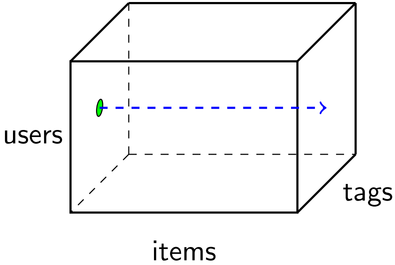
SPLATT
Efficient and Parallel Sparse Tensor-Matrix Multiplication

Shaden Smith[1] Niranjay Ravindran Nicholas D. Sidiropoulos
George Karypis

University of Minnesota
[1]shaden@cs.umn.edu

Tensor Introduction

Tensors are matrices extended to higher dimensions.



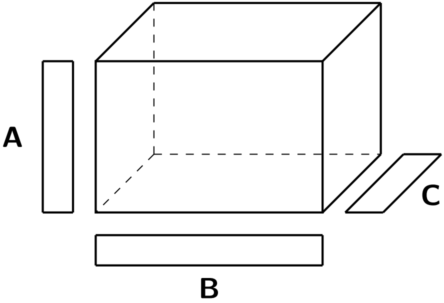
Example

We can model an item tagging system with a *user* $\times$ *item* $\times$ *tag* tensor.

- Very sparse!

Canonical Polyadic Decomposition (CPD)

- Extension of the singular value decomposition.
- Rank-F decomposition with $F \sim 10$
- Compute $\mathbf{A} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$, and $\mathbf{C} \in \mathbb{R}^{K \times F}$



Khatri-Rao Product

- Column-wise Kronecker product
- $(I \times F) \odot (J \times F) = (IJ \times F)$

$$\mathbf{A} \odot \mathbf{B} \quad = \quad [a_1 \otimes b_1, a_2 \otimes b_2, \ldots, a_n \otimes b_n]$$

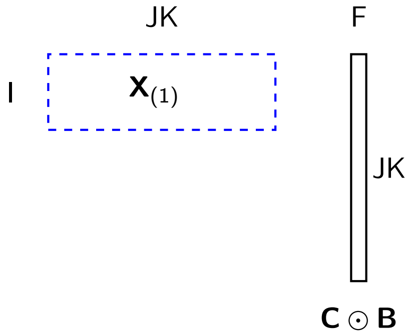
CPD with Alternating Least Squares

Computing the CPD

- We use alternating least squares.
- We operate on $\mathbf{X}_{(1)}$, the tensor flattened to a matrix along the first dimension.

$$\mathbf{A} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) \left(\mathbf{C}^{\intercal}\mathbf{C} * \mathbf{B}^{\intercal}\mathbf{B}\right)^{-1}$$

Matricized Tensor times Khatri-Rao Product (MTTKRP)

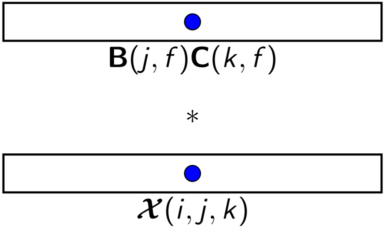


- MTTKRP is the bottleneck of CPD
- Explicitly forming $\mathbf{C} \odot \mathbf{B}$ is infeasible, so we do it *in place*.

Related Work

Related Work

Sparse Tensor-Vector Products



Tensor Toolbox

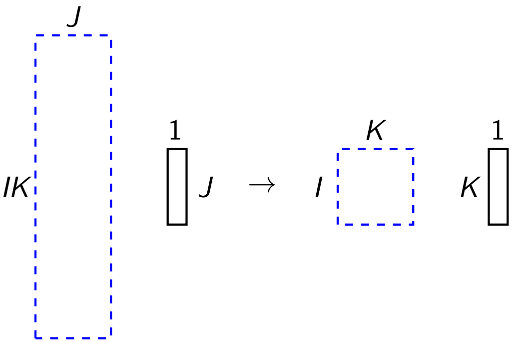
- Popular Matlab code today for sparse tensor work
- MTTKRP uses $nnz(\boldsymbol{\mathcal{X}})$ space and $3F \cdot nnz(\boldsymbol{\mathcal{X}})$ FLOPs
- Parallelism is difficult during “shrinking” stage

GigaTensor

$$\boxed{\mathbf{X}_{(1)}} \;*\; \boxed{\mathit{stretch}(\mathbf{B})} \;*\; \boxed{\mathit{stretch}(\mathbf{C})} \quad \mathbf{1}$$

