

FIST-HOSVD: Fused In-place Sequentially Truncated Higher Order Singular Value Decomposition

Benjamin Cobb
Georgia Institute of Technology
Atlanta, GA, USA
bcobb33@gatech.edu

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

Hemanth Kolla
Sandia National Laboratories
Livermore, CA, USA
hnkolla@sandia.gov

Ümit V. Çatalyürek
Georgia Institute of Technology
Atlanta, GA, USA
umit@gatech.edu

ABSTRACT

In this paper, several novel methods of improving the memory locality of the Sequentially Truncated Higher Order Singular Value Decomposition (ST-HOSVD) algorithm for computing the Tucker decomposition are presented. We show how the two primary computational kernels of the ST-HOSVD can be fused together into a single kernel to significantly improve memory locality. We then extend matrix tiling techniques to tensors to further improve cache utilization. This block-based approach is then coupled with a novel in-place transpose algorithm to drastically reduce the memory requirements of the algorithm by overwriting the original tensor with the result. Our approach's effectiveness is demonstrated by comparing the multi-threaded performance of our optimized ST-HOSVD algorithm to TuckerMPI, a state-of-the-art ST-HOSVD implementation, in compressing two combustion simulation datasets. We demonstrate up to $\sim 135\times$ reduction in auxiliary memory consumption thereby increasing the problem size that can be computed for a given memory allocation by up to $\sim 3\times$, whilst maintaining comparable runtime performance.

CCS CONCEPTS

• **Computing methodologies** \rightarrow **Shared memory algorithms; Unsupervised learning.**

KEYWORDS

tucker decomposition, kernel fusion, cache blocking, in-place, memory efficient, reduced memory high-water mark, data compression

ACM Reference Format:

Benjamin Cobb, Hemanth Kolla, Eric Phipps, and Ümit V. Çatalyürek. 2022. FIST-HOSVD: Fused In-place Sequentially Truncated Higher Order Singular Value Decomposition. In *Proceedings of Platform for Advanced Scientific Computing (PASC '22)*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/nmnnnnn.nnnnnnn>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PASC '22, Basel, Switzerland,

© 2022 ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

<https://doi.org/10.1145/nmnnnnn.nnnnnnn>

1 INTRODUCTION

The Tucker decomposition [26] is a higher order generalization of the Singular Value Decomposition (SVD) and leverages a multi-dimensional dataset's latent structure to decompose a tensor into a core tensor and series of factor matrices corresponding to each dimension of the dataset. For extending rank truncation to tensors, the Sequentially Truncated Higher Order Singular Value Decomposition (ST-HOSVD) is an efficient method for computing low-rank Tucker tensor decompositions and is effective at large scale data compression [2–4, 11, 12]. Multiplying the core tensor along each dimension by the corresponding factor matrix recovers the original tensor or, in the case of low-rank Tucker, an approximation of the original tensor. Due to this, the Tucker decomposition has found broad applications in fields such as bioinformatics, psychometrics, computer vision, and signal processing that deal with high dimensional data [19, 23, 26, 28]. The ST-HOSVD algorithm is arguably the fastest method to compute the Tucker decomposition due to its unique ability to iteratively truncate the tensor whilst computing a low-rank Tucker approximation, thereby saving on FLOPs [27].

A key constraint limiting the performance, and even feasibility, of computing the Tucker decomposition is memory consumption. The ST-HOSVD requires a considerable amount of temporary memory in addition to the input tensor, thus the available memory severely limits the problem size that can be handled. We build upon the existing ST-HOSVD algorithm by introducing several optimizations that significantly improve its memory locality and overall memory footprint. We focus on optimizing the two kernels that are the computational bottlenecks of the ST-HOSVD algorithm: 1) the Tensor Times Matrix (TTM) kernel which represents multiplication of a tensor with a matrix along a specific dimension, and 2) the Gram matrix computation which involves unfolding the tensor along a dimension and multiplying the resulting matrix with its transpose. Our primary contributions are as follows:

- We demonstrate how the TTM and Gram kernels can be combined within the context of the ST-HOSVD algorithm, i.e., you can compute the Gram matrix of the next dimension whilst computing the TTM of the current dimension.
- We utilize the observation that permuting the columns of a tensor's matricization does not change the corresponding Gram matrix to extend matrix tiling and cache blocking techniques to tensors.

- We couple this cache blocking approach with a novel in-place transpose algorithm to process several cache blocks at a time, overwriting the original tensor with their partial TTM result. This allows us to avoid allocating intermediate TTM results, thereby drastically reducing the memory high-water mark of the algorithm.

In this paper we describe these optimizations to the ST-HOSVD algorithm and implementations of these for advanced HPC architectures. We employ our optimized algorithm for performing compression of two representative combustion datasets:

- (1) Homogeneous Charge Compression Ignition (HCCI) simulation in a spatially 2D domain (resulting in a fourth order tensor)
- (2) Statistically Planar (SP) turbulent flame simulation in a spatially 3D domain (fifth order tensor)

Combustion simulations, like many other scientific applications in the exascale era, generate large volumes of data and compression is necessary to effectively manage and analyze these datasets.

2 RELATED WORK

Due to their usefulness, a wide variety of softwares and libraries have been created to provide implementations of tensor decompositions. In our work, we implement ST-HOSVD within the GenTen [20] package, which provides high performance tensor decomposition capabilities that are portable across emerging CPU and GPU architectures. It leverages Kokkos [7], which is a C++ performance portability library, to enable a single algorithmic implementation to be performant and portable across a wide variety of shared memory parallel computing architectures. Other state-of-the-art studies of ST-HOSVD and related calculations are summarized below.

TuckerMPI [3] is a C++ software package that utilizes MPI to implement ST-HOSVD. TuckerMPI's goal is to provide a framework for parallelizing massive tensor computations across multiple CPU-based nodes. TuckerMPI provides the user with the ability to set the dimensions of the processor grid upon which the tensor is block partitioned. Recently, TuckerMPI implemented a modified variant of the ST-HOSVD that leverages the QR decomposition to increase the numerical stability of the algorithm [17]. TuckerMPI has been shown to effectively compute the ST-HOSVD of datasets up to 6.7 TB, resulting in compression ratios up to 4×10^4 . Our baseline ST-HOSVD implementation was closely based off TuckerMPI. As shown in the experiments section, we benchmark our implementations against TuckerMPI in a shared memory setting.

The Matlab Tensor Toolbox (TTB) [13] is a popular tensor software package in the field of tensor decompositions. It currently utilizes the ST-HOSVD to compute the Tucker decomposition. The TTB is ideal for fast prototyping and experimentation of tensor computations. We used the TTB to prototype and verify the correctness of our approaches.

Choi et al. [6] presents a distributed multi-GPU implementation of the Tucker algorithm. The optimizations for preventing memory movement that they present are in some ways similar to our optimizations. For example, they present a method they refer to as *tensor reuse* that transposes sub-tensor blocks to prepare the tensor for the next iteration of the ST-HOSVD algorithm to avoid

communication. Whilst this shares a similar intuition to our tensor tiling strategy, we stress that their optimizations are different from the ones described in subsequent sections. The key difference is that our approach computes both the TTM and Gram in a single step, whilst Choi et al. compute them separately.

There are several existing approaches in the literature that aim to reduce the memory consumption of computing the Tucker decomposition, particularly in the sparse case due to the intermediate data explosion that arises when instantiating portions of a large, sparse tensor [14, 18]. However, to the best of our knowledge, our approach is the first to do the computation almost entirely in-place by overwriting the original tensor with the result, thereby not requiring a large memory allocation to hold the resulting Tucker decomposition's core tensor. This marks the biggest difference between our *in-place* algorithm and existing methods for computing the Tucker decomposition. For clarity, here we state our definition of in-place within the context of this paper. We define in-place as asymptotically requiring significantly less memory than the original tensor size by overwriting the original tensor with the result. This deviates slightly from the traditional definition of in-place in that it asymptotically requires more than a constant amount of memory to compute the result. We see extending the approaches described in this paper to only requiring a constant amount of memory as an avenue of future work.

