Multi-SIMD binaries

I wanted to open the discussion of enabling multi-SIMD binaries (e.g. SSE2, SSE4, AVX256). This is of course something we supported for 4.5 but dropped for 4.6. I took a spin through the 4.6 code, and the nonbonded loops would be pretty easy to do, but the NXN (and PME, I believe) code would be more of a pain, as the #includes are embedded pretty deeply.

It occurs to me that this could be much easier to fix in 5.0 (and in general it might be good to remove more of the configuration #includes). One could imagine either passing around a configuration object or subclassing depending on the configuration/optimizations present.

I wanted to get your opinion on this. Many of the Gromacs deployments I work on (Folding@home is one example, you can think of the others) are very large-scale with non-uniform hardware. Given the boost we can get from AVX256 (or AVX128/FMA), it would be nice to combine performance with backwards compatibility. And for F@H at least, multi-binary would be a huge PITA and realistically limit us to ~2 flavors.

Thanks!

Related issues:
- Related to Gromacs - Feature #951: Multiple versions of Gromacs (e.g., single... New 05/31/2012
- Related to Gromacs - Feature #1123: binary incompatibility New 01/17/2013

History

#1 - 02/27/2013 07:36 PM - Teemu Murtola
Added a few issues that contain related discussion.

#2 - 02/27/2013 07:47 PM - Erik Lindahl
Unfortunately the future is worse, not better. Soon both Intel and AMD will have FMA instructions, but different flavors - AMD benefits significantly from FMA4 while Intel only does FMA3. This affects every single function in all of Gromacs, not just a handful of kernels with special instructions, and right now we only spend ~50% of the execution time in kernels.

In general, this difference is about as large as the difference from enabling AVX256 over e.g. SSE4.1 and also affecting 50% of the code. x86 hardware is unfortunately diverging faster than ever, and there is not going to be any way to get full performance on all architectures apart from using separate binaries. Even the flavor of FFTW optimization is starting to be important for our performance.

#3 - 02/27/2013 07:51 PM - Peter Kasson
I think we need to look really hard at this. If we require separate binaries, I can say with certainty that most GROMACS flops will be spent in suboptimal binaries. I'm not sure that's a scenario that we want. I think there are some software engineering solutions to this.

Thanks, Teemu, for the links. Do you think templates versus subclassing versus just having a configuration object and conditionals (obviously not in the inner loops) instead of #ifdefs?

#4 - 02/27/2013 08:01 PM - Mark Abraham
One issue here is how much we are prepared to cater to SIMD in choosing our data layout. Currently, the group kernels read XYZ and swizzle to whatever makes sense in the kernel, while the Verlet kernels do so back at neighbour-search time (IIRC). And both back-swizzle at some point, too. We can probably do slightly better by eliminating those, but the consequences of that decision have the potential to go everywhere. C++ does offer the possibility of letting a smart-alec iterator handle the SIMD details if we just want to (say) get the value of some Y coordinate some time. But if multiple kernel flavours might want different SIMD layouts, then the object code size is going to get big, whether there's much templating involved or not.

One option for F@H is to bundle 4.6-style mdrun binaries for multiple flavours and (e.g.) steal the GROMACS detection code to decide which one to run. That could be a much smaller PITA than the alternatives for GROMACS 5.0. It does suck if you'd be requiring all participating users to download multiple binaries/libraries, though.

I don't suppose there's a partly free lunch available with 4.6 with dynamic linking of libgmx and/or libmd?

---

#5 - 02/27/2013 08:03 PM - Mark Abraham

... or custom kernel DLLs in a project fork?

#6 - 02/27/2013 08:10 PM - Teemu Murtola

Peter Kasson wrote:

Thanks, Teemu, for the links. Do you think templates versus subclassing versus just having a configuration object and conditionals (obviously not in the inner loops) instead of #ifdefs?

#include or conditionals can't solve the issue if compiler options (like -mavx) have a significant effect in large parts of the code. If that is the case, the only way is to compile those parts more than once, and resolve the duplicate symbols in some way. Templates or namespaces are the only reasonable solution that I can think of to manage large parts of the code this way within the same library. And subclassing/interfaces are the only way to isolate those parts from the parts that do not require this kind of multi-compilation (unless dynamic loading one from a collection of libraries is an option, but that's also kind of an interface).

