Sparse Tensor Transpositions in the Tensor Algebra Compiler

by

Suzanne A. Mueller

S.B., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

Author .....
Department of Electrical Engineering and Computer Science
January 28, 2020

Certified by .....
Saman Amarasinghe
Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Sparse Tensor Transpositions in the Tensor Algebra Compiler

by

Suzanne A. Mueller

Submitted to the Department of Electrical Engineering and Computer Science
on January 28, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The current state of the art for transposing sparse tensors involves converting the sparse tensor into a list of coordinates, sorting the list of coordinates and finally packing the list of coordinates into the desired sparse tensor format. This thesis explores the potential for faster methodologies. Its main contributions are an algorithm that exploits partial sortedness to minimize sorting passes and an implementation that demonstrates that this transposition algorithm is competitive with state of the art. In particular the algorithm takes advantage of the ordering that already exists to apply selective sorting passes and thereby reduce the amount of work that needs to be done to reorder the tensor.

Thesis Supervisor: Saman Amarasinghe
Title: Professor, Electrical Engineering and Computer Science

Acknowledgments

First, I would like to thank Saman Amarasinghe and Fredrik Kjolstad for advising me in my research. I would also like to thank Peter Ahrens and Stephen Chou for the many insightful discussions on transposing tacos and all of the COMMIT group for making this such an enjoyable experience.

A big thank you to my academic advisor, Aleksander Madry, for being a mentor to me throughout my time at MIT.

Finally, I would like to thank my friends and family for being there for me as I took on this challenge. As you know, I could not have done it without your support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Tensors generalize vectors and matrices to any number of dimensions. Tensors used in computation are often sparse, which means many of the values are zero. To take advantage of the large number of zeroes in the tensor, we use sparse formats that allow the zeroes to be compressed away. These formats can range from a simple list of coordinates to much more complicated structures such as CSR [4], DCSR [3], and CSF [8]. Many of these formats have a natural ordering of the dimensions. In a sorted list of coordinates, the order of the sorting keys determines this ordering. We call this ordering of dimensions a lexicographical ordering on the nonzero coordinates.

Tensor algebra is used to manipulate 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. In sparse tensors, traversing all of the nonzeros is asymptotically faster than in dense tensors. However, unlike for dense tensors, traversing the nonzeros in different lexicographical orderings may be asymptotically more expensive than the natural lexicographical ordering. Different tensor expressions may require traversing the tensor in a different lexicographical order. Thus, reordering the dimensions, or transposing the tensor, is an important operation for fast tensor computations.

A lot of work on making dense tensor transpositions fast has been completed; however relatively little work has focused on sparse tensor transpositions.

The current state of the art for transposing sparse tensors involves converting the

sparse tensor into a list of coordinates, sorting the list of coordinates, and finally packing the list of coordinates into the desired sparse tensor format.

This approach reduces the problem of transposing a tensor into a problem of sorting a list of coordinates. However, the lists of coordinates have partial orderings we can use to accelerate the sorting algorithms. Take the example matrix in Figure 1-1. In order 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.

However, we can do better than that. The coordinates are already sorted on the row coordinates. By doing a stable sort on just the column coordinate, we get the same result. In this paper, we will generalize this optimization to arbitrary tensor transpositions.

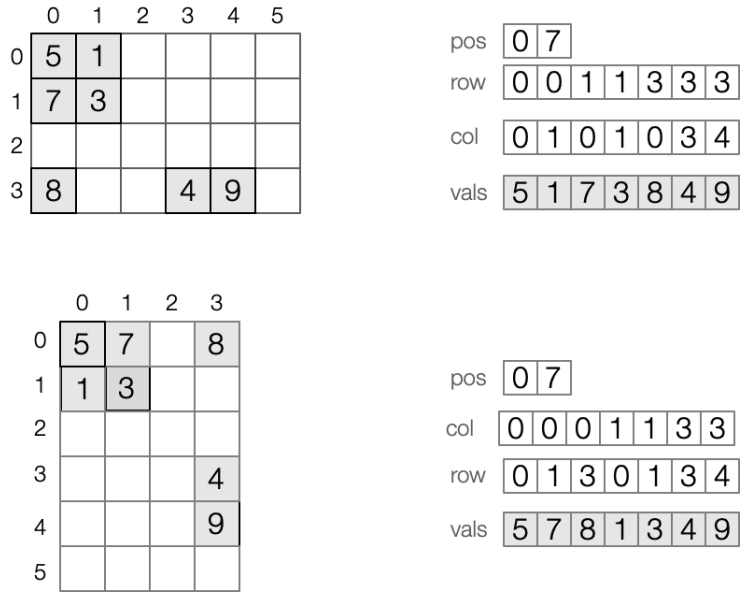


Figure 1-1: The matrix A can be represented as a list of coordinates including only the nonzero values. Transposing the tensor in this format switches the lexicographic ordering of the rows and columns, such that the columns appear first. The top list of coordinates represent the matrix in the top left. The bottom list of coordinates represent the transposed version of this matrix.

The main contributions of this thesis are:

1. A decomposition of tensor transposition into multiple partial sorts 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.

3. An implementation that demonstrates that this transposition algorithm is competitive with, and often faster than, state of the art approaches.

1.1 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 - Background provides background on different sparse tensor formats and previous work to make tensor transposes fast. It also presents the HalfPerm algorithm [6], which is a fast algorithm used for sparse matrix transposes.

Chapter 3 - Extending HalfPerm discusses work to extend HalfPerm to other sparse formats, as well as to higher order tensors and the challenges that this brings.

Chapter 4 - Partial Radix Sort poses the problem of transposing a tensor as a sorting problem. This section also introduces sorting primitives that can be used to partially sort coordinates.

Chapter 5 - Pass Minimization presents an algorithm called k-sadilla that minimizes the number of partial sorts needed to reach a given permutation of dimensions.

Chapter 6 - Results presents an implementation of k-sadilla and compares the results to other algorithms for sorting sparse tensors.

Chapter 7 - Conclusion concludes with some future directions that could further improve sparse tensor transposition techniques.

Chapter 2

Background

This section will provide background on tensor formats, as well as discuss prior work on sparse matrix transpositions. This related work provides a starting point for our exploration of k-dimensional sparse tensor transpositions.

2.1 Sparse Tensor Formats

There are many different sparse tensor formats that have been created to efficiently store and compute on sparse tensors. This section describes the COO, CSR [4], and CSF [8] formats as an introduction to some of these formats. All three of these formats create a natural ordering of the dimensions; iterating over the tensor in the natural order can be done very efficiently.

2.1.1 Coordinates (COO)

COO stores the nonzero coordinates in the tensor as a list of sorted coordinates. This can be implemented either by having a list of lists (one for each dimension) or as a list of tuples.

In terms of the number of dimensions, k, and the number of nonzeros, N, the total space to store the index is $O(k * N)$.

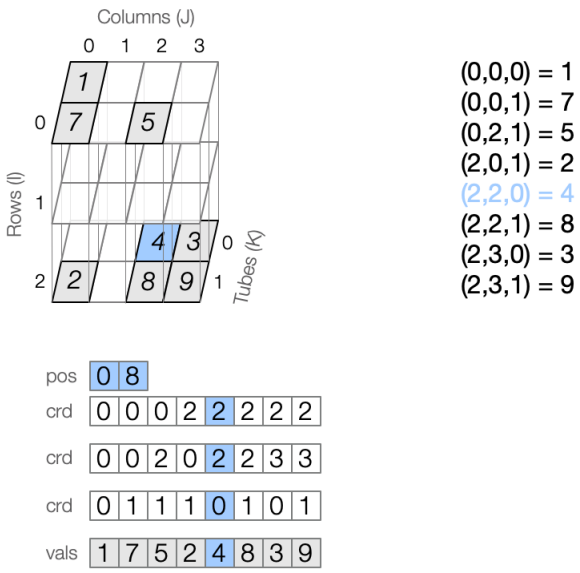


Figure 2-1: COO represents a tensor as a list of coordinates with all of the zero values compressed out. The list of arrays in the bottom left represent the tensor in the top left.

Most of this paper discusses transposes on COO in particular. COO is a popular format supported by several libraries. TensorFlow [1] does all of its sparse tensor computations in COO. It is also a convenient format to use as an intermediate stage of tensor transposition.

Listing 2.1: C code to iterate over COO in the natural lexicographic order.

```
1 // Iterate in order (0, 1, 2).
2 for(int i = 0; i < N; i ++) {
3     int i0 = crd0[i];
4     int i1 = crd1[i];
5     int i2 = crd2[i];
6     int val = vals[i];
7     printf("(\%d, \%d, \%d) = \%d", i0, i1, i2, val);
8 }
```

2.1.2 Compressed Sparse Row (CSR)

CSR is a format for storing sparse matrices that uses three arrays to store a matrix with dimensions $(d_0 \times d_1)$ with N nonzero coordinates. For each row in the matrix, the first array stores the locations of its column coordinates and values. The remaining two arrays store these column coordinates and the values.

In addition to compressing the zero coordinates, this format also compresses duplicate row coordinates using an array *pos*. The *pos* array stores the ranges of *crd* corresponding for each row. The segment of *crd* for row i is from $pos[i]$ to $pos[i+1]$. This format allows for fast access to each row of the tensor.

The array position array has a size of $d_0 + 1$ and both the column coordinates and values arrays have size N. The total space needed to store this format is $O(d_0 + N)$.

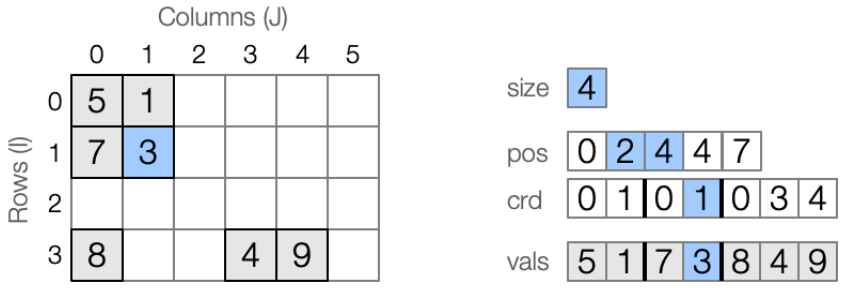


Figure 2-2: CSR stores only the columns that contain nonzero values. To find the value of (1, 1), first access $pos[1]$ to find the segment. Scan the segment in *crd* to find the index of the column coordinate. The value is located at the same index in the values array.

Listing 2.2: C code to iterate over CSR in the natural lexicographic order.

```
1 // Iterate in order (0, 1).
2 for(int i0 = 0; i0 < 4; i0 ++) {
3     for(int idx = pos[i0]; idx < pos[i0 + 1]; idx ++){
4         int i1 = crd[idx];
5         int val = vals[idx];
6         printf("(\%d, \%d) = \%d", i0, i1, val);
7     }
8 }
```

2.1.3 Compressed Sparse Fiber (CSF)

CSF is of particular interest because it is a sparse format that is very compressed. It was introduced by Smith and Karypis in a paper that discusses the Surprisingly ParalleL spArse Tensor Toolkit (SPLATT) [8].

CSF format is sparse tensor format that represents the tensor as a tree structure. Leaves in a CSF tensor represent nonzeros. The branching factor of the tree is an indicator of how well the format has compressed the values. Different dimension orderings will result in different branching factors.

CSF stores these trees by layering pointer arrays to compress the additional tensor dimensions, thereby reducing the duplication of coordinate values. The size of this structure is upper bounded by the number of nonzero values.

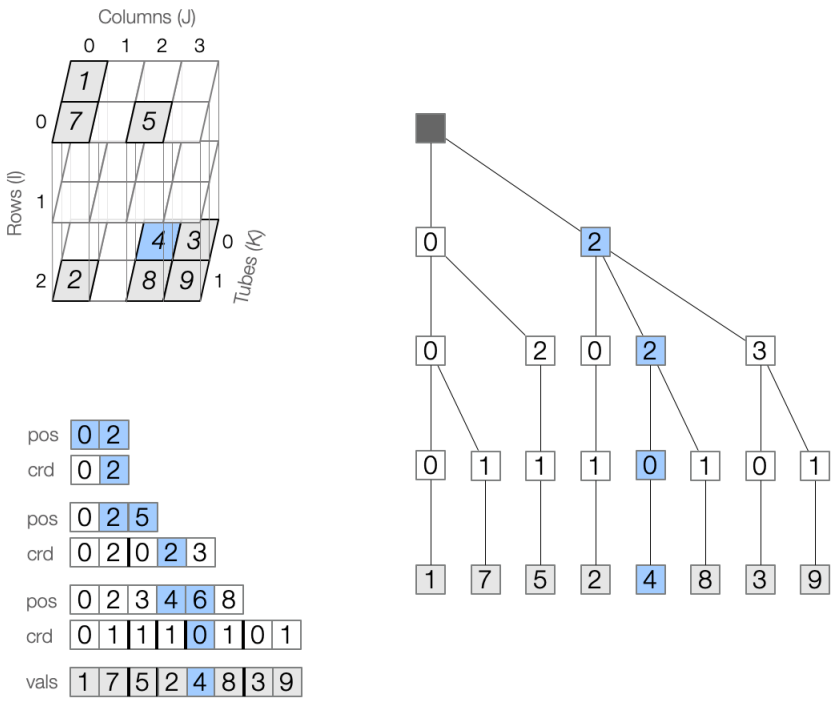


Figure 2-3: CSF stores a sparse tensor in a tree-like structure. To find the value of (2, 2, 0), we use the layered *pos* and *crd* arrays to traverse down the tree.

