Memory-Efficient Parallel Computation of Tensor and Matrix Products for Big Tensor Decomposition

N. Ravindran*, N.D. Sidiropoulos*, S. Smith†, and G. Karypis†

* Dept. of ECE & DTC, † Dept. of CSci & DTC
University of Minnesota, Minneapolis

Asilomar Conf. on Signals, Systems, and Computers, Nov. 3-5, 2014

Outline

- Rank decomposition for tensors - PARAFAC/CANDECOMP (CP)
- ALS
- Computational bottleneck for big tensors:

  $\rightsquigarrow$ unfolded tensor data times Khatri-Rao matrix product

- Prior work
- Proposed memory- and computation-efficient algorithms
- Review recent randomized tensor compression results: identifiability, PARACOMP
- Memory- and computation-efficient algorithms for multi-way tensor compression
- Parallelization and high-performance computing optimization (underway)

Rank decomposition for tensors

- Sum of outer products

$$\underline{\mathbf{X}} = \sum_{f=1}^{F} \mathbf{a}_f \circ \mathbf{b}_f \circ \mathbf{c}_f \quad (\dagger)$$

- $\mathbf{A}$, $I \times F$ holding $\{\mathbf{a}_f\}_{f=1}^{F}$, $\mathbf{B}$, $J \times F$ holding $\{\mathbf{b}_f\}_{f=1}^{F}$, $\mathbf{C}$, $K \times F$ holding $\{\mathbf{c}_f\}_{f=1}^{F}$
- $\mathbf{X}(:,:,k) :=$ k-th $I \times J$ matrix "slice" of $\underline{\mathbf{X}}$
- $\mathbf{X}_{(3)}^T :=$ $IJ \times K$ matrix whose k-th column is $\mathrm{vec}(\mathbf{X}(:,:,k))$
- Similarly, $JK \times I$ matrix $\mathbf{X}_{(1)}^T$; and $IK \times J$ matrix $\mathbf{X}_{(2)}^T$
- Equivalent ways to write ($\dagger$):

$$\mathbf{X}_{(1)}^T = (\mathbf{C} \odot \mathbf{B})\mathbf{A}^T \Longleftrightarrow \mathbf{X}_{(2)}^T = (\mathbf{C} \odot \mathbf{A})\mathbf{B}^T \Longleftrightarrow$$

$$\mathbf{X}_{(3)}^T = (\mathbf{B} \odot \mathbf{A})\mathbf{C}^T \Longleftrightarrow \mathrm{vec}(\mathbf{X}_{(3)}^T) = (\mathbf{C} \odot \mathbf{B} \odot \mathbf{A})\mathbf{1}$$

Alternating Least Squares (ALS)

- Multilinear least squares

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} ||\mathbf{X}_{(1)} - (\mathbf{C} \odot \mathbf{B})\mathbf{A}^T||_F^2$$

- Nonconvex, in fact NP-hard even for $F = 1$. Alternating least squares using

$$\mathbf{X}_{(1)}^T = (\mathbf{C} \odot \mathbf{B})\mathbf{A}^T \Longleftrightarrow \mathbf{X}_{(2)}^T = (\mathbf{C} \odot \mathbf{A})\mathbf{B}^T \Longleftrightarrow \mathbf{X}_{(3)}^T = (\mathbf{B} \odot \mathbf{A})\mathbf{C}^T$$

- Reduces to

$$\mathbf{C} = \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})(\mathbf{B}^T\mathbf{B} * \mathbf{A}^T\mathbf{A})^\dagger$$

$$\mathbf{A} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger$$

$$\mathbf{B} = \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})(\mathbf{C}^T\mathbf{C} * \mathbf{A}^T\mathbf{A})^\dagger$$

