sphere Documentation

Release 0.35

Anders Damsgaard

January 06, 2014

CONTENTS

This is the official documentation for the *sphere* discrete element modelling software. It presents the theory behind the discrete element method (DEM), the structure of the software source code, and the Python API for handling simulation setup and data analysis.

sphere is developed by Anders Damsgaard Christensen under supervision of David Lunbek Egholm and Jan A. Piotrowski, all of the department of Geoscience, Aarhus University, Denmark. This document is a work in progress, and is still in an early state.

Contact: Anders Damsgaard Christensen, http://cs.au.dk/~adc, mailto:anders.damsgaard@geo.au.dk

Contents:

CHAPTER

ONE

INTRODUCTION

The *sphere*-software is used for three-dimensional discrete element method (DEM) particle simulations. The source code is written in C++, CUDA C and Python, and is compiled by the user. The main computations are performed on the graphics processing unit (GPU) using NVIDIA's general purpose parallel computing architecture, CUDA. Simulation setup and data analysis is performed with the included Python API.

The ultimate aim of the *sphere* software is to simulate soft-bedded subglacial conditions, while retaining the flexibility to perform simulations of granular material in other environments.

The purpose of this documentation is to provide the user with a walk-through of the installation, work-flow, data-analysis and visualization methods of *sphere*. In addition, the *sphere* internals are exposed to provide a way of understanding of the discrete element method numerical routines taking place.

Note: Command examples in this document starting with the symbol `$` are meant to be executed in the shell of the operational system, and `>>>` means execution in Python.

All numerical values in this document, the source code, and the configuration files are typeset with strict respect to the SI unit system.

1.1 Requirements

The build requirements are:

- A Nvidia CUDA-supported version of Linux or Mac OS X (see the CUDA toolkit release notes for more information)
- GNU Make
- CMake
- The GNU Compiler Collection (GCC)
- The Nvidia CUDA toolkit and SDK

The runtime requirements are:

- A CUDA-enabled GPU with compute capability 1.1 or greater.
- A Nvidia CUDA-enabled GPU and device driver

Optional tools, required for simulation setup and data processing:

- Python 2.7
- Numpy

- Matplotlib
- Imagemagick
- ffmpeg

Optional tools, required for building the documentation:

- Sphinx
  - sphinxcontrib-programoutput
- Doxygen
- Breathe
- dvipng

Git is used as the distributed version control system platform, and the source code is maintained at Github. *sphere* is licensed under the GNU Public License, v.3.

1.2 Building *sphere*

All instructions required for building *sphere* are provided in a number of `Makefile`'s. To generate the main *sphere* command-line executable, go to the root directory, and invoke CMake and GNU Make:

```
$ cmake . && make
```

If successfull, the Makefiles will create the required data folders, object files, as well as the *sphere* executable in the root folder. Issue the following commands to check the executable:

```
$ ./sphere --version
```

The output should look similar to this:

```
.--------------------------------------.
|               _    Compiled for 3D   |
|              | |                     |
|    ___ _ __  | |__   ___ _ __ ___    |
|   / __| '_ \| '_ \ / _ \ '__/ _ \   |
|   \__ \ |_) | | | |  __/ | |  __/   |
|   |___/ .__/|_| |_|\___|_|  \___|   |
|       | |                           |
|       |_|           Version: 0.35    |
`--------------------------------------´
 A discrete element method particle dynamics simulator.
 Written by Anders Damsgaard Christensen, license GPLv3+.
```

The build can be verified by running a number of automated tests:

```
$ make test
```

The documentation can be read in the reStructuredText-format in the `doc/sphinx/` folder, or build into e.g. HTML or PDF format with the following commands:

```
$ cd doc/sphinx
$ make html
$ make latexpdf
```

To see all available output formats, execute:

```
$ make help
```

1.3 Work flow

After compiling the *sphere* binary, the procedure of a creating and handling a simulation is typically arranged in the following order:

- Setup of particle assemblage, physical properties and conditions using the Python API.
- Execution of *sphere* software, which simulates the particle behavior as a function of time, as a result of the conditions initially specified in the input file.
- Inspection, analysis, interpretation and visualization of *sphere* output in Python, and/or scene rendering using the built-in ray tracer.

