Lab 1: Development toolchain and package setup

MCHA4400

Semester 2 2023

Introduction

In this lab, you will install and configure the development environment on your own computer and install the following packages:

- clang++: C++ compiler
- cmake: Makefile generator
- ninja: High-performance build system
- cppcheck: Static C++ code analyser
- doctest: C++ unit testing framework
- nanobench: C++ microbenchmarking library
- boost: Additional C++ libraries
- Eigen3: Dense numerical linear algebra
- autodiff: Automatic differentiation
- SuiteSparse: Sparse numerical linear algebra
- OpenCV: Computer vision library
- VTK: 3D graphics and rendering library

Don't install these from the links above, we will use a package manager to do this in the sections below.

1 Toolchain setup

The following combinations of platforms and package managers are supported in MCHA4400:

- Section 1.1: MacOS (Intel or ARM) using homebrew
- Section 1.2: Ubuntu/Pop!_OS 22.04 (or later) using apt and homebrew
- Section 1.3: Windows 11 (WSL) with Ubuntu 22.04 (or later)
- Section 1.4: Windows 10 or 11 using MSYS2 (installed)
- Section 1.5: Windows 10 or 11 using MSYS2 (portable)

Refer to the appropriate subsection for setup instructions.

1.1 MacOS (Intel or ARM)

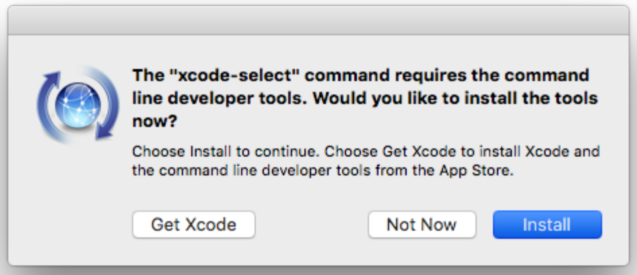
If you do not have Xcode installed[1], you can directly install the Xcode command line tools by entering the following into a terminal window:

[1] It is not necessary to install the full Xcode application from the Mac App Store, we only need the command line tools.

Terminal

```
mac:~ hipster$ xcode-select --install
```

This will generate the following prompt:


The "xcode-select" command requires the command line developer tools. Would you like to install the tools now?

Choose Install to continue. Choose Get Xcode to install Xcode and the command line developer tools from the App Store.

Click **Install** to continue installing the command line tools.

We will be using Homebrew to install packages. If you don't already have this installed, you can install it from the terminal as follows:

Terminal

```
mac:~ hipster$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Then, execute the following homebrew commands to install most of the needed packages:

Terminal

```
mac:~ hipster$ brew update
mac:~ hipster$ brew install cmake ninja pkg-config cppcheck
mac:~ hipster$ brew install doctest
mac:~ hipster$ brew install eigen autodiff suite-sparse
mac:~ hipster$ brew install boost
mac:~ hipster$ brew install vtk
mac:~ hipster$ brew install opencv
```

1.2 Ubuntu/Pop!_OS

If you have a fresh ∗nix installation you can install the basic GNU toolchain from the terminal as follows:

Terminal

```
nerd@basement:~$ sudo apt update
nerd@basement:~$ sudo apt install build-essential curl git
```

Install `clang` and set it as the default C++ compiler:

Terminal

```
nerd@basement:~$ sudo apt install clang
nerd@basement:~$ sudo update-alternatives --config c++
There are 2 choices for the alternative c++ (providing /usr/bin/c++).

  Selection    Path              Priority   Status
------------------------------------------------------------
* 0            /usr/bin/g++       20        auto mode
  1            /usr/bin/clang++   10        manual mode
  2            /usr/bin/g++       20        manual mode

Press <enter> to keep the current choice[*], or type selection number:   enter number corresponding to clang++
```

Since we need to use some package versions that are not available in the apt package manager (even for the latest version of Ubuntu), we'll use homebrew for Linux. To install homebrew, enter the following:

Terminal

```
nerd@basement:~$ /bin/bash -c "$(curl -fsSL
    ↪ https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Be sure to follow the post-install instructions displayed on screen.