For data layout, if the "generic" part of the code is not super-sensitive to performance, it should be possible to simply have a slightly slower way of accessing the coordinate arrays there (instead of compiling it also in multiple flavors), wrapping the layout behind a custom container (some discussion in #1017).

#7 - 02/27/2013 08:12 PM - Szilárd Páll

Mark Abraham wrote:

One issue here is how much we are prepared to cater to SIMD in choosing our data layout. Currently, the group kernels read XYZ and swizzle to whatever makes sense in the kernel, while the Verlet kernels do so back at neighbour-search time (IIRC). And both back-swizzle at some point, too. We can probably do slightly better by eliminating those, but the consequences of that decision have the potential to go everywhere. C++ does offer the possibility of letting a smart-alec iterator handle the SIMD details if we just want to (say) get the value of some Y coordinate some time. But if multiple kernel flavors might want different SIMD layouts, then the object code size is going to get big, whether there's much templating involved or not.
We've had some discussions recently and many core devs seemed to agree that, although fancy C++ can look cool and can potentially be an efficient way to write code, at least for 5.0, we would prefer nothing fancy especially in/close to kernels and in general in performance sensitive code. There is just too much complication involved and without a dozen very experienced C++ programmers replacing the current very experienced C developers we run the risk of:

- struggling to understand code;
- being unable to efficiently write at least low-level performance-oriented stuff that defines GROMACS;
- not being able to avoid pitfalls and bad coding practices.

One option for F@H is to bundle 4.6-style mdrun binaries for multiple flavours and (e.g.) steal the GROMACS detection code to decide which one to run. That could be a much smaller PITA than the alternatives for GROMACS 5.0. It does suck if you'd be requiring all participating users to download multiple binaries/libraries, though.

That is a good suggestion, the only concern is binary size which might prevent providing all combinations.

I don't suppose there's a partly free lunch available with 4.6 with dynamic linking of libgmx and/or libmd?

I doubt it. For 5.0 we could consider allowing some way to compile libgromacs for multiple platforms and load the right one at runtime from mdrun.

---

#8 - 02/27/2013 08:19 PM - Szilárd Páll

Teemu Murtola wrote:

Peter Kasson wrote:

Thanks, Teemu, for the links. Do you think templates versus subclassing versus just having a configuration object and conditionals (obviously not in the inner loops) instead of #ifdefs?

#endifs or conditionals can't solve the issue if compiler options (like -mavx) have a significant effect in large parts of the code. If that is the case, the only way is to compile those parts more than once, and resolve the duplicate symbols in some way. Templates or namespaces are the only reasonable solution that I can think of to manage large parts of the code this way within the same library. And subclassing/interfaces are the only way to isolate those parts from the parts that do not require this kind of multi-compilation (unless dynamic loading one from a collection of libraries is an option, but that's also kind of an interface).

I think dynamically loaded platform-specific libgromacs is the best option! I don't think we should approach this problem by trying hard to have the cack and eat it too: it's enough effort to maintain the high performance on different SIMD hardware, creating some complex subclassing only to allow run-time selection is IMO too much effort - unless there is some elegant and simple solution that I don't know of.
For data layout, if the "generic" part of the code is not super-sensitive to performance, it should be possible to simply have a slightly slower way of accessing the coordinate arrays there (instead of compiling it also in multiple flavors), wrapping the layout behind a custom container (some discussion in #1017).

The problem is that what we today think is not super-sensitive will tomorrow become the bottleneck. Examples: data shuffling in the non-bonded SIMD kernels (most MD codes have been and are still doing it just because the standard algorithms require it); bonded F was thought to require negligible amount of time and suddenly ends up being one of the major bottlenecks in 4.6.

---

#9 - 02/27/2013 08:24 PM - Peter Kasson

I think in that case we want to clearly define what parts of the code can contain architecture-specific optimizations and wall those off from the rest of the code. Then if we have libgromacs and libgromacs_opt, we can select among libgromacs_opt_SSE2, libgromacs_opt_AVX128FMA, libgromacs_opt_AVX256_CUDA55, whatever without requiring different copies of the full Gromacs code. If we require full libgromacs libraries, this is going to become prohibitively expensive space-wise.

