GROMACS - Task #1017

C++ Vector/Matrix classes

10/04/2012 06:00 PM - Roland Schulz

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

Description

We require RAII classes for vectors and matrices for C++ to replace rvec/matrix. Ideally they should also support basic operations with a nice C++ interface (vector/matrix add, vector-vector/matrix-vector/matrix-matrix multiplication, element-wise sum ...)

I think Eigen (http://eigen.tuxfamily.org/index.php?title=Main_Page) is awesome. And as far as I can see it fulfills all dependency requirements: LGPL2 compatible (recently changed to MPL2), support for more than our supported compilers, not too big (Core is 32k lines - could be reduced further if necessary).

Related issues:

Related to GROMACS - Feature #1500: Post-5.0 feature clean-up plan	**New**	**01/20/2014**
Related to GROMACS - Feature #1612: generating SIMD code	**Closed**	

Associated revisions

Revision c9cbb4cd - 01/05/2015 11:30 PM - Teemu Murtola

Minimal C++ replacement for rvec

Add a C++ class that acts as rvec, but also supports placement in STL containers (i.e., is default-constructible, copy-constructible, and assignable) and can be returned from functions. There is some ugly reinterpret_casting required for rvec arrays to make this work, which requires that the compiler generates compatible layouts for the class and rvec. The tests ensure that this is the case, but it could be nice to check some more exotic compilers as well, unless someone knows the C/C++ standards well enough to say that this layout is always ensured.

Passing RVec as 'const rvec &' unfortunately also needs an explicit function call because of issues with cppcheck and MSVC.

Minimal solution to #1017.

Change-Id: I79be58bf7be2a78f787f470044b3b5abf234ec0c

History

#1 - 10/04/2012 06:29 PM - Roland Schulz

Other alternatives are:

- Tiny Vector Matrix library using Expression Templates (tvmet) LPGL 2.1, 24k lines
- Armadillo: LGPL3 - doesn't effect the Gromacs code but might still be a problem
- Blitz++: LGPL3, see tvmet page for list of problems, 36k lines
- Boost uBLAS: large dependencies
- IT++
- Newmat.
- MTL4: depends on Boost MTL
- ViennaCL: GPU acceleration, can interface with Eigen & MTL4
- gpumatrix: GPL v3, GPU acceleration, compatible to Eigen and uses tvmet
- VC: LGPLv3

Most are not relisitc alternatives:

- Armadillo, IT++, Newmat, MTL4, and ViennaCL only support dynamic sized vectors, which is not a good choice for rvec/matrix.
- uBLAS, MTL4 too much dependencies (boost mtl)
- Blitz++: dormant and no advantage
- gpumatrix, VC: license

tvmet would be an alternative. But it is not actively maintained (last release from 2007), doesn't have SSE/Altivec/Neon auto-vectorization, doesn't use Cmake as build system, and doesn't have support for variable sized vectors. The good thing is it seems the Eigen/tvmet interface seems compatible for fixed size vectors. Thus one option would be to use Eigen but fall-back to tvmet if it isn't available. But given that tvmet is only a bit smaller (and one could probably reduce Eigen further) and doesn't seem to have other advantages, I'm not sure that extra fall-back is worth the extra cmake complexity. Also it would mean we couldn't use dynamically sized vectors. That's ok for rvec/matrix but it would be nice to have dynamically sizes vectors/matrices for the analysis tools.

#2 - 02/19/2013 05:43 PM - Teemu Murtola

Pasting Mark's thoughts from gmx-dev list on basic vector operations, where I think the discussion belongs into this issue:

- can’t keep using rvec
  - rvec can’t be put into STL containers (need copy constructor, etc.)
  - rvec guarantees we can’t use aligned loads anywhere (important for leveraging SIMD possibilities)
  - makes using RAII harder
  - probably makes writing const-correct code harder
- we want to be able to use STL containers when that makes code writing, review and maintenance easier
- we need to be able to get flat C arrays of atom coordinates with no overhead for compute kernels
- straightforward suggestion: switch to using an RVec class with a 4-tuple of reals and use them for x, y, z and q
  - in many places q won’t be used
  - 16-byte alignment for free (opportunities for compiler auto-SIMD)
  - perhaps 4/3 increase in cache traffic where q is not being used
  - std::vector< std::vector<real> > doesn’t map to a flat C array - need to write/find a “tuple” class that lets the compiler know what is going on, so that std::vector< tuple<real,4> > ends up as a flat C array of xyzqxyzqxyzq...
- separate vectors for x, y, z and q could be useful because that would help avoid the swizzling (group kernels) and coordinate copying (Verlet kernels) that currently occurs
  - downside is that x, y, and z are normally used together, so a naive approach pretty much guarantees we need 3 cache lines for each point... if we don’t re-use that data a few times, that could kill us
- internally use some kind of “packed rvec” laid out xxxxyyyyzzzz(qqqq) and have some kind of intelligent object that we can use just like we use rvec now, e.g. coords[3][YY] magically returns the 8th element of xxxxyyyyzzzz
- the needs of mdrun and analysis tools are different, and we can perhaps explore different implementations for each - but a common interface would be highly desirable
- ideally we would not commit in 2013 to an internal representation that we might regret in the future... how can we plan to be flexible?
  - run-time polymorphism, e.g. have the coordinate representation classes share a common base with virtual functions - probably too slow, and we don’t want to store the virtual function tables
  - code versioning - ugh
  - bury our heads in the sand - we might get lucky and never want to change our coordinate representation
  - compile-time polymorphism, e.g. mdrun<RVec> vs mdrun<PackedRVec,4>
    - might also allow a more elegant implementation of double- vs mixed-precision
    - code bloat if we want binaries that can run on any x86 if different CPUs will want different packings
    - compile-time bloat if compiling more than one such representation, as a lot of routines would now be parameterized

#3 - 02/19/2013 05:47 PM - Teemu Murtola

I don't have a strong view on what would be the best internal memory layout, and how would the best API for that be, but the requirements for that would be good to work our before looking into external libraries, since it can place some constraints.

Also, I agree with the comment that was voiced somewhere (probably it was Erik) that we don't really require much support for linear algebra etc. with big matrices, but mainly basic coordinate manipulation and doing some operations over the entire vector.

Will try to comment more on the points by Mark later, in particular from the implementation standpoint.

#4 - 02/19/2013 08:33 PM - Teemu Murtola

I can come up with the following general approaches for replacing rvec (whether we use an external library or not). Have not had the time to test all the corner cases, but those should hopefully not affect the big picture. Duplicates some parts from Mark's comments.

1. Minimal solution: Replace the current rvec with a C++ class that has the same memory layout, but additionally provides default construction, copy construction and assignment to be able to place it into STL containers. This is Mark's "straightforward suggestion".
   - Allows easy use of STL containers.
   - Only works with an xyz(q)xyz(q)... memory layout.
   - Possible (although ugly) to cast arrays/STL container contents into a plain old C rvec array if the memory layout is the same.
2. Custom container solution: if we want a different memory layout (or want to keep the flexibility of changing the layout in the future), then the above is not an option. Instead, we need to implement a custom "packed" coordinate class and/or a custom container that provides access to the data and manages the indexing.
   - We can't use STL containers as the public interface, unless we are fine with always indexing it first with a "block" index, and then iterating separately within that block.
   - Can implement accessors for a custom container like Mark suggests, such that it looks exactly like an std::vector<rvec> (except for some issues with the iterators, similar to #856).
   - Such accessors should be reasonably efficient as long as the stride is known at compile time. For the parts that are not the most performance-critical, a stride that is only known at run time may be an option, but would probably need some benchmarking.
   - As long as all code that is not dependent on the exact layout uses these accessors, it should be easy to change the layout.
   - Need to benchmark that the compilers we plan to support can effectively inline the code in the accessors and optimize away unnecessary complexity. Unfortunately, mdrun requires both the most complexity and the best performance here; tools will probably be fine with any approach.

Some additional points, trying to focus on the last "planning to be flexible" part:

- Different implementations are possible for tools and mdrun, but that will most likely lead to some code duplication unless code that operates on vectors is templatized to work with both implementations. Not impossible to do, but not sure whether this classifies as "simple" C++.
- Making the whole code a template with the vector type as a template parameter is probably not a good idea unless we plan to have a hard dependency on C++11 extern template (also touched on in #951).
- If it is an option performance-wise to have the stride known only at run-time in most of the code, we can also make it such that only the most performance-critical parts are template-parameterized on the vector type, and everywhere else we use a more flexible class to access the coordinates that can work with multiple strides. As long as different parts of the code do not want different strides, this should be relatively easy to do. This probably doesn't work as well with double- vs. mixed-precision (#951), though, as double- vs. mixed-precision requires different floating-point types throughout the code.
- Polymorphism is probably out of the question here. It may be possible if implemented at the container level; this is one option to implement the above alternative, but not the only one. But it is not possible with the first approach (since the virtual table messes up the memory layout, and increases the size of the type significantly). And it probably is not an option performance-wise at a single coordinate level (e.g., in iterators).

