CMake

Lars Pastewka

Table of contents

Introduction

CMake is a cross-platform build system that helps manage the compilation of C/C++ projects of any size. We have seen in "Organizing code in C++" that compiling C++ files is usually done in two stages: the creation of object files and the linking of those objects. CMake will generate these compilation steps for you, provided you give it the right information about your project. In addition, CMake has the following advantages:

- **Cross-platform**: the workflow to compile CMake projects is identical across platforms. CMake will automatically identify the relevant compilers, dependencies, etc., to use for your particular platform.
- **Dependency handling**: it correctly handles library dependencies, which can be hard to setup manually, especially in high-performance cluster environments.
- **IDE integration**: many IDEs recognize CMake projects, including CLion, VSCode, and Visual Studio.
- **Industry standard**: CMake is widely used in the scientific computing and HPC communities, particularly for projects like Kokkos.

Note on this course: the project skeleton already fetches all dependencies (Kokkos, Eigen, MPI detection, Google Test) automatically via CMake's `FetchContent` module — this is the recommended path for the course, and you normally do not need to install anything by hand. The `find_package()` and manual-installation instructions below are shown for completeness and for HPC systems where Kokkos or Eigen are already installed as modules.

Basic CMakeLists.txt Structure

A minimal CMakeLists.txt file looks like this:

```
cmake_minimum_required(VERSION 3.21)

project(MyProject VERSION 1.0 LANGUAGES CXX)

# Set C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Add executable
add_executable(myapp main.cpp)
```

The first two lines are required: - `cmake_minimum_required(VERSION 3.21)` specifies the minimum CMake version (3.21 is recommended for Kokkos) - `project(...)` defines your project name, version, and languages

Building Your Project

To build a CMake project:

```
# Create a build directory
mkdir build
cd build

# Configure the project
cmake ..

# Compile the project
cmake --build .

# Run your executable
./myapp
```

Alternatively, you can use:

```
# Configure and compile in one command
cmake --build build --target all

# Run with verbose output to see compilation commands
cmake --build build --verbose
```

Note that the often-seen `make VERBOSE=1` (or `cmake --build build -- VERBOSE=1`) only works with Makefile generators; `cmake --build build --verbose` works with any generator (Makefiles, Ninja, …).

Debug vs. Release Builds

By default, CMake's build type is *empty*: no optimization flags and no debug symbols are added. You should therefore always set `CMAKE_BUILD_TYPE` explicitly — `Debug` for development and `Release` for production runs:

```
# Debug build
mkdir debug_build
cd debug_build
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake --build .

# Release build (optimized)
mkdir release_build
cd release_build
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake --build .
```

Common build types are: - `Debug` - No optimization, includes debug symbols - `Release` - Full optimization, no debug symbols - `RelWithDebInfo` - Optimization with debug symbols - `MinSizeRel` - Minimal size with optimization

Building Multiple Executables

For organizing your milestone codes, you might have separate `main.cpp` files in separate subdirectories. A possible directory structure could look like this:

```
milestones/
   CMakeLists.txt
   01/
       CMakeLists.txt
       main.cpp
   02/
       CMakeLists.txt
       main.cpp
   ...
```

The top-level `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.21)
project(Milestones LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

```
# Add subdirectories
add_subdirectory(01)
add_subdirectory(02)
add_subdirectory(03)
```

Each milestone's `CMakeLists.txt`:

```
add_executable(milestone01 main.cpp)
target_link_libraries(milestone01 PRIVATE Eigen3::Eigen MPI::MPI_CXX)
```

Using External Libraries

Finding Libraries

CMake includes `find_package()` to locate installed libraries:

```
# Find Eigen
find_package(Eigen3 REQUIRED)

# Find MPI
find_package(MPI REQUIRED)

# Find Kokkos
find_package(Kokkos REQUIRED)

# Link to your executable
add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE Eigen3::Eigen MPI::MPI_CXX Kokkos::kokkos)
```

