matmul_sparse: I have implemented the *matmul_sparse* function in the *matrix_ops.c* file.

Functional logic: The main idea of the function is to skip the multiplication of all zero elements from both matrices. The assumption is that the input matrices have sparsely populated non-zero elements.

Algorithmic logic: The algorithm for this function is quite different in comparison to the initial matmul algorithm.

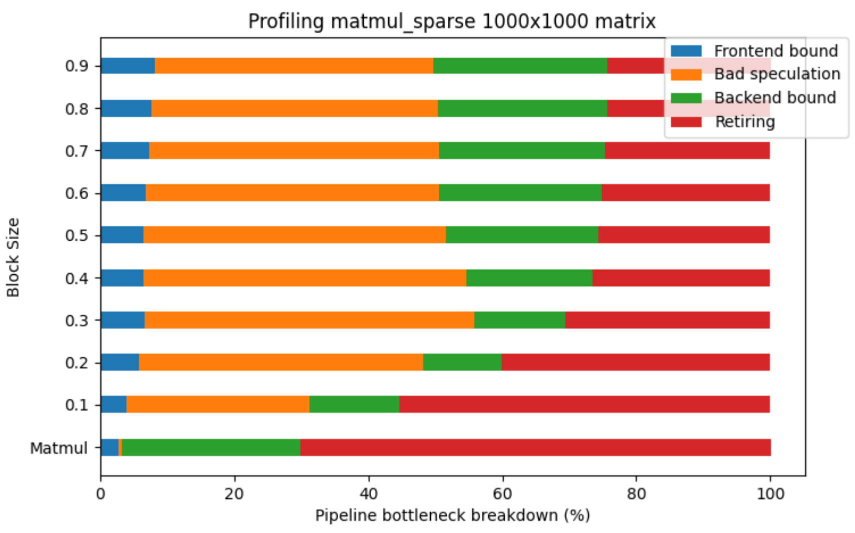
Before the actual algorithm is run, we convert the first matrix into a CSR data structure and the second matrix to a CSC data structure. The insight is that the first matrix will be called by row while the second matrix will be called by the column number in the multiplication. To create the data structures, I count the non-zero elements of the matrices to allot the exact memory. Then, I iterate through the matrices to add all non-zero matrix elements in the *'_val'* storage array, the elements row and column index for CSC and CSR respectively in the *'_idx'* array, and the relevant row and col start and end indexes in the *'_ptr'* arrays.

Once the data structures are created the actual algorithm is run. The algorithm starts with a wrapper loop that iterates several times. The main algorithm is itself divided into only two *for-loop* steps. The first iterates through the rows of the first matrix (A_rows) and the second iterates through the cols of the second matrix (B_cols) to calculate the output for each value of the output matrix. Inside the second step, we retrieve the values of the start and the end pointers of the row of the first matrix and column of the second matrix as these will contain all values that we need to multiply for the output of the current cell in the output matrix. Then, we run a while loop for all the elements in the selected row and column. For each element in the selected row, we get its corresponding column index and for each element in the selected column, we obtain the corresponding row index. Once, we have the indexes we check if they are equal, if so, then we have found a hit of two elements which should be multiplied for the matrix multiplication. If not, then there are two cases, the A_col is lower than the B_row or vice versa. The first case highlights that all cols between B_row index A_col index are zero and hence we must increase B_col index, while the second case reflects the exact opposite situation and hence we must increase A_row index. Once the while loop is run, we have essentially multiplied the entire row *i* and col *j* of the two input matrices to get the output at cell *i, j.*

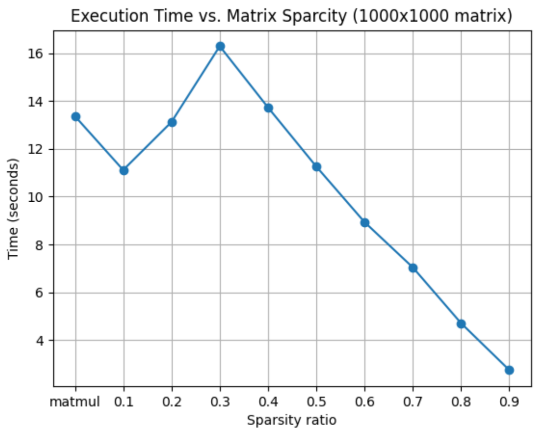
Profiling: For profiling, I ran the *matmul_sparse* function on matrices of the size (1000 x 1005, 1005 x 1000), (100 x 105, 105 x 100), and (20 x 30, 30 x 20). I ran all the inputs for matrix sparsity of 0.1, 0.3, 0.5, 0.7, and 0.9. I recorded the total runtime of all the computations and drew them in a graph (the recorded run times were the average over 5-6 runs with all the runs being quite close).