3 FORMAL DEFINITION AND NOTATION

The ST-HOSVD algorithm primarily relies upon two tensor kernels, the TTM and Gram kernels. We thus first define them and provide insight into their computation before moving onto the ST-HOSVD algorithm itself. Both the TTM and Gram kernels can be viewed as a series of matrix multiplications by first *matricizing* the tensor. To fully understand tensor matricization it is helpful to first define the concept of tensor *fibers* as follows:

DEFINITION 1. *Given that X is a tensor of order N with dimension sizes: $I_1 \times \dots \times I_N$, the mode- n fibers are the set of vectors resulting from holding all but the n 'th mode constant. In other words, the mode- n fiber $v_{i_1, \dots, i_{n-1}, i_{n+1}, \dots, i_N} = X_{i_1, \dots, i_{n-1}, :, i_{n+1}, \dots, i_N}$, where $:$ is used to denote all elements along that dimension.*

DEFINITION 2. *Given a tensor X , then $X_{(n)}$ denotes the mode- n matricization of X and is a matrix whose columns are the mode- n tensor fibers of X in column major order.*

3.1 Tensor Times Matrix kernel (TTM)

The formal definition of TTM is represented as follows:

DEFINITION 3. *Given that X is a tensor of order N with dimension sizes: $I_1 \times \dots \times I_N$ and \mathcal{U} is a matrix of size $J \times I_n$, then \times_n denotes a TTM along the n 'th dimension (mode) of X defined by*

$$\mathcal{Y} = X \times_n \mathcal{U} \iff \mathcal{Y}_{(n)} = \mathcal{U} X_{(n)} \quad (1)$$

where \mathcal{Y} is the resulting tensor with dimensions: $I_1 \times \dots \times I_{n-1} \times J \times I_{n+1} \times \dots \times I_N$. Defining $I^* = \prod_{r=1}^N I_r$, $I^\otimes = \frac{I^*}{I_n}$, the resulting asymptotic complexity to calculate $\mathcal{Y}_{(n)}$ is $O(JI_n I^\otimes)$.

Figure 1 shows an example of TTM along mode-3 of a 4th order tensor. Our initial implementation of the TTM kernel is heavily

Fused in-place sequentially truncated higher order singular value decomposition

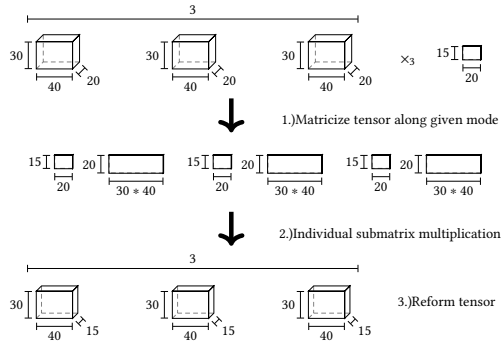


Figure 1: Tensor Times Matrix computation of $30 \times 40 \times 20 \times 3$ tensor with 15×20 matrix along mode-3.

based upon the work of Ballard et al. [3] and Li et al. [16]. Two values that will be helpful in the following discussions are:

$$I_n^> = \prod_{r=n+1}^N I_r, \quad I_n^< = \prod_{r=1}^{n-1} I_r \quad (2)$$

Assuming the entries of \mathcal{X} are stored in column-major order as a contiguous 1D array, i.e., the mode-1 fibers are contiguous in memory, then the matrix resulting from the matricization of \mathcal{X} consists of $I_n^>$ row-major submatrices that are contiguous in memory, each with I_n rows and $I_n^<$ columns for each $1 \leq n \leq N$. Figure 1 provides a simple example of the mode-3 TTM on a 4th order tensor. Keep in mind that the submatrices are contiguous and in row-major order. Taking these arrangements of the submatrices into account, the submatrix multiplications may be done via a call to a general matrix-matrix multiplication (GEMM) kernel. This grants access to the multitude of highly optimized pre-existing linear algebra libraries, such as BLAS [1]. Furthermore, due to the fact that the submatrix multiplications involve separate contiguous sections of the input and output tensors, these GEMM calls can be executed in parallel. Our baseline ST-HOSVD utilizes this approach.

3.2 Gram

The formal definition of the Gram kernel is as follows:

DEFINITION 4. Given that \mathcal{X} and $\mathcal{X}_{(n)} \in \mathbb{R}^{I_n \times I^{\otimes}}$ is as previously defined, then the mode- n Gram matrix, $S \in \mathbb{R}^{I_n \times I_n}$ is given by: $S = \mathcal{X}_{(n)} \mathcal{X}_{(n)}^T$ with asymptotic complexity $O(I_n^2 I^{\otimes})$.

Based upon this, depending on the values of J and I_n relative to each other, the TTM and Gram matrix kernel require asymptotically comparable amounts of work. Similar to the TTM kernel, the Gram kernel can be viewed as a series of row major matrix-matrix multiplications, the results of which are reduced together to form the final symmetric Gram matrix. Figure 2 shows an example of computing the Gram matrix along mode- n of a tensor.

3.3 ST-HOSVD

Algorithm 1 shows the ST-HOSVD pseudocode and Figure 3 shows a corresponding example of the Tucker decomposition. As discussed previously, the computational cost of the ST-HOSVD algorithm is

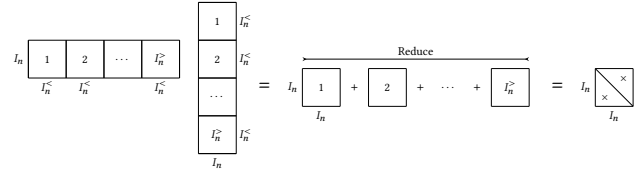


Figure 2: Gram matrix computation.

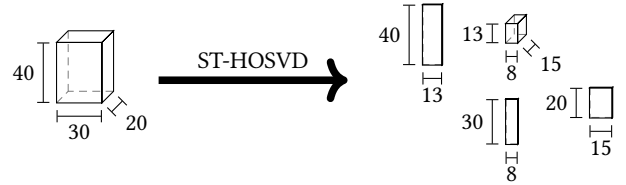


Figure 3: Tucker decomposition example of 3rd order $40 \times 30 \times 20$ tensor. The small $13 \times 8 \times 15$ tensor in the center of the matrices is referred to as the *core tensor*. The surrounding matrices are generally referred to as *factor matrices*.

dominated by the Gram matrix kernel (Line 3) and the TTM kernel (Line 7) [3]. Thus, optimizing these two kernels is integral to optimizing the ST-HOSVD algorithm as a whole. Similarly, the memory consumption of the ST-HOSVD is dominated by the intermediate TTM results. In the worst case the ST-HOSVD must allocate $3 \times$ the tensor size in memory. This occurs when there is no truncation along the first two dimensions, generally due to either low error-tolerance or high tensor rank. The memory consumption of the ST-HOSVD is thus: $O(I_{max}^2 + \prod_{r=1}^N I_r)$, where I_{max} is the size of the largest dimension. The I_{max}^2 term is the memory required to compute the largest Gram matrix and store the largest factor matrix. The $\prod_{r=1}^N I_r$ term is the memory necessary to store the core tensor and TTM result, which is the core tensor of the next iteration. In practice this term almost always constitutes the bulk of the ST-HOSVD's memory consumption as seen in the experimental results section. The goal of this paper is to drop the $\prod_{r=1}^N I_r$ term from the ST-HOSVD's memory consumption O-asymptotic bound.

Algorithm 1: ST-HOSVD

Data: Tensor \mathcal{X} , accuracy bound ϵ

Result: Tensor core \mathcal{G} , factor matrices \mathcal{F}

- 1 $\mathcal{G} \leftarrow \mathcal{X}$; /* Initialize \mathcal{G} */
 - 2 **for** $n = 1 : N$ **do**
 - 3 $S \leftarrow \mathcal{G}_n \mathcal{G}_n^T$; /* Gram matrix */
 - 4 /* λ eigenvalues in descending order, V corresponding eigenvectors */
 - 5 $[\lambda, V] \leftarrow \text{eig}(S)$
 - 6 $U_n \leftarrow V(:, 1 : R_n)$; /* R_n is smallest value that satisfies ϵ */
 - 7 $\mathcal{G} \leftarrow \mathcal{G} \times_n U_n^T$; /* TTM */
 - 8 $\mathcal{F} \leftarrow U_1 \dots U_N$
 - 9 **return** \mathcal{G}, \mathcal{F}
-

4 OPTIMIZATIONS

4.1 Kernel Fusion

The first optimization that we employed in improving the ST-HOSVD algorithm relies upon the observation that each TTM submatrix multiplication forms a single, contiguous row of the next dimension's matricized tensor used to compute the Gram matrix. Written formally:

DEFINITION 5. Given $\mathcal{X}_{(n)}$, divide $\mathcal{X}_{(n)}$'s submatrices into $\frac{I_n}{I_{n+1}}$ groups, each with I_{n+1} submatrices. Denote the resulting partition as s_n . Let $s_n[i, j]$ denote the i th submatrix in the j th submatrix group of $\mathcal{X}_{(n)}$. Accessed linearly, $s_n[i, j]$ is the $(i + (j - 1) * I_{n+1})$ 'th submatrix of $\mathcal{X}_{(n)}$. Additionally, let $\text{vec}(s[i, j]) \in \mathbb{R}^{I_{n+1} * (I_n / I_{n+1})}$ denote the row vector containing the entries of $s_n[i, j]$, $\mathcal{X}_{(n)}[j]$ be the j 'th submatrix of $\mathcal{X}_{(n)}$, and $\mathcal{X}_{(n+1)}[j](i, :)$ denote all the entries in the i 'th row of the j 'th submatrix of $\mathcal{X}_{(n+1)}$. Then $\mathcal{X}_{(n+1)}[j](i, :) = \text{vec}(s_n[i, j])$.

