Exercise 08

1 Complete your Matrix-Matrix Multiplication

Complete your matrix-matrix multiplication implementation from the last exercise such that **Alignment, Strided-Access** und **Pointer-Aliasing** are dealt with appropriately. Extend your implementation such that it makes efficient use of the blocking-technique shown in the lecture. Measure and document the speed-up after each change you implemented. Which techniques/flags/compilers had the most impact in terms of performance?
Scale your implementation within the context of a multi-core environment using OpenMP. Insert the necessary #pragma annotations at the right places and add the OpenMP flags to the compiler/linker using CMake.
Measure the observed speed-up when using one core up until the maximum number of cores on your system, for different sizes of matrices. Use the OMP_NUM_THREADS environment variable to vary the number of threads used. Draw a nice graph of your measurements. :)

2 Parallel Merge-Sort

Implement the Merge-Sort algorithm. Start with a non-parallel version and add parallelization to your program using OpenMP later on. Which OpenMP constructs are suitable candidates to speed-up your program? Are there any runtime-features that have to be activated?
Sort a randomly initialized integer-array (use std::iota and std::random_shuffle for initialization) of length N and vary N. Compare your parallel version with your non-parallel version: at what size N is using parallelization worth the effort? How could your implementation make use of this observation? How does your implementation compare to std::sort in terms of performance? What algorithm is std::sort (likely) using?