Efficient and Scalable Computations with Sparse Tensors

Muthu Baskaran, Benoît Meister, Nicolas Vasilache and Richard Lethin
Reservoir Labs Inc.
New York, NY 10012
Email: {baskaran,meister,vasilache,lethin}@reservoir.com

***Abstract*—For applications that deal with large amounts of high dimensional multi-aspect data, it becomes natural to represent such data as tensors or multi-way arrays. Multi-linear algebraic computations such as tensor decompositions are performed for summarization and analysis of such data. Their use in real-world applications can span across domains such as signal processing, data mining, computer vision, and graph analysis. The major challenges with applying tensor decompositions in real-world applications are (1) dealing with *large-scale* high dimensional data and (2) dealing with *sparse* data.**

In this paper, we address these challenges in applying tensor decompositions in real data analytic applications. We describe new sparse tensor storage formats that provide storage benefits and are flexible and efficient for performing tensor computations. Further, we propose an optimization that improves data reuse and reduces redundant or unnecessary computations in tensor decomposition algorithms. Furthermore, we couple our data reuse optimization and the benefits of our sparse tensor storage formats to provide a memory-efficient scalable solution for handling large-scale sparse tensor computations. We demonstrate improved performance and address memory scalability using our techniques on both synthetic small data sets and large-scale sparse real data sets.

I. Introduction

Using linear algebraic techniques to find structures and properties of data in data analytics applications is well established, for example, application of linear algebraic formulations for solving graph analysis problems [1]. These techniques apply for two-dimensional data and require the data to be represented in terms of matrices. However real-world data are often multi-dimensional with multiple aspects. Tensors or multi-dimensional arrays are a natural fit to represent data with multiple aspects and dimensionality. Consequently, multi-linear algebra, a generalization of linear algebra to higher dimensions, is increasingly used in real-world applications for extracting and explaining the properties of multi-attribute data. Multi-linear algebraic computations such as tensor decompositions have applications in a range of domains such as signal processing, data mining, computer vision, numerical linear algebra, numerical analysis, graph analysis [2]. Two of the prominent tensor decompositions are CANDECOMP/PARAFAC (CP) and Tucker decompositions. These decompositions are popular in scientific domains and also in modern applications such as social network analysis, network traffic analysis and web search mining.

A major challenge in real-world applications is handling the sparsity of data, as real-world data are not only multi-dimensional but also the linkages between the multiple attributes of data have a sparse characteristic. Recently, algorithms for tensor decompositions that account for the sparsity of data have been proposed [3]. In this work, we propose new sparse tensor storage formats that provide storage benefits and are efficient for performing tensor computations. The first format is called "mode-generic" sparse format that is a generic representation of tensor to conveniently store sparse and semi-sparse tensors. The second format is called "mode-specific" sparse format that is a special form of the generic representation that is suitable to perform computations along a specific mode or dimension of the tensor and improve data locality in such computations.

Real-world application data are usually large and the large size of the data poses a significant challenge to the efficiency of the tensor computations. Large-scale tensor data also poses a threat of "blowing up" system memory during computation resulting in inability to finish the computation. Memory blowup problem is a phenomenon commonly observed by users in Tucker decomposition [4]. Handling memory consumption in large-scale tensor computations is very important not only for preventing potential memory overflows but also for improving the overall execution time. We propose a novel optimization that aims to improve data reuse and reduce redundant or unnecessary computations in a sequence of tensor operations in Tucker decomposition. We couple the data reuse optimization and the advantage of our sparse tensor formats to store semi-sparse tensors and come up with a memory-efficient scalable approach to handle large-scale sparse tensor computations. We demonstrate our efficiency to improve performance and address memory scalability using our techniques on both synthetic small data sets and large-scale sparse real data sets.

The contributions of our work are as follows:

- We propose new sparse tensor storage formats that efficiently store sparse and semi-sparse tensors in such a way that helps improve data locality in sparse tensor computations and reduce unnecessary memory storage in the process of large data computations.
- We present a novel data reuse optimization technique that reuses previously computed data and avoids redundant computation in a sequence of tensor operations.
- We propose memory efficient scalable optimizations that address memory blowup issues in large tensor computations and also help improve computation speedup.

