High-Performance Sparse Tensor Computations

Somesh Singh

Post-doctoral Researcher

Team ROMA, LIP

December 07, 2023

Professional History

PhD (+ Master's) [July 2014 – June 2021] | IIT Madras, India

Scalable and performant graph processing on GPU using approximate computing

Post-doctoral researcher [September 2021 – Present] | Team ROMA at LIP, ENS Lyon

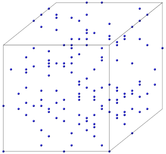
- High-performance sparse matrix and tensor computations
- Hashing-based methods

Tensors: A Recap

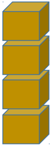
- Tensors are multi-dimensional arrays

1D Tensor / Vector

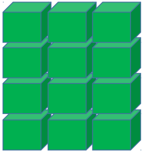
2D Tensor / Matrix



3D Tensor / Cube

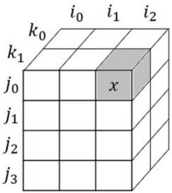


4D Tensor

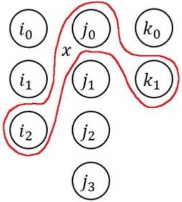


5D Tensor

- A *d*-dimensional sparse tensor corresponds to a special class of hypergraphs



3D Tensor



Hypergraph

Graphics Source: Google Images

Two Operations on Sparse Tensors

1 Querying for nonzeros in tensors

- Hyperedge queries in hypergraphs

2 Tensor contraction

- SpGETT: Sparse Tensor–Sparse Tensor Multiplication

Hyperedge Queries in Hypergraphs

The problem

Given: A d-dimensional sparse tensor $\mathcal{T}$

Goal: Answer queries of the form: "Is $\mathcal{T}[i_1, \ldots, i_d]$ zero or nonzero?"

Motivation

An algorithm for the **decomposition of (sparse) tensors** [Kolda, Hong 2020][§] in which this problem appears as a subproblem

- Sample the zeros and nonzeros of the tensor
- For sampling zeros: generate a random set of indices, and check those positions in the tensor for nonzeros

§ T. G. Kolda and D. Hong, "Stochastic gradients for large-scale tensor decomposition," SIAM Journal on Mathematics of Data Science 2(2020)

Hyperedge Queries in Hypergraphs

Characteristics of a desirable solution

- $O(d)$ query response time
- Small memory overhead
- Fast preprocessing

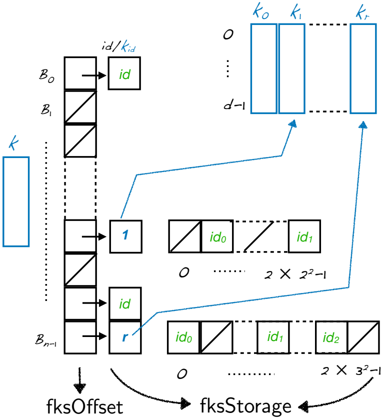
Our approach

Space-efficient hashing-based method with worst-case $O(1)$ lookup

FKSlean: The proposed method

- All the nonzeros are available at the start
- There are no duplicates
- Perfect hashing of a static set of nonzeros

FKSlean



FKSlean data structure

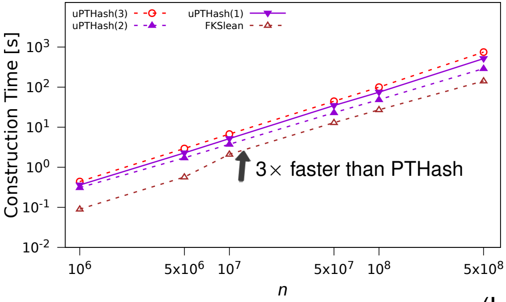
- A two-level structure
- First level hash function:
  $h(\mathbf{k}, \mathbf{x}, p, n) := (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$
- Second level hash function:
  $h(\mathbf{k}_i, \mathbf{x}, p, 2b_i^2) := (\mathbf{k}_i{}^T\mathbf{x} \bmod p) \bmod 2b_i^2$

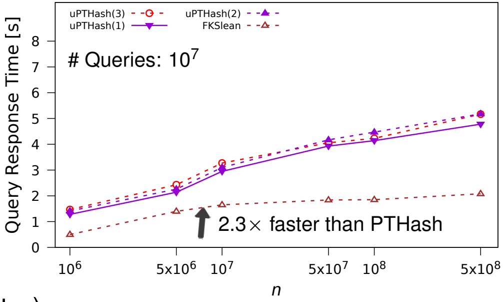
- $\mathbf{k}$, $\mathbf{k}_i$: random d-tuples
- n: number of nonzeros in tensor $\mathcal{T}$
- p: prime number $> n$
- b_i: number of nonzeros mapped to bucket B_i

