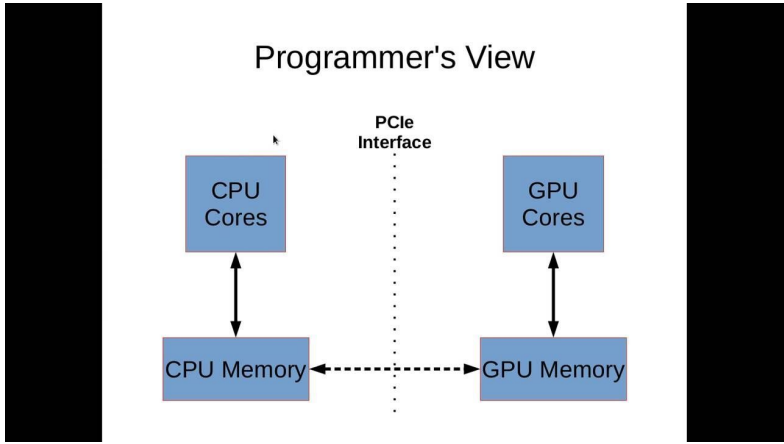
C++ Course 9 : Using cmake

2020 by Oleksiy Grechnyev

Technology of the day : CUDA



So you heard about running stuff on a GPU?
Nvidia uses CUDA.

Important facts about CUDA and GPU:

1. GPU is not a super-CPU! Each core is *much slower*!
2. Only makes sense for *massively parallel* algorithms.
3. GPU uses separate memory. CPU<->GPU transfer can be a bottleneck!
4. It is unlikely you will succeed if you are not a pro.
5. Use higher level libraries if possible: cublas, cudnn ...
6. CUDA is not very friendly to CPU multithreading.

Terminology:
HOST = CPU + CPU memory
DEVICE = GPU + GPU memory
Read : article1 article2 programming guide

How CUDA works?

It is a language extension to C/C++. Compile with **nvcc** from the CUDA distro.
It compiles *host* (CPU) code using the regular **gcc** or **cl**.
It compiles *device* (GPU) code into PTX bytecode or GPU machine code.
PTX = Parallel Thread Execution, common for all Nvidia GPU models.
GPU machine code differs for different GPU models: *compute capability* 1.0 - 8.0.
PTX is compiled to the machine code by the Nvidia driver : takes time on the first run!
When timing CUDA apps: Expect long warm-up times (loading code to the GPU).

Using CUDA with CMake: you need to enable language **CUDA**.

```
cmake_minimum_required(VERSION 3.8)
project(e_cuda LANGUAGES CXX CUDA)   # Enable language CUDA
add_executable(add1 add1.cu)    # CUDA files have extension .cu
```

CMake takes care of everything if it can find **nvcc**.
To use "CUDA runtime" in .cpp files: **find_package(CUDA REQUIRED)**.

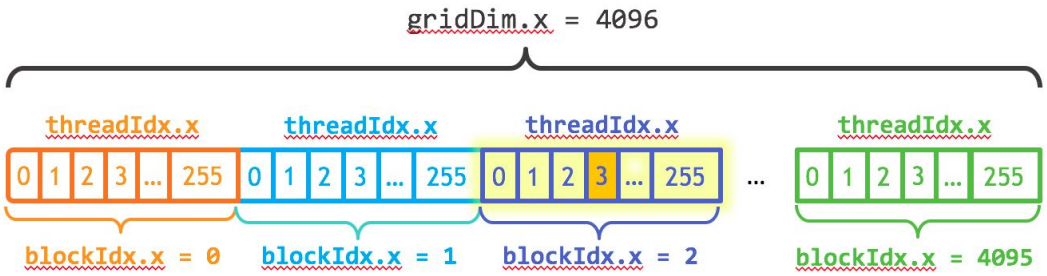
Technical note: **nvcc** often requires outdated **gcc** versions on Linux.