Kokkos Configuration

If Kokkos is not provided via `FetchContent` (as in the course skeleton) or as a preinstalled module, you can configure and install it manually. Use version 4.7.03, which is the version this course is pinned to:

```
# Clone Kokkos at the version used in this course
git clone --branch 4.7.03 https://github.com/kokkos/kokkos.git
cd kokkos

# Create build directory
mkdir build
cd build

# Configure with desired options
cmake -DCMAKE_BUILD_TYPE=Release \
      -DKokkos_ENABLE_SERIAL=ON \
      -DKokkos_ENABLE_THREADS=ON \
      -DKokkos_ENABLE_CUDA=ON \
```

```
      -DKokkos_ARCH_AMPERE80=ON \
      ..

# Build and install
cmake --build . -j $(nproc)
cmake --install .
```

Then in your project's `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.21)
project(KokkosWave LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Find the installed Kokkos
find_package(Kokkos REQUIRED)

# Create executable
add_executable(wave wave.cpp)

# Link against Kokkos
target_link_libraries(wave PRIVATE Kokkos::kokkos)
```

Build your project:

```
mkdir build
cd build
cmake ..
cmake --build .
./wave
```

GPU Support in Kokkos

When installing Kokkos, enable the appropriate GPU backend:

For NVIDIA GPUs (CUDA):

```
cmake -DKokkos_ENABLE_CUDA=ON \
      -DKokkos_ARCH_AMPERE80=ON \
      ..
```

Kokkos invokes the CUDA compiler through its `nvcc_wrapper` script automatically; do not set `CMAKE_CXX_COMPILER=nvcc` (plain `nvcc` cannot act as a general C++ compiler). Alternatively, a `clang++` with CUDA support can be used as the C++ compiler.

For AMD GPUs (HIP):

```
cmake -DKokkos_ENABLE_HIP=ON \
      -DKokkos_ARCH_AMD_GFX90A=ON \
      -DCMAKE_CXX_COMPILER=hipcc \
      ..
```

See notes/kokkos_gpu_compilation.qmd for detailed GPU compilation instructions.

Using CMake in IDEs

CLion

CLion has excellent CMake support:

1. Open your project folder in CLion
2. CLion will automatically detect `CMakeLists.txt`
3. Configure build type in `Settings → Build, Execution, Deployment → CMake`
4. Add build type and compiler options in the "CMake options" field:

   ```
   -DCMAKE_BUILD_TYPE=Release
   -DKokkos_DIR=/path/to/kokkos/install
   ```

VSCode

With the CMake Tools extension:

1. Install the CMake Tools extension
2. Open the command palette (`Ctrl+Shift+P`) and run "CMake: Configure"
3. Select your compiler
4. Build with `Ctrl+Shift+B` or the CMake sidebar

Common CMake Variables

Variable	Purpose
`CMAKE_CXX_COMPILER`	C++ compiler to use
`CMAKE_CXX_FLAGS`	C++ compiler flags
`CMAKE_BUILD_TYPE`	Build type (Debug, Release, etc.)
`CMAKE_INSTALL_PREFIX`	Installation directory
`CMAKE_PREFIX_PATH`	Directories to search for packages

Troubleshooting

"Could not find Kokkos"

Ensure Kokkos is installed and CMake can find it:

```
# When installing Kokkos, note the installation path
cmake --install . --prefix /path/to/kokkos/install

# When configuring your project, tell CMake where to find Kokkos
cmake -DKokkos_DIR=/path/to/kokkos/install/lib/cmake/Kokkos ..
```

Alternatively, set the environment variable:

```
export CMAKE_PREFIX_PATH=/path/to/kokkos/install:$CMAKE_PREFIX_PATH
cmake ..
```

Compiler not found

Explicitly specify the compiler:

```
cmake -DCMAKE_CXX_COMPILER=g++ ..
```

CUDA/HIP compiler errors

Ensure NVIDIA/AMD tools are in your PATH:

```
# For CUDA
export PATH=/usr/local/cuda/bin:$PATH

# For HIP/ROCm
module load rocm  # On HPC clusters
export PATH=/opt/rocm/bin:$PATH
```

Further Resources

- CMake Official Documentation
- CMake Tutorial
- Kokkos CMake Configuration
- CLion CMake Support