matmul_blocking:

I have implemented the *matmul_blocking* function in the *matrix_ops.c* file. The function has an additional parameter (*int block*) which I added for personal convenience as it allows me to pass the block size argument in the command line. The logic is explained as follows:

Functional logic: The main idea of the function is to exploit the spatial locality of the L1 cache while computing the matrix product. To do that, the function uses the technique called blocking where instead of iterating through each cell of the product matrix one by one to calculate the final product, we divide the entire matrix into smaller square matrices (rectangular at the edges) and calculate the products for the smaller matrices one by one. Essentially this allows us to exploit the spatial locality of the L1 cache by calculating the final answer within the already loaded cache data increasing cache hits and decreasing the time wasted in loading data from higher memory levels.

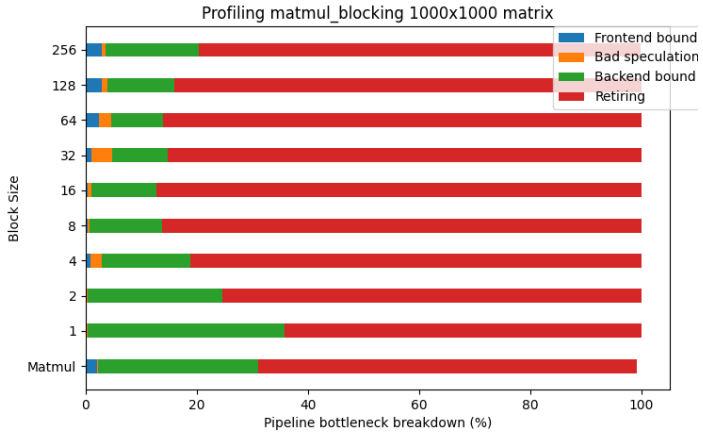
Algorithmic logic: The algorithm for this function looks similar to the initial matmul algorithm but with more specificity. This algorithm is divided into 6 steps with the first three steps being the same as the initial algorithm where the first step iterates through the rows of the first matrix (A_rows), the second step iterates through the cols of the second matrix (B_cols), and the third step iterates through the rows of the second matrix (B_rows or A_cols). In the modified algorithm, instead of iterating the first three steps one by one, we iterate them through each block size because the values inside the blocks are computed in the next 3 steps of the algorithm.

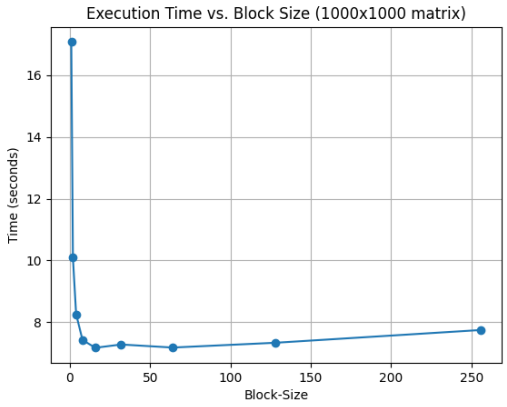
The next 3 steps of the algorithm (steps 4, 5, and 6) are logically the same as the first three steps because they compute the products of a square matrix of block_size but with two main differences:

1. The relative position of the block in the larger parent matrix. Essentially, the indexes for the smaller block_size matrix start from the first cell of the relative position of the matrix (i, j) and iterate one by one until the edge of the larger matrix or the smaller block_matrix is detected.
2. The jj index is swapped with the kk index in the 5 and 6 steps in the loops. This is fundamentally done with the observation that in the final computational equation, the jj index is accessed by two different matrices while the kk index is accessed by one and the ii index is accessed by none (in the column vector of the matrices). In the L1 cache, because each of the three matrices will be cached at the same time, it makes sense that by lopping over the jj index in the innermost loop we will exploit the spatial locality of the L1 cache in an even better way than looping over the kk index.

Profiling:

For profiling, I ran the *matmul_blocking* function on matrices of the size (1000 x 1005, 1005 x 1000) and (100 x 105, 105 x 100) as these pairs of matrices are large enough to fill an L1 cache whose size is 64kb. I ran profiling for the *matmul_blocking* function for both inputs for different block sizes ranging from 1 to 256 (128 for smaller), recorded the total runtime of all the computations and drew them in a graph. I did not run larger matrices as they would take exponentially more time to compute while smaller matrices would completely fit in the L1 cache defeating the purpose of blocking. Still, I ran profiling for the pair (20 x 30, 30 x 20) to confirm my hypothesis.





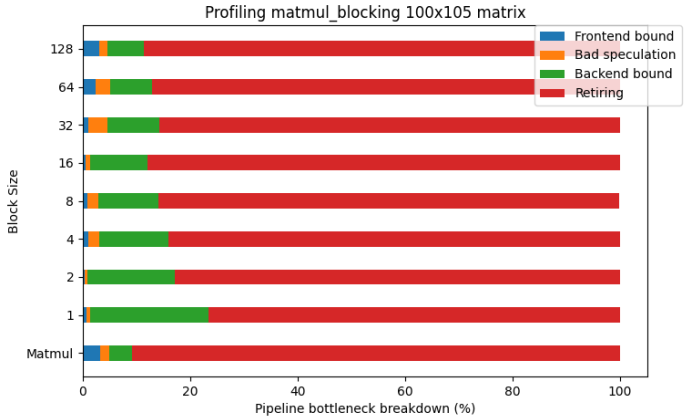
(1000 x 1005) X (1005 x 1000):

