High-Performance Dense Tucker Decomposition on GPU Clusters

Jee Choi
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
jwchoi@us.ibm.com

Xing Liu*
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
xing.research@gmail.com

Venkatesan Chakaravarthy
IBM India Research Lab
New Delhi, India
vechakra@in.ibm.com

***Abstract*—The dense Tucker decomposition method is one of the most popular algorithms for analyzing and compressing data with multi-way relationship. Its execution time is typically dominated by dense matrix multiplication operations, which makes it well-suited for GPU acceleration. State-of-the-art distributed dense Tucker implementations for CPU clusters adopt multi-dimensional partitioning that optimizes for storage and communication. This, however, leads to smaller matrix dimensions that result in under-utilizing the GPU resources. In this paper, we present our optimized implementation and performance analysis of dense Tucker decomposition on a multi-GPU cluster. We propose three key optimizations: a new partitioning strategy that improves performance for GPUs, a new tensor matricization layout that halves the number of communication and matricization steps, and a variation of the randomized SVD algorithm to overcome the eigenvalue calculation bottleneck that arises from the high speedup gained from GPU acceleration.**

When compared to the state-of-the-art TuckerMPI library, our best GPU implementation, which employs all three optimizations described above, achieves up to $11.8\times$ speedup on 64 nodes. Our best CPU implementation, which also employs all three optimizations, achieves up to $3.6\times$ speedup over TuckerMPI on 64 nodes. When we compare our best GPU implementation to our best CPU implementation, the speedup ranges from $2.1\times$ to $3.6\times$ on a single node, and from $1.8\times$ to $3.3\times$ on 64 nodes, depending on the input data set.

***Keywords*-tensor decomposition, Tucker, GPU, MPI, distributed, high-performance computing, HPC, HOSVD**

I. Introduction

The Tucker method [1] is one of the most popular tensor decomposition algorithms currently in use, and its computation time on dense data sets is largely dominated by dense matrix-matrix multiplication. As such, GPUs would be highly suited for accelerating dense Tucker decomposition, and should yield significantly higher performance than CPUs.

However, our study of state-of-the-art distributed implementations of dense Tucker decomposition for CPUs [2], [3] reveals several bottlenecks that are particularly critical to GPUs. First, the prevalent data partitioning strategy for distributed dense Tucker implementations is N-D partitioning. While dividing a mode-*N* tensor into smaller *N*-dimensional blocks reduces storage for the factor matrices and eliminates the communication requirement for tensor matricization, it also creates the problem of (a) reducing the size of the matrices involved in the multiplication, thereby lowering the achievable performance on GPUs, and (b) creating communication for the Gram matrix and tensor-times-matrix (TTM) phases.

*During the period of this research, Xing Liu was affiliated with IBM T. J. Watson Research Center.

Secondly, calculating the eigenvectors of the Gram matrix increasingly dominates the overall execution time on a GPU cluster as the number of nodes grows. This is due to two factors: (a) GPU can achieve up to an order of magnitude higher performance on dense matrix multiplication, which drastically reduces the overall execution time, but (b) *distributed* solutions for eigenvector calculation on small matrices do not scale well, even on a small number of nodes; therefore, calculating identical eigenvectors on every node using a shared-memory solution is faster in many cases (we demonstrate this issue in Section III-C and Figure 5). Consequently, as the time spent on matrix multiplication decreases with the number of nodes, the eigenvector calculation takes up a larger *proportion* of the overall execution time, making it the critical bottleneck.

In this paper, we provide three optimizations to overcome these bottlenecks. First, we propose a new partitioning scheme that keep the matrix dimension sizes high, allowing the GPUs to operate more efficiently. This, however, shifts the communication requirements from the Gram and TTM phases to the tensor matricization phase. Therefore, we propose a new tensor matricization layout that cuts the number of communication and matricization steps in half. By keeping the n^{th} mode and the $(n+1)^{th}$ mode adjacent in memory, we only need to communicate and matricize the tensor on every *other* mode. Lastly, we propose a new variant to the traditional randomized SVD algorithm that reduces the time spent on the eigenvector calculation by as much as $9.8\times$, greatly reducing the impact of this bottleneck

When compared to the state-of-the-art distributed CPU implementation by Austin et al. [3], our *best* (i.e, one that uses all three optimizations) GPU and CPU implementations achieve up to $11.8\times$ and $3.6\times$ speedup, respectively, on 64 nodes. On our largest data set – a 5-D tensor that is 275 GB in size – our best GPU implementation running on just eight nodes closely matches the performance of TuckerMPI running on 64 nodes; when our GPU implementation employs all 64 nodes, it can decompose this tensor in a mere 0.4 seconds, $5.2\times$ faster than TuckerMPI. Additionally, when we compare our best GPU implementation to our best CPU implementation, the speedup ranges from $2.1\times$ to $14.4\times$ on a single node, and from $1.8\times$ to $3.3\times$ on 64 nodes, depending on the input data set. Finally, when we compare our best GPU implementation to the GPU implementation that only uses our new partitioning method on 64 nodes, we observe a $2.3\times$ speedup.

Our interest in tensor decomposition stems from the fact that it is quickly becoming a popular technique for *analyzing*

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

and *compressing* large data sets with multi-way relationship. It has found application in fields ranging from identifying phenotypes in electronic health records (EHR) [4] to compressing scientific data [3] and convolutional neural network layers [5]. With Summit and Sierra – the next generation of supercomputers – relying on GPUs for performance, and with increasingly larger social data becoming available for data mining, an *efficient* and *scalable* GPU solution for tensor decomposition will become essential for solving critical problems in the near future.

II. Background

We begin by providing a brief overview of the Tucker decomposition method and related notations. For a more in-depth discussion of tensors and tensor computations, we direct the readers to the work by Kolda and Bader [6], [7].

A. Tensor notation

Tensors are the higher-order generalization of matrices. An N dimensional tensor is also referred to as having N *modes* or a *mode-N* tensor. We use the following notations in this paper:

1) *Scalars* are denoted by lower case letters (e. g., a).
2) *Vectors* are denoted by bold lower case letters (e. g., $\mathbf{a}$).
3) *Matrices* are denoted by bold capital letters (e. g., $\mathbf{A}$). If $\mathbf{A}$ is a $I_1 \times I_2$ matrix, it can also be denoted as $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2}$.
4) *Higher-order tensors* are denoted by Euler script letters (e. g., $\mathcal{X}$). A mode-N tensor whose dimensions are $I_1 \times I_2 \times \cdots \times I_N$ can be denoted as $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$. The total size of the tensor is denoted by $I = \prod I_n$, and the size of all modes excluding n is denoted by $\hat{I}_n = \prod_{i \neq n} I_n$.
5) *Fibers* are the higher-order analog of matrix rows and columns. A mode-n fiber of $\mathcal{X}$ is defined by fixing every mode *except* the n^{th} mode of $\mathcal{X}$.
6) *Slices* are two-dimensional sections of a tensor and are defined by fixing all except *two* of the modes.
7) *Tensor matricization* is the process of reordering the elements of a tensor into a matrix. Mode-n matricization of a tensor $\mathcal{X}$, denoted as $\mathbf{X}_{(n)}$, is a $I_n \times \hat{I}_n$ matrix. It can be achieved by taking the mode-n fibers of $\mathcal{X}$, and arranging them as the columns of the resulting matrix.
8) *Tensor times matrix (TTM)* is the product between a tensor and a matrix that results in another tensor. The mode-n TTM of $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and $\mathbf{A} \in \mathbb{R}^{J \times I_n}$ results in a $I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$ tensor and is denoted as $\mathcal{X} \times_n \mathbf{A}$. It can be better expressed in terms of matricized tensors:

$$\mathcal{Y} = \mathcal{X} \times_n \mathbf{A} \leftrightarrow \mathbf{Y}_{(n)} = \mathbf{A}\mathbf{X}_{(n)} \tag{1}$$

Note that the dimension sizes must match for the operation to be valid.

