, for example, the Alternating Least Squares (CP-ALS) algorithm [12, 17]. This is an iterative algorithm that searches for a local minimum to the sum of squared residuals starting from some given starting point with a given number of components. Since only a local minimum is found, the result of the decomposition is highly susceptible to both the chosen number of components and the starting point. For this reason, application experts often resort to performing tens or hundreds of independent decompositions of a tensor, varying the number of components and/or the starting point, to determine the quality of each generated solution. Thus far, research has focused on improving methods that compute a single decomposition. In this paper, we show that it is possible to combine several decompositions at an algorithmic level to make more efficient use of the hardware. Even though the proposed technique does not decrease the time to complete a single decomposition, the total time to complete the whole set of decompositions can nevertheless be greatly reduced.

For dense tensors, the computational cost of CP-ALS is typically dominated by the so-called MTTKRP operation [18] (see also Section 3 ahead). The processor is a bottleneck for MTTKRP only when the number of components is large. Otherwise, the performance will be limited by the transfer of data back and forth between main memory and processor; we say that MTTKRP is "memory-bound" when the components are few, since the *arithmetic intensity* [36] (the number of floating point operations divided by the number of memory accesses) is proportional to the number

of components. Low arithmetic intensity leads to poor utilization of the CPU and also reduces the benefit from multi-threaded execution and GPU offloading. Since the issue is inherent to MTTKRP (and by extension to CP-ALS and many of its alternatives), we need to look beyond the narrow problem of computing a single decomposition in order to make progress.

Algorithm 1: Common usage scenario of CP-ALS.

Input : $\boldsymbol{\mathcal{T}}$: The tensor to decompose.
$\mathcal{S}$: Set of K starting points.
Output: $\mathcal{P}$: Set of K CP models fitted to the tensor $\boldsymbol{\mathcal{T}}$.

$\mathcal{P} \leftarrow \emptyset$
foreach $\boldsymbol{\mathcal{A}} \in \mathcal{S}$ **do**
  $\boldsymbol{\mathcal{B}} \leftarrow$ CP-ALS($\boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{A}}$)
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{\boldsymbol{\mathcal{B}}\}$
return $\mathcal{P}$

Algorithm 2: Common usage scenario of CALS.

Input : $\boldsymbol{\mathcal{T}}$: The tensor to decompose.
$\mathcal{S}$: Set of K starting points.
Output: $\mathcal{P}$: Set of K CP models fitted to the tensor $\boldsymbol{\mathcal{T}}$.

$\mathcal{P} \leftarrow$ CALS($\boldsymbol{\mathcal{T}}, \mathcal{S}$)
return $\mathcal{P}$

Taking into account that a typical workflow of an application expert involves computing a set of CP decompositions (with varying numbers of components and starting points), instead of optimizing only the CP-ALS procedure, we consider the workflow as a whole (see Algorithm 1). We introduce the *Concurrent ALS* (CALS) algorithm[1], which extends the standard CP-ALS algorithm such that it concurrently computes *a set of* CP decompositions. We provide CALS as a C++ library, with an interface to MATLAB and GPU offloading via CUDA. Within CALS, we combine multiple invocations of CP-ALS at an algorithmic level such that the arithmetic intensity increases — without numerically affecting any of the invocations. Crucially, CALS remains compatible with many enhancements typically used in CP-ALS; to demonstrate, we included line search and non-negativity constraints in our library implementation. A challenge is that different instances of CP-ALS require a varying number of iterations before they converge. Therefore, we include a mechanism to dynamically insert and remove instances with minimal impact on performance.

Contributions. These are the highlights:

- CALS achieves a higher arithmetic intensity than a sequence of CP-ALS invocations, without numerically affecting the computation, and therefore completes a set of decompositions faster.
- We showcase how CALS makes offloading to a GPU worthwhile by increasing the granularity of the central MTTKRP operation, which further increases the speed.
- We demonstrate that CALS is compatible with enhancements of CP-ALS that preserve the central MTTKRP operation by incorporating line search and non-negativity constraints to the CALS library.
- To help application experts take full advantage of the CALS features within their existing source code, we also provide an interface to MATLAB.

[1]Since our algorithm is used to compute the CP decomposition, its proper name is CP-CALS. However, to avoid repetition and make it easier for the reader to differentiate between CP-CALS and CP-ALS, throughout the paper we refer to CP-CALS as just CALS.

Organization. The rest of the paper is organized as follows. In Section 2, we provide an overview of related research. In Section 3, we review the standard CP-ALS algorithm and introduce the basics of CALS. We describe how CALS handles the issue of uneven convergence in Section 4. We present several features of CALS in Section 5. In Section 6, we show experimental results that support the claim that CALS reduces the time to completion by increasing the arithmetic intensity. We also include preliminary experiments on a real dataset from fluorescence spectroscopy to demonstrate its practicality.

Notation. For vectors and matrices, we use bold lowercase and uppercase roman letters, respectively, e.g., $\mathbf{v}$ and $\mathbf{U}$. For tensors, we follow the notation in [20]; specifically, we use bold calligraphic fonts, e.g., $\boldsymbol{\mathcal{T}}$. The order (number of indices or modes) of a tensor is denoted by uppercase roman letters, e.g., N. For each mode n, a tensor $\boldsymbol{\mathcal{T}}$ can be unfolded (matricized) into a matrix, denoted by $\mathbf{T}_{(n)}$, where the columns are the mode-n fibers of $\boldsymbol{\mathcal{T}}$, i.e., the vectors obtained by fixing all indices except for mode n. Sets are denoted by non-bold calligraphic fonts, e.g., $\mathcal{S}$. Given two matrices $\mathbf{A}$ and $\mathbf{B}$, with the same number of columns, the Khatri-Rao product, denoted by $\mathbf{A} \odot \mathbf{B}$, is the column-wise Kronecker product of $\mathbf{A}$ and $\mathbf{B}$.

2 RELATED WORK

Several variations of the CP decomposition have been developed to meet the needs of applications. Examples of constraints on the factor matrices include non-negativity [11], orthogonality [28], and coherence [15]. In so called dictionary-based variants, the columns of a factor matrix are constrained to a given set [13]. Other variations include weighting [24], missing values [31], and alternative objective functions such as the Kullback-Leibler divergence [16].

Whether the given tensor is dense, sparse, or presented in factored form (e.g., Tucker or CP) has a big impact on data structures and algorithms [4]. Other classes of methods besides ALS have been proposed, e.g., methods based on eigendecompositions [14] and methods based on gradient-based (all-at-once) optimization [1].

Several modifications to CP-ALS have been proposed. Line search [26] and extrapolation [3] procedures accelerate convergence and appear to help avoid bad local minima. Pairwise perturbation is a recently proposed acceleration technique that uses error-controlled approximations in order to reduce the arithmetic cost of generating the CP-ALS subproblems [22]. For very large tensors, randomization can reduce the time and space complexity of a CP-ALS iteration. Some randomization methods use random sampling of the tensor [34] while others sample the Khatri-Rao product [7]. Compression-based techniques replace the tensor with an approximation of lower rank, thereby shrinking the effective size of the tensor decomposition problem [10]. A CP decomposition of the compressed tensor is computed and then inflated to form a decomposition of the large tensor.

In the context of large dense tensors and small ranks, the time complexity of the MTTKRP dominates the cost of CP-ALS. Efficient (parallel) algorithms for the MTTKRP have therefore been a target for research. Naïve permute-and-multiply algorithms, which explicitly permute data to generate the unfolding prior to a matrix multiplication, are easy to implement but suffer from large overheads due to the repeated permutations of data. The overhead can be reduced, but not eliminated, with a high-performance tensor transposition library [29]. A leap forward was presented in [25], where the authors showed how an MTTKRP can be done without any permutations at all. They also removed redundant computations across the sequence of related MTTKRPs that appear in a CP-ALS iteration. An alternative approach to avoiding permutation is to recognize that the mode-n unfoldings have a natural block structure. This can be exploited to create a one-step algorithm without permutations [18]. For very large problems, the size of intermediate objects (e.g., Khatri-Rao products) can become an issue. One way to overcome this issue is by using an algorithm

that only requires a constant amount of workspace, like the one proposed in [33]. Unfortunately, despite all these advances, a library that offers a high performance MTTKRP implementation still does not exist.

