**Gromacs - Bug #1470**

CUDA compiler (gcc-4.6) not compatible with AVX2 flags

03/26/2014 06:07 PM - Magnus Lundborg

<table>
<thead>
<tr>
<th>Status:</th>
<th>Rejected</th>
<th>Priority:</th>
<th>Low</th>
<th>Assignee:</th>
<th>build system</th>
<th>Category:</th>
<th>build system</th>
<th>Target version:</th>
<th>Affected version:</th>
<th>5.0-beta2</th>
</tr>
</thead>
</table>

**Description**

When compiling on an AVX2 architecture CFLAGS_AVX2_FLAG and CXXFLAGS_AVX2_FLAG get set to -march=core-avx2. If compiling also with -DGMX_GPU=ON and gcc-4.6 as CUDA_HOST_COMPILER I get

```
/home/magnus/development/gromacs/src/gromacs/gmxlib/gpu_utils/memtestG80_core.cu:1:0: error: bad value (core-avx2) for -march= switch
```

CMake Error at gpu_utils_generated_memtestG80_core.cu.o.cmake:206 (message):

```
Error generating /home/magnus/install/gromacs/src/gromacs/gmxlib/gpu_utils/CMakeFiles/gpu_utils.dir//./gpu_utils_generated_memtestG80_core.cu.o
```

I guess gcc-4.6 cannot handle -march=core-avx2. So there should probably be some way to degrade to a flag that the compiler can manage (and/or test what AVX flags work on the CUDA_HOST_COMPILER).

**History**

#1 - 03/26/2014 06:59 PM - Szilárd Páll

Thanks for the report. What CUDA version were you using?

I have the feeling that the most robust way to solve this and other issues (e.g. different C++ and nvcc host compiler) is to detect the identity of the CUDA host compiler and separately generate the compiler flags for it in gmxCFlags.cmake. This way we could ensure both that the nvcc host compiler supports the flags as well as that nvcc itself supports it.

Would this be too complicated to do for 5.0?

#2 - 03/26/2014 07:11 PM - Mark Abraham

gcc 4.6 pre-dates avx2, so it can't have been the active compiler when those flags were set (because we check in CMake that they are valid at that time). You seem to be setting CUDA_HOST_COMPILER manually, and I don't think you need to - gcc-4.8 just works for me (and is also CUDA_HOST_COMPILER).

#3 - 03/26/2014 07:40 PM - Szilárd Páll

Mark Abraham wrote:

```
gcc 4.6 pre-dates avx2, so it can't have been the active compiler when those flags were set (because we check in CMake that they are valid at that time). You seem to be setting CUDA_HOST_COMPILER manually, and I don't think you need to - gcc-4.8 just works for me (and is also CUDA_HOST_COMPILER).
```

That could explain, perhaps nvcc knows what the gcc frontend supports and can detect that 4.6 does not support avx2?

Perhaps Magnus is using an older CUDA that does not support gcc 4.8 (or at least it claims so).
I'm using CUDA 5.0 and it seems like that's the problem. It only support gcc versions <=4.6(,x?). My "main" compiler is gcc-4.8, which detects the AVX2 capabilities and the same flags are used by nvcc. I do set the CUDA_HOST_COMPILER manually since otherwise it fails.

So, in my case it seems like updating CUDA is the solution, but I guess there should at least be some elegant way of handling it. One way would be to check what flags actually work with the CUDA_HOST_COMPILER (if it is different from the "main" compiler) or just require a CUDA version that supports gcc-4.7 and above.

Mark Abraham wrote:

Thanks for the report. What CUDA version were you using?

I have the feeling that the most robust way to solve this and other issues (e.g. different C++ and nvcc host compiler) is to detect the identity of the CUDA host compiler and separately generate the compiler flags for it in gmxCFlags.cmake. This way we could ensure both that the nvcc host compiler supports the flags as well as that nvcc itself supports it.

In particular, the SIMD flags are likely irrelevant for CUDA, so we could consider some basic modularity there. Without looking into the details right now, the value of CMAKE_CXX_FLAGS at the time a normal source file is given to add_library is probably how things get set. For CUDA, that's probably the time of some equivalent call. There's no strong reason to have the @set(CMAKE_CXX_FLAGS) logic exactly where it is, and it makes some sense to manage it separately for some modules. Not wanting to propagate SIMD_CXX_FLAGS to CUDA is just the first problem that we could solve.

Would this be too complicated to do for 5.0?

If I'm right, then we don't need to get involved with detecting versions or making work-arounds. We can just issue a warning that the C++ compiler versions aren't matching (because this could lead to problems anyway) and use the SIMD-free CXX flags. A test that CUDA compilation and linking works would also be nicer than some mysterious build-time failure. I can look at the the cxxflags management - can you construct such a test, Szilard? Examples for linking tests can be found in recent commits for libxml2 or zlib.