Figure 4 gives an illustration of this. The benefit of viewing the TTM and Gram in this manner is that it becomes readily apparent that after computing the mode- n TTM we may immediately compute a portion of the mode- $(n + 1)$ Gram matrix. To do so, compute partial TTM with $s_n[:, j]$, where $s_n[:, j]$ is all the matrices in the j 'th submatrix group of $\mathcal{X}_{(n)}$. Logically concatenate vec of resulting submatrices to form $\mathcal{X}_{(n+1)}[j]$. Then compute $(\mathcal{X}_{(n+1)}[j])(\mathcal{X}_{(n+1)}[j])^T$.

For certain problem sizes, this entails that the entries involved in both computations may remain in cache for the entire computation. Regardless of dimension size, the TTM and Gram matrix may thus be fused together into a single kernel. It is well known that kernel fusion significantly improves the memory locality of computation. This is especially beneficial in the case of GPU applications [8, 24].

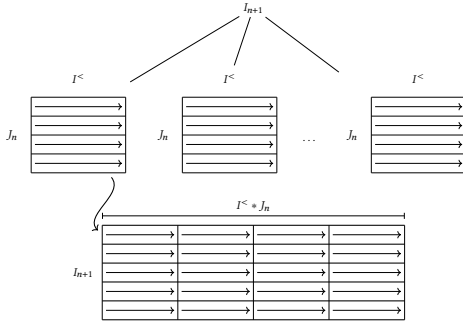


Figure 4: Example of fusing the result of I_{n+1} mode- I_n TTM submatrices, each $\in J_n \times I_n^<$ where J_n is the number of rows of the input matrix, into a single submatrix used to compute the I_{n+1} gram matrix.

Once the TTM and Gram kernels have been fused together, the ST-HOSVD algorithm may be modified to utilize the fused kernel. Essentially the first Gram and last TTM remain the same, whilst the remaining kernels can be fused. This modification to the ST-HOSVD can be seen in Algorithm 3.

As is, this process restricts us to processing to dimensions sequentially in-order. However, this restriction can be removed by the data packing technique described in the subsequent section.

4.2 Tensor Tiling

Matrix tiling, also referred to as matrix blocking, is a common method of improving cache utilization in matrix multiplication [1, 9, 15]. Due to the fact that we view both the TTM and Gram kernels as a series of matrix multiplications it is logical to extend matrix tiling techniques to tensors.

In the traditional approach, the tensor fibers of each dimension except the first are strided in memory. Thus, in order to explicitly form a given mode's fiber, a series of strided accesses is required. For architectural reasons, strided memory accesses on single elements are significantly slower than contiguous accesses and should generally be avoided. The advantage of the approach presented in Ballard et al. [3] is that it avoids strided memory accesses by only dealing with the contiguous row-major submatrices in the tensor's matricization at each iteration. The disadvantage of this approach is that for later dimensions the submatrices tend to be *skinny* with one dimension being much larger than the other. Many GEMM implementations are not primarily optimized for this case [21]. Our approach aims to alleviate this problem by packing the tensor entries into cache friendly blocks at every iteration of the ST-HOSVD algorithm. This causes the fibers corresponding to the dimension of the current iteration to be contiguous in memory. As a result, the layout of the tensor evolves in memory over the course of the computation. In conjunction with the kernel fusion described in the previous section, this leads to significantly improved memory locality, the benefits of which are shown in the experimental results section.

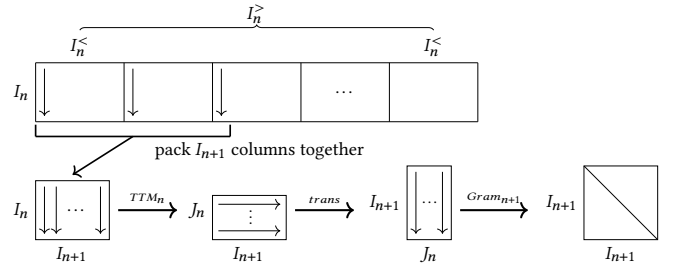


Figure 5: Fusing the TTM and Gram steps of the ST-HOSVD algorithm. Note that the ordering of the tensor entries change at every iteration to prepare it for the next iteration.

We start by assuming that the tensor begins as column-major (first dimension contiguous) and that the tensor dimensions are processed sequentially in order starting at 1. We can process the dimensions out of order, but it significantly complicates the explanation, so we leave it for future work. For each dimension of the tensor, the current mode's fibers are contiguous and packed into blocks with fibers that are $I_n^<$ offset in memory. In other words, let $\mathcal{X}_{(n)}$ be as previously defined in Definition 2 and s_n as in Definition 5. Let $V_j \in \mathbb{R}^{I_n \times (I_n^< * I_{n+1})}$ be the matrix resulting from concatenating $s_n[:, j]$ together. Then $V_j(:, i : (I_n^< * (I_{n+1} - 1) + i))$ columns of V_j are packed together into a contiguous cache block $\in \mathbb{R}^{I_n \times I_{n+1}}$, where $V_j(:, i : (I_n^< * (I_{n+1} - 1) + i))$ denotes each column vector offset by a stride of $I_n^<$ from the i 'th column of V_j . This process is repeated for each i and j , subject to $1 \leq i \leq I_{n+1}$ and $1 \leq j \leq \frac{I_n}{I_{n+1}}$.

Each block then undergoes fused multiplication, transposing the result by saving it row-major in memory to make the next dimension's fibers contiguous in memory. Figure 5 gives a visual representation of this process. This is repeated until the last dimension, wherein a single TTM with the truncated eigenvectors of the last Gram matrix is performed along the last dimension to complete the algorithm. This yields the same Tucker decomposition as the traditional ST-HOSVD algorithm with the caveat that the core entries are rearranged in memory depending on the order in which the dimensions were processed. This slightly complicates the process by which the original tensor is recovered, as the entries must be unpacked.

This process can also be viewed as transposing the submatrices formed from the first two dimensions of a special 3'rd order reshaped tensor as seen later on in Figure 6 (a). Here $\mathcal{X} \in I_1 \times \dots \times I_N$ is reshaped such that $I_n^<$ contiguous mode- n fibers form the first dimension, the second dimension is I_{n+1} , and the third dimension is $\frac{I_n^>}{I_{n+1}}$. In the following algorithms, we denote this as: \mathcal{X}_n . Here each I_n rows of the submatrices formed from the first two dimensions of \mathcal{X}_n constitute a block as described in the previous section. Each block will be multiplied by the same TTM matrix of size $J_n \times I_n$ (first J_n eigenvectors of mode- n Gram matrix). The resulting row-major block of size $J_n \times I_{n+1}$ will then immediately be multiplied by its transpose on the left to form a symmetric Gram matrix of size $(I_{n+1} \times I_{n+1})$. These Gram matrices are then reduced together to form the mode- $(n+1)$ Gram matrix. The generalized pseudocode of this process is shown in Algorithm 2. We refer to the algorithm that uses this kernel as the Fused + packed ST-HOSVD (FaST-HOSVD), as can be seen in Algorithm 3.

In the current implementation, each block is $(I_n \times I_{n+1})$ for mode- n , but could be $(I_n \times I_i)$, where I_i is an arbitrary dimension of the tensor. Viewing the tensor in this manner helps relate this approach to existing matrix blocking techniques and provides intuition as to why it works. This reshaping is key to facilitating the in-place optimization described in the next section.

