Universität Karlsruhe (TH)
Research University · founded 1825

Karlsruhe Institute of Technology

Numerical simulation of 3D particulate flows based on GPU technology

Diploma thesis

by

Tobias Hahn

Universität Karlsruhe (TH)
Institute for Applied and Numerical Mathematics IV
AG Numerical Simulation, Optimization and High Performance Computing

Supervisor

Prof. Dr. Vincent Heuveline

Second reader

Jun.-Prof. Dr. Jan-Philipp Weiß

Date of registration: November 1st 2008
Date of submission: April 30th 2009

Abstract

This thesis deals with a particular problem out of the research field of computational fluid dynamics, the numerical simulation of fluids containing soluted rigid particles. Such problems arise within a variety of applied sciences, such as medicine, ecology and engineering and need to be studied in detail in three-dimensions. So far most scientific publications on this topic either dealt with 2D phenomena only or the behavior of a very few particles in 3D. The reason for this being most of all performance constraints of the used simulation techniques and hardware.

This thesis aims to accomplish a high-performance simulation of a large number of particles in 3D by using co-processor technology. The landscape of such accelerators is reviewed thoroughly and assessed according to their versatility in scientific computing in general and the above problem in particular. Graphics processing units (GPU) are identified as technology of high potential, promising powerful computing abilities while being increasingly easier to program.

In order to fully benefit from the advantages of this new kind of hardware, the complete simulation development process is reviewed. A model for particulate flows is derived from the very basis of continuum mechanics leading to a constraint Navier-Stokes problem. The solving methodology is chosen to fit the GPU's architecture and achieve best possible performance results.

The implementation of the numerical method is done on the basis of intensive evaluation of the GPU used, by means of key scientific computation kernels. Final benchmarks of components of the particulate flow simulation demonstrate the successful use of GPUs to accelerate the computation by at least a factors of 3 and up to 20 in selected sub-routines.

Contents

1. Introduction

This thesis deals with the numerical simulation of *particulate flows*, i.e. the motion of rigid particles in viscous incompressible fluids. Clearly, the flow of fluids is influenced by the suspended particles, while the particle movement is affected likewise by while fluid phase. In this regard, particulate flows are a particular case of *fluid-structure interaction problems* and a major field of research in theoretical as well applied fluid dynamics. The combined motion of fluid and structure is described by highly coupled systems of differential equations whose solution in an efficient and accurate way is a challenging problem of *computational fluid dynamics (CFD)*.

A variety of models have been proposed for the described problem and of course, the choice of an adequate simulation method depends mainly on the simulation specifications. In any case, a high simulation performance is always of importance, thereby turning attention to the hardware: different hardware architectures favor different methods and programming techniques, such that commodity implementations may not suite current developments in hardware architecture.

In order to design a simulation method abreast with the latest developments in computer hardware, a deep insight into current chip design and computer architecture is needed for an assessment of suitability for the given purpose. In this thesis, a comparative survey of state-of-the-art *co-processor technology* is included, identifying *graphics processing units (GPUs),* as promising addition to *scientific computing* and qualified for the implementation of a particulate flow simulation.

As with every new technology, theoretical performance estimations of accelerator hardware may differ widely from the performance achieved in real-world applications. In order to successfully develop a simulation routine for execution on GPUs, thorough benchmarks are needed to identify possible bottlenecks.

Only on the basis of this expensive preliminary work, a design for a hardware-fitted particulate flow simulation algorithm can be developed and implemented. As pure data-generation is only one part of the simulation, this thesis deals as a round-up also with the problem of effectively visualizing particulate flows.

1.1 Motivation

Particulate flows occur in a huge variety of forms in large-scale as well as small-scale phenomena and am subject of research of many different sciences:

Hydrology explores transport processes in water, e.g. of nutrients, pesticides, dissolved and soluted solids and sediments. Complex phenomena, such as erosion and particle sedimentation in river beds need to be studied theoretically by means of mathematical modeling.

Ecology is another discipline in the group of environmental sciences, that aim to understand the many interactions of physical, chemical and biological processes happing in nature. One socially very important field of study of ecology is for example pollutant distribution in air and water - describable as a particulate flow.

Medicine is interested in small-scale phenomena such as atherosclerotic deposits in human blood vessels or the impact of respiratoral dust on the lung function, again examples of particulate flows.

Engineering, especially with a chemical and pharmaceutical focus, depends on the analysis of sophisticated mixing, drying and transport processes of different kinds. To understand the nature of the occurring phenomena and give predictions on their behavior under different conditions, again mathematical modeling of the underlying physics and chemistry is needed.

All these examples are taken from real-world applications and thus require a three dimensional study of the behavior of the respective fluids and partials.

The simulation by means of numerical mathematics is an computationally intensive task, especially when done in 3D. Different techniques for the simulation of particulate flows have been developed so far, the can most important two being:

- *Arbitrary Lagrangean Euler (ALE)* techniques and
- *Fictitious Domain Methods (FDM).*

The main difference between these two methods is that FDMs may use a fixed grid during the whole simulation while ALE techniques alter their meshes with each time step and require a complete re-meshing when the mesh gets too distorted after several steps.

The choice of simulation method depends firstly on the simulation specifications, both types of techniques have advantages over the other. In order to achieve a high simulation performance, the hardware aspect has to be kept in mind as well. As mentioned before, different hardware architectures favor different programming styles, such that traditional implementations may suffer from severe performance losses because of not considering developments in hardware design.

Namely these developments are multi-core and accelerator technologies. With frequency scaling of CPUs being limited by current leakage and heat production, several heavy-weight CPU cores, clocked at a reasonable rate, have been combined on a single chip with additional caches facilitating data exchange. The lower clock frequencies translate to less power consumption, while the whole multi-core CPU nevertheless achieves a higher over-all performance, when the cores are used in parallel. Currently only six-core CPUs are available, as the large size of these multi-purpose cores limits further scaling as well.
An alternative are simpler and smaller computing elements that can be combined in a larger number. This is the approach of most accelerator technologies, that accompany a conventional CPU and take over computational intensive tasks. Especially interesting in this field are graphics processing units (GPUs) that have been used exemplary in science in the past and grew in importance when outperforming state-of-the-art CPUs in special applications.

Clearly, it is of high interest to examine the new possibilities of GPU acceleration in numerical mathematics and its impact on the choice of numerical method and algorithm design.

1.2 Goals

The purpose of this thesis is to derive, implement and evaluate a numerical simulation of particulate flows with respect to current hardware developments.

The goals for the simulation are to trace a very high number of particles in viscous flow in three dimensions. It shall be time-dependent and achieve significant performance improvements compared to conventional implementations while conserving accuracy as much as possible.

The thesis further aims to give insight into the landscape of current accelerator technologies and their applicability in scientific computing. Special focus is again given to GPUs, namely the NVIDIA GT200 GPU, and their programming models.

The possibilities of GPU acceleration in scientific computing shall be assessed by thorough evaluation of this hardware/software combination using basic kernels, that constitute the main parts of most scientific algorithms.

This evaluation is the basis for designing the simulation algorithm with highest possible hardware efficiency. The task of simulating particulate flows shall give an evaluation of how GPUs perform in real-world applications of computational fluid dynamics.

Summing up, this thesis aims to solve the problem of simulating particulate flows from beginning to end: starting with a physical problem, a mathematical description is derived and implemented using newest hardware/software combinations and gained experimental results are visualized.

1.3 Outline

This thesis addresses the problem in the following structure:

A model for particulate flows is derived in **chapter two**. In order to achieve the goals of a high performance simulation of many particles, a fictitious domain method is derived and prepared for implementation.

Accelerator technology is reviewed in **chapter three**. GPUs are put into context with other available technologies, comparing their advantages and disadvantages. The programming model CUDA of the latest NVIDIA GPUs is presented as well.

Thorough hardware evaluation is conducted in **chapter four**. Basic scientific kernels, such as vector and matrix operations, are used to determine the performance of the GPU. It is furthermore reviewed what effort has to be undertaken to achieve it.

The numerical methods applied to solve the model derived in chapter two are explained in **chapter five**. The methods are chosen to fit the hardware specifications explained in chapter three and the performance results gained in chapter four.

Implementation details are given in **chapter six**. It is explained, how the computational problems of chapter five can be efficiently solved with the CUDA programming model. The achieved performance of these implementations is also reviewed in this chapter. Furthermore an example visualization technique is described and presented.

Conclusions are summed up in **chapter seven**. Furthermore, an outlook is given on how the problems dealt with in this thesis can be addressed in the future and how further hardware developments might influence the approach taken.

1.4 Acknowledgement

I wish to thank my thesis adviso Professor Vincent Heuveline for allowing this paper to be a product of my own interest and letting me explore this fascinating research field of particulate flows as a whole. I would like to further thank Prof. Heuveline for encouraging me during the course of my studies, keeping an eye on the structure of my work and steering me in the right directions, when necessary. Last but not least, this thesis would not have been possible without him providing access to the newest hardware technology available.

I am most grateful to Andrea Otzen and Björn Rocker for the many consultations we had along the way, for always suspending their own work, taking the time for a discussion and giving most valuable advice.

Finally, I wish to express my love and gratitude to all my family and friends. Particularly I like to thank my parents Doris and Edgar for their encouragement and their patience that allowed me to explore my interests and talents. Without them, this thesis would not have been written.

2. A model of moving rigid particles in viscous incompressible flow

2.1 Introduction and problem formulation

In this chapter, a model for the simulation of particulate flows is presented. The approach chosen is a *direct numerical simulation*, i.e. equations for the flow around the particles and the particle motion are derived from fundamental physical principles. A serious problem is the interaction of walls and particles and several particles with each other: here, a length scale might occur which is significantly smaller then the particle size, making it numerically difficult to apply the aforementioned general physical model. A non-direct simulation of the micro-scale phenomena will be inevitable in order to achieve the goals of this thesis.

We consider a fixed bounded domain Ω_1, bounded externally by Γ, filled with an incompressible Newtonian liquid of density ρ_f and viscosity μ. In this liquid we consider n rigid particles occupying a domain $\Omega_2 = \bigcup_{i=1}^{n} \Omega_{2,i}$ with densities $\rho_i,\ i = 1, \ldots, n$. Furthermore we denote the interface between Ω_1 and Ω_2 by Σ and the entire by domain $\Omega = \Omega_1 \cup \Omega_2$.

Because of the particles' movement, Ω_1 and Ω_2 are time dependent and should rather be denoted as $\Omega_1(t)$ and $\Omega_2(t)$. As the argument is obvious, it will be omitted for reasons of simplicity.

In the following sections, a model for this problem is derived and different possible numerical solution methods are presented. A particular fictitious domain method is further pursued as it offers several advantages in reference to the later implementation.

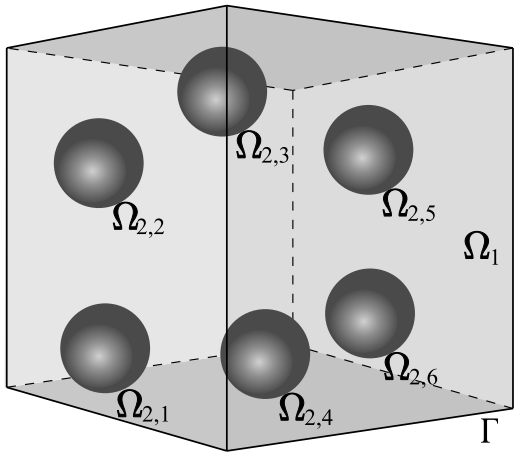

Figure 2.1: Geometry of the problem.

2.2 Fundamental physical principles

2.2.1 Continuum mechanics

Although matter is formed of atomic and subatomic particles, its behavior can be described and accurately predicted using the *continuum theory*, which neglects this molecular structure. Solids and fluids are thought to have no empty spaces, i.e. they are indefinitely divisible. Thus according to this theory, a particle in continuum has infinitesimal volume and in its neighborhood are always other particles.

In the case of flows, it is known that this model fails for length scales of the order of the mean free path of the molecules taking part in the flow. The relation of the mean free path to the smallest length scale appearing in the flow is called *Knudsen number*. A small Knudsen number depicts, that the scale of the studied phenomenon is sufficiently large such that the considered volume elements contain enough particles to neglect fluctuations of the physical quantities. In the case of air under atmospheric conditions, the mean free path is approx. $10^{-7}m$ and the smallest length scale is usually not lower then $10^{-4}m$. The Knudsen number 10^{-3} is much smaller then one in this case, such that the continuum hypothesis is a reliable basis [Pope00].

2.2.2 Newton's laws of motion

The three laws of motion describe the relations between the forces acting on a massive particle and its motion. The formulation given here is taken from [Batr06].

Newton's first law of motion
In an inertial frame of reference, a free particle continuous in its state of rest or of uniform motion.

Newton's second law of motion
In an inertial frame, the rate of change of linear momentum of a particle equals the resultant force acting on it. That is

$$\frac{\mathrm{d}}{\mathrm{d}t} M\mathbf{u} = \mathbf{F}, \tag{2.1}$$

where the momentum $M\mathbf{u}$ is the product of the (constant) mass and velocity of the particle. $\mathbf{F}$ is the (vector) sum of all forces acting on the particle.

Newton's third law of motion
To every action, there is an equal and opposite reaction.

The *inertial frame*, defined by the first law of motion is one in which this law holds. Here, the frame shall be spun by coordinate axis fixed to Earth without introducing an error. The forces acting in the second law can be divided in two groups, *body forces* and *surface forces*.

Body forces act on all particles in a body and are caused by some external body. This force is measured as a force per unit mass or per unit volume at the present location of a point in the continuum.

Surface forces are contact forces that act across a surface of the body. The intensity of a force acting on a unit area on the surface is also known as *traction* or *stress vector*.

2.2.3 Kinematics from an Eulerian and Lagrangean point of view

The *referential description* (also: *material description*) is mostly attributed to Lagrange[1], while the *spatial description* is usually attributed to Euler[2], although both were aware of either description. There may be a distinction in up to four descriptions [Malv69], but only the aforementioned ones shall be described here:

In the referential description, the velocity vector $\mathbf{u}$ of a particle depends on the coordinates in the reference configuration X_1, X_2, X_3 (the material coordinates in three dimensions) and the time t:

$$\mathbf{u} = \mathbf{u}(X_1, X_2, X_3, t). \tag{2.2}$$

The spatial description fixes the observers position, such that $\mathbf{u}$ may be expressed as function of the spatial coordinates x_1, x_2 and x_3 and time t:

$$\mathbf{u} = \mathbf{u}(x_1, x_2, x_3, t), \tag{2.3}$$

where the spatial coordinates depend on the material coordinates as

$$\mathbf{x} = \mathbf{x}(\mathbf{X}, t), \tag{2.4}$$

[1] Joseph Louis Lagrange($\star$ 1736, † 1813) [2] Leonhard Euler, ($\star$ 1707, † 1783)

with $\mathbf{x}(\mathbf{X}, t_0) = \mathbf{X} = (X_1, X_2, X_3)^T$ at the reference time t_0.

The *material time derivative*(also: *substantial time derivative*) $\frac{\mathrm{D}}{\mathrm{D}t}$ is the time rate of change of a quantity. Applied to velocity, we get an expression of the acceleration. In Lagrangian description it is equivalent to the total derivative operator

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t}(\mathbf{X}, t) = \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}(\mathbf{X}, t). \tag{2.5}$$

For the spatial description we get by (2.4) and the chain rule

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t}(\mathbf{x}, t) = \frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{u}}{\partial \mathbf{x}}\frac{\partial \mathbf{x}}{\partial t} = \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u}, \tag{2.6}$$

an acceleration consisting of a local time derivative and a spatial rate of change.

A connection between the referential and spacial description is given by the *Reynolds*[3] *transport theorem*, that is given here without proof:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_\Omega f(\mathbf{x}, t)\,\mathrm{d}\mathbf{x} = \int_\Omega \frac{\partial}{\partial t} f(\mathbf{x}, t) + (\nabla \cdot f(\mathbf{x}, t))\mathbf{u}\,\mathrm{d}\mathbf{x}, \tag{2.7}$$

where f is a general space-time dependent field and $\mathbf{u}$ is the velocity of movement of the domain Ω.

2.3 Governing equations

2.3.1 Fluid motion

For the derivation of the Navier-Stokes equations describing incompressible flow in a Newtonian fluid, we consider a subdomain ω of Ω_1 with boundary γ. This element of fluid moves with the flow, i.e. the referential description is used here.
As we consider an incompressible fluid, i.e. a fluid for which the volume of any element of fluid is time invariant, it holds for ω, that

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_\omega \mathrm{d}\mathbf{x} = 0. \tag{2.8}$$

By applying (2.7), we can rewrite this into

$$0 = \frac{\mathrm{d}}{\mathrm{d}t}\int_\omega \mathrm{d}\mathbf{x} = \int_\omega \nabla \cdot \mathbf{u}_f\,\mathrm{d}\mathbf{x},$$

with $\mathbf{u}_f$ being the fluid velocity with respect to ω.

As this holds for any ω, we get the following *incompressibility condition*:

$$\nabla \cdot \mathbf{u}_f = 0. \tag{2.9}$$

[3] Osborne Reynolds($\star$ 1842, † 1912)

Moving on, we now apply Newton's second law (2.1)

$$\frac{\mathrm{d}}{\mathrm{d}t} M\mathbf{u}_f = \mathbf{F}$$

to the same element ω.
Since the total mass of the fluid is $\int_\omega \rho_f \,\mathrm{d}\mathbf{x}$, we can expand this to

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_\omega \rho_f \mathbf{u}_f \,\mathrm{d}\mathbf{x} = \int_\omega \rho_f \mathbf{f} \,\mathrm{d}\mathbf{x} + \int_\gamma \mathbf{S} \,\mathrm{d}s. \tag{2.10}$$

where $\mathbf{f}$ denotes a density of volume forces per mass unit and $\mathbf{S}$ denotes a density of surface forces per surface unit. Assuming that the only external force acting on our system is gravity, we can set $\mathbf{f} = \mathbf{g}$. On the other hand, as the influence of gravity only depends on the density of the phase, we can omit the gravity term for the fluid and apply the gravity force only for the particles in correspondence with their density difference compared to the fluid phase.

It follows from, e.g., [GuHP91] that $\mathbf{S}$ takes the form

$$\mathbf{S} = \boldsymbol{\sigma}\mathbf{n}, \tag{2.11}$$

where $\boldsymbol{\sigma}$ is the symmetric *stress tensor* that acts on the outward *unit normal vector* $\mathbf{n}$. This action can be broken down further into *inviscid* and *viscous stress*, such that

$$\boldsymbol{\sigma} = -p\mathbf{I} + \tau, \tag{2.12}$$

where p is the *pressure*, $\mathbf{I}$ the *identity tensor* and $\boldsymbol{\tau}$ the *viscous stress tensor*, which in the case of a Newtonian fluid takes the form

$$\boldsymbol{\tau} = \mu(\nabla\mathbf{u}_f + (\nabla\mathbf{u}_f)^T), \tag{2.13}$$

with μ being the constant *viscosity coefficient*.

Going on and inserting (2.11) into (2.10) yields

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_\omega \rho_f \mathbf{u}_f \,\mathrm{d}\mathbf{x} = \int_\gamma \boldsymbol{\sigma} \cdot \,\mathrm{d}s.$$

Applying the divergence theorem on the right side and the transport theorem (2.7) on the left, we obtain

$$\int_\omega \frac{\partial}{\partial t} (\rho_f \mathbf{u}_f) + (\nabla \cdot \rho_f \mathbf{u}_f)\, \mathbf{u}_f \,\mathrm{d}\mathbf{x} = \int_\omega \nabla \cdot \boldsymbol{\sigma} \,\mathrm{d}\mathbf{x}.$$

Since only an incompressible fluid is considered, ρ_f is constant and can be placed in front:

$$\int_\omega \rho_f \left(\frac{\partial \mathbf{u}_f}{\partial t} + (\nabla \cdot \mathbf{u}_f)\, \mathbf{u}_f \right) \mathrm{d}\mathbf{x} = \int_\omega \nabla \cdot \boldsymbol{\sigma} \,\mathrm{d}\mathbf{x}.$$

Replacing the term in parentheses by the material time derivative and inserting (2.12) and (2.13) result in

$$\int_\omega \rho_f \frac{\mathrm{D}\mathbf{u}_f}{\mathrm{D}t}\,\mathrm{d}\mathbf{x} = \int_\omega -\nabla p + \nabla\cdot(\mu(\nabla\mathbf{u}_f + (\nabla\mathbf{u}_f)^T))\,\mathrm{d}\mathbf{x}.$$

We then use that $\nabla\cdot(\nabla\mathbf{u})^T = \nabla(\nabla\cdot\mathbf{u}) = \nabla\cdot 0 = 0$ and get by rearranging

$$\int_\omega \rho_f \frac{\mathrm{D}\mathbf{u}_f}{\mathrm{D}t} + \nabla p - \mu\nabla^2\mathbf{u}_f)\,\mathrm{d}\mathbf{x}.$$

As ω was chosen arbitrarily, this implies the following *momentum equation* (2.14a), that, together with (2.9), forms the *Navier-Stokes* equations for the considered fluid:

$$\begin{cases} \rho_f \dfrac{\mathrm{D}\mathbf{u}_f}{\mathrm{D}t} + \nabla p - \mu\nabla^2\mathbf{u}_f = 0 & \text{in } \Omega_1, \quad (2.14a)\\ \nabla\cdot\mathbf{u}_f = 0 & \text{in } \Omega_1. \quad (2.14b) \end{cases}$$

2.3.2 Rigid body motion

Let $\mathbf{X}_i$ be the linear position of the center of mass of the ith particle and $\boldsymbol{\theta}_i$ its angular position. Then the translational velocity $\mathbf{U}_i$ of the ith particle and its angular velocity $\boldsymbol{\omega}_i$ are obtained by the following kinematic equations

$$\begin{cases} \dfrac{\mathrm{d}\mathbf{X}_i}{\mathrm{d}t} = \mathbf{U}_i, & (2.15a)\\ \dfrac{\mathrm{d}\boldsymbol{\theta}_i}{\mathrm{d}t} = \boldsymbol{\omega}_i. & (2.15b) \end{cases}$$

As we are only considering rigid bodies that undergo no deformation, the whole motion of any point $\mathbf{X}\in\Omega_{2,i}$ can be described by

$$\mathbf{u}_i(\mathbf{x}) = \mathbf{U}_i + \boldsymbol{\omega}_i\times(\mathbf{x}-\mathbf{X}_i). \quad (2.16)$$

In order to analyze the behavior within the system, we start again with Newton's second law:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega_{2,i}} \rho_i\mathbf{u}_i\,\mathrm{d}\mathbf{x} = \int_{\Omega_{2,i}} \rho_i\mathbf{f}\,\mathrm{d}\mathbf{x} + \int_\gamma \mathbf{S}\,\mathrm{d}s.$$

The body forces $\mathbf{f}$ consist not only of gravity $\mathbf{g}$ any more, but also of forces $\mathbf{c}$ acting in the case of collisions. As stated before, we only consider the gravity to be acting according to the mass difference between the particle and a fluid of the same volume. Hence we replace the scaling factor ρ_i by $\rho_i - \rho_f$.

The surface forces $\mathbf{S}$ are now hydrodynamic interaction forces with the surrounding fluid, that can be set $\mathbf{S} = \boldsymbol{\sigma}\mathbf{n}$ as the fluid was considered to be Newtonian. The particles are assumed to be smooth, such that there are no tangential collision forces acting on them. Transforming the derivative as before, the equation reads now

$$\int_{\Omega_{2,i}} \rho_i\frac{\mathrm{D}\mathbf{u}_i}{\mathrm{D}t}\,\mathrm{d}\mathbf{x} = \int_{\Omega_{2,i}} (\rho_i-\rho_f)\mathbf{g} + \mathbf{c} + \nabla\cdot\boldsymbol{\sigma}\,\mathrm{d}\mathbf{x}. \quad (2.17)$$

The equation can be further simplified by inserting (2.16), what will be done in a later step.

2.3.3 Initial and boundary conditions

The problem requires an initial condition for the velocities, e.g. in the form of

$$\begin{cases} \mathbf{u}_f|_{t=0} = \mathbf{u}_{f,0} & \text{in } \Omega_1 \quad (2.18a) \\ \mathbf{U}_i|_{t=0} = \mathbf{U}_{i,0} & \text{in } \Omega_{2,i} \quad (2.18b) \\ \boldsymbol{\omega}_i|_{t=0} = \boldsymbol{\omega}_{i,0} & \text{in } \Omega_{2,i}. \quad (2.18c) \end{cases}$$

For the external boundary, only homogeneous Dirichlet conditions are required,

$$\mathbf{u}_f|_\Gamma = 0,$$

while on the internal boundary Σ *no-slip conditions* for the velocity are presumed, i.e. there is no difference in the particles' and the fluid's velocity at the boundary:

$$\mathbf{u}_f = \mathbf{u}_i = \mathbf{U}_i + \boldsymbol{\omega}_i \times (\mathbf{x} - \mathbf{X}_i) \quad \text{on } \partial\Omega_{2,i} \tag{2.19}$$

2.3.4 Collision handling

The accurate modeling of collisions is a very complicated problem. When two bodies approach each other, a thin fluid film is formed in between them that introduces a new spatial scale related to the film thickness. This scale can only be resolved by enormous grid refinement in space and time, what is currently incompatible with the computational possibilities of the hardware.

One implementable technique is given by *sub-grid modeling*, which results in a sort of lubrication force. The following models are taken from [VeMN07] and distinguish between particle-particle and particle-wall collisions.