Szilard Páll wrote:

I'm using CUDA 5.0 and it seems like that's the problem. It only support gcc versions <=4.6(,x?). My "main" compiler is gcc-4.8, which detects the AVX2 capabilities and the same flags are used by nvcc. I do set the CUDA_HOST_COMPILER manually since otherwise it fails.
Quick & dirty solution: edit host_config.h and bump the max CUDA version. In the majority of the cases newer gcc-s just work with CUDA versions that officially don't support it. I myself do that and gcc 4.8 has worked AFAIR with all CUDA versions since from and including 4.1.

So, in my case it seems like updating CUDA is the solution,

Updating CUDA is a solution which will also improve performance! :)

but I guess there should at least be some elegant way of handling it.

There should be and the only robust way I see is constructing a separate set of flags for NVCC_HOST_COMPILER.

---

#7 - 03/26/2014 11:00 PM - Szilárd Páll

@Mark: In general, modularity the way you propose it will cover only the cases where two versions of the same compiler (or compilers with compatible frontends) are used. However, if the main C++ compiler is totally different from the CUDA host compiler, e.g. gcc & clang like here, there can be more than just SIMD optimization flags causing problems.

#8 - 03/27/2014 11:31 AM - Mark Abraham

Szilárd Páll wrote:

@Mark: In general, modularity the way you propose it will cover only the cases where two versions of the same compiler (or compilers with compatible frontends) are used. However, if the main C++ compiler is totally different from the CUDA host compiler, e.g. gcc & clang like here, there can be more than just SIMD optimization flags causing problems.

Unless there's a real need for it, I'd be all for declaring support only for an identical compiler for normal and CUDA code. We have a hard enough time managing one set of flags. Life being inconvenient for CUDA on Mac is not a problem we have to solve.

#9 - 03/27/2014 08:35 PM - Szilárd Páll

Mark Abraham wrote:
Szilárd Páll wrote:

Unless there's a real need for it, I'd be all for declaring support only for an identical compiler for normal and CUDA code. We have a hard enough time managing one set of flags. Life being inconvenient for CUDA on Mac is not a problem we have to solve.

Identical kind or identical kind & version (=identical binary)? The latter will be a rather hard requirement which will result in unnecessary restriction on which compiler can one use for the CPU code. For instance, it can easily happen that an nvcc only supports gcc versions that don't support the SIMD capability of the CPU, e.g. nvcc 5.0 + gcc 4.6 on AVX2 CPU. In these cases, the user would have to either opt for not using the GPU or not using AVX2 which are both quite inconvenient options.

Identical kind of compiler will probably screw only OS X users, at least until clang finally grows up and gains OpenMP support - although apparently recent nvcc seem to (unofficially) work with gcc too.

Additionally, while on the short run creating a list of exceptions/special cases for the CUDA host compiler (omit C++11 flag, omit SIMD flag, etc.) may be the least effort strategy, on the long run this is perhaps no the most robust approach.

Of course, on the opposite end of the spectrum, the most simple approach would be to simply regexp-remove from a string of C++ flags the ones that are known to break certain combinations of compilers.

#10 - 03/27/2014 08:44 PM - Erik Lindahl

Remember that we are testing compilers for a bunch of more or less obscure bugs too. To fully support arbitrary combinations of compilers we might need to duplicate all those tests to run them separately for the cuda compiler too? Even if we're not that extreme.

When it comes to enabling specific SIMD flags there might be a point in only doing that for files that use it, but in general I think we also need to start considering the maintenance efforts we are introducing with ever-increasing complexity of our CMake configuration. It cannot be Gromacs’ responsibility to work around every single issue there might be with installations.

In this particular case it might be a better solution to e.g. require Cuda-5.5 if the user wants to use AVX2, unless that is slower?

#11 - 03/27/2014 10:25 PM - Szilárd Páll

Erik Lindahl wrote:

Remember that we are testing compilers for a bunch of more or less obscure bugs too. To fully support arbitrary combinations of compilers we might need to duplicate all those tests to run them separately for the cuda compiler too? Even if we're not that extreme.

Some we may have to, most checks and workarounds will not really be necessary, I think.

When it comes to enabling specific SIMD flags there might be a point in only doing that for files that use it, but in general I think we also need to start considering the maintenance efforts we are introducing with ever-increasing complexity of our CMake configuration. It cannot be Gromacs’ responsibility to work around every single issue there might be with installations.
Indeed, it can't. However, even though at the moment we don't have any SIMD or OpenMP/multi-threading in .cu files, as we are starting to port more things and move some of the features closer to the GPU, we can quickly end up in the situation where e.g. -mavx may make quite a bit of difference even in the .cu host-side code.

In this particular case it might be a better solution to e.g. require Cuda-5.5 if the user wants to use AVX2, unless that is slower?

In this case of course such a requirement is reasonable. However, this is still a specific case that we are introducing an exception for, hence it requires new code and developer effort to figure out and maintain.

Don't get me wrong, the simpler the solution the better, I' just have the feeling that, if not overly hard, it may be better to invest some effort in a future-proof solution that does not rely on continuously adding exceptions. Unfortunately, testing the flags against the nvcc compiler in use is the only robust solution I can think of.

