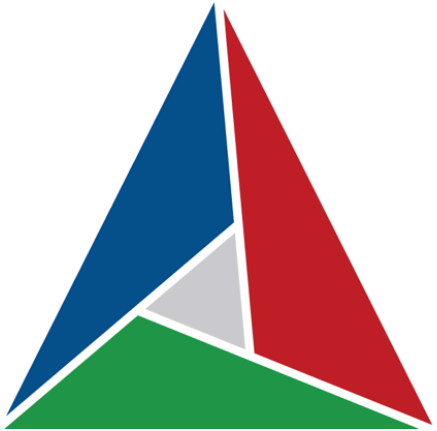


CMake

Exploring Modern CMake + CUDA

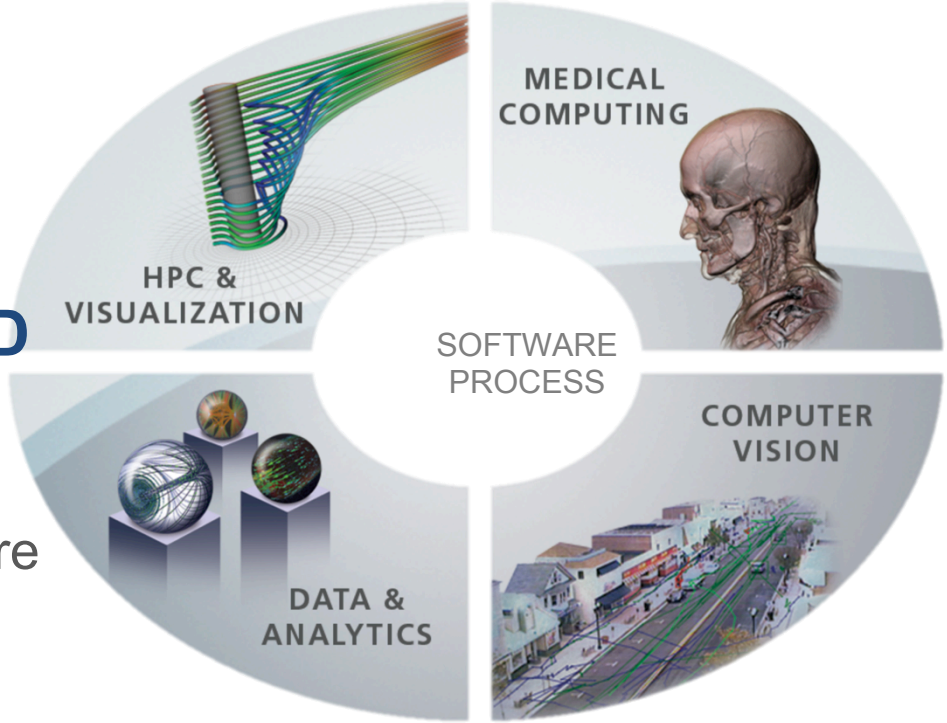
Robert Maynard
Principal Engineer, Kitware

Kitware

Kitware

Collaborative software R&D

- Technical computing
- Algorithms & applications
- Software process & infrastructure
- Support & training
- Open source leadership



Supporting all sectors

- Industry, government & academia


SUMMIT


CHEYENNE

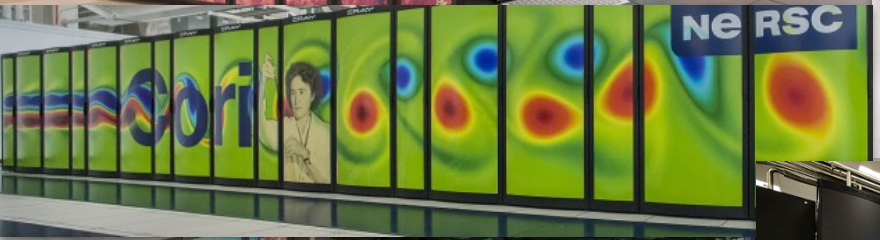

STAMPEDE2


Aurora


CSCS


MIRA

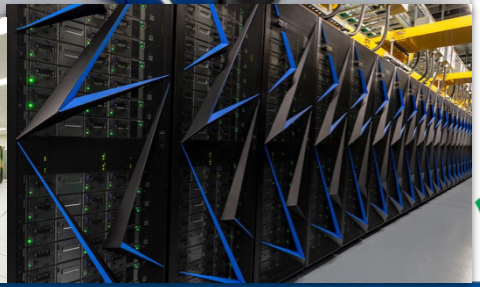

IBM
summit


NERSC
Cori


THETA


TRINITY

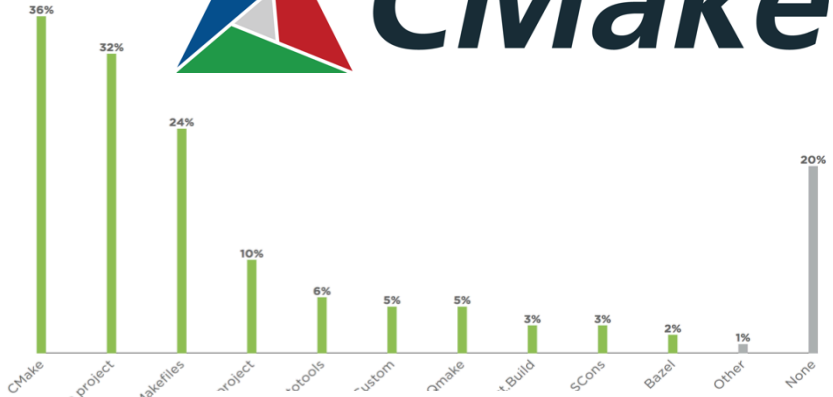

TITAN




VTK


Kitware

CMake



1	Kotlin	2.6x
2	HCL	2.2x
3	TypeScript	1.9x
4	PowerShell	1.7x
5	Rust	1.7x
6	CMake	1.6x
7	Go	1.5x
8	Python	1.5x
9	Groovy	1.4x
10	SQLPL	1.4x

