GROMACS - Task #1745

Moving to C++11 after Gromacs-5.1

05/22/2015 04:49 PM - Erik Lindahl

Status:	New
Priority:	Normal
Assignee:	
Category:	
Target version:	
Difficulty:	uncategorized

Description

We've had some discussions both in Gerrit, Redmine, and among the developers that have mostly (?) converged towards requiring C++11 for the future. This is also related to the discussion in #1732 about how far we want to go in using options to silence compiler warnings that are due to shortcomings in the compilers rather than Gromacs. This is not a complete list, but at least some thoughts to get the discussion started.

Overall, this is a balance we want to strike. Using recent language versions and ignoring less common compilers can save developer time, make code cleaner, and give us a faster development cycle on standardized hardware. However, it can/will also lead to significant porting problems, problems in using hardware where a non-perfect vendor compiler is required for good performance (e.g. embedded systems, special hardware projects, etc.), and relying too much of a mono- (or bi-) culture of compilers can fool us into thinking the code is fine just because both of those compilers swallow code that doesn't quite follow the standard.

A) Gromacs has a long history of being very portable, and this has helped us in a lot of projects, so this is a feature we want to maintain. However, at the same time we don't want to become the equivalent of FORTRAN buffs. The state of compilers has matured, open source compilers are finally competitive performance-wise, and there are tremendous amounts of hard work that go into new language standards - we shouldn't just discard that.

B) I don't think we should move to new language versions automatically "just because we can", but rather base it on (a) features that will be a significant help to us, and (b) that the standard is mature. For C++11 the standard will be five years old next year, and I think that's a good balance when we can point the finger at compiler vendors for not doing their homework. So, my first suggestion would be that we should wait ~5 years before we require a language standard as mandatory.

C) We've have tested a bunch of different compilers the last 1-2 years, and my overall experience is that many of them are VERY buggy when it comes to exceptions, but when it comes to language features things have been better. Both GCC and Clang already have full C++14 support, and vendors appear to be picking up the pace. IBM has for instance switched to the LLVM frontend, and should thus get complete C++11/14 support quite soon. For all the non-x86 systems I've tested, Fujitsu/K-computer is the only one where a vendor compiler has provided better performance. That is also the only major HPC vendor where we do not at all have any C++11 support right now, but by 2016 this system will be very old, and not worth the extra effort for us, IMHO.

Specifically, there are a handful of C++11-features that might make development life easier:

0) Extern templates should make it easier to write the nonbonded kernels

1) Even when using TBB, the native threading, atomics, and memory model support in C++11 is potentially very valuable.

2) Using lambdas and auto keywords will make it easier/cleaner to create small functions for TBB.

3) nullptr, strongly types enums, smart pointers, static_assert and type traits can help avoid bugs

4) Constexpr and move semantics can improve performance

5) Better initializer lists might help make code cleaner

However, at some point it will also get tedious for developers to keep up with long lists of what language standards they are allowed to use, so I'm somewhat inclined to simply say that we require C++11 after Gromacs-5.1, but then we can try to avoid going overboard with language features "just because we can".

Thoughts? Here too I would imagine that we might have to put limits to the amount of fancy stuff to use to keep code readable.

Related issues:

Related to GROMACS - Task #1390: manage C+11 support and CUDA better	**Closed**

Associated revisions

Revision 093190da - 11/20/2015 01:09 PM - Mark Abraham

Split wallcycle interface

Separated header for components that relate to accumulating counts
from end-of-run reporting. The former benefits from exposing a minimal
interface, e.g. so CUDA code isn't potentially exposed to C++11
headers.

Refs #1745

Change-Id: I70517ac104f002b7b4e61a3aaa26acff5193361c

Revision 4ce2aa50 - 05/10/2016 04:18 PM - Mark Abraham

Facilitate linking with libcxx

Using recent clang static analyzer versions seems to be easier with
libcxx, which we should anyway support for building and testing.

Reworked the testing for C++11 support, since it is sensible to first
test the compiler, and then the standard library. This helps users
diagnose problems. Converted this code to a function (for better
scoping), added some docs, and made the semantics clearer. Added
some explicit testing for other non-library C++11 functionality.

Introduced GMX_STDLIB_CXX_FLAGS, so that all the linked executables
can have their sources compiled with any compiler flag that might be
required. Alternatives like requiring the user to modify
CMAKE_CXX_FLAGS, or adding COMPILE_FLAGS properties to targets didn't
seem great. The latter also triggers clang to issue warnings for
source files that are still C (group kernels and TNG).

Introduced GMX_STDLIB_LIBRARIES, so that linking can proceed
correctly.

For example, the check for C++11 support needs to be passed a library
to link during the try_compile(), and the only reliable way for the
user to do that before this patch was to add the linker flag to
CMAKE_CXX_FLAGS, which then leads to clang warning about the unused
linker flag as it compiles each source file. The GMX_STDLIB_*
mechanisms probably also permit users to build against different
versions of GNU libstdc++, which may be useful on distributions like
CentOS, because CMake has no mechanism at all for this.

