Brief Announcement: Sparse Tensor Transpositions

Suzanne Mueller
MIT CSAIL
suzmue@csail.mit.edu

Peter Ahrens
MIT CSAIL
pahrens@csail.mit.edu

Stephen Chou
MIT CSAIL
s3chou@csail.mit.edu

Fredrik Kjolstad
Stanford University
kjolstad@stanford.edu

Saman Amarasinghe
MIT CSAIL
saman@csail.mit.edu

ABSTRACT

We present a new algorithm for transposing sparse tensors called Quesadilla. The algorithm converts the sparse tensor data structure to a list of coordinates and sorts it with a fast multi-pass radix algorithm that exploits knowledge of the requested transposition and the tensors input partial coordinate ordering to provably minimize the number of parallel partial sorting passes. We evaluate both a serial and a parallel implementation of Quesadilla on a set of 19 tensors from the FROSTT collection, a set of tensors taken from scientific and data analytic applications. We compare Quesadilla and a generalization, Top-2-sadilla to several state of the art approaches, including the tensor transposition routine used in the SPLATT tensor factorization library. In serial tests, Quesadilla was the best strategy for 60% of all tensor and transposition combinations and improved over SPLATT by at least 19% in half of the combinations. In parallel tests, at least one of Quesadilla or Top-2-sadilla was the best strategy for 52% of all tensor and transposition combinations. A full version is on arXiv [4].

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; • **Theory of computation** → **Sorting and searching**; • **Software and its engineering** → Source code generation.

KEYWORDS

Sparse Tensors, Transposition, Sorting, COO, Radix Sort

ACM Reference Format:
Suzanne Mueller, Peter Ahrens, Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Brief Announcement: Sparse Tensor Transpositions. In *ACM/IEEE Joint Conference on Digital Libraries in 2020 (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3350755.3400245

1 INTRODUCTION

Tensors generalize vectors and matrices to any number of dimensions. Tensors used in computation are often sparse, or mostly zero, necessitating specialized storage formats which can compress the zero entries. These formats range from a simple list of coordinates (COO) to complicated data structures such as Compressed Sparse Fiber (CSF) [7]. These formats have a natural ordering of their dimensions that provides a lexicographical ordering of the tensor nonzeros. In a sorted list of coordinates, the order of the sorting keys determines this lexicographic ordering.

Tensor algebra is used to compute with data stored in tensors. These multidimensional computations need to access the nonzero entries in one or more tensors, compute, and store the results. Accessing the nonzero entries requires some traversal of the tensor. However, unlike for dense tensors, traversing the nonzeros of a sparse tensor in different lexicographical orderings may be asymptotically more expensive than the natural lexicographical ordering. Therefore, it is often faster to first transpose input tensors by reordering their dimensions before executing tensor expressions. This way, the tensor can be accessed naturally in the expression itself.

Tensor transposition is ubiquitous in data processing. Anytime multiple tensor expressions are composed and the output of one expression must be used as an input to the next, with a different index ordering and possibly a different sparse format, we need to transpose. Element-wise operation between tensors without matching index orderings (thus requiring transposition as a bottleneck) is listed as one of the five benchmark operations in the Parallel Sparse Tensor Algorithm Benchmark Suite (PASTA) [3].

The Coordinate (COO) sparse tensor format simply stores the tensor as a list of nonzero coordinates together with their nonzero values. If the coordinates are ordered lexicographically, adjacent coordinates may share the same indices in the first several modes. The Compressed Sparse Fiber (CSF) format [7] compresses these duplicate nonzeros using a tree-like storage format. In CSF, nodes represent indices, leaves represent nonzeros and paths from root to leaf represent coordinates. The children of each node are ordered. The matrix case of CSF is called Compressed Sparse Row (CSR). Both of these formats induce a natural lexicographic ordering of the dimensions; iteration in the natural order is efficient.

The current approach to transposing sparse tensors involves converting the sparse tensor into Coordinate format, sorting the list of coordinates, and finally packing the coordinates into the desired output sparse tensor format [7]. This reduces tensor transposition to sorting coordinates. The resulting lists of coordinates often contain partial orderings we can use to accelerate the sorting step.