Then, install the packages as follows:

Terminal

```
nerd@basement:~$ ulimit -n 1048576      workaround for "Too many open files" bug
nerd@basement:~$ brew update
nerd@basement:~$ brew install cmake ninja git pkg-config cppcheck
nerd@basement:~$ brew install doctest
nerd@basement:~$ brew install eigen autodiff suite-sparse
nerd@basement:~$ brew install boost
nerd@basement:~$ brew install vtk
nerd@basement:~$ brew install opencv
```

You may need to restart the terminal session for all of these tools to appear on the path.

1.3 Windows 11 (WSL)

Search for **Turn Windows features on or off** in the Windows search bar and ensure that **Virtual Machine Platform** is turned on. You may need to restart your computer after enabling this option.

Open the **Microsoft Store** app and find and install **Windows Subsystem for Linux Preview**. Search for **Turn Windows features on or off** in the Windows search bar and ensure that **Windows Subsystem for Linux** is turned on.

Open the **Microsoft Store** app and find and install **Ubuntu 22.04 LTS** (or a later version).

Warning

At the end of the installer should be a dialog to choose a username and password for the default account of the Ubuntu instance. If this is skipped or cancelled, no user will be created and you be logged in as the root user, which is bad.

If this occurs, we can create a user from the Ubuntu terminal as follows:

Terminal

```
root# adduser USERNAME                                    Replace USERNAME with desired user name
root# usermod -aG adm,cdrom,sudo,dip,plugdev USERNAME     Replace USERNAME with desired user name
```

Then, open **Windows PowerShell** and enter the following:

Windows PowerShell

```
PS C:\> ubuntu2204.exe config --default-user USERNAME     Replace USERNAME with desired user name
Tip: Type ubuntu and press tab to autocomplete to the Ubuntu executable installed
```

Upon restarting the Ubuntu terminal, you should be logged in as the user you created above, with the host filesystem mounted with your user permissions.

Open the **Microsoft Store** app and find and install the official **Windows Terminal** app by Microsoft. This unifies the **Command Prompt**, **Windows PowerShell** and **Ubuntu** shells into one terminal that features multiple tabs, simple copy and paste support (via mouse clicks), terminal colourisation, and drag-and-drop path support.

Open a **Ubuntu** shell using **Windows Terminal** and continue with the instructions found within Section 1.2.

1.4 Windows (Installed MSYS2)

Install the MSYS2 installer from `https://www.msys2.org/` and follow the instructions on that page that lead you to completing the following in the **MSYS2 MSYS** terminal:

Terminal

```
bored@work MSYS ~
$ pacman -Syu
bored@work MSYS ~
$ pacman -Su
```

Once you have completed the installation above, close the **MSYS2 MSYS** terminal and open the **MSYS2 Clang 64-bit** terminal. Install the development packages as follows:

Terminal

```
bored@work CLANG64 ~
$ pacman -S base-devel git mingw-w64-clang-x86_64-toolchain mingw-w64-clang-x86_64-cmake
    ↪ mingw-w64-clang-x86_64-ninja mingw-w64-clang-x86_64-cppcheck
[List of packages and dependencies]
Enter a selection (default=all):            Press enter
[More packages and dependencies]
Enter a selection (default=all):            Press enter
[More stuff]
Total Download Size:    260.16 MiB
Total Installed Size:  1690.88 MiB
Net Upgrade Size:      1582.69 MiB

:: Proceed with installation? [Y/n]         Press enter
```

Press enter at the `Enter a selection (default=all)` and `Proceed with installations? [Y/n]` prompts to proceed installing the packages and their dependencies.

Install the doctest package as follows:

Terminal

```
bored@work CLANG64 ~
$ pacman -S mingw-w64-clang-x86_64-doctest
```

Install the Eigen and SuiteSparse packages as follows:

Terminal

```
bored@work CLANG64 ~
$ pacman -S mingw-w64-clang-x86_64-eigen3 mingw-w64-clang-x86_64-suitesparse
```

Since autodiff is not available as an MSYS2 package, download, build and install it as follows:

Terminal

```
bored@work CLANG64 ~
$ git clone https://github.com/autodiff/autodiff
bored@work CLANG64 ~
$ cd autodiff
bored@work CLANG64 ~/autodiff
$ mkdir .build && cd .build
bored@work CLANG64 ~/autodiff/.build
$ cmake -G "MSYS Makefiles" -DCMAKE_INSTALL_PREFIX=$MINGW_PREFIX -DAUTODIFF_BUILD_PYTHON=OFF
    ↪ -DAUTODIFF_BUILD_EXAMPLES=OFF -DAUTODIFF_BUILD_TESTS=OFF ..
bored@work CLANG64 ~/autodiff/.build
$ cmake --build . --target install
bored@work CLANG64 ~/autodiff/.build
$ cd ../..
bored@work CLANG64 ~
$ rm -rf autodiff
```

