Sheet 1: Linear Algebra

Released: 04/21/23; Submit until: 05/05/23 (**20 Points**)

This sheet is about a very small introduction into elementary concepts of C++ and how to use them to implement fast linear algebra operations using Intel's Math Kernel Library (MKL). In order to work through the problems, checkout the `git` repository nqp-exercises provided on the lecture's homepage and work with the code templates. While the header file `la_wrapper.h`, which contains an interface to some linear algebra methods of the MKL, is complete, in each code template there are files which are erroneous and need to be corrected. In a Jupyter-Hub session, use the provided *Makefile* as well as a proper make.inc to compile the corrected code templates.

Problem 1   Implementing a matrix class (5 Points)

Checkout the code template `blas_wrapper`, which contains an elementary implementation of a wrapper of the widely used linear algebra MKL.

(1.a) **(3P)** Correct the code template `blas_wrapper` such that you can compile it using `make build`. As a consistency check, you can use the default implementation of the `main`-routine and check the output when executing the compiled binary.

(1.b) **(2P)** You now have a rudimentary matrix class that implements fast linear algebra operations using Intel's MKL. Use this class and write a `main`-routine which performs a scaling analysis of the runtime needed to copy matrices with dimension $m \times m$ for $m \in \mathbb{N}$ and plot the runtime as a function of m. Extract the exponent α determining the dominating scaling of the runtime $t \sim m^{\alpha}$.

Problem 2   Implementing fast matrix contractions (10 Points)

For this exercise, checkout the code template `gemm_wrapper`, which provides an elementary implementation of the fast `xgemm` operations. Here, the `x` denotes the fundamental data type, i.e., `single`, `double`, `complex single`, or `complex double`. The acronym `gemm` abbreviates GEneral Matrix-Matrix multiplications and this convention carries over for other provided operations, for instance, GEneral Matrix-Vector multiplications (checkout Intel's developer guide for a rather complete documentation about the blas/lapack interface).

(2.a) **(5P)** Proceed as in problem (1) and correct the code template. Note how `la_operations.h` now also contains a multiplication operations, which is compiled into an object file `la_objects.o` that now implements a general matrix-matrix multiplication. Write a test case, which tests the implemented `xgemm`-functionality and returns success (return code 0) or failure (return code 1) on exit, depending on whether a successful matrix-matrix multiplication has been performed. Why is such a test case useful?

(2.b) **(5P)** Write a trivial version of a matrix-matrix multiplication by implementing the calculations of the elements

$$C_{ij} = \sum_k A_{ik} B_{kj} \tag{1}$$

of the result of a matrix-matrix product $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, explicitly. Here, we consider $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{V}_{\mathbb{R}}^{m \times m}$ for some $m \in \mathbb{N}$. Perform a runtime analysis comparing your naive implementation with `xgemm` calls, investigate the dependency of the runtime on the matrix dimension m and extract the exponent β determining the dominating scaling of the runtime $t \sim m^{\beta}$. Interpret you result.

Problem 3 Syntactic sugar for matrix contractions **(5 Points)**

For this exercise, checkout the code template `expr_templates`, which provides an elementary implementation of expression templates to overload the multiplication operator. In problem (2) we introduced an operator overload to the multiplication operator `*`, allowing for expression such as `C=A*B` in C++. However, that implementation also required an intermediate copy operation, which is necessary because `*` is a binary operator and the result of the matrix-matrix multiplication has to get through a temporary return value. This unfortunate fact can be avoided by delayed evaluation, which in C++ can be implemented using *expression templates*.

(3.a) **(3P)** Proceed as in problem (1) and (2) and correct the code template. In particular note how the binary matrix-matrix multiplication operator `*` is mapped to the unary assignment operator =, which assigns the result of the operation to an instance of `LAMatrix` without additional copy-operations.

(3.b) **(2P)** Perform a runtime analysis and extract the speed-up as a function of the matrix-dimension m obtained, using the `xgemm`-impementations of the `*`-operator from problem (2) and (3).