The thing to be clear about is the following: with the trends we're looking at, the idea that we can set optimizations at compile time and have those be tuned to the runtime hardware will be the corner case, not the other way around.

---

#10 - 02/27/2013 08:45 PM - Erik Lindahl

On my mac, libmd.dylib and libgmx.dylib are 7.3MB together, which I think is very modest rather than prohibitively expensive for each architecture? I would also bet the vast majority of that is kernel-related code, so creating an entire separate walled-off part of the code sounds like a huge amount of development work for very little extra return. Complete separate libraries is the way e.g. the Intel compiler does it.

As Mark & Szilard hinted above, the general problem is simply that we need to head in the opposite direction to be able to get significant speedups on GPUs and future heterogeneous architectures. With relatively slower CPUs and faster GPUs we will have many more small bottleneck routines where we need to use SIMD, or we will lose much more than 10-15% performance.

---

#11 - 02/27/2013 09:21 PM - Peter Kasson

7.3 MB * (SSE2/SSE4/AVX128/AVX256/FMA4/FMA3) * (noGPU/GPU1) gives us conservatively ~86M; if we start getting more combinations, particularly more coprocessor (GPU, etc) paths, this gets even nastier. 86M is a pretty substantial download per client, especially for cases where local caching is not feasible.

As Teemu alluded to (if I understand correctly), we can have architecture-specific routines in the optimization libraries that then get called by the general code. Will that hurt us so much? It sounds like a simple policy--if you're using SIMD, GPU, etc. you put those routines in library A and keep general-purpose code in library B (which can call routines from the other library).

What I worry about is that I think you're pushing things in a direction where the theoretical speed of Gromacs will be great, but the practical speed will be not so good due to these factors. That won't look good for the project. It's worth spending a bit of time and effort averting such an outcome.
We've discussed earlier having a separate reduced functionality libgromacs with a very strict policy on external dependencies, C++ features used, etc. in order to ensure that we can still compile and run mdrun on most machines, even exotic ones, after a lot of potentially complex C++ trickles into the code.

Now, this separation could have a dual role: it could also provide the line between what can be runtime-selectable arch-specific code and what does not need have to be. However, I don't see an easy way to provide runtime-selected arch-specific dynamically loaded libs for the entire libgromacs especially if we decide to provide the above, reduced, mdrun-only libgromacs.

While it might not be very complicated to compile multiple core gromacs libraries with different accelerations and write code that does the run-time selection, it will not be trivial either. Additionally, I don't think that we will all of a sudden see a large number of clusters with mixed CPU architectures showing up all of a sudden. Additionally, I expect that such clusters will have the different nodes in different partitions selectable through the queue system. Other than some mixed clusters (and F@H) what other problematic case are you expecting to be there in the coming years?

7.3 MB * (SSE2/SSE4/AVX128/AVX256/FMA4/FMA3) * (noGPU/GPU1) gives us conservatively ~86M; if we start getting more combinations, particularly more coprocessor (GPU, etc) paths, this gets even nastier. 86M is a pretty substantial download per client, especially for cases where local caching is not feasible.

There are presently only four separate accelerations for x86 Gromacs available, so it sums to maximum <30MB, and it might be 36MB with FMA4+AVX2. FMA3 is the same as AVX_128, and GPU support has nothing to do with it, since that does not limit a binary to GPU-only systems. I realize this might not be ideal for F@H, but the first 7.3MB you have to download anyway, and even on an 8Mbit line we are talking about ~30 seconds.

Obviously, anything is possible if somebody would volunteer to do the work (although a separate library that cuts straight across all classes would make the code difficult to maintain), but on balance I think it's a very reasonable solution to offer full performance on all platforms in exchange for 30 seconds extra download time.

