# Lab 1

## Lab I

### Introduction

The aim of today's class is to get acquainted with the Google benchmark library. The purpose of this library is to provide various facilities for microbenchmarking. Benchmarking is a term used to describe the measurement of execution time, the prefix "micro" indicates that these measurements should remain reliable even for very short execution times (which is not trivial!). Parameters other than execution time can further be quantified.

#### Is a dedicated library really necessary? Isn't `time` sufficient?

To demonstrate that a dedicated benchmarking library is, in fact, needed, let us try the naive approach. We will attempt to measure the time needed to compute the sine of 100 double precision floating point values. Paste the following into src.cpp:

```cpp
#include "MakeRandomVector.hpp"
#include <cmath>
#include <cstdint>
using std::ptrdiff_t;

int main()
{
    const auto random_vector = makeRandomVector(100, -3.14, 3.14); // 10
    auto       results       = random_vector;
    for(ptrdiff_t i = 0; double x : random_vector)
        results[i++] = sin(x);
}
```

To compile from the command line:

```bash
# First, download MakeRandomVector.hpp
wget https://gist.githubusercontent.com/kubagalecki/8b89de1850cf701441e031b878
# Now compile the code
g++ -std=c++20 src.cpp
```

Let us now measure the execution time:

```bash
time ./a.out
```

We can see that our code took several milliseconds to execute. However, a couple issues arise:

- If we perform a few more measurements, we will see that we get slightly different results each time. Which result is correct?
- We would like to measure only the time of the calculation. However, we have measured the execution time of the entire program. This includes the startup time, vector allocation, etc. Is this result even close to correct?

The latter problem could be solved by comparing clock values before and after the execution of the `for` loop (which is also an imperfect solution, since reading the clock takes a non-zero amount of time). However, to solve the former, we need to generate a set of measurements and perform some statistical analysis thereon. But how big should this set be? Should its size be a constant, or depend on the measured time? How should outliers be treated? As we can see, microbenchmarking is far from trivial. It's best to rely on solutions provided by people who have put considerable thought and effort into this topic.

### Library installation

Below, we propose 2 ways of installing Google benchmark

- Manual installation. This requires root privileges (`sudo`) and installs the library globally on the system.
- Installation using the Spack package manager. This does not require root privileges and is generally simpler.

### Manual installation

Manual installation is described in the repository readme. The only prerequisites are a working C++ compiler and CMake 3.5 or newer. Note that if you install the library in a custom directory, you'll later need to tell CMake how to find it. This can be done by setting the benchmark_DIR CMake variable at configure time. Your configure command will therefor have to look like this:

```
cmake -Dbenchmark_DIR="your/path/to/benchmark/installation" [other optio
```

### Installation using Spack

Spack is a package manager, which allows for easy installation and management of various libraries and tools. It's distinct in that it *builds* all required binaries, as opposed to downloading them from the relevant vendors. We will give a more in-depth explanation and guide later on in this course. For the time being, we only present a step-by-step instruction on how to install the library under discussion. Spack's documentation is available here. We do the following:

```
git clone -c feature.manyFiles=true https://github.com/spack/spack.git
spack/bin/spack external find # find available system packages, this wil
spack compiler find           # find available compilers
spack/bin/spack install benchmark ~performance_counters
```

Spack will recursively build and install all required tools and libraries. We only need to remember to call the following command each time we set up our build environment.

```
spack/bin/spack load benchmark
```

To load Spack's shell support, you can source the relevant script:

```
. /path/to/spack/share/spack/setup-env.sh
```

This will let you invoke Spack without specifying the full path. You can add this line to your .bashrc file to avoid having to retype it every time you open a new shell.