Listing 2.3: C code to iterate over CSF in the natural lexicographic order.

```
1  // Iterate in order (0, 1, 2).
2  for(int idx0 = pos0[0]; idx0 < pos0[1]; idx0 ++) {
3      int i0 = crd0[idx0];
4      for(int idx1 = pos1[idx0]; idx1 < pos1[idx0 + 1]; idx1 ++){
5          int i1 = crd1[idx1];
6          for(int idx2 = pos2[idx1]; idx2 < pos2[idx1 + 1]; idx2 ++){
7              int i2 = crd2[idx2];
8              int val = vals[idx2];
9              printf("(\%d, \%d, \%d) = \%d", i0, i1, i2, val);
10         }
11     }
12 }
```

2.2 Dimension Permutations

COO, CSR, and CSF all impose a lexicographic ordering on the coordinates. Let a lexicographic ordering of dimensions $\sigma(i)$ refers to ordering each nonzero coordinate i by the tuple $(i_{\sigma_0}, i_{\sigma_1}, i_{\sigma_2}, \ldots)$. A permutation of the dimensions σ can be expressed as a final lexicographic ordering. Thus, transposing the matrix A is equivalent to storing A in $\sigma = (1, 0)$ order. Without loss of generality, in this paper we always assume that we start in order $(i_0, i_1, i_2, \ldots)$ so that the output ordering determines all the changes that need to be made.

2.3 Reordering Tensor Dimensions

Certain computations will perform better when the dimensions can be iterated over efficiently in a different order than the storage order. When we encounter such a computation, it will be beneficial to transpose the tensor. In this section, we will discuss some of the work to make sparse matrix and tensor transposes faster.

The simple example of matrix multiplication has a different asymptotic performance depending on the storage order. In matrix multiplication, two $M \times M$ matrices are multiplied such that $C(i, j) = A(i, j) * B(j, i)$. When both matrices are stored in CSR with order (0, 1), such as in Listing 2.4, we need to search for the matching coordinate to do the multiplication. However, when B is stored with dimension ordering (1, 0) we can simply iterate over both matrices in order, as seen in Listing 2.5. Reordering the dimensions of B leads to asymptotically better performance.

Listing 2.4: C code to multiply two $M \times M$ matrices, $C = A \times B$ when dimensions of A and B are both stored in order (0, 1)

```
int n = 0;
for(int iA = 0; iA < M; iA ++) {
    for(int idxA = A_pos[iA0]; idxA < A_pos[iA0 + 1]; idxA ++){
        int jA = A_crd[idxA];
        // Locate the matching coordinate in B.
        int jB = jA;
        for(int idxB = B_pos[jB]; idxB < B_pos[jB + 1]; idxB ++){
            int iB = B_crd[idxB];
            if ( iB == iA) {
                C_pos[jA + 1] = n;
                C_crd[n] = iA
                C_vals[n] = A_vals[idxA] * B_vals[idxB];
                n ++;
            }
        }
    }
}
```

Listing 2.5: C code to iterate multiply two M x M matrices, $C = A \times B$ when dimensions of A are stored order (0, 1) and dimensions of B are stored in order (1,0).

```
int n = 0;
for(int iA = 0; iA < M; iA ++) {
    int iB = iA;
    // Iterate over both
    int idxA = A_pos[iA];
    int idxB = B_pos[iB];
    while(idxA < A_pos[iA + 1] && idxB < B_pos[iB + 1]) {
        int jA = A_crd[idxA];
        int jB = B_crd[idxB];
        if(jA == jB){
            C_pos[jA + 1] = n;
            C_crd[n] = iA;
            C_vals[n] = A_vals[idxA] * B_vals[idxB];
            idxA ++;
            idxB ++;
            n ++;
        } else if(jA > jB) {
            idxB++;
        }else if(jA < jB){
            idxA++;
        }
    }
}
```

2.3.1 Sorting Strategies

Tensor dimensions can be reordered using any sorting method. Any comparison sort can sort the coordinates. Since the coordinates come from a fixed range of values, we can also use sorts that work on fixed length keys, like histogram or radix sorts.

SPLATT, a sparse tensor library designed to be highly parallel, uses a different

sorting strategy to take advantage of potential parallelism that exists in the problem [10]. SPLATT chooses to first do a histogram sort on dimension i_{σ_0}. It then sorts the buckets for each coordinate of dimension i_{σ_0} using quicksort. In the sequential implementation this strategy benefits from smaller subproblems for quicksort. In the parallel version, SPLATT is able to sort these buckets in parallel.

2.3.2 Sparse Matrix Transpositions

In this work, we focus on sorting a list of coordinates. However, the HalfPerm algorithm directly transposes matrices in CSR format, without using an intermediate coordinates format [6]. This algorithm is used by libraries such Eigen [5] and Julia [2] to do fast sparse matrix transposes from CSR to CSR. In this section, I will present the algorithm as well as my work to generalize it to CSF matrices and tensors. I will also describe why this was ultimately not the best solution for transposing k-dimensional tensors.

2.3.3 HalfPerm Algorithm

HalfPerm computes the transpose $B = A^\intercal$ of a matrix A stored in CSR format in $O(d_0 + d_1 + N)$ time, where matrix A has d_0 rows, d_1 columns, and N nonzero values [6].

The HalfPerm algorithm transposes a matrix by computing the number of coordinates per column, or the column counts, of matrix A. These column counts allow HalfPerm to calculate row pointers using a prefix sum to random access to the segment where the coordinate belongs in the output. Finally, HalfPerm iterates over A and uses the row pointers to scatter the row coordinates and numerical values in the output matrix.

Algorithm 1: HALFPERM

Input: Matrix A has a three arrays: A_pos, A_crd, and A_vals.

A_pos is a length $d_0 + 1$ array that holds the ranges of the A_crd array for each row coordinate.

A_crd and A_vals are both length N arrays that hold the column coordinates and values respectively.

N is the number of nonzero elements

d_0 and d_1 are the number of rows and columns in A respectively.

Output: $B = A^\intercal$

$counts \leftarrow$ an integer array of length $d_1 + 1$ initialized to 0

$B\_pos \leftarrow$ an integer array of length $d_1 + 1$ initialized to 0

$B\_crd \leftarrow$ an integer array of length N

$B\_vals \leftarrow$ an integer array of length N

for $j \leftarrow 0$ **to** N **do**

  $col = A\_crd[j]$

  $counts[col + 1] \mathrel{+}= 1;$

end

for $j \leftarrow 1$ **to** $d_1 + 1$ **do**

  $counts[j] \mathrel{+}= counts[j - 1]$

  $B\_pos = counts[j]$

end

for $row \leftarrow 0$ **to** d_0 **do**

  for $j \leftarrow A\_pos[row]$ **to** $A\_pos[row + 1]$ **do**

    $col = A\_crd[j]$

    $idx = counts[col]$

    $B\_crd[idx] = row$

    $B\_vals[idx] = A\_val[j]$

    $counts[col] \mathrel{+}= 1$

  end

end

The transposition of matrix A using HalfPerm algorithm is shown in Figure 2-4. The column counts are calculated by iterating over A_crd and incrementing the count for each column encountered. The B_pos array is then computed using a prefix sum over the column counts array. The row pointers reserve enough space in the B_crd and B_vals arrays for each column coordinate and value.

To fill in the B_crds and B_vals arrays, we iterate over the coordinates and place the row coordinate in the first available position for the given column. By going over the tensor in order, we will insert the rows into the transposed matrix in order as well.

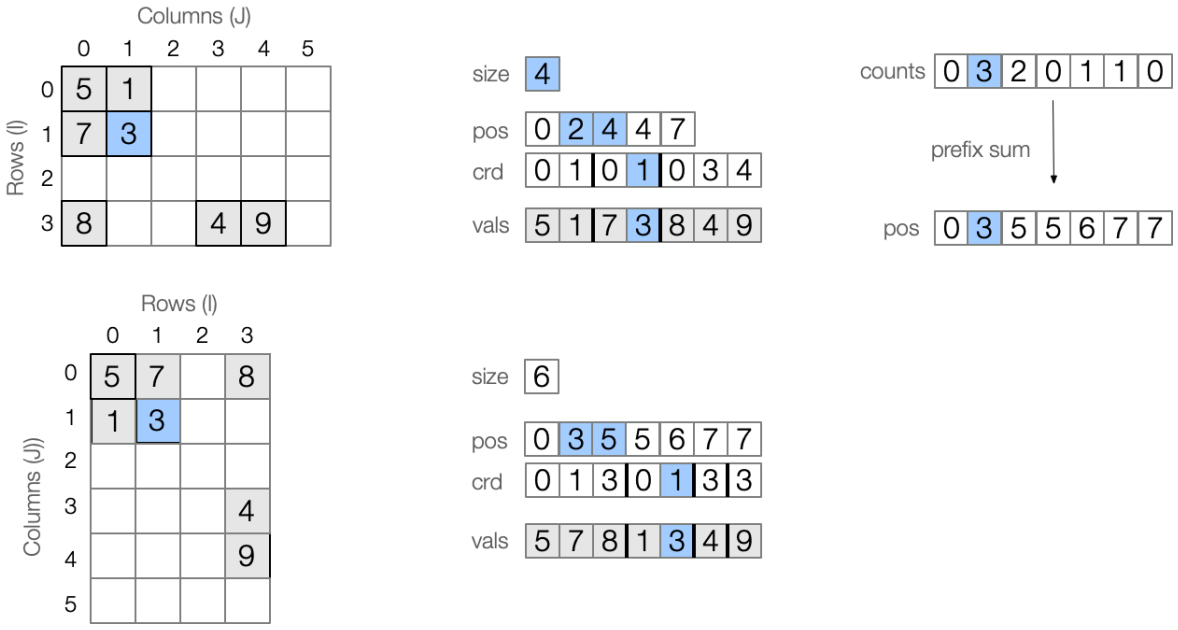


Figure 2-4: The sparse matrix is transposed using the HalfPerm algorithm. HalfPerm counts the number of items for each column in an array of size $d_1 + 1$, and then fills in the position array of size $d_1 + 1$ using these column counts.

Chapter 3

Extending HalfPerm

HalfPerm is a fast algorithm for transposing matrices that does better than sorting coordinates. The success of this algorithm brings to question whether there is a way to make all sparse tensor transposes faster. In this section, we look at how HalfPerm is able to beat sorting and what would be needed to extend this idea to CSF matrices and tensors.

3.1 HalfPerm Optimizations

HalfPerm is able to do better than a sort because it takes advantage of the existing ordering of the matrix. Iterating over the matrix in order will output the coordinates sorted on the rows due to the lexicographic ordering enforced by CSR. HalfPerm takes advantage of this by only sorting on the columns. The steps of HalfPerm closely resemble a histogram sort, which is a stable sort. Doing a stable sort on the columns preserves the ordering of the rows, so it can safely put the coordinates into the output.

Another optimization made by HalfPerm is that it is able to use the intermediate array to compute the *pos* array in the output tensor.

3.2 Extending to CSF Matrices

HalfPerm works specifically to perform a CSR to CSR transpose, but can be adjusted to transpose matrices from CSF to CSF.

First we can observe that the input format is not important for the runtime nor the strategy of the algorithm. HalfPerm only requires that the input can be iterated over in order and within a runtime of $O(d_0 + d_1 + N)$. All of the sparse formats we are considering achieve that.

The extra compressed level in the output format poses an additional problem that the transpose needs to solve. To compress the first level, we need to know how many distinct column coordinates there are. This can be accomplished by keeping a size d_1 array that stores boolean values. A prefix sum on this array will also reserve space for each column in the $B0\_crd$ array of the first dimension. The total can be used to set the $B0\_pos$ array as well as allocate the space. This provides us with a way to random access by the column coordinate as well.

HalfPerm still gives the correct locations for the row coordinates, since we will not have any duplicates to be compressed in the bottom dimension. The count array simply needs to compress out the zeroes when storing in the $B1\_pos$ array.

The runtime for this modified HalfPerm is $O(N + d_1)$. We actually drop a dependence on the dimension of the rows because iterating over the input tensor can be done in $O(N)$ instead of $O(N + d_0)$.

3.3 Extending to Tensors

I explored if there were a way to extend the ideas of HalfPerm to tensors by constructing a way to random access into the output for higher order tensors.

In Section 3.2 we were able to extend HalfPerm to work for CSF output formats by adding an additional array to find the distinct columns. Additional dimensions means that we will need to compute the distinct coordinates for each dimension. In fact, we will need to know all distinct i_{σ_j} for every coordinate pair for all distinct

coordinate pairs $(i_{\sigma_0}, i_{\sigma_1}, ..., i_{\sigma_{j-1}})$.

One way to do this would be to have a dense tensor for the top (k-1) levels in the output order. We can use the HalfPerm algorithm for CSF using the top (k-1) dimensions as one giant dimension of size $\prod_{i=0}^{k-2} d_i$. This may require a large amount of space, since we are storing all but one dimension in a dense tensor. The runtime for this is $O(N + \prod_{i=0}^{k-2} d_i)$.

It is possible instead to do this in multiple passes to decrease the space required. We can achieve this by building the tensor in output order sequence. The count arrays from previous dimensions can be used to random access into the next. The first level will use $O(d_0)$ space, where d is the size of the dimension, the next level will use $O(d_1 * z_0)$ space, where z is the number of distinct nonzero d_0 coordinates. This will continue for every level, and may use less space than a dense tensor. We have to iterate over the whole tensor $O(k-1)$ times to set up the output structure, before we are finally able to insert the values. The overall runtime for this method is $O(k * N + \sum_{i=0}^{k-2} d_i)$.

In either case, the need for random access into all dimensions results in an unacceptable amount of space used. Due to this, this line of exploration was abandoned in favor of attempting to take advantage of the specific structure of different permutations.

Chapter 4

Partial Radix Sort

The state of the art for sparse tensor transposition is to sort the coordinates into the desired lexicographic order. Because coordinates already have an initial ordering, we can think of sorting coordinates as simply changing the lexicographic ordering to prioritize different dimensions. The fixed length keys allow us to use linear time stable sorts like histogram sort to achieve asymptotic improvements over $O(N \log N)$ comparison sorts. We are also able to take advantage of partial ordering to sort them faster. HALFPERM uses a histogram sort to prioritize the second dimension in the new ordering. Depending on the size of the second dimension, this single histogram sort is often faster than a generic sort of the coordinates, and certainly faster than redundantly sorting the first dimension before sorting the second. We will use this idea of pass omission to get asymptotically better performance.