As for theoretical vs. practical, the SIMD torsions in bondfree.c and pme.c are already having a significantly larger performance improvement for the GPU code (in particular in parallel) than the additional benefit of e.g. AVX256 over SSE4.1 for kernels! SSE code is ugly as hell, so we're definitely not adding it because it's fun!
Oops, even I forgot to calculate it correctly. Obviously, we would still have to duplicate all the hardware-specific code in separate libraries, and the above calculation didn’t take that into account! Very roughly, gmxlib+mdlib are about 300k lines-of-code apart from the kernels, which add another >200k lines per architecture (then I count both group and verlet), or 40-50%. That part would still have to be duplicated no matter what. Lines-of-code do not correspond exactly to library size, but if we use that as an estimate we would still add 3.5MB per architecture. Thus, the difference between doing the work to create a separate library and simply having libgromacs.dylib hardware-specific would be ~12.5MB in total for the current four x86 accelerations. I know where I would cast my vote :-)

#15 - 02/27/2013 10:41 PM - Peter Kasson

My point for theoretical versus practical is that you're adding SIMD-dependent code that will give a big speedup in your benchmarks but the majority of gromacs runs won't be taking advantage of it, so in practice it's not helping as much as claimed.

Re. clusters, I work on clusters that are heterogeneous and where selecting by architecture at queue time is difficult/undesirable. So I can say this definitely exists.

It seems no one who has spoken up agrees with me. (If someone else does, please say so.) But I would say that the upshot of this we'll probably support only SSE2 and take a performance hit and explain why if anyone asks (which people do). This makes Gromacs look bad.

#16 - 02/27/2013 11:47 PM - Erik Lindahl

I can’t imagine anybody arguing that it’s a bad idea to support multiple accelerations. The question is how to realize it with the least amount of work.

I think Gromacs-4.6 might be able to survive the bad reputation of only providing ~20% performance improvement over Gromacs-4.5 when the user needs a multi-architecture binary ;)

#17 - 02/28/2013 01:09 AM - Szilárd Páll

I second Erik’s thought, I am not arguing against the flexibility of having a single self-contained mdrun with all features, including acceleration compiled in either. However, this is only one of the several aspects of similar nature that needs to be considered for 5.0. Others are:

- separate reduced functionality mdrun-only libgromacs;
- multiple precision modes in single binary: (smart) single, double, and mixed precision mode (wip/planned);
- support for multiple accelerator types: NVIDIA GPUs, Intel MIC (wip), OpenCL (maybe - we have a masters student who might work on this);
- support for multiple communication libraries: plain MPI, accelerated/RDMA non-blocking communication (wip);

IMHO the first point is quite important, especially as it requires careful design decisions. The others, including the multiple SIMD-acceleration support can and should be considered. We definitely need to have a thorough discussion on them to make sure that at least the design does not make implementing any of the aspects considered important overly difficult - be it 5.0 or some later version where the implementation materializes.

However, considering the limited amount of development resources, we will have to prioritize some aspects more than other others. Now, the multi-SIMD aspects, at least for now, seem to be highly beneficial for Peter’s use-cases as well as for F@H in general. Hence, I think it would be great if the respective parties could pitch in with ideas and discussion at the design phase. If they could also help with development, the solution of having multiple libgromacs_SIMDTYPE-s compiled and dynamically chosen at runtime should be doable and e.g the experience from OpenMM (where AFAIK this technique is employed) could help a lot.
Szilard's points in particular are well taken (as are Erik's).

Looking back at my original post, I might have not communicated things clearly (and mentioning 5.0 in particular might have been a mistake). I know we have a number of priorities for 5.0, and I'm not asking people to commit substantial time and resources here.

What I might suggest that we could agree on:
1. Being able to support multi-SIMD (or multiple CPU hardware acceleration) is ultimately the Right Thing to Do (TM)
2. There are easier but suboptimal and harder but more powerful ways to do this. We may at some point implement an easier way as a "halfway" measure.
3. Given our development priorities at the moment, we won't commit to any particular target version for implementing this
4. However, given this is something we'd like to do, when feasible it would be good to consider this scenario when making design decisions
5. Should we manage to get multi-SIMD support (in a way that we generally find pleasing), we'd be happy :)

Can we reach agreement on something along these lines?

I think design-wise a number of the priorities Szilard mentions have a lot in common.

In terms of my personal resources, I'm probably concentrating more on analysis-type things, but it's conceivable I'd have time to take a crack at some of this. I don't want to make promises I might not be able to deliver on, though.