#12 - 03/27/2014 10:44 PM - Mark Abraham

If using CUDA 5.5 permits the same (e.g. gcc-4.8) compiler to be used for both parts, then I do not see a serious problem with "for simplicity, GROMACS requires the same compiler binary for host and CUDA code." It seems likely to be what most people will do, people are going to want to use the latest CUDA anyway, and Mac users are partly screwed no matter how hard we are prepared to try.

#13 - 03/28/2014 05:00 AM - Teemu Murtola

One point to consider here is that CUDA wants a C++ compiler, as I've said over and over in these different issues. And it is an unfortunate fact of life that C++ code is quite often not ABI compatible even across different versions of the same compiler. Each compiler version has its own, different copy of the C++ headers, as well as the standard library. In a mixed compilation case, which library should then be linked to the resulting binary? After all, we are linking all the code into a single library (or with static linking, into a single executable).

So if we want to play safe, we need to either say that

- CUDA code is not allowed to use any C++ constructs, not internally, and not from any header that is possibly included by CUDA. We may be able to relax this a bit to allow C++ classes that are used by CUDA code internally, but don't depend on any C++ standard library functions, but that doesn't change the big picture.
- We only support sufficiently similar compilers for CUDA and C++. And since "sufficiently similar" is a difficult concept, the only really safe bet is to say the compilers must be the same.

If we pick the first alternative, how are we going to enforce that? The compiler doesn't. I don't think anyone wants to start diagnosing issues that are caused by violating the C++ one definition rule, which very easily results from not doing the above. Probably the linker just randomly discards one of the definitions, and it can lead to obscure run-time failures.
Teemu's argument seems compelling to me!

Teemu, thanks for the comments.

As compelling as the arguments for a hard requirement of CMAKE_CXX_COMPILER == CUDA_HOST_COMPILER are, I've been following the evolution of CUDA and I know that most if not all CUDA releases have been lagging behind with gcc support - let alone other compilers. Hence, we can safely state that there is a clear danger of forcing users to choose between proper CPU support or GPU acceleration.

While it would be convenient to impose such a requirement, I don't think we want to design around this assumption as this would impose rather strong limits for users e.g. only Intel Compiler v12.1 with the tag "20110811" on both CPU and GPU with any current CUDA version.

One option could be to introduce a restriction by default, but allow relaxing it with an advanced variable.

Szilárd Páll wrote:

Teemu, thanks for the comments.

As compelling as the arguments for a hard requirement of CMAKE_CXX_COMPILER == CUDA_HOST_COMPILER are, I've been following the evolution of CUDA and I know that most if not all CUDA releases have been lagging behind with gcc support - let alone other compilers. Hence, we can safely state that there is a clear danger of forcing users to choose between proper CPU support or GPU acceleration.

Only if proper CPU support ever requires a compiler version that did not work with latest CUDA. I do not think this is true now. If so, I do not think this hypothetical more compelling than the gains from simplicity and known C++ ABI compatibility.

While it would be convenient to impose such a requirement, I don't think we want to design around this assumption as this would impose rather strong limits for users e.g. only Intel Compiler v12.1 with the tag "20110811" on both CPU and GPU with any current CUDA version.

Nvidia web docs don't even claim icc is supported, so that seems like a straw man. Nobody has suggested we restrict to officially supported compilers. What objection is there to requiring the same compiler for both kinds of compilation?

One option could be to introduce a restriction by default, but allow relaxing it with an advanced variable.

Maybe, but the problem of flag propagation remains, and I think that solution would then be up to the person who wanted that advanced variable.
Mark Abraham wrote:

Szilárd Páll wrote:

Teemu, thanks for the comments.

As compelling as the arguments for a hard requirement of CMAKE_CXX_COMPILER == CUDA_HOST_COMPILER are, I've been following the evolution of CUDA and I know that most if not all CUDA releases have been lagging behind with gcc support - let alone other compilers. Hence, we can safely state that there is a clear danger of forcing users to choose between proper CPU support or GPU acceleration.

Only if proper CPU support ever requires a compiler version that did not work with latest CUDA. I do not think this is true now. If so, I do not think this hypothetical more compelling than the gains from simplicity and known C++ ABI compatibility.

Honestly, the C++ ABI incompatibility issue being is as hypothetical as the above one. I don't know of any plans to use more extensive C++ in .cu files and, while we need to watch out, ATM I see nothing to fix for now.

I think issuing a warning (when the compilers are not identical) which also states that flag propagation should be simply turned off in case if CUDA compilation errors occur is, in my opinion, a less drastic measure.

While it would be convenient to impose such a requirement, I don't think we want to design around this assumption as this would impose rather strong limits for users e.g. only Intel Compiler v12.1 with the tag "20110811" on both CPU and GPU with any current CUDA version.

Nvidia web docs don't even claim icc is supported, so that seems like a straw man. Nobody has suggested we restrict to officially supported compilers. What objection is there to requiring the same compiler for both kinds of compilation?

This is the relevant page in the NVIDIA docs you may want to have a look at CUDA_HOME_DIR/include/host_config.h too, that's where the compiler versions get enforced.