Core computation

- Computation and inversion of $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})$ relatively easy: relatively small $K \times F$, $J \times F$ matrices, invert $F \times F$ matrix, usually F is small
- Direct computation of, say, $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$, requires $O(JKF)$ memory to store $\mathbf{C} \odot \mathbf{B}$, in addition to $O(\mathrm{NNZ})$ memory to store the tensor data, where NNZ is the number of non-zero elements in the tensor $\underline{\mathbf{X}}$
- Further, JKF flops are required to compute $(\mathbf{C} \odot \mathbf{B})$, and $JKF + 2F\,\mathrm{NNZ}$ flops to compute its product with $\mathbf{X}_{(1)}$
- Bottleneck is computing $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$; likewise $\mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})$, $\mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})$
- Entire $\underline{\mathbf{X}}$ needs to be accessed for each computation in each ALS iteration, incurring large data transport costs
- Memory access pattern of the tensor data is different for the three computations, making efficient block caching very difficult
- 'Solution': replicate data three times in main (fast) memory :-(

Prior work

- *Tensor Toolbox* [Kolda *etal*, 2008-] has explicit support for sparse tensors, avoids intermediate data explosion by 'accumulating' tensor-matrix operations
- Computes $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ with $3F\ \mathrm{NNZ}$ flops using NNZ intermediate memory (on top of that required to store the tensor). Does not provision for efficient parallelization (accumulation step must be performed serially)
- Kang *et al*, 2012, computes $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ with $5F\ \mathrm{NNZ}$ flops using $O(\max(J + \mathrm{NNZ}, K + \mathrm{NNZ}))$ intermediate memory; in return, it admits parallel MapReduce implementation
- Room for considerable improvements in terms of memory- and computation-efficiency, esp. for high-performance parallel computing architectures

Our first contribution: Suite of three agorithms

- Algorithm 1: **Output:** $\mathbf{M}_1 \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) \in \mathbb{R}^{I \times F}$
  1: $\mathbf{M}_1 \leftarrow \mathbf{0}$
  2: **for** $k = 1, \ldots, K$ **do**
  3: $\quad \mathbf{M}_1 \leftarrow \mathbf{M}_1 + \underline{\mathbf{X}}(:,:,k)\mathbf{B}\,\mathrm{diag}(\mathbf{C}(k,:))$
  4: **end for**
- Algorithm 2: **Output:** $\mathbf{M}_2 \leftarrow \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A}) \in \mathbb{R}^{J \times F}$
  1: $\mathbf{M}_2 \leftarrow \mathbf{0}$
  2: **for** $k = 1, \ldots, K$ **do**
  3: $\quad \mathbf{M}_2 \leftarrow \mathbf{M}_2 + \underline{\mathbf{X}}(:,:,k)\mathbf{A}\mathrm{diag}(\mathbf{C}(k,:))$
  4: **end for**
- Algorithm 3: **Output:** $\mathbf{M}_3 \leftarrow \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A}) \in \mathbb{R}^{K \times F}$
  1: **for** $k = 1, \ldots, K$ **do**
  2: $\quad \mathbf{M}_3(k,:) \leftarrow \mathbf{1}^T\,(\mathbf{A} * (\underline{\mathbf{X}}(:,:,k)\mathbf{B}))$
  3: **end for**

Features

- Essentially no additional intermediate memory needed - updates of **A**, **B** and **C** can be effectively performed in place
- Computational complexity savings relative to Kang *et al*, similar or better (depending on the pattern of nonzeros) than Kolda *et al*
- Algorithms 1, 2, 3 share the same tensor data access pattern - enabling efficient orderly block caching / pre-fetching if the tensor is stored in slower / serially read memory, without need for three-fold replication ($\rightarrow$ asymmetry between Algorithms 1, 2 and Algorithm 3)
- The loops can be parallelized across K threads, where each thread only requires access to an $I \times J$ slice of the tensor. This favors parallel computation and distributed storage

