CMake

~ A Build System for Build Systems ~

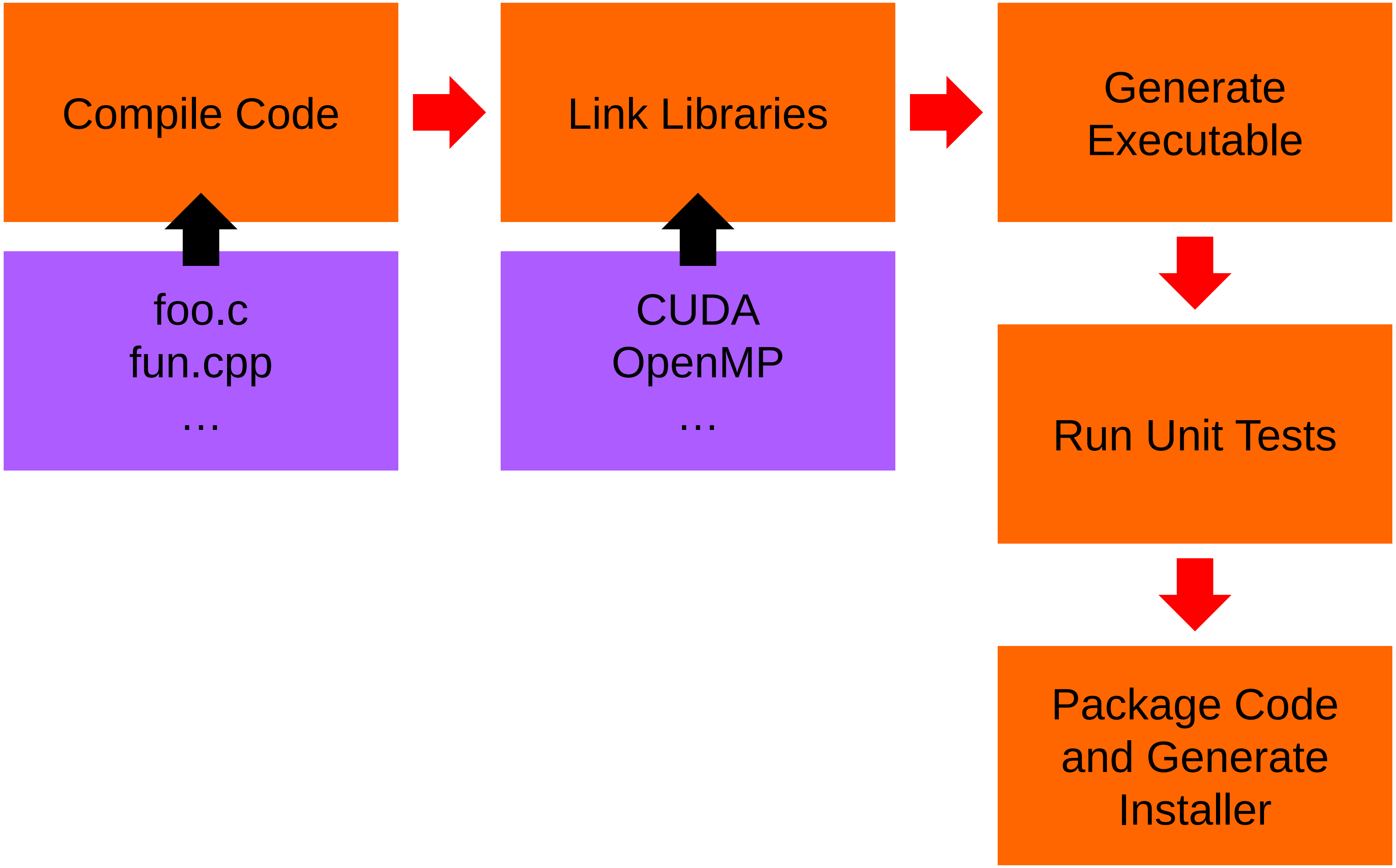
Motivating Questions

- How can we easily build, test and package software?

- What if our software needs to run on multiple platforms?
  - Linux, Windows, MacOS, etc.

- Dealing w/ multiple platforms not easy
  - How do I find all the libraries I need?
  - What compiler flags do I use?
  - Etc.

Build-Test-Package Cycle



Compile Code

foo.c
fun.cpp
…

Link Libraries

CUDA
OpenMP
…

Generate Executable

Run Unit Tests

Package Code and Generate Installer

A Few Build Systems...