(1000 x 1005) X (1005 x 1000): In the profiling graphs, the matmul bar gives the performance analysis for the original matmul function (baseline) while all others give the analysis of their respective sparsity ratio of the input matrices in the matmul_sparse function for the input of 1000 x 1005 matrices. It is obvious from the graph that the matmul_sparse function results in a much more inefficient CPU performance than the original matmul function. In fact, as the sparsity ratio of the matrices increases (frequency of zero elements increases), the performance keeps on deteriorating where the main inefficiencies arise from bad speculation. The reason behind this can be attributed to the large number of branch mispredictions. Essentially, the data in the matrices is randomly generated without any specific relation between the occurrence of non-zero elements which results in a large number of mispredictions inside the second for loop of the main algorithm. The inefficiencies in the backend bound seem to arise from the misses in the L1 cache as the data stored

For this profiling, the algorithm was iterated only once as even one iteration took more than a second to run and hence, the majority of the calculations would be in the main CSR-CSC multiplication algorithm.

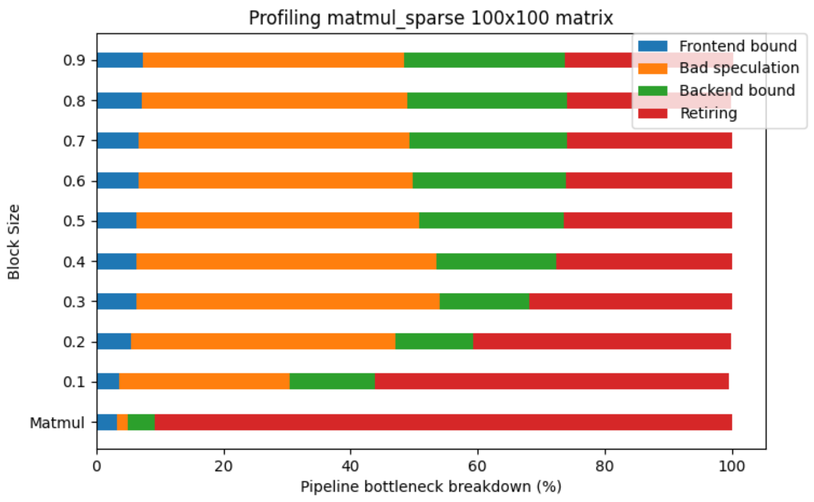


In terms of time, the naïve implementation takes around 13.364708 seconds to run while for the sparse matrix multiplication, the initial computations for sparsity 0.1 to 0.3 took progressively more time to run but then steadily decreased as the sparsity ratio increased. When the two graphs are compared, it can be observed that there is a steady increase in the bad speculation (branch misprediction) as the sparsity increases from 0.1 to 0.3 while the computations themselves do not decrease that much. This is a strong example to demonstrate that the miss cycle penalty for each branch miss prediction is quite high because that is the only explanation to justify why the algorithm takes more time to run even though the actual number of computations for sparsity 0.3 is much lower (more than 20%) than the computations for matrices with sparsity 0.1. We also observe that as the sparsity increases beyond 0.3, the bad speculation decreases slightly but that decrease is compensated by the slight increase in the frontend and backend bounds. So, even though as the matrix sparsity increases the overall inefficiencies in the computations remain relatively constant, the actual computations decrease linearly and therefore we see a steady linear decrease in the time taken for the algorithm to run.