Better IDE integration
- QtCreator
- VisualStudio 2017+

Package Managers
- Spack
- Conan.io
- Microsoft.vckpg

pip install cmake

Continued ‘Modern’ CMake improvements

Native CUDA language support

Quarterly release cycle

Kitware

“Usage Requirements” aka Modern CMake

Modern style: target-centric

```
target_include_directories(example PUBLIC "inc")
```

example and anything that links to gets `-Iinc`

Classic style: directory-centric

```
include_directories("inc")
```

Targets in this directory and subdirs get `-Iinc`

Kitware

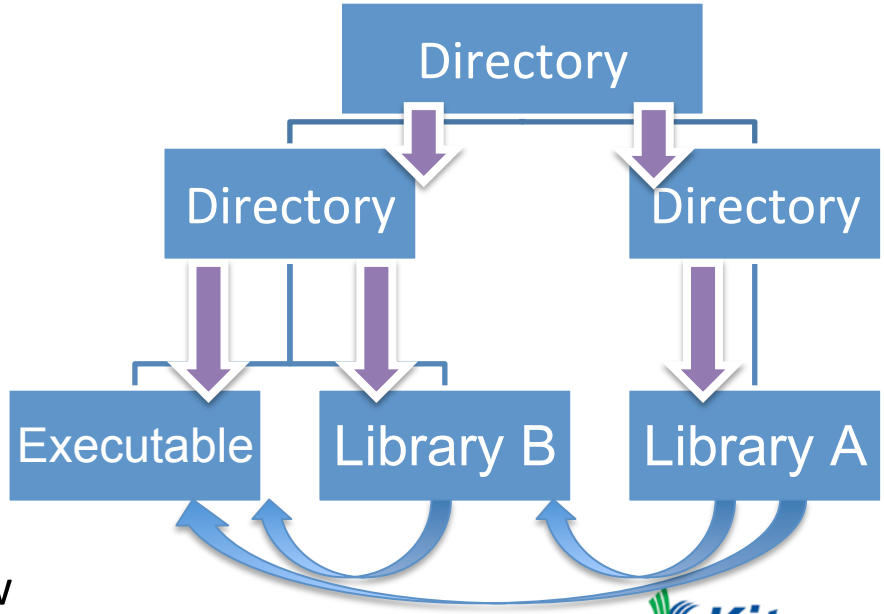
Before Usage Requirements

Before Usage Requirements existed we used directory scoped commands such as:

- `include_directories`
- `compile_definitions`
- `compile_options`

Consumers have to know:

- What dependencies generate build tree files
- What dependencies use any new external packages



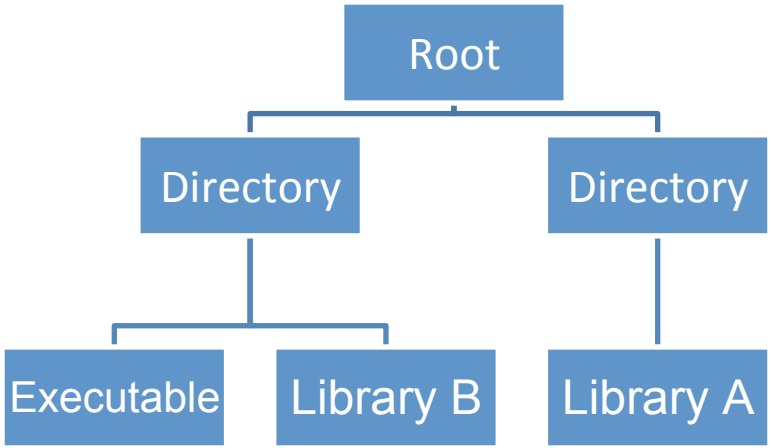
Kitware

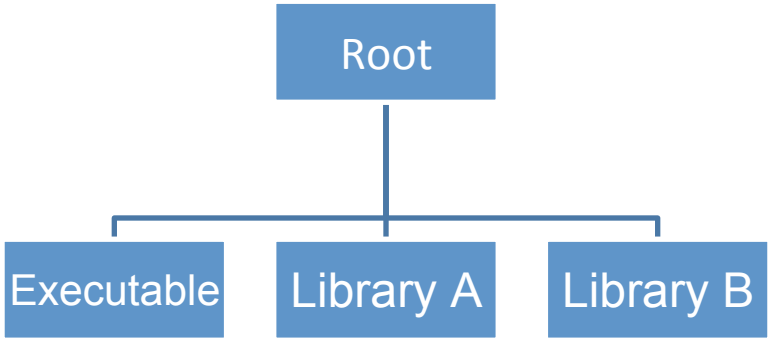
Modern CMake / Usage Requirements

Modern CMake goal is to have each target fully describe how to properly use it.

No difference between using internal and external generated targets

Modern CMake





Kitware

CMake CUDA Support

CUDA has been a first class language in CMake since v3.8

Our goal is to make building CUDA the same as C++

- `add_library`
- `target_link_libraries`

Kitware

Using CMake with CUDA

Declare CUDA as a LANGUAGE in your `project`

```
project(GTC LANGUAGES CUDA CXX)
```

CMake performs configuration checks of your CUDA environment

```
-- Check for working CUDA
compiler: /usr/local/cuda/bin/nvcc
-- works
```

Kitware

Using CMake with CUDA

Optionally enable CUDA

```
project(GTC)
option(GTC_ENABLE_CUDA “Enable CUDA” OFF)
if(GTC_ENABLE_CUDA)
  enable_language(CUDA)
endif()
```

Kitware

Using CMake with CUDA

Optionally enable CUDA

```
project(GTC)
include(CheckLanguage)
check_language(CUDA)
if(CMAKE_CUDA_COMPILER)
  enable_language(CUDA)
endif()
```

Kitware

Mixed Language Libraries

```
add_library(gtc SHARED
  Serial.cpp
  Parallel.cu)
```

Uses the C++ compiler for .cpp and the CUDA compiler for .cu

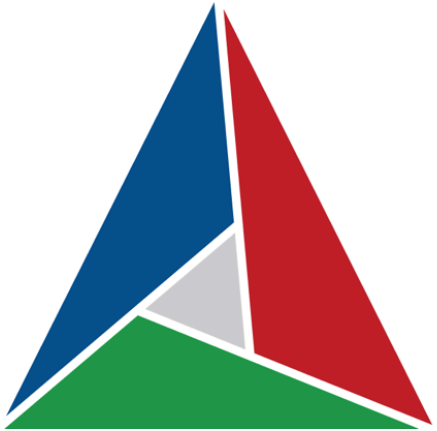
Kitware

Mixed Language Libraries

```
add_library(gtc SHARED
  Serial.cpp
  Parallel.cpp)
set_source_files_properties(Parallel.cpp
  PROPERTIES LANGUAGE CUDA)
```

