Lagrangian particle tracking in OpenFOAM with the assistance of the GPU.

MSE Vertiefungsmodul 2, HS 2009, SS 2010

University of Applied Sciences Lucerne

Nils Brünggel
Advisor: Josef Bürgler

August 27, 2010

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Ort, Datum Autor

Contents

Summary

The aim of this semester project was to understand how Lagrangian particle tracking (LPT) works, by looking at the implementation available in OpenFOAM. OpenFOAM supports polyhedral mesh handling which is required for Computational Fluid Dynamics (CFD) simulations including complex geometries. How the mesh is represented in OpenFOAM, what libraries are available and how one can search in the mesh is described in section 3.1. Having an unstructured polyhedral mesh comes also at a cost: In a regular, structured mesh it's trivial to figure out in which cell a particle is, given it's position. When support for LPT was added to OpenFOAM a robust and fast particle tracking algorithm had to be developed first. The algorithm is described in section 4. In order to test and understand the particle tracking algorithm a test case, consisting of a simple tunnel of rectangular cells, a static velocity field and random particles, which are moved by the velocity field, was created. (see section 5.1).

The second part of this work is about using graphic processing units (GPU) for general purpose computing and to understand how they can be used to speed up LPT in OpenFOAM. One of the key examples to understand the difference between a CPU and a GPU from a programmers perspective is the simple example of a dense matrix multiplication, which is presented in section 6.4. The most common problem is that the memory latency is much higher then the speed of the cores. While the architecture of a GPU is designed to deal efficiently with this limitation, the programmer has to take considerable care when arranging the data, otherwise the program will not perform optimally. In most of the studied papers and examples memory latency was a limiting factor for achieving high utilization and therefore high performance. In some examples it was possible to work around the high memory latency by properly arranging the data and therefore achieve excellent performance. One such example is the N-Body simulation. A simulation of a system of bodies which are all attracted to each other by gravity. "It is difficult to imagine a real-world algorithm that is better suited to execution on the G80[1] architecture than the all-pairs N-body algorithm.", stated one of the authors of the N-Body simulation for Nvidia GPUs [20].

However it turned out, that other problems are more difficult to solve efficiently on a GPU. CFD simulations using an unstructured mesh are challenging to implement since it's not always predictable which parts of the memory is accessed in advance, while this is a-priori known in a simulation using a structured grid. It's also crucial here to find a data structure which allows efficient memory access. The ideas developed to implement LPT on a GPU are presented in section 7.

[1] G80 is the architecture of the GPU used.

1 Introduction

1.1 Eulerian and Lagrangian specification of the flow field

The description presented here follows the second chapter from Stephen B. Pope - Turbulent Flows [22]. There are two ways of looking at the flow field. The continuum velocity field $\mathbf{U}(\mathbf{x}, t)$ is an Eulerian field, which is indexed by the position $\mathbf{x}$ and time $\mathbf{t}$. This can be visualized as sitting besides a river and watching the water passing a fixed location. The alternative Lagrangian description [1] defines a *fluid particle*, which is also a continuum concept. A fluid particle is a point that moves with the local fluid velocity: $\mathbf{X}^+(t, \mathbf{Y})$ is the position at time t of the fluid particle that is located at $\mathbf{Y}$ at the reference time t_0. Figure 1 shows the trajectory $\mathbf{X}^+(t, \mathbf{Y})$ of a fluid particle in x-t space. Again this can be visualized as sitting in a boat and drifting down a river.

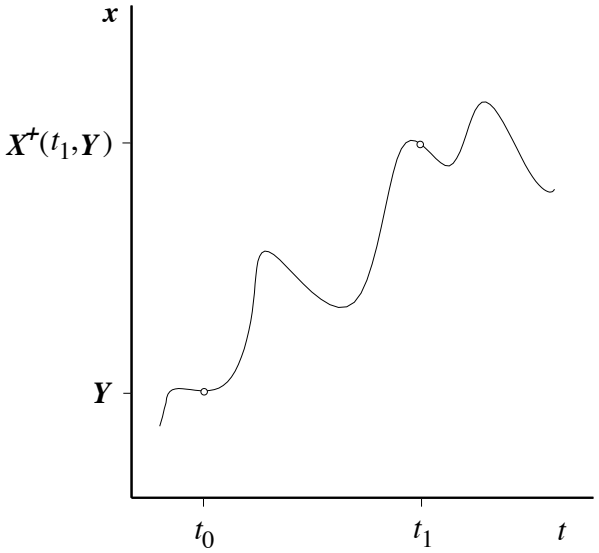

Figure 1: The trajectory of a fluid particle in x-t space.

There are two equations which define $\mathbf{X}^+(t, \mathbf{Y})$. First the position at reference time t_0

$$\mathbf{X}^+(t_0, \mathbf{Y}) = \mathbf{Y} \tag{1}$$

Second, the equation

$$\frac{\partial}{\partial t}\mathbf{X}^+(t, \mathbf{Y}) = \mathbf{U}(\mathbf{X}^+(t, \mathbf{Y}), t) \tag{2}$$

expresses the fact that the fluid particle moves with the local fluid velocity. We can now define *Lagrangian fields* using their *Eulerian* counterparts, for example

$$\mathbf{U}^+(t, \mathbf{Y}) \equiv \mathbf{U}(\mathbf{X}^+(t, \mathbf{Y}), t) \tag{3}$$

Lagrangian fields are not indexed by the current position of the fluid particle, but by their position at reference time t_0.

In a fluid simulation the domain is decomposed into cells using a mesh. The mesh is usually non-uniform, which means that cells vary in size. For example when meshing an airfoil, we require more precision (and therefore more cells) near the boundary layer of the airfoil.

One can further distinguish structured and unstructured meshes. Structured meshes have a structure which implicitly defines the arrangement of the data. Structured meshes have a basic rectangular matrix structure that makes storage and use easy as integer offsets can be used to access individual data points. Data points are arranged into rectangular or cubic structures by simply connecting them to their neighboring cells. In unstructured grids the connectivity between points must be explicitly defined for every set of points. This makes the definition of unstructured meshes more difficult and complex and the

relationships between neighboring cells or edges is no longer automatic and must be defined manually. (See [23])

Modern CFD software however is expected to support unstructured arbitrary polyhedral meshes, since realistic simulation often require complex 3D geometries generated from CAD models. How an unstructured mesh is represented in OpenFoam (OF) is explained later in section 3.1.

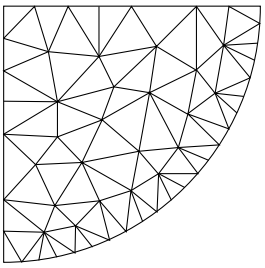

Figure 2: An unstructured mesh, taken from [2]

.

Now back to the Eulerian and Lagrangian specification of the flow filed. The mesh plays an important role, if the Eulerian and the Lagrangian field are coupled: The Eulerian equations are solved in every cell, the resulting fields (velocity, pressure, etc) are then attached to the concerning cells for every timestep. Lagrangian particles on the other hand do not need a mesh themselves, but when a coupling between the Lagrangian and the Eulerian phases is used, it is required to know in which cell a particle resides so that the coupling terms can be appended to the Eulerian phase. *More specifically we need to know for every time step through which cells the particle passes and how much time it spends in each cell.*

For some simulations it makes sense to also introduce a Lagrangian phase and couple it with the Eulerian phase. In [16], for example, Lagrangian particles are used to represent fuel droplets in a Diesel engine, while the gas is represented in the Eulerian phase using a compressible solver.

When running a large CFD simulation most of the computing time is spent solving the Eulerian equations for the Eulerian phase, while the performance impact of other parts of the code is often neglectable. If Lagrangian particles are also used then the computing time for the particle-tracking algorithms has a major impact to the overall computing time. It is therefore interesting to look a the code of the particle tracking algorithm and look for ways to speed it up.

1.2 OpenFOAM

OpenFOAM (OF) is an open source toolbox mainly for Computational Fluid Dynamics (CFD). FOAM stands for Field Operation And Manipulation. Unlike many commercial products, OF does not come with a user friendly graphical user interface; for every problem to be solve the user creates a case directory which contains the data required for the solver in the form of text files. The user has to learn the OF syntax in order to set the required properties. OF also comes with tools to create meshes from predefined geometries or import them. Once the mesh is created and the boundary settings are set, the computation is done using a solver. OF comes with a variety of different solvers, such as

- `laplacianFoam` Solves the Laplace equation.
- `icoFoam` Solver for incompressible, isothermal of newtonian fluids.
- `simpleFoam` Solver for incompressible turbulent flow.
- `sonicFoam` Compressible solver for trans-/supersonic laminar or turbulent flow.
- `dieselFoam` Solver for spray and combustion. Includes coupling of an Eulerian and Lagrangian phase.
- A complete list of available solvers can be found here:
  `http://www.opencfd.co.uk/openfoam/standardSolvers.html#standardSolvers`

An interesting aspect about OF is that it is easily possible to modify existing solvers or create a custom solver. OF is entirely written in C++ using a complete object-oriented design, which uses language features such as templates, virtual functions and operator overloading. Operator overloading makes it possible that the code can directly represent partial differential equations.

For post-processing OF relies on third party applications, such as ParaView[2]. The ideas and concepts behind OF are further described in [25]. Support for Lagrangian particles was added in version 1.3. Since then various solvers taking advantage of the Lagrangian framework have been developed.

[2] `http://paraview.org`

2 Related Work

2.1 Lagrangian Particle Tracking in OpenFOAM

There are two main contributions using OpenFOAM and Lagrangian particle tracking. The first is a PHD thesis written by Niklas Nordin in the year 2000 [16] about Turbulent Diesel Spray Combustion. The second is a paper [15] about the improved particle tracking algorithm, which was introduced later and is used today.

Nordin's thesis is about simulating a Diesel engine. In a Diesel engine the fuel is injected through a spray which has a diameter on the order of 0.1 mm, with a velocity of 200-400 m/s. The subsequent ignition and the combustion require length scales, which are even smaller. Using the finite volume method to simulate everything would require a mesh with scales so small that it would require enormous amounts of memory and computing time which are not available today or any time soon in the future.

This makes it necessary to represent the spray in a Lagrangian way: Fuel droplets are represented by points, referred as particles or parcels. The physical properties, such as location, velocity, diameter, mass, temperature and fuel composition are attached to these points. The Lagrangian (liquid) and the Eulerian (gaseous) phase are then coupled in two ways: The fluid particles are moved by the gaseous phase while the Eulerian phase needs to know in which cells the fluid/gas exchange takes place so that the interaction terms can be placed in the correct position. In other words it is required for each time step and each particles to know which cells were occupied and how much time the particle spent there.

In large (unstructured) meshes, which are used in simulations as the one described above, figuring out which cell is occupied by a particle becomes a computationally expensive task. Therefore an efficient particle tracking algorithm is required, which is explained later in Nordin's thesis. The thesis further describes a breakup model and a collision model.

The algorithm explained in Nordin's thesis however suffered from problems when large meshes with non-planar faces are used. OpenFOAM allows non-planar faces, defined by more then 3 vertices. If a particle is close to a vertex of a non-planar face there is the possibility of loosing it. This has been observed by the authors in large simulations and is documented in [15]. A modified version of the particle tracking algorithm which does not have this problems is explained later in the paper. This is the algorithm used in OpenFOAM 1.4.1 and further versions[3]. It is further described how the algorithm deals with moving meshes and how the parallelization works. Because this algorithm is the basis for the work carried out here it is explained shortly in section 4.

2.2 GPU Computing

2.2.1 Accelerating Matrix Vector operations in OpenFOAM

Wiedemann [26] investigated the possibility to move matrix and vector operations used in OpenFOAM to the GPU. The first problem mentioned is that current GPUs do not support double precision floating point (IEEE 754 double) operations and many calculations do not converge when just using single precision. Therefore he used a method, which emulates double precision by using two single precision values. The Preconditioned Conjugate Gradient Square (PCGS) method was implemented using CUDA and tested independently using the "Matrix Market"[4] provided by the National Institute of Standards and Technology (NIST).

After testing the PCGS solver, it was integrated into OF and tested using the `engineFoam` case. Comparing the time used to run the case on the CPU and using the PCGS solver on the GPU showed that it was twice as slow on the GPU. This had several reasons:

1. Since double precision must be emulated on the GPU all values had to be converted into an emulated double using two float values.

2. The matrix format was not identical, the matrix had to be converted, before sending to the GPU and converted back to the original format when retrieving it.

