GROMACS - Task #1758

Verlet scheme reorganization / modularization

06/28/2015 06:47 PM - Mark Abraham

Status:	New
Priority:	Normal
Assignee:	Mark Abraham
Category:	core library
Target version:	future
Difficulty:	uncategorized

Description

There's lots of things to consider here, so I'll edit this later when I have more time.

Meanwhile, the handling of nbnxn_const.h provoked some discussion at https://gerrit.gromacs.org/#/c/4710/, and Mark wants to remember that:

I had a quick look at the code. The only use of NBNXN_INTERACTION_MASK_* outside nbnxn_search.c is in the outer loop of the SIMD kernels, where we use it as one of two conditions on one the inner loops. But if we've done the sort_cj_excl then isn't it simpler (and maybe slightly faster) to store cjind_mask_end along with cjind_start and cjind_end? If so, then all those constants are local to CPU search code and have no need to be in a module-wide header.

Related issues:

Related to GROMACS - Feature #1666: new approach for Verlet-scheme kernel gen...	**New**
Related to GROMACS - Task #2422: write C kernel for tables in Verlet scheme	**New**
Related to GROMACS - Feature #2931: Tables in Verlet kernels	**New**

History

#1 - 06/30/2015 08:43 AM - Berk Hess

I tried this and performance goes down by about 1%. If we want to make that trade off, I can push up the change.

#2 - 07/01/2015 10:00 AM - Erik Lindahl

I'll expand this discussion a bit since I started to look into the kernels for some summer work now that we’re (almost) post 5.1.

While I think the CPU kernels should be relatively straightforward (and with the extended SIMD we can make them truly architecture-independent), we’ll need to find a good strategy for CUDA & OpenCL. We have a separate discussion about C++11 in general, so let's focus this one on kernels.

In particular, I think it’s imperative to avoid CPU and GPU kernels diverging (otherwise nobody but the experts will contribute GPU code), so we’ll need to decide what to do for GPUs.

1) For CUDA we’ll need at least CUDA 6.5 to allow C++ and templates, if we want to use templates on the CPU.

2) Does OpenCL support anything template-related? There appears to be something in the AMD APP SDK, but not in the standard.

While it would have been nice to use templates and instantiate functions directly in the compiler, this might be a dealbreaker. At least for OpenCL we likely need to use our custom Python preprocessor, and then the question is whether we should do it for CUDA and CPU kernels too?

#3 - 07/02/2015 12:12 AM - Szilárd Páll

As I wrote off-redmine, if we want the gcd(CPU/SIMD, OpenCL, CUDA) we'll definitely have to stick to C99 as OpenCL is C99+extensions and this will not change in the near future. Additionally, requiring CUDA 6.5 is somewhat harsh (CUDA 5.5 and especially 6.0 are relatively recent).

Simple templating is something we could still consider for code where this just makes sense and is more elegant, e.g. to express the choice of interaction types in the inner loop of non-bonded kernels. Of course that may mean having to treat some arch/platforms differently than others.

One additional aspect we should definitely consider is kernel JIT-ing and the ability to easily specialize code at runtime. In its simplest form that will mean replacing run-time constants with compile-time ones, but by the end it could get more complex.

#4 - 07/02/2015 03:04 AM - Roland Schulz

Currently the CUDA, OpenCL and C implementation are independent. Which parts could realistically be converged or would benefit from sharing of methods?

As you mentioned AMD's OpenCL supports templates (opencl static c++ kernel language). It seems only AMD is a useful target for OpenCL at the moment. Thus it might be acceptable to only support AMD as OpenCL target in the short term. Long term it isn't a limitation because OpenCL 2.1 supports templates.

If we decide not to use templates, I recommend we use some other meta-programming language, not just a generator which is based on text replacement. A generator has the huge disadvantage that it doesn't give any syntax errors but instead any error is reported by the compiler, and one has to backtrack that to the pregenerated code.

Possible non-template meta languages would be :
- preprocessor libraries (such as Boost). We would still use the current preprocessor approach but could potential improve some issue with it
- source-to-source compiler (such as rose, ispc, cython)

#5 - 07/07/2015 11:17 PM - Roland Schulz

- Related to Feature #1666: new approach for Verlet-scheme kernel generation added

#6 - 07/07/2015 11:29 PM - Roland Schulz

Erik Lindahl wrote:

> 1) For CUDA we'll need at least CUDA 6.5 to allow C++ and templates, if we want to use templates on the CPU.

For C++11 templates we need 6.5. Is there any reason that we would need CUDA 6.5 for C++03 templates?
We discussed we might want to use C++11 for kernel templates because of extern templates and because of constexpr.
As far as I see extern templates only give a slight improvements in code organization, while keeping the fast parallel build. We wouldn't need wrapper functions (i.e. we could get rid of the current ones and directly call the templated function) to be able to instantiate the template functions in different compilation units. But I think it is reasonable to keep the wrapper function or accept slower parallel build without c++11.
I'm not sure how much constexpr would help with the kernel.

