

Hashed Coordinate Storage of Sparse Tensors

Robert Lowe

rlowe@semo.edu

Southeast Missouri State University
Cape Girardeau, Missouri, USA

MeiLi Charles

jcharl12@vols.utk.edu

University of Tennessee
Knoxville, Tennessee, USA

Amritpreet Singh

asingh2s@semo.edu

Southeast Missouri State University
Cape Girardeau, Missouri, USA

ABSTRACT

Tensors, or n -way arrays, are becoming increasingly important in many fields. In recent applications, tensors are extremely sparse and have such a high degree that dense storage becomes intractable. To address this, several sparse storage formats have been proposed, with most formats being some variation of coordinate-based storage, or COO. These formats are created to optimize the Matricized-Tensor-Times-Khatri-Rao product (MTTKRP) operation. The current state of the art, HiCOO, provides efficient MTTKRP operations, however it is expensive to build and update. HiCOO requires an $O(n \lg n)$ operation to provide z-order sorting of the indices before building its block structure. Furthermore, updating a HiCOO tensor would require at least a restructuring of blocks. This paper presents a new format called Hashed Coordinate (HaCOO). HaCOO uses a low-collision hash function to associate tensor indices with values. HaCOO improves upon the state-of-the-art by providing $O(1)$ amortized time complexity for arbitrary value insertion, update, and retrieval with little memory overhead.

1 INTRODUCTION

Tensors are seeing increased use in data science and machine learning applications. This is because tensor encoding, and the algorithms made possible by this encoding, can capture latent features of the data as well as otherwise difficult to obtain semantic information [1, 3, 5, 8]. The most common algorithm in tensor analysis is tensor decomposition, which reveals underlying factors of a dataset. Common tensor decompositions include canonical polyadic decomposition (CPD), non-negative canonically polyadic decomposition (NNCPD), and the Tucker decomposition. For a complete treatment of the mathematics of tensor operations and these decompositions, see [5]. The most common of these operations are CPD and NNCPD, and the runtime complexity of both is dominated by the Matricized-Tensor-Times-Khatri-Rao product (MTTKRP) [5–7, 12].

In many recent applications, such as text mining, these tensors tend to be extremely sparse [1, 8]. Primary memory storage of these tensors in dense format is not feasible, even when the number of non-zero entries is well below the amount of available memory. For example, the tensor encoding of the Enron email dataset [10] from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [11] contains 54, 202, 099 non-zero values. In dense format, this tensor has just over 9.9×10^{15} entries. Assuming a 64-bit number, this would require in excess of 79 petabytes of storage. While storing the list of non-zero data points is relatively trivial, carrying out tensor analysis requires the additional information provided by the location of each value within the tensor.

Various formats have been proposed for storing and processing sparse tensors. The earliest format was the coordinate (COO) format, which simply stores a list of indices and values. COO is simple to

implement, while using very little storage space. This format has seen usage in several tool kits [2, 13], as well as being the de facto standard for disk storage in repositories such as FROSTT [11]. While easy to implement, COO does suffer from slow access time when retrieving arbitrary indices. Updates to COO also require this slow search in order to determine if an index is already present. This makes the calculation of tensor decompositions and online updates intractable when the number of non-zeroes is large. Several other formats have emerged which optimize the computation of MTTKRP, thereby speeding the computation of CPD and NNCPD. These formats include Flagged-COO (FCOO) [7], Compressed Sparse Fiber (CSF) [12], and Hierarchical Coordinate format (HiCOO) [6].

2 BACKGROUND

A tensor is a general multidimensional array. The order of a tensor, n , is the number of indices required to identify each element. Written in conventional notation, tensor $\mathcal{T} \in \mathbb{R}^{m_1 \times m_2 \times \dots \times m_n}$ is an n -mode tensor, which has n indices $t_{i_1, i_2, \dots, i_n}$ where $1 \leq i_x \leq m_x$ [5].

Dense storage of tensors is trivial to accomplish as a simple multidimensional array. Dense storage also provides the fastest insertion and retrieval times, with both operations having a time complexity of $O(1)$. Unfortunately, the space complexity of dense storage is the product of the tensor's modes. Moreover, algorithms meant to process dense tensors cannot take advantage of the sparsity of tensors and will waste time in multiplying or adding zeroes. The various sparse tensor storage formats, and their accompanying algorithms, improve upon dense storage in two principal ways: space efficiency, and computational time complexity. COO accomplishes this through storing only non-zero values along with their indices, and then only exploring the non-zero values during arithmetic operations. CSF optimizes storage and computation by using a storage format similar to compressed sparse row (CSR), and also includes a custom MTTKRP algorithm which is optimized for this format. FCOO uses a storage method which is designed for GPU acceleration in the computation of MTTKRP. HiCOO uses a format which is similar to the compressed sparse block (CSB) matrix format, and also includes an optimized MTTKRP algorithm.

3 HACOO STORAGE FORMAT

The HaCOO storage format consists of a separate-chaining hash table with a load factor of 0.70. Each hash entry contains the following fields:

- morton_index** - The Morton encoded index of the entry.
- value** - The value of the entry at the index.