- **Particle-particle collision** If $\mathbf{X}_i$ and $\mathbf{X}_j$ are the centroidal coordinates of the ith and jth particle and r_i and r_j their respective radii, then a collision took place if for the updated positions holds:

  $$\varepsilon \leq s_{i,j} := |\mathbf{X}_i - \mathbf{X}_j| - (r_i + r_j),$$

  where ε is the minimum separation distance, i.e. the maximum allowed thickness of the film between any two particles. In this case, the particles are moved alongside the line connecting the centroids at distances depending on the masses of the particles involved:

  $$\Delta r_i = \frac{M_j(\varepsilon - s_{i,j})}{M_i + M_j}, \quad \Delta r_j = \frac{M_i(\varepsilon - s_{i,j})}{M_i + M_j}$$

  This correction is equivalent to a correction of the velocity given by

  $$\Delta\mathbf{U}_i = \frac{\Delta r_i}{\delta t}\frac{\mathbf{X}_j - \mathbf{X}_i}{|\mathbf{X}_j - \mathbf{X}_i|}, \quad \Delta\mathbf{U}_j = \frac{\Delta r_j}{\delta t}\frac{\mathbf{X}_i - \mathbf{X}_j}{|\mathbf{X}_i - \mathbf{X}_j|} \tag{2.20}$$

  or the introduction of an additional force $\mathbf{c}_{i,j}$ in the particle's momentum equation.

- **Particle-wall collision** The *lubrication force* is calculated as follows

$$\mathbf{c}_i = \begin{cases} -6\pi r_i \mathbf{U}_{i,\perp} \mu \left(\frac{r_i}{s_i} - \frac{r_i}{h} \right) & \text{if } s_i < h \\ 0 & \text{otherwise} \end{cases} \tag{2.21}$$

  where $\mathbf{U}_{i,\perp}$ is the velocity component perpendicular to the wall, h is the grid size and s_i is the gap between the wall and the particle. This force is used to correct the particles' velocity in the rigid body substep.

2.3.5 Model improvements

The developed model can be seen as the most basic one, that has to undergo extensions in order to cover more sophisticated phenomena. This extension is done by including more then the above mentioned forces, e.g. the following ones.

- Drag forces, consisting of friction and form drag.
- The Buoyancy force, that gives rise to an additional pressure force in the direction of the pressure gradient.
- The Basset force arises from the fact that with the acceleration/deceleration of a particle, also a fraction of the surrounding liquid is accelerated/decelerated. The lagging of the boundary layer development on the particle causes this force.
- Slip-shear and slip-rotation lift forces have to be taken into account when the no-slip condition is not assumed.

As mentioned before, the collision model is very basic as well and will have to be extended for higher accuracy in the approximation of the physical phenomena. Also fluid-particle interaction forces have to be taken into account in special situations, among them the Coulomb force or thermophoretic forces.

2.4 Numerical simulation methods

For this model of the presented problem, different numerical solving strategies have been developed, the most important ones are presented in the following survey.

Arbitrary Lagrangean Euler technique

Hu et al.[HuJC92, HuPZ01], Maury [Maur99], and also Galdi and Heuveline [GaHe07] developed a *finite element method* based on unstructured grids for the simulation of rigid particles in Newtonian and viscoelastic fluids.
The governing equations of fluid and particle motion are incorporated into a single coupled variational equation in a way, that the hydrodynamic forces and torques acting on the particles are eliminated. The grid nodes on the particle surface move with the particle, while the nodes inside the fluid domain are computed using Laplace's

equation for a smoothly varying distribution of nodes. When the mesh gets too distorted, a new mesh is generated at each time step and the flow field is projected onto the new mesh.
The main disadvantage of this technique is that in the three-dimensional case efficient body-fitted grid generation methods are not yet satisfyingly available.

Fictitious domain method using distributed Lagrange multipliers

Glowinski et al. [GPHJ99, Glow03] proposed a fictitious domain method for the direct simulation of rigid particles in fluids. In this method the whole domain Ω is considered as being a fluid and where the particle domain $\Omega 2$ is constrained to move with rigid body motion.
In the aforementioned references, this constraint is implied by a , accompanying again a combined variational equation for the fluid-particle motion where the mutual forces cancel. The result is an implicit scheme that allows the usage of a fixed structured grid, thus eliminating the need for re-meshing. For the solution of the Lagrange multipliers, one for each particle, sub-grids around the particles are needed.

Fictitious domain method using a global Lagrange multiplier

Minev, Nandakumar et al. [DGMN03] reviewed the scheme of Glowinski and proposed to approximate both, the fluid velocity field and the Lagrange multiplier for imposition of the rigid body motion on the same fixed Eulerian grid. This way the rigid particles don't have to be gridded separately and a higher simulation speed is achieved.

Fictitious Domain Method without a Lagrange multiplier

In [VeMN07], the previous scheme was further modified, such that the rigid body motion is determined by an integral equation that can be approximated by direct extrapolation. This leads to a fully explicit scheme with respect to the rigid body constraint.

Fictitious boundary method

This method was originally presented by Turek, Wan and Rivkind [WaRT02] and advanced in further publications. It is a *multi-grid finite element method*, that extends the fluid domain into the particle domain as well, and introduces boundary conditions at the interface Σ between particles and fluid. Based on an unstructured coarse mesh, the large scale structures are described by a rough boundary parametrization. All fine-scale features are then treated as interior objects, such that the corresponding components in all resulting matrices and vectors are unknown degrees of freedom and are incorporated into the solution steps. No mesh adaptation is needed here.
In [WaTu07], this method is combined with a moving mesh method, in order to improve the accuracy for capturing the surfaces of the rigid particles, that had been only of first order before. Starting with an arbitrary block structured grid, the grid spacing is changed such that the grid points are concentrated near the surface of the rigid particles. In every time step, such a deformed grid has to be generated by the cost of solving additional linear Poisson problems.

Other methods

For completeness a number of alternative methods shall be mentioned here: *large eddy simulations* and methods solving the *Reynolds-averaged Navier-Stokes equations* are computationally less intensive, but also less accurate. Also the *lattice Boltzmann methods* do not solve the original Navier-Stokes equations but the discrete Boltzmann equation. Furthermore there have been approaches based on Stokesian dynamics simulations, and statistical approaches using particle probability density functions.

2.5 Fictitious domain formulation

As the goals of this thesis include the fast simulation of a high number of particles, a method based on fixed uniform grids is to be preferred. The fictitious domain methods proposed hold this property, where [DGMN03] and [VeMN07] promise faster computation speeds by avoiding an individual treatment of the particles as in [GPHJ99] (neglecting collisions). The following derivation of a fictitious domain method suiting the purpose of this thesis follows thus rather the process sketched in [DGMN03] and [VeMN07].

The preparations for applying this method have already been done by deriving momentum equations for both the fluid and the particles.
The fluid momentum equation (2.14a) read

$$\rho_f \frac{\mathrm{D}\mathbf{u}_f}{\mathrm{D}t} = \nabla \cdot \boldsymbol{\sigma} \quad \text{in } \Omega_1,$$

while the rigid motion was derived in (2.17) as

$$\int_{\Omega_{2,i}} \rho_i \frac{\mathrm{D}\mathbf{u}_i}{\mathrm{D}t} \,\mathrm{d}\mathbf{x} = \int_{\Omega_{2,i}} (\rho_i - \rho_f)\mathbf{g} + \mathbf{c} + \nabla \cdot \boldsymbol{\sigma} \,\mathrm{d}\mathbf{x} \quad \text{in } \Omega_{2,i}.$$

The whole domain Ω is now assumed to be filled with a fluid and an additional force $\mathbf{F}$ is inserted in order to model the interaction of the fluid and the particles.

$$\rho_f \frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = \nabla \cdot \boldsymbol{\sigma} + \mathbf{F} \quad \text{in } \Omega. \tag{2.22}$$

According to Newton's third law of motion, to every action, there is an equal and opposite reaction. In this context, the particles have to summon up an opposing force to the fluid and $\mathbf{F}$ becomes

$$\mathbf{F} = \begin{cases} 0 & \text{in } \Omega_1 \\ \rho_f \dfrac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} - \nabla \cdot \boldsymbol{\sigma} & \text{in } \Omega_{2,i}. \end{cases}$$

Using this notation, the particles' moments can be written as

$$\int_{\Omega_{2,i}} \rho_i \frac{\mathrm{D}\mathbf{u}_i}{\mathrm{D}t} - \rho_f \frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} \,\mathrm{d}\mathbf{x} = \int_{\Omega_{2,i}} (\rho_i - \rho_f)\mathbf{g} + \mathbf{c} - \mathbf{F} \,\mathrm{d}\mathbf{x} \quad \text{in } \Omega_{2,i}.$$

As $\Omega_{2,i}$, the sub-domains of the enlarged fluid velocity field occupied by particles, shall behave like rigid bodies, we set

$$\mathbf{u} = \mathbf{u}_i \quad \text{in } \Omega_{2,i}, \tag{2.23}$$

and obtain

$$\int_{\Omega_{2,i}} (\rho_i - \rho_f) \frac{\mathrm{D}\mathbf{u}_i}{\mathrm{D}t} \,\mathrm{d}\mathbf{x} = \int_{\Omega_{2,i}} (\rho_i - \rho_f)\mathbf{g} + \mathbf{c} - \mathbf{F} \,\mathrm{d}\mathbf{x} \quad \text{in } \Omega_{2,i}.$$

When inserting (2.16), the derivative of the angular velocity term vanishes and we get

$$\int_{\Omega_{2,i}} (\rho_i - \rho_f) \frac{\mathrm{D}\mathbf{U}_i}{\mathrm{D}t} \,\mathrm{d}\mathbf{x} = \int_{\Omega_{2,i}} (\rho_i - \rho_f)\mathbf{g} + \mathbf{c} - \mathbf{F} \,\mathrm{d}\mathbf{x} \quad \text{in } \Omega_{2,i}.$$

This can be further simplified by noting that $\mathbf{U}_i, \mathbf{g}$ and $\mathbf{c}$ are constant on $\Omega_{2,i}$ and by further introducing the mass difference $\Delta M_i = \int_{\Omega_{2,i}} \rho_i - \rho_f \,\mathrm{d}\mathbf{x}$, and V_i for the ith particle's volume:

$$\Delta M_i \frac{\mathrm{d}\mathbf{U}_i}{\mathrm{d}t} = \Delta M_i \mathbf{g} + V_i \mathbf{c} - \int_{\Omega_{2,i}} \mathbf{F} \,\mathrm{d}\mathbf{x} \quad \text{in } \Omega_{2,i}. \tag{2.24}$$

Of course, the incompressibility condition (2.9) holds also in $\Omega_{2,i}$, what implies

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega. \tag{2.25}$$

Thus (2.22), (2.24) and (2.25) already yield equations for $\mathbf{u}$, p and $\mathbf{U}_i$.

An equations for $\mathbf{X}_i$ can easily be obtained by (2.15a)

$$\frac{\mathrm{d}\mathbf{X}_i}{\mathrm{d}t} = \mathbf{U}_i \quad i = 1, \ldots, n \tag{2.26}$$

and $\boldsymbol{\omega}_i$ can be recovered from the no-slip condition (2.19) as follows.

$$\begin{aligned}
&\mathbf{u} = \mathbf{U}_i + \boldsymbol{\omega}_i \times (\mathbf{x} - \mathbf{X}_i) \quad \text{on } \partial\Omega_{2,i} \quad i = 1, \ldots, n \\
&\implies \int_{\partial\Omega_{2,i}} \mathbf{u} \times \mathbf{n} \,\mathrm{d}s = \int_{\partial\Omega_{2,i}} (\mathbf{U}_i + \boldsymbol{\omega}_i \times (\mathbf{x} - \mathbf{X}_i)) \times \mathbf{n} \,\mathrm{d}s \quad i = 1, \ldots, n \\
&\overset{\text{Stokes}}{\iff} \int_{\Omega_{2,i}} \nabla \times \mathbf{u} \,\mathrm{d}\mathbf{x} = \int_{\Omega_{2,i}} \underbrace{\nabla \times (\mathbf{U}_i + \boldsymbol{\omega}_i \times (\mathbf{x} - \mathbf{X}_i))}_{Vorticity} \,\mathrm{d}\mathbf{x} \quad i = 1, \ldots, n \\
&\iff \int_{\Omega_{2,i}} \nabla \times \mathbf{u} \,\mathrm{d}\mathbf{x} = 2\,\boldsymbol{\omega}_i V_i \quad i = 1, \ldots, n
\end{aligned} \tag{2.27}$$

These equations (2.22)-(2.27) fully describe the model above. Still missing is a formulations of the interaction force $\mathbf{F}$. It is determined by the condition (2.23), but not directly contained in it. Thus in [DGMN03] it was imposed via a Lagrange multiplier, but in [VeMN07] an explicit equation was derived during discretization. As the discretization step is postponed after the hardware discussion, the specific calculation of the fluid-particle interaction will be covered likewise in chapter 5.

2.6 Summary

A model of particulate flows based on the continuum hypothesis leads to the Navier-Stokes equations as a description of the fluid flow behavior and a similar momentum equation for the rigid body motion.

The general idea of the fictitious domain formulation is to extend the geometrically complex fluid domain Ω_1 into the domain occupied by the particles Ω_2 and obtaining a simpler domain Ω.

The enlargement of the fluid domain is done by introducing an interaction force that acts differently on Ω_1 and Ω_2 such that the behavior of the whole domain Ω can be described by a single equation depending on this force.

The main advantage over ALE and fictitious boundary methods is, that the domain Ω is not time-dependent anymore, such that a fixed mesh and fast direct solvers can be applied, though with the backdraw of limited resolution on the particles' surfaces.

3. Heterogeneous computing, GPGPU and CUDA

Heterogeneous, composed of the Greek words heteros ($\acute{\varepsilon}\tau\varepsilon\rho o\varsigma$) and genos ($\gamma\acute{\varepsilon}\nu o\varsigma$), means to consist of elements or parts of dissimilar kind.

As frequency scaling is physically constrained by current leakage, leading to heat generation and high power consumption, other strategies for improving computer performance have to be pursued. Meanwhile *multi-core* systems are de-facto standard with currently up to quad-core CPUs, but the complexity and size of these general purpose cores limits further scaling. An approach used since the early days in system architecture is a heterogeneous concept that delegates work to special purpose *co-processors* (cf. peripheral co-processors of CDC 6600, the first "supercomputer", 1964).

This chapter sets modern *graphics processing units (GPUs)* into the context of heterogeneous system architectures, explaining their pro's and con's. The second part will deal with general-purpose computing on graphics processing units (GPGPU), its history and current capabilities. The particular hard- and software used in this thesis, a NVIDIA GeForce GT200 GPU with CUDA, will be explained in detail in a third part.

3.1 Approaches to heterogeneous computing

All solutions presented here, besides the Cell processor, are *accelerators* complementing standard microprocessors. They all promise enormous single thread peak performance, but differ widely in their applicability in science, their power consumption and programmability. Interesting characteristics will thus be the overall performance of typical scientific kernels, the price/performance and watt/performance ratios as well as the development expenditure. Further interesting issues are IEEE 754 compliance, ECC and memory coherency:

IEEE 754, the IEEE standard for binary floating-point arithmetic[IEEE85] defines number formats, special values (e.g. NaN and infinity), rounding rules and the handling of exception conditions for typical floating-point numbers. Non complying accelerators will hence produce other results as on conventional CPUs, that are IEEE 754 compliant.

Error correction code (ECC) mechanisms detect and correct transient errors as they appear in every electronic device. ECC gains importance with the size of the system: The probability of such an error increases linearly with the number of components. Most of the following accelerator were originally not designed for the use in clusters and thus do not provide adequate ECC mechanisms. This can lead to non-deterministic behavior and incorrect results.

Commercially available servers maintain *cache coherency* across all processors. Currently accelerators cannot provide the same features, but efforts are made to achieve coherency in the future. For example, a cache coherent HyperTransport[1] version (AMD) or extensions to PCIe (Intel, IBM) could be future connectivity solution.

3.1.1 MDGRAPE-3

MDGRAPE-3 is a high performance computer, developed by the RIKEN research institute and the University of Tokyo for the simulation of molecular dynamics.[Taij03] The idea behind this system is to outsource the force calculation as it dominates the computational time. For this purpose 256 dual-core Intel Xeon host computers have been complemented with 24 MDGRAPE-3 chips each via a PCI-X interface (133 MHz, max. 10 GBit/s). Each chip houses 20 pipelines for force calculation (33 operations) and a local memory for 32,738 particles. The pipelines are capable of efficiently calculating two-body forces F_i by $\sum_j \alpha g\left(\beta \left|r_i - r_j\right|\right)\left(r_i - r_j\right)$, where $g(x)$ is an arbitrary smooth function. Figure 3.1 shows the hardware implementation with additional parameter input possibilities. The multiplications are done with single precision floating-point arithmetic (SP, 32bit) and the additions partly also in 40 and 80 bit fixed point arithmetic. As all 20 units work simultaneously, one MDGRAPE-3 chip achieves 660 operations/cycle, resulting in 165 GFLOPS at 250 MHz.

The main advantage of the MDGRAPE-3 chip is its comparatively low power consumption of 40 W per two chip board (0.12 W/GFLOPS) and fairly low development costs of 15M USD (15 USD/GFLOPS for the whole system). On the other hand, the applications for this machine are strictly limited to molecular dynamics or similar N-body simulations that furthermore should not be too data-intensive. A 2-Chip board for academic usage is available for 12,000 USD, resulting in 36 USD/GFLOPS.

3.1.2 FPGAs

Field programmable gate arrays, FPGAs, are microchips that are able to form different electronic circuits by reconfiguring internal structures. Invented in 1984, FPGAs have been intensively used in embedded and real-time systems. Meanwhile, even supercomputers like the Cray XD1 or SGI RASC Blades use FPGA technology to

[1] http://www.hypertransport.org

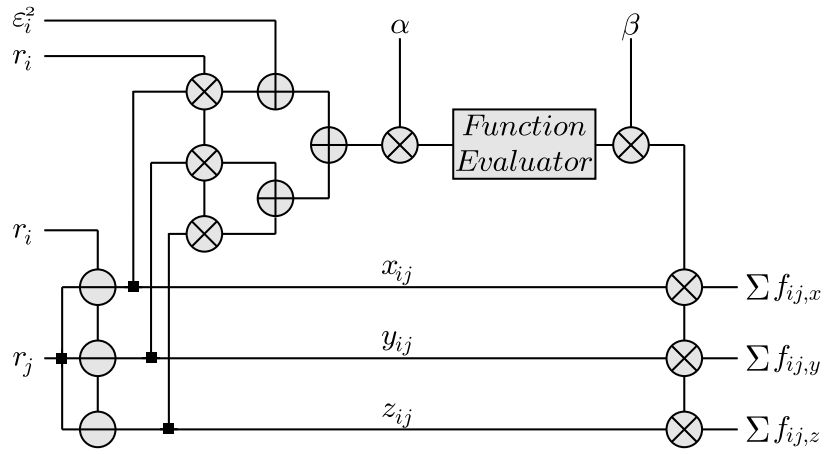

Figure 3.1: MDGRAPE-3 force pipeline

adjust the processor architecture to the need of the respective algorithm. The two above mentioned systems both use Xilinx Virtex-4 FPGAs, that offer up to 200,000 logic cells, programmable to form AND, XOR or more complex functions. Memory is available in form of up to 1.36 Mbit distributed RAM and 10 Mbit block RAM. The peak performance is 15.6 GFLOPS for DGEMM.

It is further possible to (re-)program these FPGAs even during run-time, thus providing *reconfigurable computing* . The programming is either done in traditional hardware description language (HDL) or C-based languages generating HDL code. Still, the complexity and compilation overhead leads to a slower development in comparison to non-mixed platforms: In order to speed up an existing program, the developer has to modify the code in accordance to the FPGA C language, compile to HDL (or Verilog), produce and download a "bitfile" to the FPGA and finally compile the complete application for host and FPGA. Fortran code, as still heavily used in science, is not supported yet.

Although FPGAs provide good results for compact and well defined kernels, space limitations enforce compromises. Though FPGAs usually implement data representations according to IEEE standards, the edge condition handling is often not conforming. FPGAs could theoretically implement IEEE 754 completely, but would need to sacrifice large chip space and possibly loose their performance advantage.

As it was the case with the above mentioned non-reconfigurable MDGRAPE-3 chip, FPGAs benefit from pipelining and a high number of operations per cycle. Hence, the best performance is achieved for integer operations; 64bit floating-point operations on the other hand just achieve a fraction of the 32 bit FP performance. Double precision multipliers need four times the hardware of single precision multipliers and much longer computation time.[SuPS08] While the watt/performance ratio is generally good (a FPGAs needs usually 10 to 25 W), the performance/price ratio depends on the application. Simple FPGAs with small block RAM sizes and few logic cells are cheap but not versatile, more complex chips however quickly get very expensive.

3.1.3 ClearSpeed accelerator cards

ClearSpeed offers accelerator cards for servers and standard PCs with PCIe Slot. The latest card is called "Advance e710" and includes one CSX700 chip and 2 GByte ECC-DRAM. A CSX 700 consists of two multithreaded array processors (MTPA), housing a mono (scalar) and a poly (parallel) execution unit, whereby the latter consists of 96 processing elements (PEs). The basic structure of a MTPA is shown in figure 3.2. While the mono execution unit only has a floating-point unit (FP unit), the processing elements are equipped with integer arithmetic logical units (ALUs) and integer multiply-accumulate (MAC) units.[Clea07] The peak performance of this chip is 96 GFLOPS in double precision (DP, 64bit) and the power consumption 25 W/board.[Clea08] This results in 0.3 W/GFLOPS and, assuming a price of about 3000 USD, 31.25 USD/GFLOPS.

The main advantage over FPGAs is the facilitated software development. Besides a C compiler, debugger and profiler, (Fortran callable) libraries exist that offer straightforward acceleration of selected level 3 BLAS, LAPACK and FFT functions. Even Matlab and Mathematica support is available.

The IEEE 754 and ECC support make ClearSpeed cards interesting for the use in clusters. The TSUBAME cluster of the Tokyo Institute of Technology used 360 Advance 620 chips to achieve 56 TFLOPS in the Linpack benchmark, rank 9 in top500 list as of November 2006.

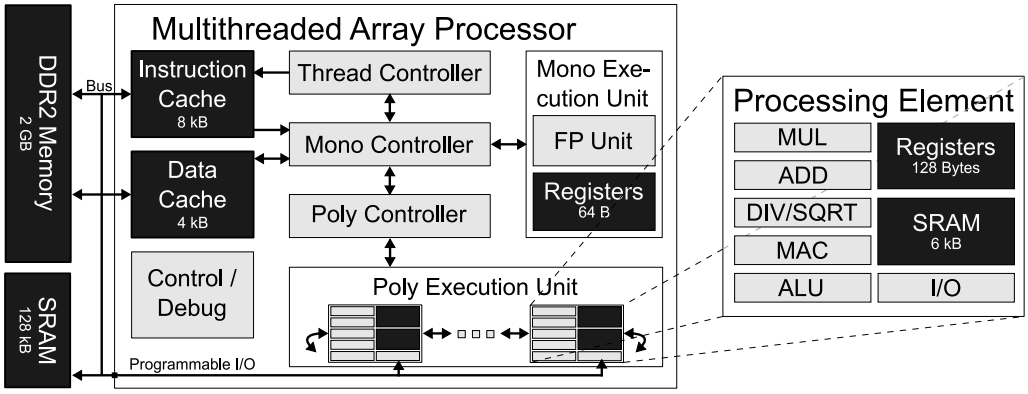

Figure 3.2: Structure of a multithreaded array processor of the CSX700 chip

3.1.4 Graphics cards

The idea of using graphics hardware for general computations is not new and its history is sketched in 3.2. There are currently three mayor producers of graphics cards, ATI (purchased by AMD in November 2006), Intel and NVIDIA. While Intel currently focuses on medium performance on-board graphics, ATI and NVIDIA also offer dedicated hardware and APIs for GPGPU applications. Historically, the strength of GPUs are single precision floating-point operations, currently 933.12 GFLOPS for a NVIDIA GeForce GTX 280 and 1.2 TFLOPS for a AMD Radeon HD 4870 (peak). These new GPU generations can also provide double precision

and integer operations (NVIDIA in hardware, ATI emulated), however with reduced FLOP rates: 77 GFLOPS in double precision for NVIDIA and 200 GFLOPS for AMD. Choosing the single precision performance for calculating the power efficiency, both chips need about 0.2 GFLOPS/W. This might sounds all right, but more than 200 W power consumption makes those chips hardly eligible for e.g. blade servers.

Because of the huge volumes graphics cards are produced in, the chips are fairly cheap. Low budget cards start at about 50 USD, midrange cards are available for 150 USD and up and the newest high performance graphics cards retail at 300 to 500 USD. At the time of introduction, a GTX 280 with 1 GB RAM had a performance/price ration of 0.5 USD/GFLOPS, a HD 4870 with 512 MB RAM 0.3 USD/GFLOPS.
In addition to high performance consumer cards that are GPGPU capable, both vendors offer dedicated versions for *high performance computing (HPC)*. These are equipped with quadrupled memory (NVIDIA TESLA/ AMD FIRESTREAM) and do not have a video output anymore. Although their prices are much higher, these solutions do not offer ECC yet and AMD's chips don't follow IEEE 754 completely. At least NVIDIA fully implements IEEE 754R for double precision, a revised version of IEEE 754 from 2006.