As an example, consider row-major to column-major sparse matrix transposition. To transpose the matrix, the column coordinates must be ordered lexicographically before the row coordinates. This could be accomplished by sorting with the column coordinate as the primary key and the row coordinate as the secondary key, but we can do better. The coordinates are already sorted on the rows. A

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400245

stable sort on just the column coordinates would achieve the desired result, and a simple histogram sort can get the job done in linear time. We have just described the famous HalfPerm algorithm for sparse matrix transposition due to Gustavson [2]. In this paper, we will generalize this optimization to arbitrary tensor transpositions.

The main contributions of this work are:

(1) A decomposition of tensor transposition into parallelizable partial sorts (one of the two partial sorts is novel) that optionally respect previous partial orderings.
(2) An algorithm that uses partial orderings in the original sparse tensor format to minimize the number of partial sorts required by the transposition algorithm. This relates the parallel span of radix sorting to partial orderings in the input.
(3) A parallel implementation that demonstrates this transposition algorithm is competitive with, and often faster than, state of the art approaches.

A complete manuscript (containing proofs) is available on the arXiv [4]. Algorithms, figures, and theorems are numbered to match.

2 ALGORITHMS

A tensor of **rank** r is a multidimensional array associating r-tuples of indices (referred to as **coordinates**) with values, or **entries**. We refer to the k^{th} position in a coordinate as mode k. The size of a tensor is an r-tuple n of **dimensions**, where each index i_k is an integer in the range $1 \le i_k \le n_k$. Let N be the number of nonzero entries in our tensor. We define lexicographic ordering on r-tuples with a tuple σ of modes in the order they should be considered. We describe the $(1, 2, ..., r)$ ordering of k-tuples as **simple**.

A naive algorithm for sparse tensor transposition uses comparison sort on a coordinate list. It takes $O(r)$ time to compare two coordinates of an r-tensor. Thus, a comparison based coordinate sort would run in $O(rN \log N)$ time. However, since the indices are bounded by the dimensions, we can use the stable, linear time histogram sort (referred to as counting sort in [1]) to sort the coordinates on a single mode k in $O(rN + n_k)$ time. If we use r histogram sorts (a radix sort on r-digit numbers), we can sort our coordinate list in $O(r^2N + \sum_k n_k)$ time, an asymptotic improvement over comparison sort when the dimensions are small. This algorithm can be improved further; for some transpositions, we do not need to use all r sorts. For example, HalfPerm uses only one histogram sort to prioritize the second dimension in the new ordering [2].

In this work, we formalize and generalize this idea to produce the Quesadilla tensor transposition algorithm, which provably performs the minimal number of partial histogram sorts. We propose two types of partial sorts, Algorithm 1 and Algorithm 2, which can be understood as two separate cases of the same function PartialSort. When $l = 0$ (Algorithm 1), PartialSort is a simple histogram sort, bringing the k^{th} mode to the top of an ordering.

When $l > 0$ (Algorithm 2), we describe PartialSort as "bucketed," since it sorts within fixed **buckets**, or groups of contiguous coordinates which agree on the first l modes in the current ordering. A naive implementation of Algorithm 2 which comparison sorts each bucket would incur a logarithmic overhead. If we histogram sort each bucket, we incur an unacceptable $O(n_{\tau_k})$ cost per bucket. Our solution is to first discover and store the buckets that each coordinate belongs to, stably sort on our desired mode with a single histogram sort, and then reimpose order on the first l modes by stably sorting on the bucket numbers themselves. This second sort on the buckets is similar to a histogram sort, but we can avoid some work by reusing information computed during the initial bucket discovery. Since there are at most N buckets, it also runs in linear time. Note that the input must be sorted under $(\tau_1, \ldots, \tau_l)$ to discover the buckets in linear time by examining adjacent coordinates.

Algorithm 1 and 2: PartialSort$(A, (\tau_1, \ldots, \tau_l), \tau_k)$

Input: A is a rank-r tensor of dimension n with N nonzeros stored in COO, sorted under the ordering τ. Our goal is to sort A on τ_k while maintaining the ordering $(\tau_1, \ldots, \tau_l)$. Algorithms 1 and 2 refer to when $l = 0$ and $l > 0$, respectively. We require that $l < k$.

