Masaryk University
Faculty of Informatics

GPU-based speedup of EACirc project

Bachelor Thesis

Jiří Novotný

Brno, Spring 2015

Contents

1 CMake

The EACirc mainly consists of *C* and *C++* code. The code was divided into reasonably logical sections but the overall structure and concept of the project was monolithic.[1] This tended to compile all sources into one big executable approximately 9 MB in size which took some non-trivial time and also this design was not apt to scale.

On top of that EACirc is developed as a cross-platform application. To provide native builds for each supported platform (Windows [1] and Linux) there was a special makefile, or an IDE specific project file, which described how to build the application. When a change in the build was introduced, e.g. a new source file was added, the change must have been manually propagated to all makefiles to provide consistency. This workflow was not easy to maintain and the violation of these rules could cause an uncomfortable pitfall.

To solve these problems the CMake [2] tool was integrated into the project of EACirc. The tool is developed and maintained by Kitware, Inc. [3] as an open-source software. The main purpose of this tool is to provide native builds of cross-platform applications and tries to minimize the effort to maintain the project.

Although there are many similar tools as CMake, and some of them provides better features, they are not so widely supported. CMake provides generation of project files for almost every common IDE and some of those IDEs comes with a support for CMake.

1.1 CMake toolset

The CMake is actually a set of several executables; the main executable `cmake` and the supportive ones `ccmake` (or `cmake-gui`), `ctest`, and `cpack`.

The `cmake` executable takes a configuration file called `CMakeLists.txt` distributed with the sources and generates the platform specific makefiles as an output. Then the user invokes a platform specific tool for building – usually `make`, `ninja`, or `MSBuild`. If the process was successful the native binaries of the project are now made.

The `ctest` executable provides a simple platform for project testing. After the binaries are successfully build the invocation of `ctest` usually runs some user-defined test on top of them.

The last but not least executable `cpack` provides a cross-platform way to deploy your application on the target system.

The remaining executables `ccmake` and `cmake-gui` are just a more convenient ways to use a `cmake` executable since `cmake` has only a command line interface. The former provides a TUI[2] and the latter provides GUI[3].

---

1. A monolithic binary is an executable that does not need any other dependencies or resources at a runtime. In other words, the binary is independent.
2. Text-based user interface (TUI)
3. Graphical user interface (GUI)

1.2 A closer look to the `cmake` executable

The `cmake` executable is not just a dummy build-system. The process of generating a makefile is far more complex. Firstly the user chooses the *source directory* and the *build directory*. Then he invokes the `cmake` command in a *build directory* with appropriate parameters. The subsequent process consists of several phases – selection of a native build-system (in a CMake terminology referenced as a *generator*), configuration based on a user-specific input, and the own generation of a makefile.[4]

The *source directory* is simply a directory where the project sources are located and where the top-level `CMakeLists.txt` file is located, which is distributed with the sources. The build directory is an empty user-created directory where he wants the binaries to be build.

The selection of the *generator* depends on the user's platform, on the user-installed native build-systems, and on the user's intentions. On Linux the generator used is usually `make` or `ninja`. When the user wants to generate a project files to a specific IDE, he chooses the appropriate generator – e.g. Visual Studio 2013 [4] on Microsoft Windows [1]. Usually the selection of appropriate generator is done by CMake automatically.

The subsequent phase, configuration, is in place. Here the user specifies variable options for the build that the project supports. For instance some features of the application can be switched on/off or the location of a third party dependencies can be specified. Also the different build configuration can be switched, i.e. release or debug.

If the configuration is all right then the makefile is successfully created in the *build directory*. Then we just invoke the appropriate tool to execute the makefile and the binaries are build.

There is a worth mentioning that the makefile automatically detects any changes made in the *source directory* so the user invokes the `cmake` executable just once to generate the makefile or when he wants to change the variable options of the build. The makefile also provides a target for installing the application and/or a target fo testing.

The minimal and the most common sequence of commands to build and install a project on Linux using the CMake are:

```
mkdir <build_directory>
cd <build_directory>
cmake <path_to_source_directory>
make
make install
```

Note that the `make` is chosen as a default generator and the default project settings and configurations are applied. The binaries are installed to the platform specific location, i.e. for Linux it is `/usr/share/local`.

