GROMACS - Task #2818

bonded GPU kernel fusion

12/21/2018 07:44 PM - Szilárd Páll

Status:	In Progress	
Priority:	Normal	
Assignee:	Magnus Lundborg	
Category:	mdrun	
Target version:		
Difficulty:	uncategorized	

Description

The launch overhead of the bonded kernels often becomes so significant that it outweighs any benefit of GPU offload. This could be mitigated with a few optimizations: most importantly kernel fusion.

Multiple approaches are possible:

- a simple decomposition of different types of bonded interactions over different blocks
- a more locality aware-decomposition would however be beneficial: interactions that share the coordinates computed in the same block.

Additionally, update groups could also be implemented in the decomposition (e.g. through block sorting which should not be difficult if coordinate ranges are already the unit of work) to prioritize work on the critical path for staggered update as well as for DD runs.

Related issues:

Related to GROMACS - Task #2694: bonded CUDA kernels	**Closed**
Related to GROMACS - Task #2675: bonded CUDA offload task	**In Progress**

Associated revisions

Revision 01b2f20b - 06/28/2019 12:37 PM - Magnus Lundborg

Fused GPU bonded kernels

To reduce the GPU kernel launch times the
GPU bonded kernels have been fused to a single kernel.

Refs #2818

Change-Id: I2299e1857df5db469739e3aefbc0f771968a6bd5

History

#1 - 12/21/2018 07:44 PM - Szilárd Páll

- Related to Task #2694: bonded CUDA kernels added

#2 - 12/23/2018 01:46 PM - Szilárd Páll

- Description updated

#3 - 03/08/2019 06:08 PM - Szilárd Páll

- Related to Task #2675: bonded CUDA offload task added

#4 - 03/08/2019 06:21 PM - Szilárd Páll

Based on the past and recent look at current kernels, these are the current conclusions

- kernels are severely memory/instruction limited mostly due to scattered coordinate loads and to a lesser extent non-vecorizable parameter loads
- due to alignment parameter loads can't always be vectorized (and even when they can, for adh on a 1070 for angles it did not help)
- the best approach would be efficiently loading coordinates but if time doesn't allow
- at least simple fusion within or across blocks should be implemented -- both would greatly improve cache reuse;
  - fusion within blocks may be more efficient if there is other work to overlap with but how to map different interaction types to warps to avoid leaving lots of threads idle is not clear.

#5 - 05/21/2019 02:44 PM - Magnus Lundborg

- Assignee set to Magnus Lundborg

I'm assigning this to myself for now. Hopefully I can do this, but will probably need some advice along the way.

Szilárd said:
"... there are two strategies "horizontal" and "vertical" fusion.

1) Horizontal, across GPU blocks: we'd need N_total = sum(N_type) threads launched, compute an index range for all types and map the GPU threadIndex to the individual bonded index ranges. This is relatively straightforward, but may not be optimal assuming there is other work to overlap with

2) Vertical: within GPU blocks: launch N_total = max(N_type) threads and call the different interaction types sequentially within the kernel; ideally we'd want to map such that warps/blocks reuse atom data at least from cache, but this would require scattering the interaction types across blocks, then compacting as much as possible to avoid empty warps. If we could do a simple but smart sorting into blocks, this may be more efficient overall (especially if it can be nearly as efficient a hypothetical fine-grained update-group based decomposition)."

I will go for alternative 1 for now. I realise that option 2 would be better, but I think it's currently good just to avoid the multiple kernel launches. Do I understand it correctly that for option 1 updateInteractionListsAndDeviceBuffers() could remain as it is?

#6 - 07/03/2019 09:01 PM - Szilárd Páll

- Status changed from New to In Progress

I've just done some quick checks and for the GluCL (ion_channel_ benchmark, the virial + energy kernel got ~1.4x slower (compared to the sum of the wall-time of all equivalent kernels in the r2019). I've also checked narrower blocks and it seems that the atomic accumulation seems to be the culprit creating even more contention on the L2 cache (additionally to the low efficiency loads). Making blocks narrower seems to improve performance by nearly 2x (even 64-wide improves further, but I realize that without direct global reduction that won't work for the shift vectors).