COEN - 319 Assignment 2
Sparse Matrix Multiplication using OpenMP

Name : Piyush Kulkarni
Campus ID: W1629006

Objective:

The purpose of this assignment is to become familiar with OpenMP, by implementing sparse matrix-matrix multiplication.

Overview:

Sparse matrices are different from regular dense matrix multiplication due to the higher presence of zeros. Due to this factor, the matrices themselves can be represented differently in order to make the multiplication more efficient (since 0 values would not need to be multiplied). This representation is called Compressed Sparse Row (CSR) data structure.

The CSR representation consists of following arrays:

ptr: points to where rows start and end

ind : lists column ids for non-zero values

val: lists values for non-zeros

Implementation:

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 (A->rowptr) and row value equal to y in the second matrix (B->rowptr) will contribute towards the result[x][y]. This is obtained by multiplying all such elements having ind (column) value common in both matrices and adding only those with the row as x in the first matrix and row as y in the second matrix to get the result[x][y].

The CSR representations of A and B are sorted first, according to rows Here, sort() function of C++ STL is used. However, it is parallelized for, with dynamic scheduling.

The for loops in actual traversing and multiplication are parallelized using OpenMP pragmas.

However, the data-write operation in the sum variable, is specified as a critical section, so that only one thread can write to the resultant vector array. Length of the resultant vector is further used to reserve space for C. Furthermore, values are copied into CSR format of the result (C).

The code is run remotely using resources from XSEDE using the command:

```
srun --partition=compute  --pty --account=scu100 --nodes=1 --ntasks-per-node=56
--mem=32G -t 00:10:00 --wait=0 --export=ALL /bin/bash
```

OBSERVATIONS

For multiplying matrices A and B, following are the execution times observed, with different fill-factors:

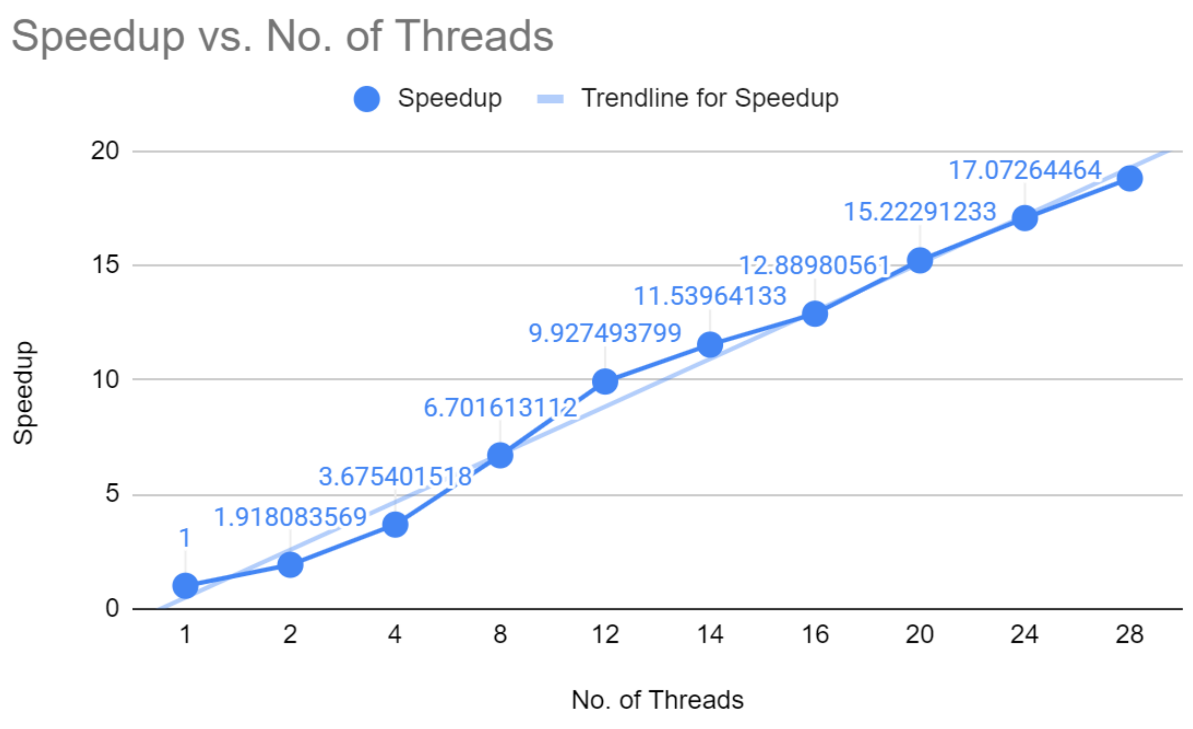
A nrows : 10000

A ncols : 10000

B ncols : 10000

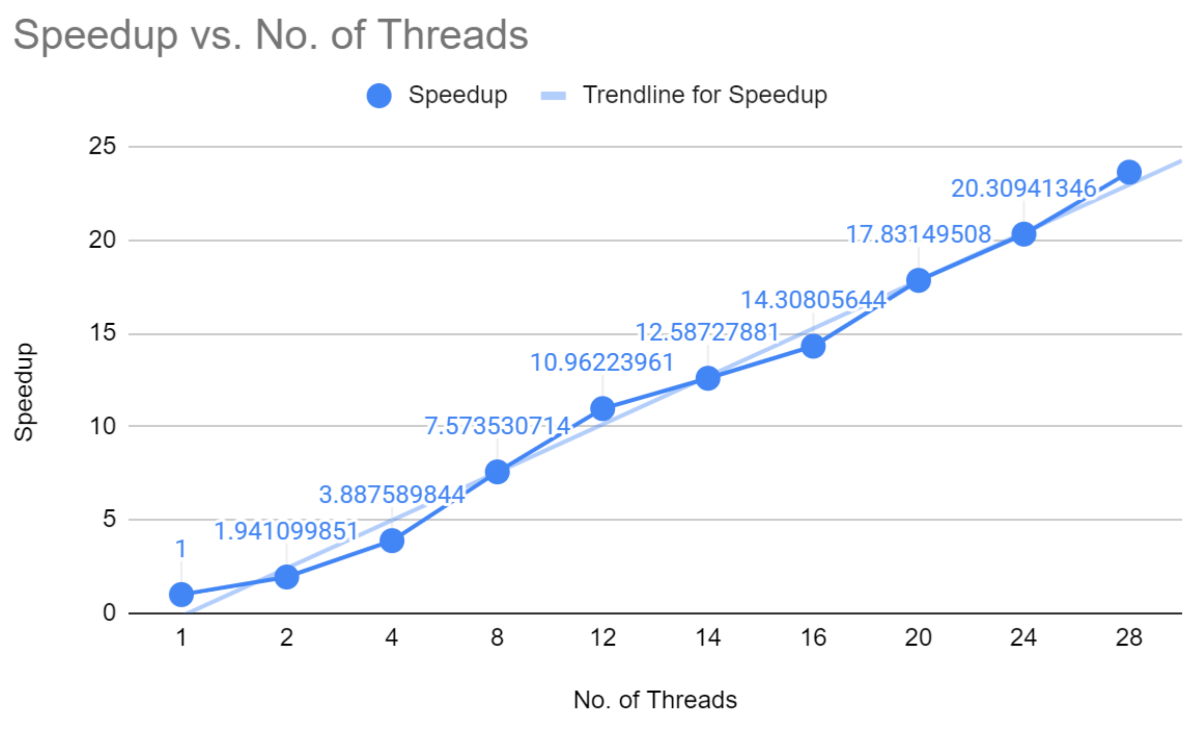
1. **Fill-factor = 0.05**

No. of Threads	Speedup
1	1
2	1.918083569
4	3.675401518
8	6.701613112
12	9.927493799
14	11.53964133
16	12.88980561
20	15.22291233
24	17.07264464
28	18.80557433