CHAPTER

TWO

DISCRETE ELEMENT METHOD

The discrete element method (or distinct element method) was initially formulated by Cundall and Strack (1979). It simulates the physical behavior and interaction of discrete, unbreakable particles, with their own mass and inertia, under the influence of e.g. gravity and boundary conditions such as moving walls. By discretizing time into small time steps, explicit integration of Newton's second law of motion is used to predict the new position and kinematic values for each particle from the previous sums of forces. This Lagrangian approach is ideal for simulating discontinuous materials, such as granular matter. The complexity of the computations is kept low by representing the particles as spheres, which keeps contact-searching algorithms simple.

CHAPTER

THREE

PYTHON API

class sphere.**Spherebin**(*np=1*, *nd=3*, *nw=1*, *sid='unnamed'*)

Class containing all data SPHERE data.

Contains functions for reading and writing binaries, as well as simulation setup and data analysis.

addParticle(*x*, *radius*, *xysum=array([0., 0.])*, *vel=array([0., 0., 0.])*, *fixvel=array([0.])*, *force=array([0., 0., 0.])*, *angpos=array([0., 0., 0.])*, *angvel=array([0., 0., 0.])*, *torque=array([0., 0., 0.])*, *es_dot=array([0.])*, *es=array([0.])*, *ev_dot=array([0.])*, *ev=array([0.])*, *p=array([0.])*)

Add a single particle to the simulation object. The only required parameters are the position (x), a length-three array, and the radius (radius), a length-one array.

adjustUpperWall(*z_adjust=1.1*)

Included for legacy purposes, calls adjustWall with idx=0

adjustWall(*idx*, *adjust=1.1*)

Adjust grid and dynamic wall to max. particle position

bond(*i*, *j*)

Create a bond between particles i and j

bondsRose(*imgformat='pdf'*)

Visualize strike- and dip angles of the bond pairs in a rose plot.

bulkPorosity()

Calculate and return the bulk porosity

consolidate(*deviatoric_stress=10000.0*, *periodic=1*)

Setup consolidation experiment. Specify the upper wall deviatoric stress in Pascal, default value is 10 kPa.

createBondPair(*i*, *j*, *spacing=-0.1*)

Bond particles i and j. Particle j is moved adjacent to particle i, and oriented randomly. @param spacing (float) The inter-particle distance prescribed. Positive values result in a inter-particle distance, negative equal an overlap. The value is relative to the sum of the two radii.

currentDevs()

Return current magnitude of the deviatoric normal stress

defaultParams(*mu_s=0.4*, *mu_d=0.4*, *mu_r=0.0*, *rho=2600*, *k_n=1160000000.0*, *k_t=1160000000.0*, *k_r=0*, *gamma_n=0*, *gamma_t=0*, *gamma_r=0*, *gamma_wn=10000.0*, *gamma_wt=10000.0*, *capillaryCohesion=0*, *nu=0.0*)

Initialize particle parameters to default values. Radii must be set prior to calling this function.

energy(*method*)

Calculate the sum of the energy components of all particles.

forcechains(*lc=200.0*, *uc=650.0*, *outformat='png'*, *disp='2d'*)
Visualizes the force chains in the system from the magnitude of the normal contact forces, and produces an image of them. @param lc: Lower cutoff of contact forces. Contacts below are not visualized (float) @param uc: Upper cutoff of contact forces. Contacts above are visualized with this value (float) @param outformat: Format of output image. Possible values are 'interactive', 'png', 'epslatex', 'epslatex-color' @param disp: Display forcechains in '2d' or '3d' Warning: Will segfault if no contacts are found.

forcechainsRose(*lower_limit=0.25*)
Visualize strike- and dip angles of the strongest force chains in a rose plot.

generateBimodalRadii(*r_small=0.005*, *r_large=0.05*, *ratio=0.2*, *verbose=True*)
Draw radii from two sizes @param r_small: Radii of small population (float), in]0;r_large[@param r_large: Radii of large population (float), in]r_small;inf[@param ratio: Approximate volumetric ratio between the two populations (large/small).

generateRadii(*psd='logn'*, *radius_mean=0.00044*, *radius_variance=8.8e-09*, *histogram=True*)
Draw random particle radii from the selected probability distribution. Specify mean radius and variance. The variance should be kept at a very low value.

initFluid(*nu=0.00089*)
Initialize the fluid arrays and the fluid viscosity

initGrid()
Initialize grid suitable for the particle positions set previously. The margin parameter adjusts the distance (in no. of max. radii) from the particle boundaries.

