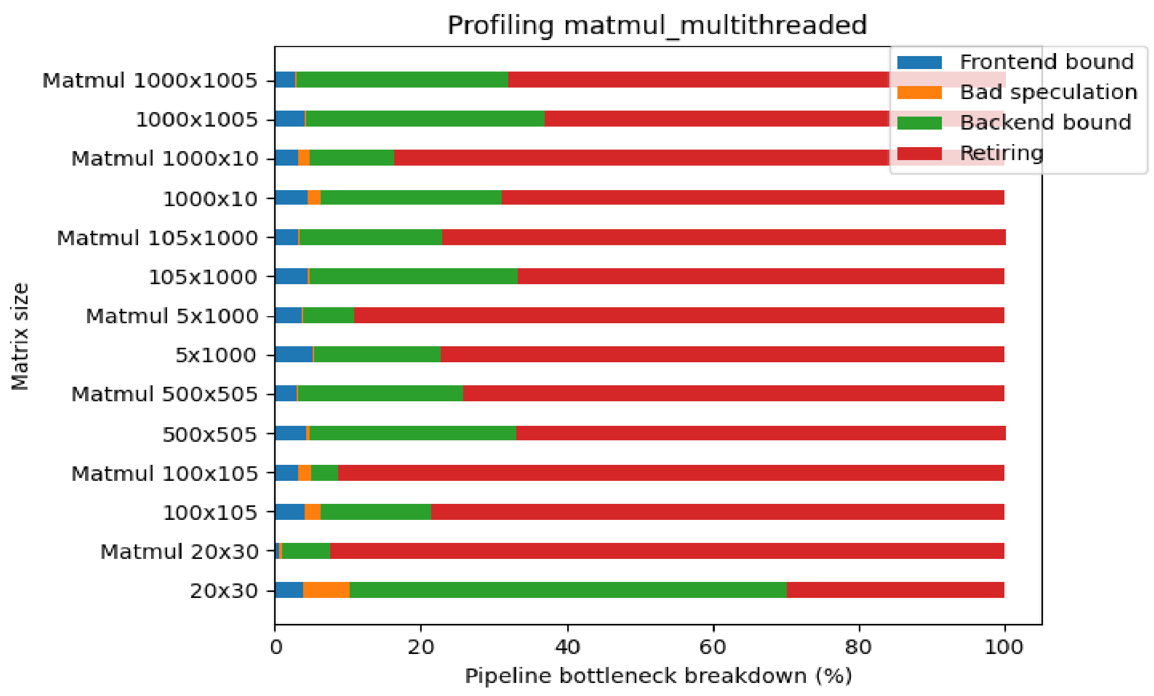
matmul_multithreaded: The *matmul_multithreaded* function in the *matrix_ops.c* file uses a struct *thread_data* defined in *matrix_ops.h* file and an auxiliary function *multiply*.

Functional logic: The main idea of the function is to parallelize the computation of the result matrix by using different CPU cores to compute different parts of the result. The insight is that the computation of every element of the output matrix is independent of every other element.

Algorithmic logic: The algorithm for this function is quite similar in comparison to the initial *matmul* algorithm. In fact, the function uses the exact same algorithm with the difference that the actual computation is distributed across threads as the input matrix rows are divided almost equally across them (the last thread has any remainder rows that are modded out).

In simpler terms, the main function runs a for loop that spawns a new thread where each thread is responsible for calculating the results for the same number of rows (except the last thread). Within this loop, the thread is assigned a struct *thread_data* that stores all useful information we need to calculate the right results for the data of that thread. Specifically, we store the pointers to all the matrices, the dimensions of the matrices required and the start and end rows which are being calculated in that particular thread. Then we pass the *multiply* auxiliary function to each thread which has the same algorithm as the original *matmul* and the core calculation of the algorithm is repeated several times. A key distinction in the algorithm with the original *matmul* algorithm is that a temporary variable is used to sum the values instead of a global call to the output matrix which ensures lesser cache evictions within the loop because instead of loading an entire row only a single value would be requested from the cache in that loop.

Profiling: For profiling, I ran the *matmul* and the *matmul_multithreading* functions on matrices of the sizes (1000 x 1005, 1005 x 1000), (1000x10, 10x1000), (105x1000, 1000x105), (5x1000, 1000x5), (500x505, 505x500), (100 x 105, 105 x 100), and (20 x 30, 30 x 20). The multithreaded implementation was run on eight threads and the relevant cores were identified using the *squeue* and *scontrol* commands to target the correct ones for profiling.

