Exercise 3

3.1 Reading

The article „The Future of Microprocessors" by Shekhar Borkar and Andrew Chien, discusses the shift away from relying only on transistor speed for microprocessor performance scaling. The Authors argue, that to keep up with Moore's law, significant architectural changes are necessary, with a particular emphasis on prioritizing energy efficiency in chip design. By simply adding as many compute cores at maximum frequency as the transistor-integration capacity allows for the power consumption of the microprocessor would quickly grow into the hundreds of Watts. Instead, the authors suggest, that cache sizes will increase, since they require less energy compared to logic and that microprocessors will instead feature smaller fine grained compute cores capable of operating at different frequencies or accelerators for special workloads.
Some predictions from the paper have become reality. For example, frequency adjustment for certain cores is common even in consumer grade CPUs (Intel Turbo Boost, AMD Turbo Core). However, cache sizes did not increase as rapidly as predicted. Most mainstream CPUs usually have a cache of around 30 MB, some reach 64 MB and therefore differ from the originally expected development. Notable exceptions, such as the IBM z15, use exotic architectures with very large off-chip caches, which are realised as eDRAMs and reach up to 960 MB in the L4 cache. The integration of smaller computing cores and accelerators in CPUs can be observed above all in exotic architectures, while x64-based systems often rely on external multi-core accelerators such as GPUs or the Xeon Phi.
Nevertheless, the core premise of the paper is correct, as energy efficiency has become a very important metric not only in the HPC sector, but also in the consumer sector, especially for mobile devices. We therefore accept the paper.

3.2 Matrix multiply – sequential version

The octance cluster has a AMD EPYC 7351P CPU, which according to Mandelbrot (https://browser.geekbench.com/v3/cpu/8555579) has a Single-core GFLOP/s performance of 3,35.

The naïve implementation uses three nested for loops to perform the matrix multiply as seen below:

```
void mmul_unoptimized(const double *A, const double *B, double *C, size_t dim) {
    /*
     * Naive matrix multiply for square matrices (NxN)
     * A, B: input matrices
     * C: output matrix (has to be initialized to zero)
     * dim: number of rows (or columns in this case) of the matrix
     */
    size_t i;
```

```
    size_t j;
    size_t k;
    for (i = 0; i < dim; ++i) {
        for (j = 0; j < dim; ++j) {
            for (k = 0; k < dim; ++k) {
                C[i * dim + j] += A[i * dim + k] * B[k * dim + j];
            }
        }
    }
}
```

It achieves the following results:
Running on one node
==========
Unoptimized :
Time 237472462879 ns
=> 0.07 GFLOP/s
==========

There are two main problems with the implementation:

- Hardware utilization: The processor used has 16 CPU Cores. The naïve code, however, only uses one thread and core.
- Cache utilization: Cache utilization in the naive version is suboptimal due to a strided access pattern, causing cache misses for accesses to matrix B.

To counteract these effects, the main loops were divided into smaller blocks to achieve better data localization as described in the lecture slides for Tiling/Blocking. Instead of running through the entire matrix, we proceed block by block in each dimension. This allows us to achieve both temporal and data locality, as more values are reused (still in the cache) and the step is limited to cacheable blocks. The innermost loops are parallelized with OpenMP as seen below:

```
void mmul_optimized(const double *A, const double *B, double *C, size_t dim) {
    /*
     * Optimized version of matrix multiply for square matrices (NxN)
     * Memory bandwidth is increased by better cache utilization via
     * block-wise access to elements instead of linear traversal
     * A, B : input matrices
     * C: output matrix (has to be initialized to zero)
     * dim: number of rows (or columns in this case) of the matrix
     */
    size_t i;
    size_t j;
```

```
    size_t k;

    size_t ii;
    size_t jj;
    size_t kk;

    #pragma omp parallel for collapse(3) private(i, j, k)
    for (ii = 0; ii < dim; ii += BLOCK_SIZE) {
      for (jj = 0; jj < dim; jj += BLOCK_SIZE) {
        for (kk = 0; kk < dim; kk += BLOCK_SIZE) {
          for (i = ii; i < dim && i < ii + BLOCK_SIZE; ++i) {
            for (j = jj; j < dim && j < jj + BLOCK_SIZE; ++j) {
              for (k = kk; k < dim && k < kk + BLOCK_SIZE; ++k) {
                C[i * dim + j] += A[i * dim + k] * B[k * dim + j];
              }
            }
          }
        }
      }
    }
}
```

It achieves the following results:

Running on one node

==========

Optimized :

Time 381646442581 ns

=> 0.05 GFLOP/s

==========

We were not able to actually improve the GFLOP/s value even when using Tiling/Blocking.

Gruppe 3: Davit Melkonyan, Florian Schrittwieser, Simon Pavicic

3.6 Willingness to present

Willing to present all exercises.