4. Note that the exact scheme of this process can differ according to which interface of CMake is being used – i.e. `cmake`, `ccmake`, or `cmake-gui`.

1.3 Changes made to the EACirc repository structure

There had been made several changes to the EACirc repository structure. The new folder design should reflect the logical structure of the EACirc philosophy.

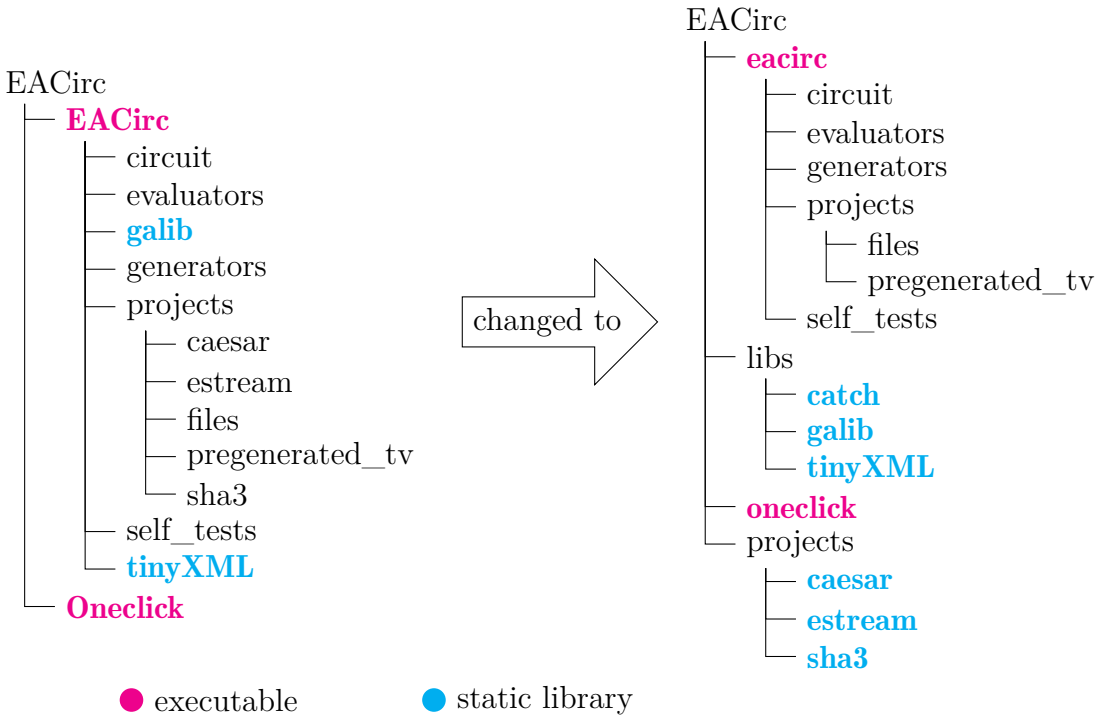

Figure 1.1: Old vs. new repository structure

The first and also the smallest change was to name all source folders with only small letters. Next the libraries from 3rd party providers, `catch`, `galib`, and `tinyXML`, were moved into the separate folder – the *libs* directory.

Then there was a request to isolate the so called *projects*. A *project* in EACirc terminology means a module for some problem solving. These *projects* are `caesar`, `estream`, `sha3`, `files` and `pregenerated_tv`. Since `files` and `pregenerated_tv` are both just small modules consisting from only one source file, it would be impractical to isolated them. On the other hand the big modules `caesar`, `estream`, and `sha3` were moved to the the separate folder, called the *projects* folder. Each of the isolated projects was made to compile into a static library.[5]

From the folders `eacirc` and `oneclick` are build executables named accordingly to their corresponding folders. The *projects*, which are now compiled into the static libraries, are now statically linked to the `eacirc` executable, which represents the EACirc tool as a whole. The `oneclick` executable is a supportive tool for automated task management developed by Ľubomír Obrátil. [5]

5. There is a plan to remade the projects to modules loaded dynamically at runtime. This would require to compile them separately into the dynamic libraries.

