ATLAS
EXPERIMENT

Building CUDA Code

Attila Krasznahorkay

CUDA Basics

- By convention CUDA code is put into the following types of files
  - `*.cuh`: Header files that require CUDA in “some way”
    - Interpreting keywords like `__host__` and `__device__`;
    - Using some type defined in <cuda.h>;
    - Having inlined calls to “CUDA functions”.
  - `*.cu`: Source files that need to be compiled with nvcc
    - At least when we do a “CUDA build”
- In my test code I had to start separating `.cuh` and `.hxx` headers very carefully
  - When writing “dual use” code, which can be interpreted both by nvcc and a regular C++ compiler, one can expect some very weird build- and runtime issues otherwise
- And then `.cuh` files are treated as “virulent”, if a source or header file needs to include a `.cuh` file, it itself must become a `.cu` or `.cuh` file

nvcc

- The nvcc compiler is **in principle** a fully fledged C++ compiler for the “host code”
  - For simple projects the recommendation from NVidia is that you should just build all your code with their compiler
  - Once they complete the rewrite of nvcc on top of LLVM, this may even become a possibility for larger projects later on…
- But not quite…
  - Right now (with CUDA 10.1) the latest C++ standard supported by nvcc is C++14
    - I had to organise my code quite carefully never to expose `.cu` and `.cuh` files to any Gaudi headers, as those use C++17 very aggressively by now
  - The error / warning messages from nvcc are **much worse** than those from GCC or Clang
    - So you really want to limit how much “complicated” code you expose to it 😕
- It has its own set of command line flags, which overlap with those used by GCC and Clang to some extent, but not perfectly by far
  - So any build system must explicitly know when it’s dealing with this compiler

CMake Basics (1)

- This is mostly for our CMS colleagues…
- ATLAS uses CMake for all of its builds since ~2 years
  - It's a high level language for describing the build of software projects of practically any size
- You point the cmake executable at a directory that has a text file called `CMakeLists.txt`, and that sets up the build of the project using a lower level build system
  - On POSIX platforms this means GNU Make by default, but many other possibilities exist
- These `CMakeLists.txt` files can do a lot of things…
  - Set up the build of libraries and executables, that other parts of the build configuration can just use "by name", and pick up all the right build settings like that;
  - Can look up "externals" by looking for the header files and directories of those during the CMake configuration, setting up the build to use those headers and libraries;
  - Can include other files, allowing us to split the configuration of a large project into many small / manageable files

CMake Basics (2)

- A “simple” C++ CMake project would look something like the following:

```
# Set up the project.
cmake_minimum_required( VERSION 3.6 )
project( SimpleProject VERSION 1.0.0 LANGUAGES CXX )

# Look for Boost.
find_package( Boost COMPONENTS program_options REQUIRED )

# Set up the build of a simple shared library.
add_library( SimpleLib SHARED lib/file1.hxx lib/file2.hxx lib/source.cxx )
set_target_properties( SimpleLib PROPERTIES
   PUBLIC_HEADER “lib/file1.hxx;lib/file2.hxx” )
target_include_directories( SimpleLib PUBLIC
   $<BUILD_INTERFACE:${CMAKE_SOURCE_DIR}/lib>
   $<INSTALL_INTERFACE:include> )

# Set up the build of an executable that uses this shared library, and Boost.
add_executable( SimpleExe app/main.cxx )
target_link_libraries( SimpleExe PRIVATE SimpleLib Boost::program_options )

# Set up the installation of them.
install( TARGETS SimpleLib SimpleExe
   RUNTIME DESTINATION bin
   LIBRARY DESTINATION lib
   PUBLIC_HEADER DESTINATION include )
```

