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

High-Performance Computer Architectures
Practical Course
SoSe 2025

Issued: 21.05.2025 / Due: 28.05.2025

Institute for Computer Science

Exercise Sheet 4

Getting Comfortable with SIMD

The following exercises will help you to get more familiar with using SIMD instructions that make use of the CPUs vector units. For convenience, these instructions can be wrapped by C++ class, which overloads standard operators ($+, -, *, >$, etc.) and provides other quality of life features.

A small header file with overloaded instructions is provided via OLAT. It wraps `__m128`[1] data type by an `F32vec4` object, providing `operator[]` for access and operators like `operator+` and `operator<` for computations. Also, a more general short name `fvec` is given to the `F32vec4` type, the vector length is saved to the variable `fvecLen` and the scalar entry type to `fscal`.

Submission: Please upload your C++ CMake project (MLP), your source files (other exercises), including your solutions and meaningful comments as well as a PDF with your results and analysis to OLAT. Please also note the additional material uploaded to OLAT and do not hesitate to ask your tutor if you have any questions.

For this exercise sheet, always use the following compilation flags:

```
<Compiler> -O3 -fno-tree-vectorize -msse4.2 filename.cpp
```

It is recommended to compile the code with `-fno-tree-vectorize` option to prevent auto-vectorization of the scalar code, otherwise comparison of the vectorized and scalar codes will not be direct.

Exercise 4.1: Fast Element-Wise Unary Operations

For `Matrix.cpp`, implement the SIMD-ized version of the element-wise square root operation and think about how to make use of the template. Make sure that you do not copy the values. Instead, use `reinterpret_cast<...>(...)` to reinterpret the memory as objects of the SIMD vector type. Afterwards, compare the speed up of the implementations and discuss it in your PDF (e.g. what is the theoretical expected speedup, what is the real speedup, why are there differences?).

The initial code gives an implemented scalar version you can use for orientation and contains blank space for a vectorized version. Therefore the initial output shows 0 time for vector calculations and an infinite speedup factor, which should be currently ignored, until you implemented your code.

Hint: You can use the `fvec`-header for an easier implementation.

[1] See for example here: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Exercise 4.2: Different Ways of Utilizing SIMD Instructions

Open the `QuadraticEqn.cpp`. This file contains a set of quadratic equations $ax^2 + bx + c$ for which we want to calculate the roots. In this exercise, we will use 4 different methods to achieve this:

1. Write the SIMD code for the root calculation using SIMD intrinsics and *copying* the data to SIMD vectors.
2. Write the SIMD code using SIMD intrinsics and type-casting the data from the scalar arrays to SIMD vectors (use `reinterpret_cast` for this). Compare the time with the previous task.
3. Write the SIMD code using the `fvec`-header and *copying* the data to SIMD vectors. Compare the time with previous tasks.
4. Write the SIMD code using the `fvec`-header and type-casting the data from the scalar arrays to SIMD vectors (use `reinterpret_cast` for this). Compare the time with previous tasks.

Compare the speedup factors of each of these implementations in your PDF and discuss what speedup factors you expected and achieved.

Now, set `NVectors = 10000000`. Compare times and speed-up factors with the previous results and discuss all of that in your PDF. In particular, explain why there are such large differences in speedup factors. To do this, consider the total size of the data before and after the modifications and how this relates to typical cache sizes.

Exercise 4.3: Optimizing Checksum Calculation with Smallest Data Type and a Trick

Open the `CheckSum.cpp` file. Here, the checksum is defined as an XOR-sum over all bytes in an array. This exercise aims to improve the efficiency of the checksum calculation by leveraging different data types. Additionally, ensure the use of `NIter` in the outer loop to minimize side effects.

i) Integer-Implementation of the CheckSum Calculation

Implement the integer version of the checksum calculation. Since the checksum operation is bit-wise, you can ignore the byte structure of these data types and consider how to fit the bits of `char` type elements into a larger data type, like `int`, to process multiple elements simultaneously.

Think about the number of iterations required within the `Sum(...)` template function when using `int`. Use `reinterpret_cast(...)` to reinterpret the memory accordingly.

Finally, consider any necessary post-processing to achieve the correct final result. Remember, the data type for the result of `Sum(...)` is now `int`, but the required checksum result, `sumI`, should be of type `unsigned char`.

ii) FVec-Implementation of the CheckSum Calculation

Vectorize the checksum calculation for the data array. Determine the appropriate number of iterations for the `Sum(...)` template function. Specifically, consider how many `char` type objects can fit into a `float` value and how many `float` values fit into the `fvec` vector.

Then, apply `reinterpret_cast(...)` to `str` and use the `Sum(...)` function template.

Finally, consider any required post-processing to achieve the final result. Again, ensure the result's data type is correctly adjusted to match the required checksum type.

Hint: Remember that pointers and arrays are closely related in C++. This relationship might be useful here when reinterpreting the memory.

Exercise 4.4 Getting Closer to Real Implementations of Neural Networks

In the last exercise, we have seen that there are many calculations and it requires a lot of time to train the network. At least to some extend, we would like to improve our previous implementation.

For the MLP program from the previous exercise, change the **`AffineTransform`** function in `Utils.cpp` to the following:

```
void AffineTransform(const std::vector< std::vector<float> >& matrixA,
                     std::vector<float>& vectorX,
                     std::vector<float>& vectorB,
                     std::vector<float>& result)
{
  // MatVecMul(matrixA, vectorX, result);  // Scalar version
  MatVecMulSimd(matrixA, vectorX, result); // SIMD SSE version
  VecAdd(result, vectorB, result);
}
```

and complete the `MatVecMulSIMD()` (you can declare and define it in `Utils.h` and `Utils.cpp` respectively) and, thus, create an implementation that makes use of vectorized calculations. After that, change between the scalar and SIMD implementation in the function `AffineTransform()` in `Utils.cpp` and compare the speed up.