The programming can be done with conventional graphics languages like OpenGL, but both NVIDIA and AMD offer more general programming models for their chips too. AMD's Close to Metal (CTM) is a quite low level API, that can be used with Brook+, a variation of the higher level GPU programming language Brook developed at the University of Stanford. Brook+ is still near to *stream processing* (vectors of variable length) and requires major code changes. [Adva08] NVIDIA's solution is called "Compute Unified Device Architecture" (CUDA), an extension to the C programming language that is not as restrictive as Brook+ (cf. 3.3.2).[NVID08b]

3.1.5 Cell processor

The original Cell processor as designed for the Sony PlayStation 3 by Sony, Toshiba and IBM, has quickly drawn interest of scientists as it offers chip level heterogeneity: One IBM PowerPC processing element controls 8 simpler synergistic processing elements (SPEs), as shown in figure 3.3.[CRDI07] Each SPEs houses a 256 kB local store that is software controlled by DMA and not managed like a usual cache. It fetches data asynchronously from DRAM, hereby hiding memory latency. This approach uses the available memory bandwidth more efficiently then ordinary prefetch techniques but makes the programming more complex. The SPEs have two parallel pipelines, one issuing computational instructions, the other issuing loads, stores, permutes and branches. The SPEs underwent revision for the use in HPC, improving most notably their double precision performance. The result is named IBM PowerXCell 8i.[HMZZ+08]

The second processor generation has a single precision peak performance of 200 GFLOPS and 100 GFLOPS in double precision, resulting in 1.84 GFLOPS/W and 25 USD/GFLOP for single precision (IBM BladeCenter QS22, 8 GB RAM). ECC is only supported for external memory and the SPEs deviate from IEEE 754 (e.g. only

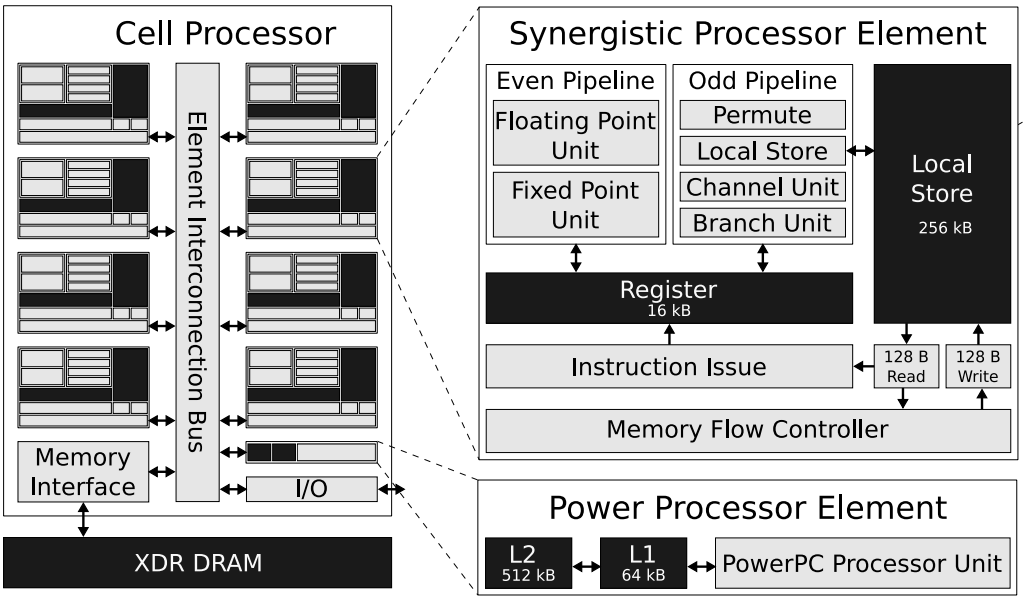

Figure 3.3: Cell processor architecture

round to zero is supported)[IBM 08]. The main advantage of this heterogeneous chip are fast memory transfers, avoiding PCIe or the even slower PCI-X as bottleneck. The main disadvantage is the more time consuming programming.

3.1.6 Comparison

One can coarsely divide the above described accelerators in two groups: Highly specialized chips for well-defined tasks (FPGAs, MDGRAPE-3) and chips that claim to be beneficial for pretty much any task in HPC (ClearSpeed, GPUs, Cell). The second group's advancements in double precision and IEEE-754 support make them especially interesting for scientific applications.

GPUs and ClearSpeed cards struggle with data-intensive kernels, but are fairly easy to program. ClearSpeed cards need very low power in direct comparison, but GPUs are unbeatably cheap though offering enormous single precision performance. Cell's great advantage is its fast memory access, but the second generation processor is not as cheaply available as the PlayStation 3 version was.

Table 3.1 gives an overview of the pro's and con's of the discussed hardware platforms. The usability does not only depend on the purpose but also on the computing environment. Large scale production clusters depend on ECC and a high mean time between failures (MTBF). Thermal issues as they can arise with graphics cards are thus problematic. But for turning a standard workstation into a teraFLOPS (SP, theor.) computer, there is no cheaper way then installing a modern graphics card.

3.2 GPGPU

General purpose computation on graphics processing units depicts calculations done on a GPU, whose primary purpose is not video output. As early as 1990, GPUs

	MDGRAPE	FPGA	ClearSpeed	GPU	Cell
Specialized in	Force calculation	Integer, $\leq$ SP	DP	Graphics, SP, DP	Graphics, SP, DP
DP Performance	N/A	Low	High	High	High
IEEE 754	No	Expensive	Yes	In parts	In parts
ECC	No	No	Yes	No	In parts
Programming	Easy	Hard	Easy	Easy	Medium
Power	Low	Low	Low	High	Medium
Price	High	Medium	High	Low	High

Table 3.1: Overview of heterogeneous computing technologies

could be used for robot motion planning[LRDG90], long before 3D capabilities were introduces. Another technique from 2001 renders two textures in an ingenious fashion on a cube, such that the use of particular parameters for perspective and texture overlay yields the matrix product of the textures' 8 bit color values.[LaMc01] A texture size of 1024×1024 pixels reached in this way more then 4 GBOPS (byte operations per second). Because of this inconvenient programming, the beginning of GPGPU is rather connected to the introduction of the fully programmable *graphics pipeline* also in 2001[OLGH+07].

The problems GPGPU deals with, were originally strongly connected to computer graphic and image processing. With the introduction of full floating-point support, scientific projects started implementing finite difference and finite element techniques for the solution of systems of partial differential equations (PDEs). Several papers in 2003 demonstrated e.g. solutions of the Navier-Stokes equations for incompressible fluid flow on the GPU [BFGS03, KrWe03] or for boundary value problems [GLLS03]. Meanwhile, even industrial applications using GPUs start emerging.

3.2.1 The graphics pipe line

The graphics pipe line uses several stages to creates a visual image out of the description of a scene. Its structure changed continuously, as it reflected the hardware's makeup. Originally, every stage was implemented in hardware and could only be adjusted by a few parameters in video memory. When some of the elements of the already parallelized texture mapping unit got its own instructions set, it was practically turned into a primitive *SIMD* (single instruction multiple data, according to Flynn's taxonomy [Flyn66]) processor array. These components where called *pixel shaders*, as they were originally meant to enhance the possibilities of rendering a texture onto an object. Soon this programmable shader concept was also transferred to the input stage of the pipeline, introducing *vertex shaders*.

Figure 3.4 shows the graphics pipeline as specified by Direct3D 10, a Microsoft API for 3D graphics, that furthermore supports *geometry shaders* after the vertex shaders.[Micr08] The input assembler supplies triangles, lines and points, that the vertex shaders process vertex wise (transformations, lighting, etc.). The geometry

shaders works on whole primitives again (triangles, lines, points), where the original primitive might be emitted and one or more others be newly created. The resulting data can now be given to the *rasterizer* or be streamed back into memory. From the memory it could be passed to the CPU or recirculated back into the pipeline as input data.
The rasterizer is not programmable, its task is to transform the objects into screened fragments, consisting of numerous pixels. These pixels are subsequently treated by the pixel shaders, generating amongst others the pixels' color values. The output merger finally combines all output data in order to generate the pipeline result.

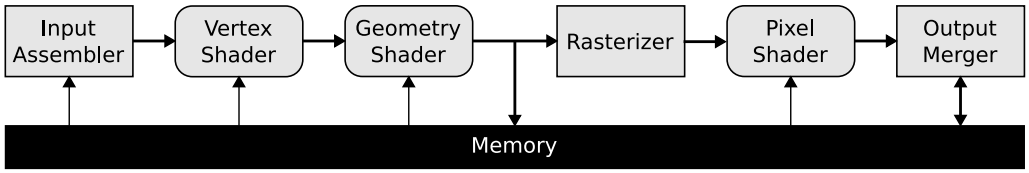

Figure 3.4: Direct3D 10 programmable pipeline

3.2.2 Stream processing with GPUs

Stream processing is a programming paradigm that facilitates eligible computational problems in hard- and software: The same series of operations is applied to every element of a set of data (*stream*). Hence, the operations can be pipelined and data locality can be exploited to minimize memory latency. The best results are achieved if data dependency and data re-use are low, such that many stream processors can process the stream in parallel.

GPUs are perfectly suited for stream processing as the shaders of figure 3.4 are nothing but independent groups of many stream processors, working on different streams within graphics processing, e.g. the sets of vertices and pixels. In the case of GPGPU, a stream is rather a data set within the texture memory, whose entries shall be processed by a shader program. Before the introduction of *unified shaders* that can fulfill any of the different shaders' tasks, the interaction between vertex and pixel shaders was quite complicated. In most cases, only the pixel shaders were used, as they provided a higher number of stream processors and vertex shaders had no memory access in the beginning. At that time, for every element in the stream, a pixel had to be created and a primitive 2D object that got mapped with the pixels as texture. The pixel shaders processed this data and wrote the result into their registers, that were later placed in the *frame buffer*. A major disadvantage of stream processing gets visible here: Data can only be written to and fetched from this buffer in a continuous fashion, data scattering is not possible. Even in Direct3D 10, that allows a write to memory before the rasterizer, data can still only be written as stream. On the other hand, GPUs extend the stream processing model by allowing data gathering, i.e. reading of scattered data.

The scattering problem has been addressed with the development of CUDA, a stream processing independent programming language for NVIDIA GPUs. In fact, the

whole graphics pipeline gets uninteresting for the programmer. This is a logical progression, as the stream processors of all shaders are meanwhile built identically, are able to communicate with each other and have unrestricted memory access.

3.3 CUDA

3.3.1 The GeForce GT200 hardware architecture

Modern graphics chips have evolved away from the original graphics pipeline, dynamically allocating hardware resources for the tasks involved in graphics. The NVIDIA GeForce GT200 GPU and 1 GByte DRAM are the main components of the GeForce GTX 280 graphics card. The core components of the GPU are the 10 *TPCs*, standing for *texture processing clusters* in graphics processing mode, and *thread processing clusters* in parallel compute mode. Each TPCs consists of 3 streaming multiprocessors (SMs), that in turn are made up of 8 *streaming processors* (SPs).[NVID08c]

Figure 3.5 shows the GPU's structure in compute mode. The *thread scheduler* at the top is hardware-based and responsible for efficiently scheduling threads across the TPCs. Below the TPCs are *texture caches* and *DRAM controllers* for interfacing the global memory. The texture caches combine memory accesses for higher efficiency and bandwidth usage, the *atomic* elements refer to atomic read-modify-write operations the GPU is capable of. The DRAM interface is 512 bit wide and achieves a peak bandwidth of 138 GB/s.
At the bottom of figure 3.5, one of the TPCs three SMs is shown in detail. Every SP has its own ALU, that operates on the 2048 local registers (8 kB). This number might seem high, but is qualified by the number of threads running quasi-parallel on a core dividing the registers among them (cf. 3.3.2.2).

Besides the ALUs basic functionality of executing one *MAD* per cycle (a fused floating-point multiply and add), a *super function unit* (SFU) executes more sophisticated operations like root extraction or supports the calculation with up to four floating-point MULs. This results in an average SP performance of $2 + 1 = 3$ FLOP/cycle in single precision. Given a chip clock rate of 1296 MHz, the overall performance equals $3 \cdot 240 \cdot 1.296 = 933.12$ GFLOPS, as mentioned in 3.1.4. Very important for scientific usage are the double precision units, one in each SM. These can perform one fused MAD per cycle in accordance with IEEE 754R. Here the overall performance computes to $2 \cdot 30 \cdot 1.296 = 77.76$ GFLOPS.

Besides the registers, a SP can access three other kinds of memory:

Shared memory - Every SM is equipped with 16 kB of local shared memory, that all its SPs can read and write in parallel. If there are no conflicts, the shared memory can be accessed as fast as registers (cf. 3.3.5.1)

Constant cache - This 8 kB cache makes data of the constant memory (see below) available with register latency.

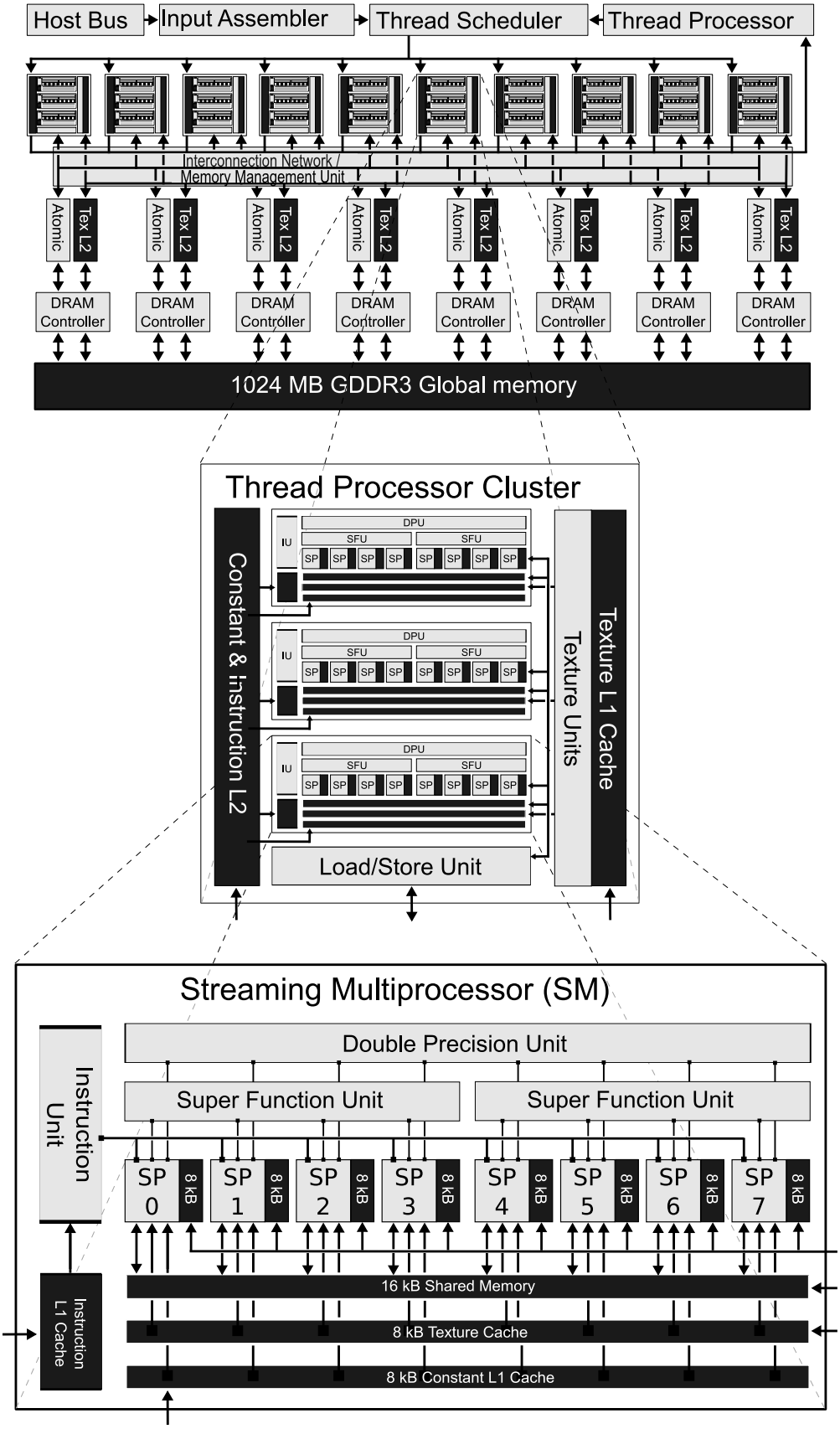

Figure 3.5: Architecture of the NVIDIA GeForce GT200 GPU

Texture cache - Another 8 kB keep previously accessed texture memory and can also be accessed with register latency.

The *texture units* between the SMs texture cache and the TPCs texture L1 cache provide special functions for linear interpolation of texture values and address translation, such that in the case of a cache miss data can be fetched from a two-dimensional neighborhood. Texture memory is not coherent up to this point and thus handled as read-only.
The TPCs constant cache gets its data from a 64 kB block of the DRAM. This memory area can not be written by the GPU but allows fast accesses (in addition to caching). The rest of the global memory is read/write accessible by all SPs, but is rather provided for data exchange between host and GPU because of its latency. Those data transfers can be done via *direct memory access (DMA)* over the PCIe bus.

It holds for every read and write operation, that data can be arbitrarily localized in memory, gathering and scattering is allowed without restrictions. The latency of these accesses to global memory depends on various conditions that are considered more detailed in 3.3.5.1.

3.3.2 The CUDA programming paradigm

3.3.2.1 Thread hierarchy

It is still possible to use CUDA enabled hardware for stream processing. In order to bridge to the CUDA programming model, stream processing can also be seen as a particular, restricted form of *multi-threading*. The individual threads of a multithreading program are identified by a unique ID, issued by a *thread scheduler* on execution. Every thread is able to run independently, read from global memory and use an ALU and registers for computation. The result is then written into particular registers.

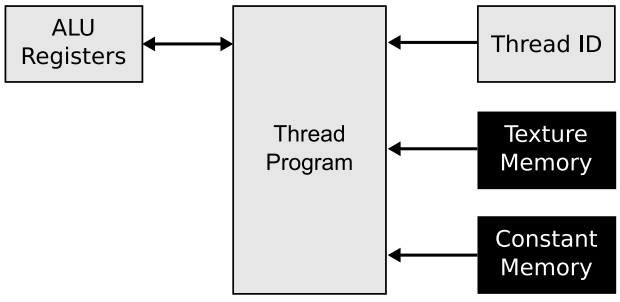

Figure 3.6: Read/write possibilities of a conventional thread program on a GPU

As it is the case with the GT200 GPU (cf. 3.3.1), every CUDA enabled device offers its threads access to global memory, a special shared memory, associated to the SP the thread is running on, and a local memory. The set of threads T is disjointly partitioned into groups $T = \coprod_{i=1}^{n} T_i$, such that the threads in T_i can

read from and write to shared memory S_i. In order to keep data dependencies, a *barrier synchronization* for all threads in T_i is available. The *local memory* is only used for automatic variables, if the register space is not sufficient (cf. 3.3.3.1). Its performance is equal to global memory.[NVID08b]

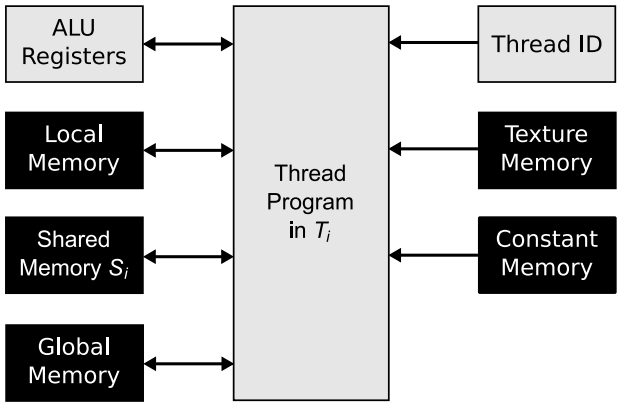

Figure 3.7: CUDA thread program with access to local, shared and global memory

In order to ensure shared memory functions, all threads of a group reside on the same SM. In the CUDA diction, such a group is called *thread block*, as its individual threads are addressable in a one-, two- or three-dimensional fashion, providing natural indexing for e.g. vectors, matrices and fields. The number of threads in a block is physically limited by physical resources of the SM, e.g. up to 512 for the GT200 GPU. However, multiple identically shaped blocks can be created, that again are virtually arranged in a one- or two-dimensional *grid*, consisting of up to 65,535 blocks (GT200), identified by *block IDs*. The blocks of a grid are meant to run without a definite order and only share the global device memory; even so they might coordinate their activities by using atomic writes to variables in global memory(cf. 3.3.2.2).
Different grids utilizing the same kernel can be executed in parallel, given enough hardware resources are available. Dependent grids can be separated by a inter-kernel barrier, as shown in figure 3.8. This barrier guarantees that all blocks of the first grid are completed before any block of the second grid is launched.[NBGS08] Different kernels are executed sequentially and are provided the unrestricted capabilities of the device.

The broad options for choosing thread block and grid sizes allow different levels of granularity, depending on the task to be addressed. The fact, that the number of thread blocks can greatly exceeds the number of processors and the thread blocks themselves can be scheduled in any order, will allow CUDA programs to scale efficiently to increasing numbers of processor cores.

3.3.2.2 Hardware mapping

Across the 10 TPCs, CUDA uses a *MIMD* (multiple instruction multiple data, [Flyn66]) processing model, the single SMs employ a model called *SIMT* (Single instruction multiple thread). The SM's instruction unit creates, manages, schedules,

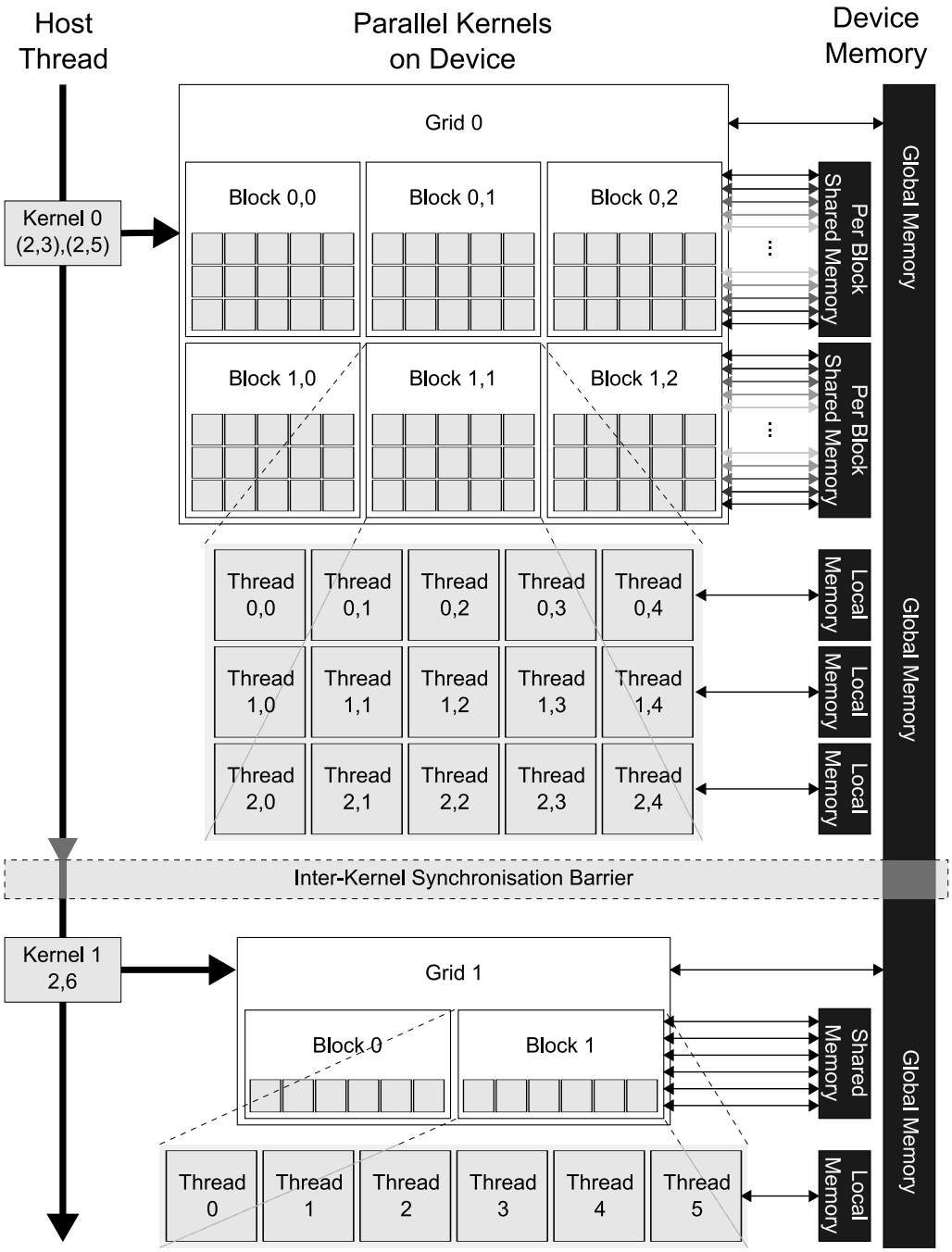

Figure 3.8: CUDA kernel sequence with exemplary levels of granularity and possible memory sharing

and executes the individual threads of a thread blocks in groups of 32, so-called *warps*.

This means, that n threads create $\lceil n/32 \rceil$ warps, with the last warp being filled with non-effective threads, if necessary. An instruction of a warp is executed in four cycles on eight threads at a time. Usually several instructions are combined, such that at the same point in time, different threads may be at different points

within the program execution. Threads of a warp may also explicitly diverge, e.g. into data-dependent conditional branches. The warp executes each branch path sequentially, disabling other threads not on this path. After converging back to the same execution path, branches are combined again. This method can lead to a 32 times higher run time, for the case that every thread is diverging. These divergences only occur within a warp; different warps are executed independently despite running potentially identical code.

A thread is always bound to a particular SP. The threads of a SP disjointly divide its registers among them, hence switching threads does not imply swapping register entries. The SM's warp scheduler may thus hide memory latency by choosing another of the SPs remaining threads for execution in the meantime. Each SM manages a pool of 32 warps of 32 threads per warp, a total of 1024 threads.
Different thread blocks are equally bound to a SM. Bringing another thread block forward is not supported, thread blocks are always executed uninterrupted and sequentially. The hardware provides a strict barrier synchronization, where all to be synchronized threads have to reach the same synchronization instruction, before any of them is allowed to continue with the following instruction. When the first thread reaches the `__syncthreads()` instruction at instruction counter value i, the execution is held until all other threads of this warp reached the same `__syncthreads()` at i. To prevent deadlocks, diverged threads should never miss out a `__syncthreads()` or execute a different one. `__syncthreads()` takes four clock cycles to issue for a warp if no thread has to wait for any other threads.
Alternatively atomic memory access to global memory can be utilized for manual synchronization with busy-waiting (spinning). In theory threads can be synchronized at different points in their programs, even across different SMs. Practically, deadlocks might be created by the scheduler as soon as there are more blocks then SMs and not all to be synchronized blocks can be ran in parallel. As mentioned before, thread blocks are executed uninterruptedly and thus block the SM when busy-waiting.
A subset of threads of the same block however can unhesitatingly be synchronized using this technique.