B. Tucker decomposition

Tucker decomposition [1] approximates the tensor $\mathcal{X}$ as

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \cdots \times_N \mathbf{U}^{(N)} \tag{2}$$

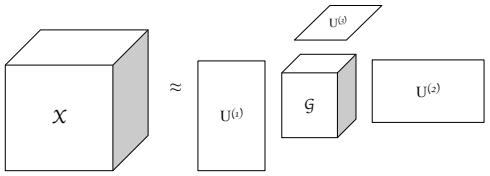


Figure 1: Tucker decomposition of a three-dimensional tensor $\mathcal{X}$

where $\mathcal{G}$ is known as the *core tensor*, with dimensions $R_1 \times R_2 \times \cdots \times R_N$, and $\mathbf{U}^{(n)}$ are factor matrices of dimensions $I_n \times R_n$ for $n = 1, \cdots, N$. This process is similar to finding a low-rank approximation, where $R_1, R_2, \cdots, R_N$ are the ranks of the reduced representation. Tucker decomposition for a three-dimensional tensor is illustrated in Figure 1.

There are two common methods for computing the Tucker decomposition. The first method is the truncated higher-order singular value decomposition (T-HOSVD) [8], in which the factor matrices $\mathbf{U}^{(n)}$ are set to be the R_n leading left singular vectors of $\mathbf{X}_{(n)}$, and the core tensor is computed by a sequence of TTMs between the tensor and the factor matrices. The second method is the higher-order orthogonal iteration (HOOI). The HOOI method uses T-HOSVD to initialize the factor matrices and further improves the accuracy through a few iterations of alternating least squares (ALS). For certain applications, it has been shown in prior work that using T-HOSVD alone is sufficient to yield accurate results [3], [9], with HOOI making only little improvement in accuracy.

In this paper, we choose to parallelize the *sequentially truncated HOSVD* (ST-HOSVD) method, a variant of T-HOSVD. For certain applications [9], ST-HOSVD reduces the number of floating-point operations to compute the decomposition over the T-HOSVD method, while simultaneously improving the approximation accuracy. The ST-HOSVD method is shown in Algorithm 1. The reduction in floating-point operations comes from replacing the calculation of SVD on the matricized tensor $\mathbf{X}_{(n)}$ with the calculation of SVD on the matricized *intermediate* tensor $\mathbf{Y}_{(n)}$, whose size gradually decreases as each mode is traversed (line 5 in Algorithm 1, see Section IV-B for more details.).

Algorithm 1: Sequentially Truncated High-Order Singular Value Decomposition (ST-HOSVD)

```
   Data: X, {R_n}
   Result: G, {U^(n)}
1  Y ← X;
2  for n = 1, ..., N do
3  |  S ← Y_(n) Y_(n)^T ;  // Gram matrix construction
4  |  U^(n) ← R_n leading eigenvectors of S ;
   |    // Eigensolve
5  |  Y ← Y ×_n U^(n)T ;                          // TTM
6  end
7  G ← Y;
```

It is worth noting that all the three Tucker decomposition methods described above (T-HOSVD, ST-HOSVD, and HOOI) share a similar computational structure (i. e., TTM

sequence and SVD). Therefore, the parallel ST-HOSVD algorithm proposed in this paper can easily be applied to both T-HOSVD and HOOI.

C. Related work

Tensor decomposition for high-performance computing (HPC) has recently gained momentum in the field of Big Data *mining* and *analytics*.

Applications: Tucker decomposition has been shown to provide high compression ratio with little quality loss for dense data, such as images [10], volume rendering [11], and scientific simulation [3]. For data analytics, Tucker decomposition has been shown to generate meaningful disease hierarchy within public healthcare records [12], and detect anomalous behavior from streaming network traffic data [13].

Shared memory optimizations: The work by Li et al. [14] improves the performance of dense TTM kernel for Tucker decomposition by replacing the expensive tensor matricization step with an in-place computation. In the work by Smith et al. [15], the authors apply compressed sparse fiber (CSF), a form of hierarchical compressed sparse row (CSR) for sparse matrices, to demonstrate significant speedup over prior state-of-the-art.

Distributed Tucker decomposition: The first distributed implementations of dense Tucker decomposition is *TuckerMPI* [3] by Austin et al. This was quickly followed by two others. First, the work by Kaya et al. [16] reformulates the successive TTM steps to increase the re-use of intermediate computation. Second, in the work by Chakaravarthy et al. [2], dynamic programming is used to determine the optimal sequence of TTM operations to minimize computation and communication.

GPU acceleration: The only other GPU implementation of Tucker decomposition is the work by Shi et al. [17], which implements the HOOI method using NVIDIA's *StridedBatchedGEMM* library on a single node. While this library provides a convenient solution for improving the performance of matrix multiply on small matrices, their work does not offer any insight into reducing communication or improving the performance of the SVD bottleneck. Also, their evaluation was limited to a small three-dimensional tensor with a mode length of 120 and decomposed rank of 10 (tensor size < 0.014 *GB*). In contrast, our largest data set is 275 *GB*.

As far as we are aware, our work is the first distributed implementation of Tucker decomposition using GPUs for arbitrarily large data sets.

III. Distributed Parallel ST-HOSVD Algorithm

A. Parallel data distribution

Prior work on distributed dense Tucker decomposition [2], [3], [18] distributes the tensor using a N-D partitioning method, also known as the *medium-grained* partitioning method. For a mode-N tensor of size $I_1 \times I_2 \times \ldots \times I_N$, the medium-grained method divides the tensor along all modes and distributes it across P processors organized in a logical N-way processor grid of size $P_1 \times P_2 \times \ldots \times P_N$. Each processor owns a distinct sub-tensor of size $I_1/P_1 \times I_2/P_2 \times \ldots \times I_N/P_N$, with I/P entries.

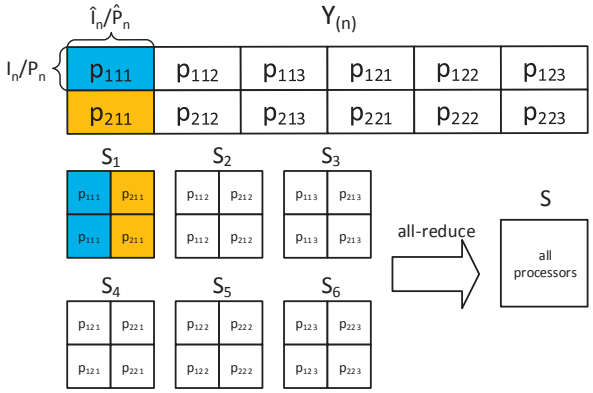


Figure 2: Computing Gram matrix using the medium-grain method

The major disadvantage of using the medium-grained method for dense Tucker decomposition on a GPU cluster is that calculating the Gram matrix and TTM on a large number of nodes requires computing matrix-matrix multiplications for matrices with *small* dimension sizes, which is well-known to performs poorly on GPUs [19] (see Table III for our own comparison). For example, Figure 2 shows the parallel mode-n Gram matrix computation for a mode-3 tensor using the medium-grained method. The data is distributed on a $2 \times 2 \times 3$ processor grid. In the figure, $\hat{P}_n = \prod_{i \neq n} P_n$ denotes the number of processors in all dimensions except n, and p_{ijk} denotes the processor with the i, j and k indices along the first, second and third dimension of the grid, respectively.