AtlasCMake / AtlasLCG

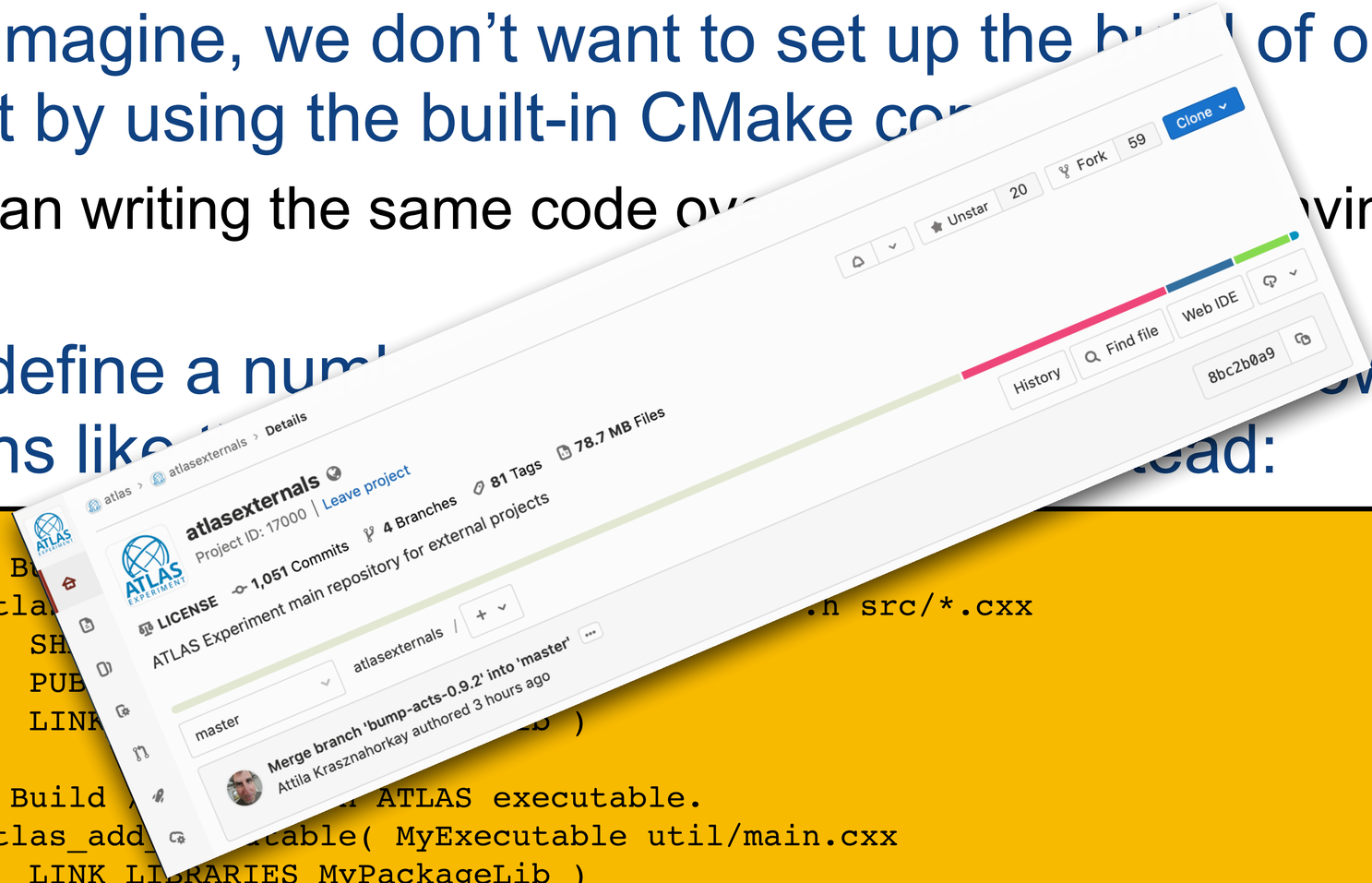
- As you can imagine, we don’t want to set up the build of our offline software just by using the built-in CMake commands
  - It would mean writing the same code over and over again, leaving a lot of room for errors
- Instead we define a number of “helper functions” that allow us to write configurations like the following in our projects instead:

```
# Build / install an ATLAS shared library.
atlas_add_library( MyPackageLib MyPackage/*.h src/*.cxx
   SHARED
   PUBLIC_HEADERS MyPackage
   LINK_LIBRARIES SomeOtherLib )

# Build / install an ATLAS executable.
atlas_add_executable( MyExecutable util/main.cxx
   LINK_LIBRARIES MyPackageLib )
```

AtlasCMake / AtlasLCG

- As you can imagine, we don't want to set up the b… of our offline software just by using the built-in CMake co…
  - It would mean writing the same code o… …ving a lot of room for errors
- Instead we define a num… …w us to write configurations lik… …ead:




```
# Bu
atla                                          .h src/*.cxx
    SH
    PUB
    LINK                                      )

# Build                 ATLAS executable.
atlas_add_              cable( MyExecutable util/main.cxx
   LINK_LIBRARIES MyPackageLib )
```

CMake and CUDA

- Since version 3.8 CUDA is treated as a “first class language” in CMake, making it possible to build “simple” projects like:

```
# Set up the project.
cmake_minimum_required( VERSION 3.8 )
project( SimpleCUDAProject VERSION 1.0.0 LANGUAGES CXX CUDA )

# Look for Boost.
find_package( Boost COMPONENTS program_options REQUIRED )

# Set up the build of a simple shared library.
add_library( SimpleLib SHARED lib/header.hxx lib/source.cxx lib/source.cu )
set_target_properties( SimpleLib PROPERTIES
   PUBLIC_HEADER “lib/header.hxx” )
target_include_directories( SimpleLib PUBLIC
   $<BUILD_INTERFACE:${CMAKE_SOURCE_DIR}/lib>
   $<INSTALL_INTERFACE:include> )

# Set up the build of an executable that uses this shared library, and Boost.
add_executable( SimpleExe app/main.cu )
target_link_libraries( SimpleExe PRIVATE SimpleLib Boost::program_options )

# Set up the installation of them.
install( TARGETS SimpleLib SimpleExe
   RUNTIME DESTINATION bin
   LIBRARY DESTINATION lib
   PUBLIC_HEADER DESTINATION include )
```