This section describes two sorting primitives that will be used to describe the sorting strategies and will also introduce notation used to specify these strategies.

4.1 Histogram Sort

A histogram sort sorts bounded integer keys by counting the number of occurrences of each of the different values and then performing a prefix sum, also known as a cumulative sum or prefix scan, over the array to determine where the groups corresponding to each key will lie in the output array. This array reserves enough space

for all of the coordinates to appear in the output order, which allows the algorithm to put each coordinate directly into its final location, as described by the prefix summed array, as it does a second scan over the input.

Algorithm 2: HISTOGRAM

Input: A is a length N array of coordinate structs. Each coordinate struct has a list of coordinates crd and a value val.
crd is an array of length k. $crd[i]$ contains the coordinate for the ith dimension.
$dims$ is a length k array that stores the size of dimension i at $dims[i]$.
j is an integer representing the dimension to be sorted on.

Output: A list of coordinates B that contains A after a histogram sort on j.

$d \leftarrow dims[j]$

$count \leftarrow$ a length $d + 1$ array of integers initialized to 0

`// Compute the count array`

for $i \leftarrow 0$ **to** N **do**

  $crd = A[i].crd[j]$

  $count[crd + 1]$ += 1

end

`// Prefix Sum`

for $crd \leftarrow 1$ **to** d **do**

  $count[crd]$ += $count[crd - 1]$

end

`// Move coordinates to final output destination`

for $i \leftarrow 0$ **to** N **do**

  $crd = A[i].crd[j]$

  $idx = count[crd]$

  $B[idx] = A[i]$

  $count[crd]$ += 1

end

4.1.1 Runtime

The histogram sort iterates over the coordinates twice and the count array once. The total runtime is $O(N + d)$, where d the size of the dimension being sorted on.

4.1.2 Effect on Ordering

A histogram sort will move the dimension to the top of the lexicographic order.

Claim 4.1.1. *Given a lexicographic ordering σ, where $\sigma_j = d$, a histogram sort on dimension d will result in a final ordering of, $\sigma' = (d, \sigma_0, ..., \sigma_{j-1}, \sigma_{j+1}, ..., \sigma_{k-1})$.*

Before the sort on d takes place, the coordinates are sorted on $(\sigma_0, ..., \sigma_{j-1})$ by definition. If the keys only contained $(\sigma_0, ..., \sigma_j)$, this is the state that the coordinates would be in before the final pass in the radix sort. Sorting on d will move the dimension to the top of this ordering.

The relative order between coordinates with the same $(i_{\sigma_0}, ..., i_{\sigma_j})$ will not change by this sort, because we use a stable sort. The equivalent coordinates will remain in the same order, since these will also have equivalent values for the d coordinate. This preserves the order of the dimensions that come after d in the initial lexicographic ordering.

This will leave the final lexicographic ordering σ'.

4.2 Bucketed Histogram Sort

Although radix sort is most commonly performed from least significant digit to most significant digit, there is a variation that does the most significant digits first and then sorts each of the buckets that occur when you fix particular values for the already sorted digits. This process sorts each bucket recursively. However, the range of the keys for each dimension may be rather large, so we need to do the histogram sort only once, not once per bucket. This process can be accomplished by saving the buckets that each coordinate belongs to, doing a histogram sort, and then moving the coordinate back to its bucket.

The buckets can be created for the first b dimensions in the current lexicographical ordering σ. The sorting guarantees described in this section do not hold if the buckets are created using any arbitrary dimension choices.

Algorithm 3: BUCKETEDHISTOGRAM

Input: A is a length N array of coordinate structs. Each coordinate struct has a list of coordinates crd and a value val.

crd is an array of length k. $crd[i]$ contains the coordinate for the ith dimension.

$dims$ is a length k array that stores the size of dimension i at $dims[i]$.

j is an integer representing the dimension to be sorted on.

$fixed\_dims$ is an integer array of dimensions to be bucketed on.

Output: A list of coordinates B that contains A after a bucketed histogram sort on j with fixed dimensions $fixed\_dims$.

```
// Save the positions of the buckets.
```

$bucket \leftarrow$ an integer array of size N

$counts \leftarrow$ an integer array of size N

$current \leftarrow 0$

for $i \leftarrow 1$ **to** N **do**

  for $j \leftarrow 0$ **to** b **do**

    $d \leftarrow fixed\_dims[j]$

    if *A[i].crd[d] != A[i-1].crd[d]* **then**

      $current \mathrel{+}= 1$

      $counts[current] = i$

      break

    end

  end

  $bucket[i] = current$

end

```
// perm[i] contains the index of A[i] before the sort.
```

A, $perm$ = histogramPerm(A, N, j, $dims$)

```
// Reintroduce the ordering recorded in the original buckets.
```

for $i \leftarrow 0$ **to** N **do**

  $B[counts[bucket[perm[i]]] = A[i]$

  $counts[bucket[perm[i]]] \mathrel{+}= 1$

end

4.2.1 Runtime

Let b be the number of dimensions being bucketed on and d be the size of the dimension that is being sorted on. Saving the buckets takes $O(b * N)$ time, the histogram sort takes $O(N + d)$ time, and moving the coordinates back to their buckets takes $O(N)$ time. The total runtime of bucketed histogram sort is $O(N * (b + 1) + d)$.

4.2.2 Effect on Ordering

Bucketed histogram sort will move the dimension d to immediately follow the bucketed dimensions in the lexicographical ordering.

Claim 4.2.1. *Given a lexicographic ordering σ, where $\sigma_j = d$, a bucketed histogram sort on dimension d with buckets for the top b levels will result in a final ordering of, $\sigma' = (\sigma_0, ..., \sigma_{b-1}, d, \sigma_b, ..., \sigma_{j-1}, \sigma_{j+1}, ..., \sigma_{k-1})$.*

The bucketed sort only does a sort within coordinates that have the same values for dimensions $(i_{\sigma_0}, ..., i_{\sigma_{b-1}})$. The lexicographic ordering of these dimensions will not change.

Within each bucket, the histogram sort will change the ordering from $(\sigma_b, ..., d, ..., \sigma_{k-1})$ to $(d, \sigma_b, ..., \sigma_{j-1}, \sigma_{j+1}, ..., \sigma_{k-1})$ as proven above.

The final lexicographic ordering is then $\sigma' = (\sigma_0, ..., \sigma_{b-1}, d, \sigma_b, ..., \sigma_{j-1}, \sigma_{j+1}, ..., \sigma_{k-1})$.

4.3 Notation

The notation SORT(i [B]) refers to the pass that sorts dimension i while bucketing on the levels in the set B. If B is the empty set, then a standard histogram sort is used, otherwise a bucketed histogram sort is needed.

4.4 Comparison 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, it will make sense to do the sort in a single pass. Although the asymptotic runtime of doing them separately and together is the same, in practice it is more efficient to run the comparison sort a single time.

Chapter 5

Pass Minimization

In this thesis we wish to take advantage of the knowledge that parts of the desired lexicographic ordering σ may already be partially satisfied. The permutation σ determines which of the $k!$ different permutations is the final goal. Not all permutations are equally difficult.

An obvious example of the variability of difficult is if the desired permutation is the same as the input permutation. This permutation requires no work at all since the data is already in the desired order. On the other hand if σ inverts the order of the dimensions, nearly every dimension needs to be resorted. Sorting all dimensions is needed because each of these dimensions was only sorted with respect to a bucket that required selecting a single value for the 0th coordinate. When the order is inverted, there is no partial ordering to take advantage of and every dimension (except the 0th) will require a sort.

In this section, we minimize the number of passes by first setting lower bounds on the number of sorting passes needed for a given σ and then describing an algorithm that achieves this lower bound.

5.1 Cost of Passes

In Chapter 4, we described two types of histogram sorting passes. These both achieved a linear runtime in the number of nonzeros and the size of the dimension being sorted

on. In this paper we assume a cost function that weights all passes equally. To minimize this cost function, we must minimize the number of passes. In later chapters, we will discuss the effects this assumption may have on performance.

5.2 Pass Bounds

The number of dimensions k is an upper bound on the number of passes needed to sort coordinates. This is the number of passes that are needed if we have a completely unsorted coordinate list and do a standard radix sort.

The histogram sort and bucketed histogram sort can only move dimensions up in the lexicographic ordering. This provides a lower bound on the number of passes needed to sort the coordinates into the new lexicographic ordering. All dimensions that end up in a position before a dimension that preceded it in the initial ordering will need to be sorted on. Any algorithm that outputs a correct permutation will at least need to sort on these levels.

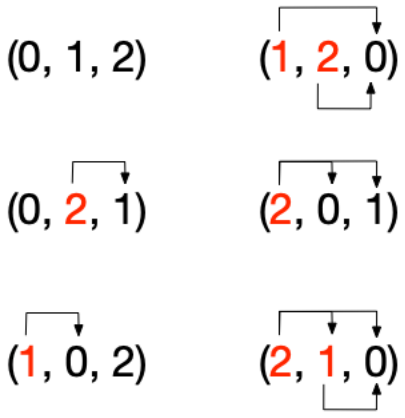


Figure 5-1: This figure shows which dimensions need to be sorted on to achieve the output permutation, starting from (0, 1, 2). There must be a sorting pass on the dimensions in red. The arrows point to the dimensions that preceded the red dimension in the input permutation. There must be a sorting pass since this is the only way to move before the preceding dimensions.

Algorithm 4: NEEDSSORT

Input: i is an integer index that refers to the ith element in the permutation array σ.
σ is a length k array that contains a permutation of (0, 1, ..., $k-1$).
k is an integer representing the order of the tensor.

Output: True, if the dimension at index i in the output order σ needs to be sorted, False otherwise

```
for j ← i + 1 to k do
    if σ_j < σ_i then
        return True
end
return False
```

5.2.1 Sort scheduling

We present an algorithm called K-SADILLA that outputs a sort schedule that achieves the lower bound of number of passes from Section 5.2 on any desired output permutation.

Algorithm 5: K-SADILLA

Input: σ is a length k array that contains a permutation of (0, 1, ..., $k-1$).
k is an integer representing the order of the tensor.

Output: A sorting schedule that achieves the minimum number of passes to sort (0, 1, ..., k-1) to permutation σ

```
bucket ← ∅
curr ← an empty list
for i ← 0 to k do
    if NeedsSort(i, σ, k) then
        curr.prepend(σ[i])
        Continue
    for d ∈ curr do
        OUTPUT "SORT(d [bucket])"
    end
    curr = [ ]
    bucket = the first i elements of σ
end
```

Claim 5.2.1. K-SADILLA *produces the minimal number of passes needed to sort a list of coordinates.*

This algorithm outputs exactly one sort per level where NEEDSSORT is true. This is how we defined our lower bound on the number of sorts. The decision of whether it needs to be sorted is based purely on the output permutation, not on any other choices made in the algorithm. This algorithm achieves the desired minimal number of sorts.

Claim 5.2.2. K-SADILLA *produces a schedule that will permute the coordinates to have dimension ordering σ.*

We make a recursive argument for the correctness of K-SADILLA. The algorithm outputs sorting passes in phases, and we argue that a portion of σ will be satisfied by each of these phases.

When we output a set of sorting passes, we assume that all of the dimensions that are bucketed on are already sorted for the recursive argument. At the beginning of the algorithm, the bucket is empty so this assumption is valid. "Bucketing" on all of the previous levels preserves their ordering.

We output a set of sorting passes when we see that the dimension at σ_j does not need to be sorted. When we encounter σ_j, this means that every coordinate that precedes σ_j in the input order also precedes it in the output order. This means that we will not have any dimensions that are in the order before index j that need to be after it.

We sort all of the dimensions we have seen since the last set of sorts in reverse order. Each sorting pass will move the dimension to be immediately after the buckets. Sorting in reverse order uses the same strategy as in a least significant digit radix sort to get the dimensions to be in the correct lexicographic ordering.

At this point, applying the sort schedule will permute the dimensions such that the first $j + 1$ dimensions are in the correct output order.

The last level does not need to be sorted on, since there are no levels after it in the output order. From above we can say that the lexicographic order after applying the sorting phase terminated by level $k - 1$ will result in a lexicographical ordering where the top k dimensions of σ are in the correct order.

Thus, the schedule produced will sort all k levels correctly.

5.2.2 Algorithm Runtime

Finding the schedule takes $O(k^2)$ time. The total time to sort using the schedule generated by K-SADILLA is $O(p * N + \sum_{i=1}^{p} d_i)$, where the number of passes that are required is p.

Chapter 6

Results

6.1 Implementation

I implemented this algorithm by generating C++ code from a python script to sort the list of coordinates into a given output permutation using the different sorting strategies. I chose to do a code generator because this allowed for future flexibility to use different intermediate formats, custom packing code, and bit packing optimizations.

Testing code was generated to use the tensor algebra compiler (taco) framework [7] to load the tensors, time the sorting code, and check correctness of results.

There is a different function call for each transpose strategy and permutation. The coordinates are stored as a list of structs. A different program was generated for each tensor order (3, 4, and 5), such that the structs held the correct number of coordinate values. In order to run all of the tensors that were being tested, the size of the integers storing each coordinates value was 32.

Listing 6.1: The struct type used to store each coordinate and its value.

```
1 struct coo_t
2 {
3     int32_t idx0;
4     int32_t idx1;
5     int32_t idx2;
6     double  val;
7 };
```

The histogram sort and bucketed histogram sort described in 4 were used as the sorting primitives for radix and k-sadilla. I generated the sorting strategies for quicksort, radix, k-sadilla, and q-sadilla, a combination of k-sadilla and quicksort.