Although it is not possible to run concurrent kernels on the same GPU, there is the possibility to run several streams via a stream scheduler on the host. Given a high number of kernel launches on disjoint areas of memory, organizing these kernels in streams may increase performance noticeably.

3.3.3 The CUDA application programming interface

NVIDIA offers on its website[2] the essential tools to program CUDA enabled hardware devices: Device drivers, the *NVCC* compiler, libraries, header files and a software development kit (SDK) with several examples. The main functionality is provided by libraries, such that programming a CUDA kernel is done in usual C style with some C++ enhancements. Just few real extensions are necessary.

[2] http://www.nvidia.com/object/cuda_home.html

3.3.3.1 C Language extensions

The programming language C is extended by four function type qualifiers, three variable type qualifiers, two directives and five built-in variables. All other functionalities are provided in a usual C/C++ style by libraries, compiler constants and templates.

Function qualifiers The three function type qualifiers `__global__`,`__device__` and `__host__` determine for which target a function is compiled and from where it may be called. A prefixed `__global__` instructs the compiler to compile the function as CUDA kernel, that can only be executed on the device and is called by a particular directive from the host. The execution is asynchronous, meaning the program continues before the kernel returns, if not stated otherwise.
In order to unitize kernels, `__device__` functions can be created that likewise run on the GPU but are only callable from `__global__` or other `__device__` functions. Further restrictions on kernels and their execution are presented in table 3.2 and motivated in [NVID08b].
By default, device functions (i.e. functions qualified by `__global__` or `__device__`) are embedded in-line if not the fourth qualifier `__noinline__` is used.
`__host__` is the default function qualifier, that is used if neither `__global__` nor `__device__` have been stated and indicates, that the function is compiled for the host. It is however combinable with `__device__`, such that the function is compiled twice, once for the host and once for the device.

	Executes on	Called by	Returns	Rec.[1]	St.Var.[2]	# Arg.[3]
`__global__`	GPU	Host	`void`	-	-	fixed
`__device__`	GPU	GPU	any	-	-	fixed
`__host__`	Host	Host	any	•	•	variable

Table 3.2: Restrictions on function types. [1] Allows recursion, [2] Allows static variables, [3] Number of Arguments

Variable type qualifiers The three variable type qualifiers `__device__`,`__constant__` and `__shared__` define that a variable is to reside in the device memory and where in particular memory shall be allocated. A prefixed `__device__` states, that this variable resides in the device's global memory and can be read and written by both host and device functions. Its memory stays allocated until it is explicitly freed or the application finishes.
The `__constant__` qualifier (optionally used in addition to `__device__`) declares a variable that is put into the constant memory space of the device and hence can only be written by the host (cf. 3.3.1).
When using the qualifier `__shared__` (optionally used in addition to `__device__` again), the variable is placed in a SM's shared memory during run-time. Only the threads of the corresponding thread block are able to read from and

write to it, hence it can not be initialized when launching the kernel. After the thread block's execution, the memory space is freed again. When using `__shared__` in connection with `extern`, as in

```
extern [__device__] __shared__ Type Array[];
```

the size of this array `Array` of type `Type` is determined when launching the kernel (see below). All similarly declared variables start at the same memory address, thus the layout of these variables must be explicitly managed through offsets. Details are again explained in [NVID08b] and summarized in table 3.3. Declaring a variable in device code without any of these qualifiers generally makes it reside in a register. In some cases the compiler might place it in local memory (cf. 3.3.2). This is often the case for large structures or arrays that would consume too much register space and arrays for which the compiler cannot determine that they are indexed with constant quantities.

	Memory	Lifetime	Static	Read	Write
`__device__`	Global	App.	•/-	Host/GPU	Host/GPU
`__constant__`	Constant	App.	•	Host/GPU	GPU
`__shared__`	Shared	Block	•	GPU	GPU

Table 3.3: Properties of variables declared with different type qualifiers

Directives Two new directives have been introduced with CUDA. The first one is a `#pragma` directive that controls the compiler's behavior in connection with loops. Usually the `nvcc` compiler unrolls small loops with a known trip count. When placing the `#pragma unroll [n]` directive immediately before the loop, the unrolling of this particular loop can be controlled. The optional number `[n]` specifies how many times the loop must be unrolled; hence a `#pragma unroll 1` will prevent the compiler from ever unrolling the loop. Attention must be paid to the fact that the loop is in any case unrolled `n` times, even if the specified trip count is smaller then `n`. Not specifying `n` leads to complete unrolling, if the trip count is known at compile time.

The second directive is needed for specifying the execution configuration when launching a (`__global__`) kernel function on the device:

```
Kernel<<<dGrid, dBlock [, allocBytes] [, stId]>>>(paramList);
```

A kernel function `Kernel` is hereby launched with the necessary parameters `dGrid` and `dBlock` of type `dim3`, specifying the grid and block dimensions. `allocBytes` and `stID` are optional parameters, the first one defining the number of bytes to be dynamically allocated in shared memory per block for the above mentioned external arrays, the second defining the stream ID of a stream that this kernel launch is appended to.

Built-in variables The five built-in variables `threadIdx`, `blockIdx`, `blockDim`, `gridDim` and `warpSize` are read-only variables to be used in device functions. All variables aside from `warpSize` are three-dimensional (`dim3` or `uint3`), with elements `x`, `y` and `z`. At run-time, the variables provide the thread ID of the running thread within the block in `threadIdx` and the block ID within the grid in `blockIdx`. `blockDim` and `gridDim` provide the respective size along the (up to) three dimensions. `warpSize` finally contains the warp size in threads.

3.3.3.2 Run-time components

The CUDA run-time components are offering a variety of functionalities and data structures. The components are tripartite into a host, device and common run-time component with respect to the context they can be called in. Host and device run-time component are exclusively accessible from the respective functions, the common run-time component can be called from both host and device functions.
Figure 3.9 shows the role of the run-time components within the CUDA software stack: The run-time libraries implement the application programming interface and interact with the device driver. Based on this, CUDA libraries for BLAS and FFT are offering higher-level functionalities.

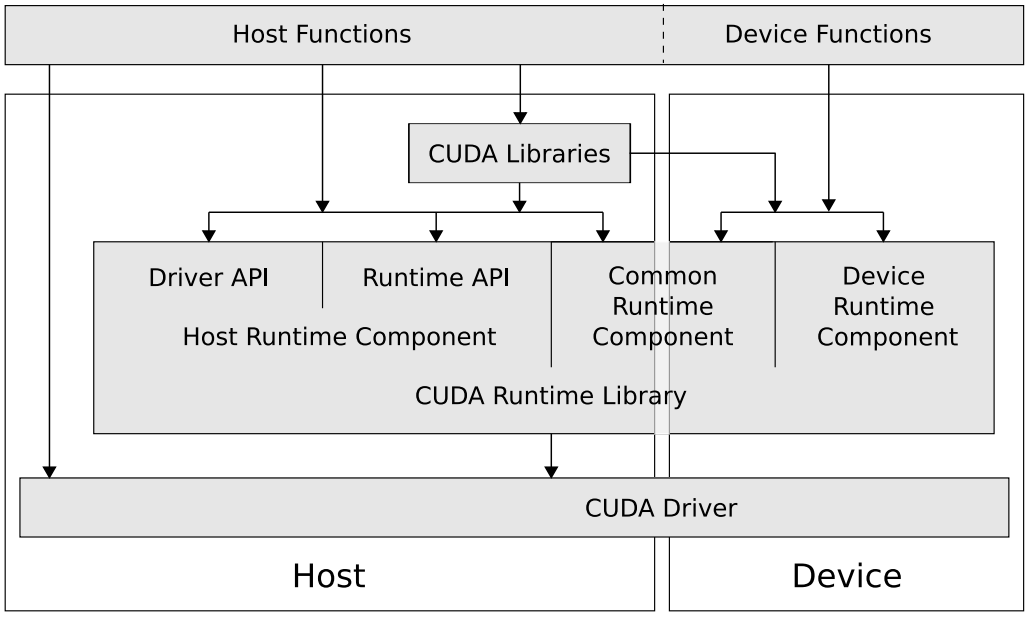

Figure 3.9: CUDA Software stack with run-time components

Not every CUDA enabled device implements the full range of functionalities provided by the run-time components. The supported subset of operations is indicated by the *compute capability* of the device. Currently NVIDIA offers graphics cards with compute capability 1.0, 1.1, 1.2 and 1.3. The following overview of the run-time components presents compute capability 1.3 and mentions limitations of former versions.

Common run-time component The common component offers the following vector data types, derived from the supported primitive data types (in EBNF notation[fSta96]):

```
[u] ( char | short | int | long ) ( 1 | 2 | 3 | 4 ) ,
          float ( 1 | 2 | 3 | 4 ), double2
```

The aforementioned `dim3` type is similar to `uint3`, with the difference, that unspecified components are initialized with 1 instead of 0.
Furthermore, the common component offers a variety of mathematical functions that are implemented for the device and make use of the C standard library, when executed on the host. These functions include for example `sqrt`, `sin`, `exp` and `erf`. A complete listing can be found in section B.1 in [NVID08b]. The common component also includes a clock counter `clock()` and some texture operations, that are explained in an extra section.

Device run-time component In addition to the math functions of the common run-time component, the device run-time component provides faster implementations of some functions albeit with lower accuracy. These functions can be used globally when compiling with the `-use_fast_math` flag.
The already discussed `syncthreads()` functions is also part of the device run-time component, as well as the atomic functions for memory access. Such an atomic function performs a read-modify-write operation in global or shared memory without interference by other threads. Atomic functions operating on 32-bit words on global memory are supported since compute capability 1.1, functions operating on shared memory and operating on 64-bit words are supported in compute capability 1.2 and above. A list of all atomic functions can be found in appendix C in [NVID08b]. Devices with compute capability 1.2 and above furthermore offer two warp vote functions `__all` and `__any`. The first one evaluates an integer predicate for all threads of the warp and returns non-zero if and only if the predicate evaluates to non-zero for all of them. `__any` returns non-zero if and only if the predicate evaluates to non-zero for any of them.
The texture fetch functionalities of the device run-time component are also explained in the extra section.

Host run-time component As shown in figure 3.9, the host run-time component is composed of two APIs, a low-level *driver API* and a higher level *run-time API*. These APIs are mutually exclusive, thus an application should only use one of them.
The driver API offers more control, but is harder to program and to debug. For example, a kernel launch cannot be done with the discussed new directive and no device emulation mode is offered for debugging. Both APIs offer the following functionalities, only through different interfaces. All driver API functions are prefixed with `cu`, all run-time API functions with `cuda`.

Device management All devices available in the system can be enumerated, their properties can be queried and one of them can be selected for kernel

execution through the APIs. One host thread can execute device code on one device only, hence multithreading on the host is needed to manage a multi-GPU system. On the other hand, different host streams may execute device code on the same device.

Memory management The APIs provide functions to allocate and free memory on both host and device. Besides the usual `malloc()` it is possible to allocate page-locked memory on the host. Page-locked memory achieves a higher bandwidth when transferring data to and from the device, but its available amount is strictly limited. Furthermore, the system's overall performance is reduced along with its physical memory available.
In 4.2, thorough test are conducted in order to measure the performance difference between pageable and page-locked memory allocation, that show that even more system factors are involved.
Device memory can be allocated either as linear memory in a (currently) 32 bit address space or as CUDA arrays. *CUDA arrays* are one-, two-, or three-dimensional and composed of elements of the above mentioned vector data types. Their memory layout is optimized for texture fetching, that is why they are only readable through texture fetching on the device. The host can write to CUDA arrays via the respective memory copy functions of the APIs.

Asynchronous Concurrent Execution As indicated previously, concurrent execution between host and device is partly possible and manageable through *streams*. A stream is identified by a stream ID and executes its operations in order. The default stream has ID zero, any kernel launch, memory set or copy without a stream parameter is assigned to it and no subsequent operation may begin until it is done. A new stream is created along with a stream object that is specified as stream parameter of a kernel launch and/or a host↔device memory copy. Different streams execute their operations out of order with respect to one another or concurrently. For example, there are `cu(da)StreamSynchronize()` functions for waiting until all preceding operations in a stream have completed. Of course, there are also functions implemented for destroying streams and returning control to the host thread.
It should be underlined, that two operations from different streams can only run concurrently if neither a page-locked host memory allocation, a device memory allocation, a device memory set, a device↔device memory copy nor any CUDA operation to stream 0 is called in-between them by the host thread.

Event management Events allow fine granular device monitoring and high accuracy timing (below one millisecond) and thus precise synchronization possibilities. Events are defined anywhere in a stream and their eventuation is recorded when all tasks (or all operations in a given stream) preceding the event have completed.
Events in stream zero are recorded after all preceding tasks/operations from all streams are completed.

Texture management Also the host run-time component provides texture management functions. These are explained together with the texture functionalities of the other run-time components in the following section.

DirectX and OpenGL interoperability For the sake of completeness, it should be mentioned that it is possible to map OpenGL or DirectX buffer objects into the address space of CUDA for interoperability. This functionality can be useful when directly visualizing computational results.

3.3.3.3 Texture management

The hardware used for graphics can partly also be used via CUDA. Placing data that is only to be read in texture memory instead of global memory can have significant performance benefits, especially because of caching and additional functionality.
Some region of memory is declared as *texture* by creating a *texture reference* via the host run-time component at compile time:

```
texture<Type[, Dim][, ReadMode]> texName;
```

`Type` specifies the type of the elements returned, when fetching the texture. These elements are called *texels*, short for "texture elements", and can be of any of the basic integer or single-precision floating-point types or even of any of the vector types defined in the common run-time component.

`Dim` specifies optionally the dimensionality of the texture reference and can be chosen 1 (default), 2 or 3.

`ReadMode` is another optional parameter that allows different interpretation of a 8- or 16-bit integer data type. When setting `ReadMode` to cudaReadModeNormalized-Float, the value is actually returned as floating-point type, where the full range of the integer type is mapped to $[0.0, 1.0]$ for unsigned integer types and $[-1.0, 1.0]$ for signed integer types. When using the default `cudaReadModeElementType`, no conversion is performed.

During run-time, further attributes of a texture reference can be modified using functions of the host run-time component. It can be specified if texture coordinates are to be normalized or not, texture filtering can be enabled and the addressing mode can be selected.
By default, textures are referenced using floating-point coordinates in $[0, N)$ where N is the size of the texture in the corresponding dimension. Normalized texture coordinates are to be specified in the range $[0.0, 1.0)$ for each dimension instead and are thus independent of the texture size.
If the texels are of a floating-point type, (bi-)linear texture filtering may perform low-precision interpolation between neighboring texels. Based on where the texture coordinates fell between the texels, the surrounding texels are read and the return value of the texture fetch is interpolated. For one-dimensional textures, simple linear interpolation is performed for two-dimensional textures, bi-linear interpolation is used.
The valid address space of the texture can be enlarged by using the addressing modes *clamping* and *wrapping*. Clamping sets coordinates values that are out of range to the maximum or minimum value, respectively. This is the default addressing mode for both normalized and unnormalized coordinates. For normalized coordinates,

wrap addressing can be used for e.g. periodic signals. It uses only the fractional part of the texture coordinate; for example, 1.25 is treated as 0.25 and -1.25 is treated as 0.75.

Naturally, texture references have to be bound to some specific device memory during run-time. Again, the host run-time component provides the respective functionality to bind references to linear memory or CUDA arrays, e.g. via `cudaBindTexture` and `cudaBindTextureToArray` for the run-time API. The aforementioned features, multi-dimensionality, texture filtering, normalized coordinates and out-of-range addressing, can only be used in connection with CUDA arrays.

The memory management functions of the APIs offer possibilities to automatically convert ordinary arrays to CUDA arrays during host→device memory copies and vice-versa. Within a Kernel, textures are read via texture fetch functions of the device run-time component.

3.3.4 NVCC and Debugging

The CUDA toolkit contains a compiler driver called `nvcc`, that is actually a gcc (Gnu Compiler Collection[3]) mimicking wrapper around a more complicated compilation process. A CUDA program usually includes C/C++ files (`.c` or `.cpp`) and CUDA files (`.cu`) containing mixed host and device code. For C/C++ files, `nvcc` simply invokes the native compiler on the system, CUDA files are further processed[Half08]: First, a preprocessor by the Edison Design Group (EDG[4]) called `cudafe` separates host from device code by writing it into separate files; the C/C++ host code is again forwarded to the default system compiler. The device code is then processed by a modified version of the Open64 compiler[5], an open source C/C++ compiler, originally developed for the Intel IA-64 (Itanium) architecture. This modified version is called `nvopencc` and generates *PTX* code (Parallel Thread eXecution), an assembly code that is configured and compiled for the target machine at installation time [NVID08d]. PTX is independent of the actual available processor and supports besides others unlimited virtual registers of different sizes, branch prediction and vector memory accesses. The PTX code is then passed to `ptxas` a variation of OCG (Optimized Code Generator), a low-level compiler for graphics codes by NVIDIA. This compiler is built into the graphics driver for doing just-in-time compilation of graphic codes, it allocates registers and schedules the instructions according to the chip being used. Using PTX as intermediate code shall make the execution of existing code on future hardware as efficient as possible without further modifications. The output is a device specific binary object (`.cubin`) that can be loaded and executed on the device using the CUDA driver API (cf. 3.3.3.2), or it can be linked to the generated host code, that now contains a translation of the execution configuration directive (3.3.3.1) into the necessary run-time start-up instructions. An overview of the whole compilation trajectory is given in figure 3.10.

`nvcc` can be invoked with a variety of common compiler options, such as for macro definition, include/library path setting and for compilation process steering. Interesting options are the already mentioned `-use_fast_math`, that substitutes math

[3] http://gcc.gnu.org [4] http://www.edg.com [5] http://www.open64.net

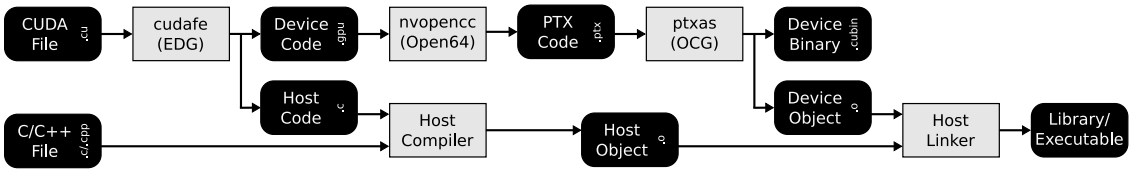

Figure 3.10: The CUDA compilation trajectory as performed by `nvcc`

functions of the common run-time library by faster GPU specific implementations, or `-keep` and `-dryrun` that make `nvcc` keep all temporary files (e.g. `.ptx`) and list all steps performed. The option `-arch sm_13` is really crucial when using e.g. the double precision functions provided by compute capability 1.3. Not specifying this option leads to implicit usage of the single precision pendants.
One of the most important flags is `-deviceemu`, that instructs `nvcc` to only emulate the complete device functionality. Programs compiled with this option run on the CPU without the need for CUDA enabled hardware. The great advantage is that now all host functions can be called from device functions and all device memories can be accessed by host functions for debugging. Besides others, `nvcc` provides the compiler constant `__DEVICE_EMULATION__` to declare separate code segments only to be compiled in emulation or non-emulation mode. A detailed description of `nvcc`'s workflow can be found in [NVID08a].

3.3.5 Performance remarks

Besides algorithmic approaches, hardware-aware programming is the key to running kernels close to peak performance.[NVID08b] Since host↔device bandwidth is a possible bottle neck, a GPU program should perform as many arithmetic operations per memory access as possible and allow a high number of threads per SM, such that the warp scheduler can hide memory accesses with calculations. A detailed list of built-in function instructions and their expense in cycles can be found in [NVID08b]. Meanwhile, the available memory bandwidth should be maximized for each category of memory. Extensive testing of the GPU's functionality and performance is done in chapter 4, section 4.3.2 deals especially with the types of memory and their usability.

3.3.5.1 Memory bandwidth exploitation

While register accesses usually do not create an overhead, read-after-write dependencies and bank conflicts can lead to extra clock cycles per instruction. Only as soon as there are at least 192 active threads per SM, the read-after-write delays can be ignored [NVID08b]. Bank conflicts are tried to be avoided by the the compiler and thread scheduler, that achieve best results when the number of threads per block is a multiple of 64. Thus a rule-of-thumb could be to choose the number t of threads per block $192 \leq t = k \cdot 64,\ k \in \mathbb{N}$.

When reading a float from local or global memory 400 to 600 clock cycles are needed as there is no cache for acceleration. The warp scheduler hides this latency efficiently as long as there are sufficient independent arithmetic operations following this read

or there are enough other threads waiting for execution. Memory accesses of threads of a *half-warp* (first or second half of a warp) are automatically *coalesced* into a single transaction if the words accessed lie in the same segment of size equal to

- 32 bytes if all threads access 8-bit words,
- 64 bytes if all threads access 16-bit words,
- 128 bytes if all threads access 32-bit or 64-bit words.

This achieves a bandwidth that is an order of magnitude higher for 32-bit accesses, four times higher for 64-bit accesses and still two times higher for 128-bit. Coalesced 32-bit accesses deliver a little higher bandwidth than coalesced 64-bit accesses and a noticeably higher bandwidth than coalesced 128-bit accesses.

Generally, it is useful to copy values that are to be used several times into shared memory or even recompute them if this space is limited. As mentioned before, shared memory can be accessed without overhead as long as there are no bank conflicts. The shared memory is organized in 16 simultaneously accessible banks, where successive 32-bit words are assigned to successive banks. When the threads of a half-warp access shared memory and two addresses of a request fall in the same memory bank, a bank conflict occurs and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary. There is also a broadcast mechanism that reduces the number of bank by allowing a 32-bit word to be read and broadcasted to several threads simultaneously. Figure 3.11 shows an example access pattern that might cause a bank conflict: If the word from bank 5 is chosen as the broadcast word during the first step, no bank conflict occurs. Otherwise, one of the threads trying to access bank 5 is served together with all other threads not trying to access bank 5. In a second step the word form bank 5 is broadcasted to the remaining threads.

If two or more threads of a warp try to write to the same location in global or shared memory with a non-atomic instruction, the number of serialized writes to that location and their order is undefined, only one of the writes is guaranteed to succeed. Atomic writes are serialized but the order is still undefined.

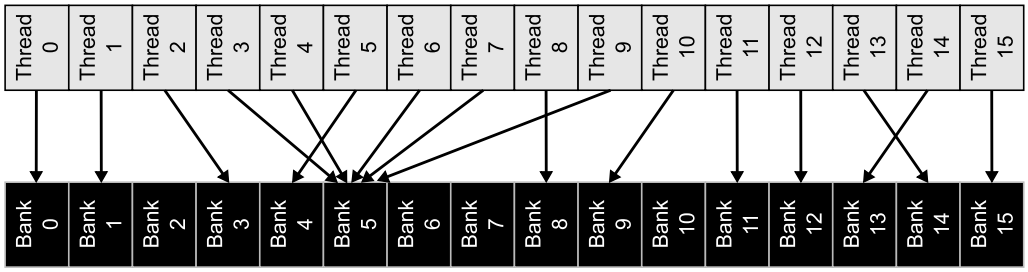

Figure 3.11: Example shared memory read access pattern that causes a bank conflicts if bank 5 is not chosen for broadcasting in the first place

3.3.5.2 Branching

Another source for performance losses is branching within warps. As mentioned before, because of the SIMD architecture, divergent threads have to be processed in serial. Thus, branching should in best case occur at warp boarders, e.g. by evaluating `threadIdx.x/warpSize`. As long as there is no block wide synchronization, different warps may be at different points in their program at the same point in time.
The compiler is also able to do simple branch prediction: The compiler replaces an `if` or `switch` instruction with predicated instructions if the number of instructions controlled by the branch condition is less or equal 4. If the compiler determines that the condition is likely to produce many divergent warps, it is less or equal 7. The predicate is a per-thread condition code that is set to true or false based on the controlling condition. Instead of branching, each instructions gets scheduled for execution, but only the instructions with a true predicate are actually executed. Instructions with a false predicate do not write results, and also do not evaluate addresses or read operands.

3.3.5.3 Block size to grid size to stream number ratio

An optimal distribution of work load is achieved by occupying all SM as efficiently as possible. Clearly, there have to be at least as many thread blocks as there are multiprocessors in the device. But running only one block per SM would force the multiprocessor to idle during synchronization and memory reads if there are not enough threads per block to cover the load latency. The number of threads per block should be chosen as a multiple of the warp size to avoid wasting resources with under-populated warps, or better, as motivated in 3.3.5.1, as a multiple of 64. Choosing a high number of threads per block facilitates efficient time slicing, but reduces the number of registers available per thread. This might even prevent a kernel from launching at all, if it tries to allocate more registers than allowed by the execution configuration.
Whenever possible, two or more blocks should be active on each multiprocessor to allow overlap between blocks that wait and blocks that can run. But running twice as many blocks as there are multiprocessors means also allowing only half the amount of registers and shared memory to be allocated per block.
The number of registers available per thread is thus predictable through

$$\frac{R}{B \times 32\lceil T/32 \rceil}$$

where R is the total number of registers per SM, B is the number of active blocks per SM and T is the number of threads per block.
In order to choose an optimal B and T, the memory usage of a kernel (registers, as well as local, shared and constant memory) can be examined when compiling with the `-ptxas-options=-v` option. It has also to be kept in mind, that variables of `double` or `long` type occupy two registers at once. The compiler actively tries to minimize register usage and hence maximize the multiprocessor occupancy, i.e. the ratio of the number of active warps per SM to the maximum number of active warps physically possible.

The number of blocks per grid should be at least 100 if one wants it to scale to future devices; 1000 blocks will scale across several generations [NVID08b].