Uses the CUDA compiler for Parallel.cpp

Kitware



CMake

Time to write code

Kitware

Ground Work

```
cmake_minimum_required(VERSION 3.12...3.14 FATAL_ERROR)
project(GTC)

#options
option(GTC_ENABLE_CUDA “Enable CUDA” OFF)

if(GTC_ENABLE_CUDA)
  enable_language(CUDA)
endif()
```

Kitware

CMake Policies

CMake policies is how CMake implements backward compatibility as a first-class feature

- CMake 3.13 can be used on a project with 2.8.12 as the minimum required version

Policies can also allow forward compatibility

- A project can opt into new behavior by using `cmake_policy`

Allows CMake to correct poor design decisions and bugs that effect backward compatibility

CMake Policies

CMake policies have two states:

- OLD
  - This makes CMake revert to the *old* behavior that existed before the introduction of the policy
- NEW
  - This makes CMake use the *new* behavior that is considered correct and preferred

CMake Policies

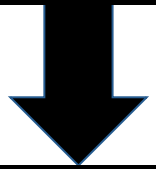
- `cmake_minimum_required` sets all policies newer than the requested version to OLD (OFF)
- The existence of a CMake policy can be queried
- You can explicitly set policies to NEW or OLD with `cmake_policy`

```
cmake_minimum_required(VERSION 3.3 FATAL_ERROR)
if(POLICY CMP0074)
  cmake_policy(SET CMP0074 NEW)
endif()
```

Kitware

CMake 3.12: Easier Policy Control

```
cmake_minimum_required(VERSION 3.12 FATAL_ERROR)
foreach(policy
    CMP0085 # CMake 3.13
    CMP0087 # CMake 3.13
    )
  if(POLICY ${policy})
    cmake_policy(SET ${policy} NEW)
  endif()
endforeach()
```



```
cmake_minimum_required(VERSION 3.12...3.14 FATAL_ERROR)
```

Kitware

Language Level

```
if(GTC_ENABLE_CUDA)
  enable_language(CUDA)
endif()

#-----------------------------------------------------------
add_library(gtc_compiler_flags INTERFACE)
target_compile_features(gtc_compiler_flags
                        INTERFACE cxx_std_11)
set(CMAKE_CXX_EXTENSIONS Off)
```

Kitware

Language Level

```
add_library(gtc_compiler_flags INTERFACE)
target_compile_features(gtc_compiler_flags
                INTERFACE cxx_std_11) # c++11 to cuda also
set(CMAKE_CXX_EXTENSIONS Off)
```

```
set(CMAKE_CXX_STANDARD 11)    # isn't part of the projects
set(CMAKE_CUDA_STANDARD 11)   # export information.
set(CMAKE_CXX_EXTENSIONS Off) # target_compile_features are!
set(CMAKE_CUDA_EXTENSIONS Off)
```

Kitware

Add our Library

```
add_library(gtc_lib STATIC)
target_sources(gtc_lib PRIVATE serial.cxx)
if(GTC_ENABLE_CUDA)
  target_sources(gtc_lib PRIVATE parallel.cu)
endif()

target_link_libraries(gtc_lib PUBLIC gtc_compiler_flags)
target_include_directories(gtc_lib
  PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}
  INTERFACE $<INSTALL_INTERFACE:include/gtc>)
```

Kitware

Usage Requirements

PRIVATE:	Only the given target will use it
INTERFACE:	Only consuming targets use it
PUBLIC:	**PRIVATE + INTERFACE**
$<BUILD_INTERFACE>:	Used by consumers from this project or use the build directory
$<INSTALL_INTERFACE>:	Used by consumers after this target has been installed

Kitware

Usage Requirements

```
target_link_libraries(trunk PUBLIC root)
target_link_libraries(leaf PUBLIC trunk)
```

```
/usr/bin/c++ -fPIC -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o libtrunk.so libroot.so
```

```
target_link_libraries(trunk PRIVATE root)
target_link_libraries(leaf PUBLIC trunk)
```

```
/usr/bin/c++ -fPIC -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o libtrunk.so
```

Kitware

TLL (target link libraries)

- TLL can propagate dependencies when using:
  - `target_include_directories`
  - `target_compile_definitions`
  - `target_compile_options`
  - `target_sources`
  - `target_link_options`

Kitware

Add our Executable

```
add_executable(gtc)
target_sources(gtc PRIVATE main.cxx)
target_link_libraries(gtc PRIVATE gtc_lib)
```

```
c++  -I/presentations/S9444 -std=c++11 -o <...> -c /presentations/S9444/serial.cxx
nvcc -I/presentations/S9444 -std=c++11 -x cu -c /presentations/S9444/parallel.cu
     -o  <...>
<...>
c++ -std=c++11 -o <...> -c /presentations/S9444/main.cxx
c++  main.cxx.o -o gtc -L/usr/local/cuda/lib64/stubs  -L/usr/local/cuda/lib64
     libgtc_lib.a -lcudadevrt -lcudart_static -lrt -lpthread -ldl
```

Kitware

Language Warning Flags

Which one is the better option?

```
set(CMAKE_CXX_FLAGS "-Wall")
set(CMAKE_CUDA_FLAGS "-Xcompiler=-Wall")
```

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(CMAKE_CUDA_FLAGS "${CMAKE_CUDA_FLAGS} -Xcompiler=-Wall")
```

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall" CACHE STRING "" FORCE)
set(CMAKE_CUDA_FLAGS "${CMAKE_CUDA_FLAGS} -Xcompiler=-Wall" CACHE STRING "" FORCE)
```

Kitware

Language Warning Flags

```
set(CMAKE_CXX_FLAGS "-Wall") # Clears any users CXX FLAGS! :(
set(CMAKE_CUDA_FLAGS "-Xcompiler=-Wall") # Clears any users CUDA FLAGS! :(
```