The rest of the paper is organized as follows. We give a brief background on tensor operations and decomposition in Section II. We discuss about our new sparse tensor storage formats in Section III. We present our data reuse optimization in Tucker decomposition in Section IV. In Section V, we discuss the memory blowup problem in tensor decomposition and our memory efficient scalable optimizations. We demonstrate the efficiency of our techniques in Section VI. We conclude with a summary and a foreword to our future work in Section VII.

II. Tensor Background

In this Section, we discuss the definitions of some of the basic tensor operations and tensor decompositions. A tensor is a multi-dimensional array and the *order* of a tensor is the number of dimensions, also called as *modes*, of the tensor. An important tensor operation that is widely used in practice is the n-Mode matrix product. Two popular and prominent tensor decompositions are CANDECOMP/PARAFAC (CP) and Tucker decompositions.

a) n-Mode matrix product: The n-Mode matrix product of a tensor $\mathcal{X}$ of size $I_1 \times \cdots \times I_N$ with a matrix $\mathbf{A}$ of size $J \times I_n$ (denoted by $\mathcal{X} \times_n \mathbf{A}$) results in a tensor of size $I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$.

$$(\mathcal{X} \times_n \mathbf{A})_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 \ldots i_N} a_{j i_n}$$

b) CP Decomposition: The CP tensor factorization decomposes a tensor into a sum of component rank-one tensors (A N-way tensor is called a rank-one tensor if it can be expressed as an outer product of N vectors). The CP decomposition that factorizes an input tensor $\mathcal{X}$ of size $I_1 \times \cdots \times I_N$ into R components (with factor matrices $\mathbf{A}^{(1)} \ldots \mathbf{A}^{(N)}$ and weight vector λ) is of the form:

$$\mathcal{X} = \sum_{r=1}^{R} \lambda_r \mathbf{a}_r^{(1)} \circ \cdots \circ \mathbf{a}_r^{(N)}$$

where $\mathbf{a}_r^{(n)}$ represents the r^{th} column of the factor matrix $\mathbf{A}^{(n)}$ of size $I_n \times R$.

c) Tucker Decomposition: The Tucker decomposition decomposes a tensor into a core tensor multiplied by a matrix along each mode. The Tucker decomposition that factorizes an input tensor $\mathcal{X}$ of size $I_1 \times \cdots \times I_N$ into a core tensor $\mathcal{G}$ of size $R_1 \times \cdots \times R_N$ and factor matrices $\mathbf{A}^{(1)} \ldots \mathbf{A}^{(N)}$ (where each factor matrix $\mathbf{A}^{(n)}$ is of size $I_n \times R_n$) is of the form:

$$\mathcal{X} = \mathcal{G} \times_1 \mathbf{A}^{(1)T} \times_2 \cdots \times_N \mathbf{A}^{(N)T}$$

The widely used algorithm for computing Tucker decomposition is the higher-order orthogonal iteration (HOOI) method [5] (presented in Algorithm 1) which can be viewed as an higher-order singular value decomposition (SVD) for tensors. The HOOI method involves various basic computational kernels that include SVD and n-Mode matrix product.

Algorithm 1 Tucker-HOOI Algorithm [5]

```
repeat
    for $n = 1 \ldots N$ do
        $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{A}^{(1)T} \cdots \times_{n-1} \mathbf{A}^{(n-1)T} \times_{n+1} \mathbf{A}^{(n+1)T} \cdots \times_N \mathbf{A}^{(N)T}$
        $\mathbf{A}_n$ = $J_n$ leading left singular vectors of $\mathbf{Y}_n$
    end for
    $\mathcal{G} = \mathcal{Y} \times_N \mathbf{A}^{(N)T}$
until convergence
```