Taking also memory transfers into account, utilization of streams is inevitable in order to achieve the shortest total computation time. When transferring memory between host and device it is in general more efficient to transfer a larger set of data at once, but the fact that kernel execution can only start after the transfer is completely finished, single transfers may have advantages. Two examples in figure 3.12 show transfer time to kernel runtime ratios, when single transfers might be recommendable:

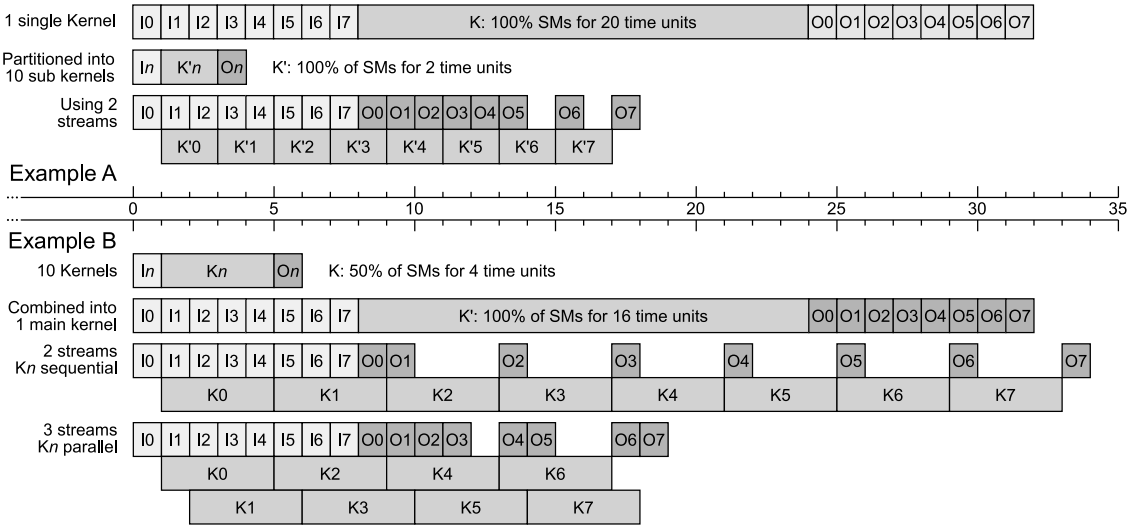

Figure 3.12: Timing differences between execution as single kernel or stream of sub kernels

Both examples predict the resulting scheduling when executing a kernel K with input data I and output data O on the device in different fashions. In example A, one big kernel is subdivided into eight partial kernels that are then executed as streams. Although the total data transfer time is increased by 12.5 % because of waiting for kernel results, the total execution time could be reduced from 32 to 18 thanks to parallelization. Example B schedules eight independent kernels that each only need half of the available SMs. Scheduling these kernels as one collective kernel would take 32 time units, while in this case using one stream for data transfers and one for kernel execution would achieve suboptimal 34 time units. Exploiting the low resource utilization of the kernels, introducing a second stream for kernel execution would reduced the total execution time to 19 time units. As unfortunately this concurrent kernel execution is up to now not supported by the API, calling different `__device__` functions according to e.g. even and odd block numbers (albeit sacrificing shared memory and registers) could be a work-around.

3.3.5.4 Scalability

The keen competition of graphics card manufacturers in the gaming market led to a high increase of GPU computing power in the past years. Mainly due to increased

parallelism, new graphic chip generations performed percentally and absolutely better then respective main stream CPUs and even better then predicted by Moore's Law.

In the future there are multiple possibilities of further increasing performance: Higher clock rates, more cores and shared memory or cache per GPU or more GPUs per board. Most of these methods would lead to a linear performance increase for many parallel programs.
But also the whole GPU accelerated computing system can be scaled: Several graphics cards can be used on the same mainboard if it provides several PCIe slots. Consumer mainboards meanwhile offer up to 4 PCIe slots, but only two of them are provided the full memory bandwidth and only few mainboards provide yet double spacing between the slots for the large graphics card's fans and coolers. It is though possible to install PCIe switches and run several more graphics cards on the same PCIe port (up to 16), but the bandwidth is similarly decreased.
Of course, a cluster can be build by connecting several GPU accelerated workstations or building a rack of GPU servers. Such system are offered currently only by NVIDIA under the name *Tesla*. A Tesla S1070 server for example, contains four GT200 GPUs together with 16 GByte memory in one unit of a 19-inch rack. The accumulated computing performance is 3.74 TFLOPS in single precision because of a higher core clock. Provided enough monetary resources, such a rack server is easily scalable.

3.4 Summary

Multiple accelerator technologies are suitable to take on tasks in HPC, these include GPGPU, FPGAs, ClearSpeed and the Cell processor. Each technology offers advantages like high single precision performance (GPGPU), flexibility (FPGAs), advanced software tools (ClearSpeed) and high external memory bandwidth (Cell), but none can provide all at once.

NVIDIA's CUDA technology constitutes a new programming paradigm in GPGPU: Nearly all of the available hardware can be easily programmed in a C similar fashion without having to care about the actual graphics pipeline. As the hardware is made up of dozens of similar SIMD processing units, programming is done in a massive multithreading fashion. The scheduling of all these threads is done in hardware for the lower two levels of parallelism, while the host manages streams as the third level:

- **Thread parallelism** allows the parallel execution of several threads within a tread block. A block is divided into warps of 32 threads that are ran by the warp scheduler in parallel on the same processing unit and share its resources. Ideally threads of a warp run instruction synchronized, but a barrier synchronization is also available.
- **Block parallelism** provides the parallel execution of several blocks within a grid. The block scheduler divides the blocks among the available multiprocessors (several processing units accompanied with shared memory) where

they run exclusively and uninterrupted. A synchronization mechanism among blocks is not provided, but manually implementable.

- **Stream parallelism** allows the concurrent execution of memory transfers and kernel execution.

The programmer is free to balance parallelism between these levels, but this decision can have a huge performance impact. By providing a profiler and an emulation debug mode CUDA makes GPGPU developed much easier, but the programmer still has to be well aware of the underlying hardware architecture to achieve best results.

4. Basic performance evaluation of the NVIDIA GT200 GPU

The tests conducted in this section evaluate the performance of the ENGTX280 graphics card using key scientific kernels utilized by the majority of equation solvers. First the test configuration is presented and basic bandwidth tests are conducted. Based on this, typical tasks such as dot product computation, matrix-matrix multiplication, sparse matrix-vector multiplication and fast Fourier transformation are ran on the graphics hardware. In addition to this, multi-GPU computing is examined, as well as the interoperability with Fortran.

4.1 Test configuration

A standard workstation offering two PCIe slots was upgraded with two ASUS ENGTX280 graphics cards and a more powerful energy supply in order to turn it into a CUDA workstation. Its components and the software used in the following tests are listed in table 4.1.
It has to be noted that the low *front side bus clock* of 533 MHz is a strongly limiting factor. Currently there are *chipsets*, available offering 800 MHz and dynamic overclocking. Furthermore only one of the PCIe slots is connected with 16 lanes to the P965 northbridge, the second one can only access 4 lanes, a fact that will be reflected in the bandwidth test.

4.2 Bandwidth tests

The following tests copy data packages of various sizes from the system RAM to the global memories of both devices and back. It was experimented with pageable and pinned memory, i.e. page-locked memory, that cannot be moved by the operating system. The pro's and con's of this memory allocation method are explained in 3.3.3.2.
The data varied in size between 1 kb and 512 MB and were each transferred ten

Hardware		Software	
Processor	Intel Core 2 Duo E6600	OS	OpenSuse 10.2
RAM	2x Corsair Value 1 GB	Kernel	2.6.26.5
Mainboard	ASUS P5B Deluxe	CUDA Driver	177.13/177.67
Southbridge	ICH8R	nvcc	1.1
Northbridge	P965	gcc	4.1
CPU Clock	2.4 GHz	icc/ifort	10.1
FSB	266 MHz	CUBLAS/-FFT	1.0
Graphics Card	2x ASUS ENGTX280	MKL	10.0.1.014

Table 4.1: Workstation configuration used for testing

times en bloc in order to obtain measurable results. The whole test was started five times in a row and the median value was plotted in the following charts. Figure 4.1 shows the achieved host-to-device bandwidth and figure 4.2 the device-to-host bandwidth.

From the charts one can clearly tell the effect of the lower number of lanes for the second PCIe slot: The maximum bandwidth achieved for device 1 ($\approx$ 650 MB/s) is about one fourth of the maximum bandwidth achieved for device 0 (2570 MB/s). Furthermore two interesting effects occur in connection with pageable memory. Firstly, the maximum bandwidth achieved on device 0 is significantly lower than for pinned memory and secondly, there is a step in speed-up of memory transfer beginning at 2^{10} kB = 1 MB, most clearly visible in figure 4.2.
The lower overall bandwidth for pageable memory is caused by the operating system that manages swapping according to current load conditions. For the slower device 1, the bandwidth is just not high enough to raise this effect.
The reason for the step at 1 MB can only be traced back to the the operating system, influences of the system BIOS are improbable. The phenomenon was not further analysed as it was not to be expected that this bandwidth limitation can be fixed.

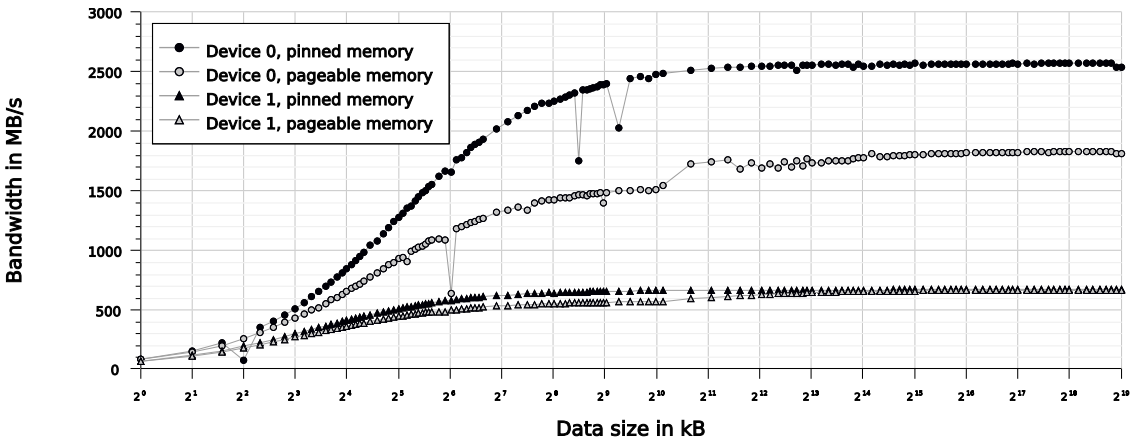

Figure 4.1: Host-to-device bandwidth test results

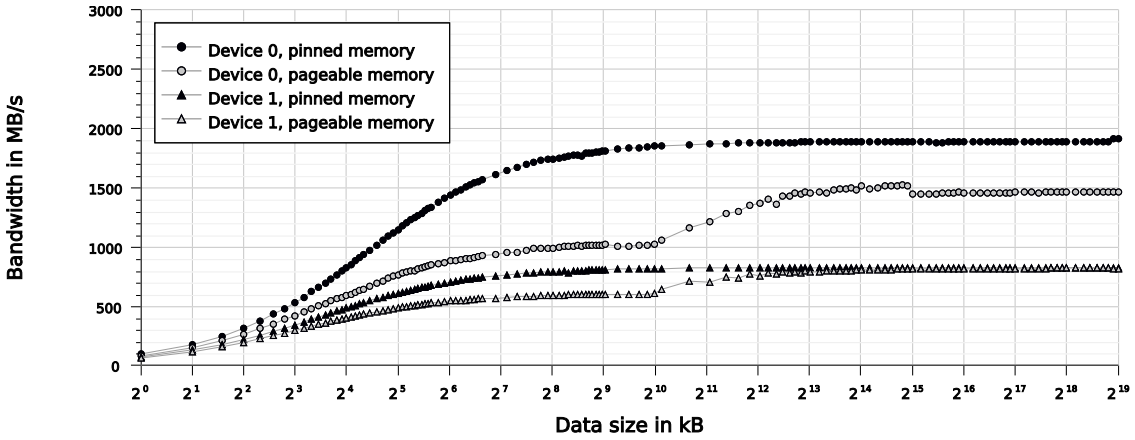

Figure 4.2: Device-to-host bandwidth test results

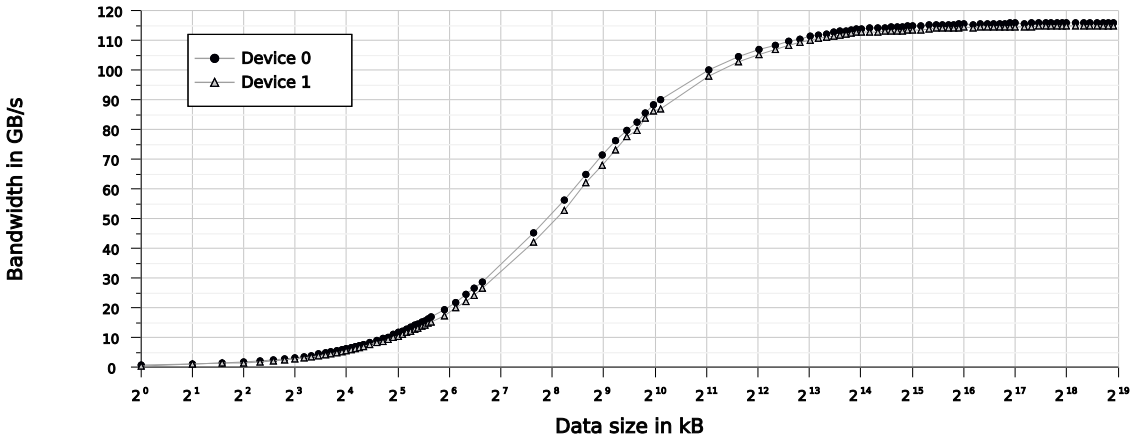

Figure 4.3: Device-to-device bandwidth test results

A general rule of thumb for transferring memory from and to the host could be to send at least 4 MB and use pinned memory whenever possible.

The bandwidth of the devices' global memory was also subject of experiments. As stated in 3.3.1 the theoretical peak bandwidth is 138 GB/s and independent from the system setup. As expected, both devices showed nearly the same performance and reached up to 115 GB/s. The test setup was the same as before, the median of 5 tests is plotted in figure 4.3.

4.3 Key scientific computation kernels

NVIDIA provides currently two kernel libraries that accelerate computations on the GPU and are callable without any CUDA knowledge from any C/C++ or Fortran program, these are *BLAS*[1] and *FFT*[2] implementations. The corresponding libraries are called CUBLAS and CUFFT.

[1] Basic Linear Algebra Subprograms [2] Fast Fourier Transformation

4.3.1 BLAS functions

The BLAS are portable, high- performance implementations of computational kernels. A reference implementation is available in Fortran 77 from www.netlib.org/blas as well as a C interface. The kernels are divided into three levels, with Level 1 BLAS providing scalar, vector and vector-vector operations, Level 2 BLAS providing matrix-vector operations and Level 3 BLAS proving matrix-matrix operations. As optimized implementations are available for a variety of platforms (e.g. ACML by AMD, MKL by Intel, ESSL by IBM, SPL by SUN) the BLAS often form the building blocks of high performance linear algebra software.

In the following tests, a subset of the BLAS was chosen for evaluating the performance of the GPU in production code. From Level 1 BLAS, *axpy*, *dot* and *nrm2* were chosen and *gemm* from Level 3 BLAS. For each operation a small programme was written that performs this kernel on randomly generated data using CUBLAS and the Intel MKL library.

All the above mentioned operations are designed for dense vectors and matrices. The also intensively used operations on sparse matrices are not part of the original BLAS set and hence not part of CUBLAS. In order to assess at least the performance of sparse matrix-vector multiplication *spmv*, this kernel was manually implemented in CUDA and compared against the respective function of the SparseBLAS library by Intel, see 4.3.2.

Each kernel has been ran and timed five times on the faster device 0 using pinned memory on the host side, plotted is the median result. As the GPU's raw computational performance is partly enormously higher then its performance including transfers or the CPU's performance, most results are plotted using a logarithmic ordinate.

Scaled vector addition - *saxpy/daxpy*

This kernel multiplies the vector $\mathbf{x}$ of dimension n by a scalar α and adds the result to the vector $\mathbf{y}$.

```
for (i=0; i<n; i++)
    y[i] += alpha * x[i];
```

The single-precision version holds the prefix s, the double precision version d.

This kernel executes $2n$ operations on $2n + 1$ values. Thus, we have in the case of *saxpy* approximately 4 operations per byte = 0.25 FLOP/B and for *daxpy* approximately 0.125 FLOP/B. Assuming an average maximum host-to-device-to-host memory bandwidth of 2250 MB/s, we get a rough upper performance limit of about 560 MFLOPS for *saxpy* and 280 MFLOPS for *daxpy*. Taking also the rule of thumb from 3.3.5.1 into account, that we should provide at least 192 threads per SM, we can only expect the GPU to achieve peak performances for vectors dimensions greater then $192 \cdot 30 = 5760$.

The actual results in figure 4.4, show a maximum performance including transfer time of 390 MFLOPS for *saxpy* and 196 MFLOPS for *daxpy*. This is reasonable

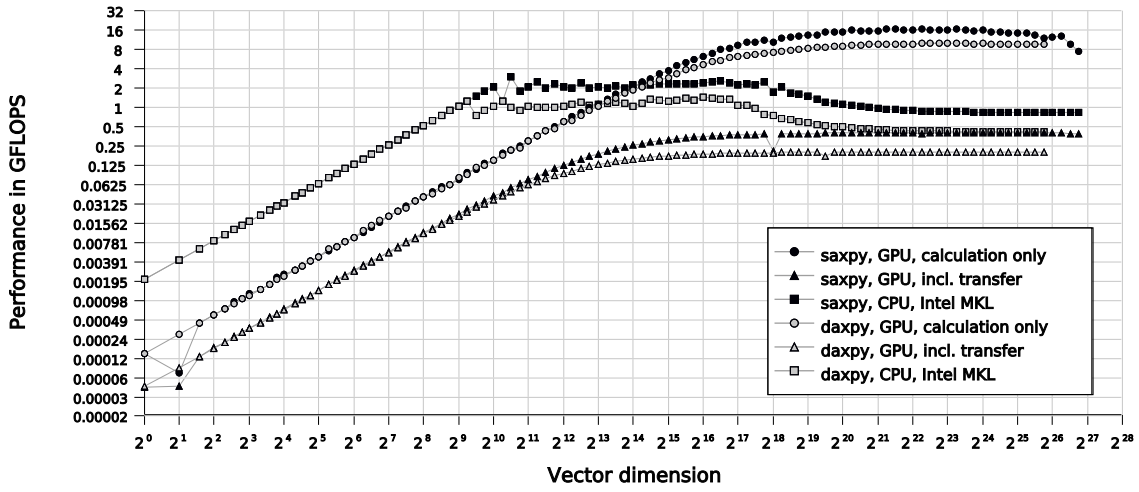

Figure 4.4: *saxpy* and *daxpy* performance test results

as the limit calculation above did not include instruction overheads and calculation time. Furthermore, at a vector dimension of about 5760, only half of the final peak performance could be reached, indicating that in praxis far more threads are needed to fully utilize the SM's performance.

In comparison to MKL, the GPU achieves only half the performance when taking transfer times into account. Surely *saxpy* is not a kernel that should solely be executed when porting existing software, but as soon as it is part of a programme residing on the GPU, the speed-up compared to MKL would be 19.5 for *saxpy* and 9 for *daxpy*, given sufficiently large vector dimensions.

Still, this is far from the theoretical peak performance, but one has to keep in mind, that CUBLAS is in an early stage of development. For example, it does not seem to use the SFU's MAD capability for *saxpy*, what would lead to another increase of performance by 50 %. Another point is, that this kernel does not reuse any data. The high latencies to fetch data from global memory are playing an important role here, too.

Vector dot product - *sdot/ddot*

This kernel computes the *dot product* of a vector **x** and a vector **y**:

```
for (result=0, i=0; i<n; i++)
    result += x[i] * y[i];
```

Again, *sdot* is the single-precision version and *ddot* is the double precision version.

Using the scheme above, the computational intensity would again be 0.25 FLOP/B for *sdot* and 0.125 FLOP/B for *ddot*, as there would be 2 operations per 2 data elements. But while the multiplication in this kernel can be compared to the addition of *axpy*, the second operation can be implemented differently here: The multiplication by α in *saxpy* can be done in $\left\lceil \frac{n}{240} \right\rceil$ steps in parallel, the parallel reduction in *dot* needs only $\left\lceil \frac{\log n}{240} \right\rceil$ steps.

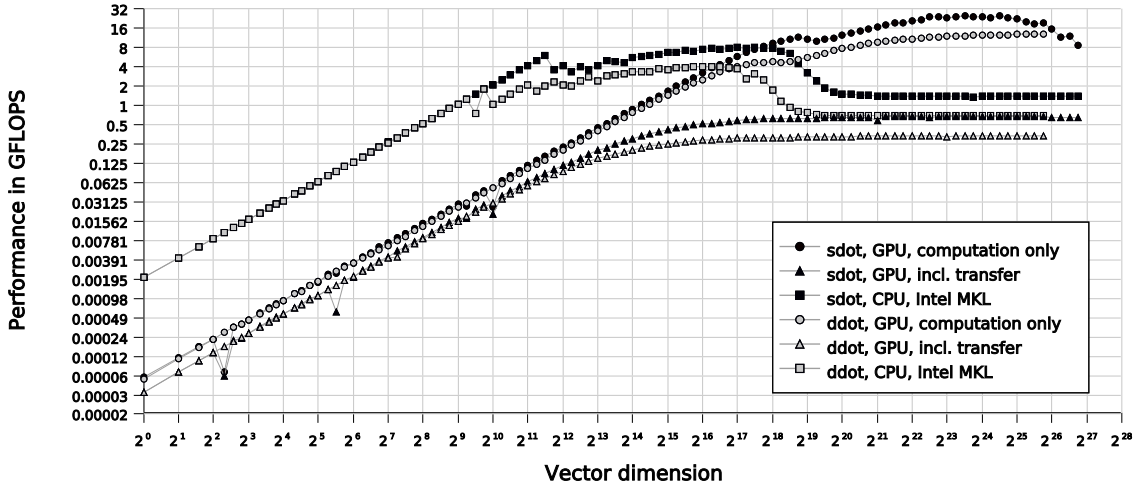

Figure 4.5: *sdot* and *ddot* performance test results

The results in figure 4.5 show hence a significantly higher peak performance, 650 MFLOPS for *sdot* and 325 MFLOPS for *ddot*, including the transfer times. As this increase cannot only be caused by the lower complexity, it is assumable, that the hardware resources have been used more efficiently here.

Still, when including the transfer times, the GPU cannot yet compete with the optimized MKL, that achieves twice the performance on the CPU. Running the kernel on the GPU with all memories initialized, the speed-up for CUBLAS over MKL is again about 19, this time for both *sdot* and *ddot*, given sufficiently large vectors.

Euclidean norm computation - *snrm2/dnrm2*

For a n-vector $\mathbf{x}$, this kernel computes the *Euclidean norm* $\|\mathbf{x}\|_2 = \sum_{i=1}^{n} x_i^2$:

```
for (result=0, i=0; i<n; i++)
    result += x[i] * x[i];
```

For single-precision data, the routine is called *snrm*2 for double-precision *dnrm*2.

While the computations are the same as for *dot*, the data intensity is cut in half, as $\mathbf{y} = \mathbf{x}$. The algorithm is still expected to be memory bound. Thus cutting the amount of data needed in half, would result in a doubling of performance (incl. transfers). Also the plain computation should run faster as now data is re-used. Figure 4.6 shows the expected results, the performance including transfers nearly doubled from 0.65 to 1.28 GFLOPS and also the plain calculation performance increased from 24.8 to 29 GFLOPS for single-precision, double precision results increased similarly.

The MKL shows a strange behaviour for *nrm*2: While the *snrm*2 performance does not drop for large vectors as much as for *dot* and keeps a twice higher level, *dnrm*2 reached only one fourth of the *ddot* performance. The speed-up would this ways calculate to 12 for *snrm*2 and 90 for *dnrm*2 for large vectors. Of course, these are

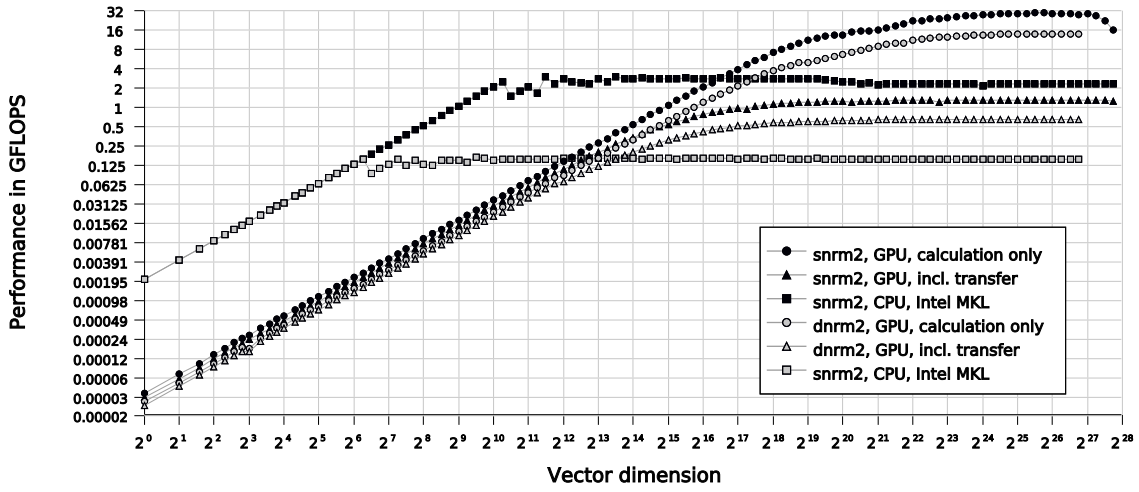

Figure 4.6: *snrm2* and *dnrm2* performance test results

no meaningful results, as the MKL's *dnrm2* implementation should be avoided in favour of *ddot*.