- GigaTensor is a recent algorithm developed for Hadoop
- Uses $O(nnz(\mathcal{X}))$ space but $5F \cdot nnz(\mathcal{X})$ FLOPs
- Computes a column at a time

DFacTo



- Two sparse matrix-vector multiplications per column
- Requires an auxiliary sparse matrix with as many nonzeros as there are non-empty fibers
- $2F(nnz(\boldsymbol{\mathcal{X}}) + P)$ FLOPs, with P non-empty fibers

SPLATT

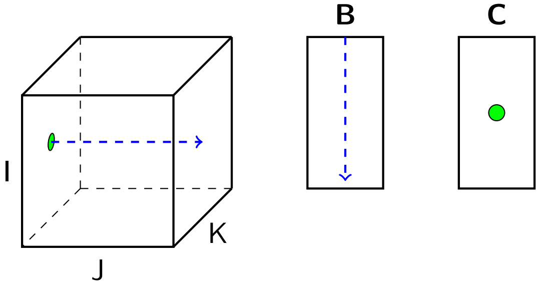
The **S**urprisingly **P**aralle**L** sp**A**rse **T**ensor **T**oolkit

Contributions

- Fast algorithm and data structure for MTTKRP
- Cache friendly tensor reordering
- Cache blocking for temporal locality

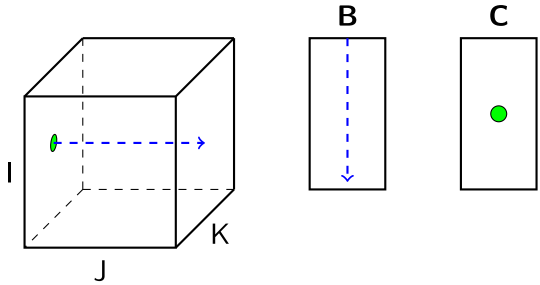
SPLATT– Optimized Algorithm

$$\mathbf{M}(i,f) = \sum_{k=1}^{K} \mathbf{C}(k,f) \sum_{j=1}^{J} \boldsymbol{\mathcal{X}}(i,j,k)\mathbf{B}(j,f)$$



$$\mathbf{M}(i,:) = \sum_{k=1}^{K} \mathbf{C}(k,:) * \sum_{j=1}^{J} \boldsymbol{\mathcal{X}}(i,j,k)\mathbf{B}(j,:)$$

SPLATT– Brief Analysis



- We compute rows at a time instead of columns
- Access patterns much better
- Same complexity as DFacTo!
- Only F extra memory for MTTKRP

Tensor Reordering

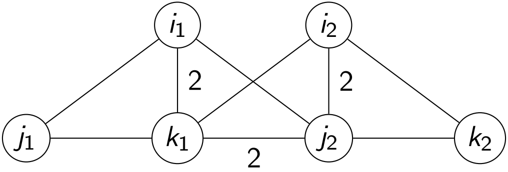
$$\left[\begin{array}{cccc|cccc|cccc} 0 & 3 & 0 & 3 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 2 & 0 & 0 & 2 \\ 0 & 3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

$$\left[\begin{array}{cccc|cccc|cccc} 3 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right]$$

We *reorder* the tensor to improve the access patterns of **B** and **C**

Tensor Reordering – Mode Independent

$$\left[\begin{array}{cc|cc} \alpha & \beta & 0 & 0 \\ 0 & \gamma & 0 & \delta \end{array}\right]$$

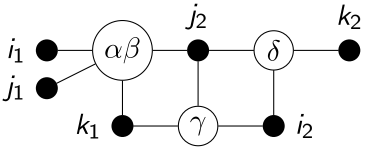


Graph Partitioning

- We model the sparsity structure of $\mathcal{X}$ with a tripartite graph
  - Slices are vertices, nonzeros connect slices with a triangle
- Partitioning the graph finds regions with shared indices
- We reorder the tensor to group indices in the same partition

Tensor Reordering – Mode Dependent

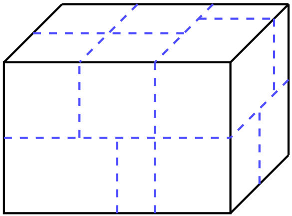
$$\left[\begin{array}{cc|cc} \alpha & \beta & 0 & 0 \\ 0 & \gamma & 0 & \delta \end{array}\right]$$



Hypergraph Partitioning

- Instead, create a new reordering for each mode of computation
- Fibers are now vertices and slices are hyperedges
- Overheads?