The objection is that:

- We would greatly limit users in using whatever compiler they want/is available on their system.
- It is an unnecessary restriction to deal with a hypothetical problem. I'm not saying the C++ ABI incompatibility is an unrealistic problem one, but it does not seem to be relevant for 5.0 and probably not for 5.1 either.

One option could be to introduce a restriction by default, but allow relaxing it with an advanced variable.

Maybe, but the problem of flag propagation remains, and I think that solution would then be up to the person who wanted that advanced variable.

The flag propagation is a rare enough issue (clang on Linux or outdated gcc wrt CUDA) that IMHO a configure-time warning is an acceptable (simple) middle-ground.
Szilárd Páll wrote:

Mark Abraham wrote:

Szilárd Páll wrote:

Teemu, thanks for the comments.

As compelling as the arguments for a hard requirement of CMAKE_CXX_COMPILER == CUDA_HOST_COMPILER are, I've been following the evolution of CUDA and I know that most if not all CUDA releases have been lagging behind with gcc support - let alone other compilers. Hence, we can safely state that there is a clear danger of forcing users to choose between proper CPU support or GPU acceleration.

Only if proper CPU support ever requires a compiler version that did not work with latest CUDA. I do not think this is true now. If so, I do not think this hypothetical more compelling than the gains from simplicity and known C++ ABI compatibility.

Honestly, the C++ ABI incompatibility issue being is as hypothetical as the above one. I don't know of any plans to use more extensive C++ in .cu files and, while we need to watch out, ATM I see nothing to fix for now.

The .cu code is not an island. It uses snew implemented in C++ (but probably fine), gmx_fatal and friends (which I will definitely be reworking to support better testing, and I'm not going to abstain deliberately from using C++), and data structures from the nxn kernel headers (which are not immune either). It is exactly this kind of unintended/unknown dependency existence/creation that we should be avoiding. The risk of a reputation of "GROMACS crashes mysteriously with GPUs" is not worth a few percent of CPU performance while support here or there catches up. Nobody wants to spend time "watching out" for things that may seem unrelated to the change at hand.

I think issuing a warning (when the compilers are not identical) which also states that flag propagation should be simply turned off in case if CUDA compilation errors occur is, in my opinion, a less drastic measure.

Compiling with CUDA is already hard enough for a typical user. Why do we want to make more options for more barriers and confused users on mailing lists? I would like to make it harder to break a GROMACS build, not easy to set up a brittle build/binary.

While it would be convenient to impose such a requirement, I don't think we want to design around this assumption as this would impose rather strong limits for users e.g. only Intel Compiler v12.1 with the tag "20110811" on both CPU and GPU with any current CUDA version.

Nvidia web docs don't even claim icc is supported, so that seems like a straw man. Nobody has suggested we restrict to officially supported compilers. What objection is there to requiring the same compiler for both kinds of compilation?

This is the relevant page in the NVIDIA docs you may want to have a look at CUDA_HOME_DIR/include/host_config.h too, that's where the compiler versions get enforced.

OK, but why is that documentation of sample code more relevant than http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#supported-host-compilers?

The objection is that:

- We would greatly limit users in using whatever compiler they want/is available on their system.

Greatly? Yes, that's the idea. But when is that limit a problem? Please name one known use case for two compilers, given latest CUDA is used. How
important is that case?

- It is an unnecessary restriction to deal with a hypothetical problem. I'm not saying the C++ ABI incompatibility is an unrealistic problem one, but it does not seem to be relevant for 5.0 and probably not for 5.1 either.

This is actually the point for not supporting two C++ compilers. There is no way we can or will be sufficiently vigilant to ensure such statements remain true. We're using C++, not avoiding it.

One option could be to introduce a restriction by default, but allow relaxing it with an advanced variable.

Maybe, but the problem of flag propagation remains, and I think that solution would then be up to the person who wanted that advanced variable.

The flag propagation is a rare enough issue (clang on Linux or outdated gcc wrt CUDA) that IMHO a configure-time warning is an acceptable (simple) middle-ground.

Neither of those uses is relevant for your concern that CPU might be wasted if nvcc support lags.

---

#19 - 04/03/2014 03:47 PM - Szilárd Páll

Mark Abraham wrote:

Szilárd Páll wrote:

Mark Abraham wrote:

Szilárd Páll wrote:

Teemu, thanks for the comments.

As compelling as the arguments for a hard requirement of CMAKE_CXX_COMPILER == CUDA_HOST_COMPILER are, I've been following the evolution of CUDA and I know that most if not all CUDA releases have been lagging behind with gcc support - let alone other compilers. Hence, we can safely state that there is a clear danger of forcing users to choose between proper CPU support or GPU acceleration.
Only if proper CPU support ever requires a compiler version that did not work with latest CUDA. I do not think this is true now. If so, I do not think this hypothetical more compelling than the gains from simplicity and known C++ ABI compatibility.

Honestly, the C++ ABI incompatibility issue being is as hypothetical as the above one. I don't know of any plans to use more extensive C++ in .cu files and, while we need to watch out, ATM I see nothing to fix for now.