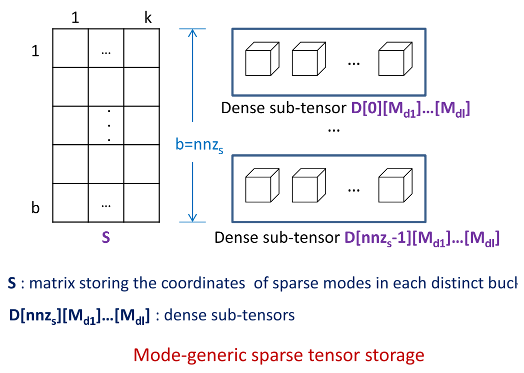
III. Sparse tensor storage formats

In this Section, we discuss the need for special sparse tensor storage formats to handle the sparsity in multi-dimensional tensor data and propose two new formats that help improve data locality in sparse tensor computations and reduce unnecessary memory consumption in large data computations.

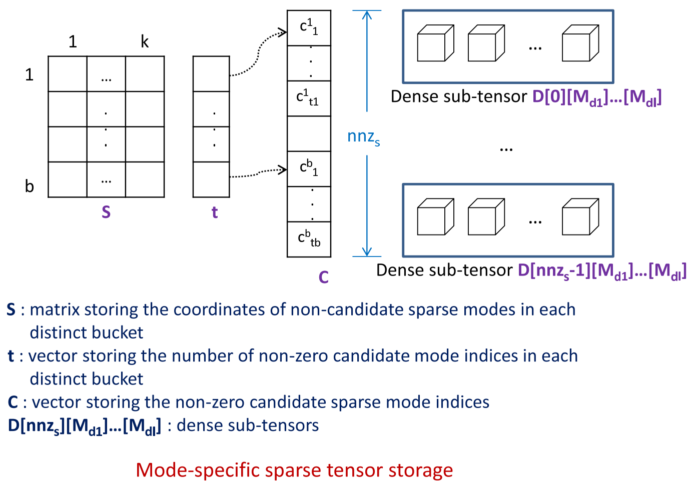
Sparse linear algebra primitives are widely used for two-dimensional data analysis. Techniques for optimizing and parallelizing key sparse linear algebraic primitives such as sparse matrix vector multiply, sparse matrix matrix multiply (SpGEMM) and the like are extensively studied and are available in literature [6], [7], [8]. However, for performance and storage reasons, it is not efficient to use any existing sparse matrix format to store sparse tensors and apply sparse matrix primitives to solve sparse tensor problems.

The common form of sparse tensor storage is the coordinate sparse tensor storage in which each non-zero is stored along with its index. While there are compressed sparse tensor storage formats like the Extended Karnaugh Map Representation (EKMR) [9] available in literature, Kolda et al. have expressed their opinion on the problems in compressed storage formats and have opted coordinate format in their work. However they have also acknowledged that the specialized storage formats such as the EKMR can be quite useful in many cases of (specific) tensor operations.

We first briefly discuss the motivation behind our new sparse tensor formats. If the non-zero values of a sparse tensor are stored in a random order of their indices, any tensor operation that is performed along a particular mode (mode-specific operation) would result in very poor locality with respect to accumulating the results of the operation in the output. This is because different non-zeros in the input tensor may contribute to the same element of the output and a random order of the indices may result in accessing the same element in different time instances that are far apart in the execution time-line (resulting in loss of locality). This motivated us to order the indices in a way that is suited for most tensor operations (most of which are mode-specific sequence of tensor operations). Of course, the problem is to choose the sort order of indices. There are $N!$ possible orderings for a N^{th} order tensor. Hence we limit our ordering to "outermost-to-innermost" mode or vice-versa. Our observation is that for any mode-specific operation, it would be beneficial if the storage format supports enumerating the "fibers" (sections of


1
k
1
b
S
b=nnz_s
Dense sub-tensor D[0][M_d1]…[M_dl]
Dense sub-tensor D[nnz_s-1][M_d1]…[M_dl]
S : matrix storing the coordinates of sparse modes in each distinct buck
D[nnz_s][M_d1]…[M_dl] : dense sub-tensors
Mode-generic sparse tensor storage

(a) Mode generic format


1
k
1
b
S
t
C
nnz_s
Dense sub-tensor D[0][M_d1]…[M_dl]
Dense sub-tensor D[nnz_s-1][M_d1]…[M_dl]
S : matrix storing the coordinates of non-candidate sparse modes in each distinct bucket
t : vector storing the number of non-zero candidate mode indices in each distinct bucket
C : vector storing the non-zero candidate sparse mode indices
D[nnz_s][M_d1]…[M_dl] : dense sub-tensors
Mode-specific sparse tensor storage

(b) Mode specific format

Fig. 1. Sparse tensor storage formats

a tensor obtained by fixing all but one index) along the mode efficiently.