Updated the install guide to clarify how to choose a standard library
in the various cases. Updated the guide for using GROMACS as a library,
and the template README.

Fixed issue where the template did not have C++11 compiler flags
propagated properly. The template now builds correctly, via both
Makefile.pkg and cmake, both with normal default libstdc++ and libc++
selected via this mechanism and propagated to the installed build
system for the template.

Fixed issue where SIMD suggestion would produce a garbage suggestion
when the linking failed and OUTPUT_SIMD was left unset.

Refs #1745,#1790

Change-Id: Ieef3b47de5c1a00a203baa1b34ebf70535cf5ff0

History

#1 - 05/22/2015 05:01 PM - Erik Lindahl

We have also discussed compiler warnings. Getting rid of warnings is important both because it prevents potential future bugs, and because a clean output will draw our attention to the bugs that really do exist.

Apart from real errors, there are some warnings that warrant fixing, for instance when we use macros or other hacks in functions that throw the compiler off so it doesn't understand the logic. However, in general it is likely better to fix that on the language level instead of adding tons of flags.

It is probably not worth working around warnings that are entirely due to compiler bugs since that either makes the code or our build system a whole lot more difficult to understand.

When it comes to compilers and systems that we want to support, I would like to think in terms of three tiers:

- Tier 0 is stuff that we have in gerrit. Things are guaranteed to work for every single patch.
- Tier 1 is stuff that we do test before a release. This might include BlueGene/Q, ARM and Power8 right now.

Everything else will be best-effort. However, I wouldn't be entirely comfortable saying that we only support compilers version X,Y,Z - that might rapidly lead to a situation where we ignore potential bugs just because they don't affect our supported-system list. When things don't work we should try to fix them, unless it's obvious that a particular compiler has had a number of bugs (meaning the next one is also likely to be a compiler bug). At that point we can point to the vendor instead.

#2 - 05/24/2015 06:26 AM - Teemu Murtola

Probably one critical issue related to the C++11 support is CUDA. As it is now, any build with GMX_GPU=ON disables any C++11 features because at least it has not been trivial to get things to work with nvcc (but that might get easier if we don't need to be conditional, but I do not know the details here). Also, if/when we require C++11, that means that we need to drop support for a lot of older compilers, which means that older CUDA versions will not work, either, without the user changing the strict compiler compatibility checks in the CUDA headers. We need to be clear on what we need to support on this front.

#3 - 05/24/2015 08:34 AM - Erik Lindahl

Good point, but right now I don't think we use any C++ at all for the CUDA parts. Perhaps one option is not to use the C++11 flags for the .cu files?

#4 - 05/24/2015 08:54 AM - Teemu Murtola

Erik Lindahl wrote:

> Good point, but right now I don't think we use any C++ at all for the CUDA parts. Perhaps one option is not to use the C++11 flags for the .cu files?

Not sure whether there is some C++ code there or not right now, but certainly there will be (and surely nvcc generates some, since it requires a C++ compiler). At least in headers included from CUDA files will have it, unless we are *very* strict on either splitting out CUDA data structures from everything else, or disallowing all C++ in various structures.

And if there is C++ in the headers, compiling them with different standard options could cause issues. Of nothing more, then on the maintainability side when there's a magic, ever-fluctuating set of headers (those that are transitively included by CUDA) where C++11 causes compilation errors on some configurations.

#5 - 05/24/2015 09:11 AM - Teemu Murtola

And just using different do flags is not enough to keep supporting old CUDA versions; we would also need to use different C++ compilers for different parts of the code, which is even more likely to lead to trouble.

#6 - 05/24/2015 09:12 AM - Erik Lindahl

Unfortunately it's worse. I just realized that if we want to use extern templates and stuff for the normal kernels we don't want a different setup for the Cuda ones. That could be a dealbreaker. Unless it is possible to compile with Cuda 7.0 and use the driver of Cuda 5.5 we would not be able to run on titan and a bunch of other supercomputers, since they have standardized on Cuda 5.5 for the lifetime of the machine.

#7 - 05/24/2015 09:41 AM - Roland Schulz

Cuda 5.5 supports gcc 4.7 and it supports almost all of C++11 (especially of the language features): https://gcc.gnu.org/gcc-4.7/cxx0x_status.html . With the only big exception being the concurrency support. That should not be too hard to work around.

#8 - 05/24/2015 10:01 AM - Erik Lindahl

That will likely only help for the host-only code for one compiler, not kernels or any other compiler the user might use (since C++11 isn't supported). Here's a one recent example where Cuda-5.5 & gcc (both 4.7.2 and 4.8) seems to fail on some very simple C++11 stuff:

http://stackoverflow.com/questions/21457974/can-i-use-c11-in-the-cu-files-cuda5-5-in-windows7x64-msvc-and-linux64-gc

#9 - 05/27/2015 05:53 AM - Teemu Murtola

On the features of C++11 that we might want to initially use, here are some thoughts:

- extern templates, yes. That should be a relatively easy requirement, since I think at least gcc has supported them from something like 3.3.
- lambdas is a must for writing code easily with TBB, and will help with other such tasks as well (e.g., with STL algorithms and for code that expects a simple callback). The code is possible to write without, but it will be very verbose and harder to read. We need to be conservative with the use of lambdas, though; I've heard that MSVC is at least for now horrible at compiling nested lambda expressions (my colleague managed to write a single C++ file that took five minutes to compile).
- std::shared_ptr and std::unique_ptr will be required if we want to get rid of the current #ifdefs. With exceptions, we will need smart pointers, and these are the ones the standard provides. We can go with using just boost::shared_ptr, but that loses several benefits from the move semantics (see next item).

