CMake - Cross-Platform Makefile Generator

Anastasia Kruchinina

January 20, 2015

Contents

Abstract

CMake is a cross-platform build systems generator which allows to build software on a broad set of platforms on easier and unified way. In this report our goal is to compile the quantum chemistry code Ergo using CMake. In addition, we discover abilities of testing tools CTest and CDash.

The project github repository:
`https://github.com/Tourmaline/cmake-project.git`.

1 Introduction to CMake

CMake is the cross-platform, open-source build system. CMake generates native makefiles and workspaces that can be used in many different compiler environments. On Linux systems the default build environment is make, but other options are possible.

CMake supports *out-of-source build.* It is advised never mix-up source files with generated files. With out-of-source build the whole code structure looks better since all cmake files are hidden in some subdirectory. Another advantage is that it allows to have more build trees for the same source code. A good practice is to set a cmake file-guard which prevent in-source build. Then we can create build directory and run cmake from it using command `cmake /path/to/source`. All cmake files will be located in this build directory.

Compilation can be done in parallel using make: `make -j [N]`, where `N` is the number of processes.

In order to specify configuration variables, one can use the command line with the `-D` option. Very useful to use the `-i` option for interactive setting of the variables. Another cache editors are cmake-gui and ccmake.

The option `-L` to cmake allow to see all (non-advanced) variables.

Cmake commands are written in the files CMakeLists.txt. Cmake allows easy check for the **existence** of the libraries, functions, header files and compiler flags.

For example, in order to check the existence of the `-pedantic` flag add to the CMakeLists.txt the following code:

```
INCLUDE(CheckCCompilerFlag)
CHECK_C_COMPILER_FLAG(-pedantic C_HAS_PEDANTIC)
IF (C_HAS_PEDANTIC)
  SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -pedantic")
ENDIF ()
```

CMake provide possibility to specify rules to run at **install** time. To change install directory one should change installation prefix:

```
cmake -DCMAKE_INSTALL_PREFIX=/install/path/
```

Interesting, but CMake does not provide an uninstall option.

CPach is a cross-platform packaging tool. To setup generator, run for example the following command

```
cpack -D CPACK_GENERATOR="ZIP;TGZ"
```

Then zip and tgz archive files will be generated in the current build directory.

CMake allow to write **configuration files**. The function `configure_file` reads input file and produce new configure file where all variable definitions from input file replaced by their values specified in CMakeList.txt. If the input file is modified the build system will re-run CMake to re-configure the file and generate the build system again. In our case the input file is cmake_config.h.in. If the variable in cmake_config.h.in. is specified using `cmakedefine`, then CMake will change `cmakedefine` on `define` or `undef` depending on whether the variable defined or not. Another option is to use variables written as `@var@`. Then CMake will change `@var@` to the value of `var` defined in CMakeFiles.txt.

2 CTest and CDash

2.1 Testing using CTest

The command in the CMakeFiles.txt `enable_testing()` adds another build target, which allows to run tests for Makefile generators, or RUN_TESTS for integrated development environments (like Visual Studio).

```
 add_test(NAME <name> [CONFIGURATIONS [Debug|Release|...]]
            [WORKING\_DIRECTORY dir]
            COMMAND <command> [arg1 [arg2 ...]])
```

It is possible to execute tests in parallel: `ctest -j [N]`

2.2 CDash

The result of a test run, reformatted for easy review, is called a "dashboard". CDash is an open source, web-based software testing server. One can install own server or use one of public servers. The default settings of the module are to submit the dashboard to Kitware's Public Dashboard, where you can register your project for free. Notice that CDash Free Edition doesn't allow more than 10 builds a day.

In order to submit to some other server, put the CTestConfig.cmake in the top level directory of your source code, and set your own dashboard preferences. If you are using a CDash server, you can download a preconfigured file from the respective project page on that server ("Settings" / "Project", tab "Miscellaneous").

There are three types of dashboard submissions:

Experimental means the current state of the project. An experimental submission can be performed at any time, usually interactively from the current working copy of a developer.

Nightly is similar to experimental, except that the source tree will be set to the state it was in at a specific nightly time. This ensures that all "nightly" submissions correspond to the state of the project at the same point in time. "Nightly" builds are usually done automatically at a present time of day.

Continuous means that the source tree is updated to the latest revision, and a build / test cycle is performed only if any files were actually updated. Like "Nightly" builds, "Continuous" ones are usually done automatically and repeatedly in intervals.

The standard procedure for runing tests and submutting results to dashboard is

```
Start -> Update (only Nightly) -> Configure ->
Build -> Test -> Coverage -> Submit.
```

To schedule a nightly build with all options turned on run

```
ctest -VV -S /path/to/my_cdash_script.cmake
```

2.3 Code coverage

Code coverage (`http://en.wikipedia.org/wiki/Code_coverage`) it is a measure of how many lines of codes are executed during tests. The information is collected by instrumentation of the code using special programs.

The CTest allow writing scripts which automate testing and submitting dashboards. Cron can run jobs which are running such scripts regularly.

Currently coverage is only supported on gcc compiler `cgov`[`http://en.wikipedia.org/wiki/Gcov`]. To perform coverage test, make sure that your code is built with debug symbols, without optimization, and with special flags. These flags are: `-fprofile-arcs -ftest-coverage`

CTest maintains the following four metrics:

- Number of lines covered
- Number of lines not covered
- Total number of lines in code
- Percentage of coverage (number of covered lines/total lines)

On the CDash webpage of the project one can find all the information about code coverage.