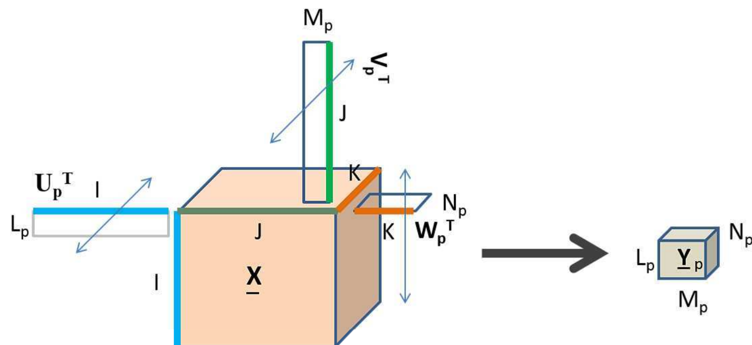
Computational complexity of Algorithm 1

- Algorithm 1: **Output:** $\mathbf{M}_1 \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) \in \mathbb{R}^{I \times F}$
  1: $\mathbf{M}_1 \leftarrow \mathbf{0}$
  2: **for** $k = 1, \ldots, K$ **do**
  3: $\quad \mathbf{M}_1 \leftarrow \mathbf{M}_1 + \underline{\mathbf{X}}(:,:,k)\mathbf{B}\,\mathrm{diag}(\mathbf{C}(k,:))$
  4: **end for**
- Let I_k be the number of non-empty rows and J_k be the number of non-empty columns in $\underline{\mathbf{X}}(:,:,k)$ and define $\mathrm{NNZ}_1 := \sum_{k=1}^{K} I_k$, $\mathrm{NNZ}_2 := \sum_{k=1}^{K} J_k$
- Assume that empty rows and columns of $\underline{\mathbf{X}}(:,:,k)$ can be identified offline and skipped during the matrix multiplication and update of $\mathbf{M}_1$ operations
- Note: only need to scale by $\mathrm{diag}(\mathbf{C}(k,:))$ those rows of $\mathbf{B}$ corresponding to nonempty columns of $\underline{\mathbf{X}}(:,:,k)$, and this can be done using FJ_k flops, for a total of $F\mathrm{NNZ}_2$

Computational complexity of Algorithm 1, continued

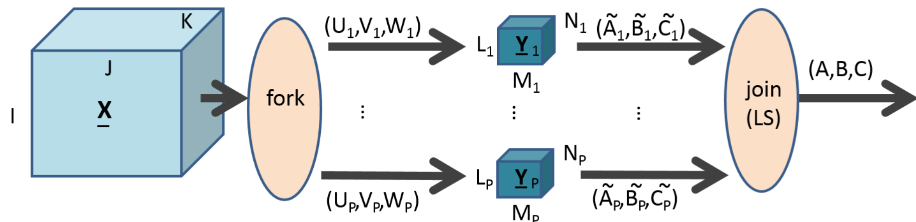
- Algorithm 1: **Output:** $\mathbf{M}_1 \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) \in \mathbb{R}^{I \times F}$
  1: $\mathbf{M}_1 \leftarrow \mathbf{0}$
  2: **for** $k = 1, \ldots, K$ **do**
  3: $\quad \mathbf{M}_1 \leftarrow \mathbf{M}_1 + \underline{\mathbf{X}}(:,:,k)\mathbf{B}\,\mathrm{diag}(\mathbf{C}(k,:))$
  4: **end for**
- Next, the multiplications $\underline{\mathbf{X}}(:,:,k)\mathbf{B}\,\mathrm{diag}(\mathbf{C}(k,:))$ can be carried out for all k at $2F\,\mathrm{NNZ}$ flops (counting additions and multiplications).
- Finally, only rows of $\mathbf{M}_1$ corresponding to nonzero rows of $\underline{\mathbf{X}}(:,:,k)$ need to be updated, and the cost of each row update is F, since $\underline{\mathbf{X}}(:,:,k)\mathbf{B}\,\mathrm{diag}(\mathbf{C}(k,:))$ has F columns; hence the total $\mathbf{M}_1$ row updates cost is $F\mathrm{NNZ}_1$ flops
- Overall $F\,\mathrm{NNZ}_1 + F\,\mathrm{NNZ}_2 + 2F\,\mathrm{NNZ}$ flops.
- Kang: $5F\mathrm{NNZ}$; Kolda $3F\mathrm{NNZ}$. Note $\mathrm{NNZ} > \mathrm{NNZ}_1, \mathrm{NNZ}_2$