In line with the above discussion, we come up with two new sparse tensor storage formats, namely, - (1) *mode-generic* sparse tensor storage format and (2) *mode-specific* sparse tensor storage format. These formats store non-zeros in an orderly fashion. The generic format is suited for any tensor operation and the specific format is suited for mode-specific tensor operations. The generic and specific formats take into account that some of the modes of the tensor may be dense (for e.g. tensors resulting from a sparse tensor dense matrix product) and hence are capable to efficiently represent dense sub-tensors within a sparse tensor. The generic format is the same as the coordinate format when all modes are sparse and is the same as the dense tensor storage when all modes are dense.

A. *Mode-generic sparse storage format*

In this section, we explain the structure of our mode-generic sparse tensor storage (shown in Fig. 1(a)). The highlights of this format are:

- sparse and dense modes are separated (a set of modes $n_1, \ldots, n_k$ is categorized as dense if the sub-tensor formed with modes $n_1, \ldots, n_k$ is a dense tensor).
- sparse mode indices are ordered from "outermost-to-innermost" or vice-versa.
- storage is as follows for a N^{th} order tensor of size $M_1 \times \cdots \times M_N$:
  1) k : scalar that stores the number of sparse modes
  2) b : scalar that stores the number of distinct sparse mode index tuples
  3) $d : l \times 1$ vector storing the list of dense modes (where $l = N - k$)
  4) $S : b \times k$ matrix storing the coordinates of the k sparse modes in each of the b distinct buckets
  5) $D : b \times M_{d_1} \times \cdots \times M_{d_l}$ dense sub-tensors

B. *Mode-specific sparse storage format*

In this section, we explain the structure of our mode-specific sparse tensor storage (shown in Fig. 1(b)). The highlights of this format are:

- sparse and dense modes are separated
- sparse mode indices are ordered from "outermost-to-innermost" or vice-versa.
- one of the sparse modes is a "candidate" mode
- storage is as follows for a N^{th} order tensor of size $M_1 \times \cdots \times M_N$:
  1) k : scalar that stores the number of sparse modes excluding the candidate mode
  2) b : scalar that stores the number of distinct sparse mode index tuples
  3) $d : l \times 1$ vector storing the list of dense modes (where $l = N - k - 1$)
  4) $S : b \times k$ matrix storing the coordinates of the k non-candidate sparse modes in each of the b buckets
  5) $t : b \times 1$ vector storing the number of non-zero candidate mode indices in each bucket
  6) $C : nnz_s \times 1$ vector storing the non-zero candidate mode indices (where $nnz_s = \sum_{i=1}^{b} t[i]$)
  7) $D : nnz_s \times M_{d_1} \times \cdots \times M_{d_l}$ dense sub-tensors

IV. Optimization for Data Reuse in Tensor Computations

We discuss a novel high-level optimization that is designed to avoid unnecessary (and costly) computations and efficiently reuse previously computed data in Tucker decomposition. The theoretical benefits from this optimization will be clear from the discussion of the optimization presented below. The practical benefits are substantiated through experimental evaluation presented in Section VI.

