GROMACS - Task #1390
manage C+11 support and CUDA better
11/30/2013 08:40 PM - Mark Abraham

Status: Closed
Priority: Normal
Assignee: mdrun
Category: mdrun
Target version: 2016
Difficulty: uncategorized

Description
For some time, master branch has disabled the propagation of CMAKE_HOST_*_FLAGS to the nvcc compiler because it's inconvenient to check for and filter out a possible C++11 flag (because nvcc can't cope with it). It's also inconvenient to just add a C++11 flag to the compiler flags for the non-CUDA C++ source files, because we glob C and C++ files together and combine them all into a single target at one location (chiefly src/gromacs/CMakeLists.txt at the moment). The globbing approach is also quite inconvenient because it is not possible to customize compilation for any file until after it has been added to a target (e.g. the call to add_library(libgromacs...)), by which time we are no longer in the module CMakeLists.txt.

One alternative (uploaded to Gerrit) is to disable C++11 across the whole project when compiling for GPUs, so that (for example) GROMACS CPU acceleration flags are available to the .cu code that is not actually off-loaded. That's probably neither best nor a disaster. Does anyone have an opinion about which of those would be better? Checking performance would be something we should do, unless one solution is clearly inferior in a way that I have not yet thought of!

If/when nvcc starts coping, we can relax the dependent predicate.

Either way, keeping the new CMake option is probably useful if some compiler actually has broken support for our C++11 subset (regex, unique_ptr, move) and someone wants to turn it off.

For proper solutions:

- We could glob the source files separately into C and C++ lists so that after we add_library(libgromacs... (and every other target) we can add the C++11 flag appropriately to the non-CUDA C++ files.
- Using object libraries per module would make solving the whole problem nicer (by adding COMPILE_FLAGS per object library target, which might also help localize additions to -I and -L search paths), but until we require CMake 2.8.9 that is inconvenient because files compiled into the object library target do not pick up necessary flags for (at least) eventually building shared libraries from them. Hacking in -fPIC is not portable, of course. CMake 2.8.9 introduces set(CMAKE_POSITION_INDEPENDENT_CODE) to handle that. Falling back on a static build iff "CMake 2.8.8 and -fPIC does not work" would not be a disaster, though...

Thoughts?

Related issues:
Related to GROMACS - Task #1745: Moving to C++11 after Gromacs-5.1

Associated revisions
Revision 80a6e3b0 - 12/29/2013 12:30 AM - Teemu Murtola
Change use of compiler flags with GPU + C++11

Fix some unused parameter warnings in CUDA code, which were made visible now that the warning flags propagate there.

Disable -Wnon-virtual-dtor, since C++-only flags seem to trip nvcc.
It is likely that this same warning is also given by cppcheck and/or the clang static analyzer, so we should not lose much. We can still enable it again if we implement partial propagation of flags to nvcc.

Refs #1390
Change-Id: I8489af3dcc3139884065abe2e5806d71992abd6c

Revision 20cc7d8e - 09/30/2015 09:44 PM - Roland Schulz
Manually propagate CUDA host compiler flags
Required to remove C++11 flag.

Should be replaced with I1404fa67af when CUDA 6.5 is required.

Fixes #1390
Refs #1833

Change-Id: I857a4c441e0b3c6b72ac776338d48e165830f1a8

History

#1 - 11/30/2013 09:44 PM - Szilárd Páll
Note that I don't expect us supporting only CUDA versions that ship NVCC with C++11 support, so the solution we pick should probably be able to cope with these compilers at least for a year or two.

#2 - 11/30/2013 10:38 PM - Mark Abraham
Sure

#3 - 12/01/2013 08:47 AM - Teemu Murtola
- Status changed from New to In Progress
- Assignee deleted (Berk Hess)

I don't think we can drop support for non-C++11 compilers in any foreseeable future, so Szilard's comment should be automatically taken care of.

I think that we should probably compile all code with the same standard flags, so the approach in Gerrit is probably a reasonable one also for the future. Using different standard flags would create an additional interface within our own code that would require extra care. This may be manageable if the standard libraries are interoperable between the different modes, but that may again be compiler-specific.

I also agree that there should be a mechanism to turn C++11 off, at least as long as it affects installed headers.

#4 - 12/19/2013 09:37 PM - Szilárd Páll
Related to change #2814, where it has come up the issue that nvcc seems to not handle correctly the -Wnon-virtual-dtor flag, we agreed that it may be beneficial to implement partial propagation of C++ flags to nvcc in order to not not constrain ourselves by what nvcc supports.

To achieve this, we would need to take the list of CMAKE_CXX_FLAGS as well as the build type-specific variants, remove those that nvcc does not support/like and pass them to the CUDA_ADD_LIBRARY macro. This macro supports passing compiler options in the following manner:

```
CUDA_ADD_LIBRARY(FOO STATIC foo.cu bar.cu
    OPTIONS -DFLAG=2 "-DFLAG_OTHER=space in flag"
    DEBUG -g
    RELEASE --use_fast_math
    RELWITHDEBINFO --use_fast_math;-g
    MINSIZEREL --use_fast_math)
```

#5 - 02/26/2014 12:13 AM - Roland Schulz
Instead of changing the glob we should remove the glob and list files manual. It is recommended by cmake. And avoids the problem that one has to remember to rerun cmake when files get added/deleted.

#6 - 06/20/2014 10:44 AM - Erik Lindahl
- Target version changed from 5.0 to 5.x

#7 - 05/29/2015 12:37 PM - Teemu Murtola
- Related to Task #1745: Moving to C++11 after Gromacs-5.1 added

#8 - 08/01/2015 11:18 AM - Roland Schulz
What is the advantage of propagating the flags? Why do we want to do it?