J. Bertrand, F. Dufossé, **S. Singh** and B. Uçar, "Algorithms and Data Structures for Hyperedge Queries", ACM Journal of Experimental Algorithmics (JEA), vol. 27, no. 1, Article 1.13, 23 pages, 2022 [**ACM Results Reproduced Badge**]

FKSlean Results

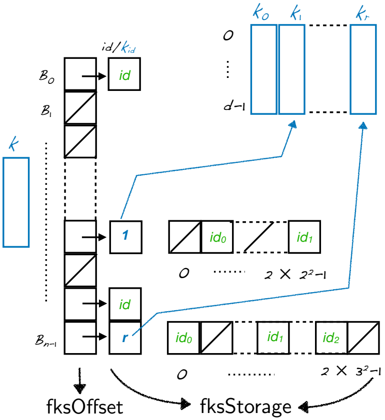
- Guaranteed constant time lookup per query in the worst-case
- Construction time is linear in the number of nonzeros, in expectation
- Total storage space less than 5n
- **Fastest** among all the competitors, including the **state-of-the-art PTHash**





(Lower is better)

PARFKSLEAN: Parallel FKSlean



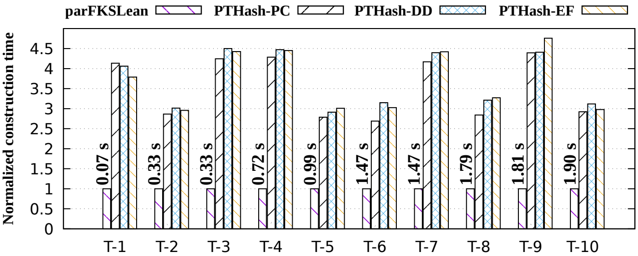
PARFKSLEAN data structure

- Parallelizes the construction and the query phase of FKSlean
- PARFKSLEAN retains the properties of FKSlean
- Parallel construction proceeds in two steps

  1: setup `fksOffset`, in parallel

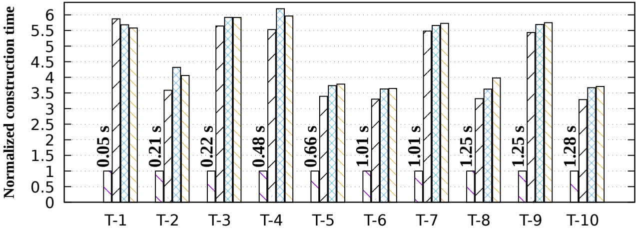
  2: populate `fksStorage`, in parallel

parFKSLean Results



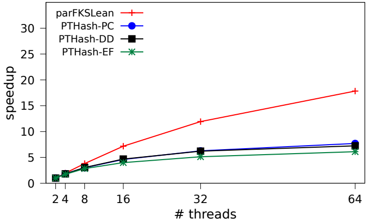
threads = 32

threads = 64

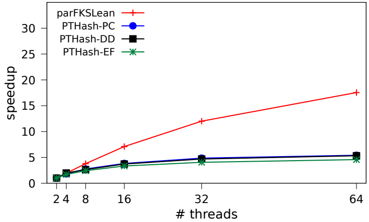


Construction of parFKSLean is **always** faster than PTHash for all thread counts

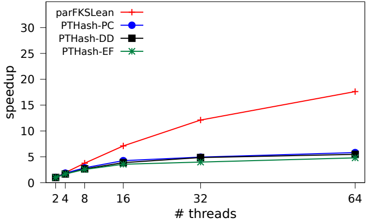
Scalability of Construction of PARFKSLEAN



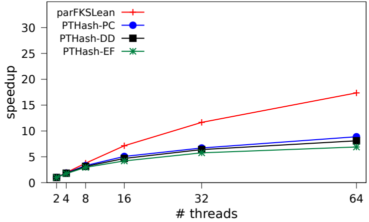
nell-2



flickr-4d



delicious-4d



nell-1

PARFKSLEAN exhibits good parallel scaling

Query Response Time of PARFKSLEAN

Tensor	#Threads	PTHash -PC	PTHash -DD	PTHash -EF	PARFKSLEAN
nell-2	2	2.01	1.64	2.10	0.97
	4	1.01	0.95	1.06	0.53
	8	0.46	0.49	0.54	0.27
	16	0.25	0.27	0.29	0.15
	32	0.14	0.15	0.16	0.11
	64	0.08	0.09	0.11	0.07
flickr-4d	2	2.51	2.04	2.20	1.07
	⋮				
	64	0.11	0.09	0.09	0.08
delicious-4d	2	2.30	2.02	2.25	1.11
	⋮				
	64	0.10	0.09	0.15	0.08
nell-1	⋮				
	64	0.11	0.10	0.08	0.08