As seen in the figure, the processors with the same j and k indices need to work together to compute a partial Gram matrix S_c, for $c \in \{1, 2, \ldots, \hat{P}_1\}$. Each processor calculates one column block of S_c (highlighted in color), which requires computing P_n matrix-matrix multiplications of size $\frac{I_n}{P_n} \times \frac{\hat{I}_n}{\hat{P}_n}$. When P_n is large, $\frac{I_n}{P_n}$ will become too small to achieve good compute efficiency on GPUs.

To address this problem, we propose a $(N-2)$-D partitioning method for data distribution. The proposed method, which we call the *slice block* partitioning method, distributes the tensor by blocks of tensor *slices* (described in Section II-A). Figure 3 shows the mode-n Gram matrix computation for the same tensor on 12 processors using the slice block distribution, in which each processor owns n_s tensor slices of size $I_n \times I_{n+1}$, where $n_s = \prod_{i \neq n, n+1} I_i/P$. To compute the Gram matrix, each processor needs to compute only one large matrix-matrix multiplication of size $I_n \times \frac{\hat{I}_n}{P}$. The size is typically large enough to achieve good performance on GPUs, even when P is very large. Note that $\frac{\hat{I}_n}{P}$ is typically larger than I_n, meaning that the matrix assigned to each processor in our slice block partitioning is *not* tall and skinny as it appears in Figure 3. It was only drawn in such manner to better illustrate how each block is assigned to a processor.

Our implementation also *batches* slices together and use CUDA *streams* to overlap communication and computation

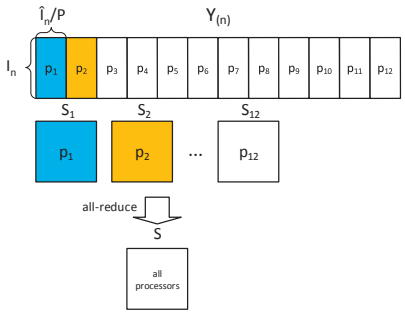


Figure 3: Computing the Gram matrix using the slice block partitioning

on the GPU to further improve performance. This is similar in idea to *StridedBatchedGEMM* [17], and our slice block partitioning strategy can leverage such libraries to achieve even higher performance. However, even in the absence of such libraries (since such highly optimized libraries may not be available on every platform), our partitioning strategy will improve performance over traditional methods.

One main difference between Tucker decompositions using the medium-grain method and the slice block distribution is *when and where* the communication occurs. For the medium-grain method, communication is required *during* the Gram matrix and TTM computation, while the tensor matricization step only involves local data movement (i.e., no communication). As seen in Figure 2, each processor computes one column block of S_c, which requires sending its sub-tensor to and receiving sub-tensors from $P_n - 1$ processors that have the same j and k indices. It is followed by an all-reduce operation across $\hat{P}_n$ processors to sum all the partial Gram matrices. Using the α–β model, where α is the latency cost and β is the per-word transfer cost, the total communication cost is

$$2(P_n - 1)(\alpha + \beta \frac{I}{P}) \tag{3}$$

Since the Gram matrix S is much smaller than the tensor ($I_n \times I_n$ for the Gram matrix vs. $\frac{I}{P}$ for the tensor), we ignore the communication cost for the all-reduce operation. While we do note that with non-uniformly shaped tensors with skewed dimension sizes, the Gram matrix may be comparable in size to the tensor, we nevertheless ignore it, as it is uncommon and only makes the communication cost higher for medium-grained partitioning.

In contrast, our slice block method does not require communication during the Gram matrix and TTM computations (it does require the same all-reduce to sum the partial Gram matrices like the medium-grained method), but requires communication for matricizing the tensor. This is done via an all-to-all operation across P processors, in which each processor sends and receives I/P elements. The total communication is

$$2((P-1)\alpha + \beta \frac{I}{P}) \tag{4}$$

Since $\frac{I}{P}$ is typically large, the bandwidth term dominates (e.g., for a $3000 \times 3000 \times 3000$ tensor running on the largest GPU-based supercomputer (Titan), $P = 20000$ and $\frac{I}{P} = 1350000$). Therefore, the slice block distribution method has lower communication overhead than the medium-grained method.

Generally, the medium-grained partitioning method has two key advantages - it strikes a balance between minimizing synchronization (coarse-grained) and load balancing (fine-grained), and saves memory by allowing the factor matrices to also be partitioned across processors. As such, it is highly suited for sparse CP decomposition [20], where the nonzero sparsity is unpredictable, and the factor matrices can be relatively large due to long mode lengths found in real data sets [21]. However, for dense Tucker decomposition, the benefits are less pronounced. since load balancing can be easily achieved using even 1-D partitioning, and the core tensor takes up much more space than the factor matrices. Therefore, our slice block method is better suited for dense Tucker decomposition, particularly when GPUs are involved.

B. New tensor matricization layout

As we mentioned in Section III-A, using our slice block partitioning method shifts the communication from the Gram matrix calculation and the TTM phases to the matricization phase and reduces the overall communication overhead. In this section, we describe our optimization that further reduces the communication overhead, now incurred at the matricization phase.

Tensor matricization involves reorganizing the data layout such that the n^{th} mode is the fastest changing dimension. The conventional scheme [6] "swaps" the 1^{st} and the n^{th} mode so that it results in the following ordering of modes: $n, 2, ..., (n-1), 1, (n+1), ..., N$. Using this conventional layout *and* our slice block partitioning method requires, for *every* mode, an all-to-all communication to re-partition the intermediate tensor, followed by a local matricization.

We propose a new matricization layout that synergises with our slice block partitioning method to reduce the number of communication and matricization steps by a factor of *two*; communication is now required for every *even* or *odd* numbered mode (i.e., every other mode). Instead of "swapping" the 1^{st} and the n^{th} mode, we "rotate" the modes so that the we end up with the following ordering instead: $n, (n+1), ..., N, 1, 2, ..., (n-1)$. That is, we have essentially left-shifted the current mode to the end, so that mode-n, and mode-$(n+1)$ would always be adjacent.

When a processor has completed calculation for mode n, it can achieve matricization for mode $n+1$ (i.e., construct $Y_{(n+1)}$ from Algorithm 1) by taking the block of n_s slices of $Y_{(n)}$ that were assigned to it through slice block partitioning, and then transposing each slice one by one and concatenating the transposed slices. With this layout, no communication is required to re-distribute the tensor for mode $n+1$. We refer to this optimization as tensor "reuse," and is illustrated in Figure 4.