6.1.1 Quicksort

The quicksort strategy was implemented by calling the C++ standard `qsort`. The comparison function passed to `qsort` compares each coordinate in the permutation order.

Listing 6.2: C++ code for the `qsort` comparison function with dimension 0 as the only key.

```
1  int cmp_0(const void *p, const void *q)
2  {
3      struct coo_t * r = (struct coo_t *)p;
4      struct coo_t * s = (struct coo_t *)q;
5      if(r->idx0 < s->idx0) {
6          return -1;
7      } else if (r->idx0 > s->idx0) {
8          return 1;
9      }
10     return 0;
11 }
```

6.1.2 Radix

The radix sort implementation concatenated histogram sort primitives to sort the dimensions by the least significant digit first. A different function was generated for each permutation.

6.1.3 K-sadilla

The k-sadilla algorithm was used to first determine the minimal sort strategy for every permutation. A different function was generated for each permutation, implementing the k-sadilla sort strategy for that permutation. The histogram sort was used when there were no coordinates to bucket on, otherwise the bucketed histogram sort was used.

6.1.4 Combining Quicksort and k-sadilla (eg q-sadilla)

I also implemented another sorting strategy that extended the idea of SPLATT. This strategy uses k-sadilla to create buckets out of the top q levels using minimal sorting passes, and then uses standard `qsort` for the buckets created. We generated this strategy for $b = [1, k)$.

If we have $\sigma = (2, 1, 0)$ and $L = 1$, then 1-sadilla will sort on dimension 2 and then perform a quicksort in every bucket created by distinct i_2 coordinates. However, if we have $\sigma = (0, 2, 1)$ and $b = 1$, then 1-sadilla will bucket on 0 without sorting on it first, since it is already sorted.

6.2 Comparison with other Frameworks (SPLATT)

I timed the SPLATT sorting implementation to compare to my implementation of the different sorting strategies. I added an additional command to SPLATT that would allow us to run the sort command from the command line and specify a permutation. This command loaded the tensor and called the `sort` function defined in SPLATT. It was built with `./configure --no-openmp` to get a sequential implementation.

6.3 Experimental Setup

I ran the different algorithms on 19 tensors from the FROSTT dataset [9]. The three largest tensors were excluded from these results, as I was unable to get them running.

Table 6.1: FROSTT Tensors

filename	order	nnz	dimensions	density	coo storage
flickr-3d.tns	3	112890310	319686x28153045x1607191	7.804412e-12	3.36GB
matmul-2-2-2.tns	3	8	4x4x4	1.250000e-01	256.00B
matmul-3-3-3.tns	3	27	9x9x9	3.703704e-02	864.00B
matmul-4-3-2.tns	3	24	6x8x12	4.166667e-02	768.00B
matmul-4-4-3.tns	3	48	12x12x16	2.083333e-02	1.50KB
matmul-4-4-4.tns	3	64	16x16x16	1.562500e-02	2.00KB
matmul-5-5-5.tns	3	125	25x25x25	8.000000e-03	3.91KB
matmul-6-3-3.tns	3	54	9x18x18	1.851852e-02	1.69KB
nell-1.tns	3	143599552	2902330x2143368x25495389	9.054155e-13	4.28GB
nell-2.tns	3	76879419	12092x9184x28818	2.402239e-05	2.29GB
vast-2015-mc1-3d.tns	3	26021854	165427x11374x2	6.914943e-03	794.12MB
chicago-crime-comm.tns	4	5330673	6186x24x77x32	1.457203e-02	203.35MB
delicious-4d.tns	4	140126220	532924x17262471x2480308x1443	4.255781e-15	5.22GB
enron.tns	4	54202099	6066x5699x244268x1176	5.458096e-09	2.02GB
flickr-4d.tns	4	112890310	319686x28153045x1607191x731	1.067635e-14	4.21GB
nips.tns	4	3101609	2482x2862x14036x17	1.829883e-06	118.32MB
uber.tns	4	3309490	183x24x1140x1717	3.849671e-04	126.25MB
lbnl-network.tns	5	1698825	1605x4198x1631x4209x868131	4.230708e-14	77.77MB
vast-2015-mc1-5d.tns	5	26021945	165427x11374x2x100x89	7.769626e-07	1.16GB

The tensors all began in the initial permutation $\sigma = (0, ..., k-1)$ and were permuted to all $k!$ permutations. The results used are the minimum of 100 consecutive iterations on a warm cache.

6.4 Results

k-sadilla was the best strategy for 66% of all tensor and permutation combinations. The improvement over SPLATT was at least 30% for half of the trials.

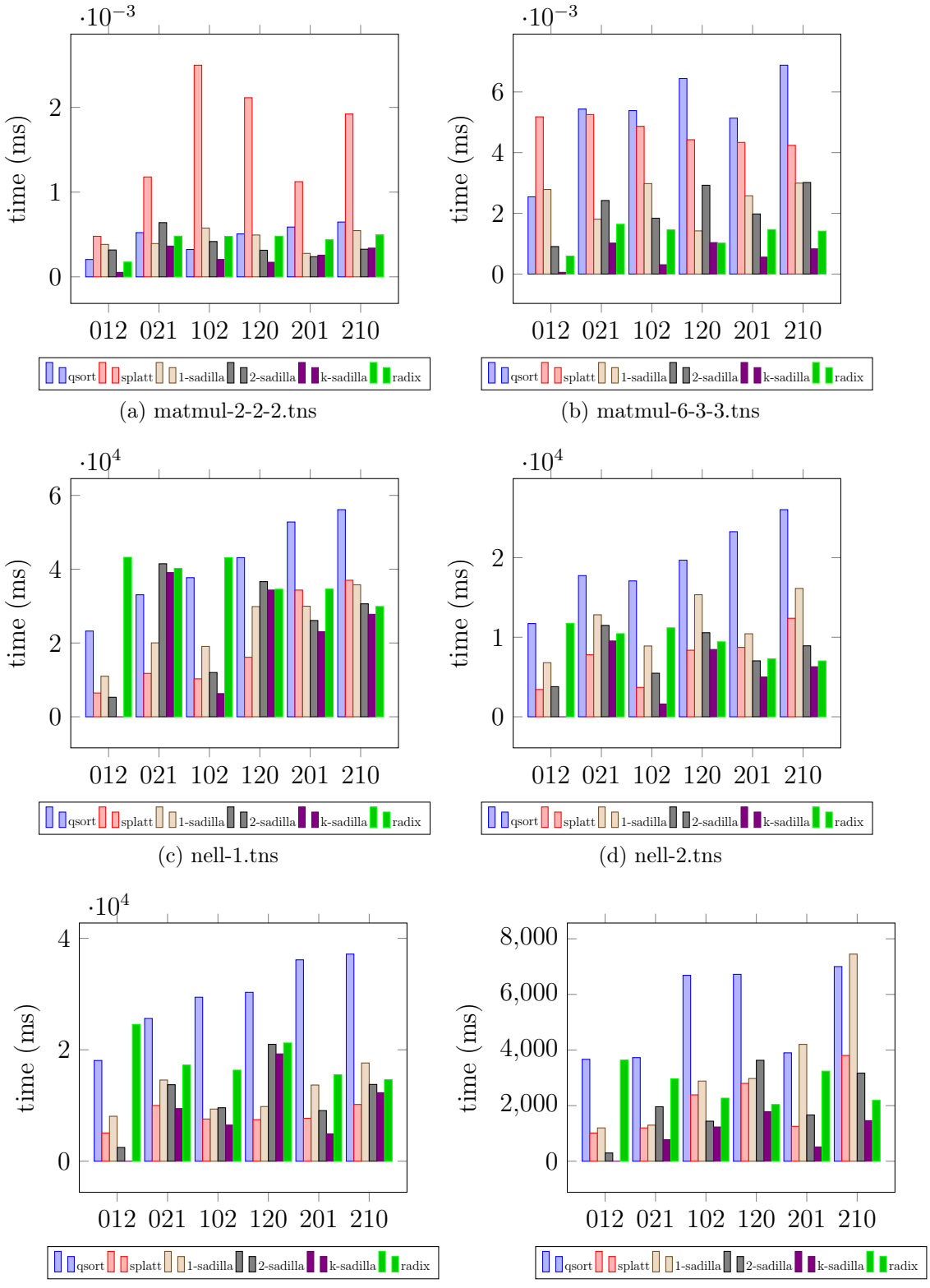
Overall there was some variation in performance across different tensors. K-sadilla

performed very well on some tensors. For five of the seven smallest tensors, k-sadilla was the best strategy for every single permutation. Those that did the best with k-sadilla, also tended to have smaller dimensions.

Table 6.2: Aggregate timing results.
Values were normalized by the SPLATT runtime for that trial. These statistics are aggregated across all possible output permutations for the FROSTT tensors.

stat	qsort	splatt	1-sadilla	k-sadilla	radix
min	0.13	1.00	0.23	0.00	0.10
Q1	1.69	1.00	1.11	0.43	0.88
median	2.04	1.00	1.39	0.70	1.16
Q3	2.67	1.00	1.60	1.10	1.85
max	6.42	1.00	3.35	4.61	6.70
wins (%)	0%	26%	6.9%	66%	1.1%

Figure 6-1: Runtimes for all permutations for order 3 tensors.



(a) matmul-2-2-2.tns (b) matmul-6-3-3.tns

(c) nell-1.tns (d) nell-2.tns

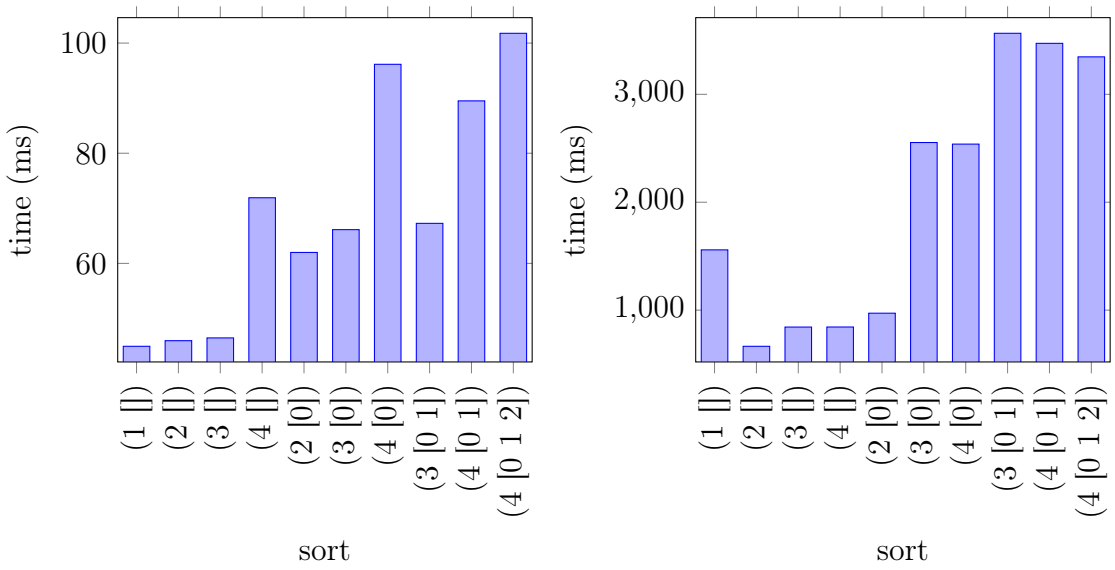
(e) flickr-3d.tns (f) vast-2015-mc1-3d.tns

These plots visualize the runtimes of the different sorting strategies for all permutations. The permutation labels indicate the output order and all tensors start in order (0, 1, 2).

6.4.1 Pass Timing

I also looked at the timings of the permutations that required k-sadilla to perform only a single pass in order to get an idea of per pass performance.

Figure 6-2: Single pass runtime.



(a) Single pass runtime for lbnl-network.tns. (b) Single pass runtime for vast-2015-mc1-5d.tns. These plots show the runtimes for the permutations of a 5 dimension tensor that require a single sorting pass. They are sorted from least number of bucketed dimensions to most number of bucketed dimensions. A lower bar indicates a faster runtime, which is better. There are no sorts on dimension 0, since it is already sorted in the initial lexicographic order and no permutations require a sort on it.

6.4.2 Histogram Sort Runtime

There is a correlation between the histogram sort runtimes and the size of the dimension being sorted on. Some variation in dimension sizes did not cause significant changes. The longer runtimes from histogram sorts on larger dimensions was expected. Asymptotically, these are going to require more computation and more space. Allocating this space as well as accessing random indices within that larger space may account for additional slowdowns.

The differences in the histogram pass runtimes can account for some of the slower

runtimes that we see for certain permutations. For example, lbnl-network has some of the worse aggregate statistics . Looking at its dimensions reveals that the last dimension is much larger than the others ($1605 \times 4198 \times 1631 \times 4209 \times 868131$). The per pass graph in Figure 6-3a reveals a much slower runtime for the histogram sort on dimension 4.

If we split the permutations up into those that will run a sort on the last dimension and those that will not, we see a big change in the statistics. k-sadilla performs best out of all the algorithms for every permutation that does not do a sort on dimension 4. The runtimes for these permutations are all at least 38% better than SPLATT. On the other hand, k-sadilla is the best of just 41.7% of the dimensions that require a sort on dimension 4. Even with this slow dimension, k-sadilla performs as well as SPLATT on half of the permutations.

Table 6.3: lbnl-network.tns statistics for permutations that sort dimension 4

stat	qsort	splatt	1-sadilla	k-sadilla	radix
min	1.48	1.00	1.07	0.61	0.90
Q1	1.98	1.00	1.46	0.88	1.11
median	2.31	1.00	1.51	1.03	1.31
Q3	3.44	1.00	1.68	1.40	1.79
max	5.24	1.00	2.02	3.33	2.83
wins	0%	58.3%	0%	41.7%	0%

Table 6.4: lbnl-network.tns4 statistics for permutations that do not sort dimension 4

stat	qsort	splatt	1-sadilla	k-sadilla	radix
min	1.44	1.00	0.98	0.00	0.86
Q1	1.66	1.00	1.25	0.36	0.89
median	1.78	1.00	1.35	0.41	0.96
Q3	2.06	1.00	1.43	0.51	1.10
max	2.81	1.00	1.87	0.62	1.45
wins	0%	0%	0%	100%	0%

6.5 Bucketed Histogram Sort Runtime

The time to run a bucketed histogram pass is always larger than the time to run a histogram pass, since the histogram pass is a subprotocol of the bucketed histogram.

