Lab 2

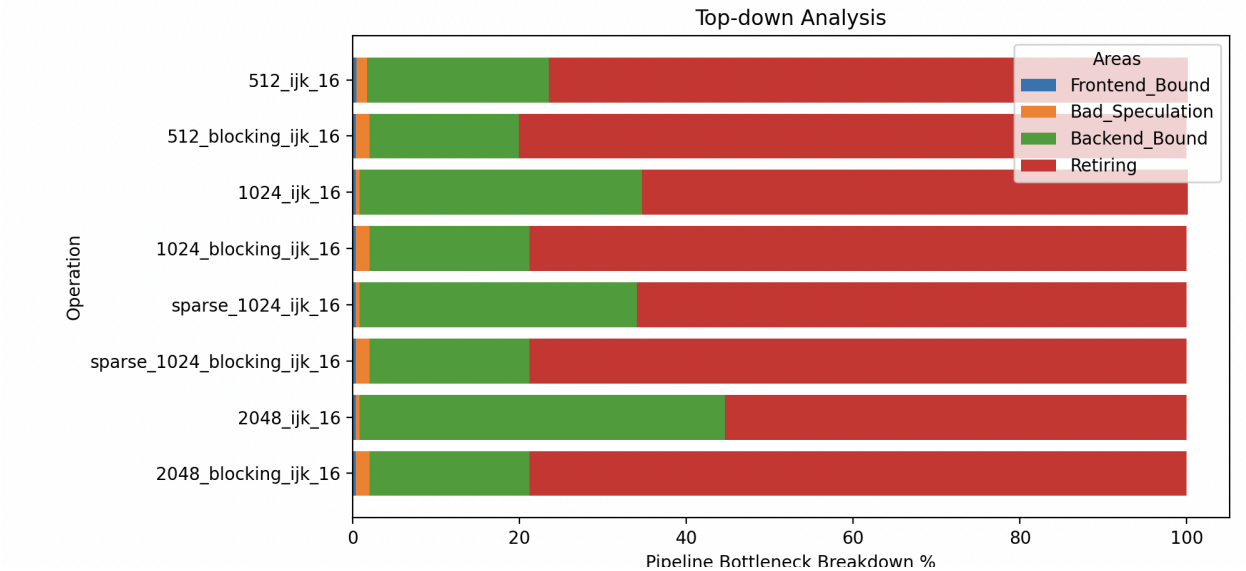
Aneesh Pandoh 10/31/2024
(Using 1 Late Day)

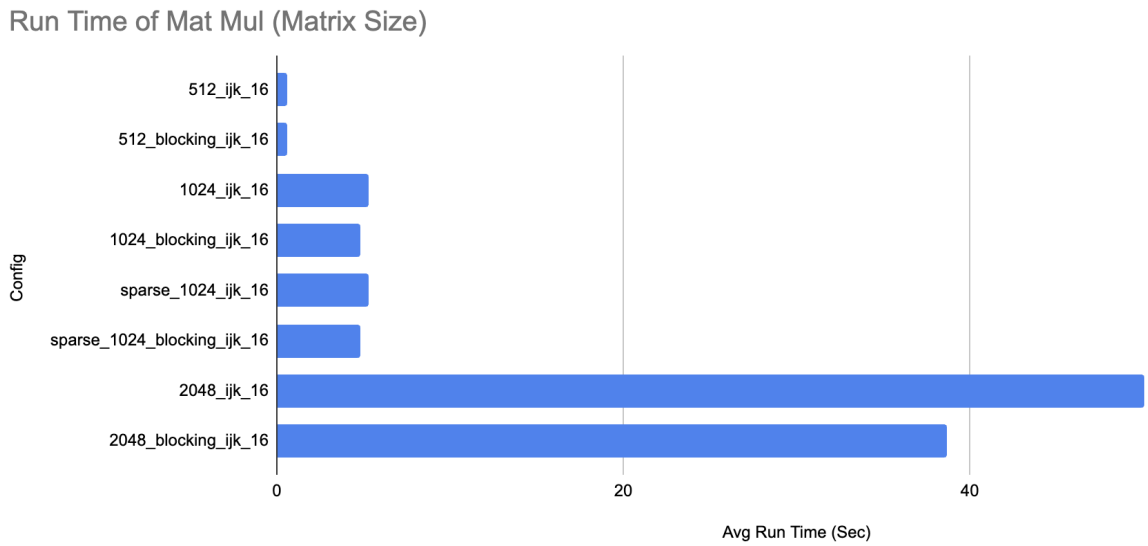
Setup:

Ran Top Down Analysis on each test 5 times. For each operation, I ran the tests 10 times besides matmul of 2048x2048 as it would log me out of the server before completing, so it is set to 5. I tested matrix sizes of 512, 1024, and 2048. I also tested between the block size of 16, 64, and 128. I also tested 3 different loop orders as well as a sparse patterning.

*In the 3 graphs **1024_ijk_16** represents the baseline which is the same configuration as matmul in lab1

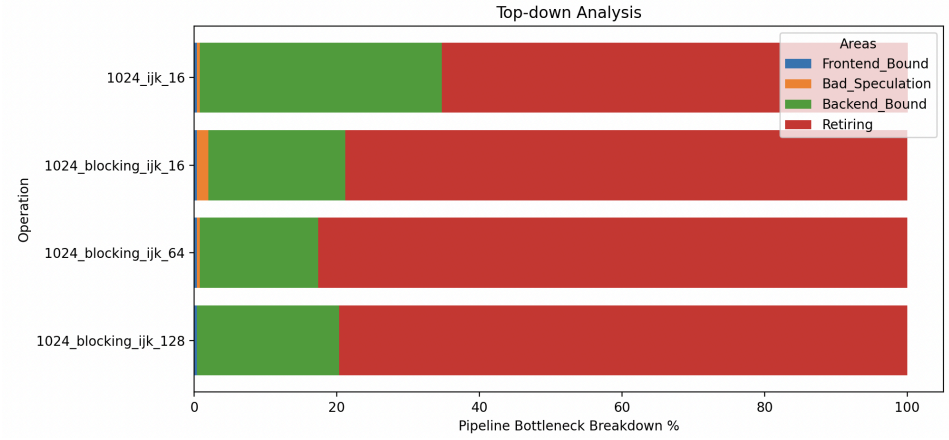
Blocking vs No Blocking (Blocking is indicated by _blocking_)

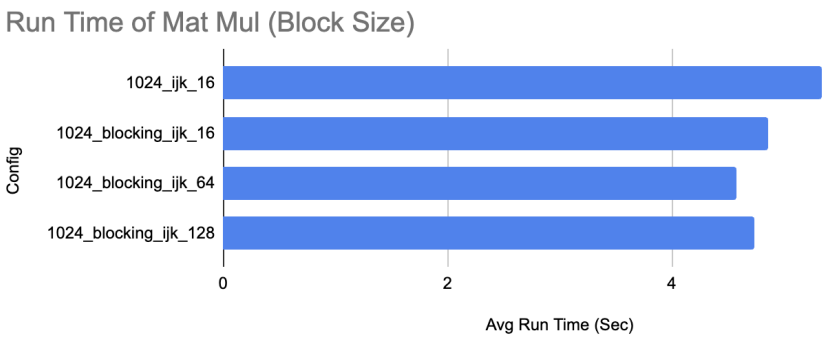




We are able to see clear performance improvements due to blocking, which get more obvious as we increase the matrix size, as illustrated in the run time graph. From the top down analysis we see that our original non-blocking implementation shows a larger “Backend Bound” bottleneck, indicating that the CPU is waiting on memory operations more frequently, which slows down processing. Blocking helps by improving spatial locality as it loads consecutive rows/cols into the cache and matmul goes in consecutive order. It also helps in temporal locality as the same elements in the matrix block are reused, and the blocking ensures that they stay in the cache. The cache misses results in the processor having to reach higher in the memory hierarchy significantly impacting runtime. We can also see from the data results that the sparse and non-spare 1024 size matrix results in the exact same runtime and top down analysis, meaning that despite having less data in the matrix that we need to process, we are not able to optimize and process the matrix with fewer operations. We can also see the “Bad Speculation” increase slightly for blocking, this is likely due to increased complexity due to the increased loops, but is still a minor hit to performance compared to the backend bound reduction we get.