Moreover, the transpose step is often unnecessary. During the subsequent Gram and TTM phases, we can leverage the

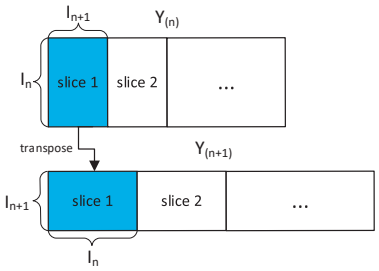


Figure 4: Tensor rotation for a $I_1 \times I_2 \times I_3$ tensor. Each *slice* is transposed to matricize the tensor for the next mode.

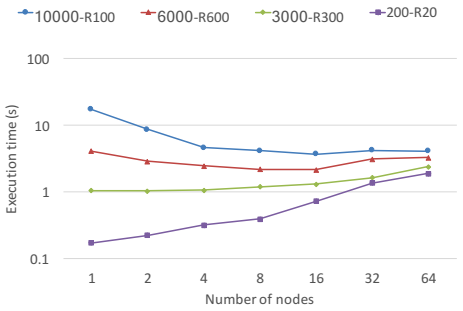


Figure 5: Execution time scalability for SCALAPACK PDSYEVX.

transpose option provided in many modern matrix multiplication libraries (e.g, LAPACK and cuBLAS) to multiply transposed matrices (i.e., matricized tensors), typically at the same performance as multiplying non-transposed matrices. And even if such an option is not available, transposing a slice (i.e., a matrix) is a simple operation, and is optimized in many libraries.

C. Improving the performance via randomized SVD

Prior work on distributed Tucker decomposition has used all-reduce to distribute the final Gram matrix to every node, and then execute the eigenvalue decomposition function redundantly on every node [2], [3]. The primary reason for this is the *inefficiency* of distributed eigenvalue decomposition on small matrices.

This inefficiency can be seen in Figure 5, which shows how the ScaLAPACK library's *pdsyevx* – **p**arallel (distributed) **d**ouble-precision **syevx** (eigenvalue decomposition) – function scales with the number of nodes. For each test matrix, the best distribution granularity was chosen. We can see from the figure that for a matrix with relatively large dimension size (i.e, "10000-R100" which refers to calculating the first 100 eigenvectors of a 10000×10000 matrix. Remember from Algorithm 1 that the ST-HOSVD algorithm only requires the first R_n leading eigenvectors), the execution time initially decreases with more nodes. However, after only *four* nodes, the execution time levels off and even starts to increase at higher number of nodes. For a dimension size of 6000 (i.e., "6000-R600"), there is little to no benefit to using more than one node, and for the two smallest dimension sizes (i.e, "3000-R300" and "200-R20"), using more than one node actually *increases* the execution time.

Since mode lengths are often small for dense tensors due to the number of elements growing exponentially with the number of modes, it is difficult to find a scalable solution to calculating the eigenvectors for dense Tucker decomposition. Even if one of the tensor mode lengths is large (e.g., a 160 GB tensor of size $100000 \times 2000 \times 100$), the lengths of the remaining modes will proportionally be smaller, leading to the same problem.

Replicating the eigenvalue calculation on every node is often "good enough" on CPU platforms, as the Gram matrix and TTM calculations tend to dominate the overall execution time, even on a large number of nodes. This, unfortunately, changes on GPU platforms, where we see an order of magnitude better performance on these two key calculations. For the input tensor 3D3000-R300 (Table I), if we replicate the eigenvector calculation on every node, it starts to dominate the overall execution time when more than *four* nodes are used. On 64 nodes, it accounts for approximately 82% of the total execution time. Therefore, an alternative solution to the distributed eigenvalue calculation is necessary to allow our GPU implementation to scale beyond four nodes. We do, however, note that if the dimension sizes were large enough (i.e., $\gg 10000$), using ScaLAPACK or other distributed solutions may give us the scalability we seek.

To overcome this bottleneck, we take advantage of matrix symmetry – for symmetric matrices, the leading R_n eigenvectors are identical to the R_n left singular vectors – to use SVD in place of eigenvalue decomposition, and propose a modified randomized SVD algorithm based on the work by Halko et al. [22]. Our modified randomized SVD algorithm is shown in Algorithm 2.

Algorithm 2: Modified randomized SVD algorithm. We modify the original algorithm [22] to further reduce the size of the target matrix via Gram matrix (line 5) construction.

Data: $\mathbf{A} \in \mathbb{R}^{m \times n}$; number of left singular vectors required, r
Result: left singular vectors $\boldsymbol{\Upsilon}$
1 $\mathbf{O} \leftarrow$ Gaussian random matrix $\in \mathbb{R}^{n \times 2r}$;
2 $\mathbf{T} \leftarrow \mathbf{AO}$;
3 $[\mathbf{Q}\ \mathbf{R}] \leftarrow QR(\mathbf{T})$; `// Find orthonormal basis`
4 $\mathbf{B} \leftarrow \mathbf{Q}^T\mathbf{A}$; `// such that` $\mathbf{A} \approx \mathbf{QQ}^T\mathbf{A} = \mathbf{QB}$
5 $\mathbf{G} \leftarrow \mathbf{BB}^T$; `// Gram matrix construction`
6 $[\hat{\boldsymbol{\Upsilon}}\ \hat{\mathbf{\Sigma}}\ \hat{\boldsymbol{\Psi}}] \leftarrow SVD(\mathbf{G})$;
7 $\boldsymbol{\Upsilon} \leftarrow \mathbf{Q}\ \hat{\boldsymbol{\Upsilon}}$;

In the original algorithm, instead of directly calculating the SVD of the input matrix $\mathbf{A}$, we first find a matrix $\mathbf{Q}$ with r orthonormal columns such that $\mathbf{A} \approx \mathbf{QQ}^T\mathbf{A}$. Then, we calculate the matrix $\mathbf{B} = \mathbf{Q}^T\mathbf{A}$, which is relatively smaller than $\mathbf{A}$, and find the SVD of $\mathbf{B}$. The singular values and the right singular vectors of $\mathbf{B}$ are identical to those of the input matrix $\mathbf{A}$, and the left singular vectors of $\mathbf{A}$ can be constructed from $\mathbf{Q}$ and the left singular vectors of $\mathbf{B}$.

However, since we are only interested in the left singular vectors of the input matrix **A**, we can further improve the original algorithm by calculating the gram matrix of **B** (matrix **G** in line 7) and applying SVD to **G**. As in the original algorithm, the left singular vectors can be constructed from **Q** and the left singular vectors of the Gram matrix **G**. Constructing the right singular vectors of the input matrix **A** involves more computation, but we do not calculate it since they are not needed. Our modified randomized SVD algorithm is currently implemented only on the CPU, as using the GPUs for such small matrices would likely takes longer.

While calculating Gram *one additional* times during the randomized SVD step can potentially lower accuracy, it is unlikely to happen in real world scenarios, as the second Gram calculation (during the randomized SVD step) is much smaller than the Gram matrix calculation on the *entire* matricized tensor (*required* by the original ST-HOSVD algorithm), and therefore, its impact should be negligible. Moreover, as long as the rank of the decomposition does not far exceed the rank of the tensor, the input matrix to the randomized SVD algorithm should always be well-conditioned. While calculating the *precise* rank of a tensor is an NP-hard problem, heuristics can be used to obtain an approximation [6], and this should ensure that the matrix is always well-conditioned before its SVD is calculated.