Install the boost package as follows:

Terminal

```
bored@work CLANG64 ~
$ pacman -S mingw-w64-clang-x86_64-boost
```

Install the OpenCV package as follows:

Terminal

```
bored@work CLANG64 ~
$ pacman -S mingw-w64-clang-x86_64-openh264 mingw-w64-clang-x86_64-ffmpeg mingw-w64-clang-x86_64-opencv
```

Install the VTK package as follows:

Terminal

```
bored@work CLANG64 ~
$ pacman -S mingw-w64-clang-x86_64-cli11 mingw-w64-clang-x86_64-pdal mingw-w64-clang-x86_64-liblas
    ↪ mingw-w64-clang-x86_64-cgns mingw-w64-clang-x86_64-adios2 mingw-w64-clang-x86_64-openslide
    ↪ mingw-w64-clang-x86_64-utf8cpp mingw-w64-clang-x86_64-gl2ps mingw-w64-clang-x86_64-qt6-base
    ↪ mingw-w64-clang-x86_64-qt6-declarative mingw-w64-clang-x86_64-openvdb mingw-w64-clang-x86_64-unixodbc
    ↪ mingw-w64-clang-x86_64-vtk
```

Force terminal colorisation to workaround a known `ninja` issue with MSYS2 as follows:

Terminal

```
bored@work CLANG64 ~
$ echo export CLICOLOR_FORCE=1 >> ~/.bash_profile   The >> appends a line to the file; only do this once!
bored@work CLANG64 ~
$ source ~/.bash_profile                            Reload profile so we don't need to restart the terminal
```

Open the **Microsoft Store** app and find and install the official **Windows Terminal** app by Microsoft. This unifies the **Command Prompt**, **Windows PowerShell** and **MSYS2** shells into one terminal that features multiple tabs, simple copy and paste support (via mouse clicks), terminal colourisation, and drag-and-drop path support.

Within **Windows Terminal**, open **Settings** from the drop-down menu and then click the **Open JSON file** button in the bottom left corner of the settings page to open the settings JSON file within a text editor. Modify and save the JSON file according to the instructions under the **Windows Terminal** heading found at `https://www.msys2.org/docs/terminals/`. Modify the UCRT64 entry in `settings.json` to use CLANG64 as follows:

```
{
    "commandline": "C:/msys64/msys2_shell.cmd -defterm -here -no-start -clang64",
    "font":
    {
        "face": "Lucida Console",
        "size": 9
    },
    "guid": "{17da3cac-b318-431e-8a3e-7fcdefe6d114}",
    "icon": "C:/msys64/clang64.ico",
    "name": "CLANG64 / MSYS2",
    "startingDirectory": "C:/msys64/home/%USERNAME%",
},
```

The above assumes the default installation directory for MSYS2. If installed elsewhere, adjust the paths above as needed. Feel free to change the font face and size as desired.

1.5 Windows (Portable MSYS2)

If you are using a PC that you do not have administrative access to and cannot install software, you may be able to use a portable version of MSYS2 and run it from a directory you have write access to.

Download the latest version of MSYS2Portable from `https://sourceforge.net/projects/mwayne/files/MSYS2Portable/` and run the setup to deploy it to a directory you have write access to. Once this is done, open the **MSYS** terminal (run `MSYS2Portable.exe`) and let the initial setup complete. Close the **MSYS** terminal and re-open it (as per the on-screen instructions).

The portable MSYS2 environment assumes a particular directory exists in your user profile for temporary files, but the setup doesn't create it. Create it manually as follows in the MSYS2 terminal:

Terminal

```
bored@work MSYS ~
$ mkdir -p $LOCALAPPDATA/Temp/MSYS2PortableTemp
```

Note: As this directory is contained within your user profile, not the portable MSYS2 directory, it may be necessary to create this directory on each PC you use portable MSYS2 on.

