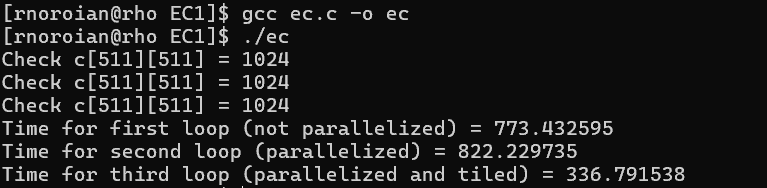
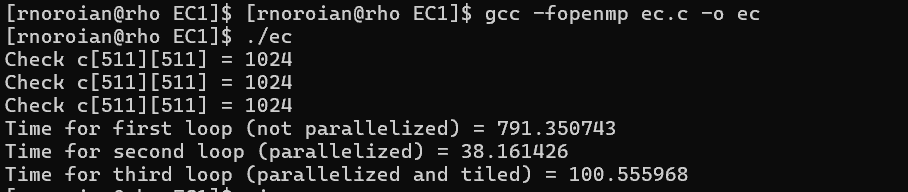
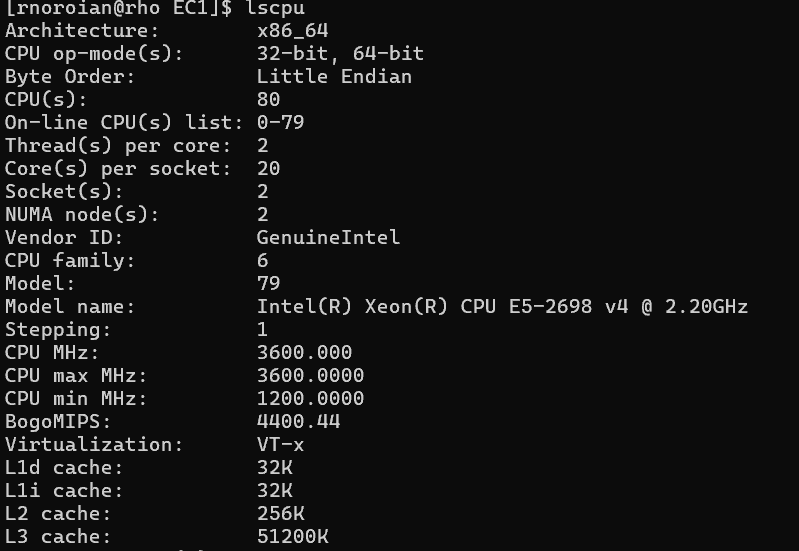
Explanation with Trials:  
The first loop is regular, without parallelism, the second is with OpenMP, the third is OpenMP with tiling.

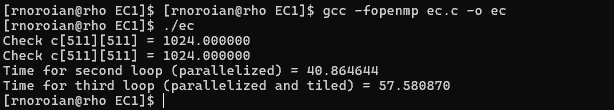
When we were in class, the code was outputting this:

The second loop, despite using an OpenMP pragma, has a greater runtime than the first loop. I realized that this was due to, as was occurring in class, the lack of the -fopenmp command in the compilation of the program. With it added, the program outputs these results: In this case, OpenMP enables a massive speedup, but with parallelism and tiling we see an increase in runtime. I also made the matrix datatype doubles to get a greater magnitude difference in the runtimes of each. To test the reasoning for inefficient tiling, the tile sizes were varied. Initially, in the given program, the B is set to 64. This means the multiplication was split into an 8 x 8 grid of tiles. However, it is possible that given my CPU specs:

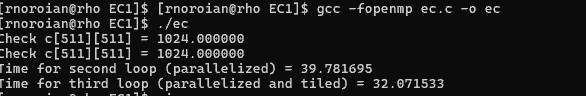


The tile sizes were too large, they may not fit into the cache, leading to slower data fetching from the main memory and a loss of temporal locality. I reduced it to 32, 16, 8, and 4.

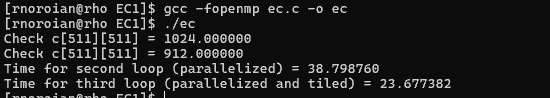
32:



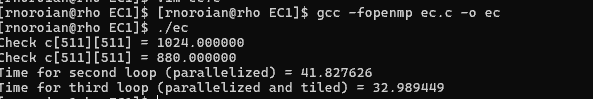
16:



8:



4:



Based on these results, it is evident that the maximum performance was achieved with 8 x 8 tiles, forming an entire grid of 64 x 64. This means that this tile size best aligns with my cache sizes of 64K, 256K, and 51200K for L1, L2, and L3. It is also possible that though the greater tile sizes still fit in the cache, they introduce overhead that makes them less efficient. The 8x8 matrices fit in the L1, enabling greater temporal and spatial locality. Additionally, since I possess 80 cores, more threads are able to run, so this takes place fully in parallel.