[3] Up to version 1.6, which is the current version at the time writing.

[4] The matrix market provides test data to test algorithms for numerical linear algebra. - `http://math.nist.gov/MatrixMarket/`

3. There was no parallelized preconditioner available, therefore the preconditioning had to be done on the CPU. This required two additional transfers of the matrix between the GPU and the CPU (while for each transfer applied the conversion explained above)

Using Nvidia's CUDA Visual Profiler, the author could show that only a small fraction of the time was actually spent on the GPU. Most of the time was spent transferring (and converting) the data and preconditioning on the CPU and concluding that a speedup will be possible, once the preconditioning can be done on the GPU. Further it is desirable to have a GPU, which supports double precision.[5]

Döffinger [9] wrote a similar thesis using the work of Wiedemann. Several ideas to write a parallel preconditioner are presented. The author manages to achieve a speedup by simplifying the algorithm, so that fewer data transfers between the CPU and the GPU are required.

2.2.2 Fast N-Body Simulation with CUDA

A fast implementation of an n-body simulation where every body has a mass and is attracted to all other bodies by gravity is explained in [20].

The force on body i, caused by body j is described by the following equation.

$$\mathbf{f_{ij}} = G\frac{m_i m_j}{||\mathbf{r_{ij}}||^2} \cdot \frac{\mathbf{r_{ij}}}{||\mathbf{r_{ij}}||} \tag{4}$$

Note that $\mathbf{r_{ij}} = x_j - x_i$ is the vector from body i to body j. The equation consist of the magnitude of the force, the product of the mass of both bodies over the square of the distance and the direction of the force.

Form this the total force F_i on body i is given by.

$$\mathbf{F_i} = \sum_{1 \leq j \leq N} \mathbf{f_{ij}} = Gm_i \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r_{ij}}}{||\mathbf{r_{ij}}||^3} \qquad j \neq i \tag{5}$$

The force between the bodies grows without bounds when they approach each other. In order to avoid this a softening factor $\epsilon^2 > 0$ is added. To integrate over time the acceleration $\mathbf{a_i} = \mathbf{F_i}/m_i$ is calculated. This leads us to the final computation.

$$\mathbf{a_i} \approx G \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r_{ij}}}{(||\mathbf{r_{ij}}||^2 + \epsilon^2)^{2/3}} \tag{6}$$

It is not required to presume $j \neq i$ anymore, because of ϵ the denominator does not become zero.

The idea behind the CUDA implementation is to calculate $\mathbf{f_{ij}}$ in an NxN grid of all pair-wise forces. The total force $\mathbf{F_i}$ (or acceleration a_i) on body can be calculated by summing over all entries in row i. Since each entry could be calculated independently there would be $O(n)$ available parallelism. However the problem is the same as with the dense matrix multiplication (see 6.4), each thread would require to access the global memory constantly, so it would spend much more time waiting for data then it would for the actual computation.

Therefore a strategy is required, which lets a thread reuse data several times. The idea which helps to balance parallelism and data reuse is the same as in the matrix multiplication example: The grid is subdivided into computational tiles[6], square regions of the grid of pair-wise forces consisting of p rows and p columns, so that the body descriptions of a computational tile fits into the much faster shared memory. The grid is then evaluated in a sequential and a parallel way: Each threads fetches one body into shared memory, after this all threads synchronize and update the acceleration for each body. When this is done, threads are synchronized, the new data is fetched into shared memory and the computation starts again.

The CUDA kernel calculates the acceleration of p bodies caused by N bodies in a system, to do this it has to reload data N/p times. The grid is one dimensional of size N/p, this results in N threads that perform N force calculations each for a total of N^2 interactions.

[5] NVIDIA [18] already promised that their next generation graphic cards will support double precision float. Other manufacturers made similar announcements.

[6] Computational tiles are mapped to thread blocks, all threads in a thread block have access to the same shared memory.

The paper concludes that the N-Body problem is one of the best suiting real world algorithms for GPU computing, because it runs more then 50 times faster as a highly tuned serial computation. The three features which helped to gain such high efficiency are summarized (direct quote):

- Straightforward parallelism with sequential memory access patterns.
- Data reuse that keeps the arithmetic units busy.
- Fully pipelined arithmetic, including complex operations such as inverse square root, that are much faster clock-for-clock on a GeForce 8800 GTX GPU than on a CPU.

2.2.3 Air pollution modeling using a Graphics Processing Unit with CUDA

In this paper [21] the following simulation scenario is discussed: A chemical active or passive species disperses from a single point source. It is simulated using a Lagrangian model with the species as particles. Weather data and interaction with isotopes are taken into account. The particles are moved by advection (wind) and turbulent diffusion. A particle can transform to other species, for example by radioactive decay. Particles can be removed from the atmosphere via wet and dry deposition processes on the surface or if they leave the simulation domain. There is no particle to particle interaction. The position and the state of the particles are stored in a grid which can be used later for visualization.

A sequential algorithm for this particle simulation is straightforward:

Algorithm 1. *Sequential air pollution simulation*

for *each time step* ***do***
Sample the weather data and calculate the turbulent diffusion coefficient.
Generate random numbers.
Move all particles: $X_i^{new} = X_i^{old} + v_i^{adv} \triangle t + \tilde{x_i}$
Calculate the stochastic deposition and remove deposited particles.
Calculate particle activities, such as radioactive decay.
Sum the activities and masses of the particles on a regular grid.
end for

Algorithm 1 gives an overview of what needs to be done in one time step of the simulation. In the equation to move the particles X_i denotes the spatial coordinates of the individual particle, v_i^{adv} is the ith coordinate of the wind velocity vector and $\tilde{x_i}$ stands for the effect of the statistic turbulent process. The random numbers generated before are required to calculate $\tilde{x_i}$.

To speed up the simulation a parallelized version was developed, which runs on a GPU using CUDA. The algorithm for the GPU version is more complicated since it needs to take restrictions on graphics hardware into account.

The main loop for the simulation runs on the CPU. The particle information and the random numbers (read only) are stored in the global memory. The weather data is stored in texture memory. Shared memory is used to cache the particle data (position, state information). There are two kernels involved in the calculation of one time step: The first kernel generates random numbers using the Mersenne-Twister random number generator provided by the CUDA SDK. The second kernel does the main calculation: First the particle information is loaded from global memory into shared memory. Then the weather is sampled from the texture memory using the built-in trilinear filter. From this the turbulent diffusion coefficient is calculated and stored in shared memory. The particle is then moved by the wind field and the turbulent diffusion, it is also tested for deposition. The final step is writing the changed particle information back to global memory. Summing and storing the particle data on the grid is done on the CPU in the same way as the CPU version. This is outlined in algorithm 2.

Algorithm 2. *GPU air pollution simulation*

Main program on the CPU.
Allocate global memory on the device.
Upload weather and isotope data to texture and constant memory on the device.
for *each time step* ***do***

Emit new particles.
Generate random numbers. (on the GPU)
Start the kernel for every particle.
end for
Download particle data from device. Free memory.

Kernel on the i^{th} particle.
Load particle information from global memory.
Load random numbers from global memory.
Sample weather data from 3D texture.
Interpolate in z dimension.
Calculate diffusion coefficient.
Move particle by wind.
Move particle by turbulent diffusion.
Stochastic deposition (using isotope data from constant memory)
Store changed particle information in global memory.

The authors found a typical speedup[7] of 80-120 depending on the number of particles used. Memory access latency turned out to be the bottleneck of the simulation. Because there are only relatively few arithmetic operations to update a particle information, too much time is spent waiting for data. The highest possible occupancy (see 6.3) turned out to be only 33%, because of the high register requirements of the kernel.

2.2.4 Running unstructured grid-based CFD solvers on modern graphics hardware

While most works on the GPU use a structured mesh, this paper [7] demonstrates a FVM simulation running on an unstructured three-dimensional mesh which runs about ten times faster compared to a CPU implementation running on a quad-core CPU.

The mesh used however consists only of tetrahedral elements (OpenFOAM allows polyhedral meshes). The solver works on a per element basis, with a separate thread for each element. In order to avoid uncoalesced (see 6.2.3) memory access, the authors choose a numbering scheme which allows the storage of neighbouring elements nearby in memory. Elements nearby in space must also be stored nearby in memory. The numbering scheme used is called "bin numbering scheme" and works by overlaying a grid of bins. The points in the mesh are assigned to a bin, the points are then renumbered by assigning numbers while traversing the bins in fixed order. The connectivity of each element is then sorted so that the indices of the four neighbours of each tetrahedral elements are in increasing order. The authors point out that any other scheme which ensures that geometrical closedness matches closedness in memory could have been used as well.

In order to minimize the possibility of divergent branches (see [19] - chapter 5.4.2) the numbering scheme was modified, so that boundary elements are stored consecutively in memory, this limits the number of divergent warps. The authors also looked at the possibility of using shared memory to speed up memory access (see 6.2.3). Shared memory can be used to share data between different threads. Because the memory access pattern of unstructured grid-based solvers is data dependent and can not be known in advance, shared memory could not be used.

[7] Compared to a sequential CPU implementation.

3 Basics

3.1 Mesh Representation in OpenFOAM

OpenFOAM (OF) supports unstructured polyhedral meshes with any number of faces. The mesh is described in the official Users Guide [4] (chapter 5), the Programmers Guide [3] (chapter 2.3) as well as in Jasak's PHD thesis [13] (chapter 3.2). Here I will recall a short summary.

A mesh is defined by **points**, these are the vertices of the mesh. Points are defined using the `vectorField` class, which is renamed to `pointField`. **Faces** are defined as a list of points, with each face having at least three points. The `List<face>` (or renamed `faceList`) holds all the faces in a mesh. **Cells** are defined in the class `List<cell>` (or `cellList`) where the `cell` class holds the face numbers. Further patches can be defined in order to define boundary values. The mesh is stored in the following files[8], which are generated by the OF meshing utilities such as `blockMesh`.

- `points` Contains a list of coordinates of vertices. OF implicitly indexes the points starting from 0 up to the number of points minus one.
- `faces` Contains a list of faces. The point indexes defined in the `points` file are used to reference the vertices. It is distinguished between internal faces and boundary faces: Internal faces are between two cells and have always an owner and a neighbour cell assigned, while boundary faces lie on the boundary of the computational domain and have just an owner cell assigned (see below). Further a face normal is defined, so that it points outside of the owner cell. In case the face is a boundary face, the normal points outside of the computational domain.

  The order of the vertices of a face is defined so that if the face vector points towards the viewer the vertices are arranged in an anti-clockwise path. The faces are also implicitly indexed, starting with 0 for the first face.
- `owner` Each face is assigned an owner cell, in case of an internal face this is the cell with the lower label of the two cells adjacent to the face. The face normal can also be defined as pointing from the cell with the lower label to the cell with the higher label.
- `neighbour` Internal faces also have a neighbour cell. Since an internal face is always between two cells, the cell not being the owner is the neighbour cell. (Which is the cell with the higher label.) Faces lying on the boundary have no neighbour cell and are omitted in the neighbour list. Since some faces always lie on the boundary, this list has less items then the owner list.
- `boundary` The boundary file contains a list of patches, to which cells are assigned.

The cells in a well defined mesh must have certain properties, which are listed below[9]. OF provides the `checkMesh` utility which checks whether the mesh respects this constraints.

- **Contiguous** The cells must completely cover the computational domain and must not overlap one another.
- **Convex** Every cell must be convex and therfore its cell centre inside the cell.
- **Closed** Every cell must be closed, both geometrically and topologically where:
  - Geometrical closedness requires that when all face area vectors are oriented to point outwards of the cell, their sum should equal the zero vector to machine accuracy.
  - Topological closedness requires that all the edges in a cell are used by exactly two faces of the cell in question.
- **Orthogonality** For all internal faces in a mesh, a centre-to-centre vector is defined. This is done for each pair of cells, which lie next to each other. The centre-to-centre vector points from the cell with the smaller label to the cell with the larger label. The orthogonality constraint requires that for each internal face, the angle between the face normal (which starts at the face and also points inside of the cell with the larger label, see above) and the centre-to-centre vector must always be less than 90°.

[8] Usually located in `constant/polyMesh` in an OF case
[9] Taken from the Users Guide [4], chapter 5.1.1.3

3.2 Mesh classes

Figure 3: Simplified inheritance diagram of the mesh classes.