General matrix product - *sgemm/dgemm*

This kernel computes a scaled matrix-matrix product of two general matrices and adds it to a third general matrix, i.e. there is no information available, if the matrix is symmetric, hermitian, etc.. The kernel is able to handle non-quadratic matrices and transposed matrices, but for this test, non-transposed quadratic matrices where chosen. With the two scaling factors α and β the algorithm computes $\alpha \cdot \mathbf{A} \cdot \mathbf{B} + \beta \cdot \mathbf{C}$ and stores the solution in $\mathbf{C}$:

```
for (i=0; i<n; i++)
{
    for (j=0; j<n; j++)
    {
        C[i][j] *= beta;
        for (k=0; k<n; k++)
            C[i][j] += alpha * A[i][k] * B[k][j];
    }
}
```

This time, approximately $3n^3$ operations are executed on approximately $3n^2$ data elements using the naive implementation above. The resulting computational intensity would be $\frac{n}{4}$ FLOP/B for single-precision and $\frac{n}{8}$ FLOP/B for single precision, what results in $0.56n$ GFLOPS SP and $0.28n$ GFLOPS DP. Taking the theoretical peak performances from 3.3.1, we get

$$\begin{aligned} 993.12 = 0.56\, n_{SP}\,\text{FLOP/B} &\iff n_{SP} = 1774 \text{ and} \\ 77.76 = 0.28\, n_{DP}\,\text{FLOP/B} &\iff n_{DP} = 278 \end{aligned}$$

This means that for matrix dimensions larger then 1774 for *sgemm* and 278 for *dgemm* the kernel is not memory bound anymore, but limited by the computational

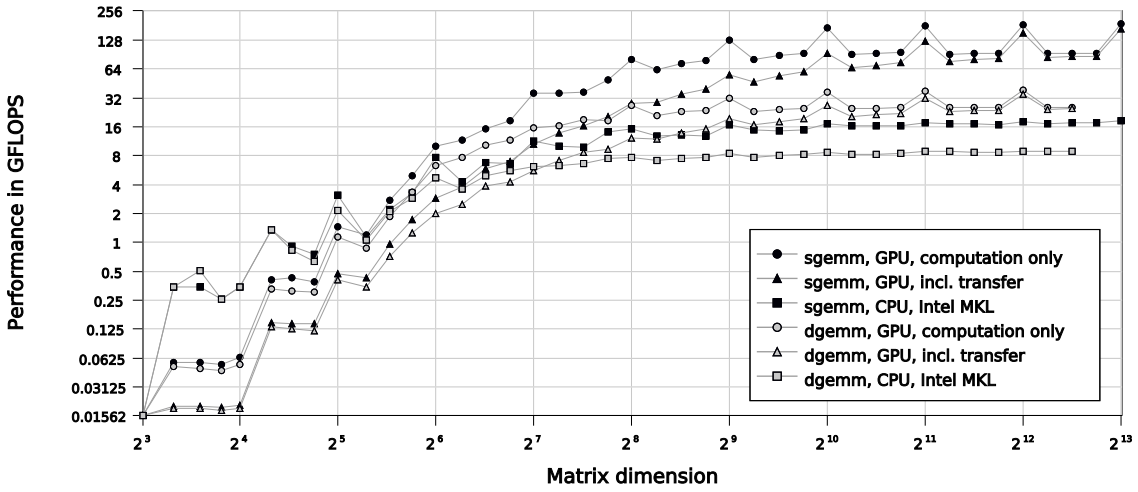

Figure 4.7: *sgemm* and *dgemm* performance test results

power of the GPU. These are again no hard boundaries as their basis was a naive implementation and several more factors were left out.

The results in figure 4.7 show for *sgemm* a behaviour as predicted, at matrix dimensions of about $2^{11} = 2048$ the results with and without transfer times are already quite near each other and approach each other further. For *dgemm*, a little larger dimensions then expected are needed, but at 2^9 a similar behaviour becomes visible. It is hence advisable to use the GPU for any *gemm* if the matrix dimensions exceed $2^8 = 256$, as this is the point when, even with transfer times, CUBLAS is faster then MKL. For larger dimensions the speed-up increases for *sgemm* up to 10 (9 incl. transfers) and up to 4.3 (3.9 incl transfers) for *dgemm*. These values are only achieved for matrix dimensions equalling a power of 2. This early version of CUBLAS shows significant drops in performance for non-power-of-2 matrix dimensions - still performing better then the MKL, though with lower advantages.

4.3.2 Sparse matrix-vector product

The multiplication of a sparse matrix and a general vector, short *spmv*, is part of many scientific computation kernels, e.g. the PDE solver presented in section 6.1.1. For a sparse matrix $\mathbf{A} = (a_{i,j})$, and a vector $\mathbf{x} = (x_i)$ it computes a vector $\mathbf{y} = (y_i)$ by

$$\forall a_{i,j} \neq 0 : \; y_i = y_i + a_{i,j} \cdot x_j \tag{4.1}$$

As depicted in figure 4.8, the matrix $\mathbf{A}$ is primarily populated by zero entries. The number of non-zero entries lies - depending on the definition of sparsity - in $O(n)$ or $O(n \log n)$ for a square matrix of dimension n^2.

In contrast to its importance stands the performance of *spmv*: Conventional implementations on single-core CPUs achieve only about 10 % of the systems peak performance [Vudu03], and modern multi-core CPUs depend on extensive manual optimization to achieve very good results [Supe07].

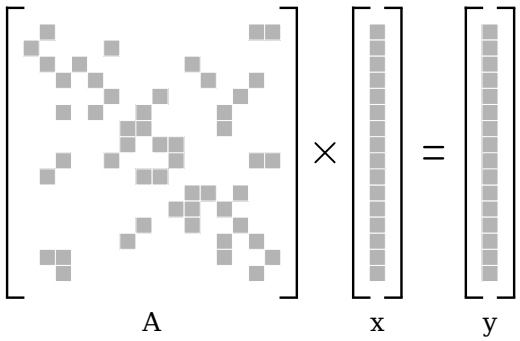

Figure 4.8: Sparse matrix vector multiplication, *spmv*

Again this is due to the impact of the memory system. While *gemm* executes $O(n^3)$ on $O(n^2)$ data elements, *spmv* executes on $O(n)$ or $O(n \log n)$ data elements and needs the same number of operations, cf. (4.1). As now the number of operations and data elements are lying in the same class of complexity, the processor cannot hide memory latency by additional computations. Furthermore the entries of **A** are only needed once in contrast to *gemm*, what makes caching these values ineffective. Only the values of **x** and **y** might be used again, but, because of the sparsity, not in a consecutive fashion.

In order not to intensify the possible problems by storing the zero-entries explicitly in a dense format, special sparse data formats can be used that only use $O(n)$ or $O(n \log n)$ values. There are a variety of formats, optimized for many different matrix structures [BBCD+94], two of them, *COO* and *CSR*, are known to achieve good results when nothing about the matrix's structure is known [Vudu03] and shall thus be shortly explained here.

Coordinate storage - *COO* This is a simple format that is implemented usually by using three data arrays to store the entries $a_{i,j}$ consecutively. The first array holds the value of a non-zero entry $a_{i,j}$, the second and third elements hold the respective row and column indices i and j. Assuming, the matrix has k non-zero entries, it can be stored using $3k$ values.

Compressed sparse row storage - *CSR* This format usually also used three arrays, here named `val`, `ind` and `ptr`. Every row of **A** is treated as a sparse vector, making it possible to easily access a certain row and iterate over its entries. All non-zero entries of $a_{i,j}$ are again stored consecutively in the array `val` and the respective column indices j are placed in `ind`. The third array `ptr` holds this time only the indices of the first element of each row with respect to `val` and `ind`, i.e. `ptr[`i`]` is the offset of the ith row. An additional entry `ptr[`$row\ number\ m+1$`]` stores the number of non-zeros k, as shown in figure 4.9.

In the case of $k \gg m$ only $\approx 2k$ values are needed to store **A**.

In order to benchmark the *spmv* performance on *CPU* and *GPU*, a basic set of routines based on *CSR* has been developed for single and double precision values.

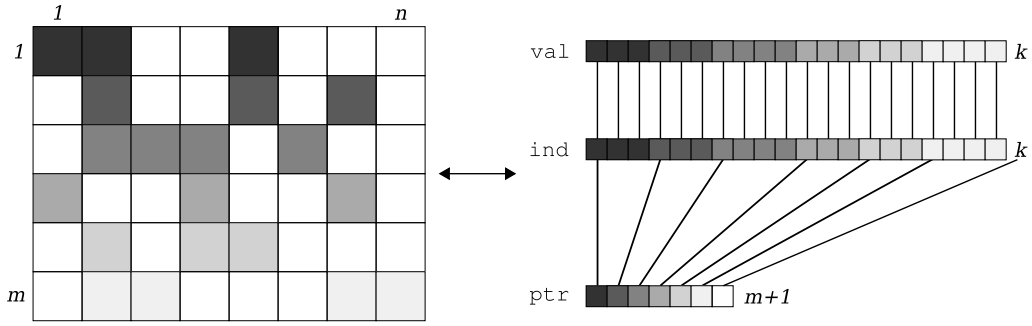

Figure 4.9: Compressed sparse row format for sparse matrices

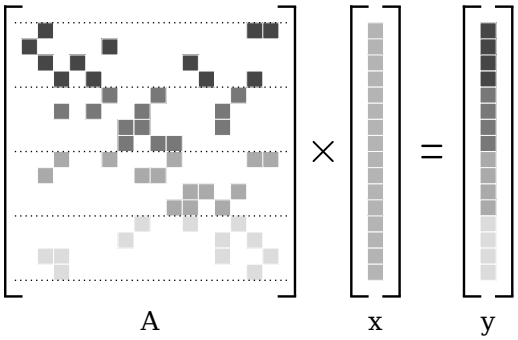

Figure 4.10: Thread blocking for *spmv*

For testing a square matrix of dimension n was filled with $n \log n$ randomly placed random values and a dense vector of length n was randomly generated. On the CPU, the multiplication was executed via the `mkl_scsrmv` kernel of the *SparseBLAS* library, the GPU kernel was manually implemented in CUDA. As mentioned before, data intensity and computational complexity are both in $O(n \log n)$, making the kernel's execution time mainly depending on the memory transfers. The measurements plotted in figure 4.11 relate to a basic scheme for *csrmv*, explained hereafter. The execution times including transfers of all other kernels developed later, did not differ remarkably, that is why these plots are omitted.
The basic implementation is based on (4.1), it iterates over all rows and their entries by

```
foreach (i in rows){
    for (j=ptr[i]; j<ptr[i+1]; j++)
       y[i] += val[j] * x[ind[j]];
```

This scheme is refined using the *thread blocking* technique shown in figure 4.10, that is used to exploit the power of multi-core CPUs. The matrix is divided row-wise into blocks, in most cases according to a fixed row number or an equally distributed number of non-zeros per block. Each core is then responsible for computing the entries of $\mathbf{y}$ belonging to its block. The easiest way to adopt this technique for the many-core GPU is to create one thread for each row, that performs the inner loop of the scheme above. The kernel in Listing 4.1 was used in figure 4.11.

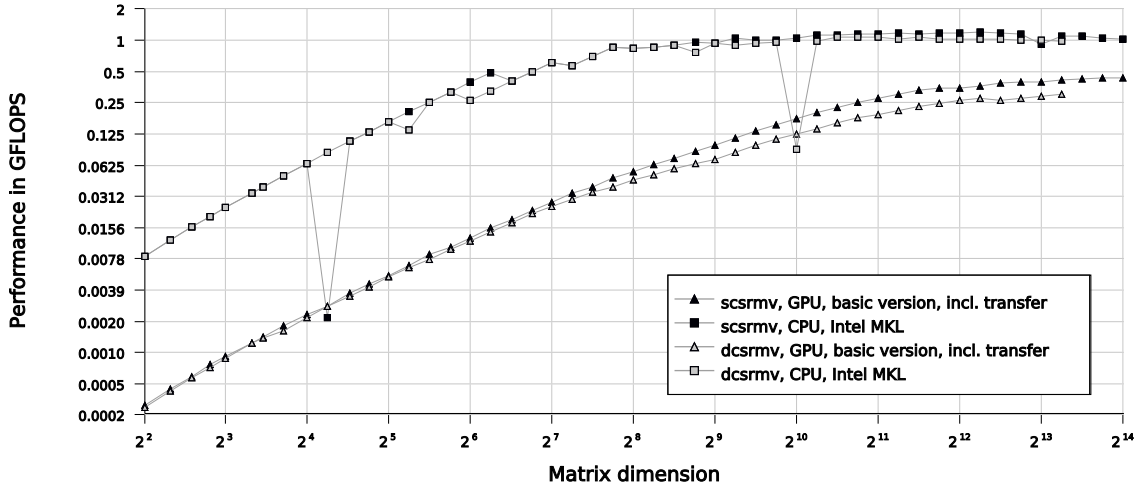

Figure 4.11: Basic *scsrmv* and *dcsrmv* implantation performance test results

```
__global__ void spmv_kernel(int m, float* val, int* pnt, int* ind,
    float* x, float* y)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m) //block dimensions may be greater than the row count
    {
        int   rowStart = pnt[row];
        int   rowEnd   = pnt[row+1];
        float sum      = 0;

        for (int col=rowStart; col<rowEnd; col++)
            sum += val[col] * x[ind[col]];

        y[row] = sum;
    }
}
```

Listing 4.1: Simple spmv kernel

One weakness of this implementation is the fetching `x[ind[col]]`, as it is indirect and can hardly be coalesced, cf. 3.3.5.1. Another weakness is, that the number of non-zeros per row is variable, what creates divergences among the threads, cf. 3.3.5.2. As `x` is only accessed for reading, one idea to speed-up the data transfer is to declare it as a texture and use the `tex1Dfetch` instruction for reading, as explained in 3.3.3.3. Although reads are now cached in theory, the random access pattern of `x` will result in a high number of cache misses. Also, as `val` and `ind` are read consecutively, the hardware is already able to perform coalesced reads, such that declaring these arrays as textures can only be beneficial if prefetching makes global memory read requests superfluous. Although, the values cannot be re-used, the texture will probably not drop them out of cache immediately, and thus uses its size only to limited extent.

These predictions are partly confirmed by the test results shown in figure 4.12, that measured the raw computational performance without host memory transfers. Up to a size, until the matrix data can still be fit into the memory managed by a single controller, the basic kernel competes very well. For larger data, the texture L2 cache (cf. 3.3.1) seems to achieve more cache hits and accelerates nearly every texture combination. The texture unit seems to gain best performance for consecutive `float` data (combinations with `val` as texture) and fetches with the same access pattern (`x`,`val`/`ind`). Declaring only the integer data in `ind` or solely `x` gave now big advantage for large data sizes, and their combination resulted in a even worse performance compared to the basic kernel. Adding `val` and declaring every major data array as texture, can boost the performance again, but does not achieve top results. This is most probably due to the limited cache size, that is overcharged with three large textures.

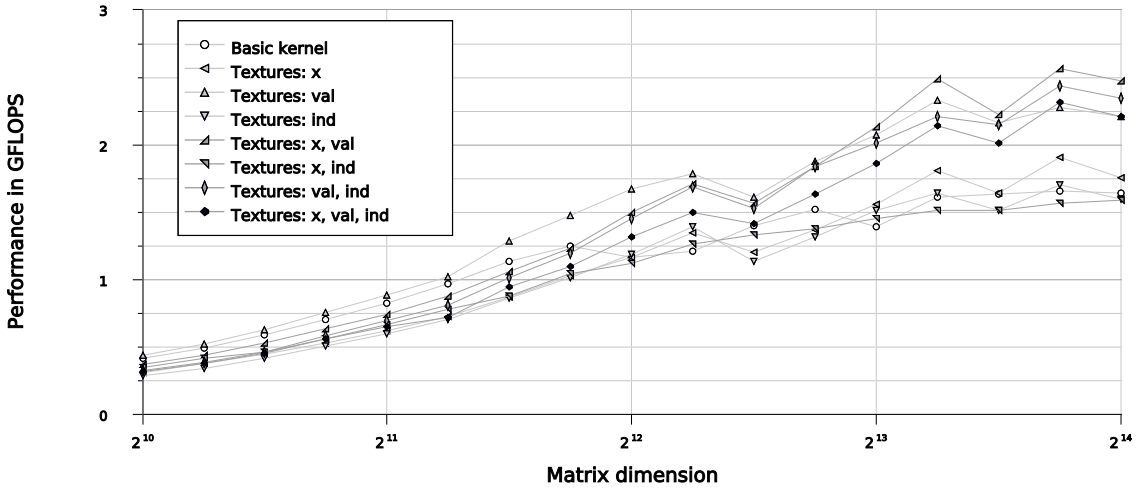

Figure 4.12: *scsrmv* performance test results with different data declared as textures

The algorithm above was implemented such that, transfer times are kept low, by copying `val`, `ind` and `ptr` in a single data block into GPU memory and using pointers to keep track of the exact starting positions. Declaring an area of memory as texture, that was not particularly allocated with `cudaAlloc`, makes the usage of an offset to the texture reference necessary, that must be respected when using texture fetches. It was tested, if these offsets create a read overhead compared to separately allocating and transferring memory for all arrays. The result was a marginal improvement in performance for those cases, that were running with less performance then the original version at maximum matrix dimensions and a slight decrease in performance for those cases running faster then the original version. Though the differences were only marginal, there is probably in fact a read overhead when being forced to use offsets, that is partly compensated by the reduced transfer times through the block transfer.

Texture fetches are yet only available for `float` data, in order to test double precision performance, a workaround had to be implemented: As textures of `int2` are allowed,

the `double` arrays were declared as such and after fetching converted using the intrinsic function `__hiloint2double`. Of course this makes further preparations needed, that are not included in figure 4.13, that only shows the calculation time. The results differ widely from the `float` version. Firstly not a combination, but the single declaration of a texture gave high performance, surprisingly the highest for `x`. Again, up to a certain matrix size, the basic kernel achieved best or second best results. The combination of two or three textures did decrease the performance, again probably due to the limited cache size, that now can only store half the number of entries.

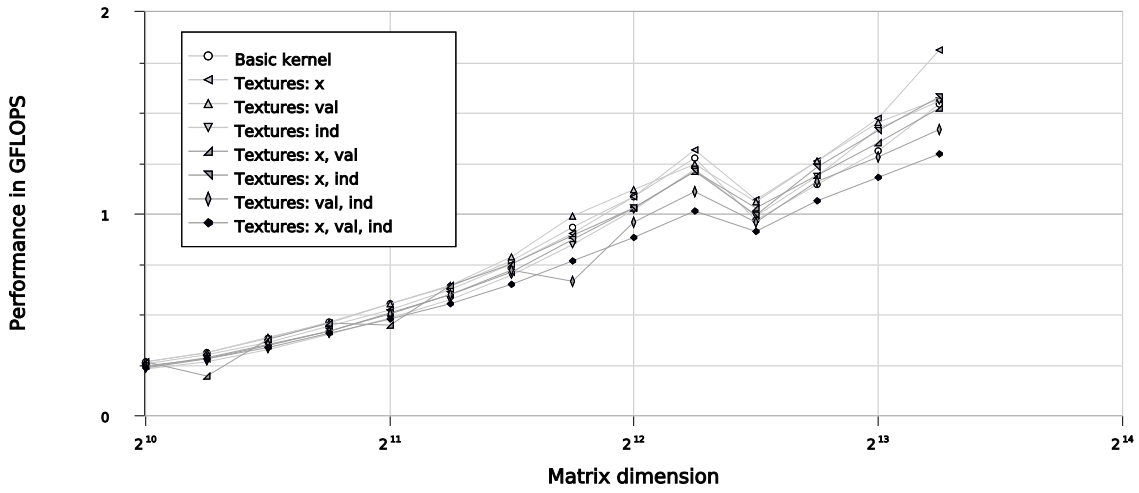

Figure 4.13: *dcsrmv* performance results with different data declared as textures

The aforementioned algorithm versions were implemented in a way, that transfer times are kept low, by copying `val`, `ind` and `ptr` in a single data block into GPU memory and using pointers to keep track of the exact starting positions. Declaring an area of memory as texture, that was not particularly allocated with `cudaAlloc`, makes the usage of an offset to the texture reference necessary, that must be respected when using texture fetches. It was tested, if these offsets create a read overhead compared to separately allocating and transferring memory for all arrays. The result was a slight improvement in performance for those cases, that were running with less performance then the original version at maximum matrix dimensions and a slight decrease in performance for those cases running faster then the original version. A possible explanation is, that there is in fact a read overhead when being forced to use offsets, but it can only partly compensate the prolonged transfer times through separate memory transfers.

It was furthermore experimented with the GPU's *shared memory*. The idea was, to create an array in shared memory of the size of a thread block and let each thread of the respective block fetch one value of `x`. This way `x` can be read coalescedly and accessed quickly by the threads needing its values. This technique of manual prefetching is quite useful for a structured matrix, for an unstructured one, that is

furthermore quite sparse, the effect is rather low or even negative, as shown in figure 4.14. In order to maximize the number of shared memory hits, the thread block size was chosen as large as possible, in this case 512 for both, *scsrmv* and *dscrmv*. This means, that at most the first 512 values of `x` are cached. The results given in figure 4.14 show, that the prefetching is only beneficial for a small number of cases. Also for matrix dimensions smaller then 1024, the basic kernel was faster in 85 % of the test cases.

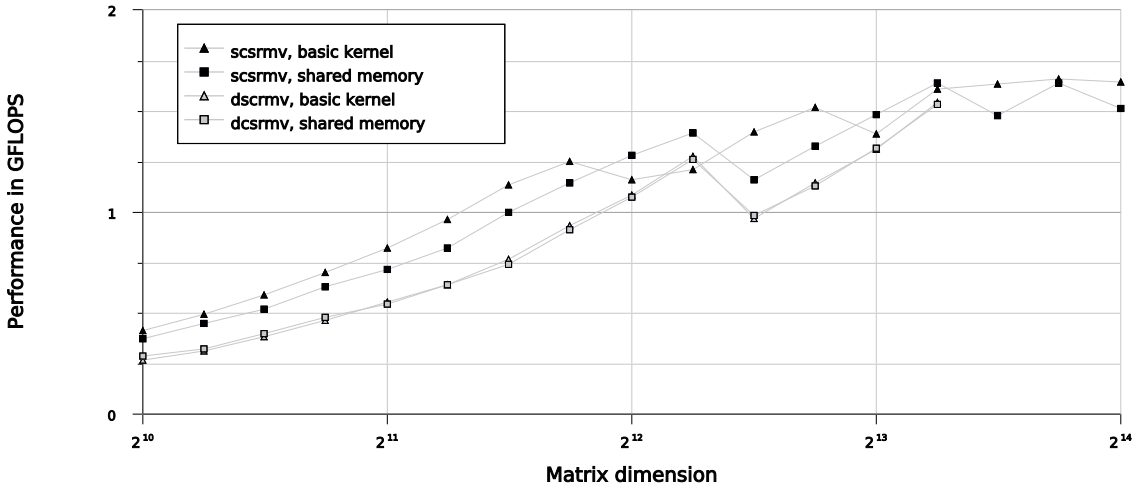

Figure 4.14: *scsrmv* and *dcsrmv* performance test results with shared memory use

Finally, a whole different approach to *spmv* was tested, which is based on segmented scan. The underlying scan algorithms were developed by Sengupta et al. in [SHZO07] and implemented in a library called CUDA Data Parallel Primitives, short CUDPP[3].
The sparse matrix-vector multiplication of CUDPP is also based on CSR and currently only available in single precision. The idea is to reduce the number of divergent threads, caused by variable numbers of non-zeros per row. CUDPP uses a temporary array `product` to compute every product `product[i] = val[i] * x[ind[i]]` in a first step. While doing so a second additional array `flag` of Boolean type is set to 0 for every entry. Afterwards a second kernel iterates over `ptr` and sets every the `flag` for every first entry in a row to 1. Now that, one *segment* per row was created, a scan (backward segmented inclusive sum scan) can be executed on the freshly marked row beginnings, that adds the values in `product` up and saves the row sum in the last element belonging to the row. In a last step only the values in `y` have to be set. The trick is, to use a scan routine to do the addition and not a loop as in the basic algorithm. On the other hand several kernel launches are necessary, each creating a start overhead.
In order to correctly set the additional array structures, *CUDPP* hides memory allocation and takes only CSR data in host memory as input data. Thus it was not possible to measure the raw computational performance and figure 4.15 shows only measurements including transfer times.

[3] www.gpgpu.org/developer/cudpp

Obviously, for small matrix sizes and thus short kernel run-time, *CUDPP* cannot make use of its more sophisticated technique. Only for matrix dimensions larger then 4096×4096, the basic implementation drops behind, and for the largest possible size, even the MKL's performance can nearly be reached.
A big disadvantage is the hidden memory management, as the current CUDA driver version suffers from errors that occur in connection with frequent allocation and deallocation of device memory. Because of that, it was not possible to time CUDPP automatically, but every matrix size had to be measured in a new process in order to avoid program or system crashes. Though the results are promising for large matrices, currently CUDPP is not suitable for variable matrix data processed in a loop.

