Chair for System Simulation

Harald Köstler, Dr.-Ing.
Rafael Ravedutti Lucio Machado

High End Simulation in Practice

Summer Term 2021

Assignment 3: Acceleration Techniques

Suggested Work Schedule

- solve the assignment: 31.05.2021 – 13.06.2021
- present the assignment: 14.06.2021 – 18.06.2021

There are no hard deadlines, however, we highly recommend sticking to this schedule as there will be new assignments regularly.

Mandatory Submission Guidelines

- submit in groups of two to four (you can use the forum to find team members)
- upload your solution as a group to StudOn
- your solution MUST include an appropriate Makefile
- the examination will contain questions about the exercises, so make sure you understood everything
- additional requirements may be stated in the tasks

We do not provide a solution for the *blocks_big* example. Due to numerical inaccuracies the results might vary. Please make sure your solution looks reasonable.

In this exercise you will improve the performance and scalability of last assignment's simulation application. You are free to use either CUDA or OpenCL (of course you can also code both versions).

1 Theory

As you have determined in the last exercise, the force calculation step of your simulation scales with $\mathcal{O}(N^2)$ which inhibits scalability. One technique to reduce the complexity is to split the computational domain into cells and assigning particles to them. With that approach the neighborhood information can be restricted. This approach is usually referred to as *binning* or *linked-cell algorithm*. Another option is explicitly storing neighbor lists for each particle. To form an efficient algorithm, the extend of the neighborhood is usually increased to make updated of the acceleration structure only necessary every couple of steps. We call this approach *neighbor lists*.

2 Implementation

The first step in implementing the linked-cell algorithm is given by supporting domains with limited extend. To this end, you shall introduce axis-aligned cuboids. Compared to *assignment 2*, you have to extend your parameter file so that it can handle these domains. Therefore the following parameters have to be added:

- Floating point values `x_min`, `y_min` and `z_min` to specify the coordinates of the lower corner point and `x_max`, `y_max` and `z_max` to specify the upper corner point.
- Unsigned integer values `x_n`, `y_n` and `z_n` to specify the number of cells in each direction (in preparation of the linked-cell algorithm).

From these parameters you can derive the remaining domain parameters. The *cell length* in x-dimension calculates as `len_x=(x_max-x_min)/x_n` (analoguous for y and z-direction). The global domain then is

$$\Omega = [\texttt{x\_min}, \texttt{x\_max}] \times [\texttt{y\_min}, \texttt{y\_max}] \times [\texttt{z\_min}, \texttt{z\_max}]. \tag{1}$$

Next, boundary conditions are required. For this assignment, you shall implement periodic boundary conditions, i.e. particles that leave the domain shall enter the domain again at the opposite side. The periodic behavior has to be considered in two ways:

1. After each position update the positions of particles have to be checked: if particles have left the domain their positions have to be corrected. Note that this also has to be considered at initialization time.

2. In the force calculation: cells on the opposite side of the domain may belong to the neighborhood and the distance vector $\mathbf{x}_{ij}$ has to be corrected if two particles are on different sides of a boundary.

After boundary conditions are in place (and working!) you can start with implementing the binning approach. As discussed in the lecture, you have to assign all the particles to a cell, according to their positions. This basically means building a linked list of indices for each cell, which has to be re-done with some periodicity (update frequency). The value for the update frequency must be included in the parameter file. Moreover, you need to add the *cut-off radius* for the force calculation, given by `r_cut`, to the parameter file. You will also have a skin parameter given by `r_skin`, which is a parameter introduced so you do not need to update the cell lists or neighbor lists every timestep. Notice that when building the acceleration structures, you should compare atoms distances with `r_cut + r_skin`, but when computing the forces, you only should compare the distances with `r_cut`. Assert that `r_cut + r_skin` is not larger than the smallest cell length. When implementing, also make sure you use the force term coming from the truncated Lennard-Jones potential. For the force calculation, you need to be able to support the following strategies:

- **brute-force:** this strategy is similar to the one presented in the previous assignment, with the exception that you should check if the distance for atoms is less than the force cutoff radius (`r_cut`) parameter introduced.
- **particle-parallel binning:** for particle-parallel binning force calculation, each particle is assigned to a GPU thread, and this thread computes the force only for this particle. The particle neighbors are obtained by iterating over the neighbor cells/bins (including the current cell/bin and checking if the neighbor particle is not the assigned particle) and then iterating over the particles within these cells.
- **cell-parallel binning:** for cell-parallel force calculation, each cell/bin is assigned to a GPU thread, and this thread computes the force for all particles within this cell. The particle neighbors are obtained in the same way as in the previous strategy, iterating over the neighbor cells/bins and then the particles within them.
- **neighbor lists:** here you should use the particle-parallel strategy (each particle is assigned to a GPU thread), and the particle neighbors must be obtained by traversing the neighbor lists for the current particle. Notice that a new kernel must be written for building the neighbor lists, in this kernel you can use the cell lists data structure in the same fashion as you used to obtain the neighbors for the binning approaches (therefore you use the cell lists to build the neighbor lists). Think about what are the advantages of using neighbor lists over the linked cells structure in the force calculation.

For all strategies: do not forget to check if the atoms distance is smaller than the force cutoff radius (r_cut) before computing the forces!

In the case of cell-parallel implementation, it is a good idea to do a 3D indexing of the work-items. Therefore, extend the parameter file by the unsigned integer values `cl_workgroup_3dsize_x`, `cl_workgroup_3dsize_y` and `cl_workgroup_3dsize_z` in addition to the 1D `cl_workgroup_1dsize` (or suitable CUDA variants). The latter might be further used for kernels that are organized in 1D indexed work-items, i.e. executed particle-parallel.

Next, you can start to implement the neighbor lists. Here, each particle keeps a list of other particles in the neighborhood. Extend the parameter file with required parameters for the update frequency (as discussed previously) for the acceleration structure, i.e. how many time steps may pass until a rebuild is necessary, and the maximum number of items in the neighborhood list. You can also calculate these parameters dynamically at runtime. Think about suitable values for these parameters!

As in the previous assignment, we provide some examples to test your code:

1. *grid*: This small ensemble should stay in perfect equilibrium, the particles should not move at all. If they move slightly then the Lennard-Jones force is probably not calculated in a numerically stable way.
2. *periodicity*: This small ensemble can be used as a check if the periodic boundaries work correctly.
3. *blocks_big*: This is a larger ensemble (1250 particles) that can be used for performance analysis. Note that the lower corner point of the domain is not the origin here! Note also that this setup can get unstable if you run it longer than specified in the parameter file.

Before you continue make sure that all variants of the code (brute force, cell-parallel binning, particle-parallel binning and neighbor lists) yield similar results!

OPTIONAL: The neighbor lists data structure can be defined by a raw bi-dimensional array in C/C++ (i.e. a matrix). For this case, you would get the **k**-th neighbor for a particle **i** by **j = neighborlists[i][k]**, where $0 \leq \mathbf{k} < \mathbf{num\_neighs[i]}$, and **numneighs** is another integer array that contains the amount of neighbors the particle **i** has. Another possibility is to change the arrangement of this array as you would have in Fortran (i.e. column major order), then you would have **j = neighborlists[k][i]** instead. Allow your neighbor lists to be flexible in this way and evaluate the impact on performance when switching the arrangement. Think about the reason that the second approach should deliver better performance on GPU devices by improving coalesced memory access. Include your remarks on the PDF containing your findings (mentioned in the next section).

3 Performance comparison

After successfully implementing the acceleration techniques it is time to compare the performance. We consider all four variants, i.e. brute force, cell-parallel binning, particle-parallel binning and neighbor lists. First, find suitable values for specialized parameters to tune the single variants. Next, compare the obtained performance results. Compile a PDF showing your findings (parameter configurations and timings!) as well as your interpretation of the results.

As always, please think about which parts of your program you want to profile and what you want to examine before taking any measurements!

4 Submission

Upload your code for all four cases (brute force, cell-parallel binning, particle-parallel binning and neighbor lists) together with a suitable Makefile and your performance comparison PDF via StudOn.