I was unable to find what caused the changes in the runtime for different buckets. There does not appear to be a clear correlation between the change in the number of buckets and the time needed to do the bucketing. In some cases, having more buckets actually resulted in a faster computation. A closer analysis would be needed to see what causes these changes.

The bucketed histogram sort is often significantly slower than its histogram counterpart. This disparity raises the question of whether to consider avoiding these all together. This change would result in only being able to omit passes at the way bottom or instead using a comparison sort for the remaining levels when a bucketed histogram sort would be required.

However, it is possible that a different implementation could also be used in practice to improve the runtime. The bucketed histogram sort was chosen since the large histogram size would dominate if we were to perform a histogram sort within each bucket (since buckets are often small). We can swap this out with either a radix with a smaller digit or a stable comparison sort within the buckets.

6.5.1 SPLATT and 1-sadilla

SPLATT and 1-sadilla implement nearly the same strategy. Section 2.3.1 explains that SPLATT will do a histogram sort on the first dimension in the output order and then quicksort each of the buckets created. 1-sadilla finds the best algorithm for the first level and then runs quicksort on each of the buckets. 1-sadilla will only do the histogram sort on the first dimensions if necessary. In the results, we can see that SPLATT outperforms 1-sadilla in 79.8% of the trials. Even when there is no histogram pass, SPLATT is still faster in 62.3% of the trials. We would expect 1-sadilla to outperform SPLATT in these cases, since it can skip some of the work. This indicates that the SPLATT quicksort implementation is faster than the C++ standard

`qsort` for the tensors and permutations when 1-sadilla loses despite doing less work. We could expect to match SPLATT by switching to the SPLATT implementation of quicksort.

Chapter 7

Conclusion

7.1 Summary

This project shows that we can transpose sparse tensors faster than sorting arbitrary coordinates. By taking advantage of the ordering that already exists, we can apply selective passes of a radix sort and thereby reduce the amount of work to sort the coordinates. We showed that this improves performance using a C++ implementation. These passes can be replaced with more optimized versions as well.

7.2 Future Work

There are several additional opportunities for optimizations that we have not implemented but that would provide additional improvements.

7.2.1 Cost Function Improvements

In this paper, we considered a histogram pass and a bucketed histogram pass to have the same cost. We also did not take the size of the dimension into account. It could be worth exploring improvements to this cost function in the future. This may also allow us to incorporate comparison sorts into strategy selection to sort some dimensions of the tensor.

7.2.2 Pass Optimizations

The size of a dimension greatly affects the runtime of the the histogram sort. Asymptotically, the histogram sort runs in $O(N + d)$, where N is the number of nonzeros, and d is the size of the dimension. We can replace this histogram sort with a radix sort that uses a smaller digit size b. This will change the asymptotic complexity to $O(\log_b(d)(N + b))$. Since these dimensions may be quite large, switching to a radix sort may provide significant speedup, especially if the count array for this smaller digit can fit in the cache.

7.2.3 Pass Combinations

The algorithm described in Section 5.2.1 chooses to switch to additional buckets when there is a pass that could be skipped.

The current implementation will move each coordinate back to its bucket after each pass. This ensures that the lexicographical ordering is as expected after performing that pass. However, it is not necessary to move the coordinates back so soon, since each pass after that will also take place within those buckets. Instead, these buckets can be applied at the end.

This modification will reduce the cost of multiple bucketed histogram passes. This seems like a more reasonable cost tradeoff since a bucketed histogram sort is roughly two histogram sorts. By incurring this cost just once, we will make it more worthwhile to switch to a new bucket.

This strategy will require more space, since the information needs to be kept around until the sorting is complete.

7.2.4 Bit Packing Optimizations

By having control over the intermediate format, we can apply further optimizations to the way we store the coordinates. Assuming that we know the size of each of the dimensions at compile time, we can pack the coordinates into fewer integers. This can also help improve performance of comparison sorts, as less comparisons need to

be made in order to do the sort.

7.3 Intermediate Formats

The intermediate steps that sort the coordinates of the tensor operate on a list of coordinates. However, the input and output formats may not be in coordinate format. Turning the sparse tensor into coordinate format requires $O(k * N)$ space for the coordinate values.

The required space is acceptable since we must be using at least $O(N)$ space to store the coordinates in both the input and output formats, and k is a small constant.

However, we may also be able to reduce both the space needed for the intermediate formats as well as the time spent sorting the coordinates. In order to see this, we need to understand the structure of these sparse tensor formats. CSF format has a position and index array for every dimension of the tensor. The method creates a type of tree structure, where each index that appears in each dimension is a node. The children of each index can be found by looking at the corresponding positions in the next level of the tensor.

While a permutation may require a complete rebuild of this tree structure, certain permutations will not need to rebuild the whole structure and others can iterate over the tree structure instead of the list of coordinates. This may save time for unpacking, space used in the intermediate format and work done to sort the coordinates. This section describes an idea for an intermediate transpose format that combines COO and CSF to accomplish these goals.

7.3.1 Grouping Dimensions

We can group the dimensions of a tensor transpose to understand which coordinates need to be materialized.

Sorting passes move dimensions up. If there is a boundary at index j where no dimensions move across, then any sorts that happen on the dimensions below that level will always occur in buckets where the first j dimensions all have the same value.

In the existing tree structure, we can easily find the roots of each of these buckets. In the result tree, the coordinates and values in each of these buckets will stay in the same buckets, even if the dimensions above $< j$ or below $> j$ are reordered.

7.3.2 Creating Smaller Problems

To reduce sorting work and space used there are a two key optimizations that we can make: we do not need to duplicate coordinates across a boundary and we only need to turn a groups into coordinate format if they contain a sort. We do this by turning each grouping into a coordinate list, and using a *pos* array to keep track of which coordinates belong in which subtrees.

For example if only the bottom dimensions are being sorted, we do not need to unpack the top of the tree. We can leave it as is during the sorting process. Figure 7-1 shows the intermediate format that avoids unpacking the first dimension of the input tensor when transposing.

7.3.3 Adding Pointers

In CSF format, the position of the coordinate at level i determines where to find the positions of the coordinates at level $i + 1$. If we sort the coordinates above a boundary, we would need to move its subtrees. However, in the contiguous CSF arrays, we cannot do this easily. In order to know where to move its subtree, we need to know how much space the coordinates before it in the new order will take up, which we cannot know until we have the ordering of all the coordinates that appear before it. We can solve this problem by introducing an extra level that stores its index as a pointer to its children. Since this is a boundary, these subtrees will remain intact even if they are reordered within each subtree. Figure 7-2 shows an example that uses pointers to index into the input tensors last dimension to pack the tensor.

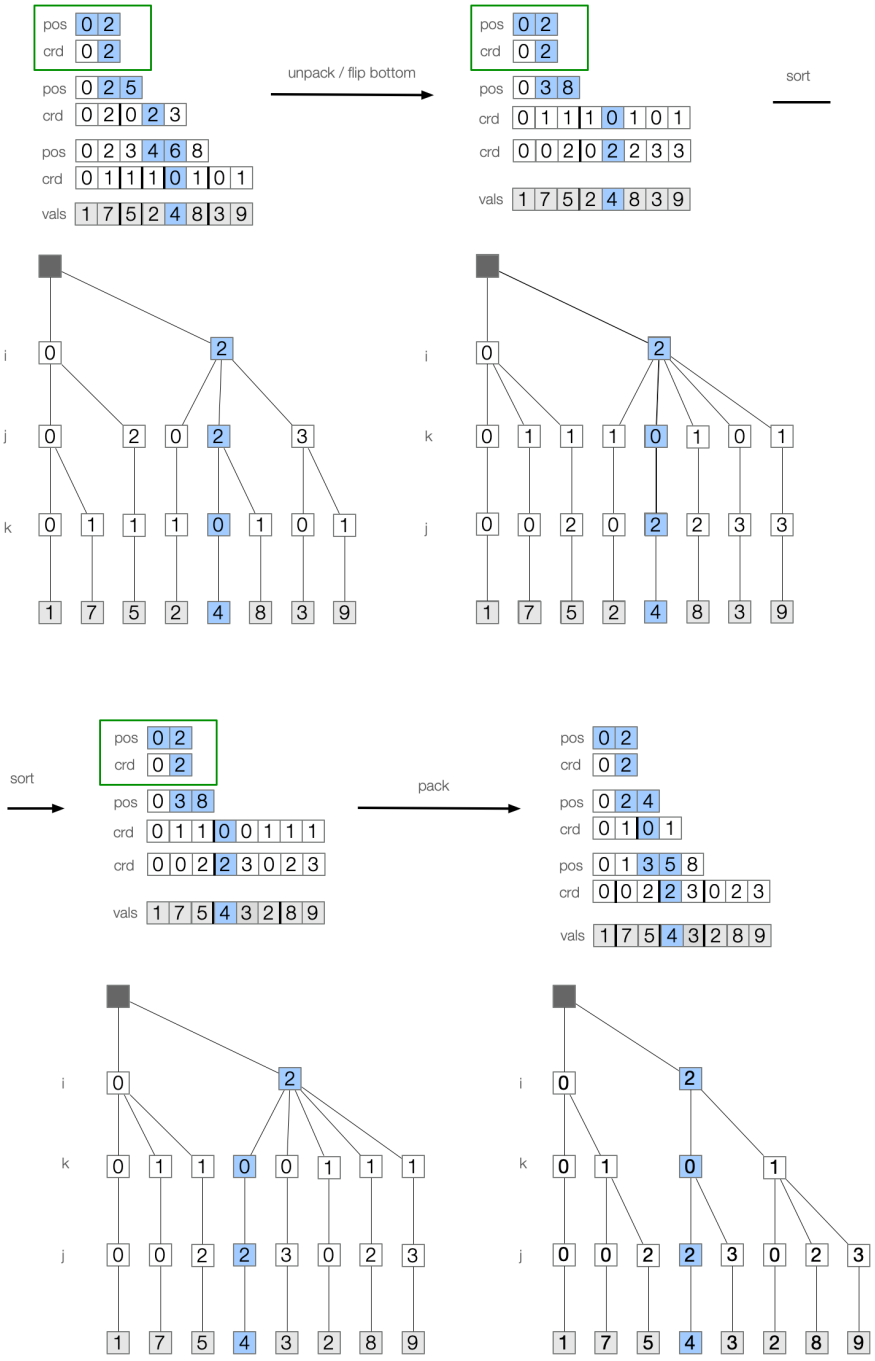


Figure 7-1: An example of the intermediate format that can be used to transpose to the permutation (0, 2, 1). The arrays boxed in green are all from the input tensor. The coordinates from dimension 0 are not turned into coordinates. No new space is needed to store dimension 0 since it will not be affected by the sort.

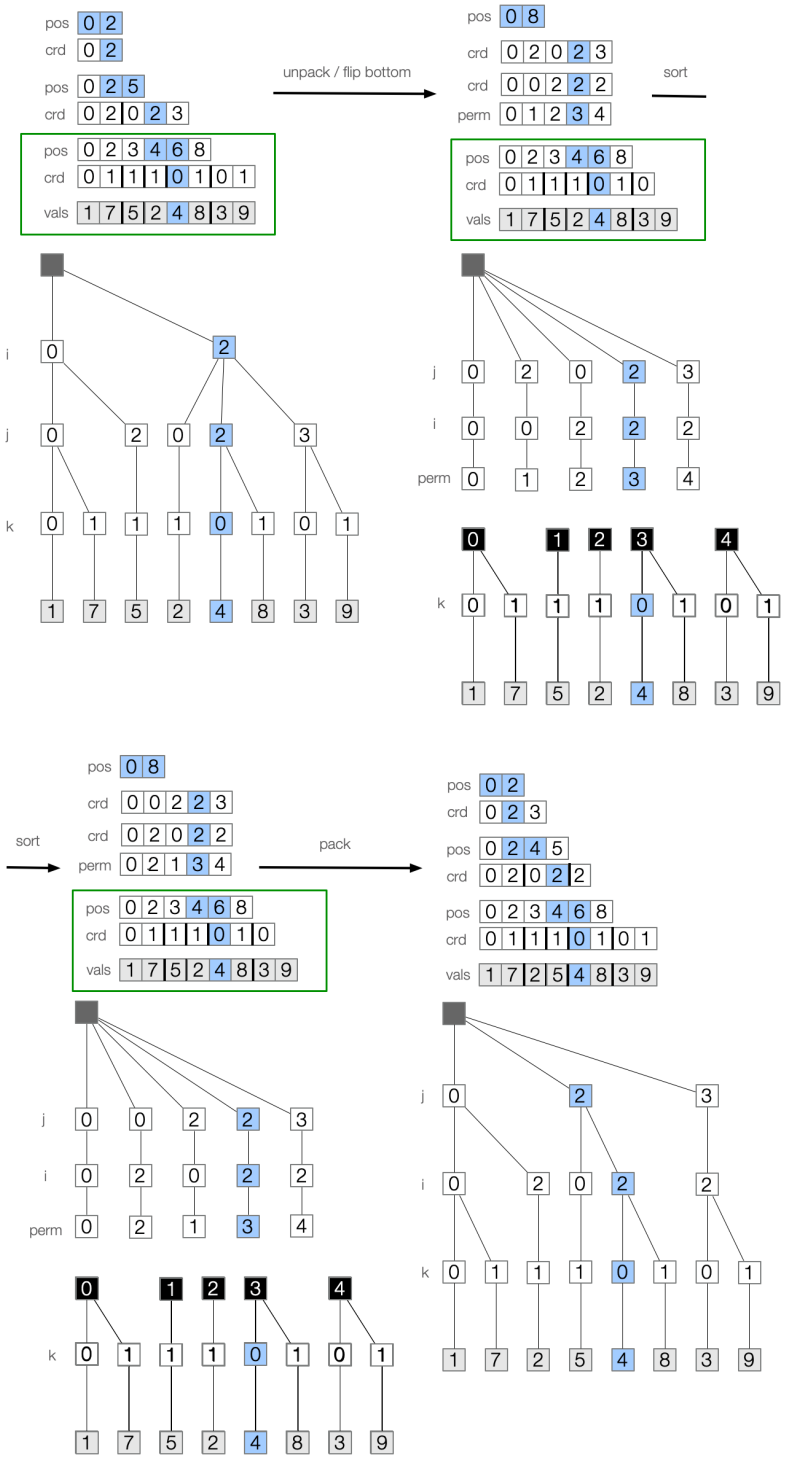


Figure 7-2: An example of the intermediate format that can be used to transpose to the permutation (1, 0, 2). The perm array is used as a pointer to the subtree that contains the remaining dimensions. We can then iterate over the appropriate subtree. The arrays boxed in green are all from the input tensor. No new space is needed to store dimension 2 since it will not be affected by the sort.