1.4 The new build-system of EACirc

The new build-system is written on the CMake platform. This platform allows to define custom options for generating the build. Here is a descriptive list of EACirc specific options:

BUILD_ONECLICK Enable building of Oneclick, the supportive tool for EACirc.

BUILD_CAESAR Enable building of the Caesar project.

BUILD_ESTREAM Enable building of the Estream project.

BUILD_SHA3 Enable building of the SHA-3 project.

BUID_CUDA When enabled, EACirc is build to support CUDA devices. This option is available only if the CUDA Toolkit [6] is installed on the build machine[6] and found by the CMake.

Since the *projects* are build into static libraries they must be linked to the `eacirc` executable at the compile time. This is done automatically when the option for the specific *project* is enabled. In the figure 1.2 are shown the dependencies of the all build targets.

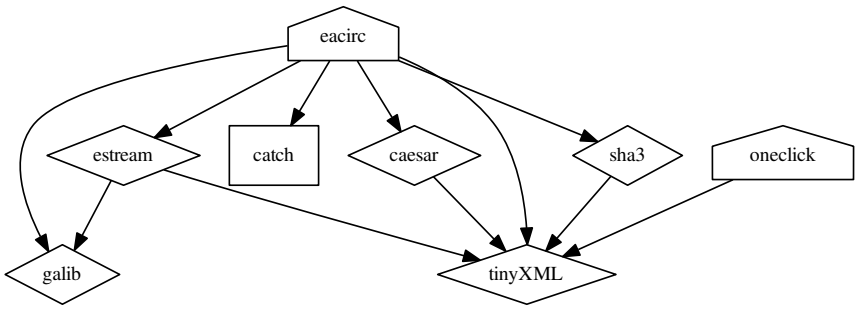

Figure 1.2: EAcirc dependency graph
The static libraries are shown in the rhombus. The executables have a house around them. The square represents an interface library.[7] The direction of the arrows represents that some build target depends on another.

The build-system is also version aware. The current version is stored in the `eacirc/Version.h` header file. The version corresponds to git commit hash [7]. That means that for correct build generation there must be a git tools properly installed on the build machine and found by CMake.[8]

The usage of CMake and the new options of building EACirc are explained closely at the project Wiki page on Github under the Building EACirc section.

---

6. A build machine is a physical or a virtual machine that is used to build the project.
8. If git tools are installed and not found automatically by CMake then the path to git tools can be specified manually.

1.5 Project settings for CUDA

To set the project for CUDA is now much more easier with CMake support. When the CUDA Toolkit [6] is installed and automatically found by CMake[9] then the option `BUILD_CUDA` becomes available. If this option is enabled then the `eacirc` executable is build using Nvidia [8] `nvcc` compiler and the C preprocessor macro `CUDA` is defined causing that the executable will be runnable on CUDA capable devices. When writing a code for CUDA the preprocessor macro `CUDA` can be queried.

9. If CUDA Toolkit is installed on the build machine but not found by CMake automatically then the path to CUDA Toolkit can be specified manually.

Bibliography

[1] Microsoft. (2015). Windows – microsoft windows, [Online]. Available: http://windows.microsoft.com.

[2] I. Kitware. (). Cmake, [Online]. Available: http://www.cmake.org/ (visited on 03/08/2015).

[3] ——, (). Kitware, inc. – leading edge, high-quality software, [Online]. Available: http://www.kitware.com/ (visited on 03/08/2015).

[4] Microsoft. (2015). Visual studio – microsoft developer tools, [Online]. Available: https://www.visualstudio.com/.

[5] Ľ. Obrátil, "Automated task management for eacirc and boing", type, FI MUNI.

[6] N. Corporation. (2015). Cuda toolkit, [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[7] B. S. Scott Chacon, *Pro Git*, 2nd editon. Apress, Dec. 24, 2014, ISBN: 978-1484200773.

[8] N. Corporation. (2015). Welcome to nvidia - world leader in visual computing technologies, [Online]. Available: http://www.nvidia.com.

TODO Fix the autors of online resources
TODO Fix the titles in the bibliography to dislay big letters correctly.
TODO Cite Lobo's theses about oneclick and fix the source.
TODO Cite Martin Ukrop thesis in Introduction. What is EACirc?