The technique we propose in this paper complements rather than competes with many of these related efforts. What we propose can be applied alongside other enhancements and even to other methods besides CP-ALS. This is important because there is good reason to believe that it is the accumulative effect of many disparate techniques that will lead to the best performance.

3 ALS AND CONCURRENT ALS (CALS)

We begin by reviewing the regular CP-ALS algorithm in Section 3.1 and introduce the CALS algorithm in Section 3.2.

3.1 ALS

Algorithm 3: CP-ALS: Alternating least squares method for CP decomposition.

Input : $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{I_1,\dots,I_N}$: The tensor to decompose.
$\mathrm{U}^{(1)}, \dots, \mathrm{U}^{(N)}$: The factor matrices of the starting point (rank R).
Output: $\mathrm{U}^{(1)}, \dots, \mathrm{U}^{(N)}$: The computed CP decomposition of $\boldsymbol{\mathcal{T}}$.

1 **repeat**
2 **for** $n = 1, 2, \dots, N$ **do**
3 $\mathrm{M}^{(n)} \leftarrow \mathrm{T}_{(n)}(\odot_{i \neq n} \mathrm{U}^{(i)})$ ▷MTTKRP
4 $\mathrm{H}^{(n)} \leftarrow *_{i \neq n}(\mathrm{U}^{(i)^T}\mathrm{U}^{(i)})$ ▷Hadamard product of Gramians
5 $\mathrm{U}^{(n)} \leftarrow \mathrm{M}^{(n)}\mathrm{H}^{(n)^\dagger}$ ▷$\mathrm{H}^{(n)^\dagger}$: pseudoinverse of $\mathrm{H}^{(n)}$
6 **end**
7 **until** *convergence detected or maximum number of iterations reached*

Algorithm 3 shows the standard alternating least squares method for CP decomposition (CP-ALS). Given a starting point, the factor matrices are repeatedly updated one-by-one in sequence until either convergence is detected or some maximum number of iterations has been reached. When updating the factor matrix for mode-n, the gradient of the objective function with respect to $\mathrm{U}^{(n)}$ is set to zero and the resulting (linear) least squares problem is solved exactly via the normal equations.

Computationally, the most expensive step is the Matricized Tensor Times Khatri-Rao Product (MTTKRP) in line 3. Conceptually, the mode-n unfolding $\mathrm{T}_{(n)}$ is multiplied with the Khatri-Rao Product (KRP) of all factor matrices except $\mathrm{U}^{(n)}$. This involves $2R\prod_i I_i$ flops (not counting the KRP, which accounts for a lower order term) and $\prod_i I_i$ accesses to tensor elements. Thus, the arithmetic intensity of the MTTKRP is $2R$ flops per tensor element access. In line 4, the Gramians of each factor matrix ($\mathrm{U}^{(i)^T}\mathrm{U}^{(i)}$) are multiplied together, to form $\mathrm{H}^{(n)}$, using the Hadamard product. Since $\mathrm{H}^{(n)}$ is of size $R \times R$, the cost of solving the linear system in line 5 is $O(R^2 I_n + R^3)$, which, for small R, is much less than the cost of MTTKRP.

Figure 1 illustrates the practical effect of the $2R$ arithmetic intensity of MTTKRP on its computational efficiency. Efficiency measures the performance of an algorithm relative to the Theoretical Peak Performance (TPP) of the machine it runs on, and is given by the formula:

$$\text{EFFICIENCY} = \frac{\text{PERFORMANCE}}{\text{TPP}} = \frac{\text{TOTAL \#FLOPS/TIME}}{\text{TPP}}$$

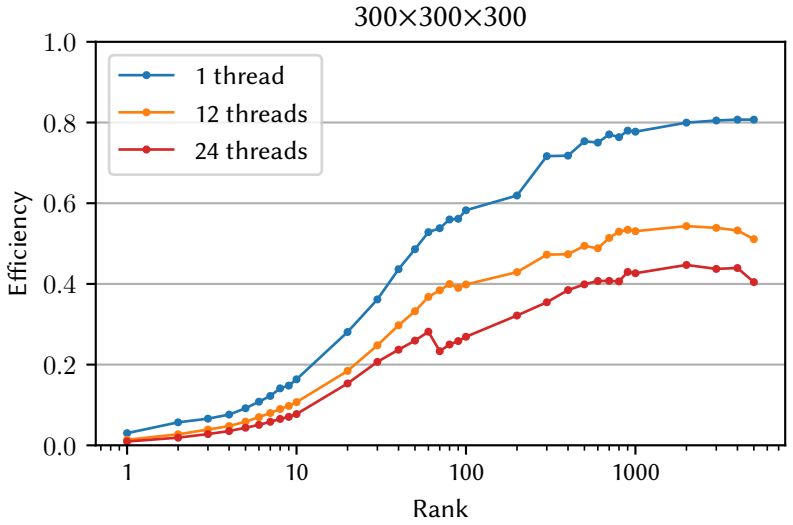

Fig. 1. Efficiency of MTTKRP on a $300 \times 300 \times 300$ tensor for increasing ranks.

For ranks up to 20, the efficiency is $< 30\%$ for single-threaded and $< 20\%$ for multi-threaded execution. An efficiency over 50% is only reached for ranks over 60 and 500 with 1 and 12 threads, respectively, and never reached (at least for ranks up to 5000) with 24 threads. According to Figure 1, the efficiency, for a tensor of size $300 \times 300 \times 300$, increases with R until eventually reaching a plateau, which in this case occurs around $R = 1000$. Hence, the computational resources tend to be better utilized when the decomposition has many components rather than few. Given that MTTKRP accounts for the bulk of the cost of CP-ALS, the efficiency profiles of MTTKRP and CP-ALS are similar. The particular implementations used to maximize efficiency are explained in Section 5.1.

3.2 Concurrent ALS (CALS)

A straightforward way to run K instances of CP-ALS on the same tensor is to run them one-by-one in sequence as in Algorithm 1 or, in the case of multi-threading, to run multiple instances in parallel. As explained above, the arithmetic intensity when fitting model i of rank R_i will be only $2R_i$. The gist of CALS is to reorganize the computations involved in K instances of CP-ALS into a form which achieves higher arithmetic intensity (and hence higher efficiency), namely $2\sum_{i=1}^{K} R_i$. In particular, if $K = 100$ and $R_i = 10$ for all i, then from Figure 1 we expect about 17% MTTKRP efficiency for a sequence of single-threaded CP-ALS instances. With CALS we will soon see that we expect an MTTKRP efficiency closer to that observed for $R = \sum_{i=1}^{K} R_i = 1000$ or about 78% — a speedup of $4.5\times$.

