Compressed Sparse Row (CSR)

Kiarash Torkian

***Abstract*—Paper focuses on highlighting the work and research that was put into comparing operations done on a regular spare matrix vs. a compressed one. A library of operations were written using the Java language to compare the two and is publicly available. A compressed matrix will benefit from not having to explicitly perform operations on zeros which won't have any effect on the final value while requiring a substantially less amount of space to store.**

I. Introduction

When it comes to matrices, having to write methods to perform operations on them are not that difficult, since they really just boil down to only two or more nested loops. Now in terms of memory and performance, this won't be a huge problem as long as the matrices themselves are not very large. But lets not forget that two nested loops is still an O^2 algorithm and will quadratically take more memory and be slower in performance as the matrices grow larger and larger. Which happens to be very frequent in real life examples. One thing that we can say for sure is having zeros in a matrix can make calculating it easier. But not for a computer, if we don't make an new algorithm that works around this since it will still go over the zero values anyway. So what if we found a way to compress our matrices in a format that contained only the nonzero values without changing the matrix itself? Turns out we can do this in multiple ways, but in all cases it won't work too well unless we have a spare matrix.

II. Background

A. Spare Matrix

Spare matrix is a type of matrix that is mostly filled with zero values and is therefore opposite of a dense matrix, which contains mostly nonzero values. A spare matrix can be vastly improved in terms of space needed and performance required since majority of its data can be removed and its operations skipped. This may not be so true for a dense matrix however for having to take the extra time of converting a matrix into a compressed format when we would only save on merely a couple of steps. Here we see a simple example of a spare matrix.

$$A = \begin{bmatrix} 1 & 0 & 6 & 5 & 3 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 3 & 0 & 3 \\ 0 & 0 & 9 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 7 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{bmatrix} \tag{1}$$

B. Compressed Sparse Row

A compressed sparse row is one of the many ways of compressing a matrix by holding onto any value that is nonzero (val), the column index of values (col_ind), and a row pointer that represents the start of a new row (row_ptr). There includes many other famous methods such as compressed column storage (CCS) that instead holds onto the row index (row_ind) and a column pointer (col_ptr), and coordinate format (COO) that instead of having a pointer vector, it will simply hold onto the column index (col_ind) and the row index (col_ind).

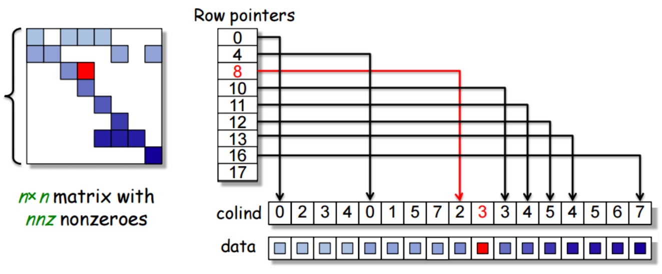

Figure 1: *A compressed sparse row can represent a matrix by holding onto only the nonzero values and two more arrays to understand their previous positions. This figure is a representation of matrix **(1)***

Consider nnz is the number of nonzero elements in a matrix and n is the length of the row_ptr vector. The memory required to store these elements is equivalent to

$$2nnz + n + 1$$

Where as to store a matrix in all its formality, one would require $m * n$ of memory where n and m represent number of rows and columns respectively. Even though memory is something we seem to worry about less and less as time goes on since it is not as limited as it used to be, it is however, not something we can ignore when working with large data such as matrices. We can still reach our limits very soon. The benefit however, that comes from compressing matrices is not just memory.

III. Algorithms

Now that we have gotten rid of zero values, our operations will no longer need to be considered about looping over zero values that will have no affect in the end result of the operation. In this section we will be going over some of the main implemented functions done for this program.