Algorithm 2: Fused + Packed Kernel

Data: Tensor \mathcal{X} , mode n , Factor matrix $\mathcal{U} \in I_n \times R_n$
Result: Core tensor \mathcal{Y} , Gram matrix $\mathcal{S} \in \mathbb{R}^{I_{n+1} \times I_{n+1}}$

- 1 $[I^<, I^>] \leftarrow \mathcal{X}$; /* calculate $I^<$ and $I^>$ */
- 2 $\mathcal{X}'_n \in \mathbb{R}^{I_n \times I^< \times I^>} \leftarrow \mathcal{X}$; /* reshape, no data copy required */
- 3 $\mathcal{X}'_{n+1} \in \mathbb{R}^{I_{n+1} \times (I^< * R_n * I^> / I_{n+1})} \leftarrow \mathcal{X}'_n$
- 4 **for** $i = 1 : (I^> / I_{n+1})$ **do**
- 5 **for** $j = 1 : I^<$ **do**
- 6 $blk \in \mathbb{R}^{I_n \times I_{n+1}} \leftarrow \mathcal{X}'_n(:, j, (i * I_{n+1}) : ((i + 1) * I_{n+1}))$
- 7 $ttm_{i,j} = blk^T \mathcal{U}$
- 8 $\mathcal{S} += ttm_{i,j} ttm_{i,j}^T$
- 9 $\mathcal{Y} \leftarrow \mathcal{X}'_{n+1}$
- 10 **return** \mathcal{Y}, \mathcal{S}

4.3 In-Place

As previously noted, in the worst case the ST-HOSVD algorithm requires $3 \times$ the tensor size in memory to hold the original tensor and intermediate TTM results. This occurs when the computation does

Algorithm 3: FaST-HOSVD

Data: Tensor \mathcal{X} , accuracy bound ϵ
Result: Tensor core \mathcal{G} , factor matrices \mathcal{F}

- 1 $\mathcal{G} \leftarrow \mathcal{X}$; /* Initialize \mathcal{G} */
- 2 $\mathcal{S}_1 \leftarrow \mathcal{G}_1 \mathcal{G}_1^T$; /* First Gram matrix */
- 3 /* λ eigenvalues in descending order, V corresponding eigenvectors */
- 4 $[\lambda, V] \leftarrow eig(\mathcal{S}_1)$
- 5 $U_1 \leftarrow V(:, 1 : R_1)$; /* R_1 is smallest value that satisfies ϵ */
- 6 **for** $n = 1 : N - 1$ **do**
- 7 $[\mathcal{G}_{n+1}, \mathcal{S}_{n+1}] \leftarrow Fused_Packed_Kernel(\mathcal{G}_n, U_n, n)$
- 8 $[\lambda, V] \leftarrow eig(\mathcal{S}_{n+1})$
- 9 $U_{n+1} \leftarrow V(:, 1 : R_{n+1})$
- 10 $\mathcal{G} \leftarrow \mathcal{G} \times_n U_N^T$; /* Last TTM */
- 11 $\mathcal{F} \leftarrow U_1 \dots U_N$
- 12 **return** \mathcal{G}, \mathcal{F}

not benefit from truncation in earlier dimensions, such as when the error-tolerance is low or the tensor is of high rank. In these instances, this memory requirement can be prohibitive in computing the Tucker decomposition of tensors that are larger than $\frac{1}{3}$ of main memory (RAM). This is born out in the experimental results section wherein when the tensor is $> \frac{1}{3}$ the size of RAM the ST-HOSVD runs out of memory, either causing the node to crash or the computation to thrash between Disk and RAM. The former case results in the computation not being able to complete, whilst the latter case results in extreme performance degradation. Here we show how the cache blocking approach described in the previous sections can be leveraged to drastically decrease the memory requirements of the algorithm. The experimental results section demonstrates that this approach generally requires a small fraction of the tensor size in memory to complete the computation, whilst still benefiting from reduced runtime due to the improved memory locality. The key intuition to this approach is that the cache blocks can be prepared by transposing I_{n+1} of the mode- n submatrices together. This can be seen in Figure 6 (a) wherein each $I_n^< * I_n \times I_{n+1}$ submatrix can be viewed as $I_n^< \times I_{n+1}$ with elements of size I_n . From here it becomes apparent that each of the $I_n^> / I_{n+1}$ submatrices can be prepared into $I_n^<$ blocks, each of size $I_n \times I_{n+1}$, by transposing the submatrix.

From here we can leverage existing in-place matrix transpose algorithms [10][5][25] to prepare the blocks in-place. These blocks are then iteratively copied into the user defined memory allocation and undergo fused TTM/Gram multiplication, overwriting the original tensor with the result and accumulating into the corresponding Gram matrix. In this manner we can do the computation in-place requiring only $I_n \times I_{n+1}$ memory. However it is worth noting that enough memory to hold multiple blocks is preferred for high-performance, especially for higher thread counts.

The astute reader may remark that this potentially requires significant data movement. We note that the data movement overhead is asymptotic with the size of the tensor. As shown in the experimental results section, this proves to be negligible relative to the TTM and Gram submatrix multiplications, for compute bound problem sizes. This is due to the fact that matrix multiplication is a $O(N^3)$ operation, whereas matrix transpose is a $O(N^2)$ operation. Thus the more compute bound the submatrix multiplications are, the less of an impact data movement has on the runtime of the computation.

This relies upon the arithmetic intensity, $\frac{FLOPs}{Bytes}$, of each sub-matrix multiplication. When either a subsequent tensor dimension is extremely small or the TTM input matrix has a small number of rows (approximately < 8 depending on number of bytes in a single value), either due to dimension size or truncation, then the computation has the potential to be memory bound. In these cases the data movement has the potential to introduce significant overhead into the computation. However, in the latter case when the input TTM matrix is small, the corresponding TTM is unlikely to contribute much to the overall runtime of the algorithm. In the instance when the dimension is extremely small, multiple blocks can be grouped together to make each TTM multiplication more compute bound. In either instance we have yet to encounter applications that yield such a degenerate case. Furthermore, the fused in-place approach is compatible with the existing unfused approach, meaning that it is possible to switch between them depending on the size of a given tensor dimension. We see developing heuristics to decide when to apply traditional versus proposed approaches as an avenue of future work.

Algorithm 4: Interleaved In-Place Transpose (IIPT)

```

Data:  $A \in \mathbb{R}^{m \times n}$ , interleaving factor  $\alpha | m$ 
Result:  $A$  overwritten by  $A_{inter}^T \in \mathbb{R}^{n \times \frac{m}{\alpha}}$ 
1  $z = \frac{m}{\alpha}$ 
   /* each entry of  $A_{inter}$  consists of  $\alpha$  contiguous entries of  $A$  */
2  $A_{inter} \in \mathbb{R}^{z \times n} \leftarrow A$ 
3  $q = n * z - 1$ 
4 for  $i = 0 : q$  do
5    $k = (i * n) \% q$ 
6    $v = A_{inter}[i]$ 
7   while  $k > i$  do
8      $k = (i * n) \% q$ 
9   if  $k = i \ \&\& \ i! = 0$  then
10     $k = (i * n) \% q$ 
11     $A_{inter}[i] = A_{inter}[k]$ 
12    while  $k > i$  do
13       $A_{inter}[k] = A_{inter}[(k * n) \% q]$ 
14       $k = (k * n) \% q$ 
15     $A_{inter}[(k * z) \% q] = v$ 

```

There are several known algorithms for in-place matrix transpose [10][5] [25]. In this work we originally selected the traditional in-place transpose algorithm based upon cycle-following (IPT) due to its ease of implementation and $O(N^2)$ complexity.

In the literature it is well known that despite accessing the matrix elements fewer times than other in-place transpose algorithms (some divide and conquer in-place transpose algorithms require accessing each element $O(\log n)$ times) this algorithm suffers from poor memory locality. This is due to its almost pseudorandom access of the matrix elements. To alleviate this problem, we developed a novel improvement to the cycle-following based algorithm which we refer to as Interleaved In-Place Transpose (IIPT), shown in Algorithm 4.