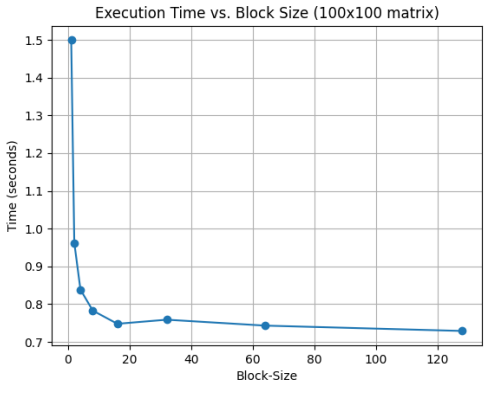
In the above graph, the matmul bar gives the performance analysis for the original matmul function (baseline) while all others give the analysis of their respective block_sizes in the matmul_blocking function for the input of 1000 x 1005 matrices. It is obvious from the graph that the matmul_blocking function results in a more efficient CPU performance than the original matmul function for block_sizes greater than 1. Furthermore, the backend bound reduces by almost half when using the block_size 16 as compared to when the block_size is 1. Moreover, as the block_size increases thereafter, there is an increase in either bad speculation or frontend bound penalties which highlight increasing inefficiencies even though the backend bound is relatively the same or even lower when compared to the block size of 16. It is safe to conclude that a block_size of around 16 to 64 results in the best CPU performance and the least inefficiencies in the matrix product calculation.

These observations are consistent with the observations from the time vs block_size graph where the execution time reduces exponentially by more than half for the matmul_blocking function as the block size decreases to 16. The time then remains relatively constant until the block size increases to 64 after which the execution time increases linearly as the block size increases.

In this implementation, I only iterated the matrix computations once as the matrix size was large enough to run the bulk of the operations for the top-down performance analysis.

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

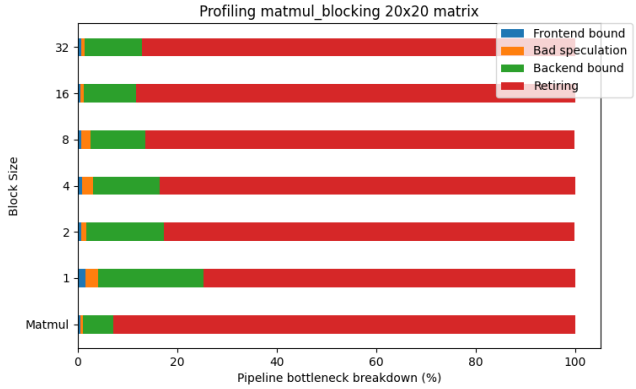


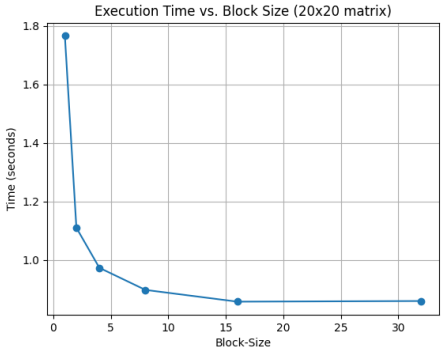


For the 100x105 matrices, I iterated over the matrix computations 100 times to ensure that the majority of the operations in the top-down analysis are run for the actual matrix computations. Doing

this, though, introduces a caveat. The algorithm in the matmul_blocking function does not compute the final answer for each cell of each block-size matrix in its own iteration, instead, the solution is temporarily stored for each cell of the final output matrix for each block-size matrix multiplication and aggregated over all the blocks. Essentially, it is not possible to initialize the values of the output matrix to zero within the actual matrix multiplication algorithm. Hence, for each iteration, the algorithm first needs to initialize the values of the output matrix to zero which is O(n^2) run-time. The actual algorithm runs in O(n^3) hence we can neglect the additional time taken by the initialization, but this initialization will surely hurt the top-down analysis for the better or the worse. For our case, the caveat seems to worsen the performance because even with a block-size of 128 which is computationally the same as the original matmul functional implementation, the performance of the matmul_blocking function is much worse than the original implementation. Even in terms of time, the original implementation took 0.656898 seconds to run 100 iterations while the block-size of 128 took 0.728936 seconds to run.

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





Similar to the previous two cases, we realize that increasing the block-size in the algorithm increases the CPU performance but given that the initial matrix itself is so small, all three matrices (both inputs and the output) only need 6.4kb of space and can hence simultaneously be accommodated in the 64kb L1 cache. Running the original algorithm for 10,000 iterations took 0.741843 seconds to run while the best run-time for the optimized algorithm was 0.860491 seconds for the block-size of 32.

Conclusion:

We can conclude that blocking improves the CPU performance for matrix multiplication significantly for large enough matrices. For smaller matrices, the L1 cache size is usually enough to store all matrices simultaneously and hence blocking only worsens the matrix multiplication computations. For mid-size matrices, it cannot be confirmed which algorithm is better because running it only once does not give an accurate CPU performance analysis due to the overhead of function calls and malloc while iterating the algorithm 100 times introduces the overhead of output matrix initialization.