What CMake Does…

- The build procedure for a CUDA + C++ library / executable is pretty elaborate with CMake…
  - It builds the C++ source files with the regular C++ compiler, and the CUDA source files with nvcc into object files
  - Once all the CUDA object files are ready, it runs nvcc on these object files to produce a “device object file” that collects all the device code from all of the object files
  - It links the library / executable using the system linker on all of the object files created previously
- I have to admit, I don’t fully understand myself how all of these things come together to be honest… 😟

Handling Device Code

- By default nvcc expects `.cu` files to "see" all the device code that they want to run, all at the same time
  - Which, as long as your GPU calculations are simple, is doable
- But as soon as developing something a bit more complicated, this is not enough
  - For instance you may want to create some helper classes that can do some common operations on a GPU, and pick those up from a single place for the entire project
- CUDA / nvcc supports this through "separable device code compilation"
  - It's not that old of a feature actually…
  - In practice you just give an extra flag (`-dc`) to the nvcc call that creates object files out of the `.cu` files that need this compilation option

Device Code and Libraries

- Device code compiled like that **must not** be put into a shared library
  - CUDA can not look up the device code at runtime. Even in this separable build mode, it must see all the device code that needs to be used in any single GPU task, during the final linking the latest.
- It is hence advertised that one should put common device code into static libraries, that get linked into all clients
  - With AtlasCMake one would do it like:

```
# Build / install an ATLAS shared library.
atlas_add_library( MyCUDALib MyCUDALib/*.h MyCUDALib/*.cuh src/*.cxx src/*.cu
   STATIC
   PUBLIC_HEADERS MyCUDALib
   LINK_LIBRARIES SomeOtherLib )
set_target_properties( MyCUDALib PROPERTIES
   CUDA_SEPARABLE_COMPILATION ON
   POSITION_INDEPENDENT_CODE  ON )
```

But…

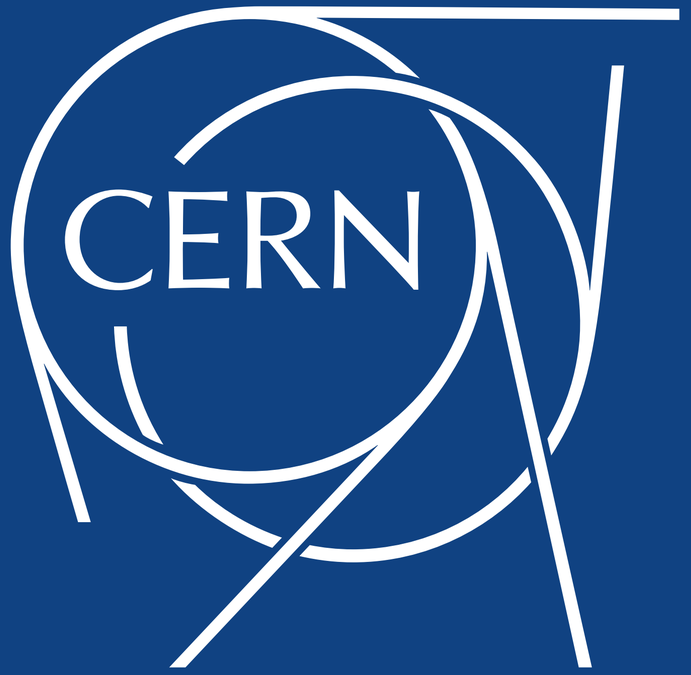
- …it doesn't quite work
- As soon as one tries to put just a little bit more complicated types into a shared library like that, the final binary becomes non-functional
- I have a bug report open with NVidia about this (which is unfortunately not publicly visible), which I demonstrated for them in:
  - https://github.com/krasznaa/CUDALinkTest
- Long story short, I can only put device code into "object libraries", I can't use static ones

And That's Not All…

- If I only had to handle the actual device code using object libraries, that would not be the end of the world
- But it seems that putting certain host-side code into shared libraries also causes problems…
- In my CUDAGaudi example I bumped into this with the AthCUDAInterfaces package/library
  - The interface defines an `AthCUDA::StreamHolder` class that I use to pass around CUDA streams and their ownership between "CUDA aware" components using a pure C++ class
    - The interface holds on to a `void*` pointer, which CUDA capable clients can cast to `cudaStream_t`
  - The package / library doesn't have a single line of CUDA code in it!
- When I compiled that code into a shared library, I started seeing issues with calling CUDA functions in my test job that had nothing to do with streams
  - Once I turned the library into an "object library", the issues disappeared
- -> It seems that CUDA is using some non-trivial memory management for its own objects that **really** doesn't like shared libraries…

Dual-Use Builds

- I also taught our `atlas_add_library(…)`, `atlas_add_executable(…)` and `atlas_add_test(…)` functions what to do with `.cu` files when encountering them in a build that doesn't have CUDA available
  - The build system just attempts to use them as regular C++ files, assuming that the code is set up with correct pre-processor macros to allow this to happen
- That works quite well, that's why I didn't go into details about that… 😛



http://home.cern