High-Performance Computing
Prof. Dr. Ivan Kisel
Robin Lakos, Akhil Mithran, Oddharak Tyagi

High-Performance Computer Architectures
Practical Course
SoSe 2025

Issued: 25.05.2025 / Due: 04.06.2025

GOETHE UNIVERSITÄT FRANKFURT AM MAIN
Institute for Computer Science

Exercise Sheet 5

Vector Class (`Vc`)

Vector classes (`Vc`) is a free software library to ease explicit vectorization of C++ code. It has an intuitive interface and provides portability between different compilers and compiler versions as well as portability between different vector instruction sets. Thus, an application written with Vc can be compiled for AVX, SSE, XeonPhi and other SIMD instructions. One of the most useful functionalities of Vc are *masks* and *random memory access*.

The *mask* functionality allows conditioning calculations. The *random memory access* functionality is provided by the *gather* and *scatter* functions. They allow to provide two parameters, one with the source / destination of the data (e.g. an array) and a vector with indices of the source / destination where the data is written to or read from.

In this exercise, we apply functionalities of `Vc` to small examples. Since `Vc` will be integrated in to the C++ standard (expected in C++26), we will use `std::experimental` to apply its functionalities in most exercises. However, the process to integrate a library into the C++ standard takes some time and, therefore, not all functionalities (including random memory access) are available there yet. For these, we will switch to the development version of Vc.

Submission: Please upload a zipped directory including your source code files along with a PDF, explaining your approach and analyzing your results.

For every exercise, we provide a Makefile that can be used to easily compile and run the code with SSE or AVX support. You can compile and run your code by using `make run-sse` or `make run-avx`. For cleanup, you can use `make clean`. If you compile the code without `make`, please note the compilation flags, as they are crucial for correct results.

Disclaimer: We do not know if all RBI computers have a processor that supports AVX instructions. If AVX does not work, please use SSE instead or try another computer.

Exercise 5.1: Element-Wise Unary Operations

In `matrix.cpp`, the initial code implements the scalar version as well as a vectorized version using the SIMD header `FVec` from previous exercises. Use it as a reference for your implementation and implement the SIMD version by using `std::experimental`.

Discuss the results in your PDF.

Exercise 5.2: AOS, SOA, and AOSOA

The data layout strategies Array of Structs (AOS), Struct of Arrays (SOA), and Array of Structs of Arrays

(AOSOA) can significantly impact performance when using SIMD instructions. While AOS maintains the logical grouping of object data, it prevents efficient vectorization due to its memory layout. SOA, by storing each data member in separate contiguous arrays, often enables much faster SIMD processing, though it requires specifically organized data. AOSOA seeks a compromise, grouping small blocks of SOA inside an array to balance structure and performance.

Open `quadratic_equation.cpp` and implement all three layouts to evaluate their effect on runtime.

Discuss the results in your PDF.

Exercise 5.3: Newton's Method

Newton's method is a root-finding algorithm that produces, starting from an initial guess x_0, successively better approximations x_k to the roots of a real-valued function f. The next estimate x_{k+1} is produced from the current estimate x_k by,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{1}$$

where f' is the derivative of f. The above steps are repeated as long as $\frac{|x_{k+1}-x_k|}{x_{k+1}} > p$, where p is the required precision.

A scalar version of the algorithm is already given in `newton.cpp`.

Vectorize this algorithm using `std::experimental` to solve a group of equations in parallel. Parallel conditions might be tricky. As a first step, vectorize the algorithm by using a fixed number of `n = 1000` iterations instead of the precision-based approach. Since the implemented tests will fail for this approach, compare the results manually and show they are reasonably close to the scalar solution.

Discuss the results in your PDF.

Afterwards, explore how masks and parallel conditional statements can be used to replicate the precision-based stopping criterion, similar to the scalar version, but by using `std::experimental` functionality. Do not use scalar loops to iterate over SIMD vector elements.

The final vectorized implementation must produce exactly[1] the same results as the scalar version.

Again, discuss the results in your PDF.

Hint: Think about if the algorithm reaches the required precision for all functions in the same number of iterations. Remember, the goal is to stop for each value that already reached the precision like it is done in the scalar version, otherwise results will differ.

Exercise 5.4: Random Access

In its current state, the parts of Vc that are already in `std::experimental` are not complete. As the name suggests, these functions are still in development. Although many functionalities are already implemented in the development version, the process to make them available in `std::experimental` is long.

The `random_access.cpp` exercise requires to use different forms of the `gather` and `scatter` functions. These functionalities are not implemented in `std::experimental` yet, but the Vc library development version is available via GitHub as well.

We prepared a bash script that will install the Vc library for on the machines on-site. If not already done, use SSH and log into the computers on-site and run the following command in the terminal from inside your exercise directory where the script `install_vc.sh` is located.

[1]That is, within floating point precision – the results must pass the implemented tests.

```
./install_vc.sh
```

Due to missing administrator privileges for students at the computers on-site, the script will install the Vc library development version in your home directory ~/Vc. If you did not change anything in the directory structure, the `random_access.cpp` should run out of the box. Otherwise, please adjust the paths inside accordingly.

Since we do not install the `Vc` library in a directory where C++ is usually searching for libraries, the installation script also adds global paths to tell C++ where to find `Vc`. If you log out of the computers, you might have to re-set the C++ include paths at the next login. In this case, please run the `export_paths.sh` script. After that, you should be able to compile and run again.

For this exercise, an input array of type `float` is provided and randomly filled. Additionally, an array of random indices is provided. It is required to

1. gather data from the input array to the `float_v tmp` variable according to the index array
2. gather data from the input array to the `float_v tmp2` variable, but only when it satisfies a given condition `i > .5f`
3. scatter the data from the `float_v tmp` variable to the output array, when it satisfies a given condition `i > .5f`

Some additional hints are provided inside the source code file. Also, please have a look in the documentation, listed in the GitHub repository of VcDevel.

Heeeeeeelp!

If your bash scripts do not run out of the box (e.g., "permission denied"), please modify the file permissions to allow execution:

```
chmod +x ./SCRIPTNAME.sh
```

If this does not help, check if you own the file. You can change the ownership to your user by running:

```
chown <your-username>:<your-username> ./SCRIPTNAME.sh
```

If nothing helps, please ask your tutor.