initGridAndWorldsize(*g=array([0., 0., -9.80665])*, *margin=2.0*, *periodic=1*, *contactmodel=2*)
Initialize grid suitable for the particle positions set previously. The margin parameter adjusts the distance (in no. of max. radii) from the particle boundaries.

initGridPos(*g=array([0., 0., -9.80665])*, *gridnum=array([12, 12, 36])*, *periodic=1*, *contactmodel=2*)
Initialize particle positions in loose, cubic configuration. Radii must be set beforehand. xynum is the number of rows in both x- and y- directions.

initRandomGridPos(*g=array([0., 0., -9.80665])*, *gridnum=array([12, 12, 32])*, *periodic=1*, *contactmodel=2*)
Initialize particle positions in loose, cubic configuration. Radii must be set beforehand. xynum is the number of rows in both x- and y- directions.

initRandomPos(*g=array([0., 0., -9.80665])*, *gridnum=array([12, 12, 36])*, *periodic=1*, *contactmodel=2*)
Initialize particle positions in loose, cubic configuration. Radii must be set beforehand. xynum is the number of rows in both x- and y- directions.

initTemporal(*total*, *current=0.0*, *file_dt=0.05*, *step_count=0*)
Set temporal parameters for the simulation. Particle radii and physical parameters need to be set prior to these.

plotFluidPorositiesY(*iteration=-1*, *y=-1*, *outformat='png'*)
Plot the porosity values from the simulation. If iteration is -1 (default value), the last output file will be shown. If the y value is -1, the center x,z plane will be rendered

porosities(*outformat='pdf'*, *zslices=16*)
Plot porosities with depth

porosity(*slices=10*, *verbose=False*)
Calculate the porosity as a function of depth, by averaging values in horizontal slabs. Returns porosity values and depth

random2bonds(*ratio=0.3*, *spacing=-0.1*)
Bond an amount of particles in two-particle clusters @param ratio: The amount of particles to bond, values

in]0.0;1.0] (float) @param spacing: The distance relative to the sum of radii between bonded particles, neg. values denote an overlap. Values in]0.0,inf[(float). The particles should be initialized beforehand. Note: The actual number of bonds is likely to be somewhat smaller than specified, due to the random selection algorithm.

readbin (*targetbin*, *verbose=True*, *bonds=True*, *devsmod=True*, *fluid=True*, *esysparticle=False*)
Reads a target SPHERE binary file

readfirst (*verbose=True*)
Read first output file of self.sid

readlast (*verbose=True*)
Read last output binary of self.sid from output/ folder.

readsecond (*verbose=True*)
Read second output file of self.sid

readstep (*step*, *verbose=True*)
Read output binary from time step 'step' from output/ folder.

render (*method='pres'*, *max_val=1000.0*, *lower_cutoff=0.0*, *graphicsformat='png'*, *verbose=True*)
Render all output files that belong to the simulation, determined by sid.

run (*verbose=True*, *hideinputfile=False*, *dry=False*, *valgrind=False*, *cudamemcheck=False*, *cfd=False*)
Execute sphere with target project

shear (*shear_strain_rate=1*, *periodic=1*)
Setup shear experiment. Specify the upper wall deviatoric stress in Pascal, default value is 10 kPa. The shear strain rate is the shear length divided by the initial height per second.

sheardisp (*outformat='pdf'*, *zslices=32*)
Show particle x-displacement vs. the z-pos

shearstrain ()
Calculates and returns the current shear strain (gamma) value of the experiment

shearvel ()
Calculates and returns the shear velocity (gamma_dot) of the experiment

status ()
Show the current simulation status

thinsection_x1x3 (*x2='center'*, *graphicsformat='png'*, *cbmax=None*, *arrowscale=0.01*, *velarrowscale=1.0*, *slipscale=1.0*, *verbose=False*)
Produce a 2D image of particles on a x1,x3 plane, intersecting the second axis at x2. Output is saved as '<sid>-ts-x1x3.txt' in the current folder.

An upper limit to the pressure color bar range can be set by the cbmax parameter.