A. Matrix-Vector Multiplication

Lets compare the below two algorithms, where one operates on a regular matrix while the other, on a compressed sparse row. Consider the following:

$$m = Number\ of\ rows$$

$$n = Number\ of\ columns$$

$$nnz = Number\ of\ nonzero$$

$$av = Average\ number\ nonzeros\ per\ row$$

Algorithm 1 Matrix-Vector Multiplication $O(n * m)$

```
Require: $A \leftarrow$ Regular Matrix $B \leftarrow$ Vector $C \leftarrow$ Final Vector
1: for i = 0 to n do
2:     for j = 0 to m do
3:         $c[i] \leftarrow$ c[i] + A[i][j] * b[j]
4:     end for
5: end for
```

Algorithm 2 CSR Matrix-Vector Multiplication ($O(m * av)$)

```
Require: $A \leftarrow$ CSR Matrix $B \leftarrow$ Vector $C \leftarrow$ Final Vector
1: for i = 0 to m do
2:     for j = A.row_ptr[i] to j <A.row_ptr[i+1] do
3:         $c[i] \leftarrow$ c[i] + A.val[j] * b[A.col_ind[j]]
4:     end for
5: end for
```

Looking at **Algorithm 1**, we can see that it contains two nested loops. First one loops till **N**, while the second one loops till **M**. Since the nested loop always starts from zero, we can say that this will result in an $O(n * m)$ overhead which will have a quadratic growth as the matrices grows larger. **Algorithm 2** also has two nested loops but it is actually not a quadratic growth. The first loop only iterates as number of rows in the matrix while the second loop has a condition using the row_ptr vector to only find out how many nonzero values are in that row. So it doesn't iterate from the start. **Algorithm 2** has only a linear growth of $O(m * av)$ or in other words, $O(nnz)$. It might be hard to see at first, but looking at **Figure 2**, we can see that we would end up performing one operation per each nonzero elements in a row by matching it with the same index of the vector's column. In conclusion, we started by an algorithm that was $O(n * m)$, which is considered to be moderately slow and turned it into a $O(nnz)$ operation by modifying our matrices. As long as, this is done on spare matrices, the performance increase will be huge, whereas for dense matrices, not much will change. A matrix for example that contains no zero values whatsoever, will force the algorithm to still do $O(nnz)$ which in this case is also $O(n * m)$.

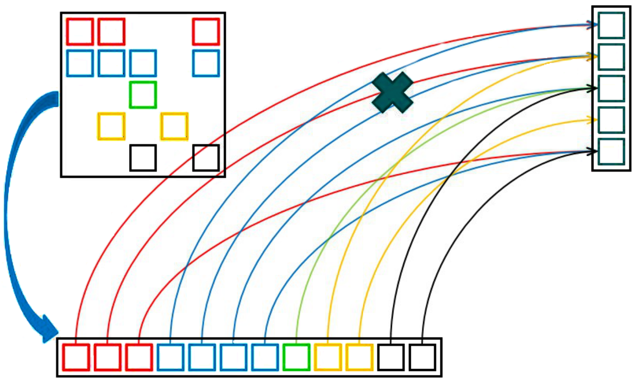
Figure 2: *Graphic representation of steps for a compressed matrix, vector multiplication. All the nonzero in the val vector are only accessed once which makes this algorithm with a $O(nnz)$ overhead.*

B. Matrix Transposition

We see a very familiar pattern in regards to **Algorithm 3** and **Algorithm 4** compared to our previous two algorithms. For one, the regular matrix operation once again is a nested loop that grows and shrinks based on the column and row size of the matrix and the nested loop always starts from zero. Therefore, the algorithm has a performance of $O(n * m)$. The second one, while having considerably more loops, the nested ones behave similarly to ones from **Algorithm 2** and are therefore not a true nested set of loops. When we try to go through the process of finding the Big O, we find out that the two nested loops have identical conditions and one of them can be removed from the equation.

$$O((n * av) + m + (n * av))$$

$$O(2(n * av) + m)$$

$$O(\cancel{2}(n * av) + m)$$

$$O((n * av) + m)$$

Algorithm 3 Matrix Transposition $O(n * m)$

```
Require: $A \leftarrow$ CSR Matrix $T \leftarrow$ CSR Matrix
1: for i = 0 to n do
2:     for j = 0 to m do
3:         $temp \leftarrow$ A[i][j]
4:         $A[i][j] \leftarrow$ A[j][i];
5:         $A[j][i] \leftarrow$ temp
6:     end for
7: end for
```