Update the package database as follows:

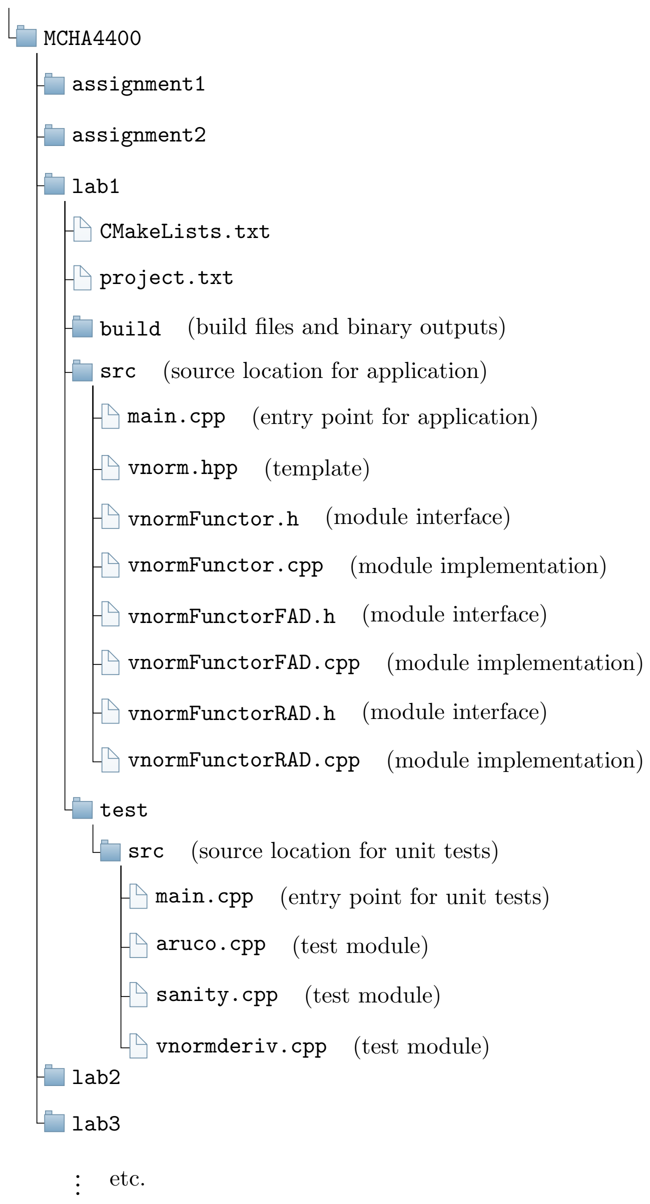
Terminal

```
bored@work MSYS ~
$ pacman -Syu
bored@work MSYS ~
$ pacman -Su
```

Close the **MSYS** terminal and open the **CLANG64** terminal (run `Clang64Portable.exe`), and continue installing the packages as described in Section 1.4. Note that you may be limited to using the terminal supplied with MSYS2 if **Windows Terminal** is not already installed on the system you are using.

2 Hello world

A simple application has been prepared that exercises several of the package dependencies (eigen3, autodiff, doctest, nanobench, opencv) in the included unit tests. Download the prepared zip file from Canvas and extract to a convenient location on your computer. The intended directory layout is shown in Figure 1, albeit absent the `build` directory, which is created during the build process described in the sections below, and the other assignment/lab directories, which you will add in later weeks.



```
MCHA4400
├── assignment1
├── assignment2
├── lab1
│   ├── CMakeLists.txt
│   ├── project.txt
│   ├── build    (build files and binary outputs)
│   ├── src    (source location for application)
│   │   ├── main.cpp    (entry point for application)
│   │   ├── vnorm.hpp    (template)
│   │   ├── vnormFunctor.h    (module interface)
│   │   ├── vnormFunctor.cpp    (module implementation)
│   │   ├── vnormFunctorFAD.h    (module interface)
│   │   ├── vnormFunctorFAD.cpp    (module implementation)
│   │   ├── vnormFunctorRAD.h    (module interface)
│   │   └── vnormFunctorRAD.cpp    (module implementation)
│   └── test
│       └── src    (source location for unit tests)
│           ├── main.cpp    (entry point for unit tests)
│           ├── aruco.cpp    (test module)
│           ├── sanity.cpp    (test module)
│           └── vnormderiv.cpp    (test module)
├── lab2
└── lab3
    ⋮   etc.
```

Figure 1: Intended directory layout.

Hint

Feel free to explore the included source code included in `lab1/src` and `lab1/test/src`. These may serve as useful examples in upcoming lab and assignment work.