#7 - 07/08/2015 07:05 PM - Szilárd Páll

Roland Schulz wrote:

> Currently the CUDA, OpenCL and C implementation are independent. Which parts could realistically be converged or would benefit from sharing of methods?

Sharing code is not very likely especially not between C/SIMD and CUDA/OpenCL but perhaps it would be possible between the latter two.

However, I thought we were still considering to mix the use of generator and templates. In the mixed caseit won't matter that some sets of the kernels will be fully generated with all hundreds of versions, others using templates will have less pre-generated versions.

> As you mentioned AMD's OpenCL supports templates (opencl static c++ kernel language). It seems only AMD is a useful target for OpenCL at the moment. Thus it might be acceptable to only support AMD as OpenCL target in the short term. Long term it isn't a limitation because OpenCL 2.1 supports templates.

Intel GPUs are also on the table, especially on Broadwell/Skylake with OpenCL 2.0 desktop setups could work well, though ATM there are no concrete plans for it.

> If we decide not to use templates, I recommend we use some other meta-programming language, not just a generator which is based on text replacement. A generator has the huge disadvantage that it doesn't give any syntax errors but instead any error is reported by the compiler, and one has to backtrack that to the pregenerated code.

Good point. I will likely be a big enough hurdle that direct optimization will become tedious.

#8 - 07/08/2015 07:32 PM - Roland Schulz

Szilárd Páll wrote:

> Intel GPUs are also on the table, especially on Broadwell/Skylake with OpenCL 2.0 desktop setups could work well, though ATM there are no concrete plans for it.

For 2.0 it took Intel 14.5 month (Jun22 13-Sep9 14) from provisional specs to release. Based on that we can expect their 2.1 (prov. specs Mar3) in 10 month (with probably a beta version before). Thus it has a good chance to be there in time for our next release.

#9 - 07/19/2015 07:13 PM - Roland Schulz

Can we move forward here? Any remaining problems with going with templates? Or does it make sense to go ahead and do the template conversion to be able to test that it has no performance effect?

#10 - 07/20/2015 08:25 AM - Mark Abraham

Templating parameters that are per-kernel constants seems certain to be good approach, whether we end up working through wrapper function calls or not. I haven't considered what code would make for the cheapest investment for experimenting, but perhaps a single 2xnn function inside its current wrapper would work?

#11 - 07/20/2015 06:11 PM - Szilárd Páll

Szilárd Páll wrote:

> > If we decide not to use templates, I recommend we use some other meta-programming language, not just a generator which is based on text replacement. A generator has the huge disadvantage that it doesn't give any syntax errors but instead any error is reported by the compiler, and one has to backtrack that to the pregenerated code.
>
> Good point. I will likely be a big enough hurdle that direct optimization will become tedious.

Actually, having had a chat about it, this may not be such a big deal. When working on existing kernels, it's always an option to tweak the already generated source files and new kernels can also be developed directly at first and moved to the generator input syntax later. It should also be quite straightforward to create a build target that triggers the generator.

> For 2.0 it took Intel 14.5 month (Jun22 13-Sep9 14) from provisional specs to release. Based on that we can expect their 2.1 (prov. specs Mar3) in 10 month (with probably a beta version before). Thus it has a good chance to be there in time for our next release.

Why is 2.1 relevant? First we'd need 2.0 kernels optimized for Intel *and* for IGP use (e.g. "zero-copy" buffer sharing). Would be good to find somebody interested (and competent). Any ideas?

> Any remaining problems with going with templates?

If you mean going with templates only that may not be feasible, but that's why we should try. E.g. there are known issues with the AMD OpenCL compiler's inlining capabilities which may mean that the pruning kernels will be probably hard to express with templates.

> Or does it make sense to go ahead and do the template conversion to be able to test that it has no performance effect?

I think so.

It would be nice if the experimentation happened in a place where code (a branch) can be shared easily. I could convert the innermost preprocessor generator pass into templated conditionals in the CUDA kernels quite easily, but have no clue how to instantiate those functions and get the function pointer that can be placed in the lookup table.

#12 - 07/20/2015 06:33 PM - Roland Schulz

Szilárd Páll wrote:

> Why is 2.1 relevant? First we'd need 2.0 kernels optimized for Intel *and* for IGP use (e.g. "zero-copy" buffer sharing). Would be good to find somebody interested (and competent). Any ideas?

Because 2.1 has support for C++ templates. Of course it is only relevant if we have Intel kernels (AMD already has templates) and want to use templates also for OpenCL.
I would start with the CPU kernels.

#13 - 03/06/2018 11:28 PM - Mark Abraham

- Related to Task #2422: write C kernel for tables in Verlet scheme added

#14 - 04/29/2019 11:44 AM - Mark Abraham

- Related to Feature #2931: Tables in Verlet kernels added