To allow for fast modulus operations via bit-masking, the size of the hash table is always a power of two. The initial size is arbitrarily set to 128. If its optimal load factor is exceeded, it doubles in size and all existing entries are rehashed into the new table. If the number

Tensor	Modes	Non-Zeros	Collision Rate	Mean Probe Depth	Median Probe Depth	Mode Probe Depth	Max Probe Depth
uber	4	3,309,490	16.43%	1.20	1	1	7
nell-2	3	76,879,419	26.45%	1.36	1	1	11
enron	4	54,202,099	37.53%	1.60	1	1	37
nips	4	3,101,609	77.31%	4.41	4	4	29

Table 1: Hash Storage Results for Several FROSTT Tensors [11]

of non-zero entries, nnz , is known in advance the hash table's size can be computed as in Equation 1.

$$size = 2^{\lceil \lg nnz / 0.7 \rceil} \quad (1)$$

4 HASH FUNCTION

Algorithm 1 determines four values required in Algorithm 2: sx , sy , sz , and $mask$. These are driven by the size of the table, also known as the number of buckets ($nbuckets$). Because these values do not change unless the size of the hash table is changed, they are computed separately. The three shift values are computed such that each of the bits in the portion of the hash which will form the key have a roughly 0.25 probability of changing. This leads to a high degree of separation in the hash table, mitigating collisions.

The hash function, shown in Algorithm 2, has two main parts. The first step is to compute the Morton encoding [9] of the index. (The index being an n -length array of integers.) The result of this operation, assuming an integer of arbitrary length, is a unique integer representing the index's position on a z-order curve. This, however, is insufficient for hashing as the range of this integer far exceeds the capacity of the hash table. To arrive at the final hash, series of simple arithmetic and bit shift operations are performed. This step is inspired by Jenkin's one-at-a-time hash function [4]. The result of these operations is a hash value which is fairly evenly distributed over the size of the hash table. The final step of the algorithm is to extract k , the hashed index of this entry. This is performed by simple bitwise arithmetic with the mask.

Algorithm 1: Hash Values

Input: $nbuckets$

Output: $sx, sy, sz, mask$

```

1  $bits \leftarrow \lg nbuckets$ 
2  $sx \leftarrow \text{ceil}(bits / 8) - 1$ 
3  $sy \leftarrow 4 * sx - 1$ 
4  $sz \leftarrow \text{ceil}(bits / 2) - 1$ 
5  $mask \leftarrow \sim nbuckets$ 
```

5 EVALUATION AND ANALYSIS

Experiments were run by simulating the storage of a sparse tensor and plotting the distribution of entries within the hash table. Heatmap style plots reveal that entries are distributed among buckets relatively evenly throughout the entire hash table. This is further supported by the average probe depth results in Table 1, where the

Algorithm 2: Hash Algorithm

Input: $index$

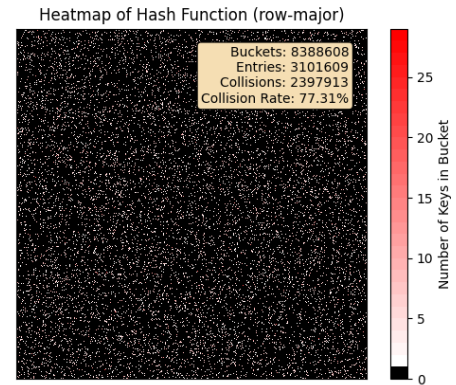
Output: m, k

```

1  $m \leftarrow \text{morton}(index)$ 
2  $hash \leftarrow hash + hash \ll sx$ 
3  $hash \leftarrow hash \oplus hash \gg sy$ 
4  $hash \leftarrow hash + hash \ll sz$ 
5  $k \leftarrow hash \& mask$ 
```

average probe depth is typically just over 1. In all cases the median and mode depth are the same as the mean probe depth.

Additionally, the nips tensor in Figure 1 serves as a near-worst case scenario. Even with a collision rate of 77.31% and maximum probe depth of 29, it still maintains an average probe depth of 4.41. These results show an expected near-constant $O(1)$ access time for the retrieval, storage, or deletion of an entry.

**Figure 1: Heatmap-style visual of the distribution of hashed entries.**

Because the traversal of the tensor is not sensitive to ordering, any sparse MTTKRP algorithm (such as the ones found in CSF and HiCOO), may be used. Also, because all access and write operations are performed in constant time, all other tensor operations can be performed efficiently using this format. That is to say, this data structure does not dominate the runtime complexity of any tensor operation.

REFERENCES

- [1] Brett W Bader, Michael W Berry, and Murray Browne. 2008. Discussion tracking in Enron email using PARAFAC. In *Survey of Text Mining II*. Springer, 147–163.
- [2] Brett W. Bader, Tamara G. Kolda, et al. 2015. MATLAB Tensor Toolbox Version 2.6. Available online. <http://www.sandia.gov/~tgkolda/TensorToolbox/>
- [3] Rasmus Bro. 1997. PARAFAC. Tutorial and applications. *Chemometrics and intelligent laboratory systems* 38, 2 (1997), 149–171.
- [4] Bob Jenkins. 2013. <https://www.drdoobs.com/database/algorithm-alley/184410284>
- [5] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.
- [6] J. Li, J. Sun, and R. Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 238–252. <https://doi.org/10.1109/SC.2018.00022>
- [7] Bangtian Liu, Chengyao Wen, Anand D Sarwate, and Maryam Mehri Dehnavi. 2017. A unified optimization approach for sparse tensor operations on gpus. In *2017 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 47–57.
- [8] Robert E Lowe and Michael W Berry. 2020. Using Non-negative Tensor Decomposition for Unsupervised Textual Influence Modeling. In *Supervised and Unsupervised Learning for Data Science*. Springer, 59–82.
- [9] Guy M Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).
- [10] Jitesh Shetty and Jafar Adibi. 2004. The Enron email dataset database schema and brief statistical report. *Information sciences institute technical report, University of Southern California* 4 (2004).
- [11] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. <http://frostdt.io/>
- [12] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. 1–7.
- [13] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. <https://doi.org/10.1038/s41592-019-0686-2>