- Using `make` via `makefile`[s]
  - Hand-written
  - Portability depends on author

- Autotools (GNU build system)
  - Most familiar: `./configure` && `make` && `make install`
  - There’s more to it though: `aclocal`, `autoheader`, `automake`, `autoconf`,…
  - Require Cygwin or MSYS for Windows

- Eclipse, Visual Studio
  - Solution specific to the IDE
  - Yields a complex setup for large projects

…and Two More

- SCons
  - Builds defined as Python scripts
  - Used by Blender, Doom3, NumPy, SciPy
- CMake
  - Can generate Eclipse projects, Visual Studio solutions, Makefiles, XCode projects, etc.
  - Used in ME759

Should I Bother?

- You’ll use `CMake` in your ME759 assignments
  - Perhaps you’d like to work on your assignment on your Windows laptop and then at the end ensure your solution runs OK on Euler
    - You’d build under Windows
    - We check your homework under Linux

Intro to CMake

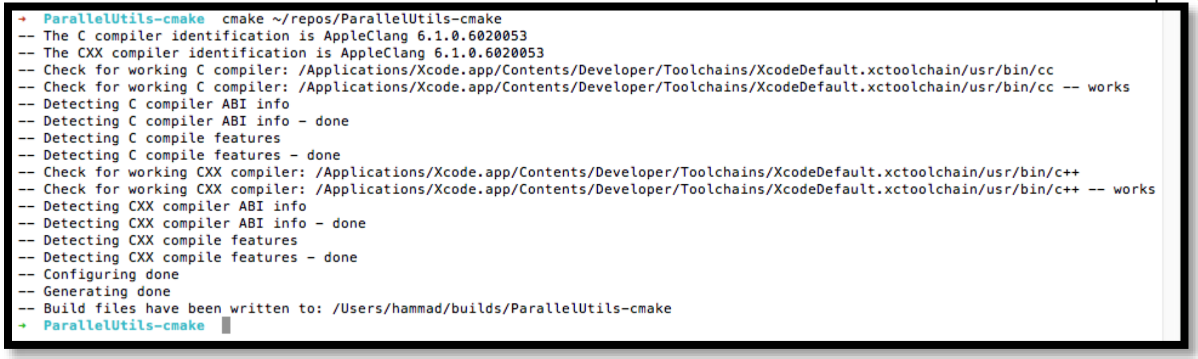
- Projects are defined via simple text files
  - Easy to diff
  - Easy to maintain under revision control (SVN, Mercurial, Git, etc.)
  - No more digging through stacks of config dialogs
  - Works on any platform (Linux, Windows, OSX)
- User-configurable options set in the `ccmake`/`cmake-gui` programs
- Once configured, project files are generated for your system's native build environment (Eclipse, Visual Studio, Makefiles, Xcode, etc.)

CMake Lingo

- CMakeLists.txt
  - Text file in which you set up variables/commands that will dictate the behavior of `CMake` in its process of producing projects/solutions
- Generate
  - The process of reading `CMakeLists.txt` and producing a project file for your IDE
- Cache
  - Stores environment-specific and user-configurable options
- Build type
  - Set of compiler/linker options
  - Some predefined setups:
    - debug, release, release with debug symbols, space-optimized release, etc.

CMake Configuration Options

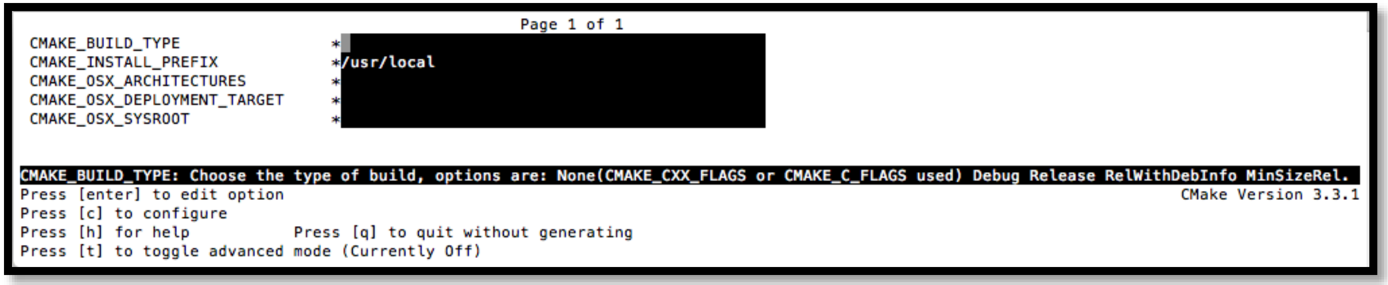
- “cmake”



```
➜  ParallelUtils-cmake  cmake ~/repos/ParallelUtils-cmake
-- The C compiler identification is AppleClang 6.1.0.6020053
-- The CXX compiler identification is AppleClang 6.1.0.6020053
-- Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc
-- Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++
-- Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/hammad/builds/ParallelUtils-cmake
➜  ParallelUtils-cmake
```

