Erick Sun

CSEN 145

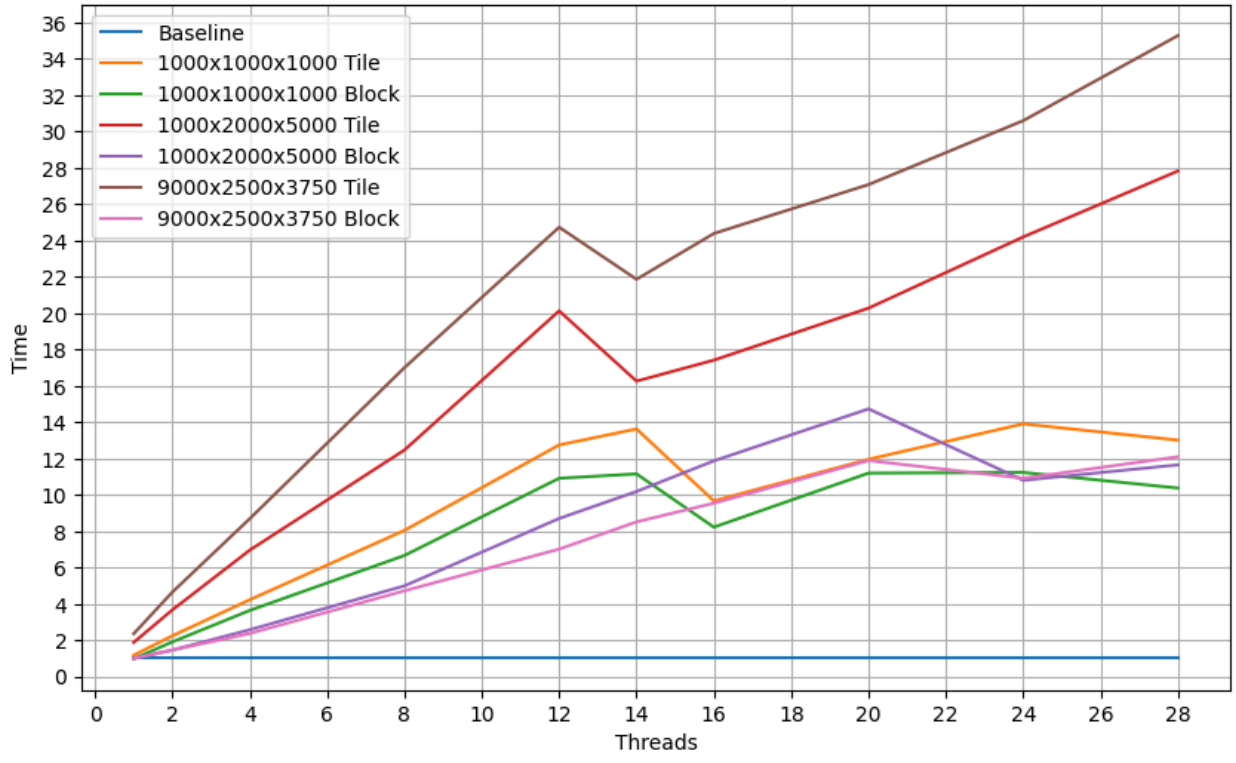
10/07/2024

Tiling vs Non-Tiling Parallelization of Dense Matrix Multiplication (OpenMP)

I. Parallelization Approach

The approach used within my program for parallelization of tiling based matrix multiplication hinges on OpenMP's task directive. The approach used was meant to distribute the smaller, more manageable tiles as tasks to the remaining threads. The code consists of 6 total *for* loops. They can be split into the first three and last three loops. The first three loops are used to loop through the tiles of both matrices, while the second set of loops is meant to act within each tile and do the necessary integer calculations. First, the *single* directive is used to loop through all combinations of rows in the first matrix and columns in the second matrix (the first two loops). It is worth noting that the speedup achieved by tiling comes at the cost of greater writes, as each position in the result matrix is written to multiple times. As a result, this single thread only loops through the first two loops in an effort to prevent collision in the remaining threads executing tasks. What this effectively does is make each task a single tile in the result matrix, which is made up of multiple tiles from the source matrices. By taking this output-based approach, we can guarantee that multiple threads will not be writing to the same location at the same time.

II. Speed Up



Above is a table for the speed up of both the parallel block version and parallel tile version against the serial block version of dense matrix multiplication. From this table, we can see a general upwards trend in speed up as the number of threads increases, excluding the serial baseline. In addition, the parallel tile version of the algorithm consistently outperformed the parallel block version. We can conclude that the parallel tile version does indeed produce more efficient code executions versus the traditional parallel block version. This can be attributed to the design of the tiles. By tiling, we are able to do more operations with more cache hits, as a tile row of the matrix can likely fit in cache much more easily than a full row in the matrix. This means we spend less time waiting for memory reads and more time doing computations. However, as referenced before, the trade off for doing this method means that for each tile in the source matrix, we are only able to fill a fraction of the results in a tile of the result matrix. This means we must perform more writes into the result matrix.

III. Analysis

As mentioned above, generally, the greater number of threads there are, the greater the speed up. That is true until an inexplicable drop at fourteen threads that seems to persist. This could be a result of many things. Due to the random nature of the source matrices, there may have been peculiarities in a given test run. Combining that fact with the diminishing returns given by increasing the number of threads, it is possible that factors influencing the execution happened to line up in favor for fourteen threads.

Through this program, it is difficult to say with certainty what exactly is the relationship between problem size and time taken, but it is clear that there is a positive correlation between the two. When comparing the runtime of the 1000x1000x1000 test versus the 1000x2000x5000 tests, there seems to be a slowdown in runtime by a factor of ten. However, this was only found in the test with a single thread. This is what we would expect since the second test has ten times as many computations to perform. However, as the number of threads increases, this relationship quickly falls apart. However, this is also to be expected. Within the graph above, as the problem size increases, so does the speed up given more than one thread. This means that the factor between the two runtimes will decrease as the problem with more computations experiences greater speed up.

As for scaling, it seems that the program already encounters a decreased rate of speed up after a certain point. As we can see from thread amounts past sixteen threads, the growth seems to stagnate ever so slightly as it approaches twenty-four threads. The improvement between twenty-four threads and twenty-eight threads begins to fall for the 1000x1000 case. This trend is likely to continue as the number of threads continues. However, it seems like the plateauing effect is less visible for larger problem sizes on a smaller number of threads. Perhaps we would need to go much further past twenty-eight threads to start to see visible differences.
Finally, we can say with confidence that the obtuse size of matrices does not seem to affect the performance of the algorithm. As seen in the figure, we achieved significant speed up regardless of the shape. That can be seen in the steep upwards angle representing speed up. In fact, the rate of speed up was significantly greater for matrices with obtuse sizes. However that may be attributed more so to the problem size instead of the problem shape. The result times can help support this statement. In the 1000x1000x1000 trials, one may expect approximately a fourth of the time when compared to the 1000x2000x5000 trials. However, in the case of sixteen cores, the 1000x2000x5000 trials were faster by almost 0.01 seconds. This shows that the shape did not significantly influence the speed of the algorithm.

IV. Remarks

Overall we can conclude that tiling, and especially tiling with parallelization contributed greatly to the speed up of dense matrix multiplication. Due to the nature of the problem, dense matrix multiplication is very well suited for parallelization. With this algorithm, we achieved great deals of speed up in various shapes of matrices.