The data can be plotted in gnuplot with: gnuplot> set size ratio -1 gnuplot> set palette defined (0 "blue", 0.5 "gray", 1 "red") gnuplot> plot '<sid>-ts-x1x3.txt' with circles palette fs transparent solid 0.4 noborder

torqueScript (*email='adc@geo.au.dk'*, *email_alerts='ae'*, *walltime='24:00:00'*, *queue='qfermi'*, *cudapath='/com/cuda/4.0.17/cuda'*, *spheredir='/home/adc/code/sphere'*, *workdir='/scratch'*)
Create job script for the Torque queue manager for the binary

triaxial (*wvel=-0.001*, *deviatoric_stress=10000.0*)
Setup triaxial experiment. The upper wall is moved at a fixed velocity in m/s, default values is -0.001 m/s (i.e. downwards). The side walls are exerting a deviatoric stress

uniaxialStrainRate(*wvel=-0.001*, *periodic=1*)
Setup consolidation experiment. Specify the upper wall velocity in m/s, default value is -0.001 m/s (i.e. downwards).

video(*out_folder='./'*, *video_format='mp4'*, *graphics_folder='../img_out/'*, *graphics_format='png'*, *fps=25*, *qscale=1*, *bitrate=1800*, *verbose=False*)
Use ffmpeg to combine images to animation. All images should be rendered beforehand.

voidRatio()
Returns the current void ratio

writeFluidVTK(*folder='../output/'*, *verbose=True*)
Writes fluid data to a target VTK file

writeVTK(*folder='../output/'*, *verbose=True*)
Writes to a target VTK file

writeVTKall()
Writes all output binaries from the simulation to VTK files

writebin(*folder='../input/'*, *verbose=True*)
Writes to a target SPHERE binary file

zeroKinematics()
Zero kinematics of particles

sphere.**V_sphere**(*r*)
Returns the volume of a sphere with radius r

sphere.**cleanup**(*spherebin*)
Remove input/output files and images from simulation

sphere.**convert**(*graphicsformat='png'*, *folder='../img_out'*)
Converts all PPM images in img_out to graphicsformat, using ImageMagick

sphere.**render**(*binary*, *method='pres'*, *max_val=1000.0*, *lower_cutoff=0.0*, *graphicsformat='png'*, *verbose=True*)
Render target binary using the sphere raytracer.

sphere.**run**(*binary*, *verbose=True*, *hideinputfile=False*)
Execute sphere with target binary as input

sphere.**status**(*project*)
Check the status.dat file for the target project, and return the last file numer.

sphere.**thinsectionVideo**(*project*, *out_folder='./'*, *video_format='mp4'*, *fps=25*, *qscale=1*, *bitrate=1800*, *verbose=False*)
Use ffmpeg to combine thin section images to animation. This function will start off by rendering the images.

sphere.**torqueScriptParallel3**(*obj1*, *obj2*, *obj3*, *email='adc@geo.au.dk'*, *email_alerts='ae'*, *walltime='24:00:00'*, *queue='qfermi'*, *cudapath='/com/cuda/4.0.17/cuda'*, *spheredir='/home/adc/code/sphere'*, *workdir='/scratch'*)
Create job script for the Torque queue manager for three binaries, executed in parallel. Returns the filename of the script

sphere.**torqueScriptSerial3**(*obj1*, *obj2*, *obj3*, *email='adc@geo.au.dk'*, *email_alerts='ae'*, *walltime='24:00:00'*, *queue='qfermi'*, *cudapath='/com/cuda/4.0.17/cuda'*, *spheredir='/home/adc/code/sphere'*, *workdir='/scratch'*)
Create job script for the Torque queue manager for three binaries

sphere.**video**(*project*, *out_folder='./'*, *video_format='mp4'*, *graphics_folder='../img_out/'*, *graphics_format='png'*, *fps=25*, *qscale=1*, *bitrate=1800*, *verbose=False*)

Use ffmpeg to combine images to animation. All images should be rendered beforehand.

sphere.**visualize**(*project*, *method='energy'*, *savefig=True*, *outformat='png'*)

Visualize output from the target project, where the temporal progress is of interest.