The key idea of CALS is to fuse K small and independent MTTKRPs with low arithmetic intensity into one large MTTKRP with higher arithmetic intensity. To see how this can be done, first note that (for appropriately sized matrices) independent KRPs can be fused into a larger KRP:

$$\begin{bmatrix} \mathrm{A}_1 \odot \mathrm{B}_1 & \mathrm{A}_2 \odot \mathrm{B}_2 \end{bmatrix} = \begin{bmatrix} \mathrm{A}_1 & \mathrm{A}_2 \end{bmatrix} \odot \begin{bmatrix} \mathrm{B}_1 & \mathrm{B}_2 \end{bmatrix}. \tag{1}$$

This allows the fusion of two (or more) independent MTTKRPs:

$$\begin{aligned}\begin{bmatrix}\textsc{mttkrp}(\mathrm{X}, \mathrm{A}_1, \mathrm{B}_1) & \textsc{mttkrp}(\mathrm{X}, \mathrm{A}_2, \mathrm{B}_2)\end{bmatrix} &= \begin{bmatrix}\mathrm{X}(\mathrm{A}_1 \odot \mathrm{B}_1) & \mathrm{X}(\mathrm{A}_2 \odot \mathrm{B}_2)\end{bmatrix} \\ &= \mathrm{X}\begin{bmatrix}\mathrm{A}_1 \odot \mathrm{B}_1 & \mathrm{A}_2 \odot \mathrm{B}_2\end{bmatrix} \\ &= \mathrm{X}\left(\begin{bmatrix}\mathrm{A}_1 & \mathrm{A}_2\end{bmatrix} \odot \begin{bmatrix}\mathrm{B}_1 & \mathrm{B}_2\end{bmatrix}\right) \\ &= \textsc{mttkrp}(\mathrm{X}, \begin{bmatrix}\mathrm{A}_1 & \mathrm{A}_2\end{bmatrix}, \begin{bmatrix}\mathrm{B}_1 & \mathrm{B}_2\end{bmatrix}).\end{aligned}$$

Starting from two independent MTTKRPs, the common matrix factor X is extracted and the KRPs are fused together using (1). The result is a single, larger MTTKRP.

Generalized to a tensor $\boldsymbol{\mathcal{T}}$ of order N, the mode-n MTTKRPs for two independent CP-ALS instances (on the same tensor) can be fused:

$$\left[\mathrm{T}_{(n)}(\odot_{i\neq n}\mathrm{U}_1^{(i)}) \quad \mathrm{T}_{(n)}(\odot_{i\neq n}\mathrm{U}_2^{(i)})\right] = \mathrm{T}_{(n)}\left(\odot_{i\neq n}\begin{bmatrix}\mathrm{U}_1^{(i)} & \mathrm{U}_2^{(i)}\end{bmatrix}\right). \tag{2}$$

This naturally extends to any number of CP-ALS instances.

Seemingly, to apply the fused MTTKRP in (2), it suffices to concatenate factor matrices and extract submatrices from the result corresponding to the small MTTKRPs. This could potentially be very expensive. One should instead work directly on the concatenated form of the factor matrices. Let $\mathrm{A}_1, \mathrm{A}_2, \ldots, \mathrm{A}_K$ be matrices with the same number of rows. Then we refer to the horizontal concatenation $\overline{\mathrm{A}} = \begin{bmatrix}\mathrm{A}_1 & \mathrm{A}_2 & \ldots & \mathrm{A}_K\end{bmatrix}$ as a *multi-matrix*. The k-th constituent matrix of $\overline{\mathrm{A}}$ is denoted by $\overline{\mathrm{A}}_{|k}$, so $\overline{\mathrm{A}}_{|k} \equiv \mathrm{A}_k$.

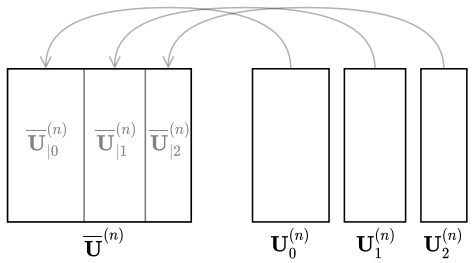

Fig. 2. Horizontally concatenating factor matrices into a multi-matrix.

During initialization, CALS creates N multi-matrices $\overline{\mathrm{U}}^{(i)}$, one for each mode $i = 1, 2, \ldots, N$ by concatenating the initial factor matrices from each starting point[2], as shown in Figure 2. When a single factor matrix is needed, say to compute a Gramian, then it is accessed as a submatrix of the corresponding multi-matrix. The fused MTTKRP is readily computed using the multi-matrices as input. The outputs of the small MTTKRPs are the constituent matrices of the multi-matrix output of the fused MTTKRP. In summary, after initialization neither concatenation nor extraction is necessary.

A simplified version of CALS is described in Algorithm 4. (In Section 4, we will extend the algorithm to appropriately handle a large number of instances and convergence at different iterations.) First, the N factor multi-matrices are initialized by packing the factor matrices from the starting points. Then the main phase begins. The K small MTTKRPs are fused into one large MTTKRP

[2]In practice, each multi-matrix is allocated to a fixed size, optionally specified by the user. Then the multi-matrices are filled with horizontally concatenated factor matrices from each starting point, until they are full. When models converge, as the algorithm progresses, space is freed up, and new starting points can take their place. More details about this mechanism are provided in Section 4.

Algorithm 4: CALS: Concurrent alternating least squares method for a set of independent CP decompositions of one tensor. Basic algorithm without handling of uneven convergence.

Input : $\mathcal{T} \in \mathbb{R}^{I_1,\dots,I_N}$: The tensor to decompose.
$\mathrm{U}_1^{(1)}, \dots, \mathrm{U}_K^{(N)}$: The factor matrices of the K initial points (rank R_i for $i = 1, 2, \dots K$).
Output: $\mathrm{U}_1^{(1)}, \dots, \mathrm{U}_K^{(N)}$: The K computed CP decompositions of $\mathcal{T}$.

1 **for** $n = 1, 2, \dots, N$ **do** ▷Initialize one factor multi-matrix for each mode.
2 **for** $k = 1, 2, \dots, K$ **do**
3 $\overline{\mathrm{U}}_{|k}^{(n)} \leftarrow \mathrm{U}_k^{(n)}$
4 **end**
5 **end**
6 **repeat** ▷Concurrently run K instances of CP-ALS.
7 **for** $n = 1, 2, \dots, N$ **do**
8 $\overline{\mathrm{M}}^{(n)} \leftarrow \mathrm{T}_{(n)}(\odot_{i \neq n} \overline{\mathrm{U}}^{(i)})$
9 **for** $k = 1, 2, \dots, K$ **do**
10 $\mathrm{H}_k^{(n)} \leftarrow *_{i \neq n}(\overline{\mathrm{U}}_{|k}^{(i)^T} \overline{\mathrm{U}}_{|k}^{(i)})$
11 $\overline{\mathrm{U}}_{|k}^{(n)} \leftarrow \overline{\mathrm{M}}_{|k}^{(n)} \mathrm{H}_k^{(n)^\dagger}$
12 **end**
13 **end**
14 **until** *convergence detected for all instances or maximum number of iterations reached*

in line 8. The remaining parts of the algorithm are essentially the same as for regular CP-ALS (Algorithm 3) and the instances are treated separately in a loop. In particular, lines 10 and 11 of Algorithm 4 correspond to lines 4 and 5 of Algorithm 3.

The efficiency of CALS is determined by the efficiency of MTTKRP (see Figure 1) for rank $\sum_{i=1}^{K} R_i$, regardless of the individual ranks. In contrast, for CP-ALS, the efficiency when computing the ith instance is determined by R_i.

4 HANDLING CONVERGENCE

The idea of CALS is to concurrently run multiple instances of CP-ALS on the same tensor. The ranks of the individual decompositions may or may not be equal. The starting points can also be different. The CALS algorithm synchronously advances all instances one iteration at a time. But the number of iterations required for an instance to converge can vary from a few to thousands. If convergence is detected for one instance, then that instance should be removed from further processing. Naturally, one may also want to add new instances during the execution. In this section, we describe how CALS handles the dynamic insertion and removal of instances and present the full CALS algorithm (Algorithm 5).