#5 - 02/20/2013 08:00 PM - Teemu Murtola

Mark Abraham wrote:

> - can’t keep using rvec
>   - makes using RAII harder

The minimal solution to this is to allow using boost::scoped_array, but that doesn't allow everything that a proper STL container would.

> - probably makes writing const-correct code harder

This is true if we want to keep some parts of the code compiled in C: const rvec a[] as a function parameter works as one would expect in C++, but in C, it is not possible to pass a rvec b[] into such a function (one gets an "incompatible pointer type" warning). And the opposite is true for rvec a[]: it is not possible to pass a const rvec b[] into such a parameter, not in C (gives the same warning) nor in C++ (gives an error). Thus, it is possible to use const rvec a[] parameters in functions called from C only if everything is compiled as C++. But if all parameters are rvec a[] without the const, it is painful to call such functions from C++ code that is const-correct and uses const rvec arrays (possible using const_cast, but ugly).

#6 - 02/20/2013 08:50 PM - Alexey Shvetsov

Seems like most portable library from list above is eigen for example list of supported arches on gentoo linux for eigen on linux
http://packages.gentoo.org/package/dev-cpp/eigen?arches=linux
Also it should work on mips64 platform.

#7 - 02/20/2013 09:46 PM - Roland Schulz

From the calls the requirements we collected:
- c array interface
- LGPL compatible
- portable

Nice to have:
- c++ iterator interface
- thread parallel support
- element wise operations on (dynamic-length) arrays of (static-length) rvecs
- element wise operations which don't use temporaries for A+B+C
- SIMD optimization
- Reasonable quick compile time

I didn't take notes for the whole time. So please add anything I forgot.

#8 - 02/21/2013 05:42 AM - Teemu Murtola

Teemu Murtola wrote:

> - probably makes writing const-correct code harder
>
> This is true if we want to keep some parts of the code compiled in C: const rvec a[] as a function parameter works as one would expect in C++, but in C, it is not possible to pass a rvec b[] into such a function (one gets an "incompatible pointer type" warning). And the opposite is true for rvec a[]: it is not possible to pass a const rvec b[] into such a parameter, not in C (gives the same warning) nor in C++ (gives an error). Thus, it is possible to use const rvec a[] parameters in functions called from C only if everything is compiled as C++. But if all parameters are rvec a[] without the const, it is painful to call such functions from C++ code that is const-correct and uses const rvec arrays (possible using const_cast, but ugly).

Correcting myself: it is possible to get this working in C++ and hide the ugliness from calling code:

```
void f(rvec a[]);
#ifdef __cplusplus
void f(const rvec a[])
{
    f(const_cast<rvec *>(a));
}
#endif
```

This allows calling f() with both const rvec * and rvec * parameters from C++ code and does not break existing C code. Still a bit ugly, but at least the ugliness is confined to one place... Of course, this should only be used if f() really does not modify the array passed in.

For the requirements, was there any requirements on the support for different memory layouts? I think that possible support for memory layout like xxxxyyyyzzzz(qqqq)xxxx.... (either out-of-the-box, or possibility to implement it easily on top of the external library) is a relatively big requirement if we want to have that, in particular if we want to abstract that away so that all code does not depend on the explicit layout. Or are we simply looking for a library that provides raw arrays of numbers, and then implement any support for 3/4-d vectors on top of that ourselves? But even here, a requirement to be able to do that without writing a lot of template magic could be a relevant one.

