GROMACS - Task #2874

Refactor Gromacs (cluster) neighborlist into separate module

02/28/2019 09:50 AM - Erik Lindahl

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

Description

The low-level neighborlist might be a good starting point to create fully orthogonal modules from the present kernel/neighborlist/neighborsearching/decomposition/gpu code after Berk's updates.

In terms of a module/class interface, the user should be able to:

1. Create a new list object with internal structures optimized to fit the hardware it will run in (e.g. different types of lists for GPUs, various SIMD). The challenge here is how to do this in an opaque way that does not make the module dependent on the GPU code, or have a module interface that depends on how the code is compiled or what hardware it is used for.
2. Add neighbor entries to the object.
3. Access the data entries (everything should go through class methods)
4. Methods for emptying lists, and possibly management task related to memory optimization, placement, rebalancing, whatever.

Open questions:

- How do we hide the complexity of different N/M sizes in the module interface?
- How do we best enable a single module/class to support different types of kernel (use superclusters, but don't expose it in the interface?)
- Should we include support for traditional neighborlists (useful e.g. in simple analysis tools) as a special 1x1 case?

Without having thought extremely deep about it, I could imagine that we create small types to represent the width (N or M atoms) as well as optional superclusters, and then have these as settings for a factory method creating objects. Then we might be able to use a bit of C++ magic for catching incorrect usage of the class interface already at compile time.

History

#1 - 02/28/2019 10:34 AM - Berk Hess

For analysis tools there is the simple list and search code Teemu wrote for the analysis module. We should keep this separate from the non-bonded module. For analysis we want all pairs of a list of atoms in a simple format. For nonbondeds we want all pairs in the system, potentially decomposed over domains, in an efficient format for the hardware.

Currently the pairlists are set up according to this enum:
enum class PairlistType : int {
Simple4x2,
Simple4x4,
Simple4x8,
Hierarchical8x8,
Count
};

That is all information that specifies the type of list. A kernel type is also passed to the search code to provide SIMD acceleration of the search. but that could actually be derived from the list type. The hierarchical list is currently of different C++ type that the others. I don't see any advantage into unifying these two types using a standard interface. All code consuming these is completely separated into different modules.

#2 - 02/28/2019 10:45 AM - Erik Lindahl

One of the big advantages of C++ is that we can have the class interfaces enforce correct usage, so I think there is a lot to say for having a common interface so we only set the type of structure **once**, and after that we completely avoid all checks for what type of list we use (or generate) in particular places - because we have a history of our conditionals causing bugs when we miss some conditions :-)

It also seems important to create an interface that does not depend on any particular setup (say, when we suddenly realize 2x2, 2x4 or 4x16 could be useful for some new hardware), and in particular not relying on checking an enum value to access data in the right way - we want the compiler to do that under-the-hood.

We want to allow freedom for different hierarchical geometries in the future, but do we also need to be prepared for anything multi-hierarchical?

Finally, when we actually **use** individual elements, we need the lowest level in the hierarchy, so I think it should be quite possible to keep the hierarchical aspect mostly internal, and only use that to accelerate the neighborsearching?

#3 - 02/28/2019 11:27 AM - Berk Hess

The hierarchical aspect is there to accelerate the non-bonded kernel, not to speed up the neighbor searching.

The CPU and GPU lists are type separated, so that is fully safe.
For the CPU lists the j-size can currently be 2, 4 or 8 which is set in the enum. This enum is derived based on the kernel type. The dispatcher uses the kernel type to choose the kernel. I don't see how we can improve upon this. The chance of bugs is also about zero. calling the wrong kernels results in a segv or completely incorrect results.

#4 - 02/28/2019 11:30 AM - Erik Lindahl

Berk Hess wrote:

> calling the wrong kernels results in a segv or completely incorrect results.

I think we might have slightly different definitions of "things that can be improved" ;-)

#5 - 02/28/2019 11:36 AM - Erik Lindahl

Simple scenario:

Imagine an advanced user-developer, who wants to modify something about the Gromacs neighbor lists - say introduce an extra layer, filter/save/restore a list, or whatever, or maybe use the list in another module they are writing that does something. Remember that we will also have to support this as part of the NBLib project.

Having a clean unified interface and representation for how we handle the "neighborship" concept is a VERY strong win IMHO, while asking an external user calling GROMACS as a library to deal with double interfaces data handling depending on whether the underlying library will be running on CPUs or GPUs is similarly user-unfriendly?