Execution time (in seconds) for 10^7 queries on four large tensors

PARFKSLEAN is faster than or comparable to PTHash for all thread counts

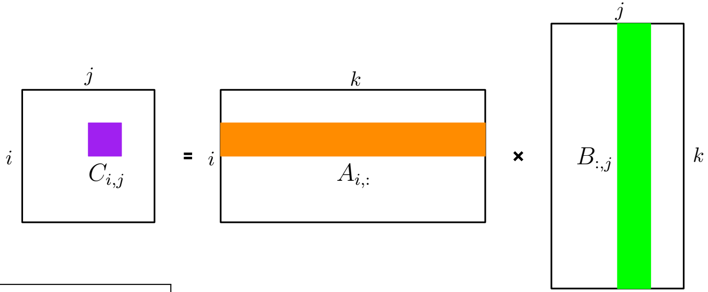
Conclusions (Part-I)

- We propose FKSlean and its parallel version PARFKSLEAN for answering hyperedge queries
- The construction phase of PARFKSLEAN exhibits good parallel scaling
- FKSlean and PARFKSLEAN both outperform the state-of-the-art in construction and query response time

Sparse Tensor Contraction (SpGETT)

- Tensor contraction is a higher-dimensional analog of matrix-matrix multiplication
- Multiplication of two matrices: $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{K \times J}$

$$\mathbf{C}_{ij} = \sum_{k} \mathbf{A}_{ik} . \mathbf{B}_{kj}$$

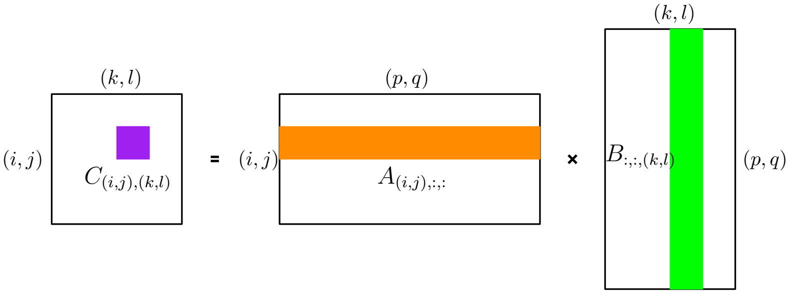


k is the *contraction* index

Sparse Tensor Contraction (SpGETT)

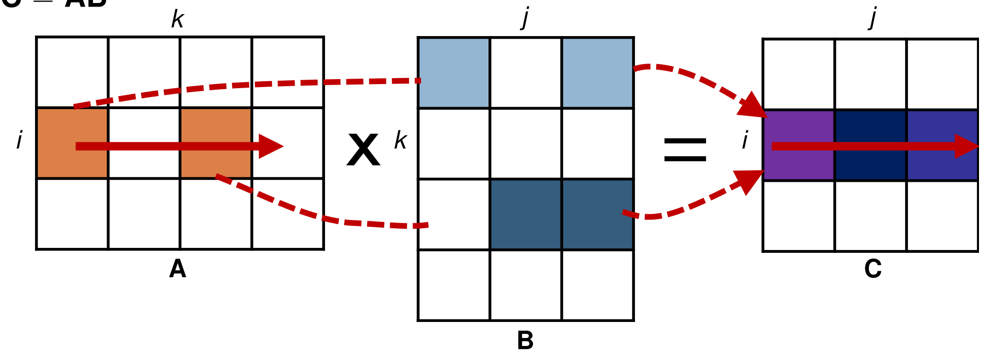
- Contraction of two tensors: $\mathcal{A} \in \mathbb{R}^{I \times J \times P \times Q}$ and $\mathcal{B} \in \mathbb{R}^{P \times Q \times K \times L}$ with p,q as contraction indices