The Tucker-HOOI algorithm presented in Algorithm 1 has a common tensor operation, namely, sequence of tensor matrix products along *all but one* modes of a tensor.

$$\mathcal{Y} = \mathcal{X} \times_1 \mathbf{A}^{(1)T} \cdots \times_{n-1} \mathbf{A}^{(n-1)T} \times_{n+1} \mathbf{A}^{(n+1)T} \cdots \times_N \mathbf{A}^{(N)T}$$

Each $\times_n$ in the computation indicates a n-Mode matrix product (multiplying a tensor with a matrix along the nth mode of the tensor). As it can be seen from Algorithm 1, there is a sequence of $N-1$ n-Mode product computations (N being the order of the tensor) within each iteration of the for loop. Each product in the sequence produces an intermediate tensor that is used to compute the next product in the sequence. A major challenge is to reduce the storage of such intermediate tensors and increase the opportunity to reuse the intermediate data for memory scalability and performance improvement. Some of the intermediate tensors produced within one iteration of the for loop can be reused in subsequent iterations of the for loop. Exploiting such data reuse needs proper scheduling of the operations in the computation.

Furthermore, a slight re-write of the above computation offers opportunities for an improved data reuse (more reuse than what could be exploited in the original form). We utilize the fact that for distinct modes in a sequence of n-Mode products, the order of multiplication is irrelevant [10].

Our approach does the following modification to the computation. The for loop goes over the modes of the tensor in a particular order and let us assume that the modes are numbered from 1 to N in that order. We split the for loop into two halves. In the first half, the sequence of n-Mode matrix products are performed in the decreasing order of modes and in the second half, the sequence of n-Mode products are performed in the increasing (original) order. By doing so, the number of reuses of the intermediate tensors produced are increased. The modified computation looks like:

```
repeat
  for $n = 1 \dots \lceil \frac{N}{2} \rceil$ do
    $\mathcal{Y} = \mathcal{X} \times_N \mathbf{A}^{(N)T} \cdots \times_{n+1} \mathbf{A}^{(n+1)T} \times_{n-1} \mathbf{A}^{(n-1)T} \cdots \times_1 \mathbf{A}^{(1)T}$
    $\mathbf{A}_n$ = $J_n$ leading left singular vectors of $\mathbf{Y}_n$
  end for
  for $n = \lceil \frac{N}{2} \rceil + 1 \dots N$ do
    $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{A}^{(1)T} \cdots \times_{n-1} \mathbf{A}^{(n-1)T} \times_{n+1} \mathbf{A}^{(n+1)T} \cdots \times_N \mathbf{A}^{(N)T}$
    $\mathbf{A}_n$ = $J_n$ leading left singular vectors of $\mathbf{Y}_n$
  end for
  $\mathcal{G} = \mathcal{Y} \times_N \mathbf{A}^{(N)T}$
until convergence
```

In the original form of computation, total number of reuses of intermediate tensors are: $\frac{N^2}{2} - \frac{3N}{2} + 1$. We refer to the reuse in the original form as the *partial data reuse* in our future discussion. However, in the modified form of computation, total number of reuses of intermediate tensors turn out to be: $\geq \frac{N^2}{2} - \frac{3N}{2} + \frac{(N-2)^2+1}{4} + 1$. We refer to the reuse in the modified form as the *optimal data reuse* henceforth.

V. Handling memory overflow problem in Tucker decomposition

Despite the popularity of Tucker decomposition, applying it on a large-scale sparse tensor is still a research problem. In spite of having enough memory to store the large input sparse tensor and the core output dense tensor, memory overflow occurs frequently in Tucker computation due to the need to store larger intermediate results. Kolda et al. call this problem as the *intermediate blowup* problem and propose Memory Efficient Tucker (MET) decomposition as a solution to the problem in their work [4].

A. Intermediate blowup problem

We first explain the intermediate blowup problem in Tucker decomposition. The computation of $\mathcal{Y}$ in Algorithm 1 (*all but one* n-Mode product sequence) has the possibility of memory blowup. Let us assume that the input tensor size is $I_1 \times \cdots \times I_N$ and the output tensor size is $R_1 \times \cdots \times R_N$. The state-of-the-art sparse tensor dense matrix product implementations store the result in a dense tensor. Hence in the *all but one* sequence computation, the result of the first n-Mode product is stored in a dense tensor of size $R_1 \times I_2 \times \cdots \times I_N$. Therefore there arises a need for a large storage to store the intermediate result and it may easily lead to memory overflow resulting in the intermediate blowup problem. It is possible to do the entire tensor matrix product sequence without storing any of the intermediate tensors. However it induces a huge overhead in terms of repeated computation of various intermediate values that would have been otherwise computed only once, if they are stored and used for computing the next product in the sequence. Therefore the issue boils down to handling the trade-off between doing redundant computations and using more memory for storing intermediate results.