Open the terminal and change to the directory where you extracted the zip file to. For brevity, we assume this directory is `~/MCHA4400` in the terminal examples in this section, but you should store your files in a sensible location and regularly back up your data.

Info

To access the host filesystem in Windows using WSL or MSYS2, some path translation is needed. For example, if the directory you wish to change to is `C:\my things\and stuff`, then in WSL use

Terminal

```
nerd@basement:~$ cd "/mnt/c/my things/and stuff"
nerd@basement:/mnt/c/my things/and stuff$
```

or in MSYS2, use

Terminal

```
bored@work MSYS ~
$ cd "/c/my things/and stuff"
bored@work MSYS /c/my things/and stuff
$
```

Note: If paths contain spaces, you need to surround the path with double quotes, or escape each space in the path with a backslash character (\).

Tip

There is a much faster way to do this on modern terminals such as MacOS Terminal or Windows Terminal, including any the path translation as required. Type `cd` followed by a space, then drag the directory from a Finder/Explorer window onto the terminal window to populate the path and then press enter.

Change to the `lab1` directory and then use `cmake` to create and prepare the `build` directory as follows (this should only ever need to be done once per project):

Terminal

```
nerd@basement:~/MCHA4400$ cd lab1
nerd@basement:~/MCHA4400/lab1$ cmake -G Ninja -B build
[cmake-ing intensifies]
If you encounter any errors at this point, check the dependencies were installed correctly
```

Change to the `build` directory and build the `lab1` example as follows:

Terminal

```
nerd@basement:~/MCHA4400/lab1$ cd build
nerd@basement:~/MCHA4400/lab1/build$ ninja
[Ninja-ing intensifies]
[Display benchmark results]
[Display unit test summary]
```

If there are any errors, check Section 4 before continuing.

Run the `lab1` application as follows:

Terminal

```
nerd@basement:~/MCHA4400/lab1/build$ ./lab1    Windows (MSYS2) users: add .exe extension
Hello world!
```

When repeatedly building and running an application during development, it is convenient to use the following command build and then run the application only if the build was successful:

Terminal

```
nerd@basement:~/MCHA4400/lab1/build$ ninja && ./lab1    Windows (MSYS2) users: add .exe extension
[Ninja-ing intensifies]
[Display benchmark results]
[Display unit test summary]
Hello world!
```

This avoids inadvertently running a stale version of the application if the build fails (and you weren't paying attention to the build output).

Info

The double ampersand (`&&`) used in the shell above is an **AND** operation that uses logical shortcutting. If the first command is unsuccessful, it will return a non-zero value, which is interpreted as **true**. Since **true** **AND**ed with any Boolean value is always **true**, the second command is not executed. If the first command is successful, it will return zero, which requires the second command to be evaluated to determine the result of the **AND** operation.

3 Try and break it

C++ is popular language for high-performance scientific and engineering computing applications due to the syntax and features of a high-level programming language, while retaining the performance of a low-level programming language such as C. Executables compiled to native machine code may be capable of significantly outperforming applications written in interpreted languages such as Python or MATLAB due to having:

- No run-time interpreter,
- No memory access bounds checking,
- No memory garbage collector.

This high performance comes at a cost, since there is no interactive debugging environment, no protection against unallocated memory access, and the potential for memory leaks. This can make certain types of bugs difficult to diagnose (detect and isolate) and then fix. So, let's look at some common errors first.

Open `src/main.cpp` in your favourite text editor and notice that there are several lines of code that are commented out. Uncomment one of these and then rebuild and run the application.

Terminal

```
nerd@basement:~/MCHA4400/lab1/build$ ninja && ./lab1    Windows (MSYS2) users: add .exe extension
[Ninja-ing intensifies]
[Display benchmark results]
[Display unit test summary]
Hello world!
[Maybe an error, maybe not]
```

Experiment by uncommenting different lines and observe the results. You should be scared about every line that you uncomment that *doesn't* crash the application, since these are the types of problems that cause spooky action at a distance that you may have previous experience with from embedded systems.

Now let's configure a debug build and see if the results are any different.

