Recursive Storage Format of Sparse Tensors and Cache Efficient Tensor Multiplication

Akshat Singhal(111496103)

May 21, 2018

Abstract

Algorithms for sparse tensors are different than those for dense as they make use of the sparsity of these tensors. There are several tools which implement algorithms for sparse tensors by taking into account the density or sparsity of a tensor. However, all of these tools use iterative algorithms which are known to be slower than the recursive ones in case of dense tensors. This project implements the recursive algorithm for multiplication of two-dimensional tensors.

1 Problem Description

Directly applying the mathematical definition of matrix multiplication gives an algorithm that takes time on the order of n^3 to multiply two nxn matrices.
However, the algorithm does not take into consideration the density of the matrices. Sparse matrices usually have a very few non-zero elements which makes the standard dense-matrix algorithms inefficient.

Tensors, in practice, are generally represented using multidimensional matrices. Operations on sparse tensors, like sparse matrices, if employed using the same algorithms as for dense tensors, yield inefficient results.

There are several tools which use the algorithms specific to sparse tensors. However, all of these algorithms are iterative.
Recursive algorithms for tensors are known to perform better as they make use of temporal locality.

Compressed tree storage(CTS) is a recursive storage format for higher dimensional sparse tensors.

The goal of this project is to implement a cache efficient recursive multiplication algorithm for multidimensional sparse tensors where the tensors are in compressed tree storage format.

2 Deliverables

The project's deliverables are broken in phases:

1. Converting a given two-dimensional sparse tensor into CTS format.
2. Recursive multiplication of two given two-dimensional sparse tensors in CTS format and comparison of the result with TACO [1].
3. Extend the implementation for multidimensional tensors.

3 Solution Methodology

At present, the project contains implementation for the following operations:

1. Converting a given two-dimensional sparse tensor into CTS format.
2. Recursive multiplication of two given two-dimensional sparse tensors in CTS format and compare its result with TACO [1].

3.1 Compressed Tree Storage Format

Sparse tensors are generally stored in the compressed sparse row(CSR) format. CTS format stores a tensor in such a way, it becomes easier to implement recursive algorithms. As the name suggests, CTS format stores a tensor in the form of a tree. In this project, a CTS format tensor contains two different types of nodes:

1. Internal nodes - As in a typical tree, each internal node contains pointers to its children. An internal node does not contain any data.
2. Leaf nodes - A leaf node contains the actual data but in the form of an square matrix which is stored in CSR format as the matrix is sparse.

3.2 Recursive Multiplication

1. The tensors to be multiplied are in the form of trees. Since, the data is only at leaf nodes, actual multiplication takes place only at leaf nodes.
2. The algorithm uses temporary space to multiply the trees.
3. Comparison of the algorithm with TACO [1], which is currently the fastest iterative sparse tensor multiplication tool, shows that although, the overall computation time is quite high, the actual multiplication and merge operations on leaf nodes are comparable to TACO [1].

4 Recursive Sparse Matrix Multiplication

The recursive matrix multiplication algorithm for sparse matrices used in this project is very similar to the recursive matrix multiplication for dense matrices, except that instead of matrices we have quad-trees. Below, is a rough pseudo code of sparse matrix multiplication.

---

procedure SPARSE-REC-MM(Z, X, Y) {Z, X, and Y are trees in CTS format, each having 4 children}

if $n \leq B$ **then** {here, B signifies base case}

$Z_B \leftarrow X_B.Y_B$ **return**

end if{Both A and B are temporary trees}

Sparse-Rec-MM(A_1, X_1, Y_1)

Sparse-Rec-MM(A_2, X_1, Y_2)

Sparse-Rec-MM(A_3, X_3, Y_1)

Sparse-Rec-MM(A_4, X_4, Y_2)

Sparse-Rec-MM(B_1, X_2, Y_3)

Sparse-Rec-MM(B_2, X_2, Y_4)

Sparse-Rec-MM(B_3, X_4, Y_3)

Sparse-Rec-MM(B_4, X_4, Y_4)

$Z \leftarrow$ *Merge*(A, B)

end procedure

4.1 Multiplication

Multiplication in itself may be viewed as a two-step process. The first one is recursion and the second is actual multiplication at leaf nodes.
Step 1 results in creation of internal nodes of the resultant tree. As can be from figure 1 that only nodes get created in the resultant tree which correspond to existing nodes in X and Y. The example in figure 1 would result in execution of recursive call 1 and 3 from the algorithm.