Output: A sorted copy of A under the ordering

$$(\tau_1, \ldots, \tau_l, \tau_k, \tau_{l+1}, \ldots, \tau_{k-1}, \tau_{k+1}, \ldots, \tau_r).$$

While extensive research has been devoted to parallel implementations of histogram sort [5, 7], our "bucketed" variant is novel. Notice that the buckets limit the travel of coordinates between input and output orderings; coordinates do not escape their buckets. Therefore, running Algorithm 2 on a contiguous region of input buckets will compute the corresponding region of the output ordering. This gives our chosen parallel algorithm where we assign to each processor the buckets which begin in their region, and each processor simply runs Algorithm 2 locally on their section.

We are ready to state Algorithm 3, which sorts the coordinates using a minimal number of calls to PartialSort, avoiding bucketed sorts. We will use a function $f(\tau, p)$ defined on an ordering τ as the set $\{\tau_{k+1}, \ldots, \tau_r\}$ where $\tau_k = p$, or the set of modes which follow p in the ordering τ. For example, $f((1, 3, 2, 4), 3) = \{2, 4\}$.

Algorithm 3: QuesadillaSort(A, σ)

Input: A is any simply ordered list of r-coordinates, σ is an ordering with r distinct modes.

Output: A, sorted in σ order.

```
l ← 0
while l < r do
    k ← l
    while k + 1 < r and f(σ, σ_{k+1}) ⊈ f((1, 2, ...), σ_{k+1}) do
        k ← k + 1
    l' ← k + 1
    while k > l do
        A ← PartialSort(A, (σ_1, ..., σ_l), σ_k)
        k ← k − 1
    l ← l'
return A
```

Theorem 2. *Given a target ordering σ, QuesadillaSort(σ) uses the minimum-length sequence of calls to PartialSort required to sort any simply ordered list of r-coordinates to σ order.*

Theorem 3. *Among minimum-length sequences of PartialSort calls that sort simply ordered lists of r-coordinates to target ordering*

σ, the sequence used by QuesadillaSort(σ) *minimizes the number of bucketed partial sorts.*

Although the two sorting primitives presented are both histogram sort variants, they could be replaced with any stable sort such as quicksort or merge sort. However, if a comparison sort is used at some level k where the current ordering is τ and $(\tau_1, \ldots, \tau_l) = (\sigma_1, \ldots, \sigma_l)$, it makes more sense to completely sort each bucket (equivalence class under $(\tau_1, \ldots, \tau_l)$) to σ order.

Thus, we also propose the Top-K-sadilla algorithm, which uses Quesadilla to sort the tensor to $(\sigma_1, \ldots, \sigma_K)$ order, then sorts each bucket using quicksort. The best choice of the value K will be investigated in our experiments, since it depends both on the permutation and on the dimension of the tensor.

3 EVALUATION

We evaluate Quesadilla and Top-K-sadilla sort, showing that our technique outperforms various state of the art approaches for sparse tensor transposition. We compare against SPLATT [7], a high-performance C++ toolkit for sparse tensor factorization that uses a combination of histogram sort, quicksort, and insertion sort to sort tensors in COO. We also evaluate against sparse tensor transposition routines that sort nonzeros with (least significant digit) radix sort (using Algorithm 1 for each pass) or glibc's `qsort`.

We ran all experiments on a 2.5 GHz Intel Xeon E5-2680 v3 machine with 24 cores, 30 MB of L3 cache and 128 GB of main memory. The machine runs Ubuntu 18.04.3 LTS with glibc 2.27. We compiled the benchmarks using GCC 7.4.0. We ran each experiment 100 times and report minimum execution times.

We transpose real-world tensors obtained from the FROSTT Tensor Collection [6]. Specifically, we used "flickr-3d," "nell-1," "nell2," "vast-2015-mc1-3d," "chicago-crime-comm," "delicious-4d," "enron," "flickr-4d," "nips," "uber," "lbnl-network," and "vast-2015-mc1-5d." We measured normalized running times of SPLATT, `qsort`, Top-K-sadilla, Quesadilla, and radix sort for transposing each tensor from its initial ordering $\sigma = (1, \ldots, r)$ to every $r!$ possible ordering. Figure 3 shows the results of these experiments aggregated over all 408 possible combinations of input tensors and output orderings.