In the CALS library, a multi-matrix is stored in column-major format at the beginning of a memory buffer of fixed size. Constituent matrices are therefore contiguous in memory, which implies that a multi-matrix can grow at the end without moving data around. One multi-matrix is allocated for each mode for the purpose of storing factor matrices. The size of the buffer for mode n is set to $I_n R^*$, where R^* is a user-defined constant specifying the maximum width (number of columns) of a multi-matrix. (R^* must be at least as large as the largest rank.) Ideally, R^* is set to the value which maximizes MTTKRP's performance, which, according to Figure 1, for a $300 \times 300 \times 300$

Algorithm 5: CALS: Concurrent alternating least squares method for a set of independent CP decompositions of one tensor. Full algorithm.

Input : $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{I_1,\dots,I_N}$: The tensor to decompose.
Q_{in}: Input queue with starting points.
Output: Q_{out}: Output queue for computed CP decompositions.

$n_{\text{active}} \leftarrow 0$ ▷Initialize.
Allocate buffers for N factor multi-matrices $\overline{\text{U}}^{(i)}$ for $i = 1, 2, \dots, N$
repeat ▷Main loop.
  while Q_{in} *is not empty* **do** ▷Fill the factor multi-matrices.
    $\boldsymbol{\mathcal{A}} \leftarrow \textsc{Front}(Q_{\text{in}})$ ▷Look at first element of queue.
    if $\boldsymbol{\mathcal{A}}$ *can be inserted into the factor multi-matrices* **then**
      $n_{\text{active}} \leftarrow n_{\text{active}} + 1$
      Insert $\boldsymbol{\mathcal{A}}$ into the factor multi-matrices
      $\textsc{Dequeue}(Q_{\text{in}})$
    else
      Exit the loop
    end
  end
  for $n = 1, 2, \dots, N$ **do** ▷Advance all instances one iteration.
    $\overline{\text{M}}^{(n)} \leftarrow \text{T}_{(n)}(\odot_{i \neq n} \overline{\text{U}}^{(i)})$ ▷Fused MTTKRP.
    for $k = 1, 2, \dots, n_{\text{active}}$ **do** ▷Separate processing of active instances.
      $\text{H}_k^{(n)} \leftarrow *_{i \neq n}(\overline{\text{U}}_{|k}^{(i)^T} \overline{\text{U}}_{|k}^{(i)})$
      $\overline{\text{U}}_{|k}^{(n)} \leftarrow \overline{\text{M}}_{|k}^{(n)} \text{H}_k^{(n)^\dagger}$
    end
  end
  for $k = 1, 2, \dots, n_{\text{active}}$ **do** ▷Remove converged and stalled instances.
    Let $\boldsymbol{\mathcal{P}}$ denote the kth active instance: $\overline{\text{U}}_{|k}^{(1)}, \dots, \overline{\text{U}}_{|k}^{(N)}$
    $E, F \leftarrow$ error and fit of $\boldsymbol{\mathcal{P}}$
    if $F - F_{prev} < \text{tol}$ *or maximum number of iterations reached* **then**
      Add $\boldsymbol{\mathcal{P}}$ to Q_{out}
      Remove the kth constituent matrix from each factor multi-matrix
      $n_{\text{active}} \leftarrow n_{\text{active}} - 1$
    end
  end
until Q_{in} *is empty and* $n_{\text{active}} = 0$

tensor, plateaus at approximately $R^* = 1000$. Choosing a larger R^* would not affect performance significantly in this case. This parameter can be chosen to trade off performance versus memory consumption. The buffer size is enough to concurrently run instances with a rank sum $\leq R^*$. Any additional instances will have to wait in a queue until space is freed up by converged instances.

Algorithm 5 takes a tensor and a queue of starting points as input. The algorithm starts by allocating buffers for the factor multi-matrices and then enters the main loop, which concludes when the input queue is empty and there are no more active instances. At the start of each iteration

(lines 4–12), the factor multi-matrices are filled with starting points from the input queue until either the buffers are full or the input queue is empty. Next (lines 14–19), all active instances are advanced one iteration. At the end of each iteration (lines 21–28), the error and fit of each instance is computed. If the difference in fit between two consecutive iterations is lower than some tolerance, then the instance is considered converged and its factor matrices are copied out of the factor multi-matrices and placed in the output queue. The removal of instances may cause the buffers of the multi-matrices to become fragmented, i.e., they may contain gaps of unused space in between constituent matrices. To get rid of the fragmentation, a compression routine packs the constituent matrices contiguously starting at the beginning of the respective buffers.

5 SOFTWARE FEATURES

In this section, we describe the various components of CALS (available on Github[3]), which include MTTKRP, line search, and non-negativity constraints as well as features, such as GPU offloading and the MATLAB interface.

CALS is written in C++ and depends on the BLAS and LAPACK libraries (supported by, e.g., Intel® MKL [19], BLIS [32], and OpenBLAS [37]). Optional dependencies are CUDA [23] (for GPU offloading) and MATLAB [30].

5.1 MTTKRP

The MTTKRP operation accounts for the vast majority of the cost of both CP-ALS and CALS. There does not exist a highly optimized black-box implementation of MTTKRP that performs well for all modes and sizes. Recent research (e.g., [18]) has shown that the keys to fast MTTKRP is to (a) avoid data permutations in memory, (b) cast the computations in terms of GEMM BLAS operations, and (c) avoid explicitly computing Khatri-Rao products in certain cases.

To this end, CALS includes a family of algorithmic variants for MTTKRP. Depending on such things as the size and shape of the tensor, the rank of the decomposition, the mode of the MTTKRP, and the method used for parallelization, the variants vary greatly in terms of performance. No single variant is best in all cases. Finding an effective way of determining a good variant for a particular case is beyond the scope of this paper. For third-order tensors, we hardcoded the choice of the best variant for each case and each algorithm (CP-ALS or CALS), according to benchmarks. For higher-order tensors, CALS falls back to computing the Khatri-Rao product explicitly and then performing one, or several, matrix multiplications.

5.2 Line search

Line search is a technique that can reduce the total number of iterations to reach convergence in CP-ALS and to some extent can also help to avoid getting trapped in bad local minima. For these reasons, line search is used by most high-quality implementations of CP-ALS (and other methods). In order to demonstrate that CALS is compatible with line search, we include a basic version (see, e.g., [8, 17]) as an optional feature. Let $\boldsymbol{\mathcal{P}}^{(i)}$ and $\boldsymbol{\mathcal{P}}^{(i+1)}$ denote points after the CP-ALS iterations i and $i+1$, respectively. Then after iteration $i+1$, line search linearly extrapolates to a new point $\boldsymbol{\mathcal{P}}^{\text{new}} = \boldsymbol{\mathcal{P}}^{(i)} + \alpha(\boldsymbol{\mathcal{P}}^{(i+1)} - \boldsymbol{\mathcal{P}}^{(i)})$, where $\alpha > 1$ determines the amount of extrapolation. If $\boldsymbol{\mathcal{P}}^{\text{new}}$ provides a better fit than $\boldsymbol{\mathcal{P}}^{(i+1)}$, then $\boldsymbol{\mathcal{P}}^{(i+1)}$ is replaced with $\boldsymbol{\mathcal{P}}^{\text{new}}$. Otherwise, the extrapolated point is rejected. Regarding the choice of α, several proposals have been made. For example, Bro [8] recommends setting $\alpha = \sqrt[3]{i}$ based on experience.

In CALS, line search is applied independently to each instance as follows. Let $\overline{\mathrm{U}}^{(n)}_{|k,i}$ and $\overline{\mathrm{U}}^{(n)}_{|k,i+1}$ be the factor matrices for each mode n, of an instance k, for iterations i and $i+1$ respectively. The new,

[3]https://github.com/HPAC/CP-CALS

extrapolated, factor matrices are given by $\overline{\mathrm{U}}^{(n)}_{|k,\mathrm{new}} = \overline{\mathrm{U}}^{(n)}_{|k,i} + \alpha(\overline{\mathrm{U}}^{(n)}_{|k,i+1} - \overline{\mathrm{U}}^{(n)}_{|k,i})$, for each n, where $\alpha > 1$ is a user-defined constant. Given $\overline{\mathrm{U}}^{(n)}_{|k,\mathrm{new}}$, the errors after iterations i and $i+1$ are computed and the extrapolated point is used only if it shows an improvement.