7.4 Parallelism

There are many opportunities to make the k-sadilla sorting strategy parallel.

There are parallel versions of all of the sorting algorithms that we considered in this paper. In particular, each radix pass can be implemented in parallel. By adding this parallel implementation to each pass, we can parallelize the k-sadilla implementation. SPLATT already implements a parallelized histogram sort.

Additionally, the sort schedule takes advantage of existing structure, producing a sort schedule that will sort on existing buckets. Each bucket can be sorted in parallel, since there will be no movement across the boundaries. Producing the schedule in this way takes advantage of the structure that is easily parallelized.

The groups from the intermediate formats described above can all be sorted in parallel, as they are separate sorting problems. This allows for parallelization between groupings, in addition to the the parallelization that we can add for each pass.

Appendix A

Tables

To understand the performance of the different algorithms, we normalized the results by the SPLATT runtime for each tensor and permutation. A smaller number means that the algorithm was faster.

A.1 Statistics

These tables contain statistics about the performance of the algorithms across all permutations and tensors. In addition, we counted the number of times that each strategy was the best of all of the strategies. We exclude 2-sadilla and 3-sadilla from these results, as the strategy is not comparable across tensors of different orders.

Table A.1: Aggregate timing results

stat	qsort	splatt	1-sadilla	k-sadilla	radix
min	0.13	1.00	0.23	0.00	0.10
Q1	1.69	1.00	1.11	0.43	0.88
median	2.04	1.00	1.39	0.70	1.16
Q3	2.67	1.00	1.60	1.10	1.85
max	6.42	1.00	3.35	4.61	6.70
wins (%)	0%	26%	6.9%	66%	1.1%

Table A.2: Median results by tensor

filename	qsort	splatt	1-sadilla	k-sadilla	radix
flickr-3d.tns	3.77	1.00	1.52	0.90	2.09
matmul-2-2-2.tns	0.38	1.00	0.27	0.13	0.31
matmul-3-3-3.tns	1.06	1.00	0.58	0.14	0.31
matmul-4-3-2.tns	0.84	1.00	0.32	0.14	0.29
matmul-4-4-3.tns	1.21	1.00	0.45	0.10	0.32
matmul-4-4-4.tns	0.91	1.00	0.43	0.13	0.25
matmul-5-5-5.tns	2.04	1.00	0.76	0.14	0.33
matmul-6-3-3.tns	1.15	1.00	0.57	0.16	0.31
nell-1.tns	2.74	1.00	1.70	0.71	2.78
nell-2.tns	2.51	1.00	1.74	0.54	1.23
vast-2015-mc1-3d.tns	2.95	1.00	1.19	0.46	1.71
chicago-crime-comm.tns	1.93	1.00	1.55	0.47	0.82
delicious-4d.tns	2.37	1.00	1.34	1.02	1.63
enron.tns	2.19	1.00	1.39	0.58	1.06
flickr-4d.tns	2.97	1.00	1.32	0.98	1.79
nips.tns	2.51	1.00	1.67	0.75	1.73
uber.tns	2.14	1.00	1.64	0.60	1.07
lbnl-network.tns	2.11	1.00	1.49	0.95	1.28
vast-2015-mc1-5d.tns	1.74	1.00	1.24	0.67	1.13

Table A.3: Wins by file

filename	qsort	splatt	1-sadilla	k-sadilla	radix
flickr-3d.tns	0%	33.3%	0%	66.7%	0%
matmul-2-2-2.tns	0%	0%	0%	100%	0%
matmul-3-3-3.tns	0%	0%	0%	100%	0%
matmul-4-3-2.tns	0%	0%	0%	100%	0%
matmul-4-4-3.tns	0%	0%	0%	100%	0%
matmul-4-4-4.tns	0%	0%	0%	66.7%	33.3%
matmul-5-5-5.tns	0%	0%	0%	100%	0%
matmul-6-3-3.tns	0%	0%	0%	83.3%	16.7%
nell-1.tns	0%	33.3%	0%	66.7%	0%
nell-2.tns	0%	33.3%	0%	66.7%	0%
vast-2015-mc1-3d.tns	0%	0%	0%	100%	0%
chicago-crime-comm.tns	0%	8.33%	0%	91.7%	0%
delicious-4d.tns	0%	54.2%	0%	45.8%	0%
enron.tns	0%	12.5%	0%	83.3%	4.17%
flickr-4d.tns	0%	41.7%	0%	58.3%	0%
nips.tns	0%	25%	12.5%	62.5%	0%
uber.tns	0%	16.7%	0%	83.3%	0%
lbnl-network.tns	0%	46.7%	0%	53.3%	0%
vast-2015-mc1-5d.tns	0%	14.2%	23.3%	61.7%	0.833%

A.2 Results

These tables contain the results of running all of the experiments. They are organized by file and the permutations are ordered lexicographically. A cell that contains a value of 1 is colored white. This value means that it performed as well as SPLATT. A cell that contains a value > 1 is colored red and performed worse than SPLATT. A cell that contains a value < 1 is colored blue and performed better than SPLATT.

Figure A-1: flickr-3d.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	3.56	1	1.59	0.49	0	4.84
021	2.56	1	1.46	1.37	0.94	1.73
102	3.89	1	1.24	1.27	0.86	2.16
120	4.06	1	1.31	2.81	2.58	2.84
201	4.69	1	1.77	1.18	0.64	2.01
210	3.65	1	1.73	1.35	1.21	1.44

Figure A-2: matmul-2-2-2.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	0.43	1	0.8	0.66	0	0.37
021	0.44	1	0.33	0.54	0.31	0.41
102	0.13	1	0.23	0.17	$8.21 \cdot 10^{-2}$	0.19
120	0.24	1	0.23	0.15	$8.14 \cdot 10^{-2}$	0.23
201	0.52	1	0.25	0.21	0.23	0.39
210	0.34	1	0.28	0.17	0.18	0.26

Figure A-3: matmul-3-3-3.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	0.41	1	0.47	0.41	0	0.16
021	0.55	1	0.31	0.42	0.15	0.19
102	1.11	1	0.63	0.42	0.12	0.32
120	2.94	1	1.51	1.65	0.72	0.84
201	1.1	1	0.6	0.47	0.13	0.33
210	1.01	1	0.56	0.59	0.24	0.31

Figure A-4: matmul-4-3-2.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	0.96	1	1.34	0.94	0	0.39
021	0.48	1	0.38	0.41	0.14	0.24
102	0.84	1	0.25	0.37	0.11	0.31
120	0.97	1	0.27	0.59	0.24	0.33
201	0.84	1	0.27	0.22	0.13	0.17
210	0.75	1	0.42	0.35	0.16	0.27

Figure A-5: matmul-4-4-3.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	1.16	1	1.76	1.14	0	0.46
021	0.5	1	0.27	0.29	0.1	0.32
102	1.47	1	0.34	0.45	$9.9 \cdot 10^{-2}$	0.15
120	1.29	1	0.59	0.55	0.41	0.61
201	1.26	1	0.5	0.23	$9.9 \cdot 10^{-2}$	0.31
210	0.63	1	0.39	0.31	0.11	0.2

Figure A-6: matmul-4-4-4.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	0.56	1	0.55	0.44	0	0.14
021	1.01	1	0.24	0.27	0.13	0.29
102	0.81	1	0.36	0.24	0.13	0.31
120	2.81	1	1.21	1.63	0.43	0.4
201	0.69	1	0.24	0.34	$6.39 \cdot 10^{-2}$	0.21
210	1.02	1	0.5	0.47	0.13	0.1

Figure A-7: matmul-5-5-5.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	2.13	1	1.99	1.39	0	0.43
021	0.92	1	0.23	0.39	0.13	0.19
102	6.42	1	2.32	1.49	0.33	0.43
120	2.01	1	0.74	0.62	0.18	0.34
201	1.7	1	0.33	0.4	0.11	0.25
210	2.07	1	0.79	0.69	0.14	0.33

Figure A-8: matmul-6-3-3.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	0.49	1	0.54	0.17	0	0.11
021	1.03	1	0.34	0.46	0.19	0.31
102	1.11	1	0.61	0.38	$6.31 \cdot 10^{-2}$	0.3
120	1.46	1	0.32	0.66	0.23	0.23
201	1.18	1	0.59	0.46	0.13	0.34
210	1.62	1	0.71	0.71	0.2	0.33

Figure A-9: nell-1.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	3.6	1	1.7	0.82	0	6.7
021	2.81	1	1.7	3.53	3.32	3.42
102	3.67	1	1.85	1.17	0.61	4.19
120	2.67	1	1.85	2.27	2.13	2.14
201	1.54	1	0.87	0.76	0.67	1.01
210	1.52	1	0.97	0.83	0.75	0.81

Figure A-10: nell-2.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	3.42	1	1.98	1.1	0	3.42
021	2.28	1	1.65	1.47	1.22	1.34
102	4.62	1	2.41	1.48	0.43	3.02
120	2.35	1	1.83	1.26	1.01	1.13
201	2.67	1	1.2	0.81	0.57	0.83
210	2.1	1	1.3	0.72	0.51	0.57

Figure A-11: vast-2015-mc1-3d.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
012	3.61	1	1.18	0.29	0	3.59
021	3.11	1	1.08	1.64	0.65	2.47
102	2.8	1	1.21	0.6	0.52	0.95
120	2.4	1	1.06	1.3	0.64	0.73
201	3.1	1	3.35	1.33	0.41	2.58
210	1.84	1	1.96	0.83	0.38	0.58

Figure A-12: chicago-crime-comm.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
0123	2.45	1	1.09	0.73	0.29	0	2.35
0132	1.98	1	1.11	0.83	0.98	0.72	1.06
0213	1.79	1	1.01	0.84	0.68	0.51	1.34
0231	1.83	1	1.11	0.99	1.23	1.04	1.29
0312	1.78	1	1.12	0.74	0.53	0.32	0.94
0321	1.98	1	1.29	1.07	0.81	0.7	1.23
1023	3.44	1	1.95	0.99	0.61	0.35	2.03
1032	2.65	1	1.72	0.98	0.93	0.77	1.04
1203	1.62	1	1.53	0.75	0.36	0.25	0.67
1230	1.65	1	1.54	1.13	0.83	0.5	0.56
1302	1.8	1	1.62	0.79	0.37	0.22	0.53
1320	1.8	1	1.66	1.26	0.61	0.29	0.48
2013	3.9	1	1.84	0.88	0.63	0.34	1.93
2031	2.8	1	1.55	0.83	1.14	0.93	1.33
2103	1.9	1	1.45	0.91	0.53	0.48	0.67
2130	1.87	1	1.46	1.32	0.81	0.49	0.56
2301	1.98	1	1.6	1.02	0.61	0.48	0.7
2310	1.92	1	1.56	1.4	0.75	0.49	0.55
3012	4.37	1	2.07	1.05	0.69	0.32	1.61
3021	2.83	1	1.66	0.97	1.35	1.16	1.29
3102	2.06	1	1.53	0.85	0.4	0.24	0.56
3120	1.91	1	1.47	1.29	0.6	0.29	0.48
3201	1.91	1	1.58	0.7	0.33	0.22	0.61
3210	1.94	1	1.62	1.13	0.76	0.47	0.49

Figure A-13: delicious-4d.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
0123	2.88	1	1.17	0.38	0.26	0	4.45
0132	3.05	1	1.25	0.41	1.9	1.53	4.1
0213	2.33	1	1.33	1.38	1.17	1.03	2.11
0231	2.33	1	1.35	1.44	1.85	1.64	2.34
0312	2.29	1	1.16	1.12	0.95	0.68	2.6
0321	2.28	1	1.25	1.29	2.12	1.92	2.57
1023	2.14	1	1.08	0.87	0.79	0.69	1.65
1032	2.15	1	1.04	0.88	1.51	1.34	1.7
1203	1.89	1	1.1	1.18	1.1	1.01	1.45
1230	1.9	1	1.1	1.26	1.25	1.15	1.35
1302	1.9	1	1.06	0.75	0.73	0.56	1.24
1320	1.83	1	1.08	0.76	1.15	1.05	1.23
2013	3.43	1	1.37	0.78	0.62	0.48	1.97
2031	3.04	1	1.33	0.78	1.83	1.64	1.94
2103	2.53	1	1.49	1.16	1.08	0.98	1.28
2130	2.47	1	1.5	1.22	1.02	0.92	1.1
2301	2.26	1	1.39	0.76	0.6	0.49	1.46
2310	2.41	1	1.53	1.03	1.23	1.13	1.17
3012	4.32	1	1.85	0.89	0.66	0.35	3.65
3021	3.33	1	1.63	0.88	2.16	1.99	2.67
3102	2.4	1	1.63	1.13	1.1	0.85	1.55
3120	2.44	1	1.67	1.22	1.53	1.4	1.59
3201	2.34	1	1.67	0.87	0.71	0.57	1.61
3210	2.5	1	1.83	1.15	1.39	1.28	1.31