Algorithm 4 CSR Matrix Transposition $O(n+m)$

```
Require: $A \leftarrow$ CSR Matrix $T \leftarrow$ CSR Matrix
 1: $k \leftarrow 0$
 2: $row\_count \leftarrow$ new int[]
 3: $col\_count \leftarrow$ new int[]
 4: for i = 0 to n do
 5:     for j = row_ptr[i] to j <row_ptr[i+1] do
 6:         $k \leftarrow$ col_ind[j]
 7:         $row\_count[k] \leftarrow$ row_count[k] + 1
 8:     end for
 9: end for
10: for i = 0 to m do
11:     $T.row\_ptr[i+1] \leftarrow$ T.row_ptr[i] + row_count[i]
12: end for
13: for i = 0 to n do
14:     for j = row_ptr[i] to j <row_ptr[i+1] do
15:         $k \leftarrow$ col_ind[j]
16:         $index \leftarrow$ T.row_ptr[k] + row_count[k]
17:         $T.col\_indx[index] \leftarrow$ i
18:         $T.val[index] \leftarrow$ T.val[j]
19:     end for
20: end for
```

C. Symmetrical

Lets also quickly talk about checking whether a matrix is symmetrical or not. A symmetrical matrix is one where its inverse is equivalent of itself.

$$A = A^T$$

$$A^{-1}A^T = I$$

The method for a regular matrix would be as usual, two nested loops where it would try to check if a value in a row, column is equal to a value in a column, row. If this is not the case then the matrix is not symmetrical and is therefore not equal to its transpose. For performing this under our compressed matrix, we can simply use one of our previously implemented functions. Here, we can use **Algorithm 4** to first find the transpose of a compressed matrix. The function should give us back a transposed version of matrix A. We will then need to make sure the values in both of these matrices are the same.

Algorithm 5 Symmetrical $O(n * m)$

```
Require: $A \leftarrow$ CSR Matrix
1: for i = 0 to n do
2:     for j = 0 to m do
3:         if A[i][j] != A[i][j] then
4:             return false
5:         end if
6:     end for
7: end for
```

Algorithm 6 Symmetrical $O(n+m+nnz)$

```
Require: $A \leftarrow$ CSR Matrix $T \leftarrow$ CSR Matrix
1: $T \leftarrow$ Algorithm 4 on A
2: if A != T then
3:     return false
4: end if
```

IV. Conclusion

Compressing spare matrices can substantially improve the performance and the space requirement of a program. It will allow you to get past quadratic growth algorithms and instead have linear ones. Compression techniques, as beneficial as they are, are not always the best options. Using them on dense matrices will make no improves in anyway and might not be worth the conversions on other cases. Therefore a small decision making is required before deciding if you want your matrices compressed.

References

[1] M. Mccourt, B. Smith, and H. Zhang, EFFICIENT SPARSE MATRIX-MATRIX PRODUCTS USING COLORINGS. [Online]. Available at: http://www.mcs.anl.gov/papers/p5007-0813_1.pdf

[2] S. Y. Cheung, Working with Sparse Matrices, Working with Sparse Matrices. [Online]. Available at: http://www.mathcs.emory.edu/c̃heung/courses/561/syllabus/3-c/sparse.html

[3] Compressed Sparse Row Format: CSR, Compressed Sparse Row Format: CSR. [Online]. Available at: http://www5.in.tum.de/lehre/vorlesungen/parnum/ws10/parnum_6.pdf

[4] J. Dongarra , Compressed Column Storage (CCS), Compressed Column Storage (CCS), 20-Nov-1995. [Online]. Available at: http://netlib.org/linalg/html_templates/node92.html

[5] J. Fineman, M. Frigo, J. Gilbert, and C. Leiserson, Parallel Sparse Matrix-Vector and MatrixTranspose-Vector Multiplication using Compressed Sparse Blocks, Parallel Sparse Matrix-Vector and MatrixTranspose-Vector Multiplication using Compressed Sparse Blocks. [Online]. Available at: http://gauss.cs.ucsb.edu/ aydin/talks/csb-spaa.pdf

[6] W. H. Press, Numerical recipes: the art of scientific computing. Cambridge, UK: Cambridge University Press, 2007. P. Stathis, D. Cheresiz, S. Vassiliadis, and B. Juurlink, Sparse Matrix Transpose Unit. [Online]. Available at: https://www.aes.tu-berlin.de/fileadmin/fg196/publication/old-juurlink/sparse_matrix_transpose_unit.pdf

[7] W. Townsend, “Irregular Applications Sparse Matrix Vector Multiplication”, http://slideplayer.com/slide/9793914/, 2016

[8] V. der Vorst, Iterative Krylov Methods for Large Linear Systems. Cambridge University Press