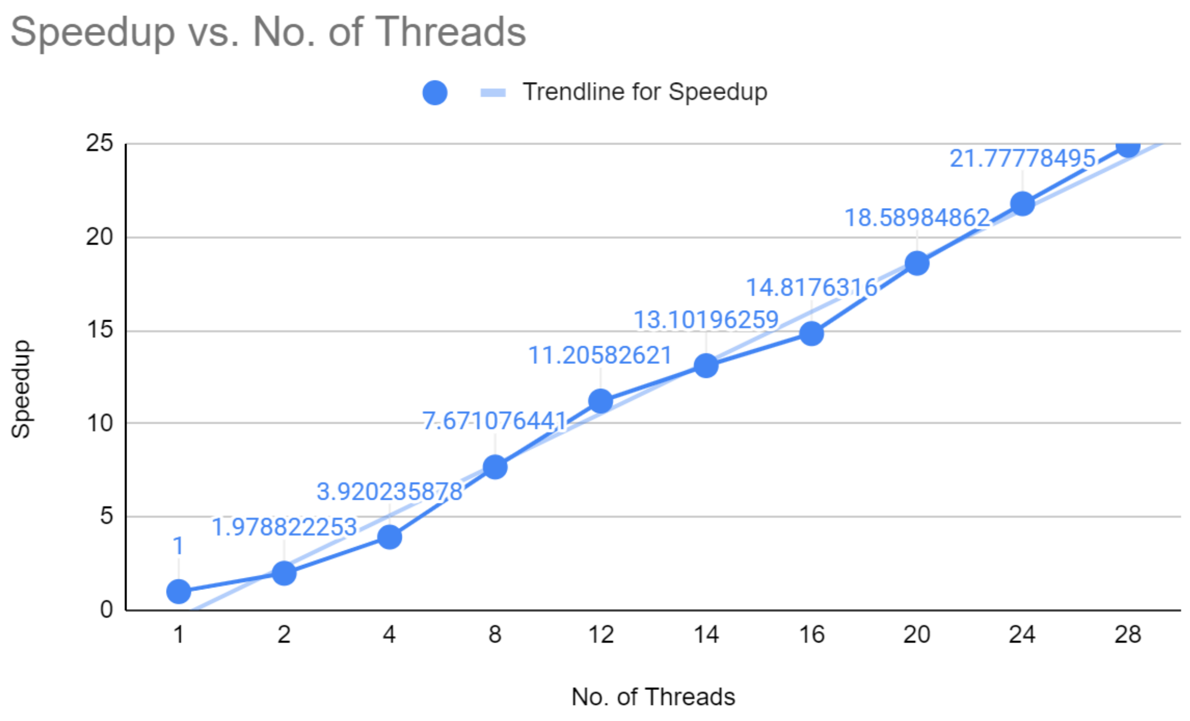
2. Fill-factor = 0.1

No. of Threads	Speedup
1	1
2	1.941099851
4	3.887589844
8	7.573530714
12	10.96223961
14	12.58727881
16	14.30805644
20	17.83149508
24	20.30941346
28	23.62859408



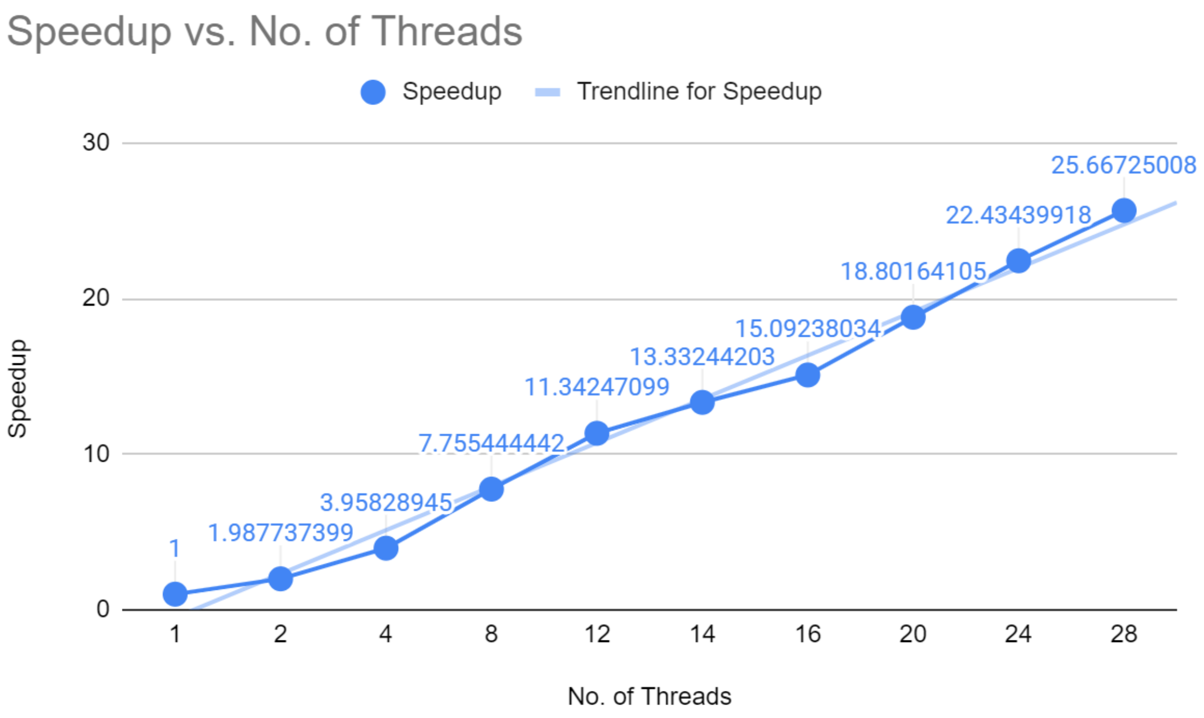
3. **Fill-factor = 0.15**

No. of Threads	Speedup
1	1
2	1.978822253
4	3.920235878
8	7.671076441
12	11.20582621
14	13.10196259
16	14.8176316
20	18.58984862
24	21.77778495
28	24.90206566



4. Fill-factor = 0.2

No. of Threads	Speedup
1	1
2	1.987737399
4	3.95828945
8	7.755444442
12	11.34247099
14	13.33244203
16	15.09238034
20	18.80164105
24	22.43439918
28	25.66725008



Inference:
As evident from the trendlines in the figures above, we see a linear growth in speedup, with the number of threads. The speedup almost tries to catch up with the number of threads in use. Different fill-factors produce different numbers of non-zeros, and thus vary in execution times. However, the constant growth trend is seen in all of them.
To further bring a reduction in runtime, we can increase the number of threads.