5.3 Non-negativity constraints

CALS is also compatible with active set-based approaches for enforcing non-negativity constraints to the factor matrices. To demonstrate this, we included the active set-based method proposed in [11] as a feature in CALS.

The non-negativity constraints propagate down to the least squares subproblem that updates a factor matrix (see line 5 in Algorithm 3). The subproblem changes into a *non-negative* least squares problem with multiple right-hand sides. Mirroring regular CP-ALS, the method proposed in [11] computes the MTTKRP and the Hadamard product of the Gramians only once per inner iteration. What changes compared to CP-ALS is the update step, which is replaced by an iterative search for the right set of active constraints. Since the size of the matrices involved in this search is of the order of the rank of an individual decomposition, the cost is negligible in our context of small ranks and large tensors.

In CALS, the active set-based method is applied to each decomposition independently after the fused MTTKRP in line 8 of Algorithm 4. The final active sets are saved for the next iteration for the purposes of warm starting, as described and justified in [11].

5.4 GPU offloading

GPUs offer tremendous computational capabilities, especially for matrix operations. While the MTTKRP in CP-ALS can in principle be offloaded to a GPU, the associated overhead likely outweighs the gains due to the low arithmetic intensity. The larger arithmetic intensity achieved by CALS makes it more suitable for offloading its MTTKRP to a GPU. To take advantage of this extra capability CALS has to offer, we developed a CUDA interface to MTTKRP.

The elements required to perform MTTKRP on the GPU are the factor multi-matrices and the tensor. Since the tensor does not change, we transfer it to the GPU during initialization and let it remain there until the end. We also transfer the multi-matrices to the GPU during initialization, and then, for each iteration, only the multi-matrix being updated is transferred back and forth between the GPU and CPU. Specifically, after line 8 of Algorithm 4, $\overline{\mathrm{M}}^{(n)}$ is brought to the CPU to update $\overline{\mathrm{U}}^{(n)}_{|k}$ for each instance k. Once all instances have had their updates completed, $\overline{\mathrm{U}}^{(n)}$ is sent back to the GPU. For the algorithmic variants of MTTKRP that require an explicit KRP, the KRP is computed on the CPU and transferred to the GPU.

Section 6.3 showcases the additional speedup gained by from offloading computation to the GPU.

5.5 MATLAB interface

Some of the most popular software packages used in applications (e.g., Tensor Toolbox [5, 6], the N-way Toolbox [9], and Tensorlab [35]) are developed for MATLAB. To make CALS accessible to MATLAB users, we provide a MATLAB interface via MEX through a function called `cp_cals`. The arguments of this function are similar to the `cp_als` function in Tensor Toolbox with some extra optional arguments that enable features such as line search, non-negativity, and GPU offloading.

6 EXPERIMENTS

In this section, we demonstrate the performance improvements of CALS over CP-ALS. First, we isolate the performance impact of the increased arithmetic intensity achieved by CALS. To this end,

we present two synthetic experiments, one targeting speedup and the other efficiency. Second, we showcase the performance improvements that CALS can offer in a real world application. Third, we highlight the effect of GPU offloading made feasible by CALS.

The experiments were performed on a Linux-based system with an Intel® Xeon® Platinum 8160 Processor, which contains 24 physical cores split in 2 NUMA regions with 12 cores each. We report results for 1 thread, 12 threads (one NUMA region), and 24 threads (both NUMA regions). The code was compiled using GCC[4] and linked with the Intel® Math Kernel Library version 19.0, which implements a superset of BLAS and LAPACK. For the CUDA experiments, an Nvidia Tesla V100 was used[5]. For the MATLAB experiments, MATLAB version 2019b was used. The source code is available online.

6.1 Arithmetic intensity

We present experiments that isolate the impact of increased arithmetic intensity.

6.1.1 Speedup per iteration. Since CALS achieves a higher arithmetic intensity than CP-ALS, we expect CALS to complete one iteration on K instances faster than CP-ALS completes one iteration on K instances. We measured the size of this speedup with the following experiment. For a given tensor and rank, we ran both CALS and CP-ALS on $K = 20$ random starting points. All instances ran for exactly 50 iterations and we measured the total time for each method. We then calculated the average speedup per iteration using the formula

$$\text{SPEEDUP PER ITERATION} = \frac{\text{TIME FOR CP-ALS}}{\text{TIME FOR CALS}}.$$

The size of the tensor, the rank, and the multi-threading configuration all play important roles in determining the execution time. We therefore repeated the experiment for all ranks $R \in \{1, 2, \ldots, 20\}$ on cubic tensors of size $n \times n \times n$ for all $n \in \{100, 200, 300\}$. For each problem configuration, we tested 1 thread, 12 threads, 24 threads, and CUDA (24 CPU threads plus one GPU).

The results are shown in Figure 3. CALS is faster than CP-ALS across all ranks, and especially so on the smaller ranks where MTTKRP is particularly inefficient. The speedup seems to increase with the volume of the tensor. In particular, GPU offloading leads to significant speedups over multi-threaded CP-ALS on the larger tensors, reaching up to $60\times$ for the smallest ranks.

6.1.2 Efficiency. The efficiency of CALS and CP-ALS, relative to the machine's theoretical capabilities, was measured with the following experiment. We used a set of $K = 400$ instances distributed over ranks 1 through 20 with 20 instances for each rank. Hence, the sum of all ranks is $\sum_{R=1}^{20} 20R = 4200$. All instances ran for exactly 50 iterations. We repeated the experiment for cubic tensors of size $n \times n \times n$ for all $n \in \{100, 200, 300\}$. For each problem configuration, we tested 1 thread, 12 threads, and 24 threads.

The buffer size for CALS (R^*) was chosen to maximize performance in the BLAS based on the MTTKRP benchmarks reported in Figure 4. Specifically, $R^* = 4200$ in all cases except for the single-threaded case on the $100 \times 100 \times 100$ tensor, where a suboptimal implementation of the BLAS routine `dgemm` made $R^* = 90$ a better choice.

The results of the experiment are shown in Figure 5. This figure requires a bit of explanation to comprehend. We partitioned the total computation into segments: One segment per iteration for CALS and one segment per instance for CP-ALS. For each segment, we measured the time and