The .cu code is not an island. It uses snew implemented in C++ (but probably fine), gmx_fatal and friends (which I will definitely be reworking to support better testing, and I'm not going to abstain deliberately from using C++), and data structures from the nxn kernel headers (which are not immune either). It is exactly this kind of unintended/unknown dependency existence/creation that we should be avoiding. The risk of a reputation of "GROMACS crashes mysteriously with GPUs" is not worth a few percent of CPU performance while support here or there catches up. Nobody wants to spend time "watching out" for things that may seem unrelated to the change at hand.

Admittedly, I'm not a C++ expert, but I still have the strong impression the reason for the restriction you're proposing are not timely and sound like a solution for a problem that has yet to arise. If you know what can cause C++ ABI breakage-prone code it should really not be that hard to put a simple compile-time check in the code and add a config to jenkins that uses this:

```c
#if defined CUDA_VERSION && ! defined CXX_AND_CUDA_HOST_COMPILER_IDENTIAL
#error this should not be included in .cu
#endif
```

Additionally, I appreciate that you are trying to root out all possible source of future issues, but I still don't see how is this relevant for 5.0 - which is supposedly imminent.

I think issuing a warning (when the compilers are not identical) which also states that flag propagation should be simply turned off in case if CUDA compilation errors occur is, in my opinion, a less drastic measure.

Compiling with CUDA is already hard enough for a typical user. Why do we want to make more options for more barriers and confused users on mailing lists? I would like to make it harder to break a GROMACS build, not easy to set up a brittle build/binary.

While it would be convenient to impose such a requirement, I don't think we want to design around this assumption as this would impose rather strong limits for users e.g. only Intel Compiler v12.1 with the tag "20110811" on both CPU and GPU with any current CUDA version.

Nvidia web docs don't even claim icc is supported, so that seems like a straw man. Nobody has suggested we restrict to officially supported compilers. What objection is there to requiring the same compiler for both kinds of compilation?

This is the relevant page in the NVIDIA docs you may want to have a look at CUDA_HOME_DIR/include/host_config.h too, that's where the compiler versions get enforced.

OK, but why is that documentation of sample code more relevant than http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#supported-host-compilers?

Because it contains the correct information. Intel Compiler is supported. Docs are sometimes buggy.
The objection is that:

- We would greatly limit users in using whatever compiler they want/is available on their system.

Greatly? Yes, that's the idea. But when is that limit a problem? Please name one known use case for two compilers, given latest CUDA is used. How important is that case?

- CUDA 4.2-5.0 and the default compilers in the Ubuntu and Fedora version of the time (all Ubuntu and friends 12.04-13.04, Fedora 19);
- CUDA 4.1-5.5 and Intel Compilers (>12.1 from 20110811).
- It's common that RHEL/CentOS installations have gcc 4.4 as default and the latest Intel Compilers or gcc.

The latter two cases have in several occasions resulted in GROMACS benchmarked using gcc 4.4 just because the Intel Compiler was too new and unsupported by CUDA.

I'll let you judge how important these cases are - especially as my experience doesn't seem to be relevant enough. I would prefer to not have to do further research and documenting on compiler history, though. ;)

Neither of those uses is relevant for your concern that CPU might be wasted if nvcc support lags.

Of course it is, see the above examples. Concrete use-case: CentOS 6.4, gcc 4.4, Intel Compilers >12.1; with the proposed restrictions the options would be to either use gcc 4.4 for the entire code or not use GPUs.
Only if proper CPU support ever requires a compiler version that did not work with latest CUDA. I do not think this is true now. If so, I do not think this hypothetical more compelling than the gains from simplicity and known C++ ABI compatibility.

Honestly, the C++ ABI incompatibility issue being is as hypothetical as the above one. I don't know of any plans to use more extensive C++ in .cu files and, while we need to watch out, ATM I see nothing to fix for now.

The .cu code is not an island. It uses snew implemented in C++ (but probably fine), gmx_fatal and friends (which I will definitely be reworking to support better testing, and I'm not going to abstain deliberately from using C++), and data structures from the nxn kernel headers (which are not immune either). It is exactly this kind of unintended/unknown dependency existence/creation that we should be avoiding. The risk of a reputation of "GROMACS crashes mysteriously with GPUs" is not worth a few percent of CPU performance while support here or there catches up. Nobody wants to spend time "watching out" for things that may seem unrelated to the change at hand.

Admittedly, I'm not a C++ expert, but I still have the strong impression the reason for the restriction you're proposing are not timely and sound like a solution for a problem that has yet to arise. If you know what can cause C++ ABI breakage-prone code it should really not be that hard to put a simple compile-time check in the code and add a config to jenkins that uses this:

```
[...]
```

But the scope of future ABI incompatibility is at least the whole of libstdc++! As we know from our experience as developers, they can and will change the ABI if it suits them, and people depending on it not changing are already in their own grave.

Additionally, I appreciate that you are trying to root out all possible source of future issues, but I still don't see how is this relevant for 5.0 - which is supposedly imminent.

You're the only person who's mentioned 5.0 ;-) You're right that I want to avoid future mysterious untraceable user-space problems, and also right that it's probably not of great concern for 5.0.