---

Some thoughts, more on the technical side of things (I simply do not want to get caught in these frustrating arguments, so I will not express any view of my personal preferences...):

- If we want to split the code into optimized/generic parts (or mdrun/non-mdrun parts), there are really three parts in the code: mdrun/optimized, non-mdrun/generic/tools, and shared (such as error handling, file I/O, command-line handling etc.). I see no value in having SIMD optimisation in the shared part, and those parts will be used extensively from the generic/non-mdrun parts as well. Unless we do a very specific split (where the optimized part does not call the shared parts at all), we will need three libraries (unless a circular dependency is OK). This issue does not arise if everything is in a single library.
- Just by switching the compilation to C++ (in particular exceptions), the object code size will increase. In master, libgromacs is 10,5 MB when built in Release mode. If we start using more C++ code on the tools part, the relative size of that will keep on increasing.
- SIMD acceleration in the analysis tools is probably not a high priority, so splitting that part out from the SIMD-optimized part should not be a big deal. And interaction between mdrun and tool code is very small, except for using the mentioned shared parts, so this should not be any significant amount of work either. All the arguments here seem to focus on GROMACS==mdrun, although we do have a significant amount of other code as well...
- I'm not an expert on how shared library loaders work on different platforms, so I can't tell whether there is an easy and portable way to just make an arbitrary split and have the linker resolve the symbols even if the library is chosen at run-time. The simple approach of dlopen() + disym() requires a very limited set of symbols that actually need to be called across the boundary.
- To implement something like this, I think the required things to do are:
  - Add build system support for building parts of the code multiple times, either into multiple libraries or into the same library (don't know how easy the latter is in CMake).
  - Create an interface that is the same for all the optimized parts, and a factory that can choose run-time the correct implementation of the interface.
  - If building into the same library, disambiguate the symbols somehow. The easiest way I can think of is to wrap everything into namespace GMX_OPT, and use -DGMX_OPT=gmx_whatever when compiling. This is mechanical and can be mostly automated.
  - If building into a different library, write portable code to actually load that library dynamically and find the correct symbols.
- From the above, I don't think there is a big difference in implementation effort for the single vs. multiple shared libraries; most of the effort goes into creating the split in the first place, and the different alternatives just require alternative implementations.
- Doing multiple precisions in the same binary requires more or less the same techniques as listed above.
Peter Kasson wrote:

My point for theoretical versus practical is that you're adding SIMD-dependent code that will give a big speedup in your benchmarks but the majority of gromacs runs won't be taking advantage of it, so in practice it's not helping as much as claimed.

That's true only if the people who provide those executables are only prepared to make available the least common denominator. I can see not wanting to provide a download with all possible binaries, but there's solutions like downloading detection routines that trigger the download of the correct binary. Or letting the user take responsibility for downloading a binary that matches their hardware if they want to earn more F@H points. Now they're doing the work for all of us!

Re. clusters, I work on clusters that are heterogeneous and where selecting by architecture at queue time is difficult/undesirable. So I can say this definitely exists.

Sure, but that's an issue for flagging at procurement time as much as it is a problem for people writing the code to run on all possible hardware. Why is it too hard to write a script to grep /proc/cpuinfo and call the right binary accordingly?

It seems no one who has spoken up agrees with me. (If someone else does, please say so.) But I would say that the upshot of this we'll probably support only SSE2 and take a performance hit and explain why if anyone asks (which people do). This makes Gromacs look bad.

GROMACS looks bad only if people answering questions want to off-load the problem... :-) At the other extreme, those users' questions could be answered like "The new GROMACS 4.6 has delivered a ~20% performance bonus for F@H just from the new force-only kernels, even though F@H organizers have only arranged for the least common denominator binary for F@H users. If F@H would let users choose their own F@H binary, then things would would even better. Or people could contribute time to the GROMACS project to fix the underlying problem."

One small step in the direction of the nirvana of a fully portable x86 GROMACS binary is to reduce our wholesale use of compiler flags.

Currently, CMake builds a set that we think are useful for the target accelerated kernels and adds them to the global CMAKE_C_FLAGS. This means that problems like those of #1123 occur, because -mavx will lead to SIGILL long before the detection code gets started. As Szilard suggested there (and again today in discussion in the lab), there may be no need for GROMACS to be so heavy-handed with its flags. It occurred to me this evening that it might be quite simple to have src/gmxlib/nonbonded have a CMakeLists.txt that adds the acceleration flags, and likewise for src/mdlib/nbnxn_kernel/CMakeLists.txt. If that works, then there's a compile-time firewall behind which all the AVX-specific stuff lives. If I'm right, that
solution could be just a dozen lines of CMake.