[4]GCC version 8.2.0 with optimization flag -O3

[5]Driver version: 450.51.06, CUDA Version: 11.0

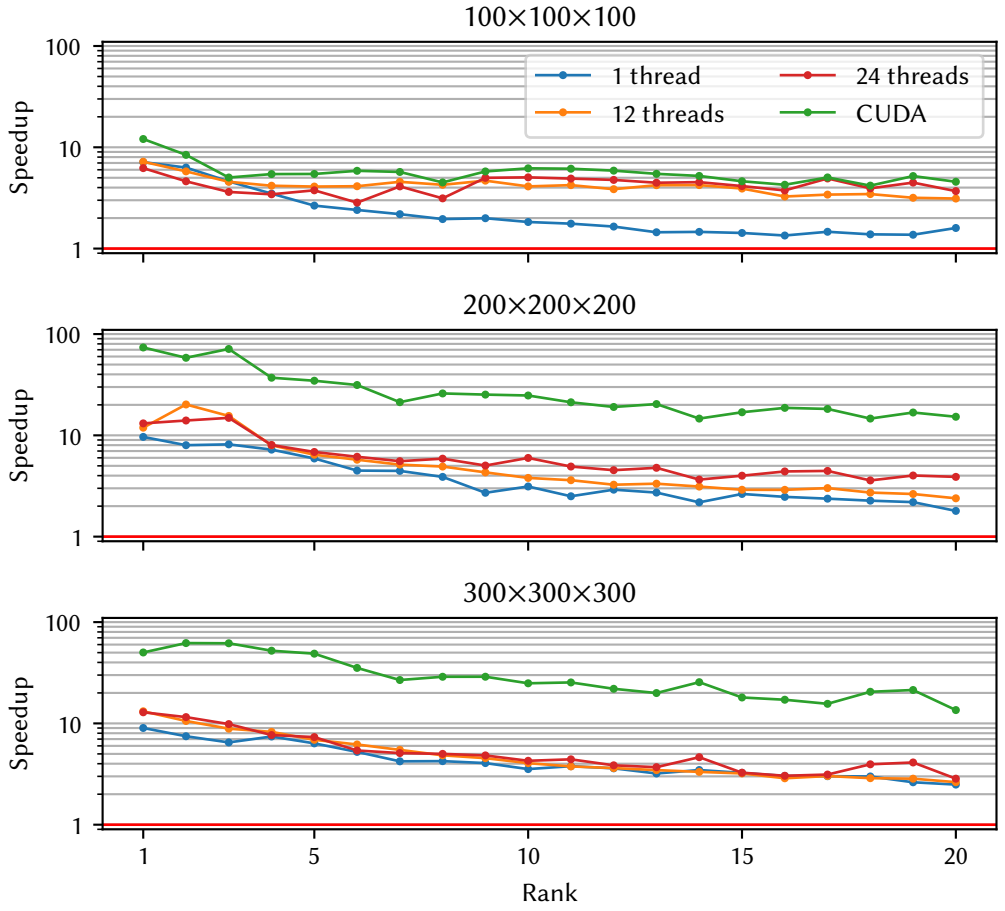

Fig. 3. Average speedup per iteration of CALS over CP-ALS.

counted the FLOPs. We then calculated the efficiency for each time segment using the formula

$$\text{EFFICIENCY} = \frac{\#\text{FLOPS}/\text{TIME}}{\text{TPP}}.$$

The figure shows the efficiency versus the fraction of the total computation performed (measured in FLOPs and normalized). Each time segment corresponds one piece of the piece-wise constant graphs. Note that both CALS and CP-ALS perform the same number of flops.

Since reaching the theoretical peak efficiency of 100% is impossible in practice, we also plot the efficiency of the BLAS routine `dgemm` as an indicator of practical peak efficiency. The practical peak is not reproducible, so we characterize it using the mean, median, and standard deviation over many repetitions. We ran `dgemm` on square matrices of size $m \times m$, where $m = \sqrt{n^3}$. From 100 samples, we calculated the arithmetic mean, median, and standard deviation and converted times to efficiency using the formula above. The mean is shown as a solid line, the median as a dashed line, and the mean ± one standard deviation is shown as a colored region.

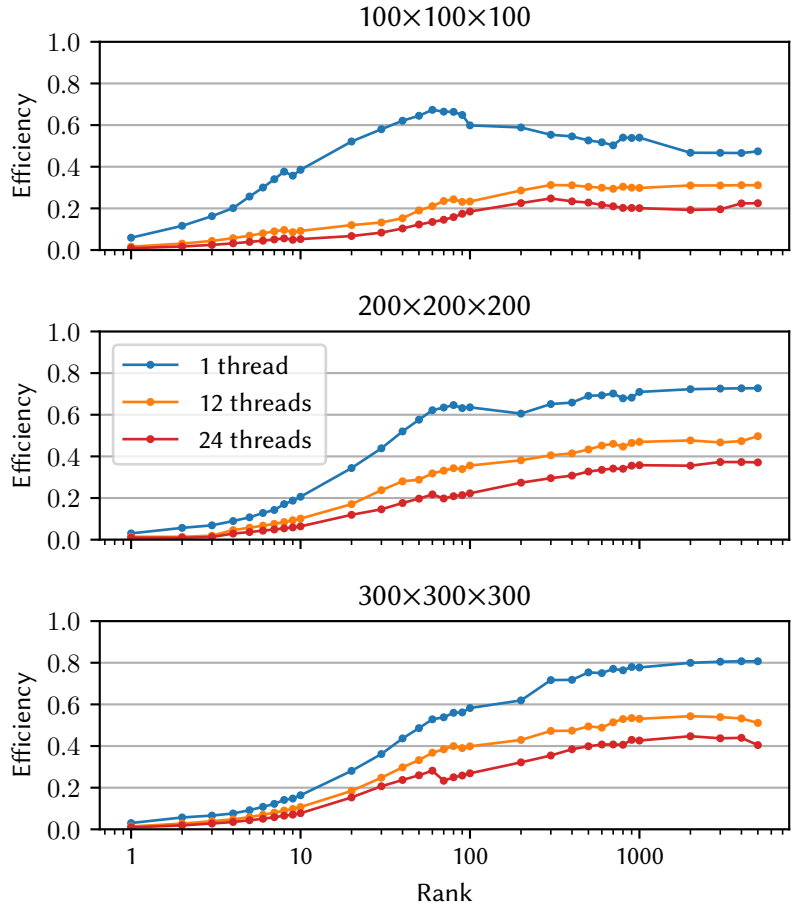

Fig. 4. Performance of the best available implementation of MTTKRP.

CP-ALS processes the ranks in ascending order, and the transition points are indicated by green tick marks above the horizontal axis. Observe that the efficiency steadily increases during the computation, which follows the MTTKRP benchmarks reported in Figure 4.

CALS, on the other hand, processes all ranks concurrently and is able to reach the peak efficiency of MTTKRP (shown in Figure 4) for R^* for all experiments. The occasional drops in performance for CALS, especially in the single-threaded case, are likely due to CPU frequency throttling. The locations of these drops are not reproducible, from which we conclude that they are not caused by the algorithm.

6.2 Application: EEM analysis

We demonstrate the effectiveness of CALS on the real *Fluorescence* dataset [21] coming from an application involving blood plasma for cancer diagnostics. We compare CALS with CP-ALS and the `cp_als` function from Tensor Toolbox. We used a total of $K = 400$ instances distributed over