Figure 2 shows the implementation of the second step.
This step of the algorithm is not recursive. Instead, the matrices are multiplied using the iterative algorithm, which multiplies two sparse matrices without converting them into the dense matrix format.

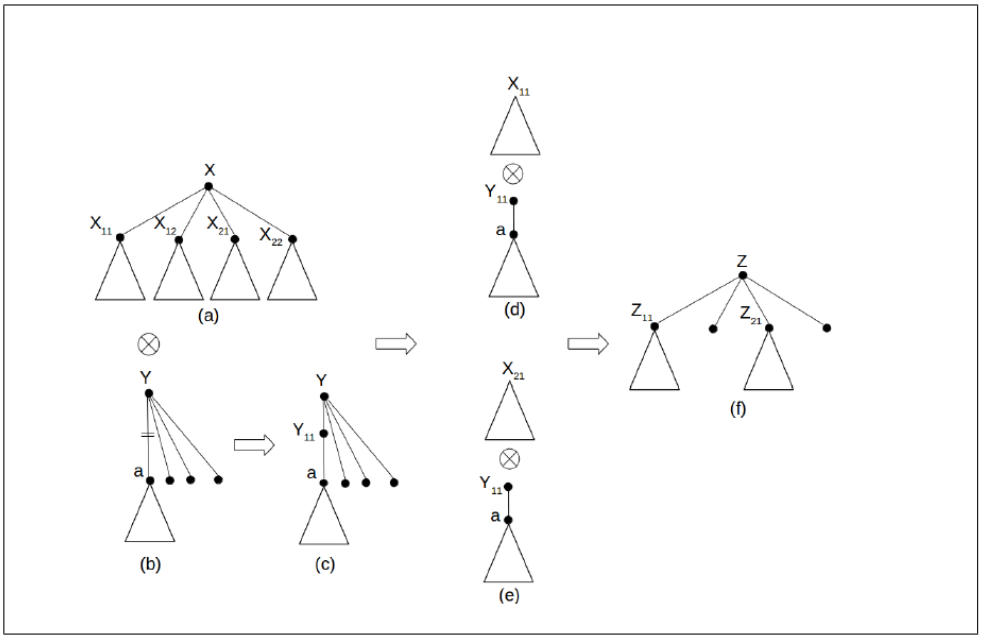

Figure 1: Multiplication of two trees, X(dense) and Y(sparse) resulting in tree Z containing two children, Z_1 and Z_3.

```
bool multiplyMatrices(const Csr& srcCsr1, const Csr& srcCsr2, Csr& targetCsr) {
    //assemble
    bool* workspace = new bool[B]();
    int* wlist = new int[B]();
    int w_size = 0;
    targetCsr.iCount[0] = 0;
    for (int i = 0; i < B; i++) {
        for (int B2_pos = srcCsr1.iCount[i]; B2_pos < srcCsr1.iCount[i+1]; B2_pos++) {
            int k = srcCsr1.idx[B2_pos];
            for (int C2_pos = srcCsr2.iCount[k]; C2_pos < srcCsr2.iCount[k+1]; C2_pos++) {
                int j = srcCsr2.idx[C2_pos];
                if (!workspace[j]) {
                    wlist[w_size++] = j;
                    workspace[j] = true;
                }
            }
        }
        targetCsr.iCount[i+1] = targetCsr.iCount[i] + w_size;
        for (int w_pos = 0; w_pos < w_size; w_pos++) {
            int j = wlist[w_pos];
            targetCsr.idx[targetCsr.iCount[i] + w_pos] = j;
            workspace[j] = false;
        }
        w_size = 0;
    }
    // compute
    double* tempVals = new double[B]();
    for (int i = 0; i < B; i++) {
        for (int B2_pos = srcCsr1.iCount[i]; B2_pos < srcCsr1.iCount[i+1]; B2_pos++) {
            int k = srcCsr1.idx[B2_pos];
            for (int C2_pos = srcCsr2.iCount[k]; C2_pos < srcCsr2.iCount[k+1]; C2_pos++) {
                int j = srcCsr2.idx[C2_pos];
                tempVals[j] += srcCsr1.vals[B2_pos] * srcCsr2.vals[C2_pos];
            }
        }
        for (int A2_pos = targetCsr.iCount[i]; A2_pos < targetCsr.iCount[i+1]; A2_pos++) {
            int j = targetCsr.idx[A2_pos];
            targetCsr.vals[A2_pos] = tempVals[j];
            tempVals[j] = 0;
        }
    }
    targetCsr.vals.resize(targetCsr.iCount[B]);
    targetCsr.idx.resize(targetCsr.iCount[B]);
    delete[] workspace;
    delete[] wlist;
    delete[] tempVals;
    return (targetCsr.iCount[targetCsr.iCount.size() - 1] > 0 ? true : false);
}
```

Figure 2: Multiplication of two matrices in CSR format.

4.2 Merging

Just like multiplication, merging is also a two-step process. The difference in case of merging is that even if one of the trees contains a node, the resultant tree has to contain that node. Figure 3 is a good example of merge where the resultant tree not only contains individual nodes of the input trees but also a merged node of the input trees.
Figure 4 shows the implementation of merging two matrices represented in CSR format.