If so, now it becomes more reasonable to work out how to compile a binary that can call the right kernel at run time. After all, the kernels get called by function pointers right now. Even if official GROMACS doesn't end up taking that route, a derivative work like F@H that has a serious need to cope with hardware heterogeneity might choose it.

Having that compiler firewall will also let us know during the 5.0 process whether what we're doing is extending the amount of code that has to be inside the firewall. That might help us not make the problem all-pervasive in GROMACS.

Is that first step something you might like to try out for us, Peter? If it works, it'll fix #1123, and if the solution is reasonably non-invasive we might regard it as a bug fix suitable for the 4.6 series.

#22 - 03/01/2013 12:14 AM - Peter Kasson
Thank you; I would certainly be happy to take a try.

I recently did a set of Gromacs ports for a cloud platform. IIRC, there were a couple calls in gmxlib as well (I split out both mdlib and gmxlib into _base, _sse2, _sse4, _avx256).
But let me take a closer look. I'll go through this and check back with you as to which source files require which calls.

I agree with you--one issue for 5.0 will be how we deal with the potential growth of files requiring/benefitting from compiler-specific flags. We might want to think if we want to restrict this to the equivalent of mdlib, a specific set of files (that could of course be modified), or something else.

One question is whether we think there are any source files that don't use SSE intrinsics or #ifdefs but do benefit performance-wise from e.g. -mavx. Those will be trickier to track down.

Once again, I appreciate this.

#23 - 03/01/2013 01:15 PM - Szilárd Páll
- File avx_only_where_needed.patch added
- Category set to mdrun

Mark Abraham wrote:

One small step in the direction of the nirvana of a fully portable x86 GROMACS binary is to reduce our wholesale use of compiler flags.

IMHO most compiler flags are meant to be used in a "wholesale" fashion and I don't see any strong relation between the use of e.g. -mavx on every file (when it's anyway supported) compared to using it only on the files that need it.

Currently, CMake builds a set that we think are useful for the target accelerated kernels and adds them to the global CMAKE_C_FLAGS. This means that problems like those of #1123 occur, because -max will lead to SIGILL long before the detection code gets started. As Szilard suggested there (and again today in discussion in the lab), there may be no need for GROMACS to be so heavy-handed with its flags. It occurred to me this evening that it might be quite simple to have src/gmxlib/nonbonded have a CMakeLists.txt that adds the acceleration flags, and likewise for src/mdlib/nbnxn_kernel/CMakeLists.txt. If that works, then there's a compile-time firewall behind which all the AVX-specific stuff lives. If I'm right, that solution could be just a dozen lines of CMake.
Less. See attached. I'm not a fan of the approach, though. This way we will always need to list the SIMD acceleration on a per-file basis and we ditch the performance benefit of compiler optimizations that come with any -mFOO. For this reason, my suggestions was in fact to have what you call a "firewall" at the highest level, i.e. compile only mdrun.c and some detection code without acceleration.

If so, now it becomes more reasonable to work out how to compile a binary that can call the right kernel at runtime. After all, the kernels get called by function pointers right now. Even if official GROMACS doesn't end up taking that route, a derivative work like F@H that has a serious need to cope with hardware heterogeneity might choose it.

That's true, but compiling into mdrun multiple accelerations as well as selection the right one is not trivial. I think for us at the moment the most useful (and feasible to implement) thing would be having a top-level detection warn the user if the acceleration mdrun is compiled with is not supported.

Having that compiler firewall will also let us know during the 5.0 process whether what we're doing is extending the amount of code that has to be inside the firewall. That might help us not make the problem all-pervasive in GROMACS.

Good point. However, I am a bit concerned about setting the goal of limiting arch-specific optimizations to a small, explicitly defined, set of source files. After all, what we were discussing before was auto-SIMD-izing as much as possible (through SIMD-enabled vector/container operations) and I think that is the right way to go. Additionally, as previously mentioned, more and more parts of the code need to get SIMD version if we don't want to wake up that something previously negligible takes 10% of the runtime.

Note:

- To test the patch compile with -DGMX_SKIP_DEFAULT_CFLAGS and you'll need to cherry-pick the flags that would be used otherwise except the acceleration flag, e.g. -mavx.
- With gcc on SB there is only 1-2% difference between using -mavx for all code or only part of it. I suspect that for BD/PD this will be higher, but don't have time to test now.