2.4 Dynamic Analysis

Dynamic Analysis can be performed using Valgrind Memcheck during running tests. Results about memory leaks, uninitialized memory reads, freeing invalid memory, invalid pointer reads and so on can be found on the project dashboard.

3 CMake for Ergo code

In order to compile the Ergo code (with Release build type) just run

```
cmake /path/to/source
```

In the Ergo code bash scripts are used for testing. Every time during the configuration CMake copy `cmake_test` directory to the build directory. The reason for that is that test files must know the location of the executable. After running CMake and compilation, the CTest run tests from the build directory.

CDash for the Ergo code:
`http://my.cdash.org/index.php?project=ergo_cmake`.

The build errors, coverage and dynamic analysis are presented there.

Notice, that one can make code invisible for public by changing settings of the project on the website.

In order to run coverage test, run cmake with Debug build type:

```
cmake -DCMAKE_BUILD_TYPE=Debug /path/to/source
```

It will add needed flags to the compiler. For testing without reporting results one can use just `make test` or `ctest` commands. For testing and reporting results to the dashboard can be used `cmake -D Experimental`.

3.1 Cron

Nigthly tests can be run using cron environment. In order to set up the cron to run nightly test every night at 22:00, one can use the following command

```
# m h  dom mon dow   command
00 22 * * *  . ~/.keychain/$HOSTNAME-sh; ctest -VV -S
    /path/to/DashScript.cmake > /path/to/logs/cdash_nightly.log 2>&1
```

The DashScript contains commands for CTest to run clone git repository, to fetch changes, configure and compile the code, run tests and submit the results to the dashboard.

On order to be able to run such command, it is needed to add ssh key to the git account to allow fetching new versions without password check and install `keychain` to make cron install needed ssh environment variables before running script.

Then after the submission of the job, one can see on dashboard which files were updated.

4 Using CMake with ninja build system

CMake by default generates Makefiles on Linux systems, but here is an option to generate files for other build systems. For example, let us consider ninja (`https://github.com/martine/ninja`). Ninja is a small build system designed for speed.

```
cmake -G Ninja -DCMAKE_MAKE_PROGRAM=/path/to/ninja ..
```

For compilation of the program just run `ninja`. Let us compare results of compilation with make and with ninja using option `-j4`:

	real	user	sys	size
cmake + ninja	6m19.640s	10m16.851s	0m9.065s	3.4MB
cmake + make	5m45.345s	9m43.693s	0m10.315s	3.4MB

In our case the program compiles a bit slower using ninja build file then using Makefiles. However, many developers report the opposite.

5 Automake vs CMake

The project Ergo originally use Autotools for generating Makefiles. We generate Makefiles using both CMake and Autotools. Compilation was done with `-O3` flag and `-j4` option for make.

The result for Autotools is presented below:

	real	user	sys	size
autotools + make	6m30.709s	9m24.159s	0m8.878s	3.2MB

Compare this results with the previous table (cmake + make). Using cmake we gain 10% speed up of the compilation.

Automake vs CMake:

- Automake/autoconf are replaced by cmake. Any change in the input config.h.in file or CMakeFiles.txt will result in the reconfiguration. Instead if one use autotools, user must call automake or autoconf manually.
- CMake provide cache (CMakeCache.txt), where it save system and user variables. During next configuration, cmake looks at the cache file and add just new variables from CMakeLists.txt or redefine the variables specified in the command line. If user modified variables in CMakeFiles.txt, it is needed to update the cache. It can be done using a CMake GUI (CMake-Setup on Windows or ccmake on UNIX), by wizard mode (cmake -i). One can use also directly delete CMakeCache.txt from the build directory and reconfigure. It make sense since usually user do not modify variables in the CMakeFile.txt, but specify them in the command line.
- Both generators allow to check existence of functions/include files. CMake allow to specify which information will be written on the configuration file. In contrast, autoconf is adding too much information, which is actually never used in the code.
- In order to compile project with CMake, developer should learn its own non-trivial syntax, different from Autotools.
- Both autotools and CMake is easy to install.
- Automake and CMake can exist together in one project, their files do not overlap.
- Autotools are done for Linux systems and work just with Makefiles. CMake provide support for many platforms, compilers and build systems. CMake is perfect for creating portable c++ software.
- Both generators require at least two steps for compiling program. Firstly, you must run cmake/configure and after that use generated makefiles.
- In addition to a build system, over the years CMake has evolved into a family of development tools: CMake, CTest, CPack, and CDash. Using CTest and CDash it is easy to monitor the status of the development.
- CMake gives nice colorful output and undestandable description of errors.
- Makefiles generated by CMake are twice shorter then Makefiles from Automake.

In conclusion, for any new project it is advisable to use CMake since it is very promising build system which is becoming very popular. In my opinion, CMake allow more control on how to configure/compile the project.

CTest and CDash tools can be used with different generators of the build system, not just cmake. It provides easy way to configure tests and present the results on the server which are visible to all developers.

6 Cmake tutorials

There are not so many good tutorials about CMake. In this section I am mentioning some if them.

The documentation can be generated by `cmake --help-html`.
Slides of Eric Noulard from github
    `https://github.com/TheErk/CMake-tutorial`
Slides from the course
    `http://bast.fr/talks/cmake/kung-fu`
Cmake wiki:
    `http://www.cmake.org/Wiki/CMake`
CMake Useful Variables:
    `http://www.cmake.org/Wiki/CMake_Useful_Variables`
CTest:
    `http://www.vtk.org/Wiki/CMake/Testing_With_CTest`