CUDA kernel: A function which runs on the device (called from host)

Add **n** numbers on GPU. Pointers must point to GPU memory!

```
__global__ void add(int n, float *x, float *y){
      int index = blockIdx.x * blockDim.x + threadIdx.x;  // Total thread index
      int stride = blockDim.x * gridDim.x;     // Total number of threads
      for (int i = index; i < n ; i += stride)
            y[i] += x[i];
}
```

Threads are divided into *blocks*:




```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *   (256)    +     (3)      = 515
```

Running the kernel: Using unified CPU/GPU memory

```
// Alloc unified CPU/GPU memory
float *x, *y;
cudaMallocManaged(&x, n*sizeof(float));
cudaMallocManaged(&y, n*sizeof(float));
...  // Initialize the arrays
// Run the kernel
int blockSize = 4;   // Number of threads per block
int numBlocks = 3;  // Number of blocks
add<<<numBlocks, blockSize>>>(n, x, y);
// Wait for the GPU, needed here because of cudaMallocManaged()
cudaDeviceSynchronize();
...
// Free memory
cudaFree(x);
cudaFree(y);
```

Running the kernel: Transfer CPU<->GPU by hand

```
int nF = n*sizeof(float);
std::vector<float> x(n, 2.0f), y(n, 1.0f);  // Create data in the CPU memory (host)
// Allocate GPU (device) memory
float *dX, *dY;
cudaMalloc(&dX, nF);
cudaMalloc(&dY, nF);
// Copy Device->Host
cudaMemcpy(dX, x.data(), nF, cudaMemcpyHostToDevice);
cudaMemcpy(dY, y.data(), nF, cudaMemcpyHostToDevice);
// Run the kernel
add<<<numBlocks, blockSize>>>(n, x, y);
// Copy Host->Device
cudaMemcpy(y.data(), dY, nF, cudaMemcpyDeviceToHost);
// Free memory
cudaFree(dX); cudaFree(dY);
```

Third alternative: use *CUDA streams* and **cudaMemcpyAsync()**.

Compiling and linking C++ code

Compile to executable (**.exe** file on Windows) :

gcc:

g++ -o myprog main.cpp file1.cpp file2.cpp

clang (LLVM):

clang -o myprog main.cpp file1.cpp file2.cpp

cl (Microsoft):

cl /o myprog main.cpp file1.cpp file2.cpp

Where: **myprog** = name of the executable

main.cpp file1.cpp file2.cpp = C++ source files

Compile ONE source file to the *object* file (**.o** or **.obj**) :