The key improvement of Algorithm 4 over the traditional cycle-following based in-place transpose algorithm is that Algorithm 4 moves the matrix entries around in contiguous blocks to improve

memory locality. These contiguous blocks can be viewed as interleaved entries of the matrix transpose. This is represented in Algorithm 4 by A_{inter} , defined on Line 2. Note that $A_{inter} \in \mathbb{R}^{\frac{m}{\alpha} \times n}$ because each entry of A_{inter} consists of α entries of A . In our experiments we observed that this significantly alleviated the memory locality issues of the traditional cycle-following based in-place transpose algorithm, whilst retaining $O(N^2)$ complexity. Additionally, due to each cycle being independent of all other cycles, this algorithm was straightforward to parallelize and adapt to changing tensor layouts.

We intend to provide a more in-depth analysis of this approach compared to existing in-place matrix transpose algorithms in a future work. The contiguous interleaved elements can be *deinterleaved* in memory to complete the transpose under normal circumstances. This is not shown in Algorithm 4 because in Algorithm 5 the interleaved columns of \mathcal{X}_n are deinterleaved by the packing phase on Line 11.

As previously mentioned, we use Algorithm 4 to prepare the blocks in-place as part of the FIST-HOSVD, shown in Algorithm 5. The interleaving factor, α , is determined based upon the available memory and such that α is a factor of $I_n^<$. α being a factor of $I_n^<$ prevents the hassle of dealing with leftover rows which would result in partial blocks. We want to maximize α within these constraints to maximize the contiguousness of the memory accesses. To this end we have developed a heuristic that consists of computing the prime factorization of $I_n^<$, then determining α as the product $I_n^<$'s prime factors subject to the aforementioned constraints via a simple monte-carlo method. Note that the in-place transpose is only applied when the I_{n+1} submatrices don't fit into memory. The interleaved columns, each of size I_n , are deinterleaved by the packing phase of the FIST-HOSVD. This is identical to the packing phase of the FaST-HOSVD with the caveat that each of the submatrices are of size $I_n \times \alpha$.

After the blocks have been prepared via the in-place transpose, $\frac{\beta}{I_n * I_{n+1}}$ blocks are copied into memory. Each of these blocks is then dispatched in parallel, computing the fused TTM/Gram multiplication, with the TTM result overwriting the original tensor and the Gram result being accumulated into the mode- $(n+1)$ Gram matrix. A visual description of this process can be seen in Figure 6.

Algorithm 6 shows this fused in-place kernel within the context of the FIST-HOSVD. The FIST-HOSVD begins with the same Gram matrix computation along mode-1 as the ST-HOSVD, followed by the fused in-place kernel along the internal dimensions, and concludes with an in-place variant of the TTM kernel. Note that at every iteration of the algorithm, the tensor fibers along the current dimension are contiguous in memory. Thus, the last TTM is on a tensor whose fibers corresponding to the last dimension of the tensor are contiguous. Therefore, the last in-place TTM call consists of iteratively copying columns of the tensor into memory and computing the partial TTM with \mathcal{U}_N^T , overwriting the corresponding memory region of the tensor with the result.

In the worst case the FIST-HOSVD only requires enough memory to hold the largest Gram matrix, which as previously mentioned is I_{max}^2 . The memory consumption of the FIST-HOSVD is thus: $O(I_{max}^2)$. We have thus reduced the memory consumption of computing the Tucker Decomposition from the ST-HOSVD's asymptotic

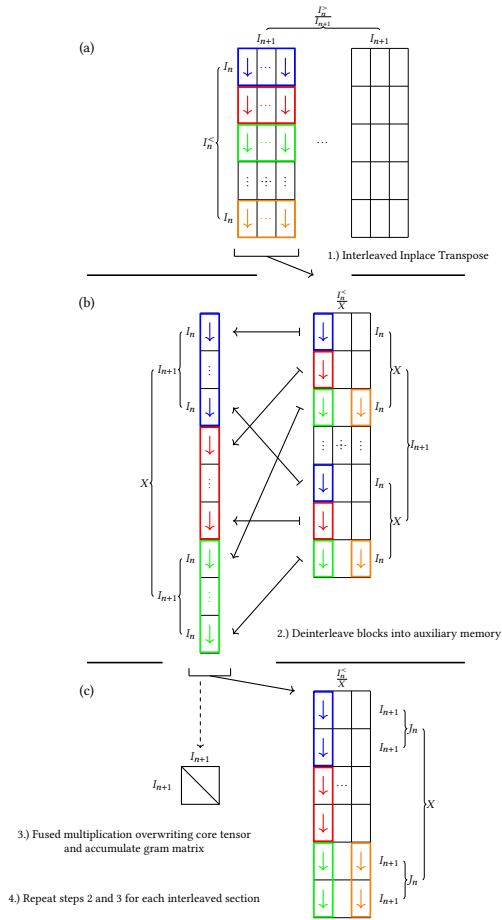


Figure 6: Example of Fused In-place kernel packing blocks from one submatrix of reshaped tensor, overwriting it with partial TTM result, and accumulating partial Gram matrix result.

bound of $O(I_{max}^2 + \prod_{r=1}^N I_r)$ to the FIST-HOSVD's asymptotic bound of $O(I_{max}^2)$.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

Benchmark results were generated on a single 28 Intel E5-2683v3 2 GHz core node. Each core has access to a 32KB L1, 256KB L2 and 35840KB L3 cache. Each node has 256 GB of main memory. This is particularly important as we aim to show that the FIST-HOSVD is able to process larger problem sizes within this given 256 GB allocation. The experimental results verify that when an implementation attempts to allocate more than the 256 GB available on the node, the run crashes and is unable to complete. In the tables below we designate this by “—”.

To ensure a fair comparison and to encourage reproducibility, all benchmarked implementations were linked to the same MKL library, run on the same (single) node, and used the same sets of environment variables. We ran TuckerMPI with multiple processes on

Algorithm 5: Fused In-place kernel

Data: Tensor \mathcal{X} , mode n , Factor matrix $\mathcal{U} \in I_n \times R_n$, number of entries that can fit in user defined auxiliary memory limit β

Result: \mathcal{X} overwritten with core tensor, Gram matrix \mathcal{S}

```

1  $[I_n^<, I_n^>] \leftarrow \mathcal{X}$ 
  /*  $\mathcal{X}_n \leftarrow \mathcal{X}$  after  $n$  fused iterations,  $n$ 'th mode is contiguous */
2  $\mathcal{X}_n \in \mathbb{R}^{I_n^< \times I_n \times I_{n+1} \times I_n^>} \leftarrow \mathcal{X}$ 
3  $\alpha \leftarrow [I_n^<, \beta]$ ; /* compute  $\alpha$  such that  $\alpha | I_n^<$  and  $\alpha < \frac{\beta}{I_n \times I_{n+1}}$  */
4 if  $\alpha < I_n^<$  then
5   for  $i = 1 : \frac{I_n^>}{I_{n+1}}$  do
6      $\mathcal{X}_n^{inter}(:, :, i) \in \mathbb{R}^{(I_n \times I_{n+1} \times \alpha) \times \frac{I_n^<}{\alpha}} \leftarrow \Pi PT(\mathcal{X}_n(:, :, i),$ 
7        $I_n * \alpha)$ ; /*  $\Pi PT$  each  $I_n^< \times I_{n+1}$  submatrix */
8      $v = \frac{\beta}{I_n \times I_{n+1} \times \alpha}$ ; /* num_sections_per_iteration */
9      $aux \in \mathbb{R}^{I_n \times I_{n+1} \times \alpha \times n}$ ; /*  $aux \leq \beta$  */
10     $\mathcal{X}'_n \in \mathbb{R}^{I_n \times \alpha \times I_{n+1} \times \frac{I_n^<}{\alpha}} \leftarrow \mathcal{X}_n^{inter}$ ; /* reshape  $\mathcal{X}$  to simplify notation */
11    for  $i = 1 : v$  do
12       $aux(:, :, :, j) \leftarrow \mathcal{X}'_n(:, :, :, i * v + j)$ ; /* packing phase */
13      /* fused block multiplication */
14      for  $j = 1 : v$  do
15         $\mathcal{X}'_n(:, :, :, i * v + j) = aux(:, :, :, j)^T \mathcal{U}$ ; /* overwrite  $\mathcal{X}$  */
16         $\mathcal{S} += \mathcal{X}'_n(:, :, :, i * v + j) \mathcal{X}'_n(:, :, :, i * v + j)^T$ ;