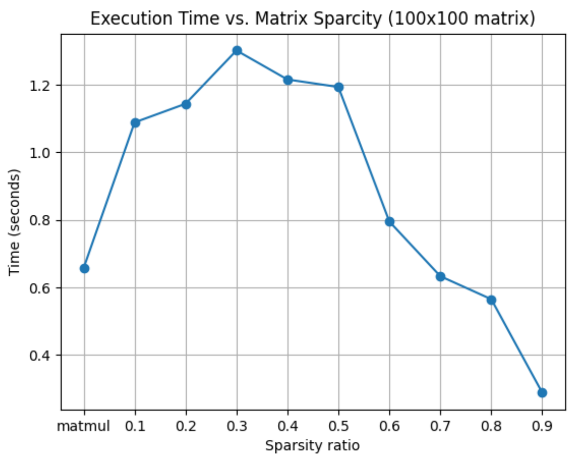


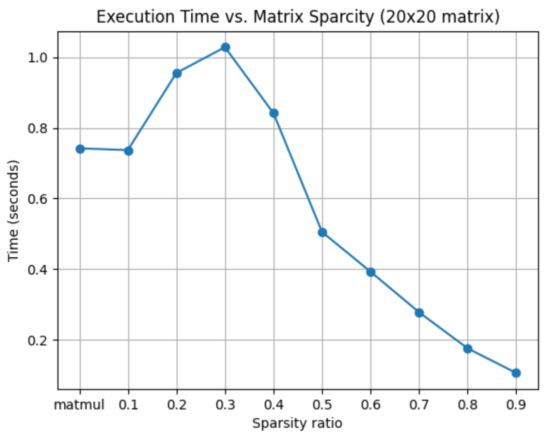
(100 x 105) X (105 x 100):

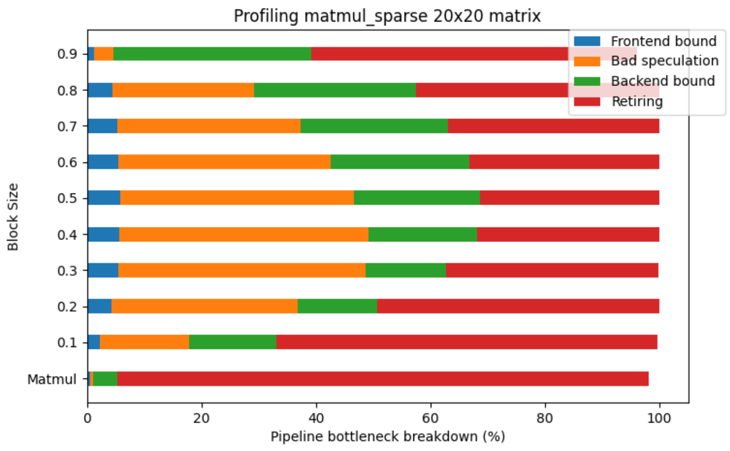
For the 100x100 matrices, we iterated over the matrix computations 100 times to ensure that most of the operations in the top-down analysis are run for the actual matrix computations. The results for the profiling of these matrices are very similar to that of the 1000x1000 matrices. Essentially, there is a significant increase in the bad speculation (branch misses) as



the sparsity increases from 0.1 to 0.3 and after that, the overall inefficiencies remain relatively constant. There is also a similar result for the time graph for the runtime of the main algorithm. There is a steady increase in the time taken to run the main algorithm as the sparsity increases to 0.3 but after that, there is a steady linear decrease of the runtime. The arguments to explain these observations are similar to those for the previous matrix sizes.







(20 x 30) X (30 x 20):

Similar to the previous two cases, increasing the sparsity of the input matrices to 0.3 steadily increases the bad speculation in the CPU and also the runtime. However, unlike the previous two cases, there is a steady decrease in the bad speculation as the sparsity of the matrices is further increased. The only logical explanation for this decrease in bad speculation is the size of the input matrix itself. Essentially, because the matrix is so small and also highly sparse, the actual computations are quite less and in the second step of the matmul_sparse algorithm, which is consistently responsible for the high branch mispredictions, and therefore, unlike the previous two cases the CPU dramatically improves the branch prediction.

Conclusion:

We can conclude that the matmul_sparse function significantly improves the runtime for highly spare matrices even though the CPU performance for matmul_sparse is significantly worse than it is for the naïve implementation of matmul. On the other hand, for lowly spare matrices, the matmul_sparse function is pretty bad in both the actual runtime of the algorithm and the CPU performance. Furthermore, this performance of the algorithm seems to be independent of the input size of the matrices as the inputs for three different magnitudes produced similar results (except for the smallest matrix inputs which can be attributed to the inherent small size of the matrix and thus an edge case).