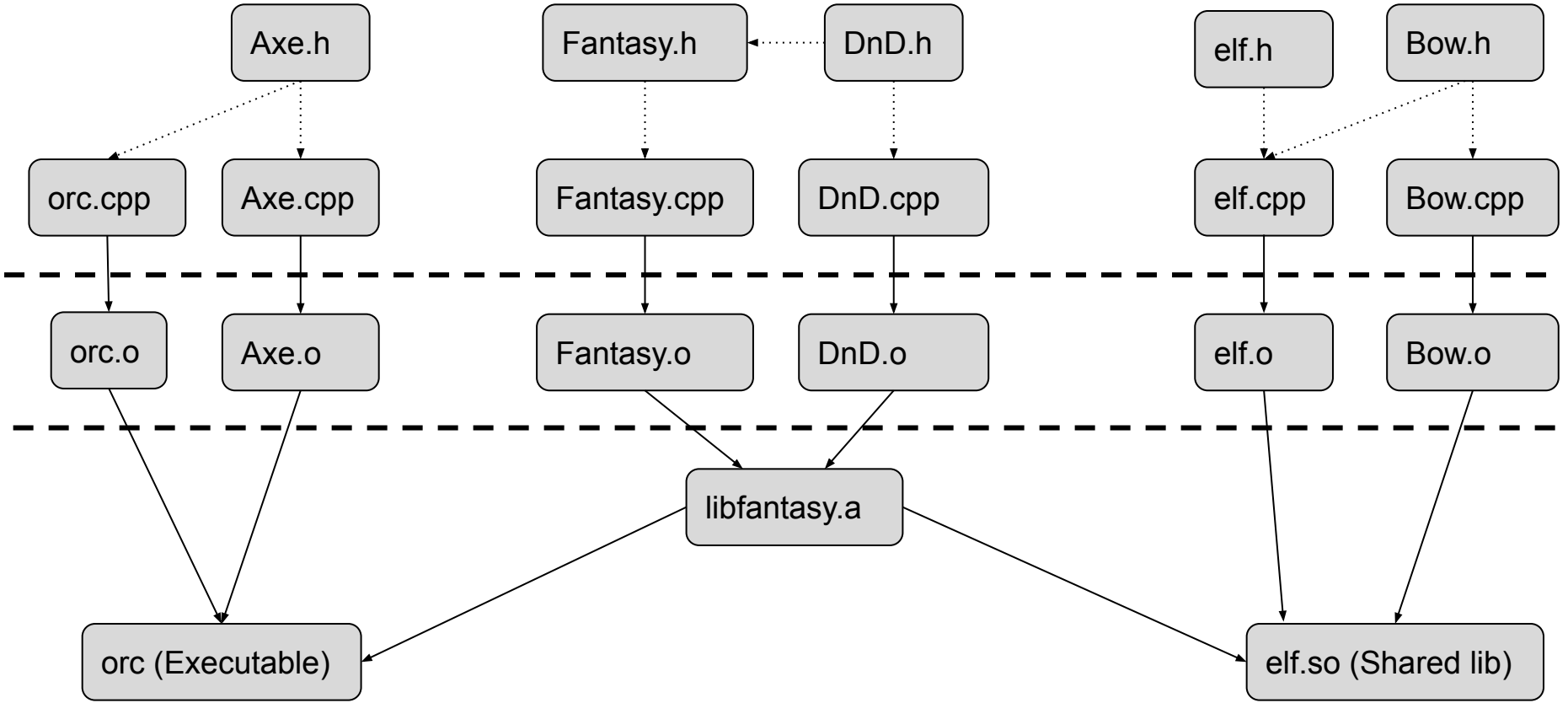
g++ -c main.cpp

...

Link *object* files (and *static libraries* **.a**/**.lib**) to executable (or shared library **.so**/**.dll**):

g++ -o myprog main.o file1.o file2.o

Build process : The code is built from *.cpp and *.h (header) files



Low-level build systems for C/C++: make, nmake, ninja, ...

Unix/Linux and MinGW/Msys use **make**. Note: **nmake** = exotic Microsoft fork of **make**.
Project file: **Makefile** . To build a project:

make	Unix/Linux
mingw32-make	MinGW (Both 32 and 64 bit)

More options:

make clean	Clean project (delete executables and **.o**)
make hello	Build target **hello**
make all	Build all targets
make -f Makefile2 -j3	Build file **Makefile2** on 3 cores

Compile and install software on Unix/Linux (ffmpeg etc.) :

./configure **<options>**
make
make install

Extra slides: Writing Makefile

Why make is not enough?

1. Different incompatible versions (**gmake, remake, mingw32-make, nmake ...**)
2. No configuration/library finding abilities. Extensions (Unix/Linux mostly):

pkgconfig Find a package (headers + libraries), used by **make**

GNU Build System (autotools) : **autoconf, automake, libtool, gnulib**

autoconf Generate **./configure** script

automake Generate **Makefile.in** (used by **./configure**)

3. This is not very popular (or convenient) on Windows
4. Compiler dependent, works best for **gcc** and Linux/Unix
5. Outdated and can be difficult to use. Mostly used for C, not C++.

CMake (since 2000) : Alternative to GNU Build System for **C, C++, Fortran, Assembler, ...**

1. Generates projects for **make, nmake, ninja, Code::Blocks, XCode, Visual Studio, ...**
2. Cross platform, including Windows + Microsoft Compiler and Android NDK (**ninja**).
3. Package and library search.
4. Powerful, with (relatively) easy-to-use syntax (and *awful* official docs/tutorials).

