

Mini Project

CSO-221N ALGORITHMS

Sparse matrix multiplication

Contents

- What is a sparse matrix ?
- How to store/represent it ?
- Fast multiplication Algorithms.
 - Algorithm 1: Gustavson's Row-wise SpGEMM3.
 - Algorithm 2: RowsToThreads.
 - Algorithm 3: Hash SpGEMM.
- Critical Assessment

Submitted by:

- ❖ *Kumar Shivam Ranjan*
- ❖ *Kundan Kumar*
- ❖ *Kshitij Sharma*
- ❖ *Madhur Sahu*
- ❖ *Naman Garg*

What is a sparse matrix ?

A **sparse matrix** or sparse array is a matrix in which most of the elements are zero. Large sparse matrices often appear in scientific or engineering applications when solving partial differential equations.

- ❖ If most of the elements are nonzero, then the matrix is considered dense. The number of zero-valued elements divided by the total number of elements (e.g., $m \times n$ for an $m \times n$ matrix) is called the **sparsity** of the matrix (which is equal to 1 minus the density of the matrix). Using those definitions, a matrix will be sparse when its sparsity is greater than 0.5.

0	1	0	0	0	0	0	0
0	0	0	0	0	0	8	5
4	0	0	5	0	0	0	7
0	0	0	9	5	0	0	2
0	0	0	1	0	4	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
0	8	0	0	0	0	0	0

Density=14/64

sparsity=50/64

How to store/represent it ?

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value)**.

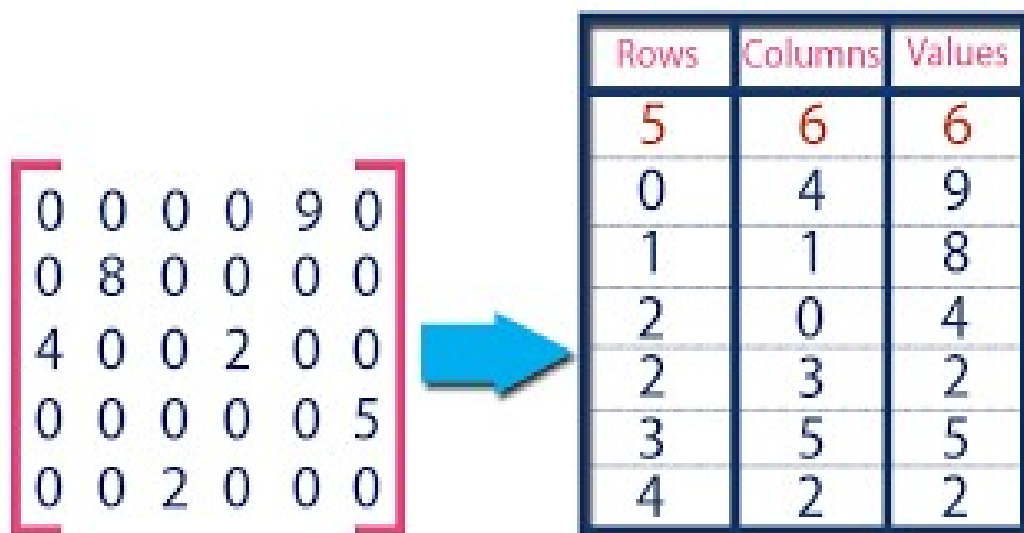
Sparse Matrix Representations can be done in many ways following are two common representations:

1. **Array representation**
2. **Linked list representation**

Method 1: Using Arrays

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non-zero element located at index – (row, column)

