Chapter 2: Case study

Version 0: Baseline

We will now write a baseline implementation that solves the problem in a straightforward manner:

```
void step(float* r, const float* d, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = std::numeric_limits<float>::infinity();
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = std::min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```

Is this a good implementation? Are we making a good use of the resources of the computer? Let's benchmark!

Platform

Throughout this chapter, we will use a normal low-end desktop computer (HP EliteDesk 800 G2 to be precise).

From our perspective the most important part is the CPU: it is **Intel Core i5-6500**, a relatively modern CPU based on the so-called "**Skylake**" architecture, introduced in 2015. More about the CPU later, but let us keep the following numbers in mind for now:

- there are **4 cores** in the CPU,
- when we use only one core, the CPU uses a clock frequency of approx. **3.6 GHz**.

The operating system is Ubuntu 16.04.4 with **Linux** kernel 4.4.0.

The C++ compiler is **GCC version 5.4.0**. We use the following command line to compile C++ code:

```
g++ -g -O3 -march=native -std=c++17
```

Here we ask the compiler to optimize heavily (`-O3`), and produce code that is tailored for this specific CPU (`-march=native`).

Benchmark instances

We use the value `n = 4000` in our benchmarks. The input consists of random floating point numbers between 0 and 1. We measure the time it takes to call the function `step` . We also print out some summary information about the result, to make sure the compiler cannot be too clever and optimize the function call away.

Results

It turns out that a single function call takes **99 seconds**. Is this a good or bad performance?

Each of the three nested loops runs for 4000 times, and hence the innermost loop runs for **64 billion** times. In the innermost loop, we do **one "+" and one "min" operation** for floating point numbers. These are the useful operations that we really want to do; everything else is just memory accesses and arithmetic related to loop counters and array indexing. That is, there are **two useful arithmetic operations** per one iteration of the innermost loop; in total we do **128 billion** useful floating point operations.

In summary, we managed to do 128 billion useful floating point operations in 99 seconds, or **1.3 billion operations per second**. This may sound like a large number, but let us keep in mind that this is only about 0.36 operations per clock cycle, and with modern CPUs we should aim at dozens of operations per clock cycle for each CPU core.