Figure A-14: enron.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
0123	2.63	1	1.68	1.12	0.32	0	2.43
0132	1.99	1	1.48	1.09	0.74	0.33	1.16
0213	1.69	1	1.24	0.93	0.73	0.58	1.06
0231	1.67	1	1.25	1.1	1.13	0.87	0.93
0312	1.75	1	1.39	0.75	0.58	0.23	0.93
0321	1.69	1	1.37	0.95	0.96	0.73	0.92
1023	3.63	1	1.8	1.46	0.65	0.35	2.33
1032	2.52	1	1.51	1.31	1	0.57	1.15
1203	2.08	1	1.39	1.13	1.04	0.83	1.13
1230	2.18	1	1.47	1.27	1.28	1.1	1.3
1302	2.23	1	1.54	0.72	0.69	0.28	0.94
1320	2.3	1	1.59	0.79	1.08	1.01	1.19
2013	2.74	1	1.26	0.91	0.63	0.42	1.07
2031	2.18	1	1.19	0.93	1.21	0.97	0.81
2103	1.9	1	1.16	0.76	0.69	0.52	0.69
2130	2	1	1.24	0.86	0.77	0.68	0.84
2301	1.86	1	1.25	0.76	0.74	0.46	0.74
2310	1.93	1	1.3	0.89	0.8	0.64	0.72
3012	4.55	1	1.83	1.46	1.07	0.32	2.03
3021	2.57	1	1.33	1.08	1.21	0.96	1.06
3102	3.09	1	1.54	1	0.95	0.41	1.24
3120	3.04	1	1.58	1.04	1.43	1.31	1.49
3201	2.2	1	1.33	0.82	0.78	0.49	0.81
3210	2.37	1	1.43	0.96	0.91	0.86	0.96

Figure A-15: flickr-4d.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
0123	2.98	1	1.27	0.41	0.25	0	4.26
0132	2.96	1	1.3	0.41	1.22	0.76	2.95
0213	2.2	1	1.26	1.14	0.9	0.81	1.45
0231	2.09	1	1.19	1.12	1.38	1.12	1.5
0312	2.92	1	1.29	1.55	1.04	0.69	3.91
0321	2.1	1	1.15	1.29	1.65	1.38	1.87
1023	3.35	1	1.02	1.05	0.89	0.73	2.02
1032	3.33	1	1.06	1.05	1.3	0.98	2.11
1203	3.41	1	1.11	2.33	2.33	2.18	2.67
1230	3.26	1	1.08	2.38	2.66	2.37	2.57
1302	3.5	1	1.1	1.03	1.08	0.76	1.81
1320	3.4	1	1.13	1.07	1.76	1.57	1.79
2013	3.8	1	1.41	0.98	0.65	0.5	1.66
2031	3.82	1	1.41	1.01	1.84	1.46	1.82
2103	3.19	1	1.53	1.14	1.19	1.04	1.38
2130	3.09	1	1.48	1.14	1.1	0.99	1.18
2301	2.66	1	1.34	0.9	0.81	0.54	1.31
2310	2.76	1	1.41	1.01	1.18	1	1.11
3012	4.24	1	2.08	1.15	0.71	0.34	3.56
3021	2.89	1	1.63	1.02	1.53	1.25	1.63
3102	2.38	1	1.72	0.85	0.8	0.64	1.3
3120	2.47	1	1.74	0.85	1.75	1.65	1.79
3201	2.2	1	1.64	0.79	0.71	0.5	1.16
3210	2.32	1	1.73	0.92	1.06	1	1.04

Figure A-16: nips.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
0123	3.44	1	1.67	1.46	0.3	0	3.34
0132	3.27	1	1.61	1.41	2.04	0.66	2.95
0213	0.83	1	0.43	0.75	0.71	0.65	0.86
0231	0.84	1	0.43	0.74	0.9	0.8	0.85
0312	3.3	1	1.63	2.26	2.06	0.67	3.06
0321	0.6	1	0.31	0.41	0.59	0.52	0.59
1023	5.12	1	2.25	2	0.87	0.56	3.62
1032	4.95	1	2.19	1.95	2.38	1.01	2.92
1203	3.41	1	1.75	1.9	1.89	1.48	2.35
1230	3.4	1	1.75	1.91	2.36	2.13	2.05
1302	5.01	1	2.21	2.29	2.23	0.83	3.04
1320	4.15	1	1.93	1.9	2.46	2.2	2.84
2013	2.52	1	1.18	0.71	0.63	0.49	1.09
2031	2.43	1	1.16	0.7	1.02	0.8	1.06
2103	2.14	1	1.4	0.76	0.75	0.63	0.81
2130	2.08	1	1.41	0.7	0.84	0.72	0.82
2301	2.51	1	1.15	1.21	0.98	0.78	1.05
2310	2.1	1	1.25	1.3	0.87	0.77	0.88
3012	3.15	1	2.87	1.97	1.73	0.46	2.91
3021	1.79	1	1.7	1.13	1.83	1.61	1.8
3102	2.42	1	2.29	1.29	1.22	0.46	1.66
3120	2.51	1	2.38	1.35	1.76	1.59	2.01
3201	1.81	1	1.67	0.82	0.64	0.48	0.84
3210	1.98	1	1.92	1.22	0.92	0.82	0.93

Figure A-17: uber.tns results normalized by splatt

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
0123	3.36	1	2.16	1.48	0.62	0	2.79
0132	2.23	1	1.65	1.32	1.06	0.78	1.43
0213	1.91	1	1.4	1.03	0.77	0.45	1.47
0231	1.99	1	1.42	1.31	1.31	1.1	1.64
0312	1.92	1	1.5	0.76	0.6	0.34	1.11
0321	2.09	1	1.64	1.11	1.09	0.9	1.38
1023	4.49	1	2.87	1.91	1.06	0.47	2.62
1032	2.7	1	1.97	1.47	1.43	1.15	1.38
1203	1.87	1	1.68	0.98	0.61	0.36	0.97
1230	1.81	1	1.68	1.34	0.98	0.68	1.04
1302	1.89	1	1.73	0.75	0.47	0.26	0.73
1320	2.02	1	1.83	1.17	0.85	0.6	0.9
2013	3.87	1	1.98	1.3	0.86	0.45	2.2
2031	2.53	1	1.64	1.18	1.38	1.29	1.63
2103	2.12	1	1.48	1.18	0.84	0.61	0.81
2130	1.94	1	1.46	1.49	0.92	0.64	0.97
2301	1.84	1	1.54	0.96	0.7	0.53	0.93
2310	1.9	1	1.6	1.31	1.05	0.71	0.88
3012	4.71	1	2	1.31	0.91	0.45	2.25
3021	2.98	1	1.66	1.2	1.53	1.29	1.67
3102	2.47	1	1.53	1.19	0.86	0.61	0.8
3120	2.31	1	1.46	1.47	0.91	0.57	0.96
3201	2.17	1	1.54	0.95	0.69	0.53	0.92
3210	2.25	1	1.6	1.29	1.04	0.79	0.88

Figure A-18: lbnl-network.tns results normalized by splatt (1)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
01234	1.99	1	1.4	1.31	1.2	1.12	0	1.45
01243	1.85	1	1.31	1.21	1.12	0.89	0.74	1.41
01324	1.62	1	1.12	1.06	1.24	1.21	0.39	0.98
01342	1.7	1	1.21	1.13	1.35	1.08	0.95	1.19
01423	1.85	1	1.31	1.22	0.82	0.76	0.61	1.29
01432	1.81	1	1.33	1.22	0.81	1.16	1.03	1.32
02134	1.45	1	1	1.21	1.16	1.08	0.33	0.97
02143	1.53	1	1.09	1.3	1.25	0.95	0.83	0.99
02314	1.44	1	0.98	1.2	1.37	1.32	0.58	0.96
02341	1.48	1	1.07	1.29	1.49	1.19	1.07	1.1
02413	1.57	1	1.12	1.37	0.89	0.9	0.84	1.15
02431	1.61	1	1.12	1.38	0.97	1.26	1.14	1.07
03124	1.5	1	1.05	1.19	1.11	1.1	0.35	0.89
03142	1.62	1	1.18	1.3	1.24	1.02	0.82	1.09
03214	1.49	1	1.04	1.17	1.31	1.28	0.56	0.93
03241	1.52	1	1.1	1.23	1.41	1.13	1.03	1.05
03412	1.69	1	1.21	1.32	1	0.99	0.87	1.16
03421	1.71	1	1.21	1.35	0.99	1.27	1.15	1.12
04123	2.01	1	1.47	0.84	0.84	0.84	0.7	1.41
04132	2.04	1	1.46	0.83	0.84	1.29	1.15	1.42
04213	2.04	1	1.47	0.86	1.18	1.17	1.04	1.38
04231	2	1	1.45	0.84	1.16	1.51	1.36	1.37
04312	2	1	1.45	0.84	1.19	1.2	1.01	1.4
04321	2.03	1	1.46	0.85	1.21	1.52	1.39	1.41

Figure A-19: lbnl-network.tns results normalized by splatt (2)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
10234	2.55	1	1.82	1.62	1.51	1.41	0.34	1.4
10243	2.26	1	1.65	1.46	1.37	1.08	0.94	1.3
10324	2.16	1	1.52	1.37	1.43	1.44	0.56	1.11
10342	2.31	1	1.69	1.51	1.59	1.38	1.24	1.25
10423	2.29	1	1.7	1.49	1.04	1.04	0.91	1.24
10432	2.3	1	1.69	1.5	0.97	1.24	1.22	1.28
12034	1.79	1	1.41	1.2	1.17	1.1	0.37	0.94
12043	1.77	1	1.43	1.25	1.22	0.98	0.88	1.02
12304	1.67	1	1.31	1.14	1.2	1.19	0.49	0.89
12340	1.79	1	1.46	1.25	1.33	0.95	0.84	0.95
12403	1.85	1	1.5	1.28	0.76	0.76	0.71	1.04
12430	1.87	1	1.51	1.32	0.83	1.05	0.94	1.08
13024	1.72	1	1.39	1.14	1.11	1.07	0.36	0.91
13042	1.95	1	1.54	1.26	1.21	1.07	0.96	1.03
13204	1.69	1	1.37	1.16	1.2	1.19	0.49	0.89
13240	1.89	1	1.49	1.27	1.33	0.95	0.85	0.9
13402	1.99	1	1.6	1.36	0.87	0.88	0.76	1.08
13420	2.01	1	1.62	1.36	0.88	1.05	0.93	1.07
14023	2.31	1	1.86	0.81	0.82	0.81	0.68	1.27
14032	2.34	1	1.88	0.82	0.81	1.24	1.04	1.24
14203	2.29	1	1.9	0.81	1	1.01	0.88	1.25
14230	2.32	1	1.91	0.82	1.01	1.23	1.09	1.27
14302	2.32	1	1.9	0.82	1.08	1.08	0.94	1.31
14320	2.38	1	1.85	0.8	1.07	1.27	1.11	1.3

Figure A-20: lbnl-network.tns results normalized by splatt (3)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
20134	2.7	1	1.75	1.63	1.56	1.45	0.36	1.36
20143	2.4	1	1.6	1.5	1.42	1.05	0.91	1.36
20314	2.06	1	1.36	1.26	1.47	1.41	0.56	1.09
20341	2.3	1	1.5	1.39	1.66	1.32	1.19	1.28
20413	2.33	1	1.58	1.45	1	1	0.87	1.33
20431	2.35	1	1.59	1.44	1.01	1.25	1.1	1.32
21034	2.07	1	1.47	1.43	1.36	1.29	0.44	1.1
21043	1.99	1	1.39	1.35	1.32	1.02	0.91	1.12
21304	1.8	1	1.28	1.24	1.32	1.3	0.54	0.96
21340	2.05	1	1.42	1.38	1.46	0.96	0.93	1.1
21403	2.04	1	1.46	1.41	0.82	0.82	0.77	1.09
21430	2.07	1	1.45	1.42	0.89	1.12	1.01	1.07
23014	1.85	1	1.27	1.24	1.18	1.14	0.39	0.96
23041	2.02	1	1.41	1.43	1.35	1.06	0.95	1.15
23104	1.64	1	1.17	1.22	1.19	1.18	0.49	0.88
23140	1.83	1	1.32	1.32	1.29	0.9	0.85	1
23401	2.07	1	1.46	1.51	0.91	0.91	0.79	1.14
23410	2.05	1	1.46	1.53	0.9	1.13	1	1.15
24013	2.53	1	1.78	0.87	0.88	0.87	0.73	1.39
24031	2.48	1	1.8	0.89	0.88	1.31	1.15	1.38
24103	2.45	1	1.8	0.82	1.1	1.16	1.01	1.43
24130	2.5	1	1.81	0.88	1.14	1.37	1.23	1.28
24301	2.56	1	1.79	0.89	1.16	1.16	1.03	1.31
24310	2.5	1	1.78	0.82	1.16	1.37	1.23	1.42