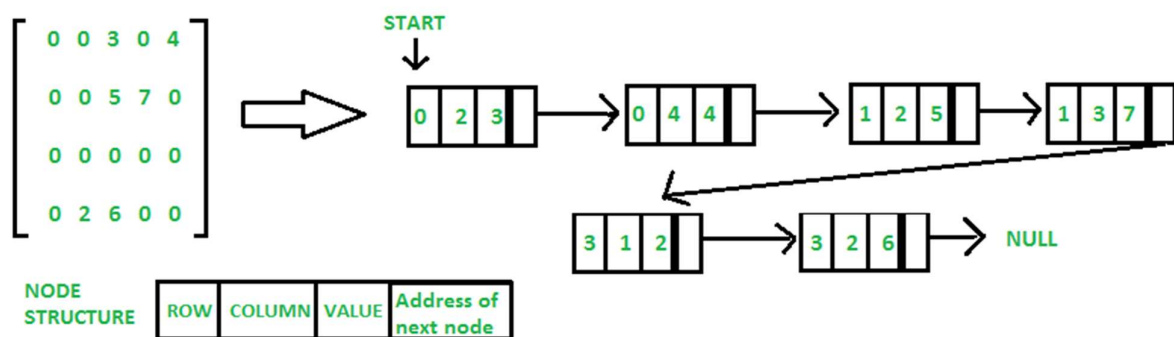


Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non-zero element located at index – (row, column)
- **Next node:** Address of the next node



Other representations:

- As a **Dictionary** where row and column numbers are used as keys and values are matrix entries. This method saves space but sequential access of items is costly.
- As a **list of lists**. The idea is to make a list of rows and every item of list contains values. We can keep list items sorted by column numbers.

Operations on Sparse Matrices

Given two sparse matrices, we can perform operations such as add, multiply or transpose of the matrices in their sparse form itself.

- To **Add** the matrices, we simply traverse through both matrices element by element and insert the smaller element (one with smaller row and col value) into the resultant matrix. If we come across an element with the same row and column value, we simply add their values and insert the added data into the resultant matrix.
- To **Transpose** a matrix, we can simply change every column value to the row value and vice-versa, however, in this case, the resultant matrix won't be sorted as we require. Hence, we initially determine the number of elements less than the current element's column being inserted in order to get the exact index of the resultant matrix where the current element should be placed. This is done by maintaining an array index $[]$ whose i th value indicates the number of elements in the matrix less than the column i .
- To **Multiply** the matrices, we first calculate transpose of the second matrix to simplify our comparisons and maintain the sorted order. So, the resultant matrix is obtained by traversing through the entire length of both matrices and summing the appropriate multiplied values. Any row value equal to x in the first matrix and row value equal to y in the second matrix (transposed one) will contribute towards $result[x][y]$. This is obtained by multiplying all such elements having col value in both matrices and adding only those with the row as x in first matrix and row as y in the second transposed matrix to get the $result[x][y]$.

Matrix 1: (4x4)

Row Column Value

1 2 10

1 4 12

3 3 5

4 1 15

4 2 12

Matrix 2: (4X4)

Row Column Value

1 3 8

2 4 23

3 3 9

4 1 20

4 2 25

Result of Addition: (4x4)

Row Column Value

1 2 10

1 3 8

1 4 12

2 4 23

3 3 14

4 1 35

4 2 37

Result of Multiplication: (4x4)

Row Column Value

1 1 240

1 2 300

1 4 230

3 3 45

4 3 120

4 4 276

Result of transpose on the first matrix: (4x4)

Row Column Value

1 4 15

2 1 10

2 4 12

3 3 5

4 1 12

Worst case time complexity: Addition operation traverses the matrices linearly, hence, has a time complexity of $O(n)$, where n is the number of non-zero elements in the larger matrix amongst the two. Transpose has a time complexity of $O(n+m)$, where n is the number of columns and m is the number of non-zero elements in the matrix. Multiplication, however, has a time complexity of $O(x*n + y*m)$, where (x, m) is number of columns and terms in the second matrix; and (y, n) is number of rows and terms in the first matrix.

Fast multiplication Algorithms

SpGEMM is a building block for many high-performance graph algorithms, including graph contraction, breadth first search from multiple source vertices, peer pressure clustering, recursive all-pairs shortest-paths, matching, and cycle detection. It is a subroutine in more traditional scientific computing applications such as multigrid interpolation and restriction and Schur complement methods in hybrid linear solvers. It also has applications in general computing, including parsing context-free languages and coloured intersection searching. The classical serial SpGEMM algorithm for general sparse matrices was first described by Gustavson and was subsequently used in MATLAB and CSpase.