Cache Blocking over Tensors



Sparsity is Hard

- Tiling lets us schedule nonzeros to reuse indices already in cache
- Cost: more fibers
- Tensor sparsity forces us to *grow* tiles

Experimental Evaluation

Summary of Datasets

Dataset	I	J	K	nnz	density
NELL-2	15K	15K	30K	77M	1.3e-05
Netflix	480K	18K	2K	100M	5.4e-06
Delicious	532K	17M	2.5M	140M	6.1e-12
NELL-1	4M	4M	25M	144M	3.1e-13

Effects of Tensor Reordering

	Time (Speedup)		
Dataset	**Random**	**Mode-Independent**	**Mode-Dependent**
NELL-2	2.78	2.61 (1.06×)	2.60 (1.06×)
Netflix	6.02	5.26 (1.14×)	5.43 (1.10×)
Delicious	15.61	13.10 (1.19×)	12.51 (1.24×)
NELL-1	19.83	17.83 (1.11×)	17.55 (1.12×)

- Small effect on serial performance
- Without cache blocking, a dense fiber can hurt cache reuse

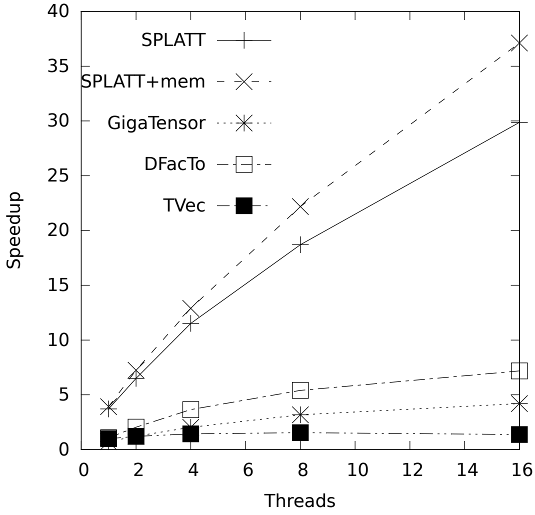
Effects of Cache Blocking

	Time (Speedup)			
Thds	**SPLATT**	**tiled**	**MI+tiled**	**MD+tiled**
1	8.14 (1.0×)	8.90 (0.9×)	8.70 (1.0×)	9.18 (0.9×)
2	4.73 (1.7×)	4.88 (1.7×)	4.37 (1.9×)	4.52 (1.8×)
4	2.54 (3.2×)	2.58 (3.2×)	2.29 (3.6×)	2.35 (3.5×)
8	1.42 (5.7×)	1.41 (5.8×)	1.26 (6.5×)	1.26 (6.4×)
16	0.90 (9.0×)	0.85 (9.5×)	0.74 (11.0×)	0.75 (10.8×)

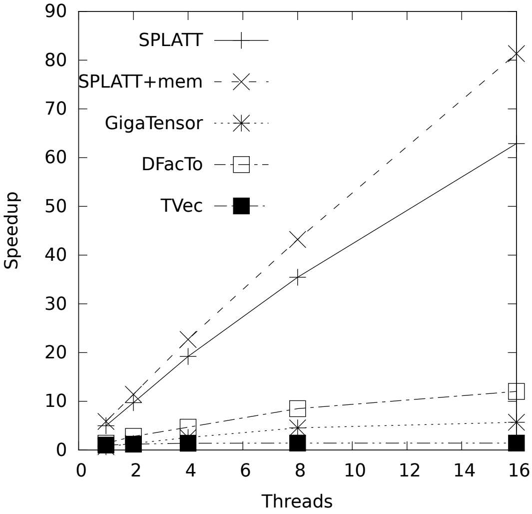
MI and **MD** are mode-independent and mode-dependent reorderings, respectively.

- Cache blocking on its own is also not enough
- MI and MD are very competitive with tiling enabled

Scaling: Average Speedup vs TVec



Scaling: NELL-2, Speedup vs TVec



Conclusions

Results

- SPLATT uses less memory than the state of the art
- Compared to DFacTo, we average 2.8$\times$ faster serially and 4.8$\times$ faster with 16 threads
- How?
  - Fast algorithm
  - Tensor reordering
  - Cache blocking

SPLATT

- Released as a C library
- `cs.umn.edu/~shaden/software/`