I think issuing a warning (when the compilers are not identical) which also states that flag propagation should be simply turned off in case if CUDA compilation errors occur is, in my opinion, a less drastic measure.

Compiling with CUDA is already hard enough for a typical user. Why do we want to make more options for more barriers and confused users on mailing lists? I would like to make it harder to break a GROMACS build, not easy to set up a brittle build/binary.

While it would be convenient to impose such a requirement, I don't think we want to design around this assumption as this would impose rather strong limits for users e.g. only Intel Compiler v12.1 with the tag "20110811" on both CPU and GPU with any current CUDA version.

Nvidia web docs don't even claim icc is supported, so that seems like a straw man. Nobody has suggested we restrict to officially supported compilers. What objection is there to requiring the same compiler for both kinds of compilation?

This is the relevant page in the NVIDIA docs you may want to have a look at CUDA_HOME_DIR/include/host_config.h too, that's where the compiler versions get enforced.

OK, but why is that documentation of sample code more relevant than http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#supported-host-compilers?
Because it contains the correct information. Intel Compiler is supported. Docs are sometimes buggy.

The objection is that:

● We would greatly limit users in using whatever compiler they want/is available on their system.

Greatly? Yes, that's the idea. But when is that limit a problem? Please name one known use case for two compilers, given latest CUDA is used. How important is that case?

● CUDA 4.2-5.0 and the default compilers in the Ubuntu and Fedora version of the time (all Ubuntu and friends 12.04-13.04, Fedora 19);
● CUDA 4.1-5.5 and Intel Compilers (>12.1 from 20110811).
● It's common that RHEL/CentOS installations have gcc 4.4 as default and the latest Intel Compilers or gcc.

The latter two cases have in several occasions resulted in GROMACS benchmarked using gcc 4.4 just because the Intel Compiler was too new and unsupported by CUDA.

I'll let you judge how important these cases are - especially as my experience doesn't seem to be relevant enough. I would prefer to not have to do further research and documenting on compiler history, though. :)

These seem like misuse cases. We already recommend people use the latest compiler they possibly can, and use the latest CUDA they possibly can. If requiring both compilers to be the same has the side effect of people upgrading an ancient default compiler so that they can use CUDA at all, then that side effect is a win for the requirement, rather than being a draconian restriction!

Neither of those uses is relevant for your concern that CPU might be wasted if nvcc support lags.

Of course it is, see the above examples. Concrete use-case: CentOS 6.4, gcc 4.4, Intel Compilers >12.1; with the proposed restrictions the options would be to either use gcc 4.4 for the entire code or not use GPUs.

The proposed change is to require the same compiler for the whole code. Using gcc-4.4 already disregards the advice of our install guide. Frankly, issuing general warnings that "gcc below 4.7 (or something) is a poor idea" seems reasonable to me. We know people don't read install guides. The case of people using CentOS, or Ubuntu 12.04, etc. for its presumed reliability, using a recent GPU+CUDA, and refusing to install an up-to-date gcc, is not something for which we should be designing, or even discussing.
It seems to me that the main problem should really be solved by Nvidia by supporting the most recent version of gcc/icc when a version of CUDA is released. We should bring it to their attention that this is an important point for us. It might be possible to have ABI compatibility between different version of GCC and between GCC and ICC by correctly setting -fabi-version. Notice that the meaning of -fabi-version doesn't agree between ICC and GCC and that other options can implicitly change -fabi-version (both important for propagating flags). Also it is possible to add -Wabi to get notified of (some) ABI issues (I just tried it and it doesn't seem to show any warnings for gcc 4.7).

ABI compatibility seems to be focused on compatibility between an app compiled with one compiler and a shared library with another (makes sense given that this is the obvious use case). I don't know whether static linking is done the same way and ABI compatibility issues are tested as much for it. If not it might help to make all CUDA code a shared library separate from libgromacs.

If we don't want to make users install ICC 12.1 or GCC 4.7 we could also solve the problem by providing binaries of Gromacs for download.

---

This is somewhat related to the issues where we would like a simple bootstrap routine to be compiled entirely without optimization. The easiest solution might simply be to see if we can move CMake to a few different classes of files with different flags, for instance "CPU/SIMD-specific optimized", "optimized", and "unoptimized". However, presently this would require us to track six different settings due to our dual C/C++ usage, so if possible I would like to hold off and see if we can move entirely to C++ (even if some files, like kernels, mostly would use C syntax) in the master branch first.

For what it simplifies, I don't think there will be any substantial need to have unoptimized code. It's true that some parts of the code will not benefit from aggressive optimization, but there should not be any great harm in applying the same optimization everywhere. Being able to ensure that startup code (and maybe some others) would not use SIMD would allow some nice things.

You might be right that we can get by with two options, but then I think we should use fully-optimized and unoptimized. The problem I've seen is that e.g. AMD CPUs require us to enable FMA4 to get good performance (it makes almost 20% difference) for general code, and those binaries won't run on Intel at all.