CHAPTER

FOUR

SPHERE INTERNALS

The *sphere* executable has the following options:

```
$ ../../sphere --help
../../sphere: particle dynamics simulator
Usage: ../../sphere [OPTION[S]]... [FILE1 ...]
Options:
-h, --help                print help
-V, --version                print version information and exit
-q, --quiet               suppress status messages to stdout
-n, --dry                 show key experiment parameters and quit
-r, --render                render input files instead of simulating temporal evolution
-dc, --dont-check         don't check values before running

Raytracer (-r) specific options:
-m <method> <maxval> [-l <lower cutoff val>], or
--method <method> <maxval> [-l <lower cutoff val>]
        color visualization method, possible values:
        normal, pres, vel, angvel, xdisp, angpos
        'normal' is the default mode
        if -l is appended, don't render particles with value below
```

The most common way to invoke *sphere* is however via the Python API (e.g. `sphere.run()`, `sphere.render()`, etc.).

subsection{The *sphere* algorithm} label{subsec:spherealgo} The *sphere*-binary is launched from the system terminal by passing the simulation ID as an input parameter; texttt{./sphere_<architecture> <simulation_ID>}. The sequence of events in the program is the following: #. System check, including search for NVIDIA CUDA compatible devices (texttt{main.cpp}).

1. Initial data import from binary input file (texttt{main.cpp}).
2. Allocation of memory for all host variables (particles, grid, walls, etc.) (texttt{main.cpp}).
3. Continued import from binary input file (texttt{main.cpp}).
4. Control handed to GPU-specific function texttt{gpuMain(ldots)} (texttt{device.cu}).
5. Memory allocation of device memory (texttt{device.cu}).
6. Transfer of data from host to device variables (texttt{device.cu}).
7. Initialization of Thrustfootnote{url{https://code.google.com/p/thrust/}} radix sort configuration (texttt{device.cu}).
8. Calculation of GPU workload configuration (thread and block layout) (texttt{device.cu}).

9. Status and data written to verb"<simulation_ID>.status.dat" and verb"<simulation_ID>.output0.bin", both located in texttt{output/} folder (texttt{device.cu}).
10. Main loop (while texttt{time.current <= time.total}) (functions called in texttt{device.cu}, function definitions in seperate files). Each kernel call is wrapped in profiling- and error exception handling functions:

1. label{loopstart}CUDA thread synchronization point.
2. texttt{calcParticleCellID<<<,>>>(ldots)}: Particle-grid hash value calculation (texttt{sorting.cuh}).
3. CUDA thread synchronization point.
4. texttt{thrust::sort_by_key(ldots)}: Thrust radix sort of particle-grid hash array (texttt{device.cu}).
5. texttt{cudaMemset(ldots)}: Writing zero value (texttt{0xffffffff}) to empty grid cells (texttt{device.cu}).
6. texttt{reorderArrays<<<,>>>(ldots)}: Reordering of particle arrays, based on sorted particle-grid-hash values (texttt{sorting.cuh}).
7. CUDA thread synchronization point.
8. Optional: texttt{topology<<<,>>>(ldots)}: If particle contact history is required by the contact model, particle contacts are identified, and stored per particle. Previous, now non-existant contacts are discarded (texttt{contactsearch.cuh}).
9. CUDA thread synchronization point.
10. texttt{interact<<<,>>>(ldots)}: For each particle: Search of contacts in neighbor cells, processing of optional collisions and updating of resulting forces and torques. Values are written to read/write device memory arrays (texttt{contactsearch.cuh}).
11. CUDA thread synchronization point.
12. texttt{integrate<<<,>>>(ldots)}: Updating of spatial degrees of freedom by a second-order Taylor series expansion integration (texttt{integration.cuh}).
13. CUDA thread synchronization point.
14. texttt{summation<<<,>>>(ldots)}: Particle contributions to the net force on the walls are summated (texttt{integration.cuh}).
15. CUDA thread synchronization point.
16. texttt{integrateWalls<<<,>>>(ldots)}: Updating of spatial degrees of freedom of walls (texttt{integration.cuh}).
17. Update of timers and loop-related counters (e.g. texttt{time.current}), (texttt{device.cu}).
18. If file output interval is reached:

item Optional write of data to output binary (verb"<simulation_ID>.output#..bin"), (texttt{file_io.cpp}). item Update of verb"<simulation_ID>.status#..bin" (texttt{device.cu}).

item Return to point ref{loopstart}, unless texttt{time.current >= time.total}, in which case the program continues to point ref{loopend}.

1. label{loopend}Liberation of device memory (texttt{device.cu}).
2. Control returned to texttt{main(ldots)}, liberation of host memory (texttt{main.cpp}).
3. End of program, return status equal to zero (0) if no problems where encountered.