- Move semantics (and std::unique_ptr) is not just about efficiency (although they will make it a bit easier to write efficient code), but even more about clear ownership of data, and they allow many more options for implementing an API for a class that owns resources.
- nullptr is nice, but only a minor convenience. We should start using it if all the compilers we would otherwise use supports it, but it will not be a dealbreaker.
- The same goes for strongly typed enums and static_assert.
- constexpr is hard; e.g., no released version of MSVC yet supports it (VS2015 might improve things, when it comes out).
- If we enable C++11, Google Test will start using variadic templates, so in order to compile the tests, we will need that support from the compiler, even if we don't want to use it ourselves.

#10 - 05/28/2015 05:03 PM - Erik Lindahl

For now I don't see any simple way around the issues that Cuda-5.5 only supports pre-C++11 versions of GCC, so unfortunately it appears that five years isn't enough for the vendors to get their act together. Thus, I think we might have to live with C++98 at least for another major version.

#11 - 05/28/2015 05:57 PM - Roland Schulz

Why is the CUDA issue a reason to not use C++11 at all? It seems we can at least use C++11 everywhere other than the header/source files also used by CUDA. And isn't it true that most shared headers are only used by the host-only code for which a work-around is possible? What's the plan on sharing the code of the kernels between CUDA and CPU? You just mentioned this in note-6 but I don't think this has been described anywhere. It seems it only makes sense to use the same kernels for CUDA/non-CUDA if the advantage of the shared code (i.e. retiring the current CUDA kernel) outweighs the advantages of using C++11 in the new kernels.

#12 - 05/28/2015 06:30 PM - Roland Schulz

At least CUDA 6.5 (which has undocumented support for C++11) should be available on Titan within a year and thus shouldn't be a problem for the next major version. This hopefully means that other Cray XK7 will be updated the same way.

#13 - 05/28/2015 06:34 PM - Erik Lindahl

It's not just the kernels, but all headers that they **might** depend on in the future. The second we want CUDA for bonded interactions (happening), PME, LINCS, etc., most of the headers describing data structures for those files might also have to be compatible, and over the next 1-2 years we might have to write a lot more such code since CPUs are not advancing at the same speed as GPUs. Although C++11 would be a nice convenience, it is not worth it if we buy that convenience at the price of forcing the few developers who write Cuda code to start maintaining their own versions of lots of datastructures, headers, etc.

It's good if Titan is upgraded, but unfortunately that is just one machine, and until that upgrade has happened we won't even be able to test and benchmark any development version there.

#14 - 05/28/2015 07:05 PM - Roland Schulz

Even if exclude all headers (or all those you just listed) we could still use lambdas for TBB in the source files. Which I think is more than just convenience because it makes the code much easier to read and much easier to convert from OpenMP.

Also if Titan will be upgraded than this means that any other XK7 can (I admit "can" doesn't mean will) be upgraded. I think the current availability of CUDA 6.5 on Titan has no impact. We can test CUDA 6.5 on other clusters. There is no reason to think that performance or C++11 support of Cuda 6.5 on Titan (or other Crays) will be any different than CUDA 6.5 on a different machine.

I only mentioned the kernels because if we have have separate implementations, then we can use extern and constexpr because it wouldn't be shared. And then it would depend on whether extern/constexpr is more than convenience (not sure).

#15 - 05/29/2015 12:37 PM - Teemu Murtola

- Related to Task #1390: manage C+11 support and CUDA better added

#16 - 06/03/2015 07:56 PM - Szilárd Páll

A few general points that came to my mind:

- The age of a standard is not the most representative aspect to characterize its readiness for adoption (see Fortran 2003 - unfair comparison, but still...). So using a ~5y rule of thumb for requiring C++11 may be worth reconsidering. Note that I'm not claiming whether it's well enough adopted/supported by compilers or not.

- We will have to be willing to work around issues (bugs, incompatibilities) - even OpenMP support required some non-trivial amount of effort to come up with awkward solutions to avoid warnings, bugs, etc. The question is where will be the balance and whether portability will be preferred or not.

- The core libgmx_core vs kitchen-sick libgmx split has not been discussed much lately, but it is becoming relevant again; specifically, if a reasonable separation did exist, adoption of C++11 would be less of an issue: if one wants the full gmx install we require e.g. gcc 4.9 or whatever is most convenient, otherwise you can compile an mdrun (and libgromacs_core) with a much better chance of success even with an older or more "exotic" compiler.