Simplest cmake projects

My first CMake project (**CMakeLists.txt**)

```
add_executable(hello hello.cpp)
```

My second CMake project

```
# This is a comment
cmake_minimum_required(VERSION 3.1)

project(hello)

set(CMAKE_CXX_STANDARD 14)

set(SRCS
#    somefile.h somefile.cpp
     hello.cpp
)

add_executable(${PROJECT_NAME} ${SRCS})
```

How to build a CMake project ?

```
mkdir build
cd build
cmake ..
cmake --build .
```

Rebuild after you have edited some source files ...

```
cmake --build .
```

Using *generators* (Example: Windows, MinGW):

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
cmake --build .
```

CMake does not call the C++ compiler directly.

Generators use low-level build systems (**make**, **nmake**, **ninja, ...**)

and IDEs (Visual Studio, Code.Blocks, xcode)

Configuring cmake project:

We run (in the directory **build**):

cmake -G ”MinGW Makefiles” ..

Configure project using the generator **MinGW Makefiles**.

.. = path to the directory with the file **CMakeLists.txt**

Directory **build** :

CMakeFiles

cmake_install.cmake

CMakeCache.txt Project configuration (can be edited)

Makefile Project file for **make**

Directory **build/CMakeFiles** :

...

hello.dir Configuration/build directory for target **hello**

...

Building

We run (in the directory **build**):
cmake --build .
Build the project in the directory **.** (current directory) using e.g. **make**

New files in **build** :
hello (or **hello.exe** in Windows) : executable
New files in **build/CMakeFiles/hello.dir** :
hello.cpp.obj Object file
objects.a Object files packed as a static lib

Build commands:
make or **mingw32-make** or **nmake** or **ninja** Build project
cmake --build . --target hello -- -j2 Build target **hello** on 2 cores (**make** flag **-j2**)
cmake --build . --target clean
cmake --build . --target install

CMake language (example lang)

set() : Create/Assign CMake variable:

```
set(CMAKE_CXX_STANDARD 14)
```

Variable value: **${CMAKE_CXX_STANDARD}**

message() : Print string or variable

```
message("Hello world !")
message("PROJECT_NAME = ${PROJECT_NAME}")
message("CMAKE_CXX_STANDARD = ${CMAKE_CXX_STANDARD}")
message(${CMAKE_CXX_STANDARD})
```

math() : Evaluate a mathematical expression, put result to **c** :

```
math(EXPR c "5*(10+13) + 7")
message("5*(10+13) + 7 = ${c}")
```

Important ! CMake script is executed at the configure (**cmake ..**) stage, NOT at the build stage!

if(), while()

if() statement :

```
set(n 15)
if(n GREATER 10)
        message("${n} > 10")
else()
        message("${n} < 10")
endif()
```

while() statement :

```
set(n 1)
while(n LESS_EQUAL 10)
        message(${n})
        math(EXPR n "${n}+1")
endwhile()
```

CMake standard variables

```
message("CMAKE_BINARY_DIR = ${CMAKE_BINARY_DIR}")
message("CMAKE_SOURCE_DIR = ${CMAKE_SOURCE_DIR}")
message("CMAKE_BUILD_TYPE = ${CMAKE_BUILD_TYPE}")
message("CMAKE_CXX_FLAGS = ${CMAKE_CXX_FLAGS}")
message("CMAKE_CXX_FLAGS_DEBUG = ${CMAKE_CXX_FLAGS_DEBUG}")
message("CMAKE_CXX_FLAGS_RELEASE = ${CMAKE_CXX_FLAGS_RELEASE}")
message("CMAKE_EXECUTABLE_SUFFIX = ${CMAKE_EXECUTABLE_SUFFIX}")
message("CMAKE_SYSTEM = ${CMAKE_SYSTEM}")
message("CMAKE_SYSTEM_NAME = ${CMAKE_SYSTEM_NAME}")
message("CMAKE_SIZEOF_VOID_P = ${CMAKE_SIZEOF_VOID_P}")   # Size of void *
message("WIN32 = ${WIN32}")
message("APPLE = ${APPLE}")
message("UNIX = ${UNIX}")
message("MINGW = ${MINGW}")

if (WIN32)
    message("Windows !!!")
else()
    message("NOT Windows !!!")
endif()
```

CMake standard variables (MinGW example)