```

Algorithm 6: FIST-HOSVD

Data: Tensor \mathcal{X} , auxiliary memory limit in β

Result: \mathcal{X} overwritten with core tensor, Factor matrices \mathcal{F}

```

1  $S_0 \leftarrow \mathcal{X}_1 \mathcal{X}_1^T$ ; /* First Gram matrix */
2 for  $n = 1 : N - 1$  do
3   /*  $\lambda$  eigenvalues in descending order,  $V$  corresponding eigenvectors */
4    $[\lambda, V] \leftarrow eig(S_n)$ 
5    $U_n \leftarrow V(:, 1 : R_n)$ ; /*  $R_n$ , smallest value that satisfies  $\epsilon$  */
6    $S_{n+1} \leftarrow \text{Fused\_Inplace\_kernel}(\mathcal{G}, U_n, \beta)$ 
7    $[\lambda, V] \leftarrow eig(S_N)$ 
8    $U_N \leftarrow V(:, 1 : R_N)$ 
9    $\mathcal{X} \leftarrow \mathcal{X} \times_N U_N^T$ ; /* Last Inplace TTM */
10   $\mathcal{F} \leftarrow U_1 \dots U_N$ 
11  return  $\mathcal{F}$ 

```

a single node, each with a single thread. We found that this yielded significantly better performance for TuckerMPI than one process with multiple threads. To ensure that each process used only a single thread, we set both OMP_NUM_THREADS and MKL_NUM_THREADS to 1 when running TuckerMPI. In addition to this, we set OMP_PLACES to cores, MKL_NUM_THREADS to 1, and OMP_PROC_BIND to c1ose for all experiments. TuckerMPI requires the user to select a processor grid layout as part of its input. The performance of a given processor grid is very problem dependent and determining which grid to use for an arbitrary problem size is still an open research topic [3, 22]. To this end we experimented with several processor grids and heuristics to determine which yielded the best performance. We eventually settled on a simple heuristic which consisted of computing the prime factorization of the number of available processors, sorting them in ascending order, and then assigning

Table 1: Random tensor runtime (in seconds).
1 slice: $64 \times 64 \times 64 \times 64 \times 64$

ϵ	Slices:	1	4	16	28
1e-09	TuckerMPI	13.4	66.5	—	—
	ST-HOSVD	4.4	142.1	—	—
	FaST-HOSVD	4.4	70.0	—	—
	FIST-HOSVD	5.5	32.3	107.6	219.3
1e-05	TuckerMPI	13.4	66.4	—	—
	ST-HOSVD	4.4	143.8	—	—
	FaST-HOSVD	4.3	74.8	—	—
	FIST-HOSVD	5.5	32.1	107.3	219.8
1e-03	TuckerMPI	13.5	65.6	—	—
	ST-HOSVD	4.4	143.3	—	—
	FaST-HOSVD	4.4	70.8	—	—
	FIST-HOSVD	5.6	32.7	107.4	219.6

them to each dimension in descending order starting with the second to last dimension. Here we assume the last tensor dimension is time and define the preceding dimensions as together representing a single *timestep* or *timeslice*. This heuristic performed well with all the process counts and problem sizes we experimented with.

To demonstrate the effectiveness of our approach, here we present benchmark results on three different datasets, over four different number of timeslices, for four different error-tolerances (denoted ϵ in tables), for two different metrics (runtime and memory consumption). Additionally, we benchmark four different algorithm implementations: a baseline of the traditional ST-HOSVD within the GenTen framework, TuckerMPI’s implementation of the ST-HOSVD, the FaST-HOSVD algorithm utilizing both the kernel fusion and tensor tiling optimizations previously described in Algorithm 3, and the FIST-HOSVD as described in Algorithm 6. This gives a good overview of the problem sizes each implementation can handle, as well as the impact that truncation plays in this. Due to the exponential growth of benchmark configurations, we show only the results from highest thread count of 28 in the memory consumption and runtime tables.

The three datasets used in our experiments are: (1) a tensor with randomly generated entries (Random), (2) HCCI combustion dataset, and (3) SP combustion dataset. The HCCI dataset, a 4-th order tensor, is from a simulation of turbulent autoignition over a 2D spatial domain. The mesh with 672×672 grid points, containing 33 solution variables at each grid point, constitute the first 3 modes of the tensor, with time being the last mode. The SP data set is a 5-th order tensor from a simulation over a 3D spatial domain; the first 3 modes are the $500 \times 500 \times 500$ spatial grid, the 4-th mode is the 11 solution variables at each grid point, and time is the last mode. Both the HCCI and SP datasets have large dimensionality along the first modes and are expected to compress more along these modes. The shape of these datasets also contrasts with the equi-sized Random data set, with more work expected along the initial modes than the later modes.

5.2 Runtime Results

The reduced memory consumption discussed in the previous section has the beneficial side effect of improved memory locality. It is well known that improved memory access patterns tend to yield

Table 2: HCCI tensor runtime (in seconds).
1 slice: $672 \times 672 \times 33$

ϵ	Slices:	176	326	476	626
1e-09	TuckerMPI	35.4	71.4	104.9	—
	ST-HOSVD	18.0	37.4	58.6	—
	FaST-HOSVD	23.9	47.8	76.0	125.2
	FIST-HOSVD	25.0	51.2	77.6	105.7
1e-05	TuckerMPI	20.0	43.7	63.8	84.2
	ST-HOSVD	9.6	23.3	35.6	47.7
	FaST-HOSVD	12.0	31.3	44.8	66.7
	FIST-HOSVD	13.5	31.5	45.7	60.2
1e-03	TuckerMPI	11.4	27.1	38.3	49.1
	ST-HOSVD	5.7	12.6	19.5	25.6
	FaST-HOSVD	6.9	16.0	24.3	33.5
	FIST-HOSVD	7.0	16.5	24.7	31.6

Table 3: SP tensor runtime (in seconds).
1 slice: $500 \times 500 \times 500 \times 11$

ϵ	Slices:	5	10	15	20
1e-09	TuckerMPI	34.0	—	—	—
	ST-HOSVD	24.9	38.6	—	—
	FaST-HOSVD	35.1	54.2	—	—
	FIST-HOSVD	25.6	49.3	72.7	92.6
1e-05	TuckerMPI	12.9	25.4	38.1	—
	ST-HOSVD	10.1	19.2	28.9	—
	FaST-HOSVD	12.2	22.8	35.5	—
	FIST-HOSVD	12.5	24.3	36.4	48.2
1e-03	TuckerMPI	8.4	16.6	24.8	33.3
	ST-HOSVD	7.0	13.9	21.36	27.6
	FaST-HOSVD	9.5	18.9	29.0	37.9
	FIST-HOSVD	9.9	19.4	28.9	38.4

better utilization of the memory hierarchy, which in turn leads to reduced runtime. This balances out the memory movement overhead incurred to reduce the memory consumption of the FIST-HOSVD.

A significant factor is the degree of truncation along the earlier dimensions. The truncation for several benchmarked problem sizes can be seen in Table 4. When the error-tolerance is low and/or the tensor has high rank, the ST-HOSVD does not benefit significantly from truncation along the first couple of dimensions. This results in the later dimensions contributing a significant portion of the computation. For the traditional ST-HOSVD, this results in long-skinny matrix multiplications, which many BLAS installations are not well optimized for [21]. The kernel fusion and tensor tiling optimizations avoid this scenario by packing the tensor entries into squarer cache blocks, which are more amenable to existing BLAS implementations. The result of this can be seen in Table 1 and Figure 7a, where the algorithms utilizing the kernel fusion and tensor tiling optimizations outperform the ST-HOSVD implementations for all error-tolerances. However, when the error-tolerance is high enough and/or the tensor is low enough rank that the computation benefits from high truncation along the earlier dimensions, then the later dimensions tend to not make up as significant a portion of the computation. In this case, the unfused and unpacked TTM and Gram kernels which do not incur the corresponding overhead

**Table 4: Sample core sizes for all error tolerances.
* only FIST-HOSVD completed**

ϵ	Dataset	Slices	Resulting Core
1e-09	Random	4	$64 \times 64 \times 64 \times 64 \times 64 \times 4$
	HCCI	326	$631 \times 610 \times 31 \times 326$
	SP *	20	$187 \times 288 \times 278 \times 9 \times 20$
1e-05	Random	4	$64 \times 64 \times 64 \times 64 \times 64 \times 4$
	HCCI	326	$433 \times 410 \times 33 \times 234$
	SP *	20	$79 \times 116 \times 117 \times 7 \times 5$
1e-03	Random	4	$64 \times 64 \times 64 \times 64 \times 64 \times 4$
	HCCI	326	$232 \times 217 \times 29 \times 81$
	SP	20	$27 \times 48 \times 48 \times 2 \times 3$