In serial tests, these results demonstrate that Quesadilla outperforms SPLATT, radix sort, and `qsort` on 60% of the sparse tensor transpositions. On half of all combinations, Quesadilla is at least 1.19× faster than SPLATT, 1.68× faster than radix sort, and 2.76× faster than `qsort`. In parallel tests, at least one of Quesadilla or Top-2-sadilla (our two novel algorithms) was the best strategy for 52% of all tensor and transposition combinations.

When $K = 1$, Top-K-sadilla reduces to the Top-1-sadilla algorithm that is similar to what SPLATT implements for sorting COO tensors, which we summarize in Section 2.2. Unlike SPLATT, which uses a custom hand-optimized implementation of quicksort, Top-1-sadilla uses `qsort` from C `stdlib` to sort nonzeros within each bucket created by the initial histogram sort. As Figure 3 shows, SPLATT outperforms Top-1-sadilla for most tensor transpositions in our experiments, thereby demonstrating that SPLATT's custom implementation of quicksort is more efficient than `qsort`. This performance difference suggests we can improve Top-K-sadilla's performance by using more optimized implementations of comparison sort to sort each bucket.

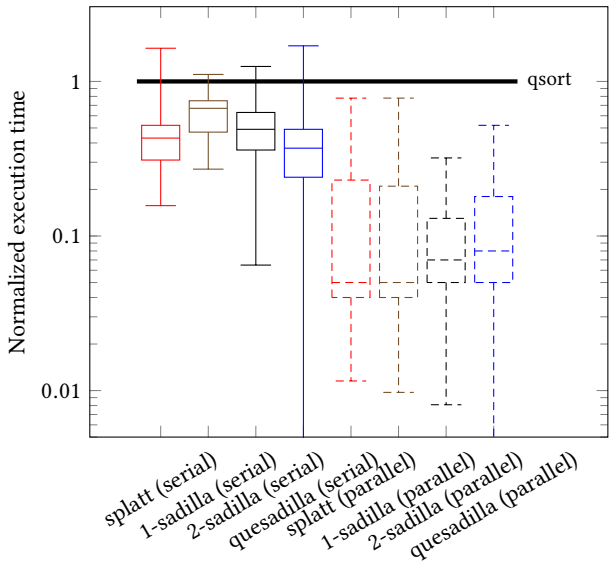

Figure 3: Normalized execution times of sparse tensor transposition with various algorithms, aggregated over all 408 possible combinations of test tensors and output orderings. Time is normalized to quicksort (horizontal line). 1-sadilla **and** 2-sadilla **denote** Top-K-sadilla **with** $K = 1$ **and** $K = 2$**.**

4 CONCLUSION

As sparse tensor representations receive increasing study, diversity in tensor formats will increase and applications will more frequently convert between formats. Sparse tensor transposition is the most basic instance of sparse format conversion, and an important subroutine in several format conversions. We have provided evidence that naive algorithms for sparse tensor transpositions can be improved substantially, and hope to inspire further improvements.

ACKNOWLEDGMENTS

This work was supported by a grant from the Toyota Research Institute, DARPA PAPPA Grant HR00112090017, and a DOE CSGF Fellowship DE-FG02-97ER25308.

REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to algorithms* (3rd ed ed.). MIT Press, Cambridge, Mass.
[2] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (Sept. 1978), 250–269.
[3] Jiajia Li, Yuchen Ma, Xiaolong Wu, Ang Li, and Kevin Barker. 2019. PASTA: a parallel sparse tensor algorithm benchmark suite. *CCF Transactions on High Performance Computing* 1, 2 (Aug. 2019), 111–130.
[4] Suzanne Mueller, Peter Ahrens, Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Sparse Tensor Transpositions. *The arXiv* (2020).
[5] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. Theoretically-Efficient and Practical Parallel In-Place Radix Sorting. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19)*. Association for Computing Machinery, Phoenix, AZ, USA, 213–224.
[6] 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.
[7] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS '15)*. IEEE Computer Society, Washington, DC, USA, 61–70.