#24 - 03/01/2013 09:42 PM - Mark Abraham

Szilárd Páll wrote:

Mark Abraham wrote:

One small step in the direction of the nirvana of a fully portable x86 GROMACS binary is to reduce our wholesale use of compiler flags.
IMHO most compiler flags are meant to be used in a “wholesale” fashion and I don’t see any strong relation between the use of e.g. -mavx on every file (when it’s anyway supported) compared to using it only on the files that need it.

It’s the use of -avx that causes the SIGILL in #1123. I forget the details, but there are some times a compiler might use AVX intrinsics when handling strings, which would explain why GROMACS dies fast if you run an AVX binary on non-AVX hardware.

Currently, CMake builds a set that we think are useful for the target accelerated kernels and adds them to the global CMAKE_C_FLAGS. This means that problems like those of #1123 occur, because -mavx will lead to SIGILL long before the detection code gets started. As Szilard suggested there (and again today in discussion in the lab), there may be no need for GROMACS to be so heavy-handed with its flags. It occurred to me this evening that it might be quite simple to have src/gmxlib/nonbonded have a CMakeLists.txt that adds the acceleration flags, and likewise for src/mdlib/nbnxn_kernel/CMakeLists.txt. If that works, then there’s a compile-time firewall behind which all the AVX-specific stuff lives. If I’m right, that solution could be just a dozen lines of CMake.

Less. See attached. I’m not a fan of the approach, though. This way we will always need to list the SIMD acceleration on a per-file basis and we ditch the performance benefit of compiler optimizations that come with any -mFOO. For this reason, my suggestion was in fact to have what you call a “firewall” at the highest level, i.e. compile only mdrun.c and some detection code without acceleration.

That’s also a reasonable approach. One advantage of putting the firewall there is that there is a slight penalty for transitioning to and from AVX regions (See http://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties). It is probably insignificant, but is definitely so if we do it during mdrun setup rather than every kernel call.

If so, now it becomes more reasonable to work out how to compile a binary that can call the right kernel at run time. After all, the kernels get called by function pointers right now. Even if official GROMACS doesn’t end up taking that route, a derivative work like F@H that has a serious need to cope with hardware heterogeneity might choose it.

That’s true, but compiling into mdrun multiple accelerations as well as selection the right one is not trivial. I think for us at the moment the most useful (and feasible to implement) thing would be having a top-level detection warn the user if the acceleration mdrun is compiled with isn’t supported.

Agreed about that first step.

However, with 4.6 infrastructure and the kind of solution in your attached patch, you can get CMake to compile the kernel files multiple times using appropriate flags for the different acceleration paths. We need a bit of file-name and function-name mangling to make it work, but either C or CMake preprocessor can do that. At mdrun detection time, the best kernel type is known, and the existing function pointer table gets loaded appropriately. As a side benefit, we no longer have to think about writing kernels for acceleration types that provide no benefit over earlier technology, we just fall back. (Much like we do now from accelerated kernels to C kernels for Buckingham, IIRC.) This would also mean we can compile the rest of the non-mdrun-entry code with optimal compiler flags if the user took responsibility (setting a CMake variable) for building a non-portable binary.

Given that we plan to design our 5.0 data structures so that there’s a SIMD-friendly flat C arrays for kernels to get at, this kind of solution probably works for 5.0, too. There’s currently not much in the way of plans to do anything to the kernels (other than maybe add ones for new hardware). A problem will come if different SIMD flavours will want different array layouts for SIMD-friendliness - now the versioning could go nearly everywhere in mdrun.

Having that compiler firewall will also let us know during the 5.0 process whether what we’re doing is extending the amount of code that has to be inside the firewall. That might help us not make the problem all-pervasive in GROMACS.

Good point. However, I am a bit concerned about setting the goal of limiting arch-specific optimizations to a small, explicitly defined, set of source files. After all, what we were discussing before was auto-SIMD-izing as much as possible (through SIMD-enabled vector/container operations) and I think that is the right way to go. Additionally, as previously mentioned, more and more parts of the code need to get SIMD version if we don’t want to wake up that something previously negligible takes 10% of the runtime.
I don't recall discussion where targetting auto-SIMDization was agreed. SIMD-friendly data structures are certainly a design target, to ensure we give up no kernel performance. Auto-SIMDization might be a side-effect of the latter, of course, but there's not much compute-intensive code in mdrun that can benefit from it at this time. I think the most important thing is that we design the data structures such that we can solve that 10% problem when we know it is real.