## Project configuration

To demonstrate how to use Google benchmark, we will first need to create a basic C++ project. We're going to require the following:

- Header files (in our case MakeRandomVector.hpp)
- Source files (in our case a single source file named BenchmarkDemo.cpp, though you may choose a different name)
- A CMakeLists.txt file, which contains the "recipe" for how to build our project

We will assume the following directory structure:

```
project_dir/
|
|__src/
|   |__BenchmarkDemo.cpp
|
|__include/
|   |__MakeRandomVector.hpp
|
|__CMakeLists.txt
```

The CMakeLists.txt file should contain the following:

```
cmake_minimum_required(VERSION 3.5)
project(my_project_name) # this does not matter for our simple case
add_executable(bench_demo src/BenchmarkDemo.cpp) # this determines the executa
target_include_directories(bench_demo PUBLIC include)
target_compile_features(bench_demo PUBLIC cxx_std_20) # we need C++20
find_package(benchmark REQUIRED)
target_link_libraries(bench_demo PUBLIC benchmark::benchmark)
```

To make sure we did everything correctly, let us test out the following code in BenchmarkDemo.cpp:

```
#include "benchmark/benchmark.h"
#include "MakeRandomVector.hpp"

BENCHMARK_MAIN();
```

To build and run our binary, we execute the following

```
# starting directory is project_dir
mkdir build
cd build
cmake ..
make
./bench_demo
```

The following text should be printed to the screen:

```
Failed to match any benchmarks against regex: .
```

The failure is expected, as we have not defined any benchmarks just yet.

### A brief explanation of what just happened

If you are even vaguely familiar with C++ development on Linux, you likely understand what we just did. If this is the case, feel free to skip ahead to the next section. For those who may be a little lost, we provide the following explanation. While not a prerequisite for the completion of this lab assignment, it's probably best to have at least a high-level understanding of the build process, especially since you'll need to build your own projects by the end of the semester. If after reading the explanation you still feel like you don't understand much, please ask the instructor for help.

Building a C++ binary is a multi-step process. An in-depth review can be found, e.g., here (though for our purposes, this level of detail is not needed). The two most important steps are:

- compilation of translation units (i.e. `.cpp` files) into object files
- linking of the compiled object files (and possibly outside libraries, as is the case here) into a binary

These require invoking the compiler and linker multiple times with various flags and arguments. Instead of doing this manually, the "recipe" for a binary is gathered into a `Makefile`, which can then be used as an input to the build tool `make` (this is done in the penultimate line of the build script above). However, the `Makefile` is still fairly low-level and writing one takes quite a bit of effort. Furthermore, it is strictly a Unix creation, so if we wish to build our project on Windows, we need a different solution. CMake was created to solve these problems. It has become the *de facto* standard, so, while other solutions exist, it is best to learn how to use it. CMake is not a build system, but a build system generator. In other words, it allows us to represent our project in a high-level and human-readable way, and then generates the `Makefile` (or its equivalent on other platforms) for us! It can also do other things, such as detect our compiler, find libraries, package software, and much more. CMake uses the concept of a "target" to describe a logical piece of a project. These can include executables, libraries (not necessarily buildable ones, e.g., header-only libraries can be targets too) and more. Targets can have properties, and we can model dependencies between different targets.

In our case, we have 2 targets:

- The executable `bench_demo`, which is defined by `add_executable(bench_demo src/BenchmarkDemo.cpp)`. Had our binary consisted of multiple `.cpp` files, we would have provided them all here, separated by spaces.
- The target `benchmark::benchmark`, which represents the Google benchmark library. This target is created by the call to `find_package(benchmark)`. As we can see, libraries packaged with CMake can be easily used by projects also using CMake.

Let us now go over the CMake file step-by-step.

1. `cmake_minimum_required(VERSION 3.5)` sets the required version of CMake. This is done to produce a clear error when attempting to use an older version.
2. `project(my_project_name)` simply names our project.
3. `add_executable(bench_demo src/BenchmarkDemo.cpp)` defines our executable target.
4. `target_include_directories(bench_demo PUBLIC include)` specifies the include directories for source files from the `bench_demo` target. Thanks to this line, we can call `#include "MakeRandomVector.hpp"` without specifying the full path. The modifier `PUBLIC` specifies that this property is inherited by all targets which depend on `bench_demo`. The other two options are `PRIVATE` - affects only the specified target and `INTERFACE` - affects only the targets which depend on `bench_demo`. This choice does not matter for our simple example (it becomes important for more complex projects).
5. `target_compile_features(bench_demo PUBLIC cxx_std_20)` specifies that we require C++20. This will result in the appropriate compiler flags being set.