4.1 Numerical algorithm

The *sphere*-binary is launched from the system terminal by passing the simulation ID as an input parameter; texttt{./sphere_<architecture> <simulation_ID>}. The sequence of events in the program is the following:

1. System check, including search for NVIDIA CUDA compatible devices (texttt{main.cpp}).
2. Initial data import from binary input file (texttt{main.cpp}).
3. Allocation of memory for all host variables (particles, grid, walls, etc.) (texttt{main.cpp}).
4. Continued import from binary input file (texttt{main.cpp}).
5. Control handed to GPU-specific function texttt{gpuMain(ldots)} (texttt{device.cu}).
6. Memory allocation of device memory (texttt{device.cu}).
7. Transfer of data from host to device variables (texttt{device.cu}).
8. Initialization of Thrustfootnote{url{https://code.google.com/p/thrust/}} radix sort configuration (texttt{device.cu}).
9. Calculation of GPU workload configuration (thread and block layout) (texttt{device.cu}).
10. Status and data written to verb"<simulation_ID>.status.dat" and verb"<simulation_ID>.output0.bin", both located in texttt{output/} folder (texttt{device.cu}).
11. Main loop (while texttt{time.current <= time.total}) (functions called in texttt{device.cu}, function definitions in seperate files). Each kernel call is wrapped in profiling- and error exception handling functions:

1. label{loopstart}CUDA thread synchronization point.
2. texttt{calcParticleCellID<<<,>>>(ldots)}: Particle-grid hash value calculation (texttt{sorting.cuh}).
3. CUDA thread synchronization point.
4. texttt{thrust::sort_by_key(ldots)}: Thrust radix sort of particle-grid hash array (texttt{device.cu}).
5. texttt{cudaMemset(ldots)}: Writing zero value (texttt{0xffffffff}) to empty grid cells (texttt{device.cu}).
6. texttt{reorderArrays<<<,>>>(ldots)}: Reordering of particle arrays, based on sorted particle-grid-hash values (texttt{sorting.cuh}).
7. CUDA thread synchronization point.
8. Optional: texttt{topology<<<,>>>(ldots)}: If particle contact history is required by the contact model, particle contacts are identified, and stored per particle. Previous, now non-existant contacts are discarded (texttt{contactsearch.cuh}).
9. CUDA thread synchronization point.
10. texttt{interact<<<,>>>(ldots)}: For each particle: Search of contacts in neighbor cells, processing of optional collisions and updating of resulting forces and torques. Values are written to read/write device memory arrays (texttt{contactsearch.cuh}).
11. CUDA thread synchronization point.
12. texttt{integrate<<<,>>>(ldots)}: Updating of spatial degrees of freedom by a second-order Taylor series expansion integration (texttt{integration.cuh}).
13. CUDA thread synchronization point.
14. texttt{summation<<<,>>>(ldots)}: Particle contributions to the net force on the walls are summated (texttt{integration.cuh}).
15. CUDA thread synchronization point.

16. texttt{integrateWalls<<<,>>>(ldots)}: Updating of spatial degrees of freedom of walls (texttt{integration.cuh}).
17. Update of timers and loop-related counters (e.g. texttt{time.current}), (texttt{device.cu}).
18. If file output interval is reached:
    - Optional write of data to output binary (verb"<simulation_ID>.output#..bin"), (texttt{file_io.cpp}).
    - Update of verb"<simulation_ID>.status#..bin" (texttt{device.cu}).
19. Return to point ref{loopstart}, unless texttt{time.current >= time.total}, in which case the program continues to point ref{loopend}.

1. label{loopend}Liberation of device memory (texttt{device.cu}).
2. Control returned to texttt{main(ldots)}, liberation of host memory (texttt{main.cpp}).
3. End of program, return status equal to zero (0) if no problems where encountered.

The length of the computational time steps (texttt{time.dt}) is calculated via equation ref{eq:dt}, where length of the time intervals is defined by:

$$\Delta t = 0.075 \min\left(m / \max(k_n, k_t)\right)$$

where m is the particle mass, and k are the elastic stiffnesses. The time step is set by this relationship in `initTemporal()`. This equation ensures that the elastic wave (traveling at the speed of sound) is resolved a number of times while traveling through the smallest particle.

subsubsection{Host and device memory types} label{subsubsec:memorytypes} A full, listed description of the *sphere* source code variables can be found in appendix ref{apx:SourceCodeVariables}, page pageref{apx:SourceCodeVariables}. There are three types of memory types employed in the *sphere* source code, with different characteristics and physical placement in the system (figure ref{fig:memory}).

