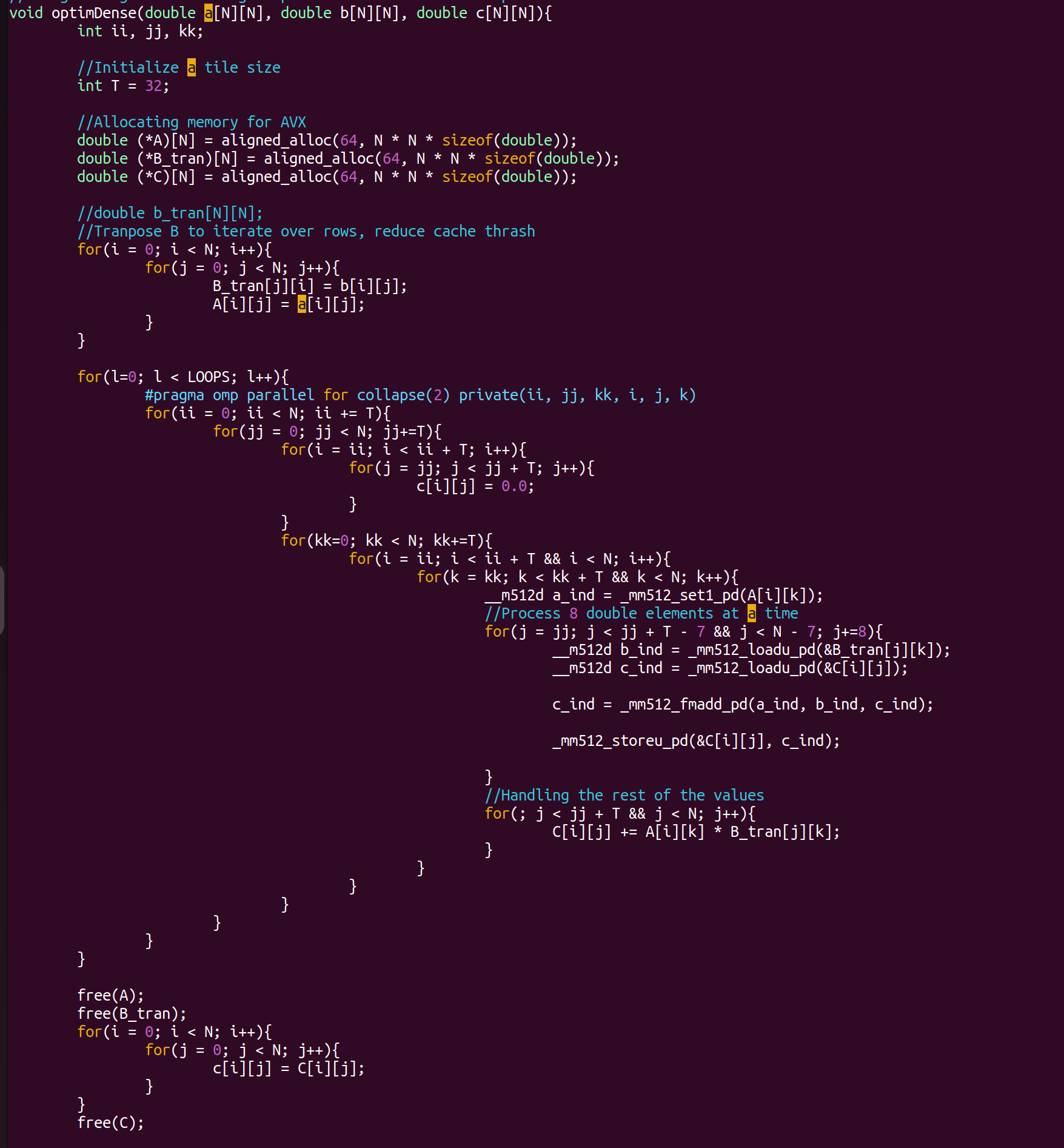
Question 3:

In this problem, you will modify the matmul.c program provided, optimizing the execution of the matrix multiplication with first a dense matrix, and second with a sparse matrix. You are welcome to use pthreads, OpenMP or any of the optimizations that were presented in class to accelerate this code. Do not change the sparsity of the matrices in matmul.c. Do not use a GPU and do not use OpenBLAS in your solution. There will be prizes awarded for the fastest dense and the fastest sparse implementations.

Dense:

My dense optimization was able to improve the efficiency of the given 256x256 matrix multiplication from 318.43 ms to 11.5 ms. This was done using a combination of techniques:



Initially, I allocated memory for 2D arrays on the heap, which enables the use of AVX512 for SIMD operations to improve efficiency. I then transposed the B matrix to reduce cache thrashing and use spatial locality effectively. For the loops, I implemented tiling, with a tiling size of 32 based on the CPU’s cache size specifications. Tiling was done in parallel with OpenMP as well, and the number of threads was configured by OpenMP and not manually set. Within each tile, computation was done with AVX, where the A index was initially loaded, and then the B and C indices were loaded in a loop that stored the updated C result values. For any remaining values, they were manually computed, though not necessary to do the fitting of the 256 in the 32 size blocks. To save the value of c, I copied it to the original matrix.

Sparse:

To optimize sparse matrix multiplication, I utilized one of the methods mentioned in class. In class, we discussed Compressed Sparse Row formatting of matrices, which collects the non-zero elements as well as the rows and columns. This data is collected in a CSRMat struct, which is called in the optimSparse() function to get the CSR Matrices for A and B. The getCSR() function is used to do this for both matrices. Rather than iterate through N x N x N loops to determine the product matrix C, with CSR matrices C can be calculated with one N loop and two nested loops that determined the value of elements at A and B at their respective indices.

In terms of results, for a 256x256 matrix, this code ran at a speed of 1.28 ms, whereas the regular matrix multiplication took around 420 ms. This is a massive speedup, and is further exemplified by the 512 x 512 matrix runtimes. At this matrix size, regular sparse matrix operations result in a 3876.43 ms runtime, while CSR matrix multiplication requires just 7.43 ms.

I attempted to parallelize the CSR computation, but due to race conditions and synchronization overhead, saw insignificant speedup and at N=256, a slight decrease in efficiency.