6. `find_package(benchmark REQUIRED)` finds the Google benchmark library. `REQUIRED` specifies that CMake should produce an error if it can't find the requested package.
7. `target_link_libraries(bench_demo PUBLIC benchmark::benchmark)` specifies that our target `bench_demo` should be linked against the Google benchmark library. This utility can also be used more broadly to express dependencies between targets.

Now let's go over the build script

1. First we create the directory where we'll be building our binary (CMake produces quite a bit of files, so it's best to do this is a separate directory) and `cd` into it.
2. We invoke CMake with the argument `..`, i.e., the project directory. This tells CMake to look for a `CMakeLists.txt` file in the specified directory. Since one is found, CMake then automatically detects our compiler and environment, configures our project, and creates the `Makefile`.
3. We invoke `make`, based on the file generated by CMake. This builds our executable. Note that we do not need to know anything about the syntax or structure of the generated `Makefile`.
4. We run the produced executable.

**A note on using CMake**: When looking for resources online, you may encounter the old style of writing CMake files, which uses lots and lots of variables to set global properties (e.g. `set(CMAKE_CXX_FLAGS "-std=c++20")`). This is very much **not** recommended. Please use targets, set their properties individually, and express the relationships between them using `target_link_libraries`. If you need to set global properties, do it from the command line, for example

```
cmake -DCMAKE_CXX_FLAGS="-std=c++20" ..
```

## Learning Google benchmark

Let us now move on to the main part of this class. The general structure of a benchmark is as follows:

```
void my_benchmark(benchmark::State& state) // 1. Function defining the b
{
    // 2. Setup code
    const auto input = makeInput();

    for(auto _ : state) // 3. Benchmark loop
    {
        // 4. Code to be benchmarked
        const auto output = functionToBenchmark(input);
    }

    // 5. Tear-down code
    cleanup();
}
BENCHMARK(my_benchmark) /* <- 6. register benchmark; 7. options -> */ -> opti
```

1. Our benchmarks are defined as functions of type `void(benchmark::State&)`. The state object is used to interact with the library. The interaction works both ways - we can use this object to, e.g., query some parameters, and the library can use this object to determine whether to keep running the main loop.
2. This is the setup code for our benchmark. Its execution time does not get measured. In the example from the beginning of this class, this is where we would create the vectors of values.
3. This is the main benchmark loop. It is run until the library has collected enough data to be statistically significant. Note that `_` is just a name for an unused variable, it's not some special syntax.
4. This is the code which gets benchmarked.
5. In the tear-down code. Here we can perform any required final tasks, e.g. deallocate memory.
6. The `BENCHMARK` macro registers our function as a benchmark (without it, the library cannot magically ascertain that the function we wrote should be run).
7. We can pass some additional options using the `->` operator. We will discuss some commonly used options later, the full list can be found in the documentation.

The last piece needed to perform the measurements is a `main` function. If we don't wish to do anything special (and in this class we do not), we can just use the `BENCHMARK_MAIN()` macro as shown in the previous section. This will run all registered benchmarks and print the results to the screen as they are computed.

### How long does it take to do nothing?

The first operation we will be benchmarking is... doing nothing. Such an operation is commonly known as a noop (pronounced "no-op"). To do this, we can simply run the empty benchmark structure above (just make sure to delete the meaningless options). After building and running we should get an output similar to that below

```
2021-12-23T15:38:01+01:00
Running ./bench_demo
Run on (16 X 5100 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 256 KiB (x8)
  L3 Unified 16384 KiB (x1)
Load Average: 0.63, 0.35, 0.22
***WARNING*** CPU scaling is enabled, the benchmark real time measuremen
-----------------------------------------------------
Benchmark            Time             CPU   Iterations
-----------------------------------------------------
bench_noop        1.08 ns         1.08 ns    607058268
```

We can see that some information regarding our CPU is printed. Below, we have a table containing the data for our benchmark. The information in the columns describes:

1. The name of the benchmark
2. The time it took the benchmark to execute
3. The time the CPU was busy
4. The size of the measurement set (number of loop iterations). This will increase as the time of a single iteration decreases.