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(CMAKE_CUDA_FLAGS "${CMAKE_CUDA_FLAGS} -Xcompiler=-Wall")
```

```
set(CMAKE_CXX_FLAGS "..." CACHE STRING "" FORCE) # Will keep appending each time
set(CMAKE_CUDA_FLAGS "..." CACHE STRING "" FORCE)# you re-configure the project
```

Kitware

Variables and the Cache

Dereferences look first for a local variable, then in the cache if there is no local definition for a variable

Local variables hide cache variables

Variables and the Cache

```
set(msg "hello" CACHE STRING "docs" FORCE)
message("message value ='${msg}'")
set(msg "world")
message("message value ='${msg}'")
```

```
message value ='hello'
message value ='world'
```

Kitware

Language Warning Flags as Targets

```
set(cxx_flags -Wall)
set(cuda_flags -Xcompiler=-Wall)
add_library(developer_flags INTERFACE)
target_compile_options(developer_flags INTERFACE
  # Flags for CXX builds
  $<$<COMPILE_LANGUAGE:CXX>:${cxx_flags}>
  # Flags for CUDA builds
  $<$<COMPILE_LANGUAGE:CUDA>:${cuda_flags}>)
target_link_libraries(gtc_compiler_flags INTERFACE
  $<BUILD_INTERFACE:developer_flags>)
```

Kitware

Get CUDA Warnings Numbers

```
set(cuda_flags "-Xcudafe=--display_error_number")  # Might be
                                                   # undocumented
```

```
../parallel.cu(9): warning #2905-D: calling a __host__
function("bar") from a __host__ __device__ function("foo") is not
allowed
```

Kitware

Control GPU Architecture

```
set(CMAKE_CUDA_FLAGS "${CMAKE_CUDA_FLAGS} -arch=sm_60")
```

```
set(cuda_flags -arch=sm_60 -Xcompiler=-Wall)
add_library(developer_flags INTERFACE)
target_compile_options(developer_flags INTERFACE
  …
  $<$<COMPILE_LANGUAGE:CUDA>:${cuda_flags}>)
```

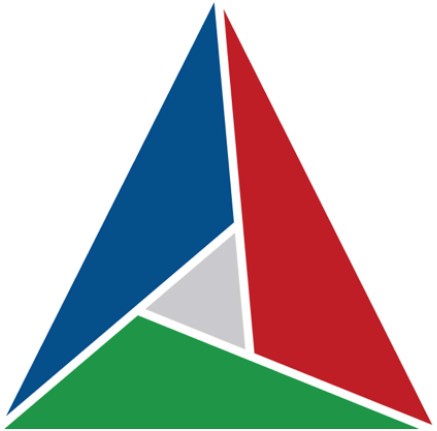
If you want to use separable compilation you will need to use CMAKE_CUDA_FLAGS as target_compile_options aren't propagated when doing device linking.

Kitware

```
cmake_minimum_required(VERSION 3.12...3.14 FATAL_ERROR)
project(GTC)

#options
option(GTC_ENABLE_CUDA "Enable CUDA" OFF)

if(GTC_ENABLE_CUDA)
  enable_language(CUDA)
endif()

#-----------------------------------------------------------
add_library(gtc_compiler_flags INTERFACE)
target_compile_features(gtc_compiler_flags
                        INTERFACE cxx_std_11)
set(CMAKE_CXX_EXTENSIONS Off)

#-----------------------------------------------------------
add_library(developer_flags INTERFACE)
set(cxx_flags -Wall)
set(cuda_flags -arch=sm_60 -Xcompiler=-Wall -Xcudafe=--display_error_number)
target_compile_options(developer_flags INTERFACE
  # Flags for CXX builds
  $<$<COMPILE_LANGUAGE:CXX>:${cxx_flags}>
  # Flags for CUDA builds
  $<$<COMPILE_LANGUAGE:CUDA>:${cuda_flags}>
)
target_link_libraries(gtc_compiler_flags INTERFACE
  $<BUILD_INTERFACE:developer_flags>)
```

```
#-----------------------------------------------------------
add_library(gtc_lib STATIC)
target_sources(gtc_lib PRIVATE serial.cxx)
if(GTC_ENABLE_CUDA)
  target_sources(gtc_lib PRIVATE parallel.cu)
endif()

#-----------------------------------------------------------
target_link_libraries(gtc_lib PUBLIC gtc_compiler_flags)
target_include_directories(gtc_lib
  PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}
  INTERFACE $<INSTALL_INTERFACE:include/gtc>)

#-----------------------------------------------------------
add_executable(gtc)
target_sources(gtc PRIVATE main.cxx)
target_link_libraries(gtc PRIVATE gtc_lib)
```

Kitware



CMake

Find Modules
A
Small Detour

Kitware

Using Find Modules

One of CMake strengths is the find_package infrastructure

CMake provides 150 find modules

- cmake --help-module-list
- https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html

```
find_package(PythonInterp)
find_package(TBB REQUIRED)
```

Kitware

Using Find Modules

CMake supports each project having custom find modules

Find modules have a convention. You should read the https://cmake.org/cmake/help/latest/manual/cmake-developer.7.html#find-modules for best practices

```
set(CMAKE_MODULE_PATH
  ${CMAKE_MODULE_PATH} ${CMAKE_CURRENT_SOURCE_DIR}/CMake)
```

```
-rw-r--r-- 1 robert robert 19434 May 10  2018 FindOpenGL.cmake
-rw-r--r-- 1 robert robert 22463 Jun  1  2018 FindOpenMP.cmake
-rw-r--r-- 1 robert robert  1766 May  1  2018 FindPyexpander.cmake
-rw-r--r-- 1 robert robert 13129 Oct 17 15:43 FindTBB.cmake
```

Using Find Modules

- Modern approach: packages construct import targets which combine necessary information into a target.
- Classic CMake: when a package has been found it will define the following:
  - <NAME>_FOUND
  - <NAME>_INCLUDE_DIRS
  - <NAME>_LIBRARIES

Kitware

Using Find Modules

Our library “trunk” needs PNG

```
find_package(PNG REQUIRED)
add_library(trunk SHARED trunk.cxx)
```

Preferred Modern CMake approach:

```
target_link_libraries(trunk PRIVATE PNG::PNG)
```

Historical (Classic) approach:

```
target_link_libraries(trunk ${PNG_LIBRARIES})
include_directories(trunk ${PNG_INCLUDE_DIRS})
```

Kitware

Using Config Modules

`find_package` also supports config modules

- Config modules are generated by the CMake `export` command
- Will generate import targets with all relevant information, removing the need for consuming projects to write a find module

Understanding Find Modules Searches

CMake's `find_package` uses the following pattern:

- <PackageName>_ROOT from cmake, than env [3.12]
- CMAKE_PREFIX_PATH from cmake
- <PackageName>_DIR from env
- CMAKE_PREFIX_PATH from env
- Any path listed in `find_package(PNG HINTS /opt/png/)`

Understanding Find Modules Searches

- PATH from env
- paths found in the CMake User Package Registry
- System paths as defined in the toolchain/platform
  - CMAKE_SYSTEM_PREFIX_PATH
- Any path listed in `find_package(PNG PATHS /opt/png/)`

Kitware

Find Module Variables

In general all the search steps can be selectively disabled. For example to disable environment paths:

```
find_package(<package> NO_SYSTEM_ENVIRONMENT_PATH)
```

You can disable all search locations except HINTS and PATHS with:

```
find_package(<package> PATHS paths... NO_DEFAULT_PATH)
```

Kitware

Direct Find Modules Searches

CMAKE_FIND_ROOT_PATH

- N directories to "re-root" the entire search under.

```
cmake -DCMAKE_FIND_ROOT_PATH=/home/user/pi .
Checking prefix [/home/user/pi/usr/local/]
Checking prefix [/home/user/pi/usr/]
Checking prefix [/home/user/pi/]
```

Kitware

Direct Find Modules Searches

CMAKE_PREFIX_PATH

- Prefix used by find_package as the second search path

```
<prefix>/                                                    (W)
<prefix>/(cmake|CMake)/                                      (W)
<prefix>/<name>*/                                            (W)
<prefix>/<name>*/(cmake|CMake)/                              (W)
<prefix>/(lib/<arch>|lib|share)/cmake/<name>*/               (U)
<prefix>/(lib/<arch>|lib|share)/<name>*/                     (U)
<prefix>/(lib/<arch>|lib|share)/<name>*/(cmake|CMake)/       (U)
<prefix>/<name>*/(lib/<arch>|lib|share)/cmake/<name>*/       (W/U)
<prefix>/<name>*/(lib/<arch>|lib|share)/<name>*/             (W/U)
<prefix>/<name>*/(lib/<arch>|lib|share)/<name>*/(cmake|CMake)/  (W/U)
```

Kitware

Direct Find Modules Searches

<PackageName>_ROOT

- Prefix used by find_package to start searching for the given package
- The package root variables are maintained as a stack so if called from within a find module, root paths from the parent's find module will also be searched after paths for the current package.

Debugging Find Modules

```
find_package(PNG REQUIRED)
```

```
strace -e trace=access cmake .
…
access("/usr/local/sbin/include/zlib.h", R_OK) = -1
access("/usr/local/sbin/zlib.h", R_OK)  = -1
access("/usr/local/bin/include/zlib.h", R_OK) = -1
access("/usr/local/bin/zlib.h", R_OK)   = -1
```

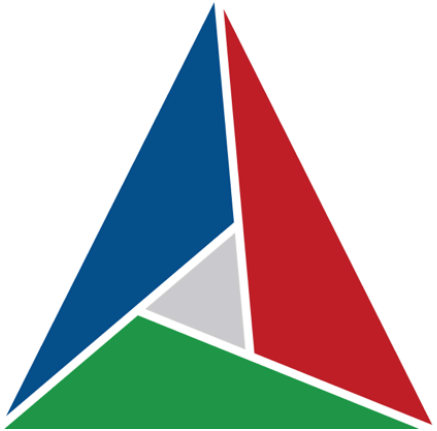
Kitware

Debugging Config Find Modules

```
find_package(PNG CONFIG REQUIRED)
```

```
cmake -DCMAKE_FIND_DEBUG_MODE=ON .
Checking prefix [/usr/local/]
Checking file [/usr/local/PNG.cmake]
Checking file [/usr/local/PNG-config.cmake]
Checking prefix [/usr/]
Checking file [/usr/PNGConfig.cmake]
```

Kitware



CMake

Onto Exporting

Exporting Targets

Install command will generate imported targets

```
install(TARGETS gtc gtc_lib gtc_compiler_flags
        EXPORT gtc-targets) # DESTINATION is automatic in 3.14
install(EXPORT gtc-targets
        NAMESPACE gtc::
        DESTINATION lib/cmake/gtc)
```

```
[0/1] Install the project...
-- Install configuration: "Release"
-- Installing: /home/robert/Work/S9444/bin/gtc
-- Installing: /home/robert/Work/S9444/lib/libgtc_lib.a
-- Installing: /home/robert/Work/S9444/lib/cmake/gtc/gtc-targets.cmake
-- Installing: /home/robert/Work/S9444/lib/cmake/gtc/gtc-targets-release.cmake
```

Now the `.config` to import Targets

We need to make a GTCConfig.cmake that will import the targets we just installed

CMakePackageConfigHelpers can help with the generation of the GTCConfig.cmake file

Exporting of find package calls has to replicated inside GTCConfig.cmake

Kitware

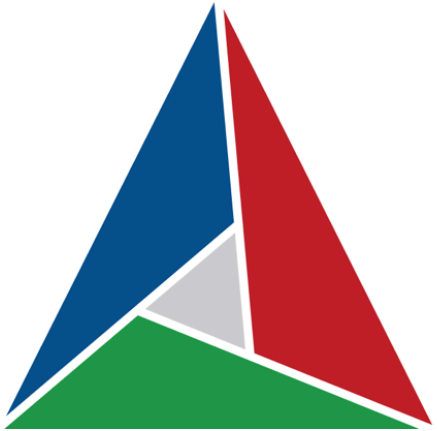
Generating Export Package