- “ccmake”

Use on Euler



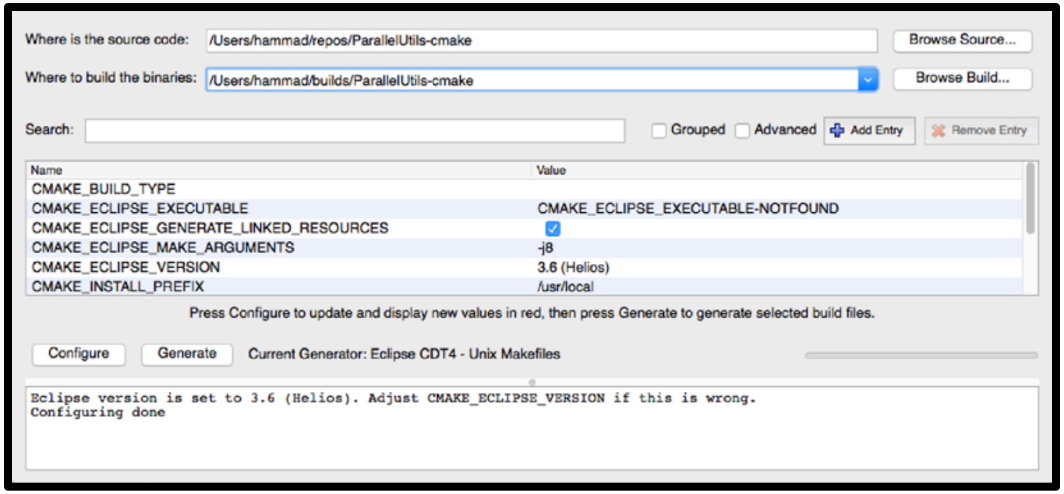
```
                                                     Page 1 of 1
 CMAKE_BUILD_TYPE                *
 CMAKE_INSTALL_PREFIX            */usr/local
 CMAKE_OSX_ARCHITECTURES         *
 CMAKE_OSX_DEPLOYMENT_TARGET     *
 CMAKE_OSX_SYSROOT               *



CMAKE_BUILD_TYPE: Choose the type of build, options are: None(CMAKE_CXX_FLAGS or CMAKE_C_FLAGS used) Debug Release RelWithDebInfo MinSizeRel.
Press [enter] to edit option                                                                                          CMake Version 3.3.1
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

- “cmake-gui”

Use on Windows



Where is the source code: /Users/hammad/repos/ParallelUtils-cmake

Where to build the binaries: /Users/hammad/builds/ParallelUtils-cmake

Search: □ Grouped □ Advanced

Name	Value
CMAKE_BUILD_TYPE	
CMAKE_ECLIPSE_EXECUTABLE	CMAKE_ECLIPSE_EXECUTABLE-NOTFOUND
CMAKE_ECLIPSE_GENERATE_LINKED_RESOURCES	☑
CMAKE_ECLIPSE_MAKE_ARGUMENTS	-j8
CMAKE_ECLIPSE_VERSION	3.6 (Helios)
CMAKE_INSTALL_PREFIX	/usr/local

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Eclipse CDT4 - Unix Makefiles

```
Eclipse version is set to 3.6 (Helios). Adjust CMAKE_ECLIPSE_VERSION if this is wrong.
Configuring done
```

CMake Workflow

1. Write `CMakeLists.txt` file[s]

2. Select build directory in `cmake-gui`

3. Choose target according to your environment
   - Eclipse, Visual Studio, `makefiles`, etc.

4. Configure project options
   - These stay persistent, saved in cache

5. Generate project files

6. Build project (in Visual Studio, for instance – compile and link, that is)

The CMakeLists.txt File

- variables/commands that dictate behavior when you generate project/solution files
- Watch out: name must be **exactly** CMakeLists.txt
- Contents themselves are case insensitive
  - But **be consistent**
  - Commonly found in recent projects:
    - functions()
    - VARIABLES
- 20/80 rule: 20% of commands do 80% of what you'll need
- Documentation (CMake 3.9):
  - https://cmake.org/cmake/help/v3.9/

add_custom_command
add_custom_target
add_definitions
add_dependencies
add_executable
add_library
add_subdirectory
break
cmake_policy
configure_file
else
elseif
endforeach
endfunction
endif
endmacro
endwhile
execute_process
export
file
find_file
foreach
function
if
include
include_directories
install
link_directories
macro
message
option
project
return
set
string
target_link_libraries
while
add_custom_command

CMakeLists.txt:
A Few Other Functions

- `configure_file`: do a find/replace on files
- `ExternalProject`: require an external project to be built before building your own
- `find_package(foo)`: see if package foo is available on this system
  - This makes setting up CUDA and MPI relatively painless
  - But, `FindFoo.cmake` script must already be written
- `math`: perform arbitrary math operations
- `{add,remove}_definitions`: set/remove preprocessor definitions

Basic CMakeLists.txt