#9 - 02/21/2013 12:49 PM - Mark Abraham

Roland Schulz wrote:

> From the calls the requirements we collected:
> - c array interface
> - LGPL compatible
> - portable

Also required, particularly for use in mdrun:

- able to support aligned allocation for the whole container and to extract aligned handles to elements and groups of elements
- play nicely when working with multiple threads (Mark's not yet sure what this means in practice)

> Nice to have:
> - c++ iterator interface
> - thread parallel support
> - element wise operations on (dynamic-length) arrays of (static-length) rvecs
> - element wise operations which don't use temporaries for A+B+C

... and which do use temporaries for e.g. M (A + B + C) where M is a matrix, [ABC] are vectors

> - SIMD optimization
> - Reasonable quick compile time
>
> I didn't take notes for the whole time. So please add anything I forgot.

We should plan now for catering to the possibility of mdrun and tools having different requirements. At a code level, this will mean having two typedefs that for now might point to the same underlying type.

We should try to find out what people like the authors of Vc and Eigen have as their biggest challenges and see what that means for us.

Mark thinks a std::vector< std::array> > plus aligned allocator might be all we need for mdrun. std::array is from Boost or TR1.

Mark had a brief look at Eigen last night and also thinks that's likely to be a good solution, particularly for tools.

Need to avoid templated constructs that lead to massive object code bloat.

Szilard point out that CUDA code does have some requirement that needs arrays of structs rather than structs of arrays, but I'm not sure what the consequence of that is.

A fruitful first step is probably implementing (some of) g_rms so that we have some relevant computation we can do while playing with how to make things work.

#10 - 02/21/2013 08:50 PM - Roland Schulz

Some more details about alignment:
http://eigen.tuxfamily.org/dox/TopicStructHavingEigenMembers.html
http://eigen.tuxfamily.org/dox/TopicStlContainers.html

Given that we need to have also aligned static variables, it isn't sufficient to only have a an aligned allocator.

VC has a paper: http://code.compeng.uni-frankfurt.de/attachments/110/_A_C___Library_for_Explicit_Vectorization__private_archive_A4_.pdf
But besides that VC's licence isn't compatible, it also isn't a vector library. It only has types for static vectors of SIMD size and no element wise operations of dynamic size arrays.

Eigen:
- slides: http://downloads.tuxfamily.org/eigen/eigen_plafrim_may_2011.pdf
- description of internals (assumes detailed C++ template understanding): http://eigen.tuxfamily.org/dox-devel/TopicInsideEigenExample.html
- comparison to it++: http://bickson.blogspot.com/2011/10/it-vs-eigen-part-2-performance-results.html

#11 - 02/22/2013 09:19 AM - Roland Schulz

A bit off topic (we don't have a general C++ feature redmine): If we ever want to compile for a platform which doesn't have a standard compliant C++ compiler (e.g. no template support), a possible solution is C++ to C compilation which is supported by EDG/Comeau and maybe by LLVM (it currently in a partial working state). The C code could than be compiled with the target C compiler.

#12 - 06/27/2013 02:34 AM - Roland Schulz

It would be nice if we could make some progress on this. A solution to at least store rvec's in container is needed, so that code Teemu is writing doesn't constantly need to work around it. (e.g. https://gerrit.gromacs.org/#/c/2381/)

#13 - 06/27/2013 05:56 AM - Mark Abraham

Indeed. I have about a fortnight of 4.6-era stuff left to do (mostly non-code), but addressing this will be one of the next top priorities.

#14 - 06/27/2013 06:20 AM - Roland Schulz

Great! Let me know if you have any questions (given that I spend quite a bit of time looking at this issue).

#15 - 10/17/2013 07:05 AM - Roland Schulz

Another promising candidate is blaze: https://code.google.com/p/blaze-lib.

#16 - 05/13/2014 10:05 AM - Mark Abraham

- Target version changed from 5.0 to 5.x

#17 - 05/26/2014 11:44 PM - Mark Abraham

- Related to Feature #1500: Post-5.0 feature clean-up plan added

#18 - 07/15/2014 05:06 PM - Teemu Murtola

- Category set to core library

#19 - 12/14/2014 03:28 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1017.
Uploader: Teemu Murtola (teemu.murtola@gmail.com)
Change-Id: I79be58bf7be2a78f787f470044b3b5abf234ec0c
Gerrit URL: https://gerrit.gromacs.org/4307

#20 - 07/07/2015 12:21 AM - Roland Schulz

- Related to Feature #1612: generating SIMD code added

#21 - 07/11/2016 08:23 PM - Mark Abraham

- Target version deleted (5.x)