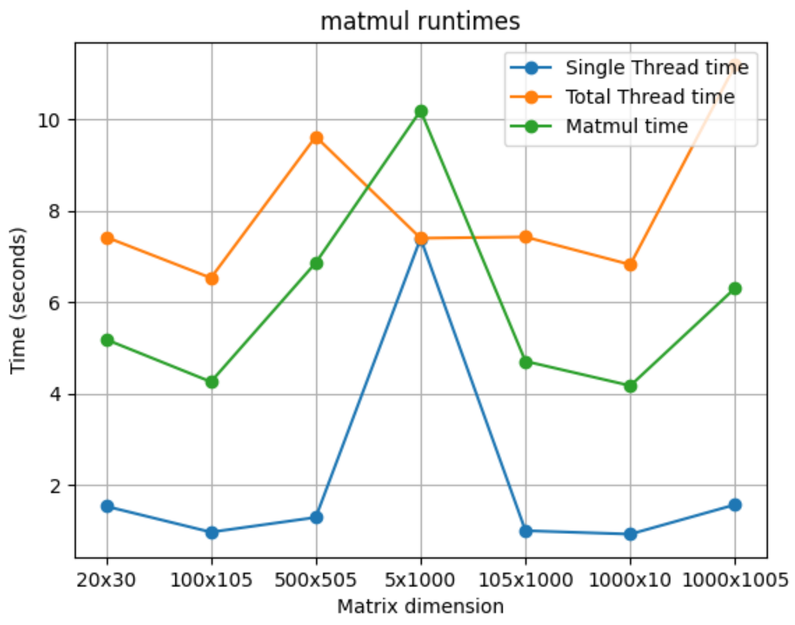


From the profiling graph, it is evident that the *matmul* function is much more efficient in the use of CPU resources compared to the multithreaded implementation. One common pattern across the profiling data is the larger percentage of inefficiencies in the front end and the backend bound. The reason for this can be attributed to the memory latency in the multithreaded program where the CPU cores need to maintain cache coherency by synchronizing cache data. Furthermore, depending on the architecture, it is possible that the cores may share certain resources such as memory bandwidth which can lead to contention among the threads for larger data structures. This is evident from the horizontal graphs where the inefficiencies in the backend and front-end bound increase for larger matrices. From the profiling data, we can observe that computing the result for 100x105 matrix is much more efficient than that for 5x1000 matrix even though 100x105 has many more data elements, which is perhaps because in the 5x1000 matrix there is a requirement to load a longer contiguous array of data in the cache than is needed for the 100x105 matrix.

There is one exception in the data which is for the matrix size 20x30 which has a huge backend bound for the multithreaded implementation which can partially be attributed to the relatively large bad speculation. Still, it seems that the backend bound for this case might be because of resource contention to update the values in the output matrix. Essentially, because the size of the matrix is so small and each thread only has 2 rows and 30 operations per row, the computation and updation of values are extremely frequent and result in performance inefficiencies as the CPU works to ensure cache coherence across the cores. In other words, using 8 threads for this small of a matrix is like an overkill.

Runtime discussion:

While calculating the runtime for the algorithm, it was observed that the *clock()* function calculates the combined runtime for all spawned threads together. So, to calculate the runtime used by one thread (the one on which the main program is running) another function, *gettimeofday*(), is used.



In the runtime graph, the **single thread time** is the actual time taken by the algorithm to run on the main thread in which the program is in execution, the **total thread time** is the time taken by all threads combined to run the algorithm and calculate the result, and the **matmul time** is the time taken by the matmul function to execute the algorithm.

From the graph, there are two noticeable takeaways, the first is that the multithreaded time is almost eight times less than the total thread time for running the algorithm (as it should be). There is one exception to this case for the matrix of size 5x1000 which is due to the nature of the algorithm. The algorithm divides the number of rows in the first matrix with the number of threads being used which in this case, is zero. Hence, the first seven threads essentially do not run any calculations and the result is calculated entirely in the last thread. The actual total thread run time is slightly higher than the multithreaded time which confirms that the other seven threads are being spawned (thus the extra time) but exited almost immediately.

The second observation is that the total time taken for the *multithreaded* function is more than the *matmul* function which is obvious because the multithreaded function essentially computes the same result using the same algorithm but with much larger overhead because of managing the threads.

There is one exception in the data and that is for the matrix size of 5x1000. It is extremely counter-intuitive that the single-threaded function is slower than the multithreaded implementation. On deeper analysis, it was discovered that the difference was due to the implementation of the algorithms in the two functions. In the original function, as mentioned earlier, the global result matrix is fetched for every single loop iteration (inside the innermost loop) while it is masked by a temporary variable in the multithreaded function requiring significantly fewer global calls to the output matrix. This single code change results in a 40% jump in the runtime of the algorithm and serves as an excellent example for cache hits, misses, and eviction in action.

Conclusion:

We can conclude from the analysis, that multithreading significantly improves the runtime of the matmul function. Still, from the profiling data we do understand that the multithreaded version is a little more inefficient than the single-threaded function because of the need for cache coherence across multiple cores and hardware architecture limitations resulting in reduced memory bandwidth. We also observe that for edge cases (where the matrix size is relatively small) there is a steep spike in performance inefficiencies. Finally, we observed how reducing the number of global calls for the output matrix (in the innermost loop of the main algorithm) can significantly improve the runtime (by almost 40%).