I still don't understand your argument. That does not change the fact that the main boundary we would like to have is that some files are compiled with architecture-specific options (that may make it crash on other architectures), and some with general-purpose options. Nothing in your argument supports not using, e.g., -O2 (or even -O3) for the latter set.

But are we sure that -O3 will never result in non-portable code? gcc might not generate -FMA4 with it for now, but there are a whole bunch of other compilers not to mention other platforms than x86.

Now, obviously, I'm sure we can find a set of flags that are conservative enough that no architecture-specific code is added, but what I was trying to say is that we likely cannot use that for parts of the code where we care about performance, and possibly not even the CUDA backend.

Naming of flags aside, then I would argue that we in effect have chosen the solution to have one set that is fully optimized, and another for files where we don't care about performance, a.k.a. "unoptimized" :-

I don't think gcc is ever going to change "-O3" to include "-mtune=native". This would break the build descriptions for many packages of most distributions. But if we want to extra sure we could add "-mtune=generic".

But I don't think the CUDA flags have anything in common with the architecture independent compiler flags we want for the startup code. For the CUDA issue it just happened to be the case that the problematic flag is the architecture dependent flag (avx2). It could be any other compiler flag which is supported by a newer version of GCC used to compile Gromacs but not supported by the older version used by CUDA. Thus if we want to support using different GCC version for main part vs CUDA, and add bootstrap code, then we need 3 sets of flags: std, no-arch flags, no flags not supported by older GCC version.

I think Roland's second part has got it nailed for this patch. This isn't about an optimization flag, but that we would need to maintain an entire separate configuration process for a different compiler version since any flag in theory might be "too new" in the future. Combined with some of the other ideas above, this sounds like it would at least turn into a lot of extra CMake code (which is bad per se), and also a maintenance nightmare for the future.

Considering gcc 4.7, 4.8 and 4.9 all work great both for Gromacs and the nvcc backend on my Mac (and I would assume Linux is even simpler), and those are completely free as well as very fast, I would vote against considering the modifications discussed here, simply from a code maintenance and CMake cleanliness point-of-view. It's a more than reasonable solution to require the same compiler for all parts of the code.

The different optimization levels is a different question, though.

I take Szilard's removal of himself as the assignee as a sign that we can reject this for now, but reopen if it turns out to be a recurring problem :-) (Hello, AVX-512)

That was just a sign of me not wanting to deal with the issue. :) The topic of different sets of compilation flags, multiple C++ compilers and ABI compatibility, and host/GPU compiler compatibility are just too complex and the entire discussion has drifted away from the original topic. I suggest opening a separate issue for these topics because the questions are still quite relevant, but neither very related to the issue description nor very timely to deal with right now.

Note that FindCUDA.cmake provides an option to use C (as opposed to C++) host compilation, which may simplify things somewhat... Not sure why that has not been used so far.
If no one is interested to do anything about this, then I suggest that at minimum, we add a warning if the CUDA host compiler is not the same as the main compiler, since that probably results in problems like in this issue.

We do have a warning already, but only for the case when we do not set the CUDA host compiler to CC because the two are known to not be compatible. In this case nvcc will pick the available system compiler (typically gcc) which will be compatible most of the time. Unfortunately even this got short-circuited by a recent CMake version (see #1248) but I'm in the process of fixing that.

The question is, how far should we go with the checking, should we also check what the user passed (which will require additional code to obtain information on the compiler passed in CUDA_HOST_COMPILER)?

I think that for the purpose of this issue, there's no need to know who set the host compiler. MPI compiler wrappers can make the comparison a bit tricky, though, but checking the flags may not be any easier, either (haven't investigated in detail, though).

Note that FindCUDA.cmake provides an option to use C (as opposed to C++) host compilation, which may simplify things somewhat... Not sure why that has not been used so far.

C host-code is not supported since CUDA 3.0, e.g. see:
https://gitorious.org/cmake/cmake/commit/d3034f9b561ab4a4d5cb4bc515d31d4770f192a8?diffmode=sidebyside

I think that for the purpose of this issue, there's no need to know who set the host compiler. MPI compiler wrappers can make the comparison a bit tricky, though, but checking the flags may not be any easier, either (haven't investigated in detail, though).
A simple compiler check could be done by comparing the --version output of the CMAKE_CXX_COMPILER and CUDA_HOST_COMPILER, you are right in that this will omit false positives if an MPI wrapper is used and this appends anything to the backend compilers --version output.

