Hardware für Eingebettete Systeme

Lab 1: SIMD Auto-Vectorization

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

1 SIMD Setup

Single Instruction Multiple Data (SIMD) operations allow the programmer to apply an instruction to multiple values at the same time. Multiple vector extensions are part of the x86 instruction set architecture. In this course we will focus on the AVX extension, but the user interfaces of the other SIMD extensions are very similar.

First, we will check whether your CPU and installed compiler support the AVX extension.
Hint: If any of the following exercises fails, ask the teaching staff for help.

a) Execute the following command.

```
cat /proc/cpuinfo | grep flags | sort | uniq -c
```

You will see a list of features supported by your CPU. Make sure *avx* is one of them.

b) Then, execute the next command:

```
gcc -march=native -dM -E - < /dev/null | egrep "SSE|AVX" | sort
```

You will see a list of SIMD related constants defined by your compiler. Verify that `__AVX__` is defined as 1.

c) Compile the test program given to you with the exercise material:

```
gcc -mavx simd_test.c -o simd_test
```

Execute the compiled program. What does the program do?

2 OpenMP Setup

OpenMP (Open Multi-Processing) is a compiler feature for C, C++ and fortran.
GCC supports OpenMP starting from version 4.2.0. If your gcc is version 4.2.0 or newer, you probably don't need to install anything else. In this exercise, we will just check, whether or not your compiler supports OpenMP and compile a short test program.

a) Check whether your gcc supports OpenMP with the following command:

```
echo | cpp -fopenmp -dM | grep -i open
```

If your compiler supports OpenMP, you should see a line similar to this:

```
#define _OPENMP 201511
```

b) Compile mptest.c with the following command:

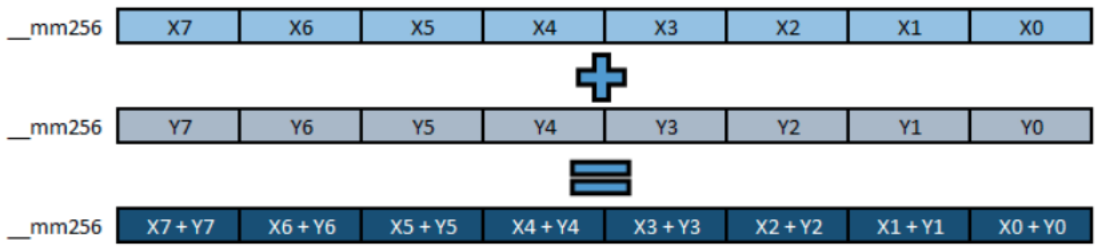
```
gcc -fopenmp mptest.c -o mptest
```

Execute the compiled program. What does the program do? How many threads are created?

We will continue with OpenMP in a few weeks.

3 SIMD Auto-Vectorization

SIMD instructions are a way to make use of data parallelism in modern processors. Instead of applying an operation to a single operand, it can be applied to multiple operands of the same type. The low-level datatypes used by the AVX extension are called `__mm256` (for floats), `_mm256d` (for doubles) and `__mm256i` (for integers). The following figure shows how an AVX operation on these datatypes works:



We will start working directly with these datatypes later, when we introduce SIMD intrinsics. Before that, we will explore other options to vectorize a C program.
There are three ways to vectorize a program: auto-vectorization, compiler-specific vector datatypes and SIMD intrinsics. With auto-vectorization, the compiler automatically determines sections of the code that can be vectorized. However, we still need to write some hints into the code, so this can be done successfully and it will still miss vectorization opportunities.
The second option are vector datatypes provided by the compiler. These datatypes give us more control over the vectorization. But the datatypes are compiler-specific and even if we use them, the compiler will not produce some available SIMD instructions.
The last option are SIMD intrinsics. Intrinsics are built-in functions that translate directly into one specific assembler instructions. That way, we get maximum control, but they are also the hardest to use.
In this lab, we will start with auto-vectorization.

a) Look at autovec.c. What does the `test` function do?

b) Compile the program and save the assembler output:

```
gcc -masm=intel -save-temps -O3 autovec.c
```

Inspect the assembler program (autovec.s) produced by the compiler. (It is not necessary to understand all details.) Which of the instructions are SIMD instructions?

As you can see, the compiler does vectorize this test automatically. However, there are two problems: the first problem is that gcc introduces some unnecessary safety checks, checking whether our two input vectors overlap and the second problem is that SIMD instructions work with aligned data access, but since gcc can't assume our data to be aligned, it needs to take care of unaligned data access.
We can give our compiler some hints, so it can produce more optimal code. The `restrict` keyword tells gcc that two vectors don't overlap. And the built-in function `__builtin_assume_aligned` tells gcc that it can assume aligned data. It is used like this:

```
double *x_aligned = __builtin_assume_aligned(x, 16);
```

c) Use the `restrict`-keyword and the `__builtin_assume_aligned` function to optimize the test function in autovec.c

d) Inspect the assembler code produced by gcc and compare it to the unoptimized output from exercise b.

Now we have optimized the test function, but we also have to make sure our inputs are actually aligned and don't overlap. We can do that without loosing any efficiency. If we use 2 different arrays, they won't be overlapping. We can use an attribute to take care of the alignment:

```
double __attribute__((aligned(16))) vec[SIZE];
```

This was a very simple example that could easily be vectorized. But as we have seen, we still need to leave some hints for the compiler, so it can produce optimal code. If the example were more complex, this would be harder and many opportunities for vectorization would be missed. In the next lab, we will look at vector datatypes for increased control over vectorization.