On the other hand, this algorithm also demonstrates a potential for performance-accuracy trade-off via the number of randomized columns r. In the original algorithm, twice the number of required eigenvectors are used to initialize the random matrix. However, we found empirically that we can achieve similar accuracy using fewer randomized columns, which further and significantly improves its performance. We explore this issue in Section IV-D.

D. Parallel Cost Analysis

Lastly, we analyze the parallel cost of the three key components of the ST-HOSVD algorithm – *Gram*, *Eigensolve*, and *TTM* – using our slice block partitioning method. All calculations are done *per node*.

a) Gram: Each processor has a block of the intermediate matricized tensor $\mathcal{Y}$ (Algorithm 1), whose size is $J_1 \times J_2 \times \cdots \times J_N$. Note that initially $J_n = I_n$ for all n. However, as we go through each mode, $\mathcal{Y}$ becomes smaller as its mode length is reduced from I_n to R_n. Assuming we traverse the modes from 1 to N in order, we define $\hat{J}_n = \prod_{i=1}^{n-1} R_i \prod_{i=n+1}^{N} I_i$. Then, the total cost of the Gram operation is

$$\sum_{n=1}^{N} \frac{2I_n^2 \hat{J}_n}{P} \tag{5}$$

b) Eigensolve: After the Gram calculation, each processor redundantly stores its result, which is a matrix of size $I_n \times I_n$. Calculating the leading eigenvectors of this matrix incurs the following cost:

$$\sum_{n=1}^{N} \frac{10}{3} I_n^3 \tag{6}$$

c) TTM: After the eigenvector calculation, the resulting factor matrix is multiplied to the slice block of the intermediate tensor $\mathcal{Y}$ that each processor owns. This is done by multiplying the factor matrix ($R_n \times I_n$) by the slice block ($I_n \times \frac{\hat{J}_n}{P}$), which incurs the following cost:

$$\sum_{n=1}^{N} 2\frac{R_n I_n \hat{J}_n}{P} \tag{7}$$

IV. Experimental Results

A. Test platform and data sets

We used a cluster of dual-socket POWER8 systems with a total of 20 cores, and four NVIDIA P100 Pascal GPUs connected via NVLink for all our experiments. Each POWER8 core is capable of executing two 128-bit vector FMA instructions at 3.624 GHz for a peak double-precision throughput of 29 GFLOPS per core, or 580 GFLOPS per node. Each P100 GPU is equipped with 3584 CUDA cores running at 1.48 GHz, for a peak double-precision FMA throughput of 5.3 TFLOPS.

Unless otherwise noted, the CPU implementation utilizes all 20 cores, with two threads per core, and the GPU implementation utilizes all four GPUs. The operating system was Red Hat Linux 4.8.5-11, and our code was compiled using IBM XL C/C++ 13.1.5 for Linux. For the GPU code, we used CUDA 8.0, and for MPI, we used Spectrum MPI 10.2.0. We compare our implementations against the latest version (as of January 23, 2018) of TuckerMPI [3], the state-of-the-art dense Tucker library for CPU clusters, compiled and running on the same system.

Table I shows a list of data sets that were used to evaluate our work. The entries "Dimension" and "Rank" indicate the length and the number of principal components along each mode, and are identical for every mode. Input tensors were generated by taking a core tensor of size "Rank" and multiplying it by the factor matrices of appropriate size along each mode. Both the core and the factor matrices were generated randomly with Gaussian distribution in double-precision. Since the data sets are dense, only the meta data (i.e., number of modes and dimension size) is important to performance, and a range of dimensions (3-5) and mode lengths (128-3000) were selected to cover those found in real data sets [3].

Name	Modes	Dimension	Rank	Size
3D3000-R300	3	3000	300	216 GB
3D2000-R1000	3	2000	1000	64 GB
4D200-R20	4	200	20	13 GB
4D400-R64	4	400	64	205 GB
5D128-R16	5	128	16	275 GB

Table I: List of data sets used for evaluation.

Note that we did not include non-uniformly shaped tensors because the amount of computation and communication required changes with the order in which the modes are traversed for such tensors. This issue has already been explored extensively in prior work [2], [3], and is orthogonal to our optimizations and analysis, and as such, our optimization and analysis can be applied to non-uniformly shaped tensors with minimal modifications. The *performance* of the key kernels

that are executed for *each mode* (i.e., TTM, SVD, and Gram) behave predictably with respect to its length and rank (i.e., the workload size), and the mode length variation only obfuscates our analysis. However, we do admit that this may change with extremely short mode lengths, which are typically uncommon.

Table II shows a list of our implementation "variations" and description of their associated optimizations. First, every implementation in our evaluation uses our slice block distribution method (Section III-A). Second, our implementation can either employ or not employ tensor re-use (Section III-B). Lastly, we have a choice of three different methods for calculating the eigenvectors:

- *DSYEVX*: replicating the Gram matrix and computing the eigenvalue decomposition on ever node;
- *PDSYEVX*: using the distributed parallel implementation from the ScaLAPACK library;
- *Randomized SVD*: using our *shared-memory* randomized SVD implementation.

The implementations variations are applicable to both the CPU implementation and the GPU implementation. The key differences between our CPU and GPU implementations are that on the GPU implementation,

1) the Gram matrix and TTM calculations are off-loaded to the GPUs, and
2) the DSYEVX operation is calculated either on the CPU (ESSL) or the GPU (cuSolver), depending on whether the data set fits in the GPU device memory, and whichever is faster.

ID	Name	Re-use	Eigenvalue
1	NoReuse + DSYEVX	No	DSYEVX
2	Reuse + DSYEVX	Yes	DSYEVX
3	Reuse + PDSYEVX	Yes	PDSYEVX
4	Reuse + RndSVD	Yes	Randomized SVD

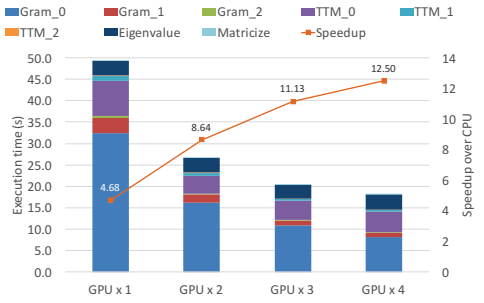
Table II: List of implementation variations and optimizations.

B. Single node performance

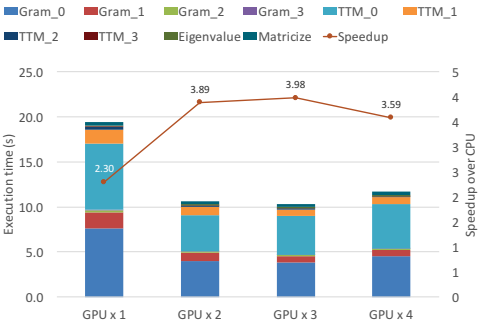
Figure 6 shows the observed GPU speedup over the CPU for the **Reuse + DSYEVX** variation on the data sets 3D3000-R300 and 4D400-R64, when the number of GPUs is increased from one to four on a single node. Note that *Gram_n* and *TTM_n* refers to the Gram and TTM calculation for mode *n*. *Eigenvalue* and *Matricize* times are summed across all modes, as they were too small to be easily differentiated in the figure otherwise.