$$\mathcal{C}_{ijkl} = \sum_{pq} \mathcal{A}_{ijpq} . \mathcal{B}_{pqkl}$$



Gustavson's algorithm for SpGEMM

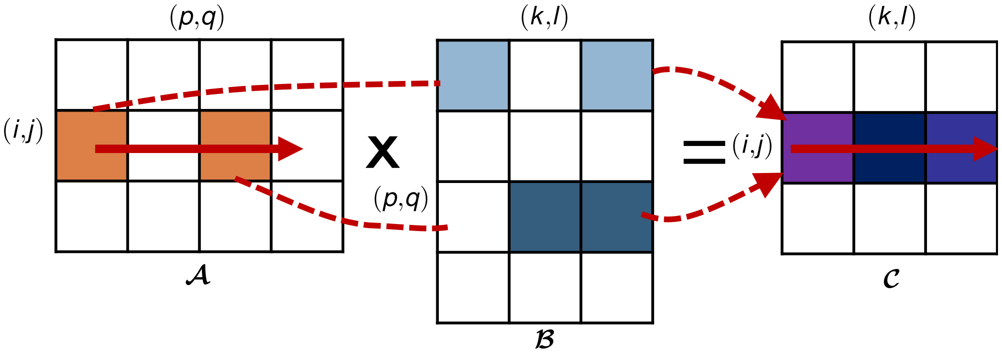
- $\mathbf{C} = \mathbf{AB}$



- Row-row formulation of SpGEMM
- $\mathbf{C}_{i,:} = \sum_k \mathbf{A}_{ik} . \mathbf{B}_{k,:}$

Gustavson's-like formulation for SpGETT

- $\mathcal{C} = \mathcal{A}\mathcal{B} \mid \mathcal{A} \in \mathbb{R}^{I\times J\times P\times Q}, \mathcal{B} \in \mathbb{R}^{P\times Q\times K\times L}$ | Contraction dimensions: P, Q

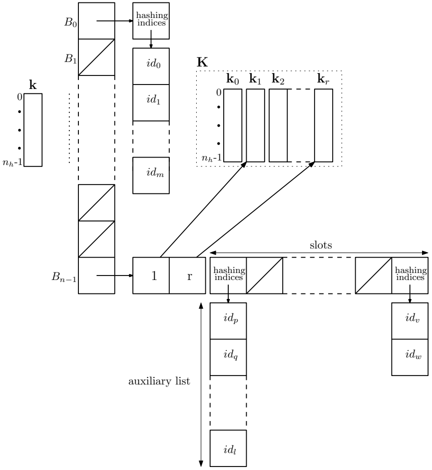


- Row-wise SpGETT
- $\mathcal{C}_{i,j,:,:} = \sum_{pq} \mathcal{A}_{ijpq} . \mathcal{B}_{p,q,:,:}$

FKSCuckoo: Dynamic perfect hashing

- Allows dynamic insertions
- There can be multiple items that share the same hashing indices
- Perfect hashing of a dynamic set of nonzeros

FKSCuckoo



FKSCuckoo data structure

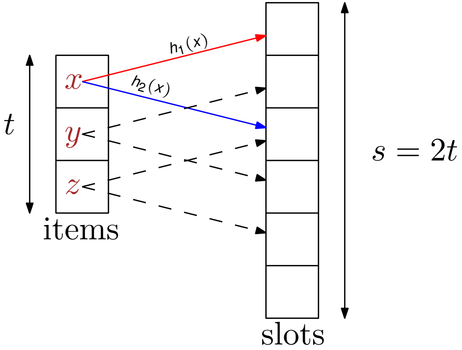
- A two-level structure
- $n_h \leqslant d$ hashing indices
- First level hash function:
  $h(\mathbf{k}, \mathbf{x}, p, n) := (\mathbf{k}^T \mathbf{x} \bmod p) \bmod n$
- Second level: Apply cuckoo hashing
- Items with the same hashing indices added to its auxiliary list

---

- $\mathbf{k}$: random n_h-tuple
- n: number of nonzeros in tensor $\mathcal{T}$
- p: prime number $> n$
- b_i: number of nonzeros mapped to bucket B_i

Cuckoo Hashing

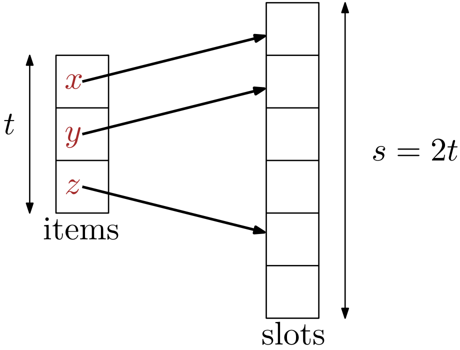
Cuckoo hashing is a perfect hashing approach with $O(1)$ lookup time in the worst case