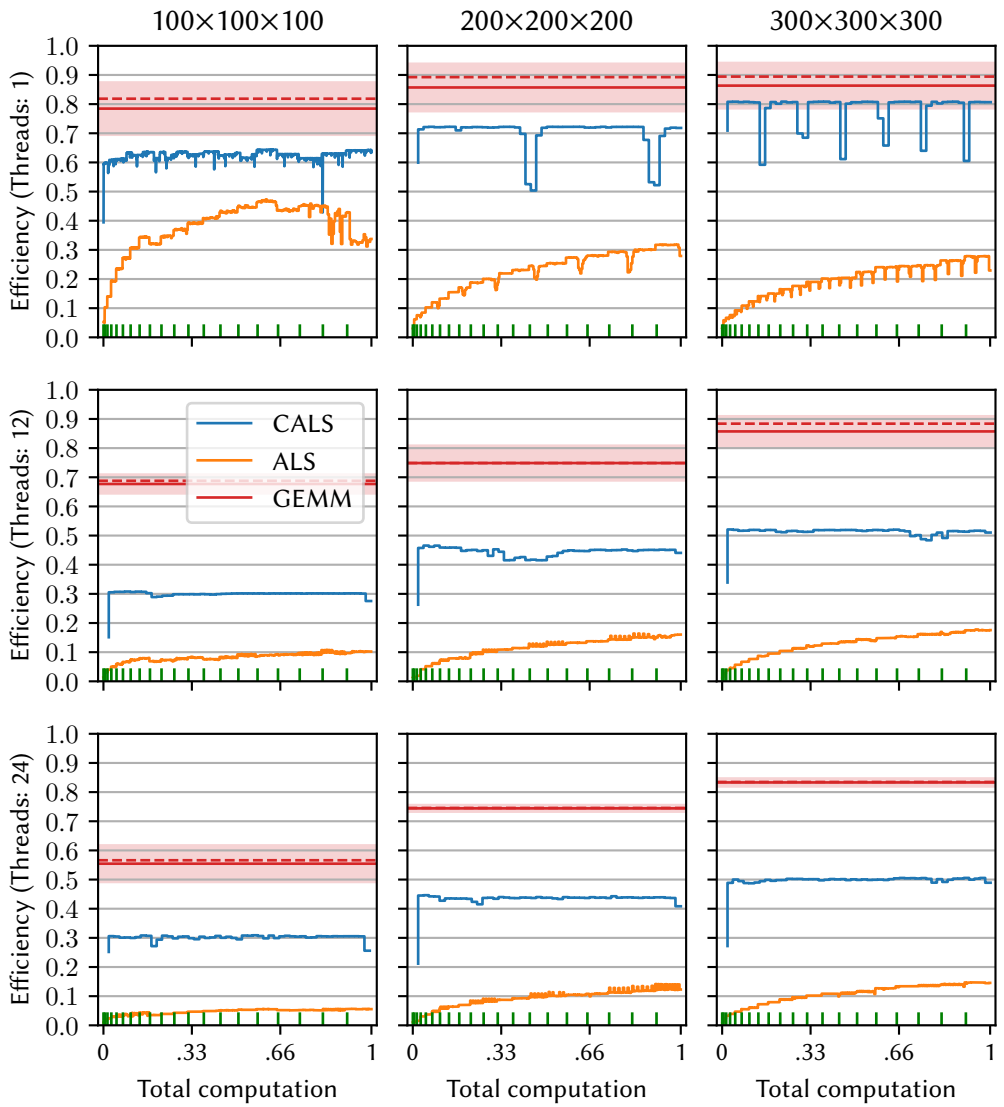

Fig. 5. The efficiency of CALS and CP-ALS during the computation of 400 decompositions of ranks 1 through 20 with 20 decompositions for each rank. CP-ALS processes the ranks in ascending order. For reference, the dgemm efficiency is shown as a red solid line (mean), dashed line (median), and region (mean ± one standard deviation); see the text for details.

ranks 1 through 20 with 20 instances per rank. The tolerance was set to 10^{-4} and the maximum number of iterations was set to 50.

The results are shown in Figure 6. There is a clear advantage for CALS, exhibiting speedups of 5.6× (1 thread), 13.8× (12 threads), and 17× (24 threads) compared to `cp_als` of Tensor Toolbox. Similarly, CALS achieves a 3.3× speedup compared to CP-ALS for all thread configuations. Note

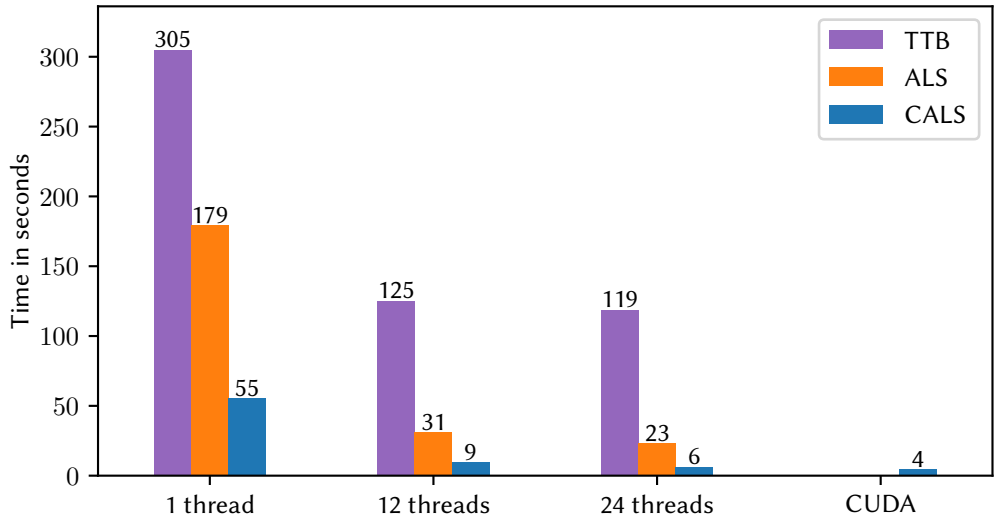

Fig. 6. Times for CALS, CP-ALS, and `cp_als` from Tensor Toolbox on the fluorescence dataset.

that only CALS benefits from GPU offloading, and its CUDA implementation achieves a $5.8\times$ speedup over CP-ALS.

6.3 GPU utilization

To demonstrate the effectiveness of utilizing the GPU within CALS, we repeated the EEM analysis experiments using the CUDA capabilities of CALS. The resulting speedups are summarized in Figure 7. Using a GPU is evidently more beneficial for larger tensors. CALS with GPU offloading achieves speedups of $\times 4.3$ and $\times 4.6$ for tensor sizes $200 \times 200 \times 200$ and $300 \times 300 \times 300$ compared to CALS without GPU offloading using 24 threads.

7 CONCLUSION

In this paper we present Concurrent ALS (CALS), an algorithm and library, which offers an interface to MATLAB, for computing multiple, concurrent Alternating Least Squares algorithms for the Canonical Polyadic Decomposition. We show that CALS is able to accommodate applications that fit multiple models of different ranks and starting points, by achieving better efficiency for the same computation. Furthermore, we demonstrate how higher efficiency favors the, otherwise impractical under CP-ALS, offloading of the computation to GPUs to further speed up computation. Finally, we showcase the effectiveness of CALS over our own optimized version of CP-ALS as well as Tensor Toolbox's implementatio `cp_als` on both artificial and real datasets.

ACKNOWLEDGMENTS

We would like to thank Professor Rasmus Bro, from the University of Copenhagen, for taking us through his development process as an application expert, which inspired the work presented in this paper. Financial support from the Deutsche Forschungsgemeinschaft (German Research Foundation) through grant IRTG 2379 is gratefully acknowledged.

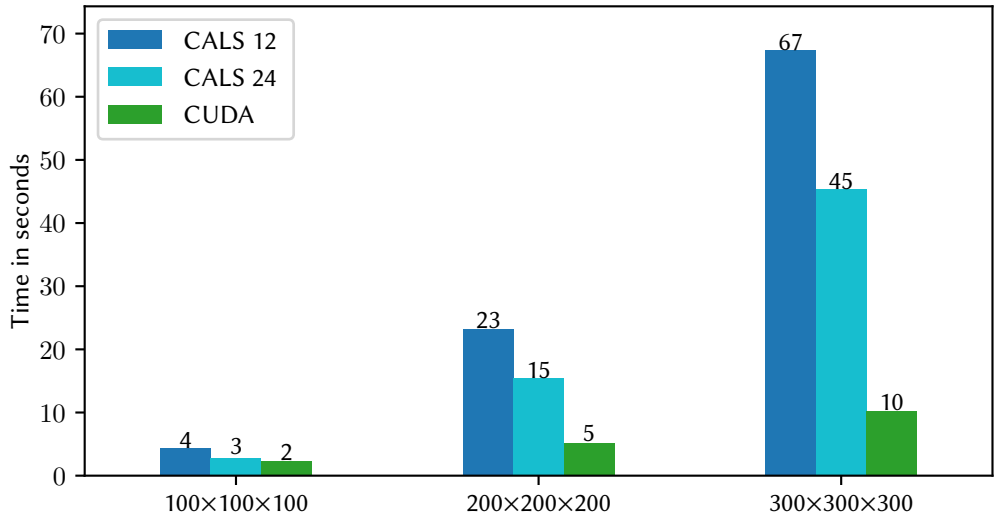

Fig. 7. Times for CALS (with 12 and 24 threads) and CALS with GPU offloading (and 24 threads) on the fluorescence dataset.