Let A, B be input matrices, and SpGEMM computes a matrix C such that $C = AB$. We assume n -by- n matrices for simplicity. The input and output matrices are sparse and they are stored in a sparse format. The number of nonzeros in matrix A is denoted with $\text{nnz}(A)$. Below Figure shows the skeleton of the most commonly implemented SpGEMM algorithm, which is due to

Gustavson . When the matrices are stored using the Compressed Sparse Rows (CSR) format, this SpGEMM algorithm proceeds row-by-row on matrix A (and hence on the output matrix C). Let a_{ij} be the element in i -th row and j -th column of matrix A and a_{i*} be the i -th row of matrix A. The row of matrix B corresponding to each non-zero element of matrix A is read, and each non-zero element of output matrix C is calculated.

Algorithm 1 : Gustavson's Row-wise SpGEMM 3 .

Input: Sparse matrices A and B

Output: Sparse matrix C

set matrix C to \emptyset

for all a_{i*} in matrix A in parallel do

 for all a_{ik} in row a_{i*} do

 for all b_{kj} in row b_{k*} do

$value \leftarrow a_{ik} b_{kj}$

 if $c_{ij} \notin c_{i*}$ then

 insert ($c_{ij} \leftarrow value$) to c_{i*}

 else

$c_{ij} \leftarrow c_{ij} + value$

 end if

 end for

end for

end for

Since each row of C can be constructed independently of each other, Gustavson's algorithm is conceptually highly parallel. For accumulation, Gustavson's algorithm uses a dense vector and a list of indices that hold the nonzero entries in the current active row. This particular set of data structures used in accumulation are later formalized by Gilbert et al. under the name of sparse accumulator (SPA) . Consequently, a naive parallelization of Gustavson's algorithm requires temporary storage of $O(nt)$ where t is the number of threads. For matrices with large dimensions, a SPA-based algorithm can still achieve good performance by "blocking" SPA in order to decrease cache miss rates. Patwary et al. achieved this by partitioning the data structure of B by columns.

Sulatycke and Ghose presented the first shared-memory parallel algorithm for the SpGEMM problem, to the best of our knowledge. Their parallel algorithm, dubbed IKJ method due to the order of the loops, has a double-nested loop over the rows and the columns of the matrix A . Therefore, the IKJ method has work complexity $O(n^2 + \text{flop})$ where flop is the number of the non-trivial scalar multiplications (i.e. those multiplications where both operands are nonzero) required to compute the product. Consequently, the IKJ method is only competitive when $\text{flop} \geq n^2$, which is rare for SpGEMM.

Several GPU algorithms that are also based on the row-by-row formulation are presented. These algorithms first bin the rows based on their density due to the peculiarities of the GPU architectures. Then, a poly-algorithm executes a different specialized kernel for each bin, depending on its density. Two recent algorithms that are implemented in both GPUs and CPUs also follow the same row-by-row pattern, only differing on how they perform the merging operation. ViennaCL implementation, which was first described for GPUs, iteratively merges sorted lists, similar to merge sort. KokkosKernels implementation, which we also include in our evaluation, uses a multi-level hash map data structure. The CSR format is composed of three arrays: row pointers array (rpts) of length $n + 1$, column indices (cols) of length nnz , and values (vals) of length nnz . Array rpts indexes the beginning and end locations of nonzeros within each row such that the range $\text{cols}[\text{rpts}[i] \dots \text{rpts}[i + 1] - 1]$ lists the column indices of row i .

Algorithm 2 : RowsToThreads.

