Advanced Programming

Matrix Operations

In this project we will

- design a class to model square matrices using row-major ordering to store the matrix elements;
- implement methods to perform the typical arithmetic operations on matrices;
- optimise the implementation of the methods.

Matrices and row-major ordering of elements

In many programming languages, the elements of higher-dimensional data structures (e.g. matrices, or more generally *2-dimensional arrays*) are stored in row- or column-major order. This means, the elements are simply stored line-by-line (or column-by-column) in subsequent memory cells.

In this project, you will design and implement a class `Matrix` that stores a *square* matrix as a *one-dimensional* array of *real* numbers. If you choose row-major numbering, an element a_{ij} will be stored in the $(in + j)$-th element of the one-dimensional array, where n is assumed to be the dimension of the matrix.

Implement two methods, `getElem` and `setElem`, which can be used to read or modify a matrix element. The methods request the usual indices i and j as parameters, and provide a method to access the elements of a matrix without having to know about its internal implementation.

After that, implement methods to add and multiply two matrices, and to transpose a single matrix. Write two different implementations for each method: one implementation which uses the methods `getElem` and `setElem` to access the matrix elements; the other which accesses the internal array elements directly. Write a constructor which allocates a matrix of dimension n and fills it with random elements. Test your routines with matrices of different sizes and record the time your routines take. What do you notice?

Cache efficient programming

Both implementations of the matrix multiplication will most probably consist of three nested loops, which result from the definition of the matrix multiplication. Each loop will have n iterations (n being the size of the matrix). In the most straightforward implementations, the inner loop accesses n elements of each of the two multiplied matrices during one iteration. As we will see, such an approach is not very advantageous, considering the computational performance of your application.

On the majority of today's computers, several levels of *cache* memory can be found which are used to buffer data between the CPU registers and the main memory. Caches provide very fast data access and can improve the performance of an application significantly if it is utilised in the right way, namely if data can be re-used frequently.

Blocked algorithms provide a possibility to speed up a matrix multiplication by making use of the available caches. Re-write the implementation of your multiplication method by making use of techniques as *loop tiling* and *loop unrolling*. Again, generate benchmark runs of your multiplication routine with different matrix sizes. Change its block size and try to optimize the runtime of your code!

Additional topics

The class `Matrix` can be further extended by dealing with the following problems and topics:

- Extend your matrix such that $m \times n$ matrices are implemented (usually $m \neq n$).
- Before doing computations, check whether the matrix dimensions are compatible. This can be done in an elegant way using exceptions. Read up on how to throw arithmetic exceptions in Java.