- Re TITAN & CUDA etc.: Regardless of what is available on TITAN, CUDA 5.5 is <3 years old and 6.0 is only a bit more than 1 year old. Dropping compatibility with these versions seems quite early to me.
  Unrelated, but it's strange that CUDA 6.5 has been rolled out a month ago on Piz Daint (so Cray does have the stuff ported to CLE), but ORNL will take their time?

#17 - 06/03/2015 08:33 PM - Roland Schulz

Szilárd Páll wrote:

> A few general points that came to my mind:
>
> - The age of a standard is not the most representative aspect to characterize its readiness for adoption (see Fortran 2003 - unfair comparison, but still...). So using a ~5y rule of thumb for requiring C++11 may be worth reconsidering. Note that I'm not claiming whether it's well enough adopted/supported by compilers or not.

I agree. There are C++14 features which are as well supported as some C++11 features.

> - The core libgmx_core vs kitchen-sick libgmx split has not been discussed much lately, but it is becoming relevant again; specifically, if a reasonable separation did exist, adoption of C++11 would be less of an issue: if one wants the full gmx install we require e.g. gcc 4.9 or whatever is most convenient, otherwise you can compile an mdrun (and libgromacs_core) with a much better chance of success even with an older or more "exotic" compiler.

Not sure how useful that is. Things which have been mentioned by multiple for being important (e.g. extern templates for kernels, lambda for TBB, unique_ptr to remote gmx_unique_ptr) would all affect core. And it would add extra complexity for developers if we had different rules for different parts. So we should only do it with significant advantage.

> - Re TITAN & CUDA etc.: Regardless of what is available on TITAN, CUDA 5.5 is <3 years old and 6.0 is only a bit more than 1 year old. Dropping compatibility with these versions seems quite early to me.
>   Unrelated, but it's strange that CUDA 6.5 has been rolled out a month ago on Piz Daint (so Cray does have the stuff ported to CLE), but ORNL will take their time?

I think it is reasonable to drop CUDA 5.5 and 6.0 support in a year. By that time CUDA 6.0 is >2 years old and more importantly two newer versions (and likely 3, assuming 7.5 comes out within a year) are out. Sure it would be nice to support older versions. But isn't it anyhow better for performance if supercomputer centers have a recent CUDA version? Thus this should generally not be a problem. ORNL tried to update to CUDA 6.5 a while ago but had some issues and thus is still working on it. Ideally the CUDA and C++11 issue wouldn't be coupled. But at least for some of the benefits (move, unique_ptr, not sure what the plan is about using sharing more code between CUDA and CPU kernels and how that effects extern template) it would be extra work to guarantee that CUDA can work without C++11 (and thus with <6.5).

I suggest we go either of two ways:

- We want the highest compatibility with old compilers and CUDA. Then we only go for few most important C++11 features in source files (e.g. lambdas for TBB). Whether we also use C++11 in kernels would depend on whether we gain more by sharing code between CPU and CUDA or whether we gain more by extern template and constexpr.
- We want highest benefit from C++11. Then we also allow C++11 in header files and require CUDA 6.5. This would allow us to use e.g. unique_ptr and move.

#18 - 06/05/2015 07:11 PM - Erik Lindahl

I think we'll get ourselves in a mess if we need to have a long list of individual features that can or cannot be used by developers, and it would also make life difficult e.g. for a hardware vendor to know what Gromacs requires. As an analogy: Even though almost all compilers support the GNU extensions, we don't want to require it since it's easier to just say that we e.g. require C++98, period.

The problem with lambdas is that it is appears to be the buggiest/most difficult C++11 feature to get right, so by requiring lambdas we'll effectively require 100% complete and stable C++11 support.

I would still like to do the move in the future, but that will have to wait until the vast majority of HPC centers actually have completed their CUDA upgrades (or we won't even be able to test the development work there), so I would suggest putting this on ice for a while until that happens.

#19 - 06/16/2015 08:52 AM - Roland Schulz

For developers we anyhow want a list of allowed features to follow our approach of Simple C++. For Hardware vendor we can say we require C++11 with an asterisk that we currently only use a subset. And if we don't use C++11 in headers (for now) it has no effect on CUDA.

#20 - 06/16/2015 09:30 AM - Erik Lindahl

I think it's better to wait a year until Cuda-5.5 is no longer a problem, and we can make more of a clean switch.

For now we would need to stay away completely from C++11 in virtually all headers, since we don't know ahead of time if somebody needs to port a new routine to CUDA, and then I don't see much point in just having C++11 inside a few source files if we can't use it in interfaces.

Second, I don't want to repeat the situation from last year when I spent several weeks fixing Gromacs so compiles worked with PGI, Pathscale, Fujitsu and XLC. Thus, when/if we decide to switch to C++11 we also need a plan for how everybody who wants to use C++11 features also shares the effort in working around shortcomings in those compilers (Fujitsu is claiming to support C++11 now, but I'm not sure if it's available on K yet).

#21 - 06/16/2015 09:41 AM - Roland Schulz

I agree we would want to either allow it in all or no header. But I see no disadvantage to allow it in source files earlier than in header files.