Kolda et al. propose a memory-efficient solution to handle this problem. They divide the modes as *element-wise* and *standard*. The idea is to do redundant computations along element-wise modes and store intermediate results of reduced size. Suppose they choose, say, e modes as element-wise, then the intermediate tensors they store are $(N-e)$ mode tensors instead of N mode tensors and they do redundant computations along the e modes. The categorization of modes is done based on, what they call, *reduction ratio* which is defined as $\frac{I_m}{R_m}$ for $m \in \{1, \dots, N\} \setminus \{n\}$, where n is the mode that is skipped in the *all but one* sequence. They select a minimal number of modes (with the largest values of reduction ratio) that guarantees memory availability as element-wise modes.

B. Memory-efficient scalable optimizations

We discuss our approach to provide a memory-efficient scalable solution to the intermediate blowup problem. First, we propose the use of mode-generic and mode-specific sparse formats to store intermediate tensors. Our sparse formats (both mode-generic and mode-specific) are designed to store completely sparse tensors and also tensors that have dense sub-tensors. As discussed earlier, the result of a sparse tensor dense matrix computation need not be stored in a dense tensor but can be stored using our mode-generic sparse format. The intermediate result is more dense than the input sparse tensor, but it may not be completely dense. Hence our technique of using mode-generic and mode-specific sparse formats for intermediate tensors reduces the memory requirement and thereby (1) reduces the possibility of memory blowup and (2) provides opportunity to store more intermediate tensors and achieve more data reuse using the data reuse optimization discussed in Section IV.

Furthermore, to handle the memory blowup problem, we also divide the modes as element-wise and standard, but in a different way from that of Kolda's approach. We order the modes based on the degree of data reuse of the intermediate tensor resulting after the n-Mode product along the mode. We assume an "initial" order (say L) of the modes (the order in which the for loop runs in Algorithm 1). In Section IV, we discussed about splitting the for loop into two halves. To choose element-wise modes for the first half computation, we choose the reverse order of L and choose minimal number of modes in that order that guarantees memory availability as element-wise. For the second half computation, the same is repeated with the original order of L. When a set of modes is selected as element-wise, we perform the sequence of n-Mode products along the element-wise modes at once (doing redundant computations and without storing intermediate tensors) and store one intermediate tensor at the end of the "element-wise" computation. This intermediate tensor has the same number of modes as the input tensor but reduced in size along the element-wise modes. This approach ensures memory availability to store intermediate tensors and also does not perturb the application of data reuse optimization.

Therefore we frame the optimizations to not only address memory scalability but also improve computation time by reducing redundant computations through improved data reuse and in this way we stand out from Kolda's MET approach.

VI. Experiments

We carried out experiments to evaluate the following: (1) efficiency of the new sparse tensor formats, (2) how we handle memory blowup problem in Tucker decomposition, using synthetic and real data sets, and (3) benefits of data reuse optimization in achieving performance improvement. The synthetic sparse tensors used in the evaluation are generated using the sparse tensor generation capability in the MATLAB Tensor Toolkit [11]. We performed the experiments on a dual socket quad core system with Intel Xeon E5504 2GHz processor and 12GB DRAM.

A. *Testing sparse formats with synthetic data set*

We performed experiments to demonstrate the effectiveness of the new sparse formats and understand the overhead of writing code for computations with the new formats. We represented the sparse tensor in the following formats in our experiments - (1) coordinate sparse tensor format, (2) mode-generic sparse tensor format, (3) mode-specific sparse tensor format and (4) compressed sparse row (CSR) format (tensor being matricized).

We took a 4th order sparse tensor of size $32 \times 32 \times 32 \times 32$ with 2048 non zeros as input for our experiments. Our first experiment was to compare the sequential performance of sparse tensor dense matrix multiplication of the input sparse tensor with a 32×32 dense input matrix along mode 4. Our next experiment was to measure the sequential performance of a sequence of two tensor matrix multiplications along mode 4 followed by mode 3. The performance achieved is shown in Table I.