```
include(CMakePackageConfigHelpers)
configure_package_config_file(ConfigTemplate.cmake.in
  "${CMAKE_CURRENT_BINARY_DIR}/GTCConfig.cmake"
  INSTALL_DESTINATION "lib/cmake/gtc"
)
```

```
include(CMakeFindDependencyMacro)
find_dependency(PNG REQUIRED)

include("${CMAKE_CURRENT_LIST_DIR}/gtc-targets.cmake")
```

Kitware

```cmake
#-----------------------------------------------------------
add_executable(gtc)
target_sources(gtc PRIVATE main.cxx)
target_link_libraries(gtc PRIVATE gtc_lib)

#-----------------------------------------------------------
install(TARGETS gtc gtc_lib gtc_compiler_flags
        EXPORT gtc-targets) #DESTINATION is automatic in 3.14
install(EXPORT gtc-targets
        NAMESPACE gtc::
        DESTINATION lib/cmake/gtc)

#-----------------------------------------------------------
include(CMakePackageConfigHelpers)
configure_package_config_file(ConfigTemplate.cmake.in
  "${CMAKE_CURRENT_BINARY_DIR}/lib/cmake/gtc/GTCConfig.cmake"
   INSTALL_DESTINATION lib/cmake/gtc)
install(
  FILES
    "${CMAKE_CURRENT_BINARY_DIR}/lib/cmake/gtc/GTCConfig.cmake"
  DESTINATION lib/cmake/gtc)
```



CMake

Separable Compilation

Kitware

Separable Compilation

Separable compilation allows CUDA code to call device functions implemented in other translation units

Separable compilation doesn’t allow for device functions to be called across dynamic library boundaries

Separable Compilation

A device link step must occur which mangles all device symbols

Only other functions that are part of the same device link invocation can call those functions

Separable Compilation

```
set_target_properties(gtc_lib PROPERTIES
  POSITION_INDEPENDENT_CODE ON
  CUDA_SEPARABLE_COMPILATION ON)
```

```
c++  -I/presentations/S9444 -std=c++11 -o <...> -c /presentations/S9444/serial.cxx
nvcc -I/presentations/S9444 -std=c++11 -x cu -dc /presentations/S9444/parallel.cu
     -o  <...>
<...>
c++ -std=c++11 -o <...> -c /presentations/S9444/main.cxx
nvcc -Xcompiler=-fPIC -Wno-deprecated-gpu-targets -shared -dlink
      main.cxx.o -o cmake_device_link.o
     -L/usr/local/cuda/lib64/stubs -L/usr/local/cuda/lib64 libgtc_lib.a -lcudadevrt
     -lcudart_static -lrt -lpthread -ldl
c++  main.cxx.o cmake_device_link.o -o gtc -L/usr/local/cuda/lib64/stubs
     -L/usr/local/cuda/lib64 libgtc_lib.a -lcudadevrt -lcudart_static -lrt -lpthread
     -ldl
```

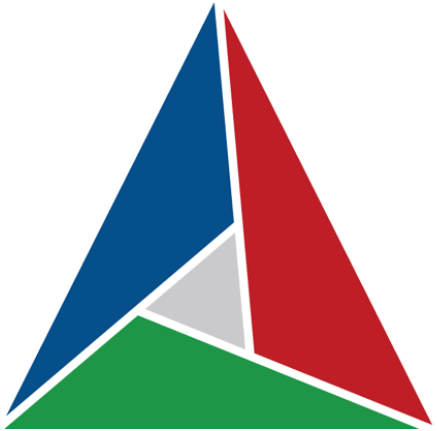
Controlling Device Linking

CMake by default does device linking of executables and dynamic libraries. For static libraries it is delayed for when they are consumed by a executable or dynamic library

CUDA_RESOLVE_DEVICE_SYMBOLS allows for full control over device linking for executables, dynamic, and static libraries

```
set_target_properties(gtc PROPERTIES
  CUDA_RESOLVE_DEVICE_SYMBOLS OFF)
```

Kitware



CMake

PTX

Kitware

Parallel Thread Execution

CMake 3.9 adds support for Parallel Thread Execution (PTX) files in CUDA

- PTX is a pseudo-assembly language for CUDA
- PTX files are Installable, Exportable, Importable, and can be used in Generator Expressions.

PTX files examples