REFERENCES

[1] E. Acar, D. M. Dunlavy, and T. G. Kolda. 2011. A scalable optimization approach for fitting canonical tensor decompositions. *Journal of Chemometrics* 25, 2 (2011), 67–86. https://doi.org/10.1002/cem.1335

[2] C.M. Andersen and Rasmus Bro. 2003. Practical aspects of PARAFAC modeling of fluorescence excitation-emission data. *Journal of Chemometrics* 17 (04 2003), 200 – 215. https://doi.org/10.1002/cem.790

[3] A. M. S. Ang, J. E. Cohen, L. T. K. Hien, and N. Gillis. 2019. Extrapolated alternating algorithms for approximate canonical polyadic decomposition. In *ICASSP 2020 – 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 3147–3151. https://doi.org/10.1109/ICASSP40776.2020.9053849

[4] B. Bader and T. Kolda. 2007. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2007), 205–231. https://doi.org/10.1137/060676489

[5] Brett W. Bader and Tamara G. Kolda. 2006. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Trans. Math. Software* 32, 4 (Dec. 2006), 635–653. https://doi.org/10.1145/1186785.1186794

[6] Brett W. Bader, Tamara G. Kolda, et al. 2019. MATLAB Tensor Toolbox Version 3.1. Available online. https://www.tensortoolbox.org

[7] C. Battaglino, G. Ballard, and T. G. Kolda. 2018. A practical randomized CP tensor decomposition. *SIAM J. Matrix Anal. Appl.* 39, 2 (2018), 876–901. https://doi.org/10.1137/17M1112303

[8] Rasmus Bro. 1998. *Multi-way analysis in the food industry. Models, Algorithms, and Applications.* Ph.D. Dissertation. University of Amsterdam.

[9] Rasmus Bro. 2020. The N-way Toolbox. MATLAB Central File Exchange. https://www.mathworks.com/matlabcentral/fileexchange/1088-the-n-way-toolbox

[10] R. Bro and C. A. Andersson. 1998. Improving the speed of multiway algorithms: Part II: Compression. *Chemometrics and Intelligent Laboratory Systems* 42, 1–2 (1998), 105–113. https://doi.org/10.1016/S0169-7439(98)00011-2

[11] Rasmus Bro and Sijmen De Jong. 1997. A fast non-negativity-constrained least squares algorithm. *Journal of Chemometrics: A Journal of the Chemometrics Society* 11, 5 (1997), 393–401.

[12] J. Douglas Carroll and Jih-Jie Chang. 1970. Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition. *Psychometrika* 35 (1970), 283–319.

[13] J. E. Cohen and N. Gillis. 2018. Dictionary-based tensor canonical polyadic decomposition. *IEEE Transactions on Signal Processing* 66, 7 (2018), 1876–1889. https://doi.org/10.1109/TSP.2017.2777393

[14] I. Domanov and L. De Lathauwer. 2014. Canonical polyadic decomposition of third-order tensors: reduction to generalized eigenvalue decomposition. *SIAM J. Matrix Anal. Appl.* 35, 2 (2014), 636–660. https://doi.org/10.1137/

130916084

[15] R. C. Farias, J. H. de Morais Goulart, and P. Comon. 2018. Coherence constrained alternating least squares. In *2018 26th European Signal Processing Conference (EUSIPCO)*. 613–617. https://doi.org/10.23919/EUSIPCO.2018.8553084

[16] S. Hansen, T. Plantenga, and T. G. Kolda. 2015. Newton-based optimization for Kullback–Leibler nonnegative tensor factorizations. *Optimization Methods and Software* 30, 5 (2015), 1002–1029. https://doi.org/10.1080/10556788.2015.1009977

[17] Richard A. Harshman. 1970. Foundations of the PARAFAC procedure: Models and conditions for an explanatory multi-modal factor analysis. *UCLA Working Papers in Phonetics* 16 (1970), 1–84.

[18] Koby Hayashi, Grey Ballard, Yujie Jiang, and Michael J. Tobia. 2018. Shared-Memory Parallelization of MTTKRP for Dense Tensors. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 393–394. https://doi.org/10.1145/3178487.3178522

[19] Intel Corporation. 2020. Intel® Math Kernel Library documentation. https://software.intel.com/en-us/mkl-reference-manual-for-c.

[20] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (September 2009), 455–500. https://doi.org/10.1137/07070111X

[21] Anders Juul Lawaetz, Rasmus Bro, Maja Kamstrup-Nielsen, Ib Jarle Christensen, Lars N Jørgensen, and Hans J Nielsen. 2012. Fluorescence spectroscopy as a potential metabonomic tool for early detection of colorectal cancer. *Metabolomics* 8, 1 (2012), 111–121.

[22] L. Ma and E. Solomonik. 2019. Accelerating alternating least squares for tensor decomposition by pairwise perturbation. arXiv:1811.10573.

[23] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 11.0. https://developer.nvidia.com/cuda-toolkit

[24] P. Paatero. 1997. A weighted non-negative least squares algorithm for three-way 'PARAFAC' factor analysis. *Chemometrics and Intelligent Laboratory Systems* 38, 2 (1997), 223–242. https://doi.org/10.1016/S0169-7439(97)00031-2

[25] A-H. Phan, P. Tichavský, and A. Cichocki. 2013. Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations. *IEEE Transactions on Signal Processing* 61, 19 (2013), 4834–4846. https://doi.org/10.1109/TSP.2013.2269903

[26] M. Rajih, P. Comon, and R. A. Harshman. 2008. Enhanced line search: a novel method to accelerate PARAFAC. *SIAM J. Matrix Anal. Appl.* 30, 3 (2008), 1128–1147. https://doi.org/10.1137/06065577

[27] N.D. Sidiropoulos, Lieven Lathauwer, Xiao Fu, Kejun Huang, Evangelos Papalexakis, and Christos Faloutsos. 2016. Tensor Decomposition for Signal Processing and Machine Learning. *IEEE Transactions on Signal Processing* PP (07 2016). https://doi.org/10.1109/TSP.2017.2690524

[28] M. Sørensen, L. De Lathauwer, P. Comon, S. Icart, and L. Deneire. 2012. Canonical polyadic decomposition with a columnwise orthonormal factor matrix. *SIAM J. Matrix Anal. Appl.* 33, 4 (2012), 1190–1213. https://doi.org/10.1137/110830034

[29] Paul Springer, Jeff R. Hammond, and Paolo Bientinesi. 2017. TTC: A high-performance Compiler for Tensor Transpositions. *ACM Transactions on Mathematical Software (TOMS)* 44, 2 (Aug. 2017), 15:1–15:21. http://arxiv.org/pdf/1603.02297v1

[30] The MathWorks, Inc. 2020. Matlab. http://www.mathworks.com/

[31] G. Tomasi and R. Bro. 2005. PARAFAC and missing values. *Chemometrics and Intelligent Laboratory Systems* 75, 2 (2005), 163–180. https://doi.org/10.1016/j.chemolab.2004.07.003

[32] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 14:1–14:33. http://doi.acm.org/10.1145/2764454

[33] N. Vannieuwenhoven, K. Meerbergen, and R. Vandebril. 2015. Computing the gradient in optimization algorithms for the CP decomposition in constant memory through tensor blocking. *SIAM Journal on Scientific Computing* 37, 3 (2015), C415–C437. https://doi.org/10.1137/14097968X

[34] N. Vervliet and L. De Lathauwer. 2016. A randomized block sampling approach to canonical polyadic decomposition of large-scale tensors. *IEEE Journal of Selected Topics in Signal Processing* 10, 2 (2016), 284–295. https://doi.org/10.1109/JSTSP.2015.2503260

[35] N. Vervliet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer. 2016. Tensorlab 3.0. Available online. https://www.tensorlab.net

[36] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[37] Z. Xianyi, W. Qian, and Z. Yunquan. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 684–691. https://doi.org/10.1109/ICPADS.2012.97