Terminal

```
nerd@basement:~/MCHA4400/lab1/build$ cd ..
nerd@basement:~/MCHA4400/lab1$ cmake --fresh -G Ninja -B build -DCMAKE_BUILD_TYPE=Debug
[cmake-ing intensifies]
nerd@basement:~/MCHA4400/lab1$ cd build
nerd@basement:~/MCHA4400/lab1/build$
```

Build the application and pay attention to the benchmark results, which display the performance in computing gradients and Hessians of a simple function using a technique known as automatic differentiation:

Terminal

```
nerd@basement:~/MCHA4400/lab1/build$ ninja
[Ninja-ing intensifies]
[Display benchmark results, which should be slow]
[Display unit test summary]
```

Compare the benchmark results to those obtained in the previous build.

A debug build is a different version of the executable that has

- Debugging symbols exported, so that an attached debugger can relate generated machine code to the appropriate lines of source code,
- All compiler optimisations disabled, so that the generated machine code more closely follows the sequence in the source code, enabling line-by-line debugging from an attached debugger,
- Assertions enabled, which can aid in troubleshooting some types of anticipated issues.

Experiment by uncommenting each problematic line in `src/main.cpp` one at a time and see if the results are any different in a debug build:

Terminal

```
nerd@basement:~/MCHA4400/lab1/build$ ninja && ./lab1
[Ninja-ing intensifies]
[Display benchmark results, which should be slow]
[Display unit test summary]
Hello world!
[Maybe an error, maybe not]
```

We will build up experience creating, diagnosing and fixing these and other types of errors throughout the course.

To switch back to the previous release build, change to the directory containing `CMakeLists.txt`, and use `cmake` as follows:

Terminal

```
nerd@basement:~/MCHA4400/lab1/build$ cd ..
nerd@basement:~/MCHA4400/lab1$ cmake --fresh -G Ninja -B build -DCMAKE_BUILD_TYPE=Release    Note: Release is
    ↪ the default if CMAKE_BUILD_TYPE is not specified
[cmake-ing intensifies]
nerd@basement:~/MCHA4400/lab1$ cd build
nerd@basement:~/MCHA4400/lab1/build$ ninja
[Ninja-ing intensifies]
[Display benchmark results, which should be fast]
[Display unit test summary]
```

4 Troubleshooting

4.1 Windows (WSL or MSYS2): permission denied/can't delete file related errors during build

This issue is known to occur under Windows when building out of a directory that is synced using Dropbox. The build process creates and deletes temporary files in rapid succession. Dropbox tries to index these files, which temporarily locks them, preventing deletion and failing the build process. This issue does not occur with Dropbox on other operating systems. The workaround is to temporarily pause syncing, or exclude the build directories from synchronisation.

If this issue occurs under Windows and you don't use Dropbox, try *temporarily* disabling any other file synchronisation software (e.g., OneDrive) or antivirus to try and find which process is interfering.

4.2 Ubuntu: "The function is not implemented" when calling `cv::imshow` from OpenCV

At the time of writing, the homebrew bottle (precompiled package) for opencv under Linux doesn't appear to be compiled against all its UI dependencies. We can fix this by compiling OpenCV from source as follows:

Terminal

```
nerd@basement:~$ ulimit -n 1048576
nerd@basement:~$ brew install gtk+
nerd@basement:~$ brew reinstall opencv --build-from-source
```

This may take 15–20 minutes to compile and there won't be much progress indication while it does.

4.3 MSYS2: `ninja` doesn't colourise output of `clang++`

This issue seems to be unique to the combination of using MSYS2, `clang++` as the compiler and `ninja` as the build system. This issue does not occur using `clang++` and `ninja` under Ubuntu or MacOS, nor does it occur with `clang++` and a different build system under MSYS2.

If the lack of compiler output colourisation is too annoying, you can use the traditional `make` build system instead of `ninja`. To do this, remove the build directory and configure the project using the `make` generator and rebuild it and run it as follows:

Terminal

```
bored@work CLANG64 ~/MCHA4400/lab1
$ rm -rf build
bored@work CLANG64 ~/MCHA4400/lab1
$ cmake -G "MSYS Makefiles" -B build && cd build
bored@work CLANG64 ~/MCHA4400/lab1/build
$ make -j8 && ./lab1            Replace 8 with number of physical CPU cores
[Make-ing intensifies]
[Display benchmark results]
[Display unit test summary]
```

Note that unlike `ninja`, `make` does not use multiple CPUs cores by default. The `-j8` parameter tells `make` how many jobs it should process in parallel, and it should be changed to the number of physical CPU cores on your system.