```
add_library(CudaPTXObjects OBJECT
   kernelA.cu kernelB.cu)
set_target_properties(CudaPTXObjects
   PROPERTIES CUDA_PTX_COMPILATION ON)
```

Instead of compiling to host/assembly code you compile to PTX and load at runtime.

Kitware

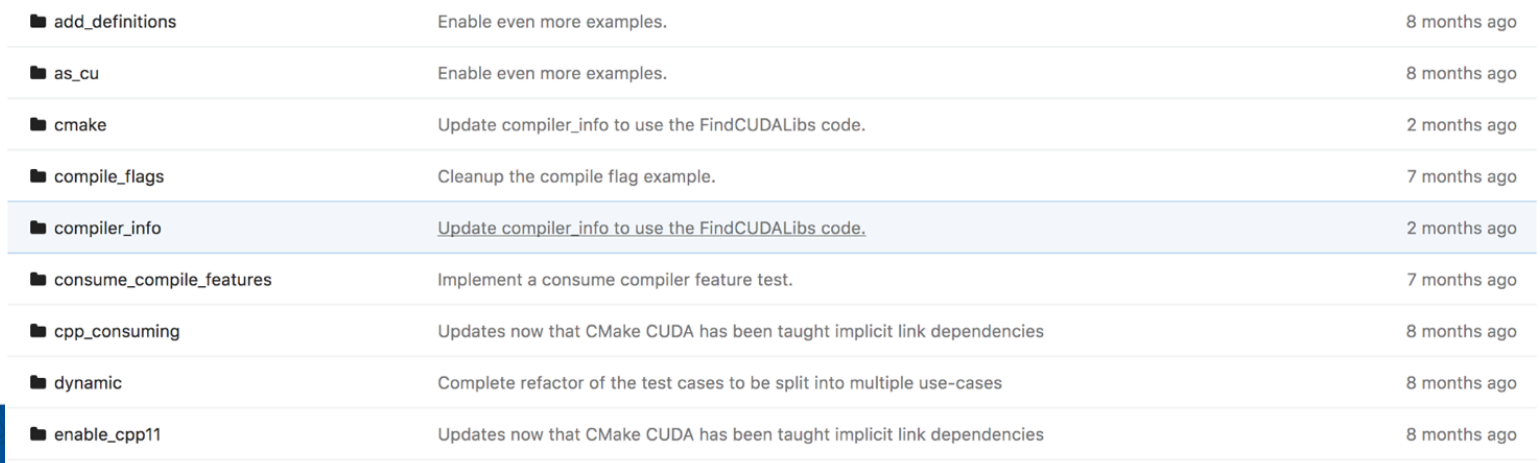
Thank You

Explore VTK-m (my CUDA+CMake project)

- https://gitlab.kitware.com/vtk/vtk-m/

Explore more CUDA+CMake snippets

- https://gitlab.kitware.com/robertmaynard/cmake_cuda_tests



add_definitions	Enable even more examples.	8 months ago
as_cu	Enable even more examples.	8 months ago
cmake	Update compiler_info to use the FindCUDALibs code.	2 months ago
compile_flags	Cleanup the compile flag example.	7 months ago
compiler_info	Update compiler_info to use the FindCUDALibs code.	2 months ago
consume_compile_features	Implement a consume compiler feature test.	7 months ago
cpp_consuming	Updates now that CMake CUDA has been taught implicit link dependencies	8 months ago
dynamic	Complete refactor of the test cases to be split into multiple use-cases	8 months ago
enable_cpp11	Updates now that CMake CUDA has been taught implicit link dependencies	8 months ago

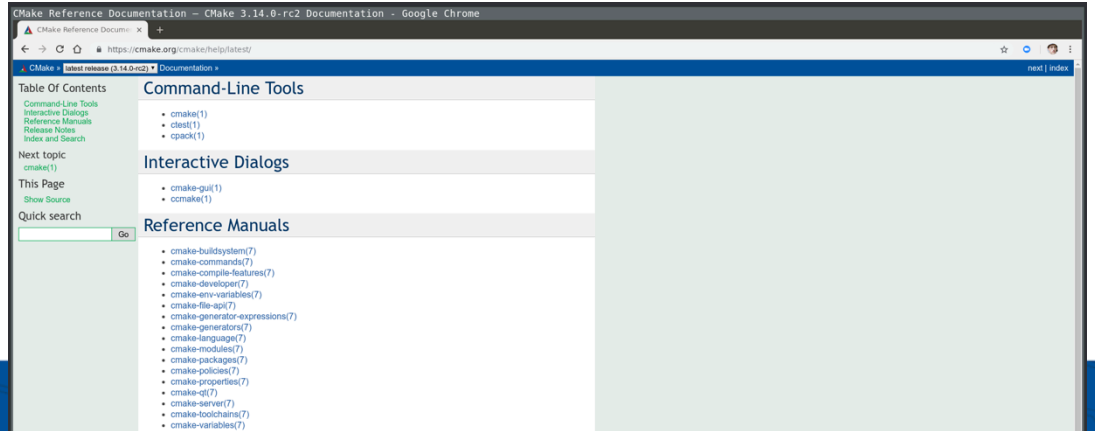
Thank You

Read “how to write a CMake buildsystem”

- https://cmake.org/cmake/help/v3.14/manual/cmake-buildsystem.7.htmlExplore the CMake documentation

Explore the CMake documentation

- https://www.cmake.org/cmake/help/v3.14/



Kitware

Thank You

Robert Maynard

robert.maynard@kitware.com

@robertjmaynard

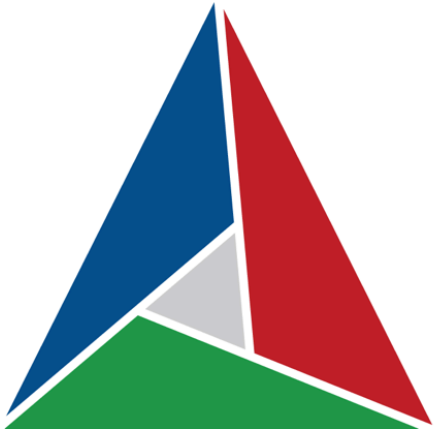
Thanks to NVIDIA for technical support when developing this work

Checkout out:

Kitware @ www.kitware.com

CMake @ www.cmake.org

Please complete the Presenter Evaluation sent to you by email or through the GTC Mobile App. Your feedback is important!



CMake

Recent Releases

Kitware

CMake 3.11 Changes

- add_library and add_executable don't require explicit source files but instead they can be added with target_sources

- Added per source compiler options property
  – COMPILE_OPTIONS

- https://cmake.org/cmake/help/v3.11/release/3.11.html

Kitware

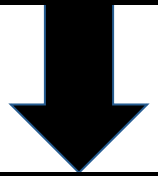
CMake 3.11: Performance

- Improved CMake's runtime performance
  - efficient handling of custom commands
  - efficient source file lookup heuristics
  - efficient import target lookups
- Better CTest parallel job execution overhead

Kitware

CMake 3.11: Import Libraries

```
find_package(TBB REQUIRED)
add_library(vtkm::tbb SHARED IMPORTED GLOBAL)
set_target_properties(vtkm::tbb PROPERTIES
  INTERFACE_INCLUDE_DIRECTORIES "${TBB_INCLUDE_DIRS}"
)
```



```
find_package(TBB REQUIRED)
add_library(vtkm::tbb SHARED IMPORTED GLOBAL)
target_include_directories(vtkm::tbb INTERFACE "${TBB_INCLUDE_DIRS}")
```

Kitware

CMake 3.12 Changes

- cmake --build *build_dir* -j *N*
- Now can request compilation with C++20 (cxx_std_20)
- Visual Studio 2017 generator now supports toolset with a minor version (“version=14.##”)
- find_package now supports <PackageName>_ROOT for all find modules
- Fortran dependency scanning now supports dependencies implied by Fortran Submodules

Kitware

CMake 3.12 Changes

- You can check if a target exists using generator expressions:
  - $<TARGET_EXISTS> and $<TARGET_NAME_IF_EXISTS>
- add_compile_definitions was added and supersedes the previous add_definitions command

- https://cmake.org/cmake/help/v3.12/release/3.12.html

Kitware

CMake 3.12: CONFIGURE_DEPENDS