I agree that many features on e.g. Teemu's list of useful features, require to use them also in the header. But at least lambda for TBB is not one of them.

#22 - 06/16/2015 09:43 AM - Teemu Murtola

If you really want to avoid a situation like that, you need to make those commercial compilers available for people that you somehow expect to help... And by far best way to do that is to have them as part of Jenkins verification. There was a lot of talk to include some there, but since nothing has happened in this long time, it seems that it is not that important for anyone.

And as far as I remember, the only more tricky issue last year was caused by a bug in the Fujitsu compiler; you cannot reasonable expect that people anticipate all possible bugs in all possible not-yet-available compilers...

#23 - 06/16/2015 09:55 AM - Erik Lindahl

xlc is available as a free trial download from IBM, and pathscale is also freely available. For PGI we have a license for now (that I hope we can renew for free, but there's also a trial download), and as we've done before it's possible to arrange K access through me. The reason we could not include them in Jenkins is that there were a few long-lived bugs related to exceptions that caused unit tests to fail. That's more of a bonus, though: Priority #1 is to make sure Gromacs itself compiles.

I'm certainly not expecting people to anticipate bugs in all compilers, and obviously we will have commits that fail on compilers not in Jenkins. However, the reason for bringing up this point now is that it's important that everybody will be willing to spend some time working around issues with C++11 constructs we introduce (even when it is a bug in a compiler somebody doesn't use him/herself). If that is not the case, I'm completely against allowing any C++11 :-)

#24 - 06/16/2015 01:13 PM - Teemu Murtola

A trial download doesn't really qualify; what if it takes more than the 30/60-day trial period to fix anything? Or if something comes up after a person has already used their trial period? And neither xlc or Pathscale is available for OS X, so setting up a VM or finding separate hardware to run Linux on does not really sound appealing for volunteers like me.

What use is a compiler that cannot produce code that we can verify? Are you sure that it is only the tests that fail, and not actual Gromacs code (e.g., the selection code, which you probably did not test, but is the only part that uses a significant amount of exceptions)? And if the priority is to just make Gromacs compile and to not actually work, why not just add them to Jenkins such that they do not run the tests?

Of course the expectation is that people will help (and I think I did my part last year as well, answering all your questions, even if I did not have time to do actual fixes on the timeframe you wanted them). But that agreement also needs to have some balance, in that the immediate response to something not compiling on an exotic compiler is not that everything must be rolled back immediately; if the compiler is too buggy or the issue simply not fixable because it violates our initial assumptions about what C++11 we want to use, it *has* to be an option to not roll back a year of work. If we cannot agree on that, I'm completely against contributing anything that even remotely touches C++11. ;)

#25 - 06/16/2015 01:28 PM - Erik Lindahl

Well, getting a free email on the internet isn't exactly difficult today, and all developers can have access to the Linux machines in Stockholm if they don't already have it. For xlc I even think there are free test drives online.
Both pathscale and PGI had problems correctly catching exceptions. Bad as that is, it is not something that affects normal usage, and they pass all regression tests fine.

However, it is **extremely** likely we will hit a number of compiler issues the second we move to C++11, and unless we have a clear buy-in from developers who want to use C++11 that they too will be willing to spend time working around these issues, we are not ready to move to C++11.

#26 - 06/16/2015 08:19 PM - Roland Schulz

Do we care for those compilers about any version other than the most recent?
The latest version of PGI and Pathscale seem to have e.g. lambdas. But not previous ones.
Which xlc is required? For BlueGene the compiler is still 12.x. The Linux one is in a few days supporting most of C++11:
http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=ca&infotype=an&appname=iSource&supplier=897&letternum=ENUS215-218
Do I assume correctly that for the Fujitsu it is OK to wait until it is available on K? Especially because you mentioned before that K computer support wouldn't be important anymore for the next version.
It would be nice if one of the machines in Stockholm had all 3 compilers installed. So that not every one needs to go through the process of installing the compilers.

#28 - 07/30/2015 09:10 PM - Erik Lindahl

Given that we now also need OpenCL support, we need to use a slightly different strategy since C++ support is not even on the horizon there. I think it's going to be easier to have a single setup for all kernels, and avoid having that depend on fancy C++ features, but there's nothing that prevents us from requiring C++11 in other parts of the code if we can come up with a reasonable way to isolate it (so we don't depend on C++11 for basic data structures required in code that we need to port e.g. to OpenCL).

So, I don't think the specific CUDA version is the dealbreaker, but that we need to create this isolation no matter what C++ version we rely on.

In other words, I think we can go ahead and require C++11 for higher-level code. Any protests?

#29 - 07/30/2015 09:48 PM - Roland Schulz

Erik Lindahl wrote:

> Given that we now also need OpenCL support, we need to use a slightly different strategy since C++ support is not even on the horizon there. I think it's going to be easier to have a single setup for all kernels, and avoid having that depend on fancy C++ features

You mean C++ or C++11? As I wrote at #1758 the AMD implementation and the OpenCL 2.1 provisional spec have C++ template support. I agree with your suggestion for C++11. Templates for kernels can be discussed on the other issue.