When the Gram matrix and TTM are computed for the *first* mode (i.e., *Gram*_0 and *TTM*_0), the intermediate tensor $\mathcal{Y}$ is identical in size to the original tensor, and therefore, they account for majority of the total execution time. Since both computations are basically dense matrix-matrix multiplication (DGEMM), we also see a significant speedup over the CPU when the work is offloaded to even a single GPU.

As more GPUs are utilized, the size of work per GPU diminishes, and as a result, the *efficiency* on the GPU also decreases (i.e., we see lower performance for DGEMM on



(a) 3D3000-R300



(b) 4D400-R64

Figure 6: Execution time breakdown across multiple GPUs.

smaller matrices), lowering the overall parallel efficiency (i.e., $\frac{speedup}{\#\ GPUs}$). This effect is also exacerbated when the dimension size of the tensor is smaller, as it can be seen from Figure 6b, where the speedup plateaus at two GPUs.

Table III shows how the performance of *Gram*_0 and *TTM*_0 scales with the number of GPUs. For the larger tensor 3D3000-R300, we see near-peak performance (95% of peak) and near-perfect scaling (parallel efficiency of 0.995) over four GPUs for *Gram*_0, which is essentially a dense matrix multiplication between two identically sized matricized tensors. The performance does not scale as well for *TTM*_0, as it is a matrix multiplication between a tensor ($\mathcal{Y}$) and a much smaller factor matrix ($\mathbf{U}^{(n)}$). The overall performance and scalability is much lower for 4D400-R64, as the dimension size of both the tensor and the factor matrices are much smaller.

Function	Performance (GFLOPS)			
	1× GPU	2× GPU	3× GPU	4× GPU
3D3000-R300				
*Gram*_0	5008	9998	14982	19941
*TTM*_0	3914	7617	7241	6446
4D400-R64				
*Gram*_0	2685	5159	5325	4486
*TTM*_0	895	1643	1524	1329

Table III: Performance scaling across multiple GPUs

C. Rank vs. performance on a single node

We also test how the performance and approximation error (Equation 8) scales as we increase the number of decomposed

rank, R_n on a single node. We use a different data set – a three dimensional tensor with mode length of 2000 and *1000* principal components, or 3D2000-R1000 – for this experiment, as we wish to demonstrate how the performance and approximation error changes with respect to the number of decomposed rank at a finer granularity.

The GPU implementation uses both tensor re-use and randomized SVD (**Reuse + RndSVD**), whereas the CPU implementation uses tensor re-use and DSYEVX (**Reuse + DSYEVX**). We chose to use DSYEVX (instead of our modified randomized SVD) for the CPU implementation in order to compare against the accuracy of DSYEVX (the default choice for Algorithm 1), and also because the DSYEVX function made up only a small percentage of the total time on the CPU implementation, and using randomized SVD did not improve the performance significantly (performance improves by less than 2%).

As it can be seen from Figure 7a, our GPU implementation using randomized SVD shows similar approximation error to that of the CPU implementation using DSYEVX at every rank (R_n), while maintaining a significant speedup ($\geq 10\times$). We calculate the approximation error using the equation

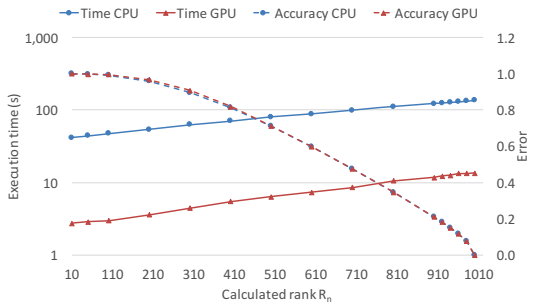
$$E = \frac{\|\mathcal{X} - \mathcal{X}'\|}{\|\mathcal{X}\|} \tag{8}$$

where $\mathcal{X}'$ is reconstructed from the calculated core and factor matrices.

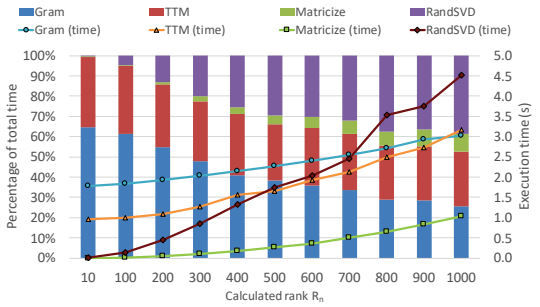
Figure 7b shows a breakdown of where time is spent as we increase the calculated rank for the GPU implementation. As we increase the rank from 10 to 1000, the *proportion* of time taken up by *all* Gram calculations decreases, since the most expensive Gram calculation – along the first mode – is independent of the rank (i.e., the execution time for *Gram*_0 remains constant). The increase in total time for all Gram calculations when rank is increased from 10 to 1000 is less than $2\times$.

On the other hand, all TTM calculations are dependent on the calculated rank R_n. For the first mode, TTM requires multiplying the mode-1 factor matrix ($R_n \times 2000$) by the full tensor (2000×2000^2), and for the last mode, TTM requires multiplying the mode-3 factor matrix ($R_n \times 2000$) by the intermediate core tensor $\mathcal{Y}$ ($2000 \times R_n^2$). Therefore, the total time for all TTM calculations increases by approximately $3\times$ when rank is increased from 10 to 1000.

Randomized SVD execution time demonstrates a linear dependence on R_n, and increases by approximately $360\times$ as the rank increases from 10 to 1000. As a result, randomized SVD makes up approximately 40% of the total execution time when the rank is 1000, as opposed to 0.5% when the rank is 10. This stresses the importance of having a fast and/or scalable solution to eigenvalue decomposition for ST-HOSVD on GPU implementations. If we had used the shared-memory DSYEVX eigenvalue decomposition function, instead of our randomized SVD algorithm, this bottleneck would have been much larger (27.1% at R_n = 10 to 62.5% at R_n = 1000).



(a) CPU vs. GPU. Performance and accuracy.



(b) GPU execution time breakdown.

Figure 7: Rank vs. performance and approximation error on a single node for 3D2000-R1000.

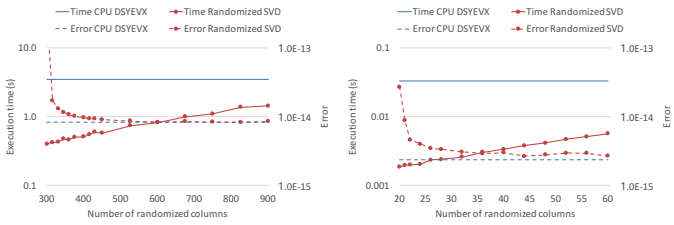
D. Randomized SVD performance

Our modified randomized SVD algorithm further improves the performance of our CPU and GPU implementations, particularly for the 3D3000-R300 tensor. First, we show how both the execution time and error changes with respect to the number of randomized columns used for our randomized SVD algorithm, and how it compares to those of using the eigenvalue function (DSYEVX). We use the CPU eigenvalue function from the ESSL library for this comparison, since our randomized SVD implementation is also CPU only, and because eigenvalue performs similarly on the two platforms for our data sets. We also do not compare the distributed eigenvalue function (PDSYEVX) and only show single node performance results, since using the distributed eigenvalue decomposition function (PDSYEVX) generally leads to lower performance than replicating the shared-memory eigenvalue or randomized SVD function on every node (Section III-C).