The mesh in Open FOAM consists of three base classes[10]. The `primitiveMesh` class provides the basic functionality of the mesh: It holds the points, faces, cells and edges of a mesh together and provides a basic mesh analysis engine. The polyhedral mesh is represented by `polyMesh`, it is a generic class and does not presuppose any particular form of discretisation. The data and functions related to the geometry needed for the finite-volume discretisation is implemented in `fvMesh`.

3.3 Shapes in a mesh

Basic geometrical objects, such as lines, planes, points, triangles, etc are called `primitiveShapes` and are defined in `$FOAM_SRC/OpenFOAM/meshes/primitiveShapes`. For example the line is defined by two reference points and comes with basic geometrical operations, such as finding the distance between a line and a point or a line and another line. These primitive shapes are then used in more complex shapes, the `meshShapes`. Classes in the `meshShapes` directory do not inherit from the `primitiveShapes` classes, instead they use them. Mesh shapes are used to construct the mesh, the most important mesh shapes are cells and faces.

3.4 Interpolation

When the velocity of a Lagrangian particle is calculated from the Eulerian phase it must be interpolated from the computational point (that is the centroid of the cell in which the particle resides) to it is actual position.

The method `interpolate` in the class `interpolationCellPointFace`[11] is used for this. It takes a given point and the cell in which the point resides as input and returns the interpolated field at this point. The function is quite complex and distinguishes between points which are inside a cell and points which are on a face. Depending on the point location and whether there is a boundary patch which is not empty, either the interpolated value on the nearest face or the values at the computational point of the cell are returned. The interpolation of face values is described in [13], chapter 3.3.1.2.

There is also an interpolation class used for post-processing, called `interpolate` (do **not** confuse with the class `interpolation`) this class is used to interpolate the volume fields to surface fields which are displayed in a visualizer like ParaView.

3.5 Mapping positions to cells

OpenFOAM comes with a collection of algorithms to search in an unstructured, arbitrary polyhedral mesh. There are two major ways to figure out in which cell a point lies. The first one uses the `primitiveMesh` class and is quite easy to understand but not so efficient. The second way uses an `octree` and is more efficient.

3.5.1 Using primitiveMesh

The `primitiveMesh` class provides `findCell(point& location)`[12], which returns the label of the cell containing the point. This method reduces the problem to the nearest neighbour problem[13].

[10] There are more specialized mesh classes which are also derived from `fvMesh` or `polyMesh`, the complete inheritance diagram can be found in the source code documentation: `http://foam.sourceforge.net/doc/Doxygen/html/classFoam_1_1fvMesh.html`

[11] The class can be found in `$FOAM_SRC/finiteVolume/interpolation/interpolation/interpolationCellPointFace`

[12] Implemented in `$FOAM_SRC/OpenFOAM/meshes/primitiveMesh/primitiveMeshFindCell.C`

[13] The nearest neighbour problem is a well studied problem, since it has various applications in Databases, Coding Theory, Computer Vision, etc just to mention a few.

Problem 1. ***Nearest neighbour search***

Given a set S of points and a query point q find the closest point in S to q.

The cell centres are taken as the point set and once the nearest point has been found it is checked whether it lies in the same cell as the query point. If this is the case, the cell label is returned. Otherwise the cell containing q must be somewhere in the neighbourhood of the cell containing the closest point to q.

The implementation in the `primitiveMesh` class is inefficient, since it is just iterated over all cell centres to find the closest point. And if q and the point closest to q is are not in the same cell, it is iterated again over to whole mesh to determine in which cell the query point lies. The algorithm to determine whether a given point lies in a cell is more interesting. The `pointInCell(const point& p, label celli)` method also in the `primitiveMesh` class does this. The algorithm is outlined below:

Description 1. ***Decide if a given point is in a cell.***

Let $\mathbf{P}$ be the given point, $\mathbf{C_f}$ the face centre, $\mathbf{S_f}$ the face normal and $\mathbf{V} = \mathbf{P} - \mathbf{C_f}$ ($\mathbf{V}$ is the vector pointing from $\mathbf{C_f}$ to $\mathbf{P}$). From the mesh definition we know that the face normal points out of the owner cell. We can therefore take the sign of the inner product of $\mathbf{V}$ and $\mathbf{S_f}$ and determinate if $\mathbf{P}$ lies on the "inner side" of a face. Depending if the face normal points in or out of the cell, $\mathbf{P}$ lies inside if the inner product is positive or negative. If $\mathbf{P}$ lies on the "inner side" of all faces, then the point lies in the given cell.

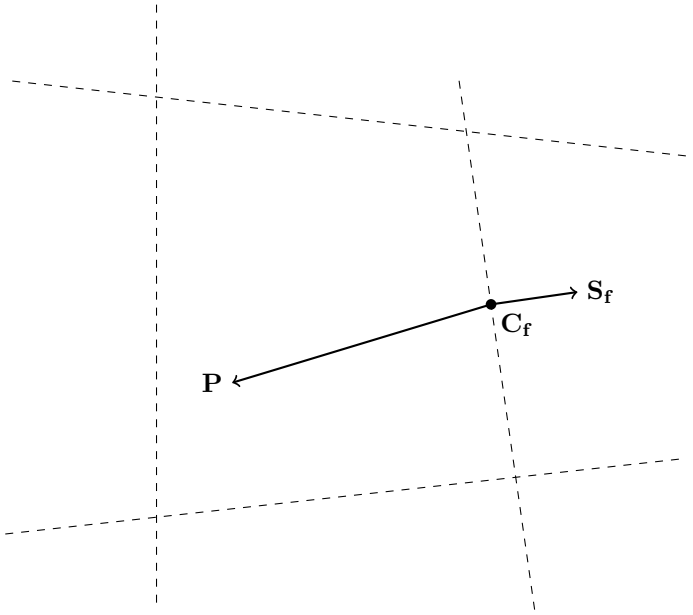

Figure 4: Example of a point $\mathbf{P}$ in a cell. The vectors from the face centre to $\mathbf{P}$ and to the face normal $\mathbf{S_f}$ are shown.

Figure 4 show an example of how it is determined whether a point $\mathbf{P}$ lies in a cell or not. In this case the cell shown is the owner of the face therefore the face normal points out of the cell and it is checked if the inner product of $\mathbf{v}$ and $\mathbf{S_f}$ is negative[14].

An alternative and more efficient way of mapping points to cells uses an octree and is explained below.

3.5.2 Oct- and Quadtrees

A quadtree can be used to divide and index 2D space in smaller shapes, while an octree is the equivalent for 3D space. Because it is easier to explain things in 2D, we start with the quadtree. In a quadtree each node has four childern, where a child can be either another node or a leaf. The children of each node are labeled NE, NW, SW and SE for the direction in which the shapes appended below are. Figure 5 shows how to divide a square into smaller squares using a quadtree.

[14]Recall that $cos(\alpha)$ with α in the interval $90° < \alpha < 270°$ is negative

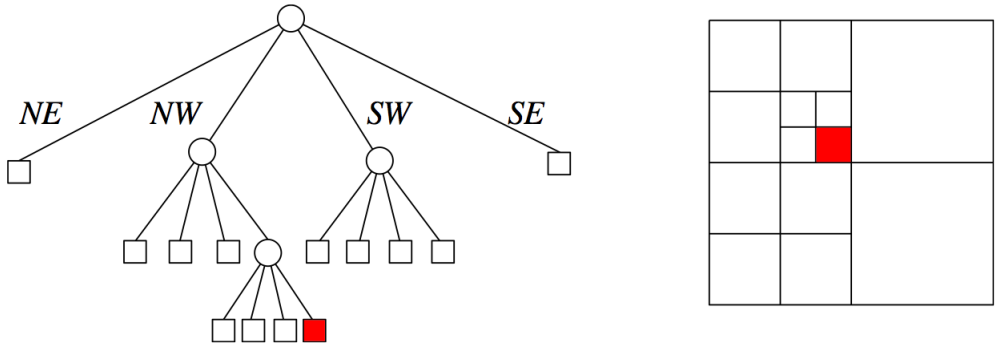

Figure 5: A quadtree is used to divide a square into smaller subdivisions, the red area is referenced by the red leaf. The figure was taken from [8]

The problem of finding the cell in which a particular point resides is again reduced to the nearest neighbour problem. A quadtree can be used to store a point set. In our case this would be the set of centres of all cells in a mesh. Starting at the middle of a bounding box, which surrounds the point set, the quadtree is constructed: The current rectangle is splitted into four quadrants. The points are assigned to each quadrant. Then, for each quadrant, a quadtree is constructed recursively until there is no more than one point in every rectangle at the leaves of the quadtree.

The constructed quadtree can now be used for neighbour finding: Given a query point the quadtree can be descended by deciding at the middle point of each quadrant weather it lies in the NE, NW, SW or SE direction and then choosing the concerning node until a leaf is found. Then it is known in which quadrant the query point resides, but not yet which point its nearest neighbour is. The nearest neighbour could also be in the adjacent quadrants. Therefore the distance to the points inside the quadrant at the leaf as well as the distance to the points in the surrounding quadrants must be calculated, before the nearest point can be finally identified. A small example, how a quadtree can be descended, is illustrated in figure 6.

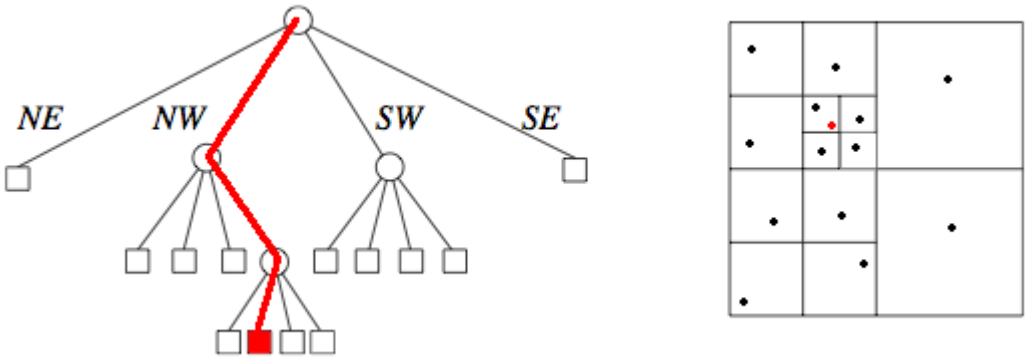

Figure 6: A quadtree used for neighbour finding. The red dot shows the query point. The path taken to descend the quadtree is shown in red.

For an extensive introduction into quadtrees, which includes an exact algorithmic description, see [8] (section 14.2 - "Quadtrees for point sets").

Octree implementation in OpenFOAM The octree implementation in OpenFOAM (OF) is generic and can be used to solve different geometrical problems, such as the nearest neighbour problem explained above. The documentation[15] specifically mentions the following use cases:

[15] See `http://foam.sourceforge.net/doc/Doxygen/html/classFoam_1_1octree.html`

1. Find the "nearest" shape to a point.
2. Find the shape which contains a point (only sensible for 3D shapes).
3. Find intersections of line with shapes.

The class `meshSearch` can be used to search the `polyMesh` using the `octree` class. OF also offers a utility, which takes a position and then returns the cell id, in which the point, defined by the position vector resides. The utility is located in `$FOAM_SRC/applications/test/findCell-octree`

The face-to-face tracking algorithm (see 4) and solvers like icoLagrangianFoam (ICL) (see 5.1) do not use the octree classes, it is not clear to the author how (and if) OF uses the octree classes. The idea of realizing a particle tracking algorithm on the basis of an octree, which holds the centroids of all cells has not been discussed in either [15] or [16]. It would be interesting to see if a fast particle tracking algorithm could be realized by using an octree. However since one needs to ensure that the particle is in the same cell as it is nearest neighbour centroid, or search in the surrounding cells in case it is not, it is doubtful that it will perform better then the face-to-face tracking algorithm.

4 The face-to-face tracking algorithm

As mentioned before, the complete particle tracking algorithms is explained in [15]. Since this algorithm is crucial for the optimization it is presented here.