#30 - 07/30/2015 09:50 PM - Erik Lindahl

There is NO way we can start requiring complete OpenCL 2.1 in the next few years given that some compilers still don't support 1.2, and on top of that the implementations are buggy :-)

#31 - 07/30/2015 09:53 PM - Erik Lindahl

Want to have a go at a patch? I guess we'd have to disable a couple of Jenkins configurations...

#32 - 07/30/2015 10:02 PM - Roland Schulz

But right now the only useful OpenCL target is AMD and it has support for C++ templates. So we wouldn't loose anything useful by requiring C++ template support. Or am I mistaken and the NVIDIA OpenCL target has any value to users? And in the future (probably in time for next release for e.g. Intel, a potential useful target) it would be supported for other targets.

What should a patch contain? Just cmake? Enable C++11 flag (when applicable) and testing the features we require? Which are those?
- unique-ptr + move (our current gmxTestCXX11)
- auto
- lambda

#33 - 07/30/2015 10:16 PM - Erik Lindahl

No, if I understand you right a *provisional* specification of a *future* standard includes support for templates. Before we can require that, we need (1) the specification to be ratified, (2) it must be available in compilers, (3) the support must be reasonable bug-free, and (4) we need to be ready to deprecate support for all earlier OS versions. I would for instance not imagine OpenCL 2.1 to be available on OS X until the next-next version in late 2016. A big point with OpenCL is the portability, so long-term I would definitely like us to support it both on AMD, NVIDIA and Intel.

No matter what happens to OpenCL, I think there's a pretty clear trend for all these special languages that they first support C, later C++, and finally all C++ features. We have no idea what new hardware or special languages that we might rely on in the future, so in any case I think it's a very good idea to be really restrictive and at least make it possible to write our innermost kernels (not just the nonbonded ones) in plain C.

I see no real point in exhuastively testing every feature we use (just as we don't test all C99 or C++98 features), at least until we see it breaking.

Testing lambdas and some other features that are nontrivial to implement might be a point, though!

#34 - 07/30/2015 10:26 PM - Roland Schulz

Yes the C++ template feature is part of a future standard. But this feature is already implemented for the most important implementation (AMD). The way I see it: If we use C++ templates now, we get an immediate adventure of being able to clean up the kernels without any downside to the user (because the only useful implementation already supports it) and without the disadvantage of other approaches such as Python scripts. Then we will get the advantages of portability over time. Of course we could also only use templates for the CPU kernels and then use them only for OpenCL later. And get the advantage of the shared approach only later.

We might want to wait testing for lambdas until we actually use them. I suspect that if we don't use TBB that then we don't need them anytime soon.

#35 - 08/01/2015 04:06 PM - Szilárd Páll

- Private changed from No to Yes

#37 - 08/01/2015 04:07 PM - Szilárd Páll

- Private changed from Yes to No

#38 - 08/03/2015 04:56 PM - Mark Abraham

> - Re TITAN & CUDA etc.: Regardless of what is available on TITAN, CUDA 5.5 is <3 years old and 6.0 is only a bit more than 1 year old. Dropping compatibility with these versions seems quite early to me.
>   Unrelated, but it's strange that CUDA 6.5 has been rolled out a month ago on Piz Daint (so Cray does have the stuff ported to CLE), but ORNL will take their time?

FYI now available https://www.olcf.ornl.gov/2015/07/29/olcf-upgrades-system-to-cuda-6-5/

(I was on my phone the first time I posed this, and it seems impossible to remove the "private" tag that got added then...)

#39 - 08/16/2015 04:32 PM - Erik Lindahl

Just figured I should update this discussion based on the testing I did with using C++11 for a new random engine class on top of Roland's change to c++11:

1. Modern compilers seem to be pretty decent. Gcc-4.8 and later works perfectly, as does icc-14 and clang-3.4. Even Pathscale and PGI are fine (there are still some bugs that will hopefully be fixed in the compilers soon, but those are unrelated to C++11).

2. Gcc-4.6.4 barely passes. I stumbled into problems with a buggy STL header, and aliases are not supported.

3. MSVC 2013 lacks support for extern templates, and it is still only partial in MSVC 2015. I could not get it working for a static const class member that is also a template specialization, but maybe it will work for something simpler like functions.

However, I guess the good news is that we have the two worst compilers (gcc-4.6.4 and MSVC 2013) in gerrit. I have talked to Daniel at PDC who will try to arrange having xlc installed on the Power8 node we have access to, and then I think we would have very good local coverage.

So, given that, I think it's reasonable that we simply allow C++11 features that build correctly in Gerrit (although we might have to change that if we find some other compiler to be problematic).

#40 - 09/30/2015 10:48 PM - Erik Lindahl

... and as of a few minutes ago, we're now officially a C++11 project. However, we should keep this thread open to discuss what features we can/cannot use.

#41 - 10/01/2015 03:27 PM - Szilárd Páll

Erik Lindahl wrote:

> ... and as of a few minutes ago, we're now officially a C++11 project. However, we should keep this thread open to discuss what features we can/cannot use.