Figure A-21: lbnl-network.tns results normalized by splatt (4)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
30124	2.81	1	1.87	1.61	1.5	1.48	0.36	1.44
30142	2.95	1	1.96	1.71	1.59	1.38	1.14	1.58
30214	2.28	1	1.55	1.32	1.57	1.53	0.62	1.14
30241	2.43	1	1.68	1.47	1.78	1.47	1.34	1.34
30412	2.62	1	1.78	1.54	1.21	1.19	1.06	1.45
30421	2.6	1	1.77	1.55	1.2	1.59	1.44	1.37
31024	1.79	1	1.36	1.14	1.09	1.09	0.37	0.9
31042	2.01	1	1.55	1.26	1.23	1.07	0.96	1.04
31204	1.77	1	1.35	1.16	1.21	1.21	0.49	0.89
31240	1.9	1	1.52	1.29	1.35	0.97	0.86	1.02
31402	2.1	1	1.64	1.4	0.9	0.9	0.78	1.11
31420	2.12	1	1.68	1.39	0.89	1.07	0.96	1.11
32014	1.67	1	1.32	1.12	1.06	1.03	0.34	0.86
32041	1.84	1	1.44	1.23	1.18	0.98	0.88	1.01
32104	1.71	1	1.34	1.22	1.16	1.19	0.5	0.9
32140	1.94	1	1.53	1.37	1.37	0.89	0.82	1.01
32401	1.95	1	1.51	1.39	0.83	0.83	0.72	1.05
32410	1.96	1	1.51	1.36	0.83	1.03	0.93	1.07
34012	2.57	1	1.97	0.88	0.84	0.88	0.75	1.29
34021	2.51	1	2.01	0.89	0.89	1.33	1.19	1.36
34102	2.52	1	2.02	0.86	1.06	1.17	1.01	1.42
34120	2.56	1	1.97	0.89	1.15	1.36	1.21	1.39
34201	2.59	1	1.96	0.89	1.1	1.09	0.96	1.37
34210	2.5	1	1.97	0.88	1.1	1.28	1.15	1.35

Figure A-22: lbnl-network.tns results normalized by splatt (5)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
40123	5.23	1	1.53	1.43	1.42	1.42	1.08	2.83
40132	5.14	1	1.43	1.34	1.41	2.69	2.27	2.79
40213	5.24	1	1.52	1.42	2.64	2.62	2.23	2.77
40231	5.01	1	1.48	1.37	2.57	3.68	3.33	2.44
40312	5.04	1	1.46	1.36	2.59	2.59	2.19	2.65
40321	5.02	1	1.47	1.37	2.37	3.47	3.31	2.68
41023	5.09	1	1.47	1.89	1.88	1.9	1.57	2.76
41032	5.01	1	1.47	1.89	1.88	3.04	2.74	2.77
41203	4.92	1	1.49	1.89	2.28	2.29	1.94	2.75
41230	4.99	1	1.49	1.9	2.3	2.75	2.39	2.75
41302	5.02	1	1.48	1.88	2.32	2.32	2	2.74
41320	5.02	1	1.47	1.89	2.35	2.71	2.4	2.75
42013	5.14	1	1.52	1.84	1.82	1.83	1.49	2.76
42031	5.22	1	1.55	1.84	1.85	3.05	2.72	2.76
42103	5.21	1	1.52	1.67	2.07	2.24	1.9	2.53
42130	5.22	1	1.54	1.84	2.26	2.72	2.37	2.48
42301	5.18	1	1.53	1.83	2.29	2.18	1.94	2.76
42310	5.03	1	1.52	1.81	2.26	2.67	2.38	2.74
43012	5.15	1	1.51	1.92	1.92	1.79	1.49	2.83
43021	5.02	1	1.51	1.94	1.94	3.05	2.71	2.79
43102	5.1	1	1.51	1.94	2.35	2.34	2.04	2.78
43120	5.02	1	1.51	1.93	2.34	2.75	2.43	2.78
43201	5.17	1	1.4	1.94	2.33	2.31	2	2.82
43210	5.06	1	1.46	1.96	2.33	2.64	2.44	2.83

Figure A-23: vast-2015-mc1-5d.tns results normalized by splatt (1)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
01234	2.67	1	0.84	0.17	0.17	0.17	0	3.34
01243	2.67	1	0.86	0.18	0.18	2.34	2.06	3.23
01324	2.69	1	0.85	0.18	2.37	2.19	2.21	3.51
01342	2.69	1	0.86	0.18	2.3	5.12	4.49	4.01
01423	2.64	1	0.85	0.18	2.22	2.37	2.14	3.46
01432	2.72	1	0.86	0.18	2.46	4.75	4.61	3.79
02134	2.26	1	0.74	1.15	0.65	0.6	0.49	2.85
02143	2.25	1	0.74	1.19	0.65	2.6	2.53	2.85
02314	1.81	1	0.83	1.1	1.17	1.12	0.97	1.92
02341	1.74	1	0.85	1.08	1.22	1.98	1.91	1.64
02413	1.86	1	0.83	1.1	1.11	1.13	0.92	1.94
02431	1.82	1	0.83	1.06	1.21	1.99	1.88	1.54
03124	1.98	1	0.92	1.13	0.95	1.02	0.86	2.03
03142	2.05	1	0.92	1.09	0.99	2.45	2.25	2.46
03214	2.03	1	0.91	1.12	1.56	1.6	1.32	1.99
03241	1.94	1	0.9	1.07	1.69	2.26	2.02	1.85
03412	1.93	1	0.9	1.14	2.02	2	1.76	2.07
03421	1.94	1	0.9	1.15	1.99	2.61	2.34	1.69
04123	2.08	1	0.93	1.13	1.01	1.03	0.85	2.15
04132	2.08	1	0.93	1.13	0.96	2.24	2.37	2.49
04213	1.98	1	0.91	1.13	1.63	1.63	1.43	2.16
04231	1.93	1	0.9	1.07	1.7	2.19	2.04	1.86
04312	1.95	1	0.9	1.16	2.11	2.01	1.91	1.9
04321	1.98	1	0.91	1.16	2.06	2.52	2.35	1.91

Figure A-24: vast-2015-mc1-5d.tns results normalized by splatt (2)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
10234	1.99	1	0.85	0.42	0.45	0.44	0.38	1.32
10243	2.06	1	0.84	0.44	0.45	1.56	1.45	1.28
10324	2.11	1	0.84	0.46	1.51	1.61	1.45	1.3
10342	1.96	1	0.85	0.43	1.45	2.5	2.58	1.56
10423	1.92	1	0.84	0.44	1.44	1.57	1.43	1.25
10432	2	1	0.83	0.45	1.49	2.52	2.62	1.44
12034	1.67	1	0.7	0.89	0.52	0.54	0.5	1
12043	1.64	1	0.68	0.87	0.5	1.4	1.29	1.06
12304	1.34	1	0.8	0.92	0.57	0.5	0.45	0.95
12340	1.34	1	0.8	0.89	0.7	0.59	0.5	0.63
12403	1.32	1	0.76	0.88	0.58	0.5	0.42	0.89
12430	1.32	1	0.79	0.87	0.71	0.61	0.49	0.64
13024	1.4	1	0.86	0.61	0.47	0.48	0.42	1.03
13042	1.48	1	0.85	0.61	0.47	1.23	1.1	1.14
13204	1.37	1	0.84	0.58	0.81	0.68	0.62	1.03
13240	1.4	1	0.88	0.71	0.87	0.61	0.52	0.74
13402	1.34	1	0.85	0.68	0.58	0.54	0.48	0.68
13420	1.34	1	0.86	0.7	0.55	0.72	0.7	0.73
14023	1.39	1	0.82	0.58	0.49	0.49	0.43	0.96
14032	1.47	1	0.84	0.61	0.46	1.17	1.04	1.22
14203	1.34	1	0.81	0.6	0.82	0.64	0.62	0.92
14230	1.38	1	0.85	0.7	0.88	0.63	0.51	0.75
14302	1.36	1	0.82	0.73	0.58	0.54	0.45	0.75
14320	1.39	1	0.82	0.72	0.55	0.72	0.63	0.74

Figure A-25: vast-2015-mc1-5d.tns results normalized by splatt (3)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
20134	2.18	1	2.43	0.97	0.46	0.46	0.32	2.48
20143	2.21	1	2.39	0.95	0.46	2.43	2.3	2.69
20314	1.57	1	1.71	0.84	1.14	1.08	1.01	1.55
20341	1.54	1	1.66	0.85	1.17	1.86	1.69	1.33
20413	1.61	1	1.68	0.84	1.21	1.09	1.01	1.66
20431	1.53	1	1.65	0.83	1.15	1.81	1.7	1.43
21034	1.24	1	1.3	0.58	0.33	0.34	0.27	0.82
21043	1.31	1	1.3	0.57	0.33	1.01	0.87	0.74
21304	1.21	1	1.27	0.78	0.55	0.46	0.38	0.9
21340	1.26	1	1.26	0.79	0.67	0.55	0.47	0.6
21403	1.21	1	1.27	0.76	0.57	0.43	0.42	0.85
21430	1.23	1	1.25	0.73	0.66	0.55	0.42	0.58
23014	1.3	1	1.32	0.6	0.25	0.21	0.17	0.84
23041	1.26	1	1.34	0.63	0.28	0.55	0.47	0.91
23104	1.32	1	1.42	0.99	0.56	0.47	0.45	0.8
23140	1.33	1	1.32	0.96	0.65	0.57	0.52	0.63
23401	1.18	1	1.29	0.9	0.53	0.27	0.2	0.7
23410	1.34	1	1.37	1.01	0.82	0.55	0.47	0.59
24013	1.3	1	1.34	0.59	0.25	0.2	0.16	0.87
24031	1.23	1	1.33	0.62	0.28	0.5	0.45	0.9
24103	1.32	1	1.41	1.01	0.54	0.47	0.45	0.82
24130	1.35	1	1.39	0.97	0.66	0.59	0.54	0.65
24301	1.22	1	1.28	0.9	0.53	0.27	0.19	0.71
24310	1.31	1	1.35	1.01	0.83	0.55	0.43	0.59

Figure A-26: vast-2015-mc1-5d.tns results normalized by splatt (4)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
30124	4.33	1	1.86	0.7	0.54	0.56	0.39	2.76
30142	4.09	1	1.81	0.68	0.53	1.61	1.42	3.38
30214	4.13	1	1.82	0.67	0.98	0.83	0.7	2.76
30241	3.38	1	1.65	0.64	0.9	1.59	1.3	2.32
30412	3.33	1	1.57	0.62	1.23	1.27	1.06	2.69
30421	3.36	1	1.6	0.62	1.22	1.55	1.26	2.35
31024	1.96	1	1.4	0.73	0.64	0.61	0.53	1.19
31042	1.9	1	1.4	0.78	0.65	1.05	1.09	1.37
31204	1.87	1	1.32	0.72	0.89	0.78	0.73	1.14
31240	1.78	1	1.36	0.83	1.05	0.86	0.73	0.95
31402	1.84	1	1.32	0.87	0.72	0.68	0.62	0.96
31420	1.79	1	1.36	0.84	0.8	0.99	0.84	0.96
32014	2.86	1	1.27	1.65	0.8	0.71	0.67	2
32041	2.52	1	1.24	1.54	0.82	1.26	1.2	1.69
32104	1.63	1	1.17	1.39	0.71	0.55	0.5	0.96
32140	1.65	1	1.17	1.33	0.81	0.73	0.62	0.77
32401	1.57	1	1.15	1.32	0.67	0.33	0.26	0.85
32410	1.61	1	1.2	1.37	1.03	0.63	0.57	0.69
34012	1.67	1	1.25	0.68	0.31	0.29	0.22	1.11
34021	1.73	1	1.26	0.7	0.32	0.43	0.36	1.01
34102	1.74	1	1.33	1.06	0.67	0.59	0.57	0.76
34120	1.67	1	1.26	1.06	0.61	0.82	0.69	0.76
34201	1.61	1	1.19	0.64	0.86	0.47	0.43	0.9
34210	1.6	1	1.29	1.02	1.2	0.68	0.6	0.66

Figure A-27: vast-2015-mc1-5d.tns results normalized by splatt (5)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
40123	4.22	1	1.84	0.67	0.54	0.55	0.38	2.99
40132	4.3	1	1.9	0.7	0.55	1.68	1.51	3.18
40213	4.2	1	1.87	0.67	1	0.88	0.67	2.9
40231	3.32	1	1.62	0.61	0.87	1.53	1.23	2.26
40312	3.36	1	1.57	0.62	1.21	1.17	1.13	2.42
40321	3.41	1	1.6	0.63	1.31	1.55	1.36	2.38
41023	1.95	1	1.35	0.74	0.63	0.58	0.58	1.16
41032	1.91	1	1.41	0.72	0.62	1.08	1.01	1.37
41203	1.89	1	1.34	0.77	0.92	0.76	0.74	1.14
41230	1.8	1	1.37	0.87	1.04	0.82	0.7	0.99
41302	1.87	1	1.36	0.89	0.79	0.78	0.69	0.99
41320	1.81	1	1.31	0.83	0.75	0.94	0.82	0.91
42013	2.93	1	1.28	1.69	0.89	0.8	0.61	2
42031	2.51	1	1.24	1.53	0.81	1.29	1.19	1.83
42103	1.67	1	1.21	1.36	0.71	0.55	0.54	0.98
42130	1.65	1	1.15	1.35	0.78	0.68	0.62	0.83
42301	1.58	1	1.15	1.32	0.67	0.32	0.25	0.87
42310	1.62	1	1.2	1.35	1.02	0.66	0.52	0.68
43012	1.72	1	1.2	0.69	0.3	0.28	0.21	1.06
43021	1.72	1	1.24	0.67	0.31	0.43	0.35	1.03
43102	1.73	1	1.29	1.05	0.66	0.57	0.57	0.76
43120	1.68	1	1.29	1.06	0.67	0.8	0.69	0.74
43201	1.63	1	1.19	0.63	0.88	0.51	0.42	0.9
43210	1.65	1	1.28	1.03	1.18	0.65	0.55	0.74

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[3] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA 09, pages 233–244, New York, NY, USA, 2009. Association for Computing Machinery.

[4] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package i: The symmetric codes. *International Journal for Numerical Methods in Engineering*, 18(8):1145–1151, 1982.

[5] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[6] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978.

[7] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[8] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 61–70, May 2015.

[9] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[10] Shaden Smith and George Karypis. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit, 2016.