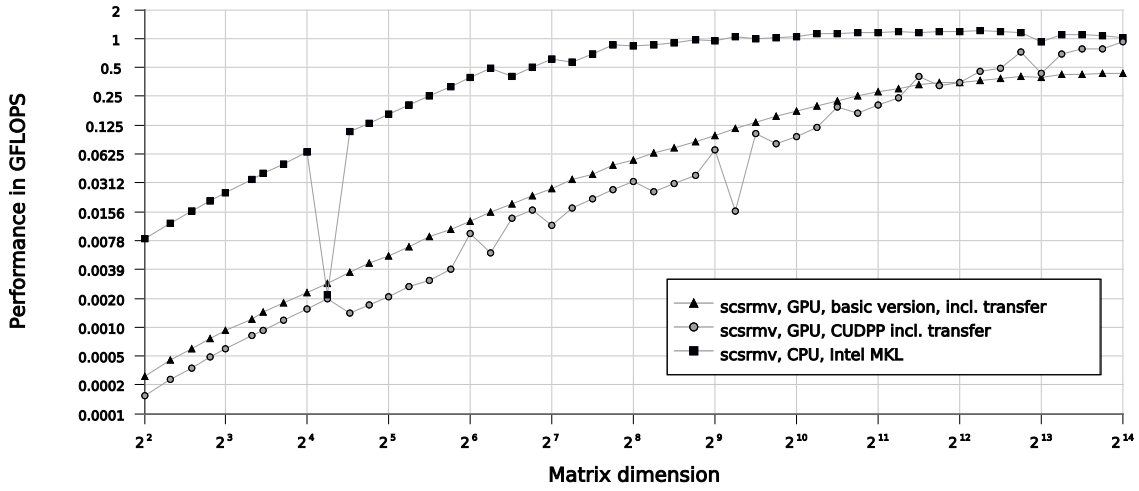

Figure 4.15: *scsrmv* performance results using CUDPP

4.3.3 Discrete Fourier transformation - *DFT/FFT*

The discrete Fourier transformation (DFT) does not only have many applications in signal processing but can also be used in 2D Poisson solvers with Dirichlet boundary conditions [BuGN70]. Another application of the DFT is e.g. fast integer multiplication via the Schönhage-Strassen algorithm [ScSt71].

The discrete Fourier transformation of a vector $\mathbf{x} = (x_0, \dots x_{N-1}) \in \mathbb{C}^N$ is defined as

$$\hat{x}_k = \sum_{n=0}^{N-1} x_j \cdot e^{-2\pi i \frac{nk}{N}} \quad k = 0, \dots, N-1$$

Clearly the complexity of this 1D-DFT is in $O(N \log N)$, as for each of the N components one sum is computed in $O(\log N)$

Generalizing the DFT for higher dimensions is simply done by applying the DFT in each coordinate direction. In the 2D case, the DFT reads

$$\hat{x}_{k,l} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{m,n} \cdot e^{-2\pi i \frac{mk}{M}} e^{-2\pi i \frac{nl}{N}} \quad k = 0, \dots, M-1;\ l = 0, \dots, N-1.$$

Hence, the complexity of a 2D DFT results in M times the complexity of a 1D DFT: $O(MN \log N)$ or $O(N^2 \log N)$ in the square case.

The fast Fourier transformation (FFT) is an algorithm that allows the DFT computation of vectors of a power-of-2 dimension in $O(N \log N)$ on serial processors by re-using previously computed values [CoTu65]. Parallel implementations can speed-up the calculation - depending on the hardware functionality - by a factor up to the number of execution units.

From the test results, presented in figure 4.16, it is clearly visible, that a parallel FFT technique was used whenever possible. The cases, where the matrix dimensions are powers of 2, achieve a far higher performance then the other ones. For this test, a real-to-complex and a complex-to-real transform have been executed successively for different matrix sizes. The time measurements were converted into GFLOPS by assuming $2.5N \log N$ operations for each transform. This is a common convention, that is e.g. also used in the FFTW benchmarks[4].

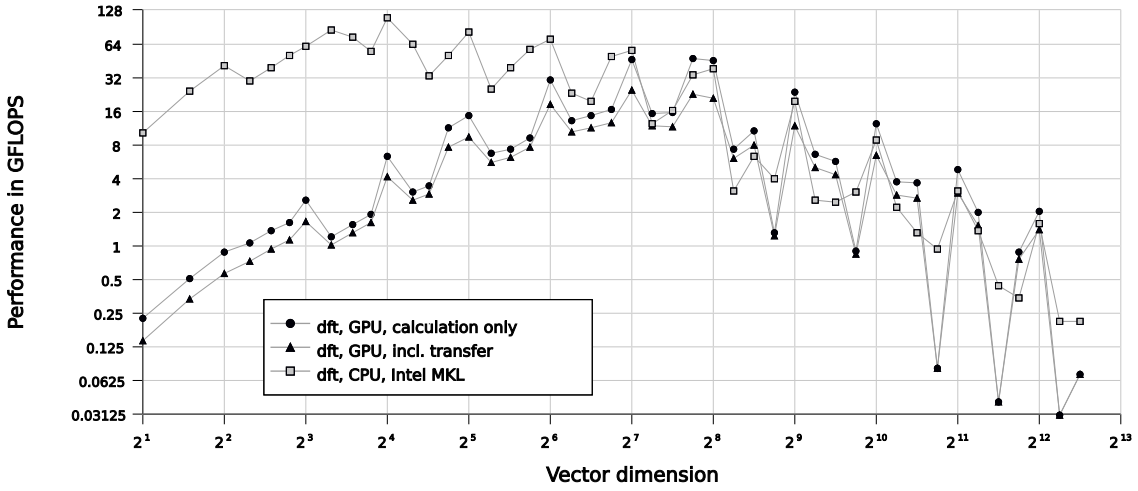

Figure 4.16: 2D-DFT performance test results

Currently CUFFT, the CUDA DFT library, can only outperform MKL for power-of-2 matrix dimension larger then 1024×1024. The current version of CUFFT can be seen as preliminary, it shows considerable performance fluctuations and behaves often unexpectedly. During the creation of this thesis, a much faster 1D-FFT implementation for vectors of up to 8192 elements was presented in the CUDA developers online forum by Vasily Volkov[5], achieving up to 200 GFLOPS (approximately three times faster then CUFFT) and another paper described DFT of arbitrary sizes with CUDA [GLDS+08]. This raises expectations, that faster and more stable DFT implementations in CUDA will be available in the medium term.

4.4 Multi-GPU programming

Up to now, only one of the two ENGTX 280 cards has been uses, the one with the higher host connection bandwidth. The second device or other CUDA capable

[4] http://www.fftw.org/speed/method.html [5] http://www.eecs.berkeley.edu/~volkov/

devices in the system can easily be accessed with the help of the `cudaSetDevice` instruction. In order to manage several devices in parallel, each device has to have its own context, i.e. a new thread or process has to be created.

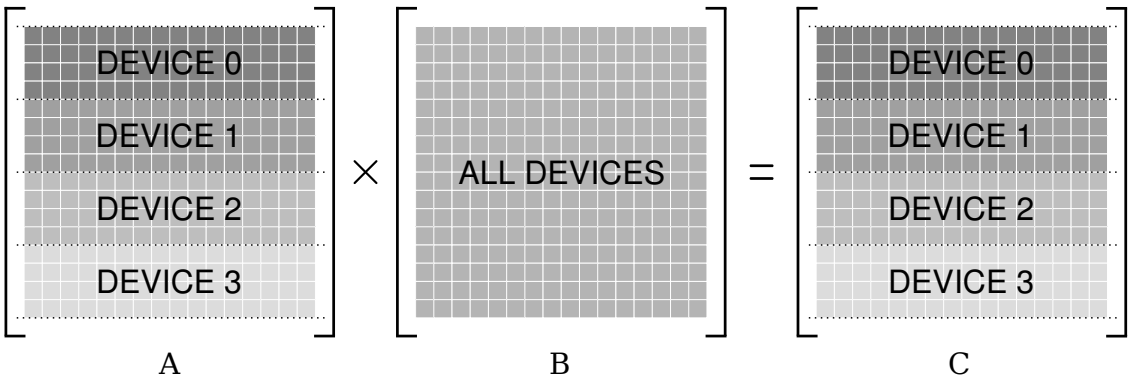

Figure 4.17: Device blocking for parallel *sgemm*

In order to demonstrate the benefits of multi-GPU programming, the *sgemm* kernel (cf. 4.3.1) is revisited. The easiest way to distribute the computational load onto multiple devices is to partition the matrix row-block-wise (for column-major data storage) and let each device compute a stripe of the result matrix, as shown in figure 4.17. This way, no synchronization is needed and the communication costs increase only linearly with the number of devices D ($D \cdot (1/D + 1 + 1/D) = 2 + D$ matrices are communicated), while the computation costs decrease linearly as well ($\frac{1}{D}N^3$ for each parallel device). As *sgemm* is not memory bound for sufficiently large matrices, it is to be expected, that using several devices in parallel will speed up the calculation.

It seems natural, to create different contexts for multi-GPU computing by using `pthreads`, a POSIX thread implementation that is available on every major platform. The following code excerpt uses the non-Microsoft implementation.
Here, a worker function `gemmThread` sets the device via `cudaSetDevice` and calls the respective `CUBLAS` function. This function can later be started as thread for a number of devices (`DEVICES`) with thread specific parameters, e.g. stored in an array of structs, `parameters`. The master programme finishes, after all threads have been joined again:

```
pthread_t threads[DEVICES];

for (int i = 0; i < DEVICES; i++)
        pthread_create(&threads[i], NULL, gemmThread,
            (void*)(parameters + i));

for (int i = 0; i < DEVICES; i++)
   pthread_join(threads[i], NULL);
```

Listing 4.2: Multi-GPU programming with pthreads

As the graphics cards have a different host bandwidths and *sgemm* is memory bound for small matrix dimensions, it is reasonable to distribute the matrix blocks unevenly. The test results in figure 4.18 are hand optimized, meaning, several partition sizes were tested and one with a good overall result was chosen for plotting. In the case of two GPUs, the division was 57 % on device 0 and 43 % on device 1, the division for one GPU and one CPU was 90/10 and the one for two GPUs and CPU was 56/39/5.

The timing was started synchronously for all threads using a barrier. As both

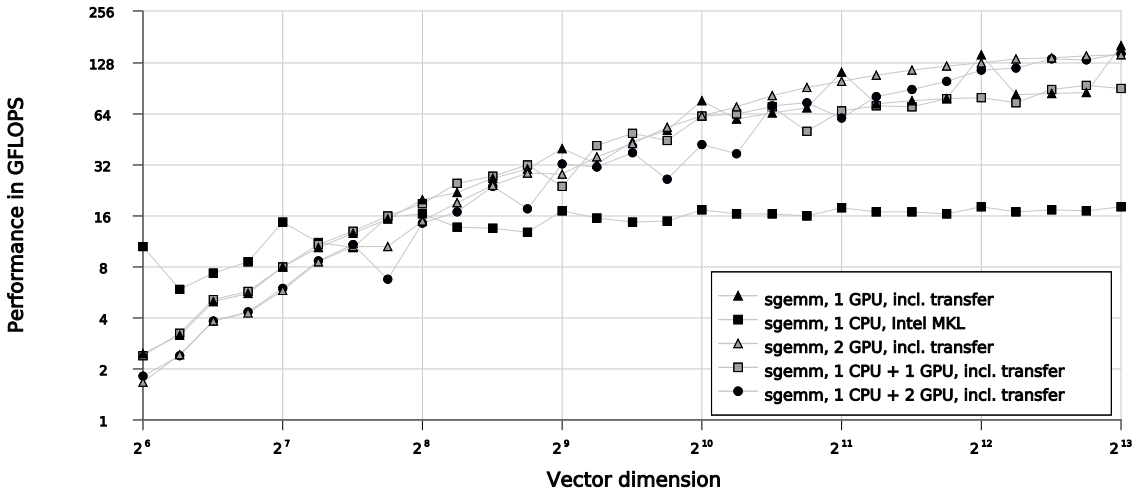

Figure 4.18: *sgemm* performance test results using multiple devices

graphics cards started immediately with memory transfers their performance is in some cases lower then for a single GPU, especially in the power-of-2 cases, when the GPU achieves fairly high performances. Generally speaking, for large matrix sizes, dual-GPU computation gains more stable performance results and achieves speed-ups of 1.5 to 1.7 compared to a single GPU for non-power-of-2 matrices larger then 2048×2048 (for these dimensions kernel is not anymore memory bound, cf. 4.3.1).

Assigning also a small part to the CPU according to a percentage rule did not result in performance improvements. A more sophisticated rule would be needed that assigns power-of-2 matrix blocks to the GPU and leaves the rest to the CPU. But in order to find a decent rule, tests for non-square *sgemm* would have to be conducted in the first place. Using the CPU besides two GPUs was neither beneficial, one additional reason for this could be, that the workstation's dual-core processor is already busy with managing the two GPUs and has not much resources left for own computations.

4.5 Fortran interoperability

Although Fortran's usage in science decreased, mainly in favour of C/C++, still a lot of Fortran programmes are in use and are further developed. In order to use the C oriented CUDA functions in Fortran, *wrapper functions* have to be written in C that

allow Fortran external calls with referenced parameters. In the case of CUBLAS, such a wrapper file is distributed with the library, in the case of CUFFT or self-written kernels, wrappers have to be manually developed. An example of how to call a complex 2D forward in-place FFT from Fortran is given in listings 4.3 and 4.4. Data allocation instructions are not shown here, they can be included as wrapped runtime functions too or taken from CUBLAS (cublasAlloc, cublasSetMatrix, etc.).

```
#include <stdint.h>
#include <cuda_runtime_api.h>
#include "cufft.h"

typedef int devptr_t; //currently 32 bit address space on GPU

#if defined (__cplusplus)
extern "C" {
#endif /* __cplusplus */

int set_cuda_device_ (int *id );
int cufft_plan_2d_ (void *plan, const int *nx, const int *ny, const int
    *type);
int cufft_exec_c2c_ (const int *plan, const devptr_t *idata, const
   devptr_t *odata, const int *direction);
int cufft_destroy_ (const int *plan);

#if defined(__cplusplus)
}
#endif /* __cplusplus */

int set_cuda_device_ (int *id){
  cudaSetDevice(*id);
  return 0;
}

int cufft_plan_2d_ (void *plan, const int *nx, const int *ny, const int
    *type){
        return (int) cufftPlan2d(plan,*nx,*ny,*type);
}

int cufft_exec_c2c_ (const int *plan, const devptr_t *idata, const
   devptr_t *odata, const int *direction){
        void *iPtr = (void *)(uintptr_t)(*idata);
        void *oPtr = (void *)(uintptr_t)(*odata);
        return (int) cufftExecC2C(*plan,iPtr,oPtr,*direction);
}

int cufft_destroy_ (const int *plan){
        return (int) cufftDestroy(*plan);
}
```

Listing 4.3: Wrapped CUDA and CUFFT functions, e.g. wrapper.c

Note that this example was written for `ifort` on Linux, other system may differ in naming conventions (capitalization, name decoration, etc.), pointer sizes and return types. Using `g77` is also possible, but requires the argument `-fno-second-underscore` to use the code as given here.

```
external set_cuda_device, cufft_plan_2d, cufft_exec_c2c, cufft_destroy
integer :: plan
...
call set_cuda_device(1)
call cufft_plan_2d(plan, dim_x,dim_y,41)          !41 = CUFFT_C2C
call cufft_exec_c2c(plan,data_ptr,data_ptr,-1)    !-1 = CUFFT_FORWARD
call cufft_destroy(plan)
```

Listing 4.4: Fortran code excerpt, e.g. fft.f90

4.6 Summary

From the large amount of test results presented in this chapter, we can draw several conclusions:

1. Computationally less intensive kernels, like the level 1 BLAS, are clearly memory bound and benefit only from GPU acceleration within a larger GPU programme.
2. Random accesses slow the GPU performance tremendously as demonstrated in the SpMV test.
3. The GPU achieves the highest speed-ups over the CPU for problem sizes that do not fit into the CPU's cache anymore and use all streaming processors of the GPU efficiently.
4. The NVIDIA libraries leave a lot of room for optimisation. The results for the FFT routine are still quite disappointing.

It had been said in the previous chapter, that performance/watt and peformance/USD ratios are interesting criteria for comparing accelerator hardware. In this chapter we achieved very diverse results reaching from several hundred MFLOPS to 180 GFLOPS, with double-precision versions achieving approximately half the performance in GFLOPS on both, GPU and CPU. Also the speed-ups varied between 0 and 20, assuming the GPU is only used when it is really able to accelerate.

All in all, it is hard to give an assessment on the performance/watt and performance/USD ratios, as the outcome is very sensitive to the actual problem. For this reason, we postpone a final judgement on the GPU's performance until an actual real-world application has been tested and evaluated.

5. Discretization of the model for particulate flows

This chapter describes the discretization process in time and space for the system of equations derived in chapter 2, as well as numerical strategies for its solution.

The following system of equations has been derived so far for $\mathbf{u}$, p, $\mathbf{U}_i$, $\boldsymbol{\omega}_i$ and $\mathbf{X}_i$:

$$\rho_f \frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{F}, \ \nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega, \tag{5.1}$$

$$\Delta M_i \frac{\mathrm{d}\mathbf{U}_i}{\mathrm{d}t} = \Delta M_i \mathbf{g} + V_i \mathbf{c} - \int_{\Omega_{2,i}} \mathbf{F} \, \mathrm{d}\mathbf{x} \quad \text{in } \Omega_{2,i}, \tag{5.2}$$

$$\int_{\Omega_{2,i}} \nabla \times \mathbf{u} \, \mathrm{d}\mathbf{x} = 2 \, \boldsymbol{\omega}_i V_i \quad i = 1, \ldots, N_p, \tag{5.3}$$

$$\frac{\mathrm{d}\mathbf{X}_i}{\mathrm{d}t} = \mathbf{U}_i \quad i = 1, \ldots, N_p, \tag{5.4}$$

$$\mathbf{u} = \mathbf{U}_i + \boldsymbol{\omega}_i \times (\mathbf{x} - \mathbf{X}_i) \quad \text{in } \Omega_{2,i}. \tag{5.5}$$

5.1 Discretization in time with operator splitting

The above problem can be understood as a constrained Navier-Stokes problem. As it is done in most current Navier-Stokes solver, time-discretization is achieved by an *operator splitting* procedure. From the two main families of methods, derived from either the *Peaceman-Rachford method*[PeRa55] or the methods of *Marchuk*[Marc75] and *Yanenko*[Yane71], we choose the latter one. The reason for this being, that the Peaceman-Rachford schemes have several drawbacks, when more then two operators are involved[Glow03]. Here we deal with the following five sub-problems to each of which can be associated a specific operator.

1. **Advection:** The fluid velocity field is updated by only taking the transport term in the Navier-Stokes equations into consideration.
2. **Diffusion:** The fluids viscosity is simulated by applying a diffusion operation to its velocity field.
3. **Projection:** The incompressibility constraint is assured by making the velocity field divergence-free.
4. **Rigid body motion:** The rigid body motion in Ω_2 is imposed on the extended fluid velocity field.
5. **Collision handling:** In case of particle-particle or particle-wall collisions, the particle positions and velocities have to be corrected.

The idea of all operator-splitting methods is to decompose the operator A in the following initial value problem

$$\begin{cases} \dfrac{\mathrm{d}\phi}{\mathrm{d}t} + A(\phi, t) = 0, \\ \phi(0) = \phi_0, \end{cases}$$

where $\phi(t) \in \mathbb{R}^d, \ \forall t > 0, \ \text{and} \ \phi_0 \in \mathbb{R}^d$, into

$$A = \sum_{j=1}^{k} A_j$$

with A_k being individually simpler then A. The A_j are later supposed to be $d \times d$ real matrices, independent of t, which will be achieved by discretization in space.

The decomposed problem is then solved step-wise for $j = 1, \ldots, k$ for $\phi^{n+j/k}$, depicting the value of ϕ at time level $n+1$ after the jth step of the operator-splitting scheme with k partial operators. Clearly $\phi^{n+j/k}$ is also the initial value for the $(j+1)$th step of the scheme. After all k steps have been executed, the result is obviously given by $\phi^{n+k/k} = \phi^{n+1}$:

$$\begin{cases} \dfrac{\mathrm{d}\nu}{\mathrm{d}t} + A_j(\nu, t^{n+1}) = 0 \quad on(t^n, t^{n+1}) \\ \nu(t^n) := \phi^{n+(j-1)/k} \end{cases}$$

$$\phi^{n+j/k} := \nu(t^{n+1})$$

The very simple and only first-order accurate Marchuk/Yanenko scheme implies one step of the backward Euler scheme using a small enough time-discretization step δt to solve the above problem:

Set $\phi^0 := \phi_0$ and for $n \geq 0$, obtain ϕ^{n+1} from ϕ^n via

$$\frac{\phi^{n+j/k} - \phi^{n+(j-1)/k}}{\delta t} + A_j(\phi^{n+j/k}, t^{n+1}) = 0, \ \forall j = 1, \ldots, k$$

The Marchuk/Yanenko scheme is also chosen in [GPHJ99], while [DGMN03] proposes a predictor-corrector scheme that is second-order accurate for single-phase flow, but performs significantly worse in other cases.

As we have seen in chapter 4, bandwidth is a limiting factor for GPU algorithms. The better choice is hence the first-order Marchuk/Yanenko scheme, as it only requires the values computed at the previous sub-step. Furthermore it is hard to estimate, what benefit there is to gain from the formally second-order scheme.

The concrete scheme to be applied is a three-operators splitting, that decouples advection, diffusion and incompressibility, such that the incompressibility constraint can be treated with an L^2-projection method. (For details on the derivation, see chapter VII in [Glow03].)

5.1.1 Advection sub-step

Following the Marchuk/Yanenko splitting idea, the velocity $\mathbf{u}^n$ at time level n is advected by solving the initial value problem

$$\begin{aligned} \frac{\partial \mathbf{u}^{n+1/5}}{\partial t} &= (\mathbf{u}^n \cdot \nabla)\mathbf{u}^{n+1/5}, && \text{on } \Omega, \\ \mathbf{u}^{n+1/5} &= 0, && \text{on } \Gamma, \end{aligned} \tag{5.6}$$

where $\mathbf{u}^{n+1/5}$ is the "advected" velocity field at time level $n+1$. It is to be noted, that the discretization in time by a finite difference has not yet been applied.

This problem can be solved again with several methods, e.g. an upwind scheme, the method of characteristics or the wave-like equation method. As the method of characteristics is comparably tricky to implement and the upwind scheme is discussed in literature almost only for piece-wise linear bases, we are going to apply the wave-like equation method. As its description depends on the finite element discretization in space, it will be explained later.

At this sub-step, the rigid body constraint is not taken into account yet, i.e. $\mathbf{F} = 0$. Thus equation (5.2) is not suitable for gaining a prediction for the particle velocities, it would yield a zero guess. In order to derive an expression for $\mathbf{U}_i$ anyway, equation (5.5) can be differentiated and integrated over $\Omega_{2,i}(t^{n+1})$.

It is to be noted, that variables referring to particles, such as $\mathbf{U}_i$ and $\mathbf{X}_i$, are not involved in the second and third step of the scheme. Hence only three values are computed in all and the notation for these variables is chosen to be $\mathbf{U}_i^{j/3}$, $\mathbf{X}_i^{j/3}$, etc.

$$
\begin{aligned}
\int_{\Omega_{2,i}(t^{n+1})} \frac{\partial \mathbf{u}}{\partial t}\,\mathrm{d}\mathbf{x} &= \int_{\Omega_{2,i}(t^{n+1})} \frac{\partial\left(\mathbf{U}_i + \boldsymbol{\omega}_i \times (\mathbf{x}-\mathbf{X}_i)\right)}{\partial t}\,\mathrm{d}\mathbf{x} \\
\Leftrightarrow \int_{\Omega_{2,i}(t^{n+1})} \frac{\partial \mathbf{u}}{\partial t}\,\mathrm{d}\mathbf{x} &= \int_{\Omega_{2,i}(t^{n+1})} \frac{\partial \mathbf{U}_i}{\partial t}\,\mathrm{d}\mathbf{x} \\
\Rightarrow \int_{\Omega_{2,i}(t^{n+1})} \frac{\mathbf{u}^{n+1/5}-\mathbf{u}^n}{\delta t}\,\mathrm{d}\mathbf{x} &= \int_{\Omega_{2,i}(t^{n+1})} \frac{\mathbf{U}_i^{n+1/3}-\mathbf{U}^n}{\delta t}\,\mathrm{d}\mathbf{x} \\
\Leftrightarrow \int_{\Omega_{2,i}(t^{n+1})} \mathbf{u}^{n+1/5}-\mathbf{u}^n\,\mathrm{d}\mathbf{x} &= V_i\left(\mathbf{U}_i^{n+1/3}-\mathbf{U}^n\right), \quad i=1,\ldots,N_p. \qquad (5.7)
\end{aligned}
$$

An Euler backward step was used here and the fact, that $\mathbf{U}_i$ is constant on $\Omega_{2,i}$.. The prediction $\mathbf{U}_i^{n+1/3}$ will later be corrected by making use of equation (5.2).

Also in this sub-step, a prediction for the center of mass of the ith particle is computed by

$$\mathbf{X}_i^{n+1/3} = \mathbf{X}_i^n + \delta t \mathbf{U}_i^n. \qquad (5.8)$$

This extrapolated value for $\mathbf{X}_i$ will be needed in a later sub-step to determine $\Omega_{2,i}(t^{n+1})$. As $\mathbf{U}^n$ is already present for the previous computation, it is reasonable to do the calculation at this point.

5.1.2 Diffusion sub-step

Now, the "advected" solution is diffused by applying the next operator of the splitting scheme:

$$
\begin{aligned}
&\rho_f \frac{\mathbf{u}^{n+2/5}-\mathbf{u}^{n+1/5}}{\delta t} = \mu_1 \nabla^2 \mathbf{u}^{n+2/5} && \text{in } \Omega, \\
&\mathbf{u}^{n+2/5} = 0 && \text{on } \Gamma.
\end{aligned} \qquad (5.9)
$$

This boundary value problem will later be solved by a finite element method, the required a weak formulation will be given later together with the ones of the other subproblems..

5.1.3 Projection sub-step

The incompressibility constraint is imposed by solving the following system:

$$
\begin{cases}
\rho_f \dfrac{\mathbf{u}^{n+3/5}-\mathbf{u}^{n+2/5}}{\delta t} = -\nabla p^{n+1} & \text{in } \Omega, \\
\nabla \cdot \mathbf{u}^{n+3/5} = 0 & \text{in } \Omega, \\
\mathbf{u}^{n+3/5} = 0 & \text{on } \Gamma.
\end{cases} \qquad (5.10)
$$

For this boundary value problem in $\mathbf{u}^{n+3/5}$ and p^{n+1}, a weak formulation is derived later for the solution with a finite element method.

5.1.4 Rigid body constraint

In [GPHJ99] and [DGMN03] the rigid body constraint is imposed iteratively in contradiction to the idea of the splitting scheme. So far, in each step the use of an explicit method is possible, an important precondition for achieving a high solver speed later. As this should preferably also be possible in this step, an approximation of the solution can be computed explicitly, according to [VeMN07].