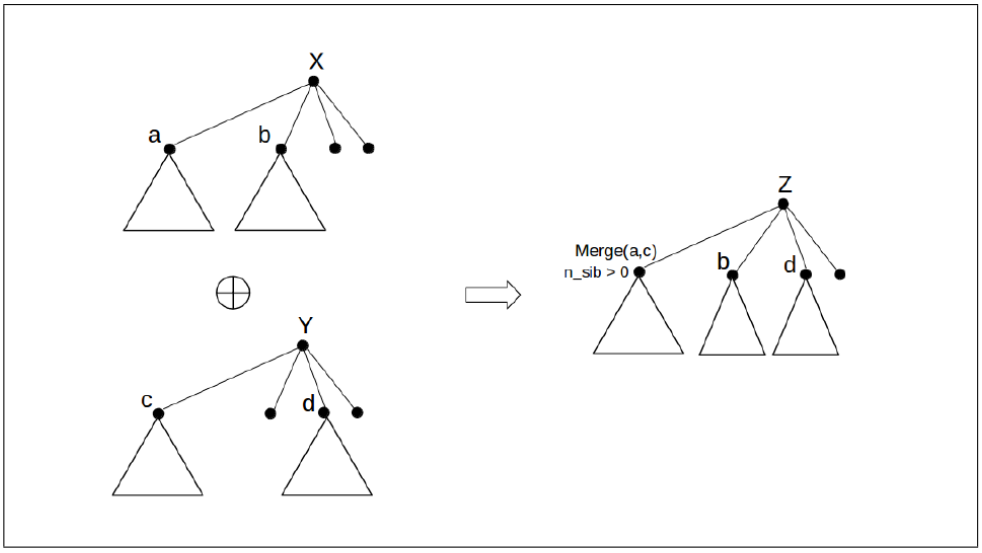

Figure 3: Merging X and Y results in creation of 3 nodes of Z, Z_1, Z_2 and Z_3. Where Z_2 and Z_3 are b and d respectively, Z_1 gets created by a recursively merging a and c.

```
void mergeMatrices(const Csr& srcCsr1, const Csr& srcCsr2, Csr& targetCsr) {
    for(int i = 0; i < B; i++) {
        int iCount1 = srcCsr1.iCount[i+1];
        int iCount2 = srcCsr2.iCount[i+1];
        int j = srcCsr1.iCount[i];
        int k = srcCsr2.iCount[i];
        int t = targetCsr.iCount[i];
        while(j < iCount1 && k < iCount2) {
            if(srcCsr1.idx[j] < srcCsr2.idx[k]) {
                targetCsr.idx[t] = srcCsr1.idx[j];
                targetCsr.vals[t] = srcCsr1.vals[j];
                j++;
            } else if(srcCsr1.idx[j] > srcCsr2.idx[k]) {
                targetCsr.idx[t] = srcCsr2.idx[k];
                targetCsr.vals[t] = srcCsr2.vals[k];
                k++;
            } else {
                targetCsr.idx[t] = srcCsr1.idx[j];
                targetCsr.vals[t] = srcCsr1.vals[j] + srcCsr2.vals[k];
                j++;
                k++;
            }
            t++;
        }
        while(j < iCount1) {
            targetCsr.idx[t] = srcCsr1.idx[j];
            targetCsr.vals[t] = srcCsr1.vals[j];
            j++;
            t++;
        }
        while(k < iCount2) {
            targetCsr.idx[t] = srcCsr2.idx[k];
            targetCsr.vals[t] = srcCsr2.vals[k];
            k++;
            t++;
        }
        targetCsr.iCount[i+1] = t;
    }
    targetCsr.vals.resize(targetCsr.iCount[B]);
    targetCsr.idx.resize(targetCsr.iCount[B]);