#9 - 08/01/2015 03:33 PM - Szilárd Páll
Roland Schulz wrote:

What is the advantage of propagating the flags? Why do we want to do it?

Because we don't want inconsistent compiler behavior across CPU C++ and CUDA host code.
Why? What exactly would be inconsistent and what would be the issue? Why do we not need to propagate them for xlc? Or is that something we should fix?

I would think that the main driver is that users generally expect that if they set some compiler flags through the standard CMake mechanisms, those will be used throughout. And convenience for us that we do not need to have separate sets of flags for nvcc.

Breaking the user expectation can cause, e.g., performance regression if optimization flags are not propagated to nvcc, or really bad behavior if one of the compiler flags affects, e.g., the calling convention used or some C++ ABI details.

And the current solution for xlc is really ugly...

And using a different set of flags can also cause headaches for us, if, e.g., different parts of the code use a different set of warnings (either in the form of bugs missed because warnings were inadvertently disabled, or confusion when the same code produces different results), or if some flags affect the language features recognized.

Teemu's points cover the issues I thought of, so there isn't much I can add :)  

Do we want to require 2.8.9 (or more) for the next version? If we release in a year then 2.8.9 will be 4 years old by that time. 2.8.12 would be almost 3 year old and 3.0 would be over 2 years old.

gmxlib/gpu_utils contains both cu and cpp files. Thus having a object library per module would still require to distinguish between both file types.

A solution is available at http://public.kitware.com/Bug/view.php?id=15240. But we probably don't want to go back to having our own patched FindCUDA do we?

What CUDA versions do we want to support? We could duplicate the section in FindCUDA which does the propagation with the change suggested in 15240. This would be less duplicated code if we don't need support CUDA 4.2 anymore and don't need the _cuda_fix_g3.

Do you prefer a solution similar to the proposed CUDA_NON_PROPAGATED_HOST_FLAGS or a solution to glob the files into 2 or 3 lists (C, C++, and CU) and then set the C++11 flag only for C++?

It isn't recommended to link C++03 and C++11 files together because of ABI compatibility issues. Of course if we want to support CUDA 6.0 for GROMACS 2016 we don't have a choice. But we need to make sure we don't use any of the C++ classes with that issue: https://gcc.gnu.org/wiki/Cxx11AbiCompatibility

Gerrit received a related patchset '1' for Issue #1390.
Uploader: Roland Schulz (roland@rschulz.eu)
Change-Id: I1404fa67af7b9ffbe31dec4922e1eed6a283f3cc
Gerrit URL: https://gerrit.gromacs.org/5075

It's not recommended to link C++03 and C++11 files together because of ABI compatibility issues. Of course if we want to support CUDA 6.0 for GROMACS 2016 we don't have a choice. But we need to make sure we don't use any of the C++ classes with that issue: https://gcc.gnu.org/wiki/Cxx11AbiCompatibility

Gerrit received a related patchset '1' for Issue #1390.
Uploader: Roland Schulz (roland@rschulz.eu)
Change-Id: I857a4c441e0b3c6b72ac776338d48e165830f1a8
Gerrit URL: https://gerrit.gromacs.org/5079

Options
1) use CUDA_PROPAGATE_HOST_FLAGS=off and propagate flags manual (other than c++11) to CUDA_NVCC_FLAGS
I uploaded this as https://gerrit.gromacs.org/#/c/5079/
CUDA_NVCC_FLAGS was already anyhow always overwritten (cannot be changed by user). Thus overwriting also CUDA_NVCC_FLAGS $<config> probably doesn't matter too much. So I think this is probably the best solution.

2) Use COMPILER_OPTIONS property to set C++11 flag. The option doesn't get passed to nvcc. But it requires different lists of source files for C and C++ to not also pass the C++11 flag to C file compilation.

3) Modify FindCUDA. Easiest would be to base it on the solution in 3.3 and just not set --std c++11. But this would prevent the user to use a newer FindCUDA (by using a newer cmake) if there is some other problem.
Roland Schulz wrote:

Options
1) use CUDA_PROPAGATE_HOST_FLAGS=off and propagate flags manual (other than c+11) to CUDA_NVCC_FLAGS
I uploaded this as https://gerrit.gromacs.org/#/c/5079/
CUDA_NVCC_FLAGS was already anyhow always overwritten (cannot be changed by user). Thus overwriting also
CUDA_NVCC_FLAGS_${config} probably doesn't matter too much.

Actually, it does matter. To be a good citizen, I changed CUDA_NVCC_FLAGS to be always (re)generated and not cached. That left
CUDA_NVCC_FLAGS_${config} as the only variable to pass custom flags. Eliminating this possibility would make it quite hard for development.

However, note that AFAIK the CUDA_ADD_* macros do take as explicit argument the flags for various configs. That should also allow explicitly
passing the set of flags, right?

So I think this is probably the best solution.
2) Use COMPIL_OPTIONS property to set C++11 flag. The option doesn't get passed to nvcc. But it requires different lists of source files for C
and C++ to not also pass the C++11 flag to C file compilation.

3) Modify FindCUDA. Easiest would be to base it on the solution in 3.3 and just not set --std c++11. But this would prevent the user to use a
newer FindCUDA (by using a newer cmake) if there is some other problem.

If the above suggestion does not work, I tend to think (though I'm not sure) that a custom FindCUDA may still be best.

#20 - 09/30/2015 09:45 PM - Roland Schulz
- Status changed from In Progress to Resolved
Applied in changeset 20cc7d8e130e6374a4df20bccef94a340bc936b.

#21 - 06/27/2016 08:28 PM - Mark Abraham
- Status changed from Resolved to Closed
- Target version changed from 5.x to 2016