Block Sizes: (Block size is indicated by _X)

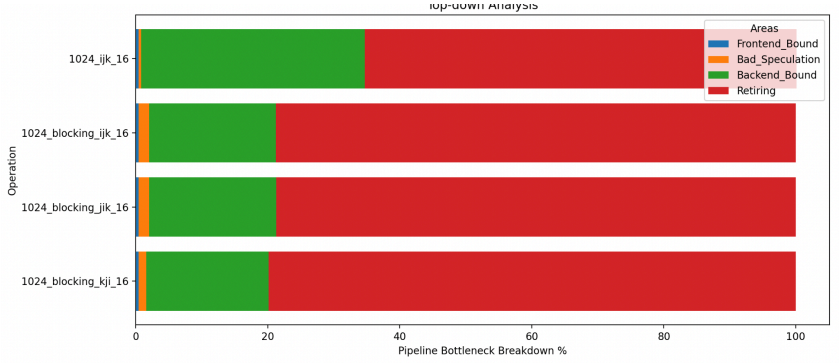


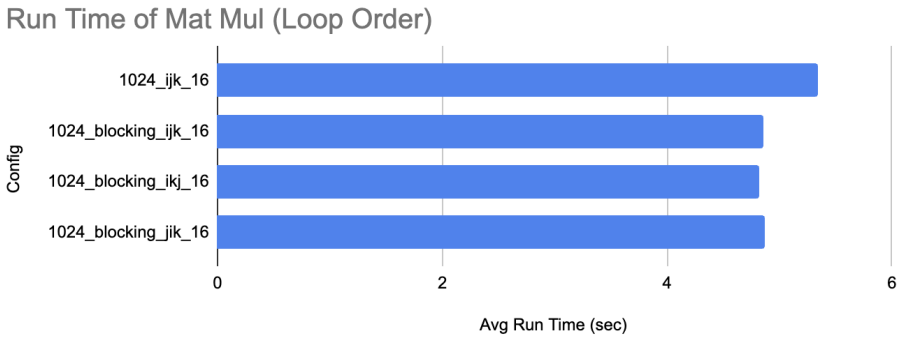


We can see that the block size of 64 seems to be the best for our processors as it utilizes the cache most effectively. The smaller block size of 16 did help compared to not blocking at all, but we might be wasting memory bandwidth by reading entire cache lines but the relevant data is only a part of the cache line. In the case of 128, the block size is too large; it may be more data than what fits into the cache line, making us need to take up more space in the cache for a single

block, which can have us lose some of the benefits of temporal locality. It looks like in this case a block size of 64 is ideal as it fits within the lower level caches while maximizing the amount of data cached.

Loop Order: (Loop order is indicated by _xyz_)





Here I modified the loops where i iterates over rows of matrix A, j iterates over columns of matrix B, and k iterates over columns of matrix A/rows of matrix B. For these changes of ordering, I changed them both for the iterating over blocks and the elements within the blocks. We can see that ordering did not end up mattering very much relative to changing block size or simply just blocking in general. However, changing the loop ordering should usually drastically change the results as we could be constantly be missing the cached data because we are not looking for data in order, but this has probably not been found in this example because the processor is recognizing the pattern of how the matrix is being accessed and prefetching accordingly. I also believe that the blocking makes the loop ordering redundant but for matmul without blocking, there could be a bigger effect of changing loop orders.

Issues:
Did not have many issues with this lab.
*Used github copilot / and chatGPT to speed up the process writing tests and implementation

Improvement:
Use Compressed Spare Row form for sparse matrices
Implement data/thread level parallelization