```
# Set the required version of CMake
cmake_minimum_required(VERSION 2.8)

# Set your project title
project(ME759)

# Look for CUDA and set up the build environment
# Flag ‘REQUIRED’ forces us to set up CUDA correctly before building
find_package(“CUDA” REQUIRED)

# Finally, we would like to create a program ‘foo’
# from the files ‘foo.cu’ and ‘bar.cu’
# Using cuda_add_executable tells CMake to use with nvcc instead of gcc
cuda_add_executable(foo foo.cu bar.cu)
```

CMake for ME759

- A template available at https://github.com/uwsbel/ParallelUtils-cmake

- Has macros for CUDA, MPI, and OpenMP projects
  - To use:
    - Copy to your source directory
    - Uncomment relevant sections of CMakeLists.txt
    - Modify for your assignments

- Useful command: **`add_subdirectory`**
  - Allows you to have a single main CMakeLists.txt with assignment-specific ones in subdirectories

CMakeLists.txt from Template

```
# Minimum version of CMake required. Don't touch.
cmake_minimum_required(VERSION 2.8)

# Set the name of your project
project(ME759)

# Include macros from the SBEL utils library
Include(ParallelUtils.cmake)

## Example CUDA program
enable_cuda_support()
cuda_add_executable(bandwidthTest bandwidthTest.cu)
```

What This Shows...

- Including commands from another file
- Running a macro (no arguments)
- Adding a CUDA executable to build
- **`ParallelUtils.cmake`** has more, see comments

In-source v. Out-of-source Builds

- In-source builds
  - Binaries & project files generated alongside source code
  - Need to pay attention if using version control
  - IDEs (Eclipse) prefer this method
    - See http://www.cmake.org/Wiki/Eclipse_CDT4_Generator
- Out-of-source builds
  - Binaries & project files in separate directory
  - Easy to clean – just delete it
  - Only need to **`checkin/commit`** the source directory
  - This is the recommended way to build your code
    - For instance, it allows you to have at the same time two version of the same executable – one release and one debug

cmake-gui

- User-configurable options are set here

- Set source and build directories
  - Must decide between in-source v. out-of-source build
- New build dir/cleared cache: nothing there
  - Hit ‘Configure’ to select generator & start configuring
- New/changed options are shown in red
  - Modify if need be, then keep hitting configure until done
- ‘Generate’ creates the project files
- Feel free to venture into ‘Advanced’ options
  - Can manually set compiler/linker options here
  - Remember this: do a “**`File > Delete Cache`**” if something gets messed up

cmake-gui gotchas

- If you rely on a library/path/variable, make sure it is found
  - Will show up as **`{FOO}_NOT_FOUND`** in the config options
  - Can be manually set if need be
    - But you should probably first determine why it’s not being done automatically
- Option not showing up? Hit Configure again, check advanced
- Strange issues? Clear the cache
  - Similar to “**`make distclean`**”

Using Projects, Compiling

- After generating the project files, open in your IDE
  - Eclipse: **`File > Import Project`**
  - Visual Studio: open the solution (double click the `sln` file)
  - Makefiles/Eclipse: **`make`** (**`make –j4`** for parallel build w/ 4 threads)

- Source code should be in there, even if using out-of-source (linked to the source directory)

- **`CMake`** will automatically run when building to update project/make files
  - No need to open **`cmake-gui`** again unless changing options
  - Visual Studio may ask to reload the project; do it, if prompted so

Example Directory Structure

- me759_homework/
  - CMakeLists.txt — **Main CMakeLists.txt**
  - homework_01/
    - CMakeLists.txt — **Homework Specific CMakeLists.txt**
    - hw01.cpp
  - homework_02/
    - CMakeLists.txt — **Homework Specific CMakeLists.txt**
    - hw02.cu
  - …

Example, Shows 3 CMakeLists.txt files

Main CMakeLists.txt

```
# Set the required version of CMake
cmake_minimum_required(VERSION 3.9)
# Set your project title
project(ME759_Homework)
# Include macros from the SBEL utils library
Include(ParallelUtils.cmake)
enable_cuda_support()

add_subdirectory(homework_01)
add_subdirectory(homework_02)
…
```

Homework Spefic CMakeLists.txt

```
add_executable(hw01 hw01.cpp)
…
```

Homework Spefic CMakeLists.txt

```
cuda_add_executable(hw02 hw02.cu)
…
```

End Build Tools/Approaches