PROGRAMACIÓN PARALELA
SIMD Tutorial

In the accompanying archive, there is a compressed directory called `Lab6`. Once the archive is uncompressed change to that directory. Inside it there is a `Makefile`. Use `make` to compile the programs in that directory.

The tutorial purpose is to explore how gcc generates code regarding auto-vectorisation. Then, we can execute the `gen_asm` Bourne shell script to generate several versions of the stream_triad code, a reduced implementation of the STREAM benchmark (McCalpin, circa 1995), a memory performance test. It simply calculates

$$\vec{C} = \vec{A} + scalar \times \vec{B}$$

to measure the time to read and write a large array. As can be seen, the above triad has three operands with a multiply and an add (two arithmetic operations).

As output we have several assembly files (`.s` extension) from which you can infer the vector extensions used (you can check Henk-Jan Lebbink's list of x86aAssembly instructions here). We can see also that gcc gives us some compiler feedback (`-fopt-info-vec-optimized`): it vectorises both the initialisation loop and the stream triad loop.

We may also use grep to determine which vector extensions are available in our CPU:
`grep <regs> *.s`

Since this visual analysis is somewhat awkward, we can use another approach: using tools that allows us to see what kind of instructions the CPU executes when running a program.

Before running the programs, we will need to install LIKWID, a suite of powerful performance tools and library for the GNU Linux operating system. They use on-die counters and registers, and therefore are less intrusive than using a software-based approach for detecting events and measuring times.

We can display information about the CPU architecture by entering `clear; lscpu`. Using `lscpu | egrep 'mmx|sse|avx'` we can check the CPU's support for vector extension operations. More detailed info about the CPU and on-die memory organisation can be obtained via `clear; likwid-topology`.

The first program you should try is called `stream_triad`. The size of the vectors should be big enough so that they do not fit into the cache memories. You can check the size of your cache memories via `lscpu | grep cache`. If needed, modify the `MEGAS` manifest constant.

`likwid-perfctr` is a lightweight command line application to configure and read out hardware performance monitoring data on supported x86 processors.

`likwid-perfctr` requires the Linux `msr` kernel module. The Linux kernel adheres to a modular kernel design via *Loadable Kernel Modules* (`LKM`). The `msr` module allows access to the *Model-Specific Registers* (*MSRs*), several control registers in the x86 instruction set used for debugging, program execution tracing, computer performance monitoring, and toggling certain CPU features.

We need to check that the msr module is loaded before proceeding. To do so in Ubuntu Linux 18.04 LTS, enter:

```
sudo cat /etc/modules
```

The output should show a line with the word `msr`. If it is not present, you should add an `msr` entry to the `/etc/modules` file, since it contains the names of kernel modules that should be loaded at boot time. If necessary, reboot the machine.

Now we can use `likwid-perfctr`:

```
likwid-perfctr -C 0 -f -g MEM_DP ./stream_triad
```

where:

`- C 0`	means use the CPU with the physical ID `0`
`-f`	forces overwriting registers if they are in use
`-g MEM_DP`	indicates that the performance group to monitor is the one that gives us an overview of main memory and arithmetic performance

At first, `likwid-perfctr` prints some basic information about the used hardware: The `CPU name` is the actual CPU model name read from the system; the `CPU type` indicates the Intel's codename for the underlying micro-architecture (and thus determine the actual set of performance counters and registers available on the CPU); the `CPU clock` is a measurement of the current clock speed.

After that information we can see the program's output (printed in bright green using ANSI escape codes).

Below the line `Group 1: MEM_DP` we can see the `Event` section that shows us the number of occurrences in several categories. Here we are interested in just three of them:
`FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE`
`FP_ARITH_INST_RETIRED_SCALAR_DOUBLE`
`FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE`
which indicates us the number of executed floating-point operations using 128 and 256 bits

instructions as well as the number of floating-point scalar operations.

Where are the most of the operation counts? Try different optimisation levels (by modifying the `OPTFLAGS` macro in the `Makefile`) and different vector sizes (adjusting the `MEGAS` and `STREAM_ARRAY_SIZE` manifest constants).

Can you say anything about the timing results?

The above project uses a pair of portable (C, C++, and Fortran with a wrapper routine) routines to measure execution times from Robey and Zamora (2021). They are provided in the `timer.c` file. Here, the `clock_gettime` function is used with a `CLOCK_MONOTONIC` type to avoid problems with clock time adjustments (represents monotonic time since some unspecified starting point).

The next program to test is called `cpit1`. It is the same code from the previous practical session, but now it makes use of the timer routines by Robey and Zamora.

What do you think of the approach used to measure time? Can you figure it out a better approach?