}
```

Figure 4: Merging of two matrices in CSR format.

5 Results

Since, a part of the goal was to compare the result with TACO [1], the same five sparse matrices have been used here to test multiplication.
NNZ stands for number of non-zero entries in a matrix.

Matrix	NNZ	Density
bcsstk17	219,812	1.8 x 10^{-3}
pdb1HYS	2,190,591	1.6 x 10^{-3}
rma10	2,374,001	1.0 x 10^{-3}
cant	2,034,917	5.2 x 10^{-4}
consph	3,046,907	4.4 x 10^{-4}

Table 1: Test matrices from the SuiteSparse [2] Matrix Collection

Each of the matrices has been multiplied with two randomly generated matrices of density 1e-4 and 2.5e-3. Below are the tabulated results for each of matrix.

Since, TACO mentions time with sorted data, time has been computed for TACO in both formats. TACO(U) means TACO in unsorted format and TACO(S) means TACO in sorted format.

There are two time calculations done for each run. The column corresponding to 'Base' shows the time taken by multiplication and merging at the base levels. The column corresponding to 'Total' shows the total time taken by the project. It includes time to create and destroy temporary trees and vectors.

Table 2: Time in ms for multiplication with matrix of density 1e-4

Matrix	TACO(U)	TACO(S)	Base	Total
bcsstk17	4	11	51	624
pdb1HYS	48	333	536	19941
rma10	69	456	682	26519
cant	83	572	683	19108
consph	173	1174	759	364030

Table 3: Time in ms for multiplication with matrix of density 2.5e-3

Matrix	TACO(U)	TACO(S)	Base	Total
bcsstk17	40	292	161	947
pdb1HYS	1366	9787	1695	28880
rma10	2052	12918	2556	40162
cant	2420	13968	3378	42831
consph	4863	28809	3687	173676