To demonstrate the difference between the "Time" and "CPU" columns, we will now benchmark waiting for a short period of time. To suspend the current thread of execution, we will use STL facilities from the `chrono` and `thread` headers:

```
using namespace std::chrono_literals;
std::this_thread::sleep_for(1s); // waits 1 second
```

Benchmarking the code above reveals that the execution time was roughly 1 second, however the CPU was busy for much less. To present the measurement in a more readable fashion, we can change the units as follows:

```
BENCHMARK(bench_wait)->Unit(benchmark::kSecond);
```

We can also name our benchmark:

```
BENCHMARK(bench_wait)->Unit(benchmark::kSecond)->Name("Wait 1 second"); // orc
```

### Some useful features

Let us now take a step towards solving the problem posed in the introduction. Before we compute the sine of a vector of values, we will start with just one.

**Sine of a value** We are now ready to write a benchmark which measures how much time it takes to compute the sine of an arbitrarily chosen value. Please do so on your own and note the result. Calculate the number of cycles it took to compute the sine (multiply the time by the clock frequency).

If we paid attention during the lecture, we will notice that one piece of the puzzle is missing. Namely, we are not enabling compiler optimization. To do this, we need to reconfigure our CMake project. We could manually set the compiler optimization flags, however this is not recommended. Since building a project in Debug (no or minimal optimization) and Release (full optimization) modes is ubiquitous, CMake provides a simple way of doing this - setting the `CMAKE_BUILD_TYPE` variable. We can leave our `CMakeLists.txt` file unchanged and simply call

```
cmake -DCMAKE_BUILD_TYPE=Release ..
```

After that we proceed as usual. If we rerun our benchmarks, we will see the following results:

```
-----------------------------------------------------------
Benchmark                 Time             CPU   Iterations
-----------------------------------------------------------
noop                  0.000 ns        0.000 ns   1000000000
Wait 1 second          1.00 s         0.000 s            10
Sine of a value       0.000 ns        0.000 ns   1000000000
```

The first result is good news - under an optimized build, doing nothing takes no time at all. The second benchmark should also not be a surprise - we asked to wait 1 second, and optimization should not affect the correctness of our code. However, the last result should give us pause. Is the CPU a trigonometric oracle, which knows the sine of all values without performing any operations at all? Of course not. What happened is simple - since the computed value is not used anywhere else in our program, it was optimized away by the compiler. We are therefore effectively benchmarking a noop - with the expected results. We therefore seem to be facing an impossible dilemma - either we compile under debug and get unrealistic results (since any real world computations will have optimization enabled), or we get no result at all. Fortunately, this is a common problem, for which the Google benchmark library provides a solution. We can use the `DoNotOptimize` function to prevent the compiler from optimizing away the result:

```
for (auto _ : state)
{
    double y = sin(x);
    benchmark::DoNotOptimize(y);
}
```

A benchmark written this way should give us a meaningful result. Before you run it, take an educated guess as to how many cycles it takes to compute a sine. Now run the benchmark and see if your reasoning was correct!

**Sine of a vector of values** We can now proceed to the problem posed in the introduction - computing the sine of a vector of values. Knowing the number of cycles it took to compute a single sine, take a guess as to how long it will take to process the whole 100-element vector. Now please write the relevant benchmark and take look at the results. Were you close?

**Note on optimization**: As of the writing of this text, `gcc` is not smart enough to optimize away the computation performed in this exercise, so `DoNotOptimize` isn't needed. However, if your compiler is able to do this (or you just want to be extra careful), please add the following code to the end of you benchmark loop:

```
benchmark::DoNotOptimize(y.data()); // y is the result vector
benchmark::ClobberMemory();
```

The first line will prevent optimizing away the result (`y.data()` is the address of the vector's underlying allocation) and `ClobberMemory` will prevent any clever tricks involving writing to memory.