**Table 5: Random tensor memory consumption (in GB).
1 slice is ~ 8 GB**

ϵ	Slices:	1	4	16	28
1e-09	ST-HOSVD	24.2	96.2	—	—
	FaST-HOSVD	16.2	64.1	—	—
	FIST-HOSVD	1.2	1.6	1.9	1.2
1e-05	ST-HOSVD	24.2	96.2	—	—
	FaST-HOSVD	16.2	64.1	—	—
	FIST-HOSVD	1.2	1.6	1.9	1.2
1e-03	ST-HOSVD	24.2	96.2	—	—
	FaST-HOSVD	16.2	64.1	—	—
	FIST-HOSVD	1.2	1.6	1.9	1.2

**Table 6: HCCI tensor memory consumption (in GB).
1 slice is ~ 0.12 GB**

ϵ	Slices:	176	326	476	626
1e-09	ST-HOSVD	49.0	94.0	135.6	—
	FaST-HOSVD	34.2	65.2	94.1	123.0
	FIST-HOSVD	1.1	1.1	1.1	1.1
1e-05	ST-HOSVD	18.4	47.3	65.8	82.5
	FaST-HOSVD	15.2	37.9	54.1	70.1
	FIST-HOSVD	1.1	1.2	1.1	1.1
1e-03	ST-HOSVD	6.7	17.7	24.1	30.6
	FaST-HOSVD	6.8	17.0	23.4	29.8
	FIST-HOSVD	1.1	1.2	1.3	1.3

tend to do marginally better. This can be seen in Tables 2 and 3, as well as Figures 7b and 7c.

These empirical observations motivate our goal of developing heuristics for determining a-priori when to apply each optimization. It is worth noting that truncating the dimensions in ascending order is a relatively arbitrary convention. Processing the later dimensions first would also yield a valid Tucker decomposition. In this scenario, the later dimensions would generally make up a larger portion of the computation. We would thus expect this to benefit even more from the fused approach utilized by the FaST-HOSVD and FIST-HOSVD. We plan to demonstrate this in future work by adding the capability to support processing the dimensions in any order to our implementations.

**Table 7: SP tensor memory consumption (in GB).
1 slice is ~ 11 GB**

ϵ	Slices:	5	10	15	20
1e-09	ST-HOSVD	35.5	70.3	—	—
	FaST-HOSVD	30.8	60.2	—	—
	FIST-HOSVD	1.5	1.8	1.5	1.7
1e-05	ST-HOSVD	10.4	20.4	30.4	—
	FaST-HOSVD	10.3	20.2	30.2	—
	FIST-HOSVD	1.2	1.2	1.4	1.3
1e-03	ST-HOSVD	3.2	6.3	9.3	12.3
	FaST-HOSVD	3.3	6.3	9.4	12.4
	FIST-HOSVD	1.1	1.1	1.1	1.1

5.3 Memory Consumption Results

Here we analyze the memory consumption of each implementation to provide insight into why some of the subsequent runtime results are unable to complete for certain problem sizes and why the FIST-HOSVD is able to handle them all. We begin by noting that we compute the amount of *memory consumed* as the memory high-water mark over the course of the computation minus the size of the original tensor. In our experiments we measured the memory consumption before and after the computation via the `getrusage()` function. Our FIST-HOSVD implementation allows the user to specify an approximate byte limit for the FIST-HOSVD. This is the β variable used in Algorithms 5 and 6 multiplied by the size of a single entry. In this case, the number of bytes in a single entry is 8, as all experiments were done in double precision. This byte limit controls the size of memory that is used to store the tensor blocks used in the fused computation. As of now we do not count the bytes of the matrices used to accumulate the Gram matrix. Generally, these will be small compared to the size of the tensor blocks, but we plan to incorporate these bytes in the user defined allocation in future work. In our experiments we allocated only 1 GB of memory for the FIST-HOSVD. The memory consumption results in Tables 5-7 and Figure 8 are in GB. From these tables we can see in all cases the FIST-HOSVD uses significantly less memory than both the FaST-HOSVD and ST-HOSVD implementations.

The most extreme case can be seen in Figure 8b for 626 timeslices of the HCCI dataset with an error tolerance of $1e-7$. Each HCCI timeslice is ~ 0.12 GB, thus the entire tensor is ~ 74.6 GB. In this case the ST-HOSVD, FaST-HOSVD and FIST-HOSVD, each respectively consumes approximately 161.9, 114.4, and 1.2 GB of memory. For example, in this instance memory consumption of the ST-HOSVD is computed as previously defined by: memory high water mark of the run minus the size of the original tensor, which is $236.5 - 74.6 = 161.9$. Thus, for this problem size and error tolerance, the FIST-HOSVD consumed approximately $\frac{161.9}{1.2} = 135\times$ less memory than the ST-HOSVD. For this problem size and error-tolerance, the size of the original tensor plus the amount of memory consumed for each implementation is less than the node's 256 GB limit and thus each is able to complete.

However, when the error tolerance is further reduced to $1e-9$, ST-HOSVD runs out of memory and crashes as seen in Table 6. This scenario is repeated several times for each dataset. Note that when the error tolerance is increased the computation benefits from more

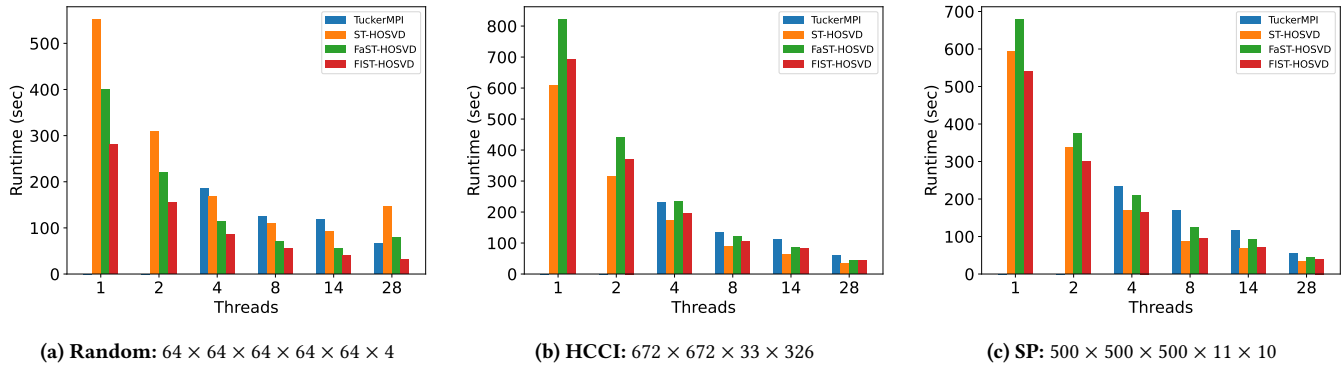


Figure 7: Sample bar charts of runs with an error tolerance (ϵ) of $1e-07$.

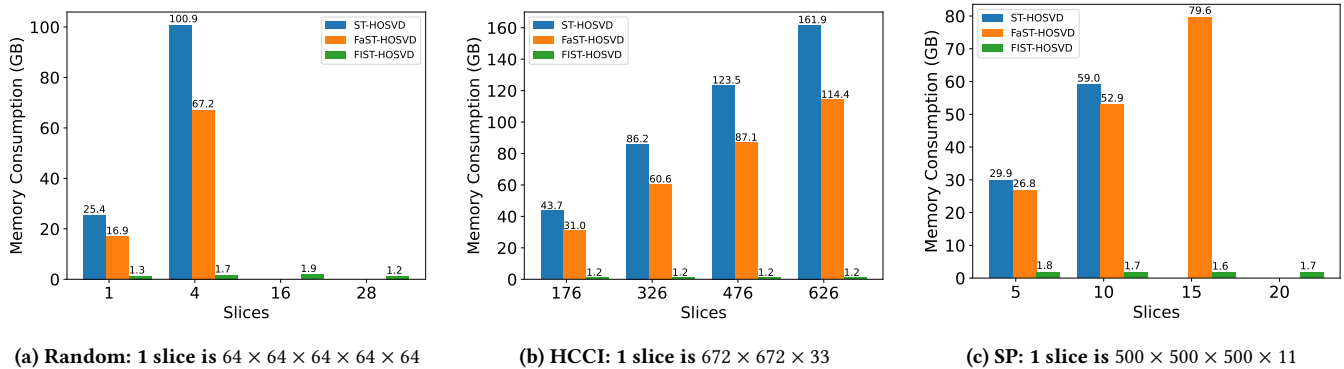


Figure 8: Memory consumption over different timeslice counts for an error tolerance (ϵ) of $1e-07$. Bars not shown did not complete due to running out of memory.

truncation, causing each intermediate TTM result to shrink after every iteration, thereby requiring less memory. This results in the out-of-place implementations being able to complete. The effect of error tolerance on truncation is demonstrated in Table 4. Note that the Random dataset is not truncated even for the highest error tolerance. Due to its lack of low-rank structure, the Random dataset is expected to have high rank. In this case, or when the error tolerance approaches 0, all datasets’ intermediate TTM results do not benefit from truncation and thus require at least $\sim 3\times$ the original tensor size in memory, resulting in the out-of-place algorithms not being able to complete for any problem size greater than $\frac{1}{3}$ the size of main memory.