There's much more of a case for designing tools code with auto-SIMDization in mind, of course, but there's not much return for tools that are I/O-bound.

Note:

- To test the patch compile with `-DGMX_SKIP_DEFAULT_CFLAGS` and you'll need to cherry-pick the flags that would be used otherwise except the acceleration flag, e.g `-mavx`.
- With gcc on SB there is only 1-2% difference between using `-mavx` for all code or only part of it. I suspect that for BD/PD this will be higher, but don't have time to test now.

#25 - 03/02/2013 01:36 AM - Peter Kasson

I think this question of whether we get a substantial performance benefit from "auto" SIMD i.e. `-mFOO` is important. Szilard, your data suggest a 1-2% difference?

If we think that most/all of our performance benefit is from explicit SIMD calls, then separating SIMD code from non-SIMD code gets easier (with the caveat Mark referenced about penalties for transferring in and out of SIMD regions; it wasn't clear to me from skimming the article how much of a penalty we take if we use vzeroupper or similar). If we think we get a big win from `-mavx` on code that doesn't use intrinsics, then life gets more complicated.

At the moment, bondfree.c is the major piece of code in gmxlib outside of the nonbonded kernels that uses SSE explicitly (gmx_CPUID has compilation #ifdefs; the best way to deal with this isn't immediately clear to me). There are several pieces of SSE intrinsic code in mdlib.

Where do we anticipate the growth in SSE-specific code (and are there pieces of "allied" code that we'd want to throw in)? Would it make sense to have a set of files that would get SSE-specific optimizations and set that wouldn't? We could either just put this in CMakeLists.txt or swap the directory organization to reflect this.

One way to imagine this "intermediate" scenario would be to have a CMake option that would be GMX_MULTI_ACCELERATION or similar. If set to off, everything would get compiled for the single target acceleration setting. If set to on, then we'd break out files by acceleration setting. (I think this is similar to what Mark suggests above.)

This wouldn't solve issue 1123 in the GMX_MULTI_ACCELERATION=OFF case (there we might run checks a bit earlier in the code?).

11/25/2015
I think this question of whether we get a substantial performance benefit from "auto" SIMD i.e. -mFOO is important. Szilard, your data suggest a 1-2% difference?

See comment #2. The 1-2% is only in a special case for Intel where all other instructions are identical. On AMD bulldozer there is a 17% difference on the ion channel benchmark from using "-march=bdver1 -Ofast" since that enables the compiler to use FMA4 instructions outside the kernels too (Haswell should be similar). In comparison, there is only 9% performance difference from using AVX256 instead of SSE4.1 on Sandy Bridge.

Again, this is not primarily a matter of SIMD code usage in Gromacs, but the fact that the architectures are diverging, so we simply need different flags for most of the code.

---

Not going to happen for 5.0, so removed that from the title.

From other discussion, it seems that there was some support for a solution based on dynamic libraries, where part of the source would get compiled into a separate library that has SIMD optimization on, and non-critical parts get compiled into a generic library that dynamically selects and loads one of the optimized libraries.

To put a boon on the source code reorganization, I can offer (but not promise, since it can be a long time before we actually get there) to look into the dynamic loading code and the necessary build system modifications, but a prerequisite for that is that there is clear line where such a division can be put. So if we get to a point where

- all the code is in subdirectories under src/gromacs/ (i.e., no legacyheaders/), and
- those subdirectories can be split into generic and SIMD-optimized, and
- the generic ones do not depend on any SIMD-optimized code except through a thin interface, and
- I still have time to put into the project,

I'll look into how the multi-library approach could be done in practice.
A further option when gcc 5 is out might be https://gcc.gnu.org/wiki/JIT. Not as good as the real thing, but the bits that use explicit SIMD would be easy to handle, and that is decent for distros or FAH to use.

Files

| avx_only_where_needed.patch | 1.44 KB | 03/01/2013 | Szilárd Páll |