**Passing arguments to benchmarks** It is often the case that we are tuning a piece of software and wish to establish optimal values for some heuristic parameters. Such examples were shown during the lecture (tile size, among others). The most straightforward way to achieve this is by performing an exhaustive search in the admissible parameter space, i.e., try all possible parameter combinations and see which one yields the best results. It would be quite inconvenient to have to write separate benchmarks for every parameter combination. Therefore, Google benchmark allows the user to define a parameter range as an option and query the current value using the state object. The code looks as follows:

```
void benchmark_fun(benchmark::State& state)
{
    const int param = state.range(0);
    // the rest of the benchmark uses the parameter value
}
BENCHMARK(benchmark_fun)->Arg(1)->Arg(2)->Arg(13)->Arg(42) /* -> other options
```

This will result in `benchmark_fun` being run 4 times, once for each of the arguments 1, 2, 13, and 42. If we wish to search a sparse parameter space, we can use the convenient shorthand of

```
BENCHMARK(benchmark_fun)->Range(1, 4096) /* -> other options */;
```

This will run the benchmark on the geometric sequence 1, 8, 64, 512, 4096. More generally, `->Range(min, max)` will choose some suitable parameters between `min` and `max`, each of them being 8 times greater than the previous one. This multiplier can be adjusted with the `RangeMultiplier` option. For example,

```
BENCHMARK(benchmark_fun)->RangeMultiplier(2)->Range(1, 4096);
```

will run the benchmark for each power of 2 between 1 and 4096. Google benchmark provides some more ways of passing arguments (including all possible combinations of multiple parameter sets), but the simple ones shown above should be sufficient for this course. If you are curious, you can always refer to the documentation.

Let us now practice passing arguments. Write a benchmark which computes the sine of a vector of values. The length of this vector should be a parameter of this benchmark. Measure the results for the lengths 256, 1024, 4096, 16,384, and 65,536.

**Throughput-oriented benchmarks** In the last example, reviewing the results may have been a little inconvenient. Different values of the passed parameter imply different amounts of work, so we need to divide the time by the problem size before comparing the results. For more complex examples, this can get quite tricky. For this reason, Google benchmark provides a way of explicitly printing how much work was done during execution. During the tear-down phase of the benchmark, we can call `state.SetBytesProcessed(<integral value>)` to indicate how much work was done during benchmarking. We will then get a nice bytes per second value in the results table. Note that this function expects the amount of work done during all iterations, not just a single one, so we need to use the `iterations` member function of the state object to query how many iterations were run. The code looks something like this:

```
void benchmark_fun(benchmark::State& state)
{
    const int work_per_iteration = 42;
    // main loop
    state.SetBytesProcessed(state.iterations() * work_per_iteration);
}
```

We can further pass the `--benchmark_counters_tabular=true` option when calling our executable to get better looking results (the throughput is aligned in a separate column).

**Pausing and resuming the timer** The last feature of the Google benchmark library we will learn about today is controlling the timer from within the main benchmark loop. It is sometimes necessary to do some setup or tear-down every iteration, but without counting the time it takes towards the measurement (see item 1 below for a concrete example). If we encounter such a situation, we can call `state.PauseTiming()` and `state.ResumeTiming()` to create a section of the loop which does not count towards the measured execution time. However, this functionality should be used only when absolutely necessary, since it introduces overhead and may skew the results.

### Problems for self-study

1. **Sorting.** Write a benchmark which measures the time it takes to sort `vector<double>`. See how it behaves for different sizes of the vector. Try to estimate the time complexity of the sorting algorithm and compare it with the theoretical value of O(n*log(n)). Does it take longer to sort a vector of `int`s or `double`s? **Note**: You will need to pause the timer at the beginning of every iteration to generate a new random vector (or to shuffle the old one, or assign a pre-computed random vector to the one which gets sorted), otherwise you will be measuring the time it takes to sort a sorted vector (which is a valid benchmark for a sorting algorithm, just not one that we're interested in right now).
2. **Matrix-matrix multiply.** Try for yourself some of the code shown during the lecture. Confirm that the parameters (e.g. for the tiling approach) were chosen correctly. Maybe the optimal values are different for your CPU?
3. **Follow your curiosity.** Are there any problems that you'd be interested in benchmarking? Maybe you think you can beat the implementation of some feature from some library (or the standard library). Maybe you're just curious why something seemingly simple takes a long time. Take the time to explore the questions you'd like to answer, and be sure to ask the instructor for help with interpreting the results, if you need it.