The floating point precision operating internally in *sphere* is defined in texttt{datatypes.h}, and can be either single (texttt{float}), or double (texttt{double}). Depending on the GPU, the calculations are performed about double as fast in single precision, in relation to double precision. In dense granular configuraions, the double precision however results in greatly improved numerical stability, and is thus set as the default floating point precision. The floating point precision is stored as the type definitions texttt{Float}, texttt{Float3} and texttt{Float4}. The floating point values in the in- and output datafiles are emph{always} written in double precision, and, if necessary, automatically converted by *sphere*.

Three-dimensional variables (e.g. spatial vectors in *E^3*) are in global memory stored as texttt{Float4} arrays, since these read and writes can be coalesced, while e.g. texttt{float3}'s cannot. This alone yields a *sim'20*times' performance boost, even though it involves 25% more (unused) data.

paragraph{Host memory} is the main random-access computer memory (RAM), i.e. read and write memory accessible by CPU processes, but inaccessible by CUDA kernels executed on the device.

paragraph{Device memory} is the main, global device memory. It resides off-chip on the GPU, often in the form of 1–6 GB DRAM. The read/write access from the CUDA kernels is relatively slow. The arrays residing in (global) device memory are prefixed by `dev_` in the source code.

marginpar{Todo: Expand section on device memory types}

paragraph{Constant memory} values cannot be changed after they are set, and are used for scalars or small vectors. Values are set in the `transferToConstantMemory(...)` function, called in the beginning of texttt{gpuMain(ldots)} in texttt{device.cu}. Constant memory variables have a global scope, and are prefixed by `devC_` in the source code.

%subsection{The main loop} %label{subsec:mainloop} %The *sphere* software calculates particle movement and rotation based on the forces applied to it, by application of Newton's law of motion (Newton's second law with constant particle mass: *F_{mathrm{net}} = m cdot a_{mathrm{cm}}*). This is done in a series of algorithmic steps, see list on

page pageref{loopstart}. The steps are explained in the following sections with reference to the *sphere*-source file; texttt{sphere.cu}. The intent with this document is emph{not} to give a full theoretical background of the methods, but rather how the software performs the calculations.

subsection{Performance} marginpar{Todo: insert graph of performance vs. np and performance vs. *Delta t*}. subsubsection{Particles and computational time}

subsection{Compilation} label{subsec:compilation} An important note is that the texttt{C} examples of the NVIDIA CUDA SDK should be compiled before *sphere*. Consult the *Getting started guide*, supplied by Nvidia for details on this step.

sphere is supplied with several Makefiles, which automate the compilation process. To compile all components, open a shell, go to the texttt{src/} subfolder and type texttt{make}. The GNU Make will return the parameters passed to the individual CUDA and GNU compilers (texttt{nvcc} and texttt{gcc}). The resulting binary file (texttt{sphere}) is placed in the *sphere* root folder. `src/Makefile` will also compile the raytracer.

4.2 C++ reference

class **DEM**

Public Functions

DEM(std::string inputbin, const int verbosity = 1, const int checkVals = 1, const int dry = 0, const int initCuda = 1, const int transferConstMem = 1, const int darcyflow = 0)

~DEM(void)

void **readbin**(const char * target)

void **writebin**(const char * target)

void **checkValues**(void)

void **reportValues**(void)

void **startTime**(void)

void **render**(const int method = 1, const float maxval = 1.0e3f, const float lower_cutoff = 0.0f, const float focalLength = 1.0f, const unsigned int img_width = 800, const unsigned int img_height = 800)

void **writePPM**(const char * target)

void **porosity**(const int z_slices = 10)

Float3 **minPos**(void)

Float3 **maxPos**(void)

void **findOverlaps**(std::vector< std::vector< unsigned int > > & ij, std::vector< Float > & delta_n_ij)

void **forcechains**(const std::string format = "interactive", const int threedim = 1, const double lower_cutoff = 0.0, const double upper_cutoff = 1.0e9)

void **printNSarray**(FILE * stream, Float * arr)

void **printNSarray**(FILE * stream, Float * arr, std::string desc)

void **printNSarray**(FILE * stream, Float3 * arr)

void **printNSarray**(FILE * stream, Float3 * arr, std::string desc)

void **writeNSarray**(Float * array, const char * filename)

void **writeNSarray**(Float3 * array, const char * filename)

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *search*

PYTHON MODULE INDEX

S

INDEX

W

Z