4.1 Basic Particle Tracking Algorithm

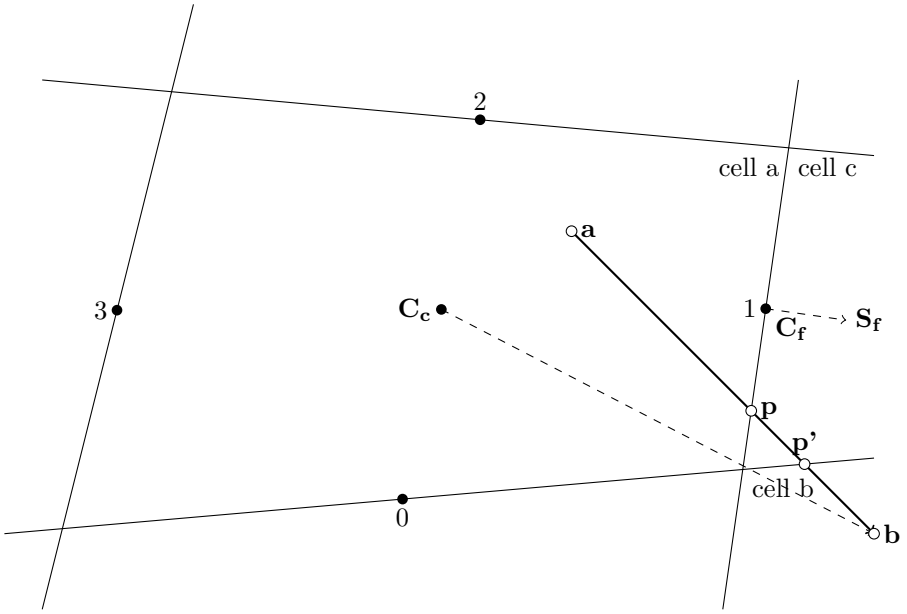

Figure 7: A particle starts at position **a** and moves to **b**, during the move it changes the cell twice at point **p** and **p'**. $\mathbf{C_c}$ is the cell centre and $\mathbf{C_f}$ the face centre.

For the Lagrangian-Eulerian coupling it is required to know, for every time step, which cells a particle crosses and how much time it spent there. Figure 7 illustrates the particle tracking algorithm during a time step. The particle is initially located at **a** and moves to **b**. While traveling along the straight line from **a** to **b** it changes the cell twice at **p** and $\mathbf{p}'$.

The point where the particle hits the face is calculated by the following equation:

$$\mathbf{p} = \mathbf{a} + \lambda_a(\mathbf{b} - \mathbf{a}) \tag{7}$$

λ_a denotes the fraction of the velocity vector which the particle travels until it hits the first face (the distance from **a** to **p**). In this equation λ_a and **p** are unknown, we only know the start and end position of the particle (**a** and **b**). The vector from $\mathbf{C_f}$ to **p** along the face is orthogonal to the face normal. This let's us define another equation, which can be used to calculate **p**:

$$(\mathbf{p} - \mathbf{C_f}) \cdot \mathbf{S} = 0 \tag{8}$$

In the equation above we used the fact that dot product of two orthogonal vectors is zero. Now substituting equation 7 into equation 8 and solving for λ_a gives:

$$\lambda_{a,f} = \frac{(\mathbf{C_f} - \mathbf{a}) \cdot \mathbf{S_f}}{(\mathbf{b} - \mathbf{a}) \cdot \mathbf{S_f}} \tag{9}$$

Now since **p** no longer appears in equation 9 we can calculate λ_a for each face respective of the cell. The face with the smallest $\lambda_a (0 \leq \lambda_a \leq 1)$ is the face, which is crossed by the particle. The particle is then moved to cell c and the particle's cell occupancy information is updated. The next tracking event works in the same way as the one just presented. This is repeated until the whole time step is over. If $\lambda_{a,f}$ is not in the interval $[0, 1]$, then the particle, i.e. **b** must be inside the same cell.

4.2 Modified Tracking algorithm

If a face is defined by more than three vertices, then these do not necessarily lie in a plane. In case a face is non-planar the mesh stores the face centroid and interpolates a normal vector which represents the effective plane of the face. When using the effective planes as cell faces, the cells in a mesh are no longer space-filling! It is therefore possible to loose track of a particle when it crosses a face close to a vertex. The deficiencies of the basic particle tracking algorithm are further described in [15] (page 267).

With reference to figure 7, instead of taking the starting point of a particle, the cell centre is taken to determine whether the particle is inside a cell or not and which faces it may crosses. This is because problems with non-planar faces occur if the particle is close to a vertex of the cell. Replacing **a** with $\mathbf{C_c}$ in equation 9 gives:

$$\lambda_{c,f} = \frac{(\mathbf{C_f} - \mathbf{C_c}) \cdot \mathbf{S_f}}{(\mathbf{b} - \mathbf{C_c}) \cdot \mathbf{S_f}} \tag{10}$$

The line from $\mathbf{C_c}$ to **b** (shown dashed in figure 7) crosses face 1 and 0[16]. Equation 10 therefore yields $\lambda_{c,f}$ between 0 and 1 for face 0 and face 1. If $\lambda_{c,f} < 0$ or $\lambda_{c,f} > 1$, then **b** must be in the same cell as **a**. Otherwise it is also necessary to calculate $\lambda_{a,f}$ using equation 9 for the faces, which are crossed by the line from $\mathbf{C_c}$ to **b** (face 0 and 1 in this case). The lowest value of $\lambda_{a,f}$ determines then which face was actually hit and the fraction of the time step, which it took to travel to the face can be determined using $\lambda_{a,f}$. The cell occupancy is changed to the neighbouring cell of the face which was crossed (c in this case). With $\lambda_m = min(1, max(0, \lambda_{a,f}))$ equation 11 is then used to move the particle to p.

$$\mathbf{p} = \mathbf{a} + \lambda_m(\mathbf{b} - \mathbf{a}) \tag{11}$$

The complete tracking algorithm is shown below as pseudo code. This was taken from [15] (algorithm 1).

Algorithm 3. *Complete tracking algorithm*

while *the particle has not yet reached its end position at* **b** ***do***
  find the set of faces, F_i for which $0 \leq \lambda_c \leq 1$
  if *size of $F_i = 0$* ***then***
    move the particle to the end position
  else
    find face $f \in F_i$ for which λ_a is smallest
    move the particle according to equation 11 using this value of λ_a.
    set particle cell occupancy to neighbouring cell of face f
  end if
end while

[16] Or more precisely the plane which is defined by the face normal of face 0.

4.3 Parallelization in OpenFOAM

Simulations in OpenFOAM (OF) can be parallelized using the Message Passing Interface (MPI). Various implementations such as OpenMPI or Local Area Multicomputer (LAM) are supported. These interfaces will work on a network or on a single computer with multiple cores. Parallelization happens on process level, which means that there is no shared memory used, each process has it is own memory assigned. When running a simulation in parallel with multiple processes the simulation domain must be decomposed and each process will compute a subdomain. The mesh should be decomposed in a way that each process is assigned about the same number of cells. The decomposed mesh contains processor patches which act as boundary: Everything which goes through a processor patch must be synchronized with the concerning process. Inside a process the whole solver is running sequentially. This is ideal to distribute a large simulation over multiple machines, which are connected by a (fast) network.

The downside of this approach is that, after each time step, the calculations of all subdomains must be synchronized. On older multicore systems this parallelization approach is useless, since the memory bandwidth is not doubled for two cores and every core just gets half the bandwidth and the performance does not increase significantly. On more recent dual core systems a significant speedup is observed, but it is still much lower then the theoretical factor two[17]. For a comparison of two recent systems running a preconditioned conjugate gradient solver sequentially as well as parallelized on two and four processes using MPI see [9] (chapter 5, table 5.1).

Currently OF does not support the parallelization of basic operations such as matrix- and vector operations on thread level with shared memory. This however is required in order to use massive-parallel hardware such as a GPU. It is the intent of this project to introduce finer granular parallelization with regard for Lagrangian particle tracking.

[17] If everything can be parallelized and there is no communication overhead.

5 Prototype and test cases

To test the particle tracking algorithm, icoLagrangianFoam (ICL) [11] was used. ICL was modified so that it does not solve an Eulerian equation and uses a given vector field as the Eulerian phase instead. The resulting solver is called `basicParticleFoam`.

For this prototype `freefoam-0.1.0rc4`[18] was used. This is essentially the same as OpenFOAM 1.5, but comes with a build system based on cmake, which makes it easier to compile. FreeFOAM was built on Mac OSX 10.6 using GCC-4.2.

5.1 icoLagrangianFoam

icoLagrangianFoam (ICL) is based on `icoFoam` a solver for incompressible flow. The original purpose of ICL was to be able to use particles outside of `dieselFoam`. From the description on [11] it's features are:

```
The particle code is a heavily lobotomized version of stuff found in the dieselSpray
classes. It features:

- a simple random injector
- a drag force model that is horrible and not very physical
- particles can bounce from walls or die (switchable)
- particles leave the system at in or outlet. All other boundary types are not treat
  correctly
- the particles can add a source term to the moment equation of the gas (switchable)
- there is no particle-particle interaction
- This solver is not to be used for simulations that resemble the real world.
  It's just a demo.
```

ICL uses a one way coupling between the Eulerian phase (which is the same as the `icoFoam` solver provides) and the Lagrangian phase: The flow affects the particles, but the particles do not affect the flow. In case of dilute suspension with a low volume fraction of particles, the particles effect on turbulence are negligible. It would be possible to write a physical accurate solver in this case using one-way coupling. For a complete description of ICL please see [11] and [24].

[18] Project Website - `http://freefoam.sf.net`

5.1.1 Configuration of a case for basicParticleFoam

ICL requires an additional configuration file to configure the particle cloud[19]. Below is a commented version of the file (header is omitted)

```
interpolationSchemes
{
  U cellPointFace;
}

g       (0 -9.81 0);      // Gravity on earth. Note that the coordinate system in
                          // paraview is arrange so that y points in the upwards
                          // direction.
density         0.1;      // Assigned to each parcel, is the particle density
                          // inside a particle.
drag            1e0;      // Drag coefficient
subCycles        20;
useMomentumSource 1;      // Should add the source term to the momentum equation
                          // of the gas, but turning it off resulted in very
                          // strange solutions (particles outside the
                          // simulation domain?!).

wall
{
  reflect     1;          // Particles bounce when they hit a wall
  elasticity  2;
}

injection
{
  thres   0; // A particle is injected in every timestep if a calc rand num in [0,1]
             // is smaller than a threshold. Here we don't want additional particles
             // injected and set it zero. The values below are therefore irrelevant
             // for this case.
  center  (0.005 0.01 0.005);
  r0             0.02;
  vel0           0.01;
  vel1     (0. 0. 0.);
  d0            0.001;
  d1            0.001;
  tStart         0.01;
  tEnd              1;
}
```

5.2 Particle Tunnel

A simple test case was created to see how particle tracking works. The geometry is defined in `constant/blockMesh/blockMeshDict/` and can be created using `blockMesh`. Below an excerpt from the `blockMeshDict` is shown. It starts with defining the vertices. OpenFOAM (OF) automatically numbers the vertices, starting at 0 to 7. A hexahedron is defined in the blocks section. The numbers in the first brackets reference the vertices, defined above (again 0 is the first vertex, 7 the last). The order, in which the vertices must be given is defined in [4], page 123. The last two entries on the same line tell OF to subdivide the geometry into 10 cells in x-direction. Figure 8 show the geometry of the tunnel.

[19] It can be found in `constant/cloudProperties` in the concerning case.

```
vertices (
    (0 0 0)
    (1 0 0)
    (1 0.1 0)
    (0 0.1 0)
    (0 0 0.1)
    (1 0 0.1)
    (1 0.1 0.1)
    (0 0.1 0.1)
);

blocks (
    hex (0 1 2 3 4 5 6 7) (10 1 1) simpleGrading (1 1 1)
);
```

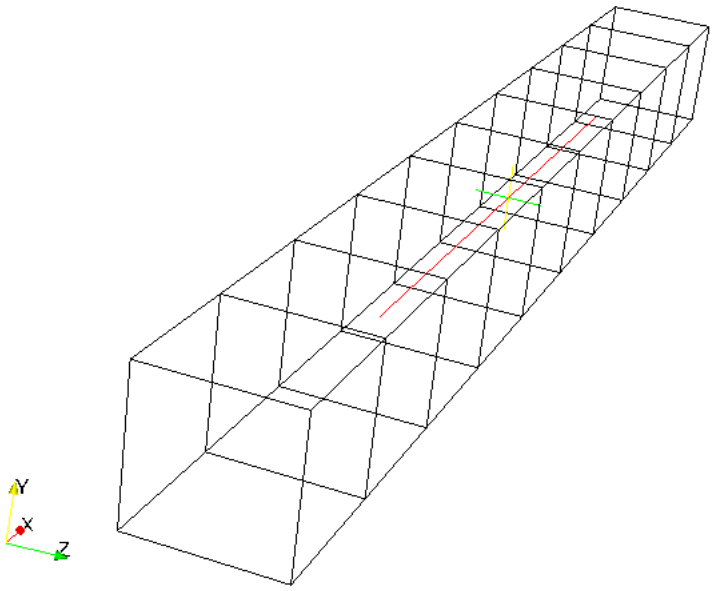

Figure 8: The geometry of the tunnel case.