6 CONCLUSIONS AND FUTURE WORK

We have presented two novel optimizations, kernel fusion and tensor tiling, that are aimed at improving the memory locality of the Sequentially Truncated Higher Order Singular Value Decomposition algorithm.

This block based approach was then coupled with a novel in-place transpose algorithm to drastically reduce the memory requirements of the ST-HOSVD. We demonstrated that the resulting algorithm, the FIST-HOSVD, is capable of computing the Tucker

decomposition of significantly larger tensors than the traditional ST-HOSVD, without compromising runtime performance.

From here, we aim to port our implementations and optimizations to the GPU. As previously mentioned, our algorithms are currently implemented in the Kokkos programming model to facilitate a GPU port. However, we need several more modifications to our implementations to ensure optimal performance, although we expect the general algorithmic structure of the FaST and FIST-HOSVD to remain the same.

Past literature has shown that GPU applications can especially benefit from kernel fusion due to memory bandwidth and latency being major bottle necks of GPU performance. Thus, porting kernel fusion optimizations and performing similar experiments to those previously shown is the natural next step. Furthermore, we believe the FIST-HOSVD can be extended to iteratively process tensor sections that each fit in a given GPU’s memory. In this way we can greatly increase the tensor size that a GPU can handle.

ACKNOWLEDGMENTS

We would like to thank Grey Ballard for many insightful discussions pertaining to this work, as well as the anonymous reviewers for their helpful comments and suggestions over the course of the review process. This work was performed as part of the ExaLearn

Co-design Center, supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. The views expressed in the article do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

REFERENCES

- [1] 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (June 2002), 135–151. <https://doi.org/10.1145/567806.567807>
- [2] Sardar Afra, Eduardo Gildin, and Mohammadali Tarrahi. 2014. Heterogeneous reservoir characterization using efficient parameterization through higher order SVD (HOSVD). In *2014 American Control Conference*. 147–152. <https://doi.org/10.1109/ACC.2014.6859246>
- [3] Grey Ballard, Alicia Klinvex, and Tamara G. Kolda. 2020. TuckerMPI: A Parallel C++/MPI Software Package for Large-Scale Data Compression via the Tucker Tensor Decomposition. *ACM Trans. Math. Softw.* 46, 2, Article 13 (June 2020).
- [4] Rafael Ballester-Ripoll and Renato Pajarola. 2015. Lossy volume compression using Tucker truncation and thresholding. *The Visual Computer* 32 (05 2015). <https://doi.org/10.1007/s00371-015-1130-y>
- [5] Bryan Catanzaro, Alexander Keller, and Michael Garland. 2014. A Decomposition for In-Place Matrix Transposition. *SIGPLAN Not.* 49, 8 (2014), 193–206.
- [6] J. Choi, X. Liu, and V. Chakaravarthy. 2018. High-Performance Dense Tucker Decomposition on GPU Clusters. In *SC'18: Int. C. for High Performance Computing, Networking, Storage and Analysis*. 543–553.
- [7] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. <https://doi.org/10.1016/j.jpdc.2014.07.003>
- [8] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *J. of Supercomputing* (2015).
- [9] Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations* (third ed.). The Johns Hopkins University Press.
- [10] Fred G. Gustavson and Walker David W. 2019. Algorithms for in-place matrix transposition. *Concurrency and Computation* 31, 13 (2019).
- [11] D. R. Hatch, D. del Castillo-Negrete, and P. W. Terry. 2012. Analysis and Compression of Six-Dimensional Gyrokinetic Datasets Using Higher Order Singular Value Decomposition. *J. Comput. Phys.* 231, 11 (jun 2012), 4234–4256. <https://doi.org/10.1016/j.jcp.2012.02.007>
- [12] Azam Karami, Mehran Yazdi, and Grégoire Mercier. 2012. Compression of Hyperspectral Images Using Discrete Wavelet Transform and Tucker Decomposition. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 5, 2 (2012), 444–450. <https://doi.org/10.1109/JSTARS.2012.2189200>
- [13] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.
- [14] Tamara G. Kolda and Jimeng Sun. 2008. Scalable Tensor Decompositions for Multi-aspect Data Mining. In *2008 Eighth IEEE International Conference on Data Mining*. 363–372. <https://doi.org/10.1109/ICDM.2008.89>
- [15] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. *SIGPLAN Not.* 26, 4 (April 1991), 63–74. <https://doi.org/10.1145/106973.106981>
- [16] J. Li, C. Battaglini, I. Perros, J. Sun, and R. Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
- [17] Zitong Li, Qiming Fang, and Grey Ballard. 2021. Parallel Tucker Decomposition with Numerically Accurate SVD. In *50th International Conference on Parallel Processing (Lemont, IL, USA) (ICPP 2021)*. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3472456.3472472>
- [18] Jinoh Oh, Kijung Shin, Evangelos Papalexakis, Christos Faloutsos, and Hwanjo Yu. 2017. S-HOT: Scalable High-Order Tucker Decomposition. 761–770. <https://doi.org/10.1145/3018661.3018721>
- [19] L. Omberg, G.H. Golub, and O. Alter. 2007. A Tensor Higher-Order Singular Value Decomposition for Integrative Analysis of DNA Microarray Data From Different Studies. *Proceedings of the National Academy of Sciences* 104, 47 (2007).
- [20] Eric T. Phipps and Tamara G. Kolda. 2019. Software for Sparse Tensor Decomposition on Emerging Computing Architectures. *SIAM Journal on Scientific Computing* 41, 3 (2019), C269–C290.
- [21] Cody Rivera, Jieyang Chen, Nan Xiong, Shuaiwen Leon Song, and Dingwen Tao. 2020. TSM2X: High-Performance Tall-and-Skinny Matrix-Matrix Multiplication on GPUs. arXiv:2002.03258 [cs.DC]
- [22] Shaden Smith and George Karypis. 2016. A Medium-Grained Algorithm for Sparse Tensor Factorization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 902–911. <https://doi.org/10.1109/IPDPS.2016.113>
- [23] Thiago Souza, A. L. Aquino, and D. Gomes. 2019. An Online Method to Detect Urban Computing Outliers via Higher-Order Singular Value Decomposition. *Sensors (Basel, Switzerland)* 19 (2019).
- [24] Siham Tabik, Gloria Ortega Lopez, and E M Garzon. 2014. Performance evaluation of kernel fusion BLAS routines on the GPU: iterative solvers as case study. *The Journal of Supercomputing* (11 2014). <https://doi.org/10.1007/s11227-014-1102-4>
- [25] A. Tretyakov and E. Tyrtshnikov. 2009. Optimal in-place transposition of rectangular matrices. *J. Complexity* 25 (08 2009), 377–384. <https://doi.org/10.1016/j.jco.2009.02.008>
- [26] L. R. Tucker. 1963. Implications of factor analysis of three-way matrices for measurement of change. In *Problems in measuring change*. Madison WI.
- [27] Nick Vannieuwenhoven, Raf Vandebril, and Karl Meerbergen. 2012. A New Truncation Strategy for the Higher-Order Singular Value Decomposition. *SIAM Journal on Scientific Computing* 34 (04 2012). <https://doi.org/10.1137/110836067>
- [28] M. Alex O. Vasilescu and Demetri Terzopoulos. 2002. Multilinear Analysis of Image Ensembles: TensorFaces. In *Computer Vision — ECCV 2002*. Anders Heyden, Gunnar Sparr, Mads Nielsen, and Peter Johansen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 447–460.