Multi-way tensor compression



- Multiply (every slab of) $\underline{\mathbf{X}}$ from the I-mode with $\mathbf{U}^T$, from the J-mode with $\mathbf{V}^T$, and from the K-mode with $\mathbf{W}^T$, where $\mathbf{U}$ is $I \times L$, $\mathbf{V}$ is $J \times M$, and $\mathbf{W}$ is $K \times N$, with $L \leq I$, $M \leq J$, $N \leq K$ and $LMN \ll IJK$
- Sidiropoulos *et al*, IEEE SPL '12: if columns of $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are sparse, can recover LRT of big tensor from LRT of small tensor, under certain conditions

PARACOMP: PArallel RAndomly COMPressed Cubes



- Sidiropoulos *et al*, IEEE SPM Sep. '14 (SI on SP for Big Data)
- Guaranteed ID of big tensor LRT from small tensor LRTs, sparse or dense factors and data
- Distributed storage, naturally parallel, overall complexity/storage gains scale as $O\left(\frac{IJ}{F}\right)$, for $F \leq I \leq J \leq K$.

Multi-way tensor compression: Computational aspects

- Compressed tensor $\underline{\mathbf{Y}}_p \in \mathbb{R}^{L_p \times M_p \times N_p}$ can be computed as

$$\underline{\mathbf{Y}}_p(l, m, n) = \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} \mathbf{U}_p(l, i)\mathbf{V}_p(m, j)\mathbf{W}_p(n, k)\underline{\mathbf{X}}(i, j, k),$$

$$\forall l \in \{1, \ldots, L_p\}, m \in \{1, \ldots, M_p\}, n \in \{1, \ldots, N_p\}$$

- On the bright side, can be performed 'in place', can exploit sparsity by summing only over the non-zero elements of $\underline{\mathbf{X}}$
- On the other hand, complexity is $O(LMNIJK)$ for a dense tensor, and $O(LMN(\mathrm{NNZ}))$ for a sparse tensor
- Bad $\mathbf{U}_p, \mathbf{V}_p, \mathbf{W}_p$ memory access pattern (esp. for sparse $\underline{\mathbf{X}}$) can bog down computations

Multi-way tensor compression: Computational aspects

- Alternative computation schedule

$$\underline{\mathbf{T}}_1(l,j,k) = \sum_{i=1}^{I} \mathbf{U}(l,i)\underline{\mathbf{X}}(i,j,k),$$
$$\forall l \in \{1,\ldots,L_p\}, j \in \{1,\ldots,J\}, k \in \{1,\ldots,K\} \quad (1)$$

$$\underline{\mathbf{T}}_2(l,m,k) = \sum_{j=1}^{J} \mathbf{V}(m,j)\underline{\mathbf{T}}_1(l,j,k),$$
$$\forall l \in \{1,\ldots,L_p\}, m \in \{1,\ldots,M_p\}, k \in \{1,\ldots,K\} \quad (2)$$

$$\underline{\mathbf{Y}}(l,m,n) = \sum_{k=1}^{K} \mathbf{W}(n,k)\underline{\mathbf{T}}_2(l,m,k),$$
$$\forall l \in \{1,\ldots,L_p\}, m \in \{1,\ldots,M_p\}, n \in \{1,\ldots,N_p\} \quad (3)$$

Multi-way tensor compression: Computational aspects

- Complexity $O(LIJK + MLJK + NLMK)$ for dense tensor, as opposed to $O(LMNIJK)$
- Compressing the I mode first, followed by the J mode, and finally the K mode
- Lower complexity by storing intermediate results in tensors $\underline{\mathbf{T}}_1$ and $\underline{\mathbf{T}}_2$, instead of computing them multiple times
- For $I \leq J \leq K$ with $\frac{I}{L} = \frac{J}{M} = \frac{K}{N}$, it is advantageous to perform the multiplications in the order shown in (1)-(3), i.e., compress the smallest uncompressed mode first
- But ... very large intermediate memory, and sparsity is lost after the first multiplication - $\underline{\mathbf{T}}_1$ and $\underline{\mathbf{T}}_2$ are, in general, dense, potentially requiring even more memory than the original tensor!
- Tradeoff between complexity and memory savings, with or without sparsity