```
Useful CMake variables :
CMAKE_BINARY_DIR = D:/alex/w/cpp-course/examples_11/lang/build
CMAKE_SOURCE_DIR = D:/alex/w/cpp-course/examples_11/lang
CMAKE_BUILD_TYPE =
CMAKE_CXX_FLAGS =
CMAKE_CXX_FLAGS_DEBUG = -g
CMAKE_CXX_FLAGS_RELEASE = -O3 -DNDEBUG
CMAKE_EXECUTABLE_SUFFIX = .exe
CMAKE_SYSTEM = Windows-10.0.15063
CMAKE_SYSTEM_NAME = Windows
CMAKE_SIZEOF_VOID_P = 8
WIN32 = 1
APPLE =
UNIX =
MINGW = 1
Windows !!!
```

Executables and libraries :

Build an executable :

```
add_executable(<target> <sources>)
add_executable(hello hello.cpp)                    # Build hello from hello.cpp
add_executable(${PROJECT_NAME} ${SRCS})            # The same with variables
```

Specify libraries needed by a target :

```
target_link_libraries(<target> <libraries>)
target_link_libraries(${PROJECT_NAME} a b)
```

Build a library :

```
add_library(<target> [STATIC|SHARED] <libraries>)
add_library(b ${SRCS_B})                           # Default (static?) library
add_library(b STATIC ${SRCS_B})                    # Static library  .a/.lib
add_library(b SHARED ${SRCS_B})                    # Shared (dynamic) library   .so/.dll
```

Include a subdirectory (with its own **CMakeLists.txt**) :

```
add_subdirectory(liba)
```

Example lib : hello + 2 libraries : Main CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1)
project(hello)
set(CMAKE_CXX_STANDARD 14)

# Build library a from the separate directory liba
add_subdirectory(liba)
include_directories(liba)           # Search for header files (a.h) in liba

# Build SHARED library b : in this directory
set(SRCS_B B.cpp)
add_library(b SHARED ${SRCS_B})

# Build hello
set(SRCS_HELLO main.cpp)
add_executable(${PROJECT_NAME} ${SRCS_HELLO})
target_link_libraries(${PROJECT_NAME} a b)
```

Here **hello** and **b** are built in the same directory.

It's better to put every target to a separate directory (like **liba** for the library **a**).

Example lib : CMakeLists.txt in subdirectory liba

```
cmake_minimum_required(VERSION 3.1)
project(a)
set(CMAKE_CXX_STANDARD 14)

set(SRCS
    a.cpp
)
add_library(${PROJECT_NAME} STATIC ${SRCS})    #  Static library a
```

Files after build in the directory **build** (MinGW) :

hello.exe	Executable
libb.dll	Shared library **b**
libb.dll.a	Import lib for **libb.dll** (Used for **.dll**, not **.so** !)
CMakeFiles/b.dir/	
CMakeFiles/hello.dir/	
liba/liba.a	Static library **a**
liba/CMakeFiles/a.dir/	

Configuring a CMake project

Build types (**CMAKE_BUILD_TYPE**) : **Release, Debug, MinSizeRel, RelWithDebInfo**

Warning ! This is ignored by Visual C++ ! For VC++, choose build type at the build stage!