There are two forces, which act on the particle inserted. First the gravity, defined in `const/polyMesh` and then the predefined vector field defined in `0/U` using the `internalField` keyword. An excerpt of `0/U` is shown below. `U` stands for velocity in a first test 10 cells with just an initial velocity of 10 in x-direction were used. The entry `(0.1 0 0)` is repeated ten times, but it's omitted here.

```
internalField   nonuniform List<vector>
10 (
(0.1 0 0)
);
```

As mentioned before the solver is based on `icoLagrangianFoam`, but the PISO loop was commented out. Further the file `HardBallParticle.C` was extended for debugging purposes. All particles are labeled with an id, so it's possible to find them later in the log, further a log entry is added every time a particle changes the cell. We call the new solver `basicParticleFoam`.

```
if(cell_before != cell()) {
Info << "Particle " << id << " changed from cell " << cell_before
```

```
  << " to " << cell() << "." << endl;
  cell_before = cell();
}
```

To see the log entry when running the solver it's necessary to add the concerning debug switch to `$FOAM/etc/controlDict.in`: `HardBallParticle 1;` After running the solver, a grep on the output log for "Particle 1" returns:

```
Particle 1(0.020053  0.00643112 0.000675545) changed from cell 1 to 2.
Particle 1(0.0300511 0.00584854 0.000675545) changed from cell 2 to 3.
Particle 1(0.0400492 0.00526595 0.000675545) changed from cell 3 to 4.
Particle 1(0.0500472 0.00468337 0.000675545) changed from cell 4 to 5.
Particle 1(0.0600453 0.00410078 0.000675545) changed from cell 5 to 6.
Particle 1(0.0700434 0.0035182  0.000675545) changed from cell 6 to 7.
Particle 1(0.0800414 0.00293561 0.000675545) changed from cell 7 to 8.
Particle 1(0.0900395 0.00235303 0.000675545) changed from cell 8 to 9.
```

This tells us that our second particle (indexing starts at 0) travelled from cell one through all cells until it arrives at the wall in cell 9. Also the z coordinate is constant, this is because we have just two forces acting on the particle: The vector field pointing in x direction and the gravity pointing in $-y$ direction.

The geometry in this example is very simple, but can be easily made more complicated. Every face is defined by four vertices we can easily tweak the numbers a bit, so that the faces of a cell do not lie in a plane anymore and we have non-planar faces.

5.2.1 Using a random initial particle cloud

In order to make the test case more interesting a random particle cloud is used: A number of particles is generated with a random initial position and a random initial velocity. By creating the following files in the `0/lagrangian/defaultCloud` directory the initial particle cloud is defined.

```
U            d            m            positions
```

The `positions` file contains the locations of the points, representing the particle center. The corresponding class is `incompressibleCloud`. `U` contains a `vectorField` which assigns every particle an initial velocity. The files `d` and `m` contain arrays of scalars, which represent the diameter and the mass of the particles.

A Python script was written, which generates the files in the initial clouds using random numbers in a defined range for position, velocity, mass and diameter. This script can be used to generate large number of particles which is required to create test cases for the tracking algorithm. Figure 9 shows a tunnel with random initial particle properties (position, velocity and mass), the colors of the particles represent their mass.

Shortcomings of icoLangrangianFoam

While experimenting with the `basicParticleFoam` solver some problems were observed.

- If the elasticity parameter is set too high (> 1) in `constant/cloudProperties` the velocity can grow too fast if a particle hits walls repeatedly. This leads to numbers larger then the range that the machine can represent (using IEEE 754 double), with the result that further calculations cannot proceed. The chance that this happens is very high when using 100 or more particles with random initial position and velocity. Since this can be confusing for the user ICL should warn when using an elasticity parameter higher than 1. From a physical point of view it makes no sense that the magnitude of the velocity of a particle is higher after it hits a wall than it was before.
- When the particle tunnel is running with gravity and many particles touch the ground[20] so that they "lie" on it the simulation becomes very slow, because these particle repeatedly hit the bottom

[20] More precisely the set of faces to which the gravity vector points.

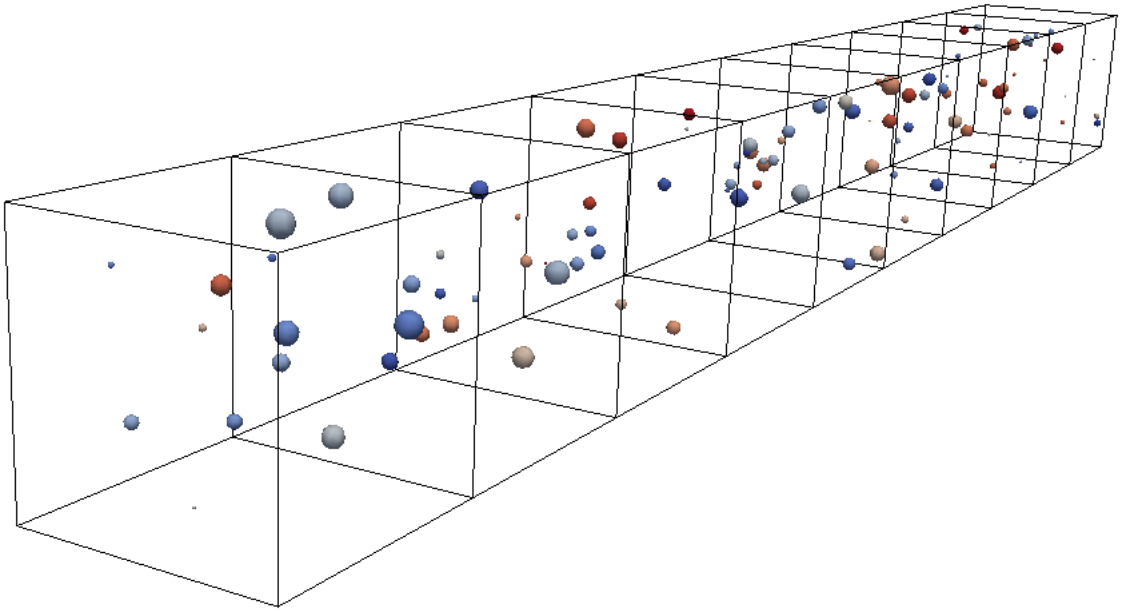

Figure 9: The tunnel with random particles.

wall. In the log the `wallsHit` count increases rapidly. It seems that ICL does not deal well with particles that "lie" on a wall (eg constantly touch it). A possible solution could be to turn of the reflection mechanism and assume no movement in the normal direction to the wall once a particle was very close to a wall for a certain time.

- As mentioned in the description of ICL the drag model is bad and fits only for demonstration purposes. Vallier [24] suggested improvements to get a better drag model, which could be integrated in ICL.

5.2.2 Postprocessing

So far the only way to post-process Lagrangian particles is the use of the `foamToVTK` a utility which comes with OF and creates native files for ParaView in the Visualization Toolkit (VTK) format. Neither `paraFoam` or the native OpenFOAM reader[21] for ParaView can be used to post-process Lagrangian particles.

OF 1.6 comes with the `particleTracks`[22] utility which converts the tracks of all particles as lines in VTK format. This can be used to track individual particles if few particles (around 10) are used, when more particles are used the net of lines becomes so dense that it's very difficult to follow an individual particle.

5.2.3 Profiling

Profiling is a dynamic program analysis which can help to understand where the computer spends it's CPU time. It measures the frequency and duration of function calls and assigns the time spent at an individual function as percentage of the time the whole program took to perform a specific task.

The tunnel case (with 100 particles) was analyzed using Apple's profiling tool Shark [6]. In order to use a program with Shark the symbols should not be striped (GCC option `-s`) and debugging should be enabled. (GCC option `-g`)

FreeFOAM is compiled by default with optimization (GCC option `-O2`). As a result elementary operations, such as vector calculations are optimized and take much less time. When trying to perform

[21] See `http://openfoamwiki.net/index.php/Tip_Building_ParaView3_With_OpenFOAM_Native_Reader`
[22] Can be found in `$FOAM/applications/utilities/postProcessing/lagrangian/particleTracks/`

an optimized build this however leads to problems. GCC cannot keep track of the line numbers anymore which leads to confusing time percentages assigned to lines for which it makes no sense (for example comments). However the time percentages are usually no more then two lines off. When looking at the assembly code next to the source it's sometimes possible to figure out what's happening and where the time percentage actually belongs to.

Another problem is that OF uses a lot of macros and inline functions which make the job of the keeping track of the line numbers very hard for the compiler (if possible). When compiling without optimization the time percentages are usually at the correct place. The problem is just that the performance behavior of the solver is not the same as with optimization and therefore it's necessary to profile the optimized build as well.

Below an excerpt of the profiling reports from shark are shown. They show the ten functions which took most of the CPU time.

```
# Report 1 - ParticleTunnelNonOptimized.mshark - Time Profile of ff_basicParticleFoam
SharkProfileViewer
# Generated from the visible portion of the outline view
- 7.4%, Foam::minusOp<double>::operator()(double const&, double const&) const,
        ff_basicParticleFoam
- 5.1%, Foam::innerProduct<Foam::Vector<double>, Foam::Vector<double> >::type
        Foam::operator&<double>(Foam::Vector<double> const&, Foam::Vector<double>
        const&), ff_basicParticleFoam
- 3.6%, Foam::interpolationCellPointFace<Foam::Vector<double> >::findTet
        (Foam::Vector<double> const&, int, Foam::Vector<double>*, int*, int*,
        double*, double*, int&, double&) const, libfiniteVolume.dylib
- 3.4%, Foam::eqOp<double>::operator()(double&, double const&) const,
        ff_basicParticleFoam
- 2.4%, Foam::multiplyOp3<double, double, double>::operator()(double const&, double
        const&) const, ff_basicParticleFoam
- 2.0%, Foam::divideEqOp2<double, double>::operator()(double&, double const&) const,
        ff_basicParticleFoam
- 2.0%, double Foam::mag<Foam::Vector<double>, double, 3>
        (Foam::VectorSpace<Foam::Vector<double>, double, 3> const&),
        ff_basicParticleFoam
- 1.9%, void VectorSpaceOps<3, 0>::op<Foam::Vector<double>,
        Foam::VectorSpace<Foam::Vector<double>, double, 3>, Foam::minusOp<double> >
        (Foam::Vector<double>&, Foam::VectorSpace<Foam::Vector<double>, double, 3>
        const&, Foam::VectorSpace<Foam::Vector<double>, double, 3> const&,
        Foam::minusOp<double>), ff_basicParticleFoam
- 1.7%, void VectorSpaceOps<3, 1>::eqOp<Foam::VectorSpace<Foam::Vector<double>,
        double, 3>, Foam::VectorSpace<Foam::Vector<double>, double, 3>,
        Foam::eqOp<double> >(Foam::VectorSpace<Foam::Vector<double>, double, 3>&,
        Foam::VectorSpace<Foam::Vector<double>, double, 3> const&, Foam::eqOp<double>),
        ff_basicParticleFoam
- 1.7%, Foam::Vector<double> Foam::operator-<Foam::Vector<double>, double,
        3>(Foam::VectorSpace<Foam::Vector<double>, double, 3> const&,
        Foam::VectorSpace<Foam::Vector<double>, double, 3> const&),
        ff_basicParticleFoam
```

Above the report of the non-optimized build is shown. Below the report of the optimized build. Note the huge difference.

```
# Report 2 - ParticleTunnelOptimized.mshark - Time Profile of ff_basicParticleFoam
SharkProfileViewer
# Generated from the visible portion of the outline view
- 15.5%, Foam::interpolationCellPointFace<Foam::Vector<double>
         >::findTet(Foam::Vector<double> const&, int, Foam::Vector<double>*, int*, int*, double*, do
- 9.0%, Foam::interpolationCellPointFace<Foam::Vector<double>
```

```
        >::interpolate(Foam::Vector<double> const&, int, int) const, libfiniteVolume.dylib
- 8.0%, Foam::Particle<Foam::HardBallParticle>::findFaces(Foam::Vector<double> const&)
        const, ff_basicParticleFoam
- 6.5%, longest_match, libz.1.dylib
- 5.1%, deflate_slow, libz.1.dylib
- 1.8%, compress_block, libz.1.dylib
- 1.7%, Foam::HardBallParticle::updateProperties(double,
        Foam::HardBallParticle::trackData const&, int, int), ff_basicParticleFoam
- 1.5%, pqdownheap, libz.1.dylib
- 1.5%, lck_mtx_unlock_darwin10, mach_kernel
- 1.5%, __vfprintf, libSystem.B.dylib
```

The effect of using `-O2` is huge. While without optimization most of the time is spent for trivial operations (such as `minusOp`) these functions do not appear on the top anymore when running an optimized build. In both reports the function `findTet(..)` appears. The method `interpolate(..)` interpolates the cell values at a given point. It uses `findTet(..)` which decomposes the cells (usually hexahedra) into tetrahedra and reports the tetrahedra which contains the given point.