```
file(GLOB_RECURSE srcs CONFIGURE_DEPENDS
  "${CMAKE_CURRENT_SOURCE_DIR}/src/*.cxx"
  "${CMAKE_CURRENT_SOURCE_DIR}/src/*.cu"
)
add_library(objs OBJECT ${srcs})
```

Kitware

CMake 3.13: Changes

- cmake -S *source_dir* -B *build_dir*
- target_link_libraries can now modify targets outside the current directory
- install(TARGETS) can install targets created in anywhere

Kitware

CMake 3.13: target_sources

```
target_sources(vtkm_cont PRIVATE
  ${CMAKE_CURRENT_SOURCE_DIR}/AlgorithmsOpenMP.cxx
  ${CMAKE_CURRENT_SOURCE_DIR}/ArrayManagerOpenMP.cxx
  ${CMAKE_CURRENT_SOURCE_DIR}/RadixSortOpenMP.cxx
)
```

```
target_sources(vtkm_cont PRIVATE
  AlgorithmsOpenMP.cxx
  ArrayManagerOpenMP.cxx
  RadixSortOpenMP.cxx
)
```

CMake 3.13: target_link_options

```
add_library(objs OBJECT controller.cxx kernels.cu)
target_link_libraries(objs
    PUBLIC compiler_info
    PRIVATE Catch
)
target_link_options(objs PUBLIC -fuse-ld=gold)
```

Kitware

CMake 3.13: target_link_options

- SHELL: Disables CMake logic to de-duplicate strings (-D A -D B stays as is)
- LINKER: Allows for passing flags to the linker tool with having to use -Wl/-Xlinker
- Allows for FindMPI and FindThreads to properly support CUDA

Kitware

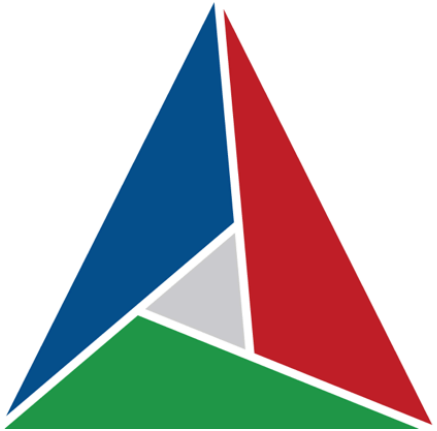
CMake 3.14: Changes

- Supports cross-compilation for iOS, tvOS, or watchOS using simple toolchain files
- CMAKE_BUILD_RPATH_USE_ORIGIN for relocatable and reproducible builds that are invariant of the build directory
- install(TARGETS) can now install to an appropriate default directory for a given target type
- Install(CODE|SCRIPT) now support generator expressions

Kitware

CMake 3.14: Changes

- if(DEFINED CACHE{VAR}) now checks the existence of a cache variable
- cmake --build <build> gained a verbose flag (-v / --verbose)
- A file-based api for clients to get semantic build-system information has been added. This will replace cmake-server



CMake

other bits and pieces

Kitware

GoogleTest integration

```
include(GoogleTest)
add_executable(tests tests.cpp)
target_link_libraries(tests GTest::GTest)
```

- gtest_discover_tests: added in CMake 3.10.
  - CMake asks the test executable to list its tests.
    Finds new tests without rerunning CMake.

```
gtest_discover_tests(tests)
```

Kitware

Build Configurations

- With Makefile generators(Makefile, Ninja):
  - CMAKE_BUILD_TYPE:STRING=Release
  - known values are: Debug, Release, MinSizeRel, RelWithDebInfo

- To build multiple configurations with a Makefile generator, use multiple build trees

Build Configurations

- With multi-config generators (Visual Studio / Xcode):
  - CMAKE_CONFIGURATION_TYPES
    - = list of valid values for config types
  - All binaries go into config subdirectory

```
${CMAKE_CURRENT_BINARY_DIR}/bin/Debug/
${CMAKE_CURRENT_BINARY_DIR}/bin/Release/
```

Kitware

Build Configurations

- To set per configuration information:
  - per target:
    - $<CONFIG>

```
target_compile_definitions(Tutorial PRIVATE
  $<$<CONFIG:DEBUG>:ENABLE_DEBUG_CHECKS>
)
```

  - globally:
    - CMAKE_CXX_FLAGS_<CONFIG>

Kitware

Build Configurations

- To get the current configuration type from multi-conf:
  - Generate Time:
    - $<CONFIG>
  - Build-time (deprecated):
    - ${CMAKE_CFG_INTDIR}
  - In source file
    - CMAKE_INTDIR which is defined automatically

Kitware

OBJECT Libraries

- Generate the object files but does not construct an archive or library
  - Can be installed [3.9]
  - Can be exported/imported [3.9]
  - Can be consumed with target_link_libraries [3.12]
  - Can have transitive information [3.12]

Kitware

OBJECT Libraries

```
add_library(root  OBJECT root.cxx)
add_library(trunk OBJECT trunk.cxx)
add_library(leaf  SHARED leaf.cxx)
target_link_libraries(leaf root trunk)
```

```
[100%] Linking CXX shared library libleaf.so
/usr/bin/c++ -fPIC   -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o root.cxx.o trunk.cxx.o
```

Kitware

OBJECT Libraries

```
add_library(root  OBJECT root.cxx)
add_library(trunk OBJECT trunk.cxx)
add_library(leaf  SHARED
            leaf.cxx
            $<TARGET_OBJECTS:root>
            $<TARGET_OBJECTS:trunk>)
```

```
[100%] Linking CXX shared library libleaf.so
/usr/bin/c++ -fPIC   -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o root.cxx.o trunk.cxx.o
```

Kitware

OBJECT Libraries Caveats

- CMake 3.9 added ability for OBJECT libraries to be:
  - Installed / Exported / Imported
  - $<TARGET_OBJECTS> to be used in more generator expression locations

OBJECT Libraries Caveats

- CMake 3.12 added ability to link to OBJECT libraries:
  - Will behave like any other library for propagation
  - Anything that links to an OBJECT library will have the objects embedded into it.