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Set default build type in **CMakeLists.txt** :

```
if(NOT CMAKE_BUILD_TYPE )
  set( CMAKE_BUILD_TYPE "Release" )
endif()
```

Configuring parameters:

```
cmake -DWITH_MAGIC=YES ..
```

option() : declare boolean parameters (with description+default !) in **CMakeLists.txt**:

```
option(WITH_DRAGONS "Any dragons in our story ?" OFF)
option(WITH_ELVES "Any elves in our story ?" OFF)
option(WITH_ORCS "Any orcs in our story ?" ON)
```

Configuring options:

```
cmake -DWITH_ELVES=ON ..
```

Passing variables to C++

Using **add_definition() :**

```
if(DEFINED WITH_DRAGONS)
      add_definitions(-DWITH_DRAGONS=${WITH_DRAGONS})
endif()
```

Using **configure_file() :**

```
configure_file(config.h.in config.h)
include_directories("${PROJECT_BINARY_DIR}")    # To find config.h
```

File **config.h.in** :

```
#cmakedefine WITH_MAGIC   @WITH_MAGIC@      // Defined only if NOT OFF/0/NO
#define WITH_ELVES   @WITH_ELVES@          // Defined always
#cmakedefine01 WITH_ORCS
```

File **config.h** :

```
#define WITH_MAGIC  YES           // Defined only if NOT OFF/0/NO
#define WITH_ELVES  ON               // Defined always
#define WITH_ORCS 0
```

The complete example (vars) CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1)
project(hello)
set(CMAKE_CXX_STANDARD 14)
# Options
option(WITH_DRAGONS "Any dragons in our story ?" OFF)
option(WITH_ELVES "Any elves in our story ?" OFF)
option(WITH_ORCS "Any orcs in our story ?" ON)
message("WITH_MAGIC = ${WITH_MAGIC}")    # And all others
..
# Pass definition to C++ using add_definition()
if(DEFINED WITH_DRAGONS)
 add_definitions(-DWITH_DRAGONS=${WITH_DRAGONS})
endif()
# Pass definitions to C++ using configure_file()
configure_file(config.h.in config.h)
include_directories("${PROJECT_BINARY_DIR}")     # To find config.h
# Build executable hello
set(SRCS  hello.cpp)
add_executable(${PROJECT_NAME} ${SRCS})
```

Multiple parameters?

Do we really have to write ?

cmake -G ”MinGW Makefiles” -DCMAKE_BUILD_TYPE=Debug -DWITH_MAGIC=YES -DWITH_ELVES=ON -DWITH_DRAGONS=ON -DWITH_ORCS=OFF ..

No, we can (usually) work in steps :

cmake -G ”MinGW Makefiles” ..
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake -DWITH_MAGIC=YES ..
cmake -DWITH_ELVES=ON ..
cmake -DWITH_DRAGONS=ON ..
cmake -DWITH_ORCS=OFF ..

Note: "**cmake ..**" does not reset to defaults, it keeps all options unchanged !

CMake stores variables in **CMakeCache.txt**

Reset to defaults: Delete **CMakeCache.txt**, then run "**cmake ..**" with no options.

Extra slides: CMake cache

Finding external libraries in CMake : 1. The CMake way

```
cmake_minimum_required (VERSION 3.0)
project( grabcam )
set (CMAKE_CXX_STANDARD 14)

find_package( OpenCV REQUIRED )
include_directories( ${OpenCV_INCLUDE_DIRS} )
add_executable( grabcam grabcam.cpp )
target_link_libraries( grabcam ${OpenCV_LIBS} )
```

It finds OpenCV package installed at the standard locations only.
Otherwise you must specify OpenCV directory:
cmake -DOpenCV_DIR=/home/seymour/opencv/411/lib/cmake/opencv4 ..
This must be the directory with ***.cmake** files !

Alternatively, you can specify CMake package search path (works as a filesystem root !):
cmake -DCMAKE_PREFIX_PATH=/home/seymour/opencv/411 ..

Finding external libraries in CMake : 2. find_library()

Standard libraries of Linux/MinGW can be found simply by name (without *lib*- prefix):

```
target_link_libraries(dijkdemo gdi32 png)        # Needed by CImg
```

Alternative names can be checked by **find_library()**:

```
find_library(GLFW_LIB NAMES glfw glfw3)
find_library(GLEW_LIB NAMES glew GLEW glew32)
find_package(OpenGL REQUIRED)
target_link_libraries(triangle ${GLEW_LIB} ${GLFW_LIB} ${OPENGL_gl_LIBRARY})
```

How to enable C++ threads (links thread library on Unix/Linux)?
Don't try to put "**-pthread**" anywhere. The proper cross-platform CMake way is:

```
find_package(Threads)
target_link_libraries(${PROJECT_NAME} ${CMAKE_THREAD_LIBS_INIT})
```

Windows (including MinGW): do nothing. Unix/Linux: Add "**-pthread**" to the linker flags.

Finding external libraries in CMake : 3. pkg-config

```
cmake_minimum_required(VERSION 3.1)
project(hello)
set(CMAKE_CXX_STANDARD 14)

# gtkmm libraries
find_package(PkgConfig)        # Find pkg-config CMake plugin
pkg_check_modules(GTKMM gtkmm-3.0)
link_directories(${GTKMM_LIBRARY_DIRS})
include_directories(${GTKMM_INCLUDE_DIRS})

add_executable(${PROJECT_NAME}
   HelloWorld.h
   main.cpp
   )
target_link_libraries(${PROJECT_NAME} ${GTKMM_LIBRARIES}   )
```

PkgConfig is a pre-CMake package finder, still popular on Linux/Unix.
It is mostly used for C libraries (e.g. ffmpeg), but also for gtkmm.
Tricky for Visual Studio (Install some pkg-config by hand ?).

Brief OpenCV building guide (Linux mostly)

So you want to build OpenCV from the source?
First, you need to download OpenCV as ZIP file (particular version), or clone from github.

Then configure the build:
mkdir build; cd build
cmake ..
You see the detailed opencv build configuration. Check everything!
Check that you have found : ffmpeg (for video), jpeg+png+tiff+..., gtk+ (for imshow).
Build debug (release is default): **cmake -DCMAKE_BUILD_TYPE=Debug ..**
Build static libs (shared is default): **cmake -DBUILD_SHARED_LIBS=OFF ..**
Build with Qt backend (gtk+ is default): **cmake -DWITH_QT=ON ..**

Finally, build OpenCV (**-j4** = build in 4 threads):
make -j4
sudo make install
Right? **!!! WRONG !!!**

"sudo make install" is evil!

- It writes stuff to *system directories*! Requires root access.
- It is difficult to undo (**sudo make uninstall**, but only if you keep the build dir !).
- Can even overwrite files of DEB (or RPM) packages!

Solution 1: Use *prefixes* (recommended, does not require root access).

```
mkdir ~/opencv
cmake -DCMAKE_INSTALL_PREFIX=~/opencv ..
make -j4
make install
```

Solution 2: Create your own DEB package and install (Debian/Ubuntu, very robust).

```
cmake ..
make -j4
sudo checkinstall
```

Solution 3: Use CPack to create a package, then install (Not always works).

Building OpenCV with contrib (+nonfree algos aka SIFT+SURF)

1. Download both opencv and opencv_contrib from github.
   git clone git@github.com:opencv/opencv.git
   git clone git@github.com:opencv/opencv_contrib.git
2. Choose the same version (or latest master) for both, important!
   cd opencv;git checkout 4.2.0
   cd ../opencv_contrib;git checkout 4.2.0
3. Configure and build (**make** takes forever ! Especially on Raspberry Pi 3):
   cd ../opencv;mkdir build;cd build
   **cmake -DCMAKE_INSTALL_PREFIX=~/opencv_full **
   **-DOPENCV_ENABLE_NONFREE=ON **
   -DOPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules ..
   make -j4
   make install
4. To build only some modules, copy selected modules from **opencv_contrib/modules** to some other location. Then specify that directory in **-DOPENCV_EXTRA_MODULES_PATH**.

Building OpenCV with CUDA (and contrib)

- CUDA is in contrib (since OpenCV 4.0.0)
- CMake must be able to find CUDA
- Needs older gcc, particular version depends on the CUDA version!
- Don't worry, compiled OpenCV will still work with your gcc-9 (hopefully).
- Compiled libraries will NOT work without CUDA or on a different CUDA version!
- OpenCV team hates CUDA, prefers OpenCL.