6 Modern GPUs and General Purpose Computing

6.1 Introduction

When improving their applications, most software developers, relied on the rapid performance increase of a single central processing unit (CPU). This is especially true for computationally intensive applications, such as games, computer aided design (CAD), computational fluid dynamics (CFD), image or video processing. This trend is described by Moore's Law which states that the complexity of an integrated circuit is doubled in about every two years.

Since 2003 this drive was slowed down, because of issues with energy-consumption and heat-dissipation[23], which limited the of the clock frequency and the level of productive activities that can be performed in each clock cycle. Therefore most microprocessor vendors switched to models where multiple processing units can be used at the same time. These processing units are referred as core and lie within the same chip. While the (theoretical) performance, thanks to multiple cores, still increases, software no longer profits automatically from multiple cores: The software must be able to run multiple threads concurrently so it can use multiple cores. This has a tremendous impact on the software developer community, since algorithms must be parallelized and software must be (at least partly) rewritten. It should also be noted that the practice of writing concurrent programs was nothing new at this time, it was however uncommon that desktop computers were capable of executing concurrent programs physically concurrent[24]. Therefore most software developers developed sequential programs only.

Nowadays there are two main designs for microprocessors, according to [12]: The multicore approach, which is used for the CPUs. An example for a multicore CPU is the recent Intel Core i7 processor, which has four cores, each of which implementing the full x86 instruction set. The processor is further optimized to execute sequential programs fast: If just one core is used heavily, while the others are inactive, it allows the active core to run at a higher clock speed then usual.

The other major design is the many-core design, which focuses on the execution throughput of parallel applications. Many-core designs are used for graphics processing units (GPU), such as the NVIDIA GeForce GTX 283 with 30 streaming multiprocessors, each having 8 cores. In this design however each core shares control and instruction cache with 8 other cores. Because of this GPUs lead the floating-point performance race since 2003.

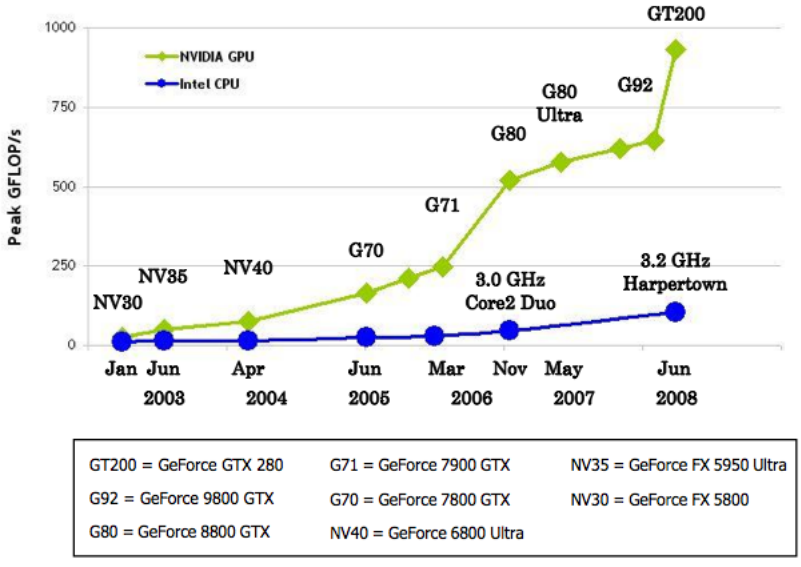

Figure 10: Floating-Point Operations per second of Nvidia GPUs and Intel CPUs. Taken from [19]

[23] Heat production increases non-linearly with the clock frequency, while it increases linearly with the number of cores.

[24] The operating system manages threads created by a program. If only one core is available and a program uses multiple threads, then the threads are scheduled by the operating system sequentially. The end user could still have the impression that things were running in parallel, this is because of the short interval during which a thread runs until it is interrupted and another thread can run for a short time. If multiple cores are available, then the scheduler distributes these threads to the cores available.

Looking at figure 10 one may ask why GPUs perform so much more floating point operations than CPUs do. First of all, this is just for single precision floating point operations. The picture looks much different if double precision floating point operation are required. Only recent GPUs support floating point operations with double precision and are much slower, newer generations of GPUs will probably come with faster double precision support. Further reasons are that the cores in a GPU do not implement the whole x68 instruction set and it is expected that all threads follow more or less the same execution path, because the control cache is shared among cores. Also cache must be managed manually on a GPU and GPU cores have a much simpler instruction pipeline. GPUs do not support things like branch prediction[25] as it is done on modern CPUs.

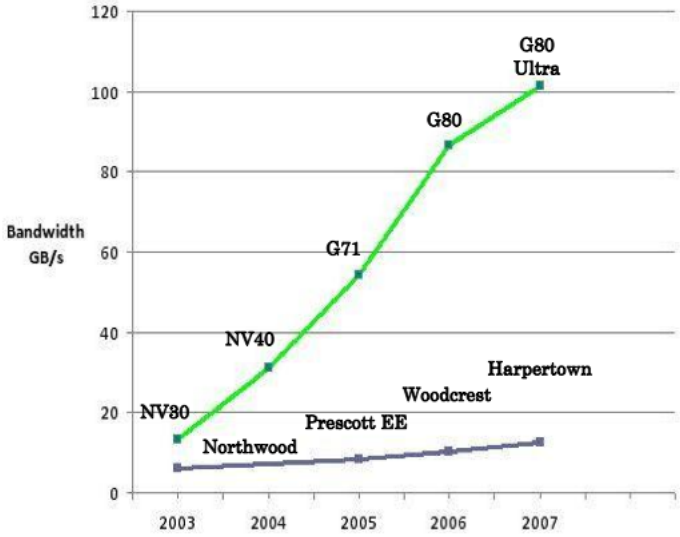

Figure 11: Memory Bandwidth of Nvidia GPUs and Intel CPUs. Taken from [19]

GPUs have about ten times the bandwidth of currently available CPUs (see figure 11). The reasons for this are that CPUs have to satisfy requirements from legacy operating systems, applications and I/O devices which make it difficult to increase the memory bandwidth. GPUs on the other hand have a relaxed memory model and are designed with a huge memory bandwidth in mind. The main cause for this is how realtime 3D graphics work, which is described later.

It is worth mentioning here that the the total revenue of the video game industry, which is the main driver behind the recent GPU development, overcame the total revenue of the film industry a few years ago. It is total revenue was estimated $41.9 billion and it is expected to grow to $68 billion in 2012 [5] . This lead to two consequences: GPU vendors have a large budget to maximize the numbers of floating-point operations possible per video frame. Furthermore because of the huge market created for GPUs, many desktop computers today have a decent GPU. NVIDIA, for example, estimates that their G80 processors have been sold more than 200 million units to date ([14], page 6).

The opportunity created to accelerate applications by the wide spread usage of modern GPUs is huge. Roughly speaking a GPU can accelerate numerical problems which can run on many parallel threads. This includes applications from, but not limited to the following topics:

- Digital image processing
- Linear algebra with dense and sparse matrices.
- Cryptography
- Computer Graphics

[25] A branch predicator tries to guess in which way a branch goes before it is known for sure. This helps to increase the flow in the instruction pipeline. Branch prediction also requires that operations which are made under an assumed branch can be undone in case it turns out that the branch prediction was wrong.

- Molecular Dynamics
- Computational Fluid Dynamics
- Audio/Video compression

6.2 CUDA

CUDA stands for Compute Unified Device Architecture and is an abstraction layer which helps to write programs without having to know how exactly the underlying hardware works. CUDA is scalable, which means programs written in CUDA can run on any CUDA enabled GPU, independent of the exact hardware configuration. For example the same program can run on a GPU with two streaming multiprocessors or on one with four or eight multiprocessors. When it comes to performance optimization knowledge of the underlying hardware is often still required.

CUDA defines three key abstractions - a hierarchy of thread groups, shared memory and barrier synchronization. These are exposed to the programmer as a simple language extension for the C language. Below an overview of the most important CUDA concepts along with some examples are given. For further reference see the CUDA Programming Guide [19].

6.2.1 Kernels

Let's start with a simple example, adding two vectors. A sequential implementation on a CPU is straightforward.

```
1
2 void addVectors(float* a, float* b, float* c, int N) {
3     for(int i=0; i<N; i++)
4         c[i] = a[i] + b[i];
5 }
```

Adding two vectors on a CPU

When using CUDA a computational kernel, which can be executed on a GPU, must be defined. CUDA defines the **host** as the machine itself and the **device** as the GPU(s) used by the machine to do general purpose computing. A kernel is defined using the `__global__` declaration specifier. As the reader will notice the loop in the CPU implementation is gone, this is because the kernel will be executed `N` times in parallel by `N` threads. The kernel is called on the host code. (line 12) The `<<<.. >>>` is called the execution configuration: `N` specifies the number of threads, which must correspond to the number of components in the vector, while `1` specifies the number of thread blocks. The variable `threadIdx` on line 3 is a built in variable which holds the index of each thread.

```
1  __global__ void addVectorsOnDevice(float* a, float* b, float* c, int N)
2  {
3    int i = threadIdx.x;
4    if(i<N) {
5      c[i] = a[i] + b[i];
6    }
7  }
8
9  int main()
10 {
11 ..
12 addVectorsOnDevice <<< 1, N >>> (a_d, b_d, c_d, N);
13 }
```

Adding two vectors using CUDA

6.2.2 Logical Thread Hierarchy

The `threadIdx` variable provides a 3-component vector, so that threads can be indexed using a one-dimensional, two-dimensional or three-dimensional thread index. Threads can be grouped together to threads blocks, which are indexed by a `blockIdx` variable similar to the `threadIdx` variable. A thread block can hold a maximum of 512 threads. (See [19] - chapter 2.2)

6.2.3 Memory Model

Registers and local memory Each threads has on-chip registers and local private memory. The compiler usually places local variables into registers. Exceptions are arrays for which the compiler cannot determinate the size or large structures or unions as well as any variable if the kernel uses more registers than available. In these cases the variables are placed in local memory, which is off-chip and has a much higher latency and therefore can cause performance implications. The cuda compiler `nvcc` supports the option `--ptxas-options="-v"` which displays the register usage.

Shared Memory Threads in the same block can share data by using **shared memory**, on-chip memory which is much faster then local or global memory. Each thread block has 16k shared memory. Shared memory can be used to avoid slow access to the global memory by caching data in shared memory.

Global Memory Global memory is on-device, off-chip memory, typically implemented with dynamic random access memories (DRAMs). Compared to shared memory, access to global memory is slow, this is because of the way how DRAM works: DRAM is basically a very large array of tiny semiconductor capacitors, the presence of a tiny amount of electrical charge distinguishes between 0 and 1. To read the data the charge must be shared with a sensor and if a sufficient amount of charge is present, the sensor detects a 1 (0 otherwise). This is a slow process, the latency for a global memory access is around 400 to 800 clock cycles (See [19] - chapter 5.2.3). Modern DRAMs use a parallel process to increase the rate of data: Each time a location is requested many consecutive locations, which include the requested location are read. Data at consecutive locations can be read at once and then be transferred at high speed to the processor. The programmer must ensure that the program arranges its data in a way that it can be accessed consecutively whenever possible. Otherwise the application will not perform optimally, because more transactions then theoretically needed will be required.