Figure 8a shows the result for a 3000×3000 matrix (i.e., used for 3D3000-R300) and Figure 8b shows the result for a 200×200 matrix (i.e., used for 4D200-R20). Note that only the first 300 (3D3000-R300) or 20 (4D200-R20) left-singular vectors or eigenvectors are calculated. Since DSYEVX is independent of the number of randomized columns, they are straight lines in both plots. Note that the execution times are for the randomized SVD and DSYEVX functions *only*, whereas the error is calculated between the original tensor and the final decomposition using Equation 8.

We can make three key observations.

- Using our randomized SVD algorithm can achieve tensor



(a) 3D3000-R300 *(b)* 4D200-R20

Figure 8: Exec. time and approx. error scaling for our randomized SDV algorithm, with respect to the number of randomized columns.

approximation error (Equation 8) comparable to that of using DSYEVX when *enough* randomized columns are used.

- Using the recommended $2r$ randomized columns (i.e., 600 for 3D3000-R300, and 40 for 2D200-R20), our randomized SVD algorithm achieves a 4.2× speedup for the 3000×3000 matrix (3D3000-R300) and 9.8× speedup for the 200×200 matrix (4D200-R20) over the DSYEVX function (also calculating *only* the necessary eigenvectors).
- We can use fewer than the recommended number of randomized columns to achieve comparable accuracy and observe even higher speedups .

We see that the accuracy starts to converge at 400 and 32 randomized columns for the 3D3000-R300 and 4D200-R20 data sets, respectively. These are approximately $1.5r$ randomized columns, and they yield an even higher speedups of 6.9× and 12.7×. This suggests that there is a potential for accuracy vs. performance trade-off. However, we leave this for future studies.

E. Multi-node performance

Table IV shows the achieved speedup and parallel efficiency of our GPU and CPU **Reuse + RndSVD** implementations on 64 nodes using four GPUs and two CPUs per node. We can see from the table that, for the CPU implementation, we see good speedup and parallel efficiency on every data set except 4D200-R20, our smallest data set in terms of both total size and mode lengths. The poor speedup and parallel efficiency for 4D200-R20 are caused by the increase in communication time at 64 nodes, which results in an increase in the total execution time when going from 32 nodes to 64 nodes. This occurs due to the latency component dominating the communication time (Equation 4) for our slice block partitioning method on small data sets (i.e., large P and small J).

GPU implementation shows a different trend. As the mode length increases from 128 to 3000, the scalability becomes worse due to the eigenvalue calculation bottleneck. Even with our randomized SVD optimization, the percentage of time spent on the eigenvalue calculation increases from 5.2% on one node to 56% on 64 nodes for 3D3000-R300. However, for the 5D128-R16 data set, that percentage only ranges from 0.02% on one node to 1.05% on 64 nodes. The exception to this is 4D200-R20, which shows poor scaling due to the aforementioned issue of latency dominating the communication time.

Input	GPU		CPU	
	Parallel Eff.	Speedup	Parallel Eff.	Speedup
3D3000-R300	0.17	10.59	0.73	46.84
4D200-R20	0.15	9.79	0.34	21.69
4D400-R64	0.44	28.08	0.74	47.23
5D128-R16	0.72	45.93	0.67	43.03

Table IV: Achieved speedup and parallel efficiency on 64 nodes.

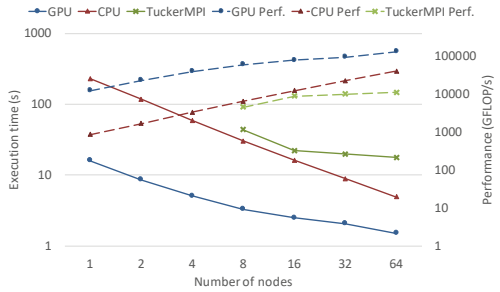
Figure 9 shows how our best CPU and GPU implementations (**Reuse + RndSVD**) strong scale up to 64 nodes, and how their execution time and performance compare against each other. When comparing the CPU implementation to the GPU implementation, we always achieve better performance on the GPU, even at 64 nodes. However, with enough nodes, this performance gap will likely close, since the GPU performance for small enough matrices will not be higher than the CPU performance.

Our GPU implementation for the *entire* ST-HOSVD calculation achieves as much as 59% and 10% of total system peak on one node and 64 nodes, respectively. This is lower than what we observed for *Gram_0* in Table III (over 94% of aggregate peak FLOP/s on four GPUs) and happens due to the tensor dimension lengths gradually decreasing as we traverse the modes, lowering the overall efficiency of the overall ST-HOSVD calculation, as well as due to increased communication with increased number of nodes.

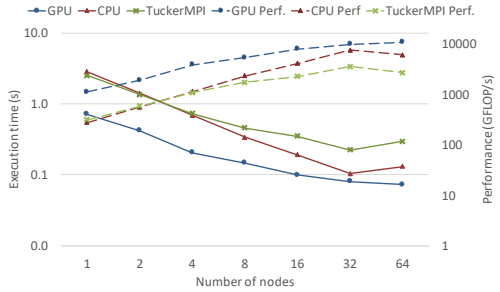
Table V shows a summary of **GPU** implementation execution times that compares our four implementation variations. On 64 nodes, our fastest implementation (**Reuse + RndSVD**) achieves up to 2.32× speedup over the baseline implementation (**NoReuse + DSYEVX**) for 3D3000-R300, and between 1.4× and 1.5× for the remaining sets. The implementation using distributed eigenvalue decomposition performs very poorly, as predicted.

Impl. ID	Nodes	Execution Time (s)				Nodes	Execution Time (s)			
		3D3000 -R300	4D200 -R20	4D400 -R64	5D128 -R16		3D3000 -R300	4D200 -R20	4D400 -R64	5D128 -R16
1	1	19.99	0.89	16.36	21.04	16	5.06	0.19	1.51	1.75
2		18.56	0.74	13.71	18.53		4.82	0.13	1.09	1.35
3		18.98	0.88	13.76	18.67		5.79	0.89	1.80	1.86
4		16.08	0.71	13.63	18.52		2.48	0.10	1.02	1.34
1	2	12.67	0.59	9.48	11.99	32	4.64	0.19	1.12	0.98
2		10.91	0.48	6.94	9.03		4.62	0.11	0.74	0.80
3		11.16	0.68	7.03	9.25		6.27	1.46	1.84	1.93
4		8.59	0.44	6.85	9.01		2.23	0.08	0.67	0.79
1	4	8.03	0.29	4.84	5.85	64	4.14	0.17	0.84	0.70
2		7.29	0.24	3.73	4.62		3.99	0.15	0.62	0.51
3		7.74	0.53	3.95	4.92		6.48	2.01	2.63	2.08
4		5.22	0.21	3.66	4.61		1.78	0.12	0.55	0.50
1	8	5.94	0.23	2.45	3.11					
2		5.56	0.19	1.94	2.45					
3		6.26	0.69	2.24	2.91					
4		3.42	0.16	1.88	2.44					

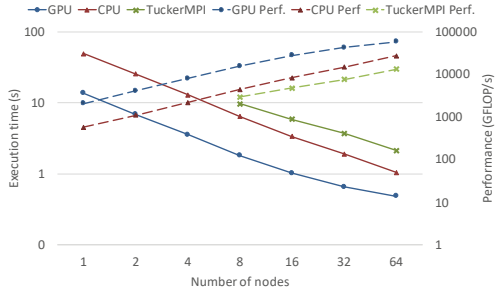
Table V: Summary of exec. times for our GPU implementations.



