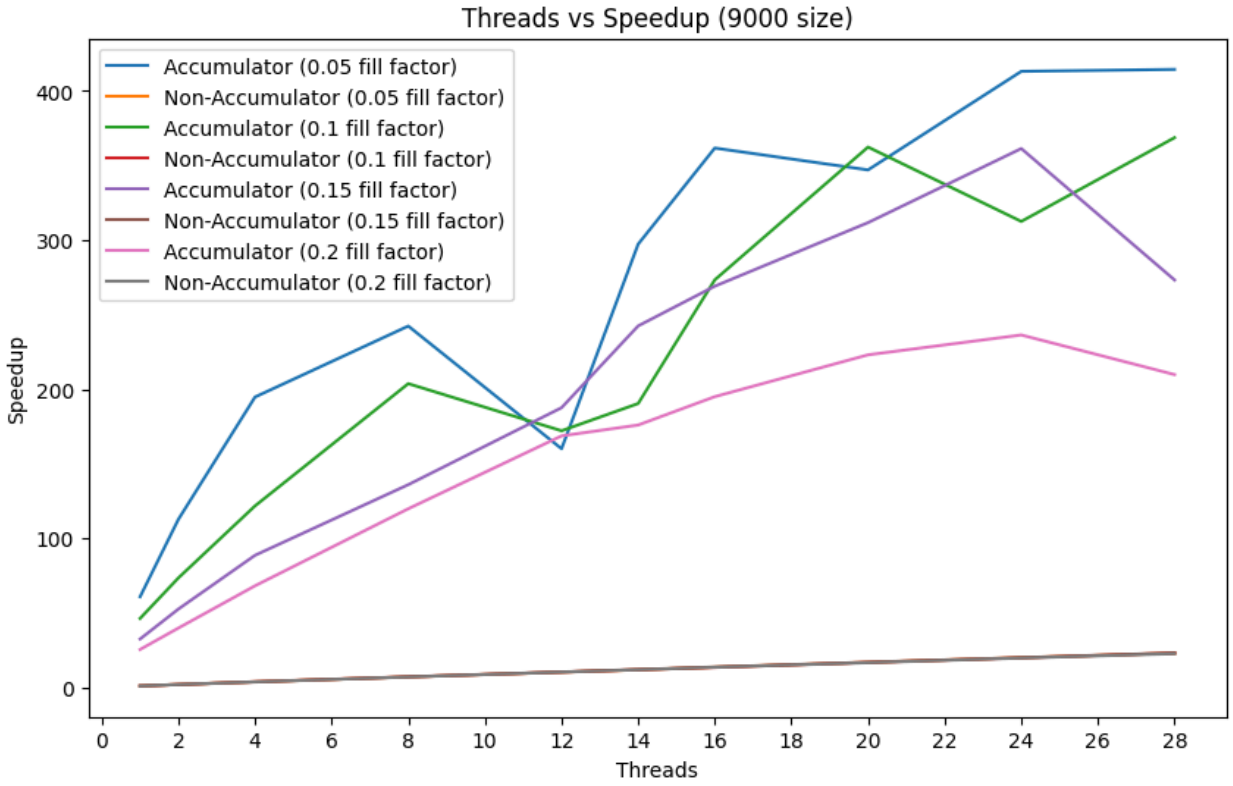
Erick Sun
CSEN 145
10/07/2024

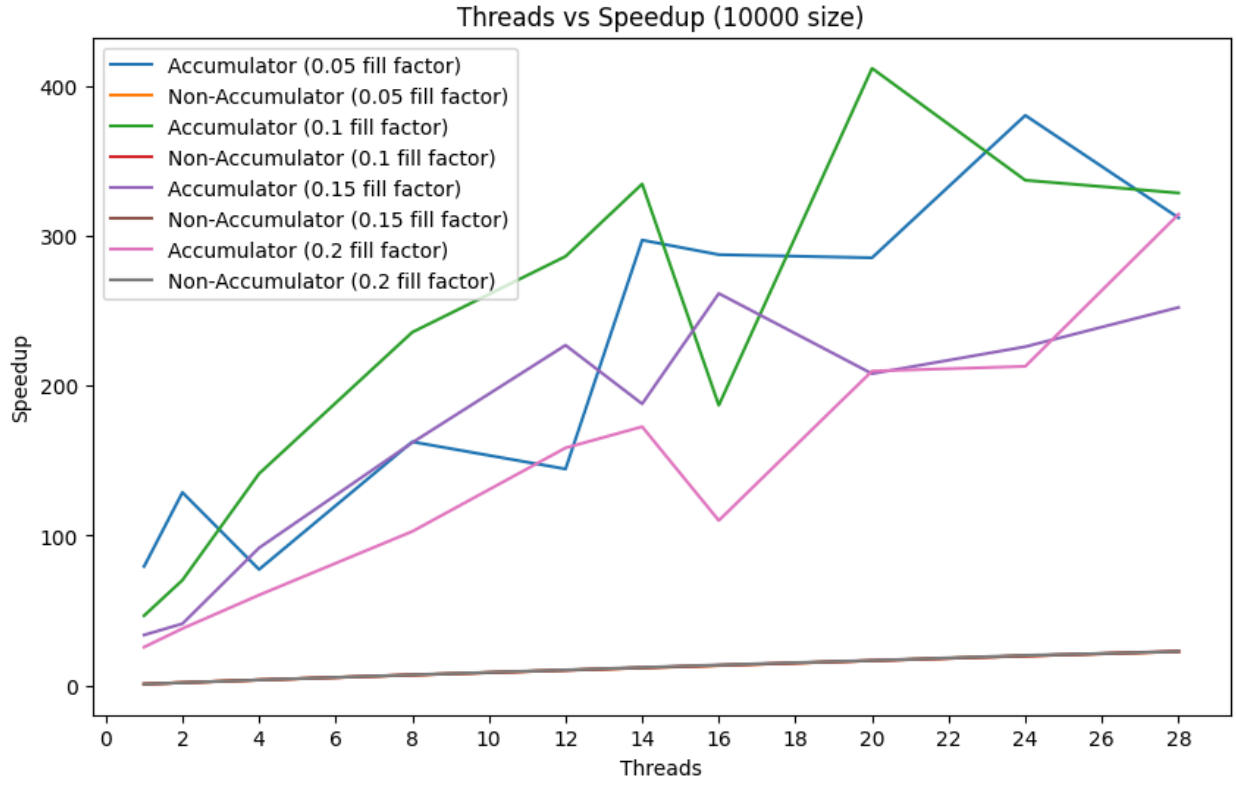
Accumulator-Based Approach to SpGEMM (OpenMP)