I have to admit, I have conflicting feelings about this, and I'm not sure if this is a sad or happy moment.

For many, including me, this is definitely a challenge. Hence, it would be good to not forget that many of us still think in C. C++, especially with the "11" part and the stuff it comes with can easily end up bringing difficulties and become a new barrier to contribution and review.

Hence, it could be beneficial to spend some time, perhaps even devote some videoconf-based workshops the coming months to bring each other up to speed with the new coding standards, techniques, language features etc.

#42 - 10/01/2015 03:48 PM - Carsten Kutzner

I agree. Good idea!

#43 - 11/16/2015 06:17 PM - Szilárd Páll

I would appreciate some feedback on this from the proponents / those working actively on of C++11 adoption in GROMACS. The codebase is shifting quickly and there has been little to no discussion in the wider developer-base about new guidelines, coding standards, the challenges in adopting C++11, etc. I'm concerned that continuing like this, some (many?) will experience an overall decrease in productivity and ease of contribution.

#44 - 11/16/2015 06:30 PM - Szilárd Páll

Erik Lindahl wrote:

> I think it's better to wait a year until Cuda-5.5 is no longer a problem, and we can make more of a clean switch.
>
> For now we would need to stay away completely from C++11 in virtually all headers, since we don't know ahead of time if somebody needs to port a new routine to CUDA, and then I don't see much point in just having C++11 inside a few source files if we can't use it in interfaces.
>
> Second, I don't want to repeat the situation from last year when I spent several weeks fixing Gromacs so compiles worked with PGI, Pathscale, Fujitsu and XLC. Thus, when/if we decide to switch to C++11 we also need a plan for how everybody who wants to use C++11 features also shares the effort in working around shortcomings in those compilers (Fujitsu is claiming to support C++11 now, but I'm not sure if it's available on K yet).

It seems that your former comment did not get enough attention and discussion (I guess your latter point took over). However, I would like to revive it because C++11 is becoming an issue for CUDA development.

We ran into issues because timing/wallcycle.h now contains an #include <array>. This makes it impossible to use cycle counters from CUDA files which is a limitation that would be best to eliminate. More generally, I see a clear danger of assuming that any header that's not included *now* in a .cu file can include C++11 code, but that's IMO not going to work. Hence, I strongly support Erik's proposal of no C++11 in headers - except perhaps in high-level (e.g. cmdline manager module) and tools-related code.

#45 - 11/18/2015 03:59 PM - Mark Abraham

Szilárd Páll wrote:

> I would appreciate some feedback on this from the proponents / those working actively on of C++11 adoption in GROMACS. The codebase is shifting quickly and there has been little to no discussion in the wider developer-base about new guidelines, coding standards, the challenges in adopting C++11, etc. I'm concerned that continuing like this, some (many?) will experience an overall decrease in productivity and ease of contribution.

This all makes sense, but so does developers actively participating in their own education. We've been on a path to C++ for years, and some devs are still proposing new code that manages C-style dynamic arrays, etc. I can probably find several useful small projects to help people get comfortable with writing classes (e.g. adding some unit tests), or using std::vector, using std::string, etc. without biting off the full complexity of also getting high performance. Likewise, doing code review of C++ code others write is a great education - e.g. lots of stuff Erik has in gerrit. I know I learned by doing, by reviewing Teemu's code, and watching lots of video casts from C++ conferences. :-)

I don't think it is reasonable for anyone to give anything approaching a graduate-level course in C++ programming. I can suggest some good "introduction to useful things in C++11" online videos. But until people all have some experience in trying to write C++, they're not going to be able to participate in discussions of guidelines, standards and challenges.

There are quite a few things none of us know yet, such as how effective templating will be for kernels, which is a challenge for trying to write standards and guidelines...

The coding guidelines we have are at http://jenkins.gromacs.org/job/Documentation_Nightly_master/javadoc/dev-manual/index.html. We should definitely add some C++11 things, such as

- minimize inclusion of C++11 headers in headers used by other modules (and installed headers)
- avoid combining fanciness (static data members, template classes, OpenMP and virtual functions can each be fine, but a class that uses all of them is in practice a high risk for non-portability)
- don't depend on metadata propagation (e.g. struct elements and captured lambda parameters tend to have restrict and align qualifiers discarded by compilers)
- plan for code that runs in compute-sensitive kernels to have useful data layout for re-use, alignment for SIMD memory ops
- recognize that some parts of the code have different requirements - compute kernels, mdrun setup code, high-level MD-loop code, simulation setup tools, and analysis tools have different needs, and the trade-off point between correctness vs reviewer time vs developer time vs compile time vs run time will differ
- anything else?