Format	One product (GFLOPS)	Two products (GFLOPS)
`Compressed Sparse Row`	0.0012	0.0012
`Coordinate`	0.1416	0.1492
`Mode-specific and Mode-generic`	0.3072	0.3273

TABLE I
Performance of tensor matrix multiplication

From the results in Table I, we can see that converting to sparse matrix and performing the sparse tensor dense matrix multiplication as sparse matrix dense matrix multiplication results in very poor performance. The performance when sparse tensors are represented in our new formats is more than 2x better than that when spare tensors are represented in coordinate format. It is to be noted that the input sparse tensor was stored in mode-generic format, then converted to mode-specific format before the computation, and the result was stored in mode-generic format. We included the time taken for conversion from generic to specific format in the timing of the kernel.

B. *Testing memory blowup handling capability and data reuse optimization*

We evaluated the memory consumption and the computation time taken for Tucker decomposition using various synthetic and real input tensor data sets. We specifically measured the *all but one* sequence of tensor matrix products (multiplying a tensor by a sequence of matrices along all but one modes) occurring in the Tucker decomposition as the different evaluation versions of implementation differed only in that computation.

We implemented the following versions of tensor matrix product sequence: (1) "standard" - standard way to compute tensor matrix product sequence i.e. computing and storing all intermediate results in standard (dense) form, (2) "elementwise" - storing no intermediate results and performing redundant computations, (3) "other MET" - best performing memory-efficient Tucker following Kolda's approach of selecting few modes as element-wise and few modes as standard, (4) "our MET" - our approach of selecting element-wise and standard modes, storing intermediate results in mode-generic sparse format and applying the data reuse optimization.

1) Evaluation using large real data set: We used a large real data set extracted from the Enron email database [12], similar to the one used by Kolda et al. in their work on MET, to evaluate our approach to handle memory scalability and performance issues in large-scale data. The data set has 4 modes - sender, receiver, date and keyword. The dimensionality is $1000 \times 1000 \times 1100 \times 200$, representing emails of 1000 users over 1100 days with 200 keywords. The tensor is very sparse with only 5.5 million non zeros (0.0025%) out of 220 billion possible elements. The dimensionality of output core tensor is fixed as $10 \times 10 \times 10 \times 10$. As mentioned above, we evaluated the *all but one* sequence of tensor matrix products occurring in the Tucker decomposition. Being a 4^{th} order tensor, there are 4 *all but one* sequence of tensor matrix products.

The "standard" way of computing an *all but one* sequence of tensor matrix products in the state-of-the-art approach needs the intermediate tensors to be stored in dense form and runs out of memory for the first intermediate tensor. However our approach using the mode-generic sparse format successfully handles the memory requirement and computes even the largest possible intermediate tensor. The memory efficient Tucker following Kolda's approach successfully executes without memory overflow only when 2 or more modes are handled element-wise and fails to execute otherwise. The best performing version of Kolda's MET approach turns out to be the one with 2 element-wise modes.

Version	Time (s)
`elementwise`	175.17
`other MET`	21.79
`our MET (partial data reuse)`	9.29
`our MET (optimal data reuse)`	7.12

TABLE II
PERFORMANCE OF SEQUENCE OF TENSOR MATRIX MULTIPLICATIONS ON LARGE REAL DATA SET

Our approach not only handles the memory scalability better than the other approaches but also achieves better computation speed. Table II clearly substantiates the superior performance of our approach. The optimal data reuse, as explained in Section IV, yields good performance improvement as it can be seen from the results in Table II. The version with optimal data reuse performs 1.3x better than the version with partial data reuse and 3x better than the version without data reuse.

We then parallelized the version of implementation using our approach (using OpenMP directives) and measured the performance. We observed scalable performance improvement as the number of executing processor cores increased. The time taken for execution on 1, 2, 4, and 8 processor cores was 7.12s, 6.25s, 3.80s, and 2.57s, respectively.