The reason why it matters who set the compiler is that in most cases the user won't set CUDA_HOST_COMPILER, but since CMake 2.8.10 we can't distinguish the case when the user set it and when FindCUDA set it. That's easy to work around, though, I'd just need to check before calling FindCUDA.

```
#38 - 06/24/2014 08:00 PM - Roland Schulz
Does any important MPI wrapper append anything? Can't remember having seen that. If not I would just compare the version.
```

```
#39 - 06/24/2014 11:02 PM - Szilárd Páll
Roland Schulz wrote:

Does any important MPI wrapper append anything? Can't remember having seen that. If not I would just compare the version.
```

The only MPI wrapper that AFAIK messed with the mpicc --version output was the Bull MPI - and even that observation was made, I don't have access to such a machine anymore.

```
#40 - 06/25/2014 12:48 AM - Roland Schulz
I don't think that's important. As long as it is just a warning. Let's just put a warning that using a different C++ and cuda-host compiler is unsupported based on a version check.
```

```
#41 - 08/05/2014 01:08 AM - Ben Brummel
I recently installed NVIDIA CUDA TOOLKIT V6.0 and attempted to install gromacs-5.0 using an Ubuntu 12.04 desktop with an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, and NVIDIA GeForce GTX 650. I received the following error after inputting "cmake .. -DGMX_BUILD_OWN_FFTW=ON -DGMX_GPU=ON" (and with "cmake .. -DGMX_BUILD_OWN_FFTW=ON -DGMX_GPU=ON -DCUDA_TOOLKIT_ROOT_DIR=/usr/local/cuda");

```
-- Try C compiler AVX2 flag = [-march=core-avx2]
-- Performing Test C_FLAG_march_core_avx2
-- Performing Test C_FLAG_march_core_avx2 - Success
-- Performing Test C_SIMD_COMPILES_FLAG_march_core_avx2
-- Performing Test C_SIMD_COMPILES_FLAG_march_core_avx2 - Success
-- Try C++ compiler AVX2 flag = [-march=core-avx2]
-- Performing Test CXX_FLAG_march_core_avx2
-- Performing Test CXX_FLAG_march_core_avx2 - Failed
-- Try C++ compiler AVX2 flag = [-mavx2]
-- Performing Test CXX_FLAG_mavx2
-- Performing Test CXX_FLAG_mavx2 - Failed
-- Try C++ compiler AVX2 flag = [/arch:AVX]
-- Performing Test CXX_FLAG_arch_AVX
-- Performing Test CXX_FLAG_arch_AVX - Failed
-- Try C++ compiler AVX2 flag = [-hgnu]
-- Performing Test CXX_FLAG_hgnu
-- Performing Test CXX_FLAG_hgnu - Failed
-- Try C++ compiler AVX2 flag = []
-- Performing Test CXX_FLAG_
-- Performing Test CXX_FLAG_ - Success
-- Performing Test CXX_SIMD_COMPILES_FLAG_
-- Performing Test CXX_SIMD_COMPILES_FLAG_ - Failed
CMake Error at cmake/gmxTestSimd.cmake:244 (message):
Cannot find AVX2 compiler flag. Use a newer compiler, or choose AVX SIMD (slower).
```
#42 - 08/05/2014 01:13 AM - Erik Lindahl

gcc-4.6 will never support AVX2, since it's simply too old. When it comes to CUDA, NVIDIA will eventually support newer compilers. In the mean time you could try hacking the line in the NVIDIA include file that tells you gcc-4.8 (or 4.9) is not supported, and remove the check for gcc version. In practice this usually works perfect.

Alternatively you can try compiling Gromacs with AVX_256 acceleration.

#43 - 08/05/2014 01:45 AM - Ben Brummel

Erik Lindahl wrote:

gcc-4.6 will never support AVX2, since it's simply too old. When it comes to CUDA, NVIDIA will eventually support newer compilers. In the mean time you could try hacking the line in the NVIDIA include file that tells you gcc-4.8 (or 4.9) is not supported, and remove the check for gcc version. In practice this usually works perfect.

Alternatively you can try compiling Gromacs with AVX_256 acceleration.

Thanks. I got it working with: "cmake .. -DGMX_BUILD_OWN_FFTW=ON -DGMX_GPU=ON -DCUDA_TOOLKIT_ROOT_DIR=/usr/local/cuda-6.0/ -DCMAKE_C_COMPILER=/usr/bin/gcc-4.8 -DCMAKE_CXX_COMPILER=/usr/bin/g++-4.8" after installing the 4.8 versions for both the gcc code and the g++ code.

#44 - 08/05/2014 03:00 AM - Mark Abraham

Szilárd Páll wrote:

Teemu Murtola wrote:

I think that for the purpose of this issue, there's no need to know who set the host compiler. MPI compiler wrappers can make the comparison a bit tricky, though, but checking the flags may not be any easier, either (haven't investigated in detail, though).

The reason why it matters who set the compiler is that in most cases the user won't set CUDA_HOST_COMPILER, but since CMake 2.8.10 we can't distinguish the case when the user set it and when FindCUDA set it. That's easy to work around, though, I'd just need to check before calling FindCUDA.

That's a further statement of fact, but it doesn't say why you think it is important we can tell who set the host compiler.
More generally, I am still of the opinion that supporting more than one compiler version per build tree is a poor use of our time. I still haven't seen an important use case where someone needs two different C++ compilers/versions. Those offered so far are not consistent with the usual advice of "use the most recent stuff you possibly can, hack around Nvidia's errors if you need to."

I'm also still in favour of an all-purpose CMake (and mdrun?) warning that using gcc older than (say) 4.7 is basically wasteful in all cases. This is off-topic to restricting proceedings to one C++ compiler, but somehow benchmarking with gcc-4.4 got introduced to the thread earlier.