The architecture of modern GPUs is built around this limitations: All threads running on the same multiprocessor execute the same instruction at any time, when all threads execute a load instruction it is detected whether the threads access consecutive global memory locations, if this is the case the hardware combines or coalesces all these accesses into one transaction. Further a streaming multiprocessor (SM) is usually assigned much more threads then it can execute at the same time. So if a thread has to wait for data from global memory, another thread is executed at the same time. This makes it possible to utilize the available hardware properly. (See [14] - chapter 6)

Global memory can be allocated by using `cudaMalloc(..)` and memory from the host to the device (and back) can be copied using `cudaMemcpy(..)`. Threads can fetch data from global memory in parallel if size and alignment requirements are considered. (See [19] G 3.2)

Constant Memory On device, off-chip and automatically cached. It can be read within one cycle of latency if there is a cache hit. The size of constant memory is limited to 64k. (see [19] - page 140) It is optimized for the case when all threads read the same location, therefore it is often used to store the size of arrays. (See [10])

Texture memory Texture memory resides in the device memory (off-chip) and is cached in texture cache. The costs for a memory fetch only arise in case of a cache miss. The texture cache is optimized for 2D spatial locality.

6.2.4 Debugging Code on a GPU

Since CUDA 3.0 it is possible to debug CUDA code on Linux[26] by using an enhanced version of gdb, cuda-gdb. cuda-gdb requires a dedicated GPU only for debugging, it is not possible to debug on the same GPU which runs the X11 Window System. Before CUDA 3.0 the only way to debug was to compile CUDA programs with emulation mode so that they run on the CPU instead of the GPU. Further Nvidia provides `cuprintf(..)`, which works in a similar way as `printf(..)` in host code and makes it possible to write a log while a CUDA program is executed on the GPU.

It is important to mention here that there is absolutely no memory protection on a GPU. If a program running on a CPU tries to access an invalid address, it is process is killed by the operation system and an error is displayed.[27] On the GPU nothing like this happens, except that the author noticed that it is possible to crash the GPU driver (and with this freeze the whole operating system..) in some rare cases. Fortunately the cuda-gdb debugger provides memory checks. If one suspects a memory management problem the program should be run inside cuda-gdb, which can be turned on by typing `set cuda memcheck on`, the debugger will then report the line at which the invalid memory access happens.

6.3 Remarks on the architecture of a GPU and performance considerations

The architecture of a CUDA capable GPU is built around a scalable array of multithreaded *Streaming Multiprocessors (SMs)*. The architecture of a SM is called *SIMT*[28] *(Single-Instruction, Multiple-Thread)* and allows the execution of hundreds of threads concurrently. Unlike with threads running on a CPU the thread scheduler is implemented fully in hardware and it is therefore possible to switch threads instantly and create them with very little overhead. Threads are scheduled, executed and managed in a group of 32 threads called *warps*. Warps are part of the CUDA implementation and not of the logical thread model. Threads in a warp start at the same address and execute instructions in parallel, but they have their own registers and instruction address counters. A simplified overview of the Nvidia's G80 architecture is shown in 12

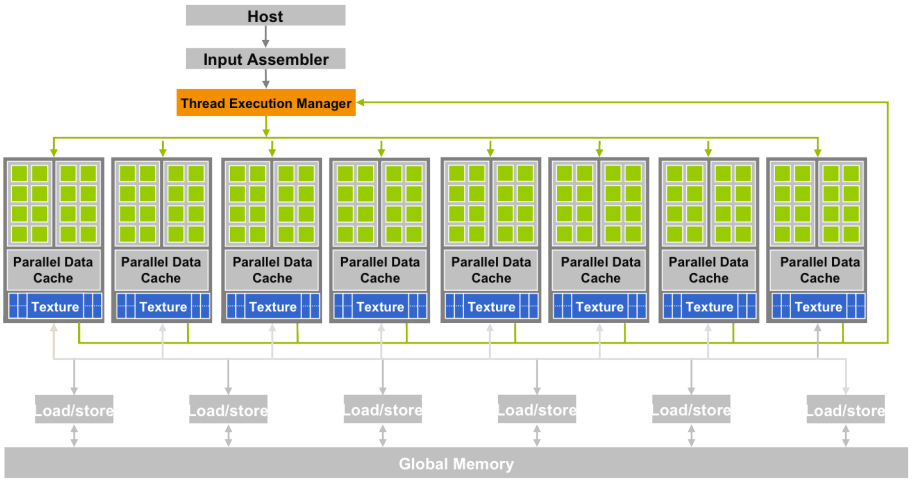

Figure 12: Overview of Nvidia's G80 architecture. Taken from the lectures of David Kirk and Wen-mei W. Hwu

[26] At the time writing this there is a beta version of an extension for Visual Studio on Windows available which makes debugging on Windows possible. For Macs, however, cuda-gdb is not available yet.

[27] For example a "segmentation fault" if the program tries to access an address which it is not allowed to or a "bus error" in case of an address that the CPU cannot physically address. (The errors are called this way in the case of an UNIX-like operating system like Mac OSX, other modern operating systems have similar concepts.)

[28] GPUs are also often referred to as SIMD (Single Instruction Multiple Data) architectures, because each thread is expected to follow more or less the same execution path, while the data which each thread processes is different. Nvidia avoids calling their devices SIMD, because of some architectural differences with SIMD architectures like Intels Streaming SIMD Extensions (SSE). A SIMT architecture can switch quickly between threads (for example if a thread is waiting for data, another thread can be executed), SIMD architectures like SSE cannot do anything like this.

As mentioned before several threads are bundled for execution. Each block is partitioned into warps, the size of a warp can vary, it is 32 threads on the G80.

Partitioning of 1D indexed threads: Warp 0 starts with thread 0 and ends with thread 31, warp 1 starts with thread 32 and ends with thread 63 and so on. Warp n starts with thread 32*n and ends with thread 32(n+1)-1. Blocks whose number of threads is not a multiple of 32 will be padded with extra threads.

For blocks consisting of multiple dimensions (2 or 3) of threads, the dimensions are projected in a row-major order into warps. Let's assume a block with 16x16 threads. Warp 0 starts with thread(0,0) and ends with thread (1,15). Warp 2 starts with thread(2,0) and ends with thread(3,15). For a two dimensional block all threads whose z-index is 0 are treated as a two dimensional block, then the linearized two dimensional block of all threads with z-index 1 follows and so on.

The warp scheduler selects and executes one warp on each Streaming Multiprocessor (SM). Instructions are run simultaneously on all threads in a warp. The cost of fetching and processing an instruction is amortized if a large number of threads execute the same instruction at time. Branching at an if-then else construct is simple as long as all threads take either the then or the else path. If some threads take the then path and some the else path, we say that their execution path diverges. In this case all threads go through the then path and through the else path with the concerning threads disable in either the then or else part of the branch. Divergence in looping constructs results in that all threads have to go through all instructions in the loop until the last thread finishes. Execution divergence happens when branching is done on the basis of thread IDs. Therefore branches like `if(threadIdx.x == 3)` should be avoided. While small divergence in the execution path do not slow down the execution threads with complex diverging execution paths can be a problem. Nvidia defines a metric called *Occupancy*, which measures how much the GPU is actually utilized. A low occupancy could mean for example that the threads are constantly waiting for data from global memory. Occupancy depends on:

- Number of threads per block.
- Registers per thread.
- Shared memory requirements per block.
- As well as the compute capability of the GPU.

Register usage is important, the registers available on a streaming multiprocessor (SM) are limited. If the register requirements of the kernels are high, then the scheduler can assign less thread blocks to the SMs and their execution must be serialized. Nvidia provides a spreadsheet, which makes it easy to compute the occupancy and figure out how to improve it, it is available here: [17].

6.4 Fast Matrix Multiplication Using Shared Memory

Multiplying two matrices $A * B = C$ seems to be trivial to implement since every position in the C matrix can be calculated independently.

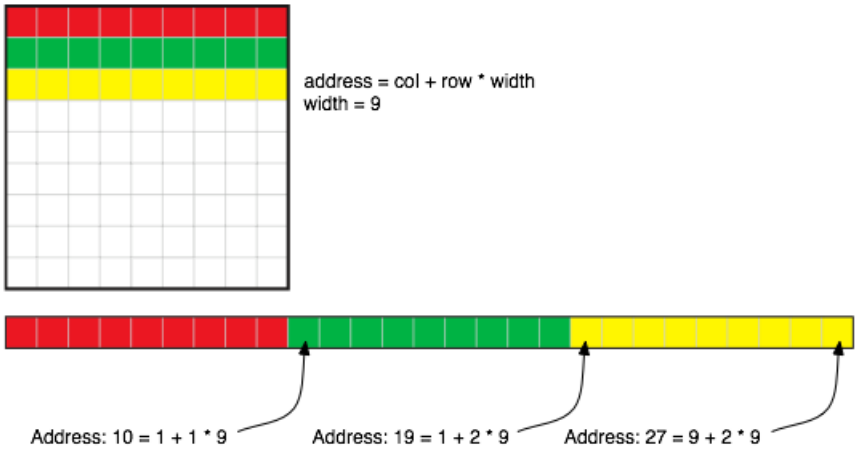

Figure 13: Row major order addressing of a Matrix in a one dimensional array.

Below a kernel is shown which takes the matrices A, B and C as well as the number of elements n as arguments. The matrices are stored in one dimensional arrays using row-major order (see figure 13 for illustration).

```
__global__ void mat_mul_kernel
  (const float* A, const float* B, float* C, int n)
{
    float c = 0;

    int row = threadIdx.x + blockIdx.x * BLOCK_SIZE;
    int column = threadIdx.y + blockIdx.y * BLOCK_SIZE;

    for (int k=0; k<n; k++) {
        c += A[row*n + k] * B[k*n + column];
    }

    C[row*n + column] = c;
}
```

Naive Matrix Multiplication

The main problem with the naive matrix multiplication is that every thread must access global memory n times, because arrays A, B and C are stored in global memory. The calculation is done much faster then the memory access, so in this case memory latency is a bottleneck which prevents the program from performing optimally. By using the on-chip shared memory, which is much faster then the global memory, it is possible to work around this bottleneck.

The idea behind the new algorithm is to divide the matrix into sub matrices so that the two sub matrices involved in the calculation can be stored in shared memory at the same time. This will require much less access to global memory. Thread blocks are mapped to blocks of the square matrix C in horizontal direction. In a thread block, each thread fetches one element from the corresponding block in the matrix A and B and stores it in shared memory. Before any calculation can be done, it must be assured that all elements (of the corresponding sub matrices) are loaded, this is what the `__syncthreads();` call is for. Then the row and column corresponding to the element in the sub matrix of C are multiplied, the result is stored in the thread-local register `Cval`. Each thread iterates over all sub matrices which correspond to one element in the C matrix and stores it back in global memory when done. Figure 14 illustrates this process: In order to compute element (1,1) in the C matrix, the corresponding threads need to reload three times. The commented code below should be easy to understand now.