```
cmake -DCMAKE_INSTALL_PREFIX=/home/seymour/opencv_cuda  \
      -DOPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules \
      -DCMAKE_C_COMPILER=/usr/bin/gcc-7 \
      -DCMAKE_CXX_COMPILER=/usr/bin/g++-7 \
      -DWITH_CUDA=ON \
      -DWITH_CUBLAS=ON \
      -DOPENCV_ENABLE_NONFREE=ON ..
```

configure+make example: building ffmpeg from the source

First, install many external libraries (dev-versions):

sudo apt install libass-dev libaom-dev ...

Then, configure and build ffmpeg (and tell where to place **.pc** files for PkgConfig):

```
mkdir ~/ffmpeg
PKG_CONFIG_PATH=/home/seymour/ffmpeg/lib/pkgconfig/ ./configure \
  --prefix=${HOME}/ffmpeg \
  --enable-gpl --enable-nonfree \
  --enable-libass  --enable-libaom \
  --enable-libfdk-aac --enable-libfreetype \
  --enable-libmp3lame --enable-libopus \
  --enable-libvorbis --enable-libvpx \
  --enable-libx264  --enable-libx265
make -j4
make install
```

Options to enable nvidia GPU codecs:

--enable-cuda --enable-cuvid --enable-nvdec --enable-nvenc --enable-libnpp

Thank you for your attention !

title

text

Makefile example (Makefile2)

```
CXX = g++
CXXFLAGS = -Wall -g
all: example
example : main.o a.o B.o
<tab>$(CXX) $(CXXFLAGS) -o $@ $^
main.o : main.cpp a.h B.h
<tab>$(CXX) $(CXXFLAGS) -c $<
a.o : a.cpp a.h
<tab>$(CXX) $(CXXFLAGS) -c $<
B.o : B.cpp B.h
<tab>$(CXX) $(CXXFLAGS) -c $<
clean:
<tab>-rm *.o example
<tab>-del *.o example.exe
```

Makefile targets

make targets:

target : dependencies

<tab>command

main.o : main.cpp a.h B.h

<tab>$(CXX) $(CXXFLAGS) -c $<

make variables:

CXX = g++

CXXFLAGS = -Wall -g

$(CXX) $(CXXFLAGS) -c $<

Special variables:

$@ : target name

$^ : All dependencies

$< : First dependency

Makefile with rules (Makefile)

```
CXX = g++
CXXFLAGS = -Wall -g
LIBS =
OBJS = main.o a.o B.o
DEPS = a.h B.h
.PHONY: all
all: example
example : $(OBJS)
<tab>$(CXX) $(CXXFLAGS) -o $@ $^ $(LIBS)
%.o : %.cpp $(DEPS)
<tab>$(CXX) $(CXXFLAGS) -o $@ -c $<
.PHONY: clean
clean:
<tab>-rm *.o example
<tab>-del *.o example.exe
```

CMake : How it works

Project directory (folder) :
CMakeLists.txt
hello.cpp

We run:
mkdir build
cd build

Directory **build** appears:
build
CMakeLists.txt
hello.cpp

All build process takes place in the directory **build** !

Useful external libraries and APIs (C++/C) :

1. GUI : **Qt, gtkmm**
2. Video/Audio processing : **ffmpeg (C), gstreamer (C), openmax (C)**
3. Image processing : **CImg, OpenCV**
4. Cross-platform TCP/UDP : **Boost.Asio**
5. Unit tests : **CppUnit, CppTest, Google Test, Boost**
6. 3D graphics : **OpenGL (C) + glew/glad/epoxy, glfw, glm**

CMakeCache.txt

```
//Any dragons in our story ?
WITH_DRAGONS:BOOL=ON

//Any elves in our story ?
WITH_ELVES:BOOL=ON

//No help, variable specified on the command line.
WITH_MAGIC:UNINITIALIZED=YES

//Any orcs in our story ?
WITH_ORCS:BOOL=OFF
```

To view CMake cache :

cmake -L ..