Input: Sparse matrices A and B

Output: Array offset

1. Set FLOP vector

```
for i ← 0 to m in parallel do
    flop [ i ] ← 0
    for j ← rptsA [ i ] to rptsA [ i+1] do
        rnz ← rptsB [ cols A [ j ] + 1 ] – rptB [ colsA [ j ] ]
        flop [ i ] ← flop [ i ] + rnz
    end for
end for
```

2 . Assign rows to thread

```
Flopps ← ParallelPrefixSum ( flop )
Sumflop ← flopps [ m ]
tnum ← omp_get_max_threads ( )
aveflop ← sumflop / tnum
offset[0] ← 0
for tid ← 1 to tnum in parallel do
    offset [ tid ] ← lowbnd ( flopps, aveflop * tid )
end for
offset [ tnum ] ← m
```

3.Hash SpGEMM

We use hash-table for accumulator in SpGEMM computation, based on GPU work Algorithm 3 shows the algorithm of Hash SpGEMM for multi- and many-core processors. We count a flop per row of output matrix. The upper limit of any thread's local hash-table size is the maximum number of flop per row within the rows assigned to the thread. Each thread once allocates the hash-table based on its own upper limit and reuses that hash-table throughout the computation by reinitializing for each row. Next is about hashing algorithm we adopted. A column index is inserted into hash-table as key. Since the column index is no less than 0, the hash-table is initialized by storing -1 . The column index ($=$ key) is multiplied by constant number, c , and divided by hash-table size to compute the remainder. In order to compute modulus operation efficiently, the hash-table size is set as $2n$ (n is an integer). The hashing algorithm is based on linear probing.

In symbolic phase, it is enough to insert keys to the hash-table. In numeric phase, however, we need to store the resulting value data. Once the computation on the hash-table finishes, the results are sorted by column indices in ascending order (if necessary), and stored to memory as output. The Hash SpGEMM for multi/many-core processors compute each row of output matrix by single thread. On the other hand, multiple threads are assigned to a row of output matrix in the GPU version of Hash SpGEMM in order to exploit massive number of threads on GPU. Due to this design for GPU version, the Hash SpGEMM on GPU requires some form of mutual exclusion since multiple threads access the same entry of the hash-table concurrently. We were able to remove this overhead in our present Hash SpGEMM for multi/many-core processors.

Algorithm 3 : Hash SpGEMM.

Input: Sparse matrices A and B

Output: Sparse matrix C

$\text{offset} \leftarrow \text{RowsToThreads} (A, B)$

{Determine hash-table size for each thread}

$\text{tnum} \leftarrow \text{omp_get_max_threads} ()$

for $\text{tid} \leftarrow 0$ to tnum in parallel **do**

$\text{size}_t \leftarrow 0$

for $i \leftarrow \text{offset} [\text{tid}]$ to $\text{offset} [\text{tid} + 1]$ **do**

$\text{size}_t \leftarrow \max (\text{size}_t , \text{flop} [i])$

end for

{Required maximum hash-table size is N_{col} }

$\text{size}_t \leftarrow \min (N_{\text{col}} , \text{size}_t)$

{Return minimum 2^n so that $2^n > \text{size}_t$ }

$\text{size}_t \leftarrow \text{lowest_p2} (\text{size}_t)$

end for

Symbolic (rpts_C , A, B)

Numeric (C, A, B)

Critical Assessment

SpGEMM computation has two critical issues unlike dense matrix multiplication. Firstly, the pattern and the number of non-zero elements of output matrix are not known beforehand.

For this reason, the memory allocation of output matrix becomes hard, and we need to select from two strategies. One is a two-phase method, which counts the number of non-zero elements of output matrix first (symbolic phase), and then allocates memory and computes output matrix (numeric phase).

The other is a one-phase method, where we allocate large enough memory space for output matrix and compute. The former requires more computation cost, and the latter uses much more memory space. Second issue is about combining the intermediate products to non-zero elements of output matrix.

Since the output matrix is also sparse, it is hard to efficiently accumulate intermediate products into non-zero elements. This procedure is a performance bottleneck of SpGEMM computation, and it is important to devise and select better accumulator for SpGEMM.