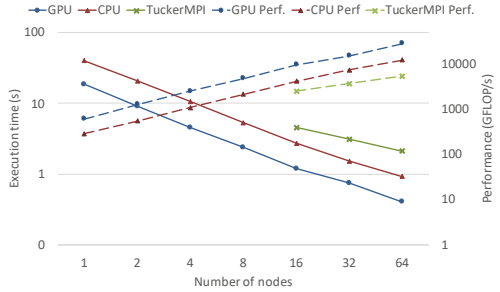
(a) 3D3000-R300



(b) 4D200-R20



(c) 4D400-R64



(d) 5D128-R16

Figure 9: Execution time and performance scaling for TuckerMPI and our CPU and GPU implementations on 64 nodes.

F. Comparsion against state-of-the-art

As far as we are aware, ours is the first GPU implementation of the STHOSVD method. Therefore, we are only able to compare against a state-of-the-art CPU implementation for STHOSVD - TuckerMPI [3]. Figure 9 shows how our CPU and GPU performance compares against that of TuckerMPI.

There two things to note. First, some of the execution times are missing for TuckerMPI: execution time on nodes 1 – 4 for 3D3000-R300 and 4D400-R64, and 1 – 8 for 5D128-R16. We were unable to run TuckerMPI on fewer nodes because the file I/O function used in the library was unable to access more than 2.1 GB of data per MPI rank; the program printed a warning and then crashed when more than 2.1 GB was accessed. Unfortunately, assigning too many MPI ranks per node degraded performance severely and produced misleading execution times. Therefore, they were omitted from the comparison.

Secondly, TuckerMPI uses the *dsyev* function which calculates every eigenvector, rather than using the *dsyevx* function that only calculates the first R eigenvectors, as is done in our CPU implementation. As such, the execution time was *significantly* higher for TuckerMPI, particularly for the data set 3D3000-R300. To make the comparison as fair as possible, for the data shown in Figure 9, we subtracted the original eigenvector execution times from the TuckerMPI times, and then added our *dsyevx* time instead, lowering the overall execution times.

With these adjustments, It can be seen from Figure 9 that our CPU implementation and TuckerMPI demonstrate similar scaling trends. However, our implementation performs slightly better, with speedup ranging from 2.0× to 3.6× on 64 nodes. The difference in performance comes from our two optimizations - using randomized SVD instead of dsyevx, and our new tensor matricization layout.

Speedup from our GPU implementation over TuckerMPI is even greater, ranging from 4.1× to 11.8× on 64 nodes.

V. Conclusion

We have presented our implementation and performance analysis of distributed and GPU-accelerated Tucker decomposition for dense tensors. We proposed three optimizations: 1) the slice block partitioning method that improves performance for GPUs, 2) a new tensor matricization layout that allows us to reduce the number of all-reduce communication and matricization steps by half, and 3) an variation of the randomized SVD algorithm to overcome the eigenvector bottleneck that could not easily be solved with a distributed solution.

With these optimizations, we can beat the performance of the state-of-the-art TuckerMPI library running on a 64-node two-socket CPU cluster with just a single node equipped with four GPUs each (for data set 3D3000-R300), or compress a 275 GB tensor in 0.4 seconds using 64 nodes (for data set 5D128-R16). Additionally, our analysis identified critical bottlenecks for the ST-HOSVD algorithm in the context of GPU acceleration, which should provide insight into other tensor algorithms that share similar computational structure.

Acknowledgment

The authors would like to thank Shaden Smith and Tyler Simon for their useful feedback, and the reviewers for their constructive comments.

References

[1] L. Tucker, “Some mathematical notes on three-mode factor analysis,” *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[2] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar, "On optimizing distributed tucker decomposition for dense tensors," in *IPDPS'17: Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium*, May 2017.

[3] W. Austin, G. Ballard, and T. G. Kolda, "Parallel tensor compression for large-scale scientific data," in *IPDPS'16: Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium*, May 2016, pp. 912–922.

[4] J. C. Ho, J. Ghosh, and J. Sun, "Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: ACM, 2014, pp. 115–124. [Online]. Available: http://doi.acm.org/10.1145/2623330.2623658

[5] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *CoRR*, vol. abs/1511.06530, 2015.

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

[7] B. W. Bader and T. G. Kolda, "Efficient matlab computations with sparse and factored tensors," *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, vol. 30, no. 1, pp. 205–231, 2007.

[8] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 4, pp. 1253–1278, Mar. 2000. [Online]. Available: http://dx.doi.org/10.1137/S0895479896305696

[9] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen, "A new truncation strategy for the higher-order singular value decomposition," *SIAM J. Sci. Comput.*, vol. 34, no. 2, pp. 1027–1052, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1137/110836067

[10] R. Ballester-Ripoll and R. Pajarola, "Lossy volume compression using tucker truncation and thresholding," *Vis. Comput.*, vol. 32, no. 11, pp. 1433–1446, Nov. 2016. [Online]. Available: http://dx.doi.org/10.1007/s00371-015-1130-y

[11] A. Karami, M. Yazdi, and G. Mercier, "Compression of hyperspectral images using discerete wavelet transform and tucker decomposition," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 5, no. 2, pp. 444–450, April 2012.

[12] I. Perros, R. Chen, R. Vuduc, and J. Sun, "Sparse hierarchical tucker factorization and its application to healthcare," in *2015 IEEE International Conference on Data Mining*, Nov 2015, pp. 943–948.

[13] J. Sun, D. Tao, and C. Faloutsos, "Beyond streams and graphs: Dynamic tensor analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 374–383. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150445

[14] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc, "An input-adaptive and in-place approach to dense tensor-times-matrix multiply," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 76:1–76:12.

[15] S. Smith and G. Karypis, "Accelerating the tucker decomposition with compressed sparse tensors," in *European Conference on Parallel Processing*. Springer, 2017.

[16] O. Kaya and B. Uçar, "High performance parallel algorithms for the tucker decomposition of sparse tensors," in *2016 45th International Conference on Parallel Processing (ICPP)*, Aug 2016, pp. 103–112.

[17] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka, "Tensor contractions with extended blas kernels on cpu and gpu," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, Dec 2016, pp. 193–202.

[18] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 77:1–77:11. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807624

[19] F. Wang, C.-Q. Yang, Y.-F. Du, J. Chen, H.-Z. Yi, and W.-X. Xu, "Optimizing linpack benchmark on gpu-accelerated petascale supercomputer," *Journal of Computer Science and Technology*, vol. 26, no. 5, p. 854, Sep 2011. [Online]. Available: https://doi.org/10.1007/s11390-011-0184-1

[20] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," *30th IEEE International Parallel & Distributed Processing Symposium*, 2016.

[21] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[22] N. Halko, P. G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Review*, vol. 53, no. 2, pp. 217–288, 2011. [Online]. Available: https://doi.org/10.1137/090771806