- Construct a bipartite graph on items and slots
- The slots where an item, $\mathbf{x}$, can be placed:
  $h_1(\mathbf{x}) := (\mathbf{k}_1^T \mathbf{x} \bmod p) \bmod s$
  $h_2(\mathbf{x}) := (\mathbf{k}_2^T \mathbf{x} \bmod p) \bmod s$
- Find a perfect matching of items to slots, using a deterministic approach

Cuckoo Hashing

Cuckoo hashing is a perfect hashing approach with $O(1)$ lookup time in the worst case



- Construct a bipartite graph on items and slots
- The slots where an item, $\mathbf{x}$, can be placed:

$$h_1(\mathbf{x}) := (\mathbf{k}_1^T \mathbf{x} \bmod p) \bmod s$$
$$h_2(\mathbf{x}) := (\mathbf{k}_2^T \mathbf{x} \bmod p) \bmod s$$

- Find a perfect matching of items to slots, using a deterministic approach

SpGETT using FKSCuckoo

- Two 4D tensors: $\mathcal{A} \in \mathbb{R}^{I,J,P,Q}$ and $\mathcal{B} \in \mathbb{R}^{P,Q,K,L}$
- Tensor contraction $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ along dimensions P, Q

1. Create hash data structures for $\mathcal{A}$ and $\mathcal{B}$: $\mathcal{H}_{\mathcal{A}}$ and $\mathcal{H}_{\mathcal{B}}$ respectively
    - For $\mathcal{H}_{\mathcal{A}}$, hash $\mathcal{A}$ using (i,j)
    - For $\mathcal{H}_{\mathcal{B}}$, hash $\mathcal{B}$ using (p,q)

SpGETT using FKSCuckoo

- Two 4D tensors: $\mathcal{A} \in \mathbb{R}^{I,J,P,Q}$ and $\mathcal{B} \in \mathbb{R}^{P,Q,K,L}$
- Tensor contraction $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ along dimensions P, Q

2. To generate a slice $\mathcal{C}(i,j, :, :)$, pick a slot in $\mathcal{H}_{\mathcal{A}}$ for (i,j)
    - Pick each (p,q) in the auxiliary list of (i,j) in $\mathcal{H}_{\mathcal{A}}$
    - Find the (p,q) in $\mathcal{H}_{\mathcal{B}}$
    - Go over the nonzeros in the auxiliary list of (p,q) in $\mathcal{H}_{\mathcal{B}}$
    - Multiply and accumulate the partial products in $\mathcal{H}_{SPA}$ and write to $\mathcal{C}$ when the entire slice is populated

FKSCuckoo Results

- Operation considered for evaluation: $\mathcal{C} = \mathcal{A}\mathcal{A}^T$
- Baseline: Sparta, the state-of-the-art for SpGETT
- Sequential execution: Ours is $1.01\times$ to $1.20\times$ faster than Sparta on real-world tensors from FROSTT
- Parallel execution: Ours is $1.08\times$ to $1.25\times$ faster than Sparta on real-world tensors from FROSTT across thread counts of {16, 32, 48, 64, 80, 96}

Conclusions

- We propose a hashing-based method for SpGETT
- Our method outperforms the state-of-the-art both in the sequential and parallel setting on real-world tensors
- Hashing-based methods for sparse tensor computations are promising

Conclusions

- We propose a hashing-based method for SpGETT
- Our method outperforms the state-of-the-art both in the sequential and parallel setting on real-world tensors
- Hashing-based methods for sparse tensor computations are promising

Thank You

Backup Slides

Input Tensors

Tensor	d	Dimensions	n
chicago_crime (T-1)	4	6,186 × 24 × 77 × 32	5,330,673
vast-2015-mc1-3d (T-2)	3	165,427 × 11,374 × 2	26,021,854
vast-2015-mc1-5d (T-3)	5	165,427 × 11,374 × 2 × 100 × 89	26,021,945
enron (T-4)	4	6,066 × 5,699 × 244,268 × 1,176	54,202,099
nell-2 (T-5)	3	12,092 × 9,184 × 28,818	76,879,419
flickr-3d (T-6)	3	319,686 × 28,153,045 × 1,607,191	112,890,310
flickr-4d (T-7)	4	319,686 × 28,153,045 × 1,607,191 × 731	112,890,310
delicious-3d (T-8)	3	532,924 × 17,262,471 × 2,480,308	140,126,181
delicious-4d (T-9)	4	532,924 × 17,262,471 × 2,480,308 × 1,443	140,126,181
nell-1 (T-10)	3	2,902,330 × 2,143,368 × 25,495,389	143,599,552

Input tensors from FROSTT dataset