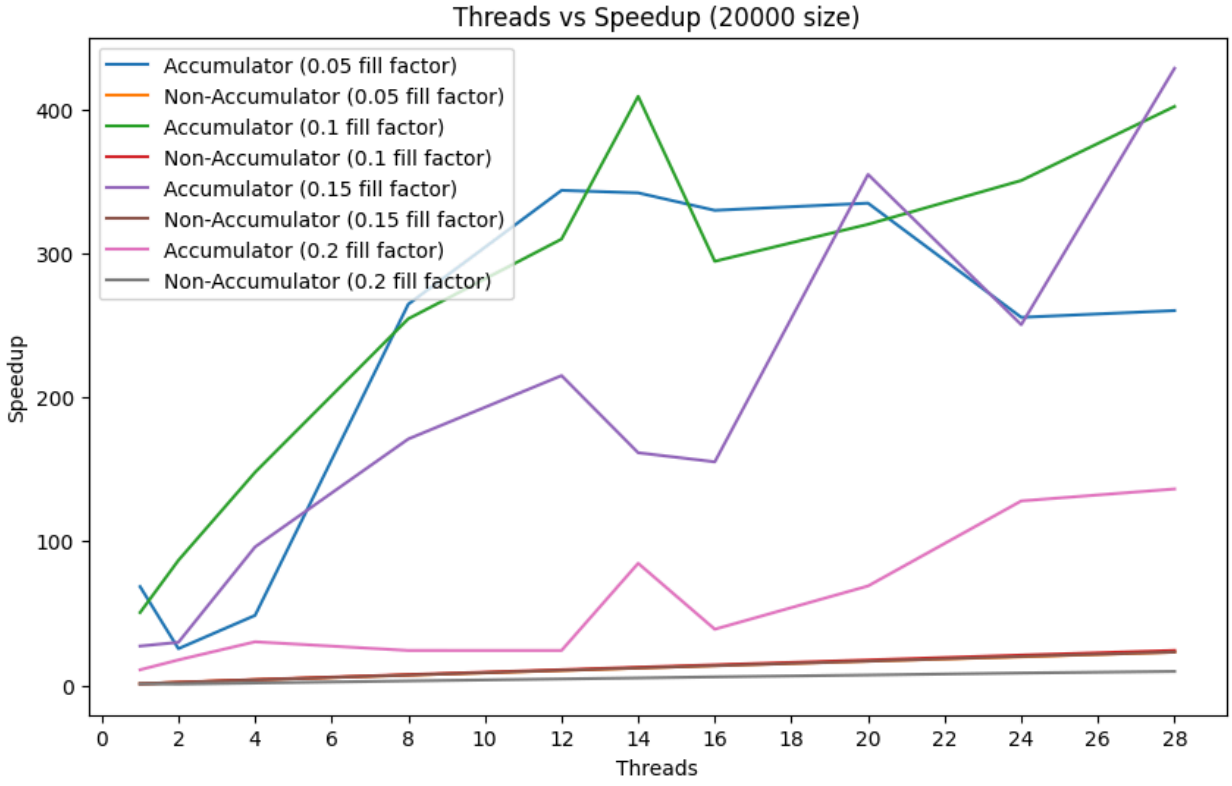
I. Parallelization Approach

When parallelizing this code, I opted to take the route of data decomposition. All other decomposition techniques were ill-suited for handling the task of SpGEMM. However, within data decomposition, I chose to divy up the work based on the output data size, instead of the input. This is to prevent any issues that may arise where memory is being written to and read simultaneously. However, by assigning the data based on the outputs, there will be no need for intercommunication between threads and no need to wait on other threads to write. However, after computation, race conditions will occur when writing the final computations to the C array. When updating the C array, due to the nature of the CSR structure and the pointer array, the order of the rows must be preserved when writing. As a result, an *ordered* directive was used to maintain this order.

II. Speedup




Threads vs Speedup (10000 size)
Accumulator (0.05 fill factor)
Non-Accumulator (0.05 fill factor)
Accumulator (0.1 fill factor)
Non-Accumulator (0.1 fill factor)
Accumulator (0.15 fill factor)
Non-Accumulator (0.15 fill factor)
Accumulator (0.2 fill factor)
Non-Accumulator (0.2 fill factor)
Speedup
Threads
0
100
200
300
400
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28


Threads vs Speedup (20000 size)
Accumulator (0.05 fill factor)
Non-Accumulator (0.05 fill factor)
Accumulator (0.1 fill factor)
Non-Accumulator (0.1 fill factor)
Accumulator (0.15 fill factor)
Non-Accumulator (0.15 fill factor)
Accumulator (0.2 fill factor)
Non-Accumulator (0.2 fill factor)
Speedup
Threads
0
100
200
300
400
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28

III. Analysis

From the graphs above, we can see the general upwards, positive correlation between the number of threads and the speedup for both implementations with and without an accumulator. But from the graph, the rates of speedup between the two implementations are very stark. The version utilizing an accumulator consistently reached a speedup of about four hundred. In comparison, the non-accumulator version had reached a max speedup of approximately thirty. This difference results in the plotting of the non-accumulator algorithms very difficult to view. In addition, across all densities, the speedup of the non-accumulator based algorithm was overwhelmingly consistent. As a result, not only are the speedup lines very difficult to interpret, they are also essentially the same line with the same values for almost all cases. This is in stark contrast with the cases for the accumulator. Although not completely consistent, it seems to follow that the speedup is greater for cases of lower fill factors. This may be a result of the nature of an accumulator-based algorithm. As the number of non-zero values in the matrix approaches the full size, the benefits