#46 - 11/18/2015 04:08 PM - Mark Abraham

Szilárd Páll wrote:

> Erik Lindahl wrote:
>
>> I think it's better to wait a year until Cuda-5.5 is no longer a problem, and we can make more of a clean switch.
>>
>> For now we would need to stay away completely from C++11 in virtually all headers, since we don't know ahead of time if somebody needs to port a new routine to CUDA, and then I don't see much point in just having C++11 inside a few source files if we can't use it in interfaces.
>>
>> Second, I don't want to repeat the situation from last year when I spent several weeks fixing Gromacs so compiles worked with PGI, Pathscale, Fujitsu and XLC. Thus, when/if we decide to switch to C++11 we also need a plan for how everybody who wants to use C++11 features also shares the effort in working around shortcomings in those compilers (Fujitsu is claiming to support C++11 now, but I'm not sure if it's available on K yet).
>
> It seems that your former comment did not get enough attention and discussion (I guess your latter point took over). However, I would like to revive it because C++11 is becoming an issue for CUDA development.
>
> We ran into issues because timing/wallcycle.h now contains an #include <array>. This makes it impossible to use cycle counters from CUDA files which is a limitation that would be best to eliminate. More generally, I see a clear danger of assuming that any header that's not included *now* in a .cu file can include C++11 code, but that's IMO not going to work. Hence, I strongly support Erik's proposal of no C++11 in headers - except perhaps in high-level (e.g. cmdline manager module) and tools-related code.

Indeed, that's a good point, but actually I think the lesson different. The internal data representation used by the cycle-counting code, and the representation used during sum-and-print (which is the current use of std::array) should not leak into the header that client code uses. It's wrong whether I'd used std::vector, std::array, or C-style array.

This is simple to fix in the current code (move either the enum into its own header, or wallcycle_sum and wallcycle_print into its own header). I effectively have this fixed already in my private branch, but review of https://gerrit.gromacs.org/#/c/5244/ would be welcome, first.

#47 - 11/18/2015 07:29 PM - Szilárd Páll

Mark Abraham wrote:

> Indeed, that's a good point, but actually I think the lesson different. The internal data representation used by the cycle-counting code, and the representation used during sum-and-print (which is the current use of std::array) should not leak into the header that client code uses. It's wrong whether I'd used std::vector, std::array, or C-style array.

The two lessons seem more or less orthogonal: your suggestion has a narrow scope that does not address the general issue Erik originally referred to and I echoed.
I have the feeling that this general issue could grow big and annoying in the future, so it would be worth to avoid as much as possible. A slightly stronger wording of your #1 proposed addition to the guidelines and careful review should help. So does if everybody is well-aware of this potential pitfall and not just the two of us.

The solution specific to this case has occurred to me, but I just wanted to try a simple thing and did not expect to have to refactor code to achieve that, so I used the 5.1 code-base instead.

#48 - 11/18/2015 09:45 PM - Mark Abraham

Szilárd Páll wrote:

> Mark Abraham wrote:
>
> > Indeed, that's a good point, but actually I think the lesson different. The internal data representation used by the cycle-counting code, and the representation used during sum-and-print (which is the current use of std::array) should not leak into the header that client code uses. It's wrong whether I'd used std::vector, std::array, or C-style array.
>
> The two lessons seem more or less orthogonal: your suggestion has a narrow scope that does not address the general issue Erik originally referred to and I echoed.

No, the principle I mentioned is of much larger scope, because following it goes a long way to removing the need for an active principle of avoiding C++11 in headers.

The purpose of a header file is to declare an interface to some code (e.g. between modules in GROMACS, or to GROMACS in the case of installed headers). Doing that well means that you can change the implementation without touching the code that uses the interface (e.g. like the way we can use FFTW, MKL or fftpack and the PME code doesn't need to know which). Historically, we've had very ad-hoc internal interfaces (e.g. "Everything to do with wallcycle counting"), which is why details of the representation of a summed counter (a std::array) could leak into the header in this early phase of making a decent module. The purpose of avoiding including C++11 headers in our headers is so that the interfaces they declare are accessible from non-C++11 code (e.g. CUDA). But that's just a special case of not exposing implementation details in the interface.

In the present case we want one interface for code that needs to start and stop timers, and for the time being (at least) we need another for the end-of-run reporting code. That's because those are quite distinct roles, and gmx_wallcycle_t should not be handling all of it (and won't, when my series of changes are done).

> I have the feeling that this general issue could grow big and annoying in the future, so it would be worth to avoid as much as possible. A slightly stronger wording of your #1 proposed addition to the guidelines and careful review should help. So does if everybody is well-aware of this potential pitfall and not just the two of us.

Yes.

> The solution specific to this case has occurred to me, but I just wanted to try a simple thing and did not expect to have to refactor code to achieve that, so I used the 5.1 code-base instead.

Yes, but that will no longer be needed when I push a patch I've prepared.

#49 - 11/18/2015 09:54 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1745.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: I70517ac104f002b7b4e61a3aaa26acff5193361c
Gerrit URL: https://gerrit.gromacs.org/5353

#50 - 04/04/2016 02:46 AM - Gerrit Code Review Bot

Gerrit received a related patchset '10' for Issue #1745.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: Ieef3b47de5c1a00a203baa1b34ebf70535cf5ff0
Gerrit URL: https://gerrit.gromacs.org/5753

#51 - 06/01/2016 03:01 PM - Mark Abraham

https://gerrit.gromacs.org/#/c/5364/ is relevant here, but I need to incorporate existing feedback (and probably receive more)