Our second contribution: Algorithm 4

- Algorithm 4: Compression of $\underline{\mathbf{X}}$ into $\underline{\mathbf{Y}}_p$

1: **for** $k = 1, \ldots, K$ **do**
2: $\quad \mathbf{T}_2' \leftarrow \mathbf{0}$
3: $\quad$ **for** $b = 1, B+1, 2B+1, 3B+1, \ldots, J$ **do**
4: $\quad\quad \mathbf{T}_1' \leftarrow \mathbf{U}_p^T \underline{\mathbf{X}}(:, b : (b+B-1), k)$
5: $\quad\quad \mathbf{T}_2' \leftarrow \mathbf{T}_2' + \mathbf{T}_1' \mathbf{V}_p(b : (b+B-1), :)$
6: $\quad$ **end for**
7: $\quad$ **for** $n = 1, \ldots, N_p$ **do**
8: $\quad\quad \underline{\mathbf{Y}}_p(:, :, n) \leftarrow \underline{\mathbf{Y}}_p(:, :, n) + W(n, k)\mathbf{T}_2'$
9: $\quad$ **end for**
10: **end for**

- Same flop count as three-step sequential mode-product approach for dense tensors, and better flop count than 'scalar computation' for sparse tensors, but requires only limited intermediate memory in the form of $\mathbf{T}_1' \in \mathbb{R}^{L \times B}$ and $\mathbf{T}_2' \in \mathbb{R}^{L \times M}$, for any choice of $B \leq J$
- Choice of B does not affect computational complexity, choose $B > 1$ for better cache utilization

How Algorithm 4 exploits sparsity

- Matrix multiplication in Step 4 can fully exploit sparsity in the tensor data
- If the empty columns of $\mathbf{T}_1$ can be identified, then these columns need not participate while updating $\mathbf{T}_2$ in Step 5.
- Hence, overall complexity for sparse data is $O(L(\mathrm{NNZ}) + LM(\mathrm{NNZ}_2) + LMNK)$. Note that 'scalar computation' has higher complexity $O(LMN(\mathrm{NNZ}))$, since $\mathrm{NNZ} > K, \mathrm{NNZ}_2$
- Hence, the above algorithm exploits sparsity better than the method of looping over all the non-zero elements of $\underline{\mathbf{X}}$
- However, despite this, it is clear sparsity is only fully exploited while compressing the first mode. Only fully empty columns after compressing the first mode can be exploited while compressing the second mode
- Tempting to believe that further optimizations of Algorithm 4 for sparse data are possible

Algorithm 4: Parallelization

- $\exists$ more than one way of parallelizing Algorithm 4.
- Parallelizing the P replicas over P threads requires that each thread accesses the entire tensor once.
- However, if the parallelization is done over the for loop in Step 1 in Algorithm 4, i.e., over as many as K threads, where each thread handles $p = 1, \ldots, P$, only a "slice" of the tensor data, $\underline{\mathbf{X}}(:, :, k)$, is required at each thread.
- Similar to Algorithms 1, 2, and 3, this architecture favors situations where the tensor data is stored in a distributed fashion, with only a portion of the data locally available to each thread.
- Algorithm 4 can be generalized to tensors with more than 3 modes: the computations in steps 4 and 5 can be used to compress a pair of modes at a time, while the computation in step 8 can be used for the last mode in the case of an odd number of modes. The main result that the intermediate memory required is typically less than the final result will continue to hold.

Web, papers, software, credits

- Nikos Sidiropoulos, UMN http://www.ece.umn.edu/~nikos/
  - Signal and Tensor Analytics Research (STAR) group https://sites.google.com/a/umn.edu/nikosgroup/home
- George Karypis, UMN http://glaros.dtc.umn.edu/gkhome/index.php
- Sponsor
  - NSF-NIH/BIGDATA: Big Tensor Mining: Theory, Scalable Algorithms and Applications, Nikos Sidiropoulos, George Karypis (UMN), Christos Faloutsos, Tom Mitchell (CMU), NSF IIS-1247489/1247632.