2) Evaluation using synthetic data sets: We further evaluated our techniques using few synthetic data sets that are representatives of some real data sets in terms of size and sparsity. We used the following input tensors - 1) 4-way tensor of size $28 \times 501 \times 24 \times 8$, with 26400 non-zeros and 2) 4-way tensor of size $1000 \times 1000 \times 200 \times 12$, with 686400 non-zeros.

Version	Tensor 1 Time (s)	Tensor 2 Time (s)
`elementwise`	0.751	23.47
`other MET`	0.065	2.04
`our MET (optimal data reuse)`	0.050	1.78

TABLE III
PERFORMANCE OF SEQUENCE OF TENSOR MATRIX MULTIPLICATIONS ON SYNTHETIC DATA SETS

Table III presents the results of the experiment. Our approach performed better than the other versions and also handled memory consumption better as our approach successfully stored all intermediate tensors necessary for reducing redundant computations and increasing data reuse.

VII. CONCLUSION

We have described new sparse tensor formats and scalable computation scheduling optimizations for algorithms that use multi-linear tensor algebraic formulations of analysis of data with multiple linkages, and demonstrated their efficiency on synthetic and real data.

While these optimizations could be implemented "by hand" to provide immediate benefit in graph optimization implementations of algorithms such as [2] and [3], we expect their major impact will be obtained when they are implemented automatically in the context of other automatic high-level optimizations, such as those that extract parallelism, improve locality, and perform other format improvements. In particular we note the availability of algorithms such as [13] implemented in our R-Stream compiler for the high-dimensional, imperfect, static control loop nests and high-dimensional arrays that characterize multi-linear algebraic operations at the semantic level. With such optimizations performed semantically, mode-specific format conversions will be applied in code generation along the lines of [14].

REFERENCES

[1] J. Kepner and J. Gilbert, Eds., *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.

[2] D. Dunlavy, T. Kolda, and W. P. Kegelmeyer, "Multilinear Algebra for Analyzing Data with Multiple Linkages," in *Graph Algorithms in the Language of Linear Algebra*, J. Kepner and J. Gilbert, Eds. SIAM, 2011.

[3] E. C. Chi and T. G. Kolda, "On Tensors, Sparsity, and Nonnegative Factorizations," arXiv:1112.2414 [math.NA], December 2011. [Online]. Available: http://arxiv.org/abs/1112.2414

[4] T. G. Kolda and J. Sun, "Scalable Tensor Decompositions for Multi-aspect Data Mining," in *ICDM 2008: Proceedings of the 8th IEEE International Conference on Data Mining*, December 2008, pp. 363–372.

[5] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "On the Best Rank-1 and Rank-(R1,R2,. . .,RN) Approximation of Higher-Order Tensors," *SIAM J. Matrix Anal. Appl.*, vol. 21, pp. 1324–1342, March 2000.

[6] F. G. Gustavson, "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition," *ACM Transactions on Mathematical Software*, vol. 4, no. 3, pp. 250–269, 1978.

[7] T. A. Davis, Ed., *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[8] A. Buluç and J. R. Gilbert, "Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments," *SIAM Journal of Scientific Computing (SISC)*, 2012.

[9] C.-Y. Lin, J.-S. Liu, and Y.-C. Chung, "Efficient Representation Scheme for Multidimensional Array Operations," *IEEE Transactions on Computers*, vol. 51, pp. 327–345, 2002.

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

[11] T. Kolda and B. Bader, "Tensor toolbox." [Online]. Available: http://www.sandia.gov/ tgkolda/TensorToolbox

[12] A. Fiore and J. Heer, "UC Berkeley Enron email analysis." [Online]. Available: http://bailando.sims.berkeley.edu/enron_email.html

[13] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin, "Joint Scheduling and Layout Optimization to Enable Multi-Level Vectorization," in *IMPACT-2: 2nd International Workshop on Polyhedral Compilation Techniques*, Paris, France, Jan. 2012.

[14] W. Pugh and T. Shpeisman, "SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations," in *LCPC*, 1998, pp. 213–229.