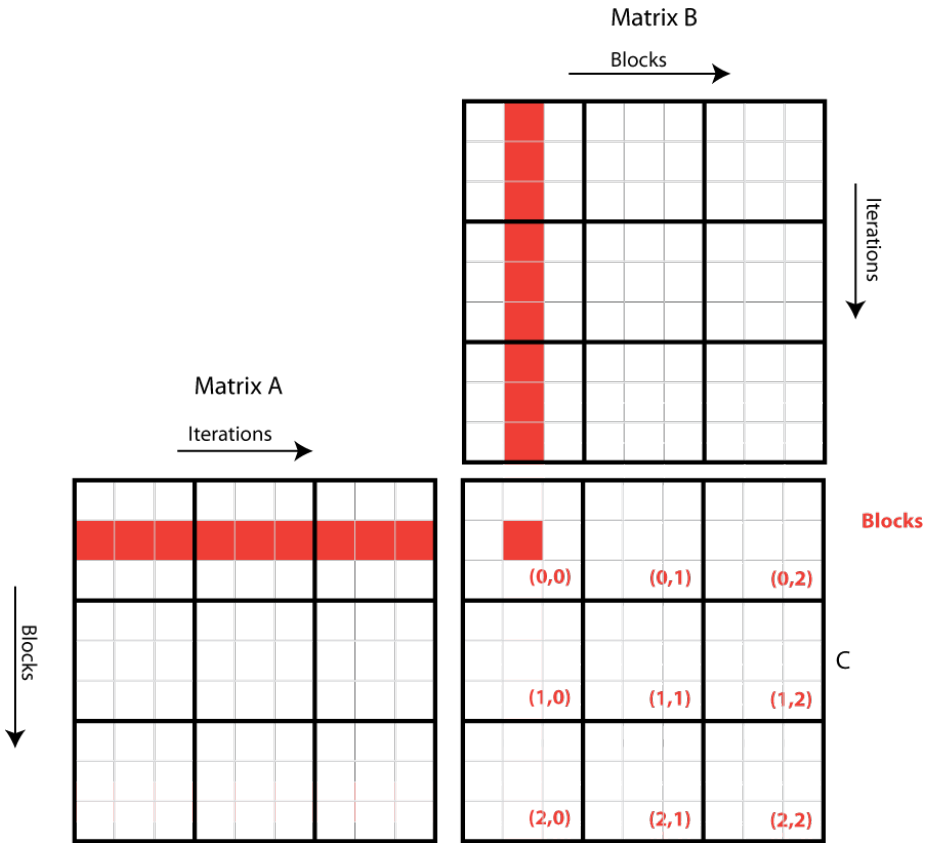

Figure 14: A 9x9 matrix is divided into 3x3 sub matrices which are block wise multiplied.

```
// Getting an element using row-major ordering
__device__ float getElement
  (const float* A, int row, int col, int width)
{
  return A[col + row*width];
}

// Recall that A.width == B.height
//         and  A.height == B.width
// A * B = C
__global__ void MatMulKernel
  (float* A, float* B, float* C, int Awidth, int Bwidth)
{
// Threads correspond to values in a block of the C matrix
// Each value of the C matrix is calculated by exacly one thread
int threadRow = threadIdx.y;
int threadCol = threadIdx.x;

// Blocks of the C matrix correspond to thread blocks
int blockRow = blockIdx.y;
int blockCol = blockIdx.x;

float Cval = 0;

// The number of blocks that we have in a row of the matrix A
// or in a col in case of matrix B
int nrIter = Awidth / BLOCK_SIZE;

// In order to calculate one block in the C matrix
```

```
// we need to load Awidth/BLOCK_SIZE sub matrices from A and B
for (int i=0; i<nrIter; i++)
{
  __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

  // Now every thread loads one element into shared memory
  // Recall row-major ordering in a 1D array: address = col + row * width

  // Load corresponding element from the matrix A
  As[threadRow][threadCol] = A[
    threadCol + threadRow * Awidth   // Address of the thread in the
                                     // current block
    + i * BLOCK_SIZE                 // Pick a block further left for i+1
    + blockRow * BLOCK_SIZE * Awidth // for blockRow +1 go one blockRow down
  ];

  // Load corresponding element from the matrix B
  Bs[threadRow][threadCol] = B[
    threadCol + threadRow * Bwidth // Address inside the current block
    + i * BLOCK_SIZE * Bwidth      // Note that this is the same as for A
    + blockCol * BLOCK_SIZE        // just blocks and cols swapped
  ];

  // Synchronize to make sure all the matrices are loaded
  __syncthreads();

  // Now calculate the element which corresponds to the current thread.
  // This is done only for one sub matrix and summed up for all sub matrices.
  for (int j=0; j<BLOCK_SIZE; j++) {
    Cval += As[threadRow][j] * Bs[j][threadCol];
  }

  // Ensure that calculation is done before writing it back
  __syncthreads();
}

// Write back the value into the array for the C matrix
  // Again this is the Adress inside the sub matrix (below)
C[threadCol + threadRow * Bwidth
  // and this the adress of the beginning of the sub matrix (below)
  + blockCol * BLOCK_SIZE + blockRow * BLOCK_SIZE * Bwidth
  ] = Cval;
}
```

Dense Matrix Multiplication using Shared Memory

7 Lagrangian Particle Tracking on the GPU

In this section the ideas to implement Lagrangian particle tracking in a non-structured mesh on a GPU are presented.

The description is using the symbols introduced in section 3. Calculating the λ for each face for each cell of in which a particle resides initially can be parallelized. Therefore the main idea is to run a kernel which calculates the smallest λ, along with the face hit, if any for each particle on the GPU, while running the rest of the algorithm on the CPU. The advantage of this is that no complicated logic, which requires complicated control flow and could lead to divergent wraps, must be implemented on the GPU.

Algorithm 4. *Particle tracking on a GPU.*

Main program on the CPU

while *not all particles are done.* ***do***
    upload $\mathbf{a}$ *and* $\mathbf{b}$ *to the GPU (for all particles which are not done)*
    invoke the GPU kernel, which computes $\lambda_{smallest}$ *and* f_{hit}
    download $\lambda_{smallest}$ *and* f_{hit}
    for *particles with* $\lambda_{smallest} = -1$ ***do***
        move particle to the end position and mark particle as done
    end for
    for *particles with* $\lambda_{smallest} \neq -1$
        move the particle along $\mathbf{v} = \mathbf{b} - \mathbf{a}$ *to* f_{hit}
        change its cell occupancy to the neighbouring cell of face f_{hit}
        Update $\mathbf{a}$*:* $\mathbf{a_{new}} = \mathbf{a} + \lambda_{smallest}(\mathbf{b} - \mathbf{a})$
    end for
end while

Particle tracking on the GPU, one thread per particle

find the set of faces, F_i *for which* $0 \leq \lambda_c \leq 1$
if *size of* $F_i = 0$ ***then***
    Set $\lambda_{smallest}$ *and* f_{hit} *to -1*
else
    find face $f \in F_i$ *for which* λ_a *is smallest*
    store $\lambda_{smallest}$ *and* f_{hit}
end if

The GPU kernel needs to be able to access the face centres and the face normals, as well as the vectors $\mathbf{a}$ and $\mathbf{b}$ of a cell in a coalesced (6.2.3) way, so that a high occupancy can be achieved. This requires a suiting data structure, for now a prototype was created, which stores a two dimensional mesh one dimensional arrays in a similar fashion as OpenFOAM, e.g. an array of points, then an array of faces.

Conclusion

Using Lagrangian particles in a simulation adds the following steps to the calculations, which need to be done in one timestep:

- Calculate the movement of every particle using a given Eulerian phase. (Or a fixed vector field, as in the test case.)
- In order to calculate the velocity at every timestep for every particle, it is further required to interpolate it, since the values of the Eulerian phase are only known at the centroid of the cells.
- Figure out through which cells the particle is moving and how much time it spends there. The time the particle spent in every cell could be used for coupling: The calculation of the particles velocity from the Eulerian phase and if a two-way coupling is used, also the influence the particle has on the Eulerian fields in the cells it crossed. Further it needs to be checked if the particle hits a wall and if this is the case an appropriate reflection function must be applied on the velocity of the particle.

- Simulation dependent calculations, such as breakup models, collisions models, etc.

Profiling of a simple test case (see 5.2.3), showed that the interpolation contributes a large percentage to the total execution time, while the functions, which implement the face-to-face tracking algorithms are not contributing much. It is needs to be verified weather this is the case in general or just in this example. This of course rises the question weather interpolation should also be done on the GPU and weather it could be accelerated.

In any case it is for sure that the mesh needs to be stored on the GPU in a way which allows fast access to mesh elements like face centres or normals on a per cell basis. Because of the high latency of DRAM this will be challenging. I has been shown in [7] that it is possible to store an unstructured tetrahedral grid so that it is elements can be accessed in a coalesced way. The grid used there however only consisted of tetrahedras, while OpenFOAM allows polyhedral meshes. While shared memory could not be used in this case, it needs to be considered for implementing the particle tracking algorithm: Is it possible to fetch the data of a group of particles (eg their position at the beginning and the end, as well as the face centres and normals) concurrently into shared memory and then avoid global memory access until a reload is necessary? A similar strategy as the one considered in [20] needs to be considered.

References

[1] Lagrangian and Eulerian specification of the flow field. http://en.wikipedia.org/wiki/Lagrangian_and_Eulerian_specification_of_the_flow_field.

[2] Wikipedia - Unstructured Grid. http://en.wikipedia.org/wiki/Unstructured_grid.

[3] *OpenFOAM Programmers Guide*. OpenCFD UK, 2008.

[4] *OpenFOAM Users Guide*. OpenCFD UK, 2008.

[5] Frank Caron. Gaming expected to be a $68 billion business by 2012. http://arstechnica.com/gaming/news/2008/06/gaming-expected-to-be-a-68-billion-business-by-2012.ars, 2008.

[6] Apple Computer. Shark user guide. http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/SharkUserGuide/Introduction/Introduction.html, 2009.

[7] Andrew Corrigan, Fernando F. Camelli, Rainald Löhner, and John Wallin. Running unstructured grid-based cfd solvers on modern graphics hardware. *International Journal For Numerical Methods in Fluids*, 2009.

[8] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry - Algorithms and Applications*. Springer, third edition, 2008.

[9] Reimar Döffinger. Physikalische Hochleistungssimulation auf GPU-Systemen. Master's thesis, Universität Karlsruhe, 2009.

[10] Rob Farber. Cuda, supercomputing for the masses: Part 11. http://www.drdobbs.com/high-performance-computing/215900921.

[11] Bernhard Gschaider. icoLagrangianFoam - A solver for incompressible flow and hard particles. http://openfoamwiki.net/index.php/Contrib_icoLagrangianFoam, 2006.

[12] Wen-Mei Hwu, Kurt Keutzer, and Timothy G. Mattson. The concurrency challenge. *IEEE*, 2008.

[13] Hrvoje Jasak. *Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows*. PhD thesis, Imperial College, 1996.

[14] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Nvidia, 2010.

[15] Graham B. Macpherson, Niklas Nordin, and Henry G. Weller. Particle tracking in unstructured, arbitrary polyhedral meshes for use in CFD and molecular dynamics. *Communications in Numerical Methods in Engineering*, 2008.

[16] Niklas Nordin. *Complex Chemistry Modeling of Diesel Spray Combustion*. PhD thesis, 2000.

[17] Nvidia. Occupancy spreadsheet. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[18] Nvidia. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 09.

[19] Nvidia. Cuda programming guide. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf, 2 2010.

[20] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with cuda. *GPU Gems*, 2007.

[21] Air pollution modelling using a Graphics Processing Unit with CUDA. Air pollution modelling using a graphics processing unit with cuda. *Computer Physics Communications*, 2009.

[22] Steven Pope. *Turbulent Flows*. Cambridge, 2000.

[23] Data Analysis US Department of Defense and Assessment Center. DAAC - Types of Grids. `https://visualization.hpc.mil/wiki/Types_of_Grids`.

[24] Aurelia Vallier. Tutorial icolagrangianfoam / solidparticle. Technical report, Stromninsteknik - Energivetenskaper LUND TEKNISKA HOGSKOLA, 2009.

[25] H. G. Welle, G. Tabor, and C. Fureby H. Jasak. A tensorial approach to computational continuum mechanics using object-oriented techniques. *AMERICAN INSTITUTE OF PHYSICS*, 1998.

[26] Sebastian Wiedermann. Evaluation und Validierung paralleler Koprozessorstrukturen für Anwendung der numerischen Simulation. Master's thesis, Universität Karlsruhe, 2008.

8 Appendix

8.1 Aufgabenstellung

Das Open-Source Programm OpenFOAM bietet seit Version 1.3 [11] die Möglichkeit von Lagrangian Particle Tracking (LPT). Im Rahmen dieses Projekts soll die implementierte Version von LPT zuerst verstanden, dann anhand eines einfachen Beispiels getestet und schliesslich auf Effizienz untersucht werden. Mit Hilfe von Algorithmen und Hardware Architekturen aus der Computergraphik (CUDA, OpenCL, etc.) soll die bestehende Implementation schliesslich optimiert werden.

Am Ende der ersten Phase ist die aktuelle Implementation von LPT in OpenFOAM verstanden, wenn die wesentlichen Klassen identifiziert, ihre grundlegenden Methoden und Datenstrukturen bekannt sind, und diese auf ein einfaches Testbeispiel angewendet werden können. Dieses einfache Testbeispiel besteht darin, dass im Simulationsgebiet zufällig verteilte Partikel in einem stationären Geschwindigkeitsfeld transportiert werden.

In einer zweiten Phase sollen die für das Particle tracking verwendeten Algorithmen untersucht und mit entsprechenden Algorithmen aus der Computergraphik verglichen werden. Es soll eine erste Schätzung für die Performancesteigerung bei Verwendung von Graphikhardware gemacht werden.

In einer dritten Phase sollen rechenintensiven Algorithmen auf der dedizierter Graphikhardware ausgeführt werden. Dabei sollen die Möglichkeiten von CUDA, OpenCL, SIMD-Architekturen, etc. in Betracht gezogen werden.

Ziel dieses Projektes ist zu verstehen, wie der LPT-Code in OpenFOAM arbeitet und zu zeigen, dass die aktuelle Implementation durch die Verwendung von dedizierter Graphikhardware wesentlich optimiert werden kann. Es soll ein ausührlich dokumentierter Tutorial-Case aufgesetzt werden.