of an accumulator begin to become less apparent. The benefit of the accumulator hinges on the fact that each product calculated will be placed in some value within the matrix, and there is no situation where the indices of the two values must be checked for validity beforehand. However, as the matrix becomes more full, the benefit diminishes as the effect of not having to make comparisons becomes a smaller portion of the overall calculations made. Furthermore, it seems that increasing the problem size seems to affect the performance of trials with higher fill factors. As the size increases, so does the time spent by each thread in the *ordered* section. In particular, since the maximum number of values that need to be copied in the *ordered* section is equal to the number of columns in the B matrix, the wider the matrix, the greater the time spent. This is because the rows of C are computed in parallel, however the values within the rows are all computed and copied in serial. This makes the program much more susceptible to wide matrices.

Taking a look at the graphs, we can conclude that there is a positive correlation between speedup and the number of threads. In addition, we can see the characteristic logarithmic growth associated with speedup, where the difference in threads begins to affect speedup less as the number of threads increases. However, this correlation seems to be very sporadic and chaotic, with a lack of consistency. The trend does not appear to be strictly increasing and there are many instances where an increase in threads seems to lead to the same, or even a drop in speedup. Although the reason for inconsistency is not completely known, I believe that it may lie in the randomness in generating the sparse matrices. If in a given matrix, there happens to be many more non-zero values in a specific column of B, this can result in substantial slowdowns in the overall runtime. This is a result of the nature of this parallelization approach, which utilizes an *ordered* segment of code. Within this execution, it is likely that the majority of the time spent is spent waiting to traverse through the ordered section, instead of doing actual computation. If a specific thread gets assigned a column of B that has many more non-zero unique row ID values, that single thread would spend more time in the ordered section copying the values from the temporary array to the C matrix. If this one thread takes much longer, this may cause a large buildup blockage for all other threads attempting to enter the ordered section. To minimize this, the copying of values to the C matrix should be as evenly distributed as possible. Due to the sparsity of the matrix, it is very difficult to evenly distribute the workload. In addition, any further subdivisions of the data distribution will likely result in many more critical or atomic segments of code.