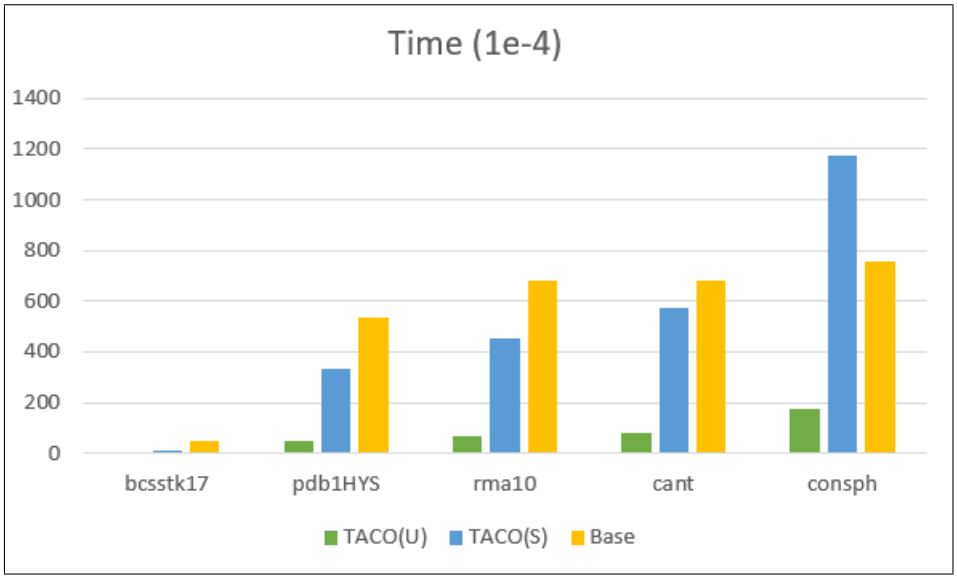

Figure 5: Running time comparison using matrix of density 1e-4.

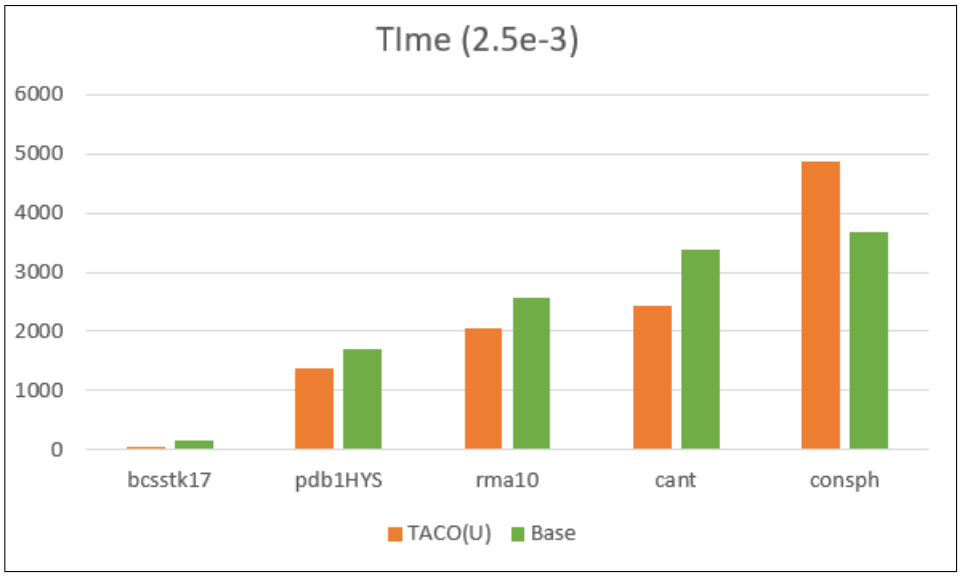

Figure 6: Running time comparison using matrix of density 2.5e-3.

Both the tables clearly show that unsorted TACO is faster but not by a very big margin. In fact, since the time in 'Base' is (merge + multiply), and since, this time is comparable to that of TACO, it means that operations at base level at optimal.

The total running time is quite high as compared to the base level running time. This means that overhead time for creation and deletion of temporary trees and vectors can be improved.

6 Future Scope

1. As we have seen in the result section that the total running time is quite high, this implies that there is scope to improve upon the creation and deletion of temporary trees and vectors.

2. The project, currently works for only two dimensional tensors. However, the comparable running time of the base case shows that recursion is efficient and should be explored for n-dimensional tensors.

7 References

References

[1] http://tensor-compiler.org/

[2] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. 38, 1, Article 1 (Dec. 2011).

[3] https://en.wikipedia.org/wiki/Sparse_matrix