Hardware für Eingebettete Systeme

Lab 3: Basics of AVX intrinsics

Prof. Dr. Mario Porrmann
M.Sc. Marc Rothmann

1 Basic AVX Intrinsics

In the last lab, you have used GCC vector datatypes to access SIMD functionality. Some SIMD instructions available in the AVX extension can not be accessed that way. The only way to make us of them is to use SIMD intrinsics. Intrinsics are C functions that map directly to one assembler instruction that will be inserted by the compiler.

Most AVX intrinsic names use the following notational convention:

`_mm256_<intrinop>_<suffix>`

The suffix consists of 2 parts: The first letter determines whether the operation is packed (p) or scalar (s) and the rest of the suffix denote the datatype (In our case, we will use single-precision floats (s)). Two quick examples:

- `_mm256_add_ps` performs packed addition on single-precision floats.
- `_mm256_mul_sd` performs scalar (sequential) multiplication on double-precision floats.

Hint: The suffix you need will almost always be "ps" (packed, single-precision float).
The arguments and return values of intrinsics are usually either of type `__m256` oder `__m256i`.

To make the intrinsics available, you need to include the `x86intrin.h` Header. Furthermore, you need to compile with the `-mavx` flag, like this:

```
gcc -mavx -O3 test.c
```

Write a simple test program that performs the following operations on two vectors:

a) Addition

b) Scalar Product

Hint: Use `_mm256_hadd_ps` to compute the sum efficiently.

c) Square Root (of all elements of one of the vectors)

You can use the Intel Intrinsics Guide to look up available functions and how they are used:
`https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=3676&techs=AVX`
You can find more information here: `https://www.cs.uaf.edu/courses/cs441/notes/avx/`

2 Matrix Computations

The matrix.c file in this lab is very similar to the previous one.

a) Implement the matrixAddSIMD function using intrinsics.

b) Implement the matrixMultSIMD function using intrinsics.

Some hints:

You can use your matrix multiplication from last lab as a starting point.
You will likely need the following functions for your implementation of `matrixMult`:

- `_mm256_load_ps`
- `_mm256_store_ps`
- `_mm256_storeu_ps`
- `_mm256_maskload_ps`
- `_mm256_maskstore_ps`
- `_mm256_cmp_ps`
- `_mm256_set1_ps`

The maskload/maskstore functions take a mask as an argument, which is of type `__m256i`. The compare instrinsic returns as mask of type `__m256`. You can just cast between these types.