Following the Marchuk/Yanenko splitting scheme for (5.1), (5.2) and (5.5) in the space occupied by the particles, we get:

$$\begin{cases} \dfrac{\mathbf{u}^{n+4/5} - \mathbf{u}^{n+3/5}}{\delta t} = \dfrac{1}{\rho_f}\mathbf{F} \quad \text{in } \Omega, & \text{(5.11a)} \\ \dfrac{\mathbf{U}_i^{n+2/3} - \mathbf{U}_i^{n+1/3}}{\delta t} = \mathbf{g} + \dfrac{V_i}{\Delta M_i}\mathbf{c} - \dfrac{1}{\Delta M_i}\displaystyle\int_{\Omega_{2,i}(t^{n+1})} \mathbf{F}\,\mathrm{d}\mathbf{x}, & \text{(5.11b)} \\ \mathbf{u}^{n+4/5} - \left(\mathbf{U}_i^{n+2/3} + \boldsymbol{\omega}_i^{n+1} \times (\mathbf{x} - \mathbf{X}_i^{n+1/3})\right) = 0 \quad \text{in } \Omega_{2,i}(t^{n+1}). & \text{(5.11c)} \end{cases}$$

Substracting (5.11a) and (5.11b) and inserting the rigid body constraint as formulated in (5.11c) yields:

$$\begin{aligned} &\frac{1}{\rho_f}\mathbf{F} - \mathbf{g} - \frac{V_i}{\Delta M_i}\mathbf{c} + \frac{1}{\Delta M_i}\int_{\Omega_{2,i}(t^{n+1})} \mathbf{F}\,\mathrm{d}\mathbf{x} \\ &\quad = \frac{\mathbf{u}^{n+4/5} - \mathbf{u}^{n+3/5}}{\delta t} - \frac{\mathbf{U}_i^{n+2/3} - \mathbf{U}_i^{n+1/3}}{\delta t} \\ &\quad = \frac{\mathbf{U}_i^{n+1/3} - \mathbf{u}^{n+3/5}}{\delta t} + \frac{\mathbf{u}^{n+4/5} - \mathbf{U}_i^{n+2/3}}{\delta t} \\ &\quad = \frac{\mathbf{U}_i^{n+1/3} - \mathbf{u}^{n+3/5}}{\delta t} + \frac{\boldsymbol{\omega}_i^{n+1} \times \left(\mathbf{x} - \mathbf{X}^{n+1/3}\right)}{\delta t} \quad \text{in } \Omega_{2,i}(t^{n+1}),\ i = 1, \ldots, N_p. \end{aligned}$$

Finally, by inserting (5.3), we get

$$\begin{aligned} &\frac{1}{\rho_f}\mathbf{F} - \mathbf{g} - \frac{V_i}{\Delta M_i}\mathbf{c} + \frac{1}{\Delta M_i}\int_{\Omega_{2,i}(t^{n+1})} \mathbf{F}\,\mathrm{d}\mathbf{x} = \frac{\mathbf{U}_i^{n+1/3} - \mathbf{u}^{n+3/5}}{\delta t} + \\ &\frac{1}{2\delta t V_i}\left(\int_{\Omega_{2,i}(t^{n+1})} \nabla \times \mathbf{u}^{n+4/5}\,\mathrm{d}\mathbf{x}\right) \times \left(\mathbf{x} - \mathbf{X}^{n+1/3}\right) \text{ in } \Omega_{2,i}(t^{n+1}),\ i = 1, \ldots, N_p. \end{aligned} \tag{5.12}$$

To gain an expression of the integral of $\mathbf{F}$ we integrate (5.12) over $\Omega_{2,i}(t^{n+1})$. As the integral of the term in the bottom row is equal to zero it can be omitted immediately.

From the upper row we obtain:

$$\left(\frac{1}{\rho_f} + \frac{V_i}{\Delta M_i}\right)\int_{\Omega_{2,i}(t^{n+1})} \mathbf{F}\,\mathrm{d}\mathbf{x} - V_i\mathbf{g} - \frac{V_i^2}{\Delta M_i}\mathbf{c} = \int_{\Omega_{2,i}(t^{n+1})} \frac{\mathbf{U}_i^{n+1/3} - \mathbf{u}^{n+3/5}}{\delta t}\,\mathrm{d}\mathbf{x}.$$

By using $M_i = \rho_i V_i$ and $\Delta M_i = (\rho_i - \rho_f)V_i$, and rearranging in order to get the needed expression, we conclude

$$\begin{aligned}
&\frac{\rho_i V_i}{\rho_f \Delta M_i} \int_{\Omega_{2,i}(t^{n+1})} \mathbf{F}\,\mathrm{d}\mathbf{x} - V_i \mathbf{g} - \frac{V_i^2}{\Delta M_i}\mathbf{c} = \int_{\Omega_{2,i}(t^{n+1})} \frac{\mathbf{U}_i^{n+1/3} - \mathbf{u}^{n+3/5}}{\delta t}\,\mathrm{d}\mathbf{x} \Rightarrow \\
&\frac{1}{\Delta M_i} \int_{\Omega_{2,i}(t^{n+1})} \mathbf{F}\,\mathrm{d}\mathbf{x} = \frac{\rho_f}{\rho_i}\mathbf{g} + \frac{\rho_f \mathbf{c}}{\rho_i(\rho_i - \rho_f)} + \frac{\rho_f}{M_i} \int_{\Omega_{2,i}(t^{n+1})} \frac{\mathbf{U}_i^{n+1/3} - \mathbf{u}^{n+3/5}}{\delta t}\,\mathrm{d}\mathbf{x}.
\end{aligned} \tag{5.13}$$

When inserting (5.13) into (5.12), we get an expression for the global interaction force:

$$\begin{aligned}
\frac{1}{\rho_f}\mathbf{F} &= \sum_{i=1}^{N_p} \Bigg[\mathbf{g} + \frac{V_i}{\Delta M_i}\mathbf{c} - \frac{1}{\Delta M_i} \int_{\Omega_{2,i}(t^{n+1})} \mathbf{F}\,\mathrm{d}\mathbf{x} + \frac{\mathbf{U}_i^{n+1/3} - \mathbf{u}^{n+3/5}}{\delta t} \\
&\quad + \frac{1}{2\delta t V_i} \left(\int_{\Omega_{2,i}(t^{n+1})} \nabla \times \mathbf{u}^{n+4/5}\,\mathrm{d}\mathbf{x} \right) \times \left(\mathbf{x} - \mathbf{X}^{n+1/3} \right) \Bigg] \mathbf{1}_{\Omega_{2,i}} \\
&= \sum_{i=1}^{N_p} \Bigg[\left(1 - \frac{\rho_f}{\rho_i} \right) \mathbf{g} + \frac{\mathbf{c}}{\rho_i} \\
&\quad - \frac{\rho_f}{M_i} \int_{\Omega_{2,i}(t^{n+1})} \frac{\mathbf{U}_i^{n+1/3} - \mathbf{u}^{n+3/5}}{\delta t}\,\mathrm{d}\mathbf{x} + \frac{\mathbf{U}_i^{n+1/3} - \mathbf{u}^{n+3/5}}{\delta t} \\
&\quad + \frac{1}{2\delta t V_i} \left(\int_{\Omega_{2,i}(t^{n+1})} \nabla \times \mathbf{u}^{n+4/5}\,\mathrm{d}\mathbf{x} \right) \times \left(\mathbf{x} - \mathbf{X}^{n+1/3} \right) \Bigg] \mathbf{1}_{\Omega_{2,i}}, \\
&= \sum_{i=1}^{N_p} \Bigg[\left(1 - \frac{\rho_f}{\rho_i} \right) \mathbf{g} + \frac{\mathbf{c}}{\rho_i} + \left(1 - \frac{\rho_f}{\rho_i} \right) \frac{\mathbf{U}_i^{n+1/3}}{\delta t} + \frac{\rho_f}{M_i} \int_{\Omega_{2,i}(t^{n+1})} \frac{\mathbf{u}^{n+3/5}}{\delta t}\,\mathrm{d}\mathbf{x} \\
&\quad - \frac{\mathbf{u}^{n+3/5}}{\delta t} + \frac{1}{2\delta t V_i} \left(\int_{\Omega_{2,i}(t^{n+1})} \nabla \times \mathbf{u}^{n+4/5}\,\mathrm{d}\mathbf{x} \right) \times \left(\mathbf{x} - \mathbf{X}^{n+1/3} \right) \Bigg] \mathbf{1}_{\Omega_{2,i}},
\end{aligned} \tag{5.14}$$

where

$$\mathbf{1}_{\Omega_{2,i}} = \begin{cases} 1, & \text{in } \Omega_{2,i} \\ 0, & \text{in } \Omega \setminus \Omega_{2,i} \end{cases}$$

is the characteristic function of $\Omega_{2,i}$.

Substituting (5.14) in equation (5.11a), and (5.13) in (5.11b), we finally obtain equations for $\mathbf{u}$ and $\mathbf{U}_i$:

$$\begin{aligned}\mathbf{u}^{n+4/5} = \mathbf{u}^{n+3/5} + \sum_{i=1}^{N_p} \Bigg[& \left(1 - \frac{\rho_f}{\rho_i}\right) \delta t \mathbf{g} + \frac{\delta t}{\rho_i} \mathbf{c} \\ & + \left(1 - \frac{\rho_f}{\rho_i}\right) \mathbf{U}_i^{n+1/3} + \frac{\rho_f}{M_i} \int_{\Omega_{2,i}(t^{n+1})} \mathbf{u}^{n+3/5} \, \mathrm{d}\mathbf{x} - \mathbf{u}^{n+3/5} \\ & + \frac{1}{2V_i} \left(\int_{\Omega_{2,i}(t^{n+1})} \nabla \times \mathbf{u}^{n+4/5} \, \mathrm{d}\mathbf{x} \right) \times \left(\mathbf{x} - \mathbf{X}^{n+1/3} \right) \Bigg] \mathbf{1}_{\Omega_{2,i}} \end{aligned} \tag{5.15}$$

$$\mathbf{U}_i^{n+2/3} = \left(1 - \frac{\rho_f}{\rho_i}\right) \delta t \mathbf{g} + \left(1 - \frac{\rho_f}{\rho_i}\right) \mathbf{U}_i^{n+1/3} + \frac{\delta t}{\rho_i} \mathbf{c} + \frac{\rho_f}{M_i} \int_{\Omega_{2,i}(t^{n+1})} \mathbf{u}^{n+3/5} \, \mathrm{d}\mathbf{x}. \tag{5.16}$$

The equation for $\mathbf{u}^{n+4/5}$ contains with $\nabla \times \mathbf{u}^{n+4/5}$ a non-local term, that would lead to a non-sparse matrix when discretized in space. There are two ways of avoiding this problem. On the one hand, an iterative procedure can be applied, on the other hand, [VeMN07] proposes to approximate $\nabla \times \mathbf{u}^{n+4/5}$ with $\nabla \times \mathbf{u}^{n+3/5}$. The advantage would be a fully explicit algorithm for the problem, the disadvantage, that the time step has to be chosen smaller. As we are going to solve the advection sub-problem by an upwind scheme, we already have restrictions on the time step, such that using this approximation here, is in line with the overall method.

As soon as the velocity fields are computed, the position of the particles are updated according to (5.4) by

$$\mathbf{X}_i^{n+2/3} = \mathbf{X}_i^n + \delta t \mathbf{U}_i^{n+2/3}, \quad i = 1, \ldots, N_p. \tag{5.17}$$

5.1.5 Collision handling

Although the collision force terms where included in the previous sub-step, they have not been attributed a value yet. The model described in chapter 2 is based on handling the collisions of one particle at a time, what is going to be done in this step.

According to 2.3.4 for one particle after the other it is checked whether the distances to the walls or other particles have dropped below the defined minima. In this case equations (2.20) and (2.21) are evaluated and the forth sub-step is repeated.

This procedure is reiterated until all minimum separation distances are preserved and the final area occupied by the particles complies with rigid body motion. In the end we gain the final values $\mathbf{X}_i^{n+1}$, $\mathbf{U}_i^{n+1}$ and $\mathbf{u}_i^{n+1}$.

5.2 Discretization in space with finite elements

5.2.1 Triangulation of the domain

The discretization in space will be done with a finite element method, although finite differences or finite volumes would also be possible. For this, a triangulation $\mathcal{T}$ of Ω has to be found, i.e. a partition into a finite number of geometrical bodies such that

- $\bigcup_{\overline{T}\in\mathcal{T}} T = \overline{\Omega}$ and
- $T, T' \in \mathcal{T} \wedge T \neq T' \Rightarrow T \cap T' = \begin{cases} \emptyset & \text{or} \\ \text{a common node} & \text{or} \\ \text{a common edge} & \text{or} \\ \text{a common face.} \end{cases}$

From the vast possibilities of triangulations we focus only on structured ones, because of the benefit to be expected when implementing the simulation on the GPU. Structured grids lead to predictable data accesses that achieve a much higher performance then non-structured ones.

The following three types of geometrical objects are the most obvious possibilities:

- **Tetrahedra:** The triangulation of a cube is shown in figure 5.1, where the dots depict the nodes for interpolation in $\mathcal{V}^1$ (black) and $\mathcal{V}^2$ (black and gray).
- **Hexahedra:** The use of a hexahedral grid reduces the number of elements heavily while keeping the number of nodes stable. As now an element includes 8 or 27 nodes, the $\mathcal{P}_{1/2}$ interpolation is substituted by $\mathcal{Q}_{1/2}$ interpolation, that is approximation by trilinear or triquadratic polynomials. A discretization with cubes or bricks is perfectly sufficient for simple geometries and can be mixed with tetrahedra if necessary.
  In figure 5.2, the black dots depict again nodes used for $\mathcal{Q}_1$ interpolation and the gray dots the additional ones for $\mathcal{Q}_2$.
- **Prisms:** A compromise between tetrahedral and hexahedral meshes would be the use of prisms, also known as wedge and shown together with hexahedra in figure 5.2. This approach is seldom taken as usually one of the two aforementioned ones, performs sufficiently well.

The degrees of freedom, i.e. the unknown values at each node, are the same for each triangulation. Thus, for the systems of equations to be solved, it does not matter which triangulation is chosen. However, it does matter for the computation of the integrals contained in the equations derived in section 5.1. These integrals will later be computed for each element, meaning if less elements are present, the computation time will be shorter.
On the other hand, as we are using a fixed grid that does not adopt to the particles geometries, we might get inaccurate results when evaluating e.g equation (5.16).

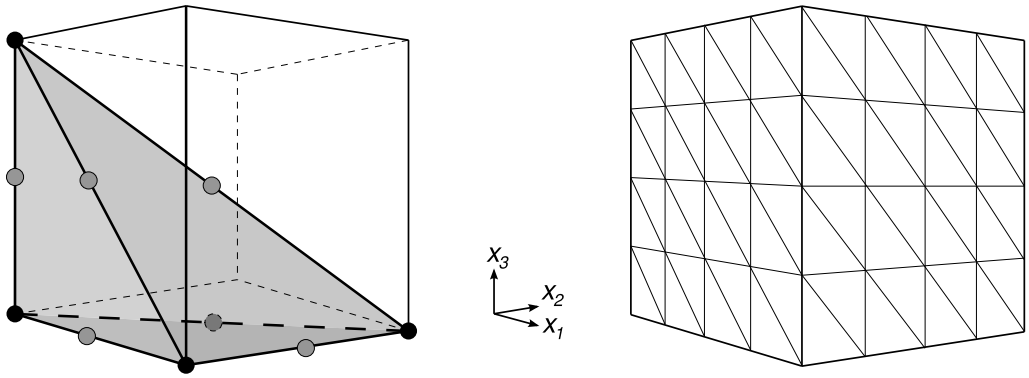

Figure 5.1: Triangulation of $\Omega = (0,1)^3$ with a tetrahedra grid

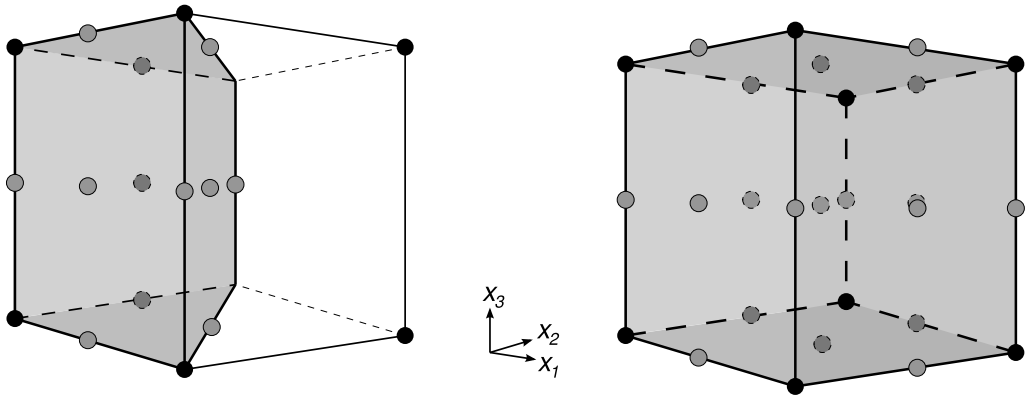

Figure 5.2: Prism shaped and cubic grid elements with corresponding nodes.

There we integrate over the domain occupied by the particles, or in the discrete case over the elements intersected by particles. For elements that are only partly occupied by a particle, $\mathbf{u}$ is most probably discontinuous and the result of the integration can become highly inaccurate.

Tetrahedra are thus chosen for triangulation, in order to gain more accuracy while not enlarging the degrees of freedom.
To gain a structured grid, we divide the cube enclosing the tetrahedron in figure 5.1 into six congruent tetrahedra as shown in figure 5.3.

In the following, we are going to depict by $\mathcal{T}_h$ the triangulation with tetrahedra as shown in figure 5.3 with the enclosing cube having edge length h. Furthermore we are going to number the nodes beginning by 0 and going along the x_1 axis, thereafter in x_2 and finally in x_3 direction.

5.2.2 Choice of elements

For reasons of stability, it is usual to use mixed finite elements for the solution of the velocity and pressure in the Navier-Stokes equation. As the derived scheme is only of first order accurate, a low order approximation of pressure and velocity will be sufficient.

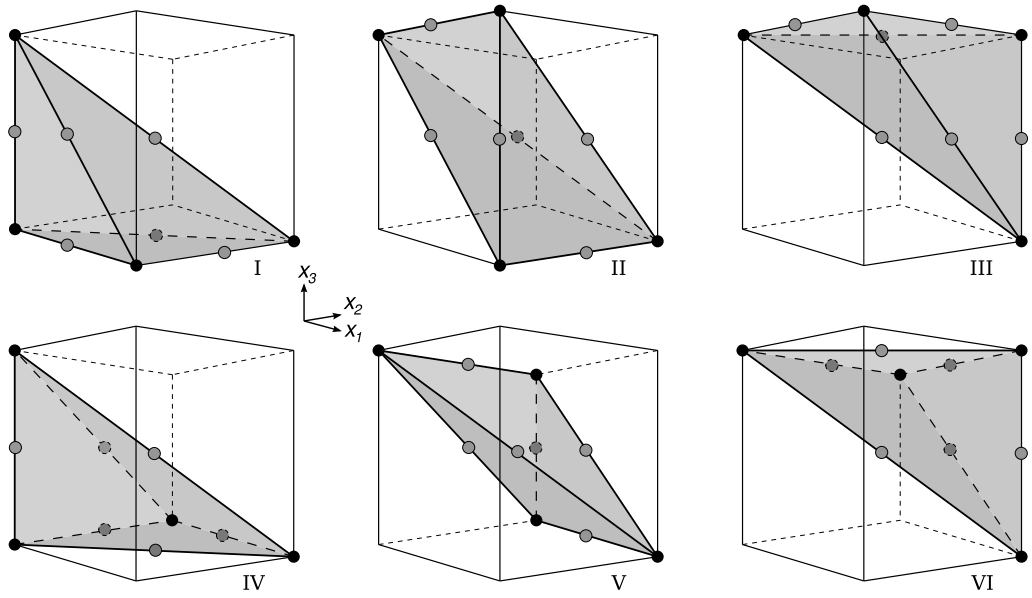

Figure 5.3: Tetrahedra used for triangulation of Ω.

As tetrahedra were chosen for triangulation of Ω, $\mathcal{P}_2/\mathcal{P}_1$ *Taylor-Hood elements* are the obvious choice. These elements satisfy the *inf-sup condition*, i.e. they are guaranteed to provide solvability for any grid, and have been widely used together with discretization in time by operator splitting.

$\mathcal{P}_2$ interpolation is used for the velocity $\mathbf{u}$ and $\mathcal{P}_1$ interpolation for the pressure p. The respective finite dimensional solution spaces used later can now be defined as follows

$$\mathcal{V}_h^2 = \left\{ \boldsymbol{\nu}_h \,|\, \boldsymbol{\nu}_h \in C(\overline{\Omega})^3 \cap \mathcal{H}_0^1(\Omega)^3,\, \boldsymbol{\nu}_h|_T \in \mathcal{P}_2,\, \forall T \in \mathcal{T}_h \right\},$$
$$\mathcal{V}_h^1 = \left\{ q_h \,|\, q_h \in C(\Omega) \cap L_{2,0}(\Omega)\,,\, q_h|_T \in \mathcal{P}_1,\, \forall T \in \mathcal{T}_h \right\},$$

where as usual $\mathcal{P}_i$ is the space of the polynomials in three variables of degree $\leq i$, $L_{2,0}$ is the space of square Lebesgue[1] integrable functions with integral 0 and $\mathcal{H}$ the Sobolev[2] space with square integrable derivatives up to second order and zero boundary conditions.

Referring to figure 5.1, the black dots depict the pressure and velocity nodes and the gray ones additional nodes for the velocity.

5.2.3 The Galerkin finite element method

We are going to approximate $\mathbf{u}$ and p in the finite-dimensional spaces $\mathcal{V}_h^2$ and $\mathcal{V}_h^1$ by a Galerkin[3] finite element method. In order to solve the boundary value problems formulated in (5.9) and (5.10), weak formulation have to be derived in a first step. For this, we multiply by an arbitrary but fixed test function $\mathbf{v}$ and q, and integrate over Ω.

[1] Henri Léon Lebesgue ($\star$ 1875, † 1941) [2] Sergei Lvovich Sobolev ($\star$ 1908, † 1989)
[3] Boris Grigoryevich Galerkin($\star$ 1871, † 1945)

In a second step, we substitute the variables by piecewise linear functions $p_h \in \mathcal{V}_h^1$ and piecewise quadratic functions $\mathbf{u}_h \in \mathcal{V}_h^2$ by

$$\mathbf{u}_h = \sum_{k=1}^{3N} \mathbf{u}_k : \boldsymbol{\psi}_k \qquad p_h = \sum_{k=1}^{M} p_k \chi_k \tag{5.18}$$

where $\{\boldsymbol{\psi}_i\}_{i=1}^{3N}$ is a basis of $\mathcal{V}_h^2$, $\{\chi_i\}_{i=1}^{M}$ is a basis of $\mathcal{V}_h^1$, with M and N being the respective numbers of nodes in $\mathcal{T}$ and " : " denotes the elment-wise product.
Both bases are chosen such that for each node $\boldsymbol{\xi}_j$: $\chi_i(\boldsymbol{\xi}_j) = \delta_{i,j} \quad j = 1, \ldots, M$ and

$$\boldsymbol{\psi}_{d\cdot i}(\boldsymbol{\xi}_j) = \delta_{ij} \begin{pmatrix} \delta_{d1} \\ \delta_{d2} \\ \delta_{d3} \end{pmatrix} \quad d = 1, 2, 3, \; i, j = 1, \ldots, N$$

It is to be noted, that the Galerkin finite element method relies on using the same bases $\{\boldsymbol{\psi}_i\}$ and $\{\chi_i\}$, to span the test spaces for $\mathbf{v}$ and q.

In the following the subindex h is mostly omitted, as it is obvious when dealing with a discrete description.

5.2.3.1 Weak formulations of sub-problems

Advection sub-problem

As discussed before, a wave-like equation method will be used to compute $\mathbf{u}^{n+1/5}$ determined by the transport equation (5.6), which read:

$$\frac{\partial \mathbf{u}^{n+1/5}}{\partial t} = (\mathbf{u}^n \cdot \nabla)\mathbf{u}^{n+1/5}.$$

It is first to notice, that the above equations can be solved independently for the single components $\mathbf{u} = (u_{x_1}, u_{x_2}, u_{x_3})$:

$$\frac{\partial u_{x_d}^{n+1/5}}{\partial t} = (\mathbf{u}^n \cdot \nabla)\, u_{x_d}^{n+1/5}, \quad d = 1, 2, 3. \tag{5.19}$$

The wave-like equation method as explained e.g. in [Glow03], uses the fact that a solution to the transport problem is formally also a solution to the wave equation

$$\frac{\partial^2 \phi}{\partial t^2} = a^2 \nabla^2 \phi.$$

In order to produce this equation structure, we differentiate (5.19) by t and substitute the gradient in the right-hand side by inserting (5.19):

$$\frac{\partial^2 u_{x_d}^{n+1/5}}{\partial t^2} \left(= (\mathbf{u}^n \cdot \nabla) \frac{\partial u_{x_d}^{n+1/5}}{\partial t} \right) = \mathbf{u}^n \cdot \nabla \left(\mathbf{u}^n \cdot \nabla u_{x_d}^{n+1/5} \right), \qquad d = 1, 2, 3.$$

Note that besides the misleading notation, u is still continuous on (t^n, t^{n+1}) here, hence the above equation as to be fulfilled for all $t \in (t^n, t^{n+1})$. Besides the obvious initial condition $u_{x_d}^{n+1/5}(t^n) = u_{x_d}^n$, we now also have to fulfill $\dfrac{\partial u_{x_d}^{n+1/5}}{\partial t}(t^n) = \mathbf{u}^n \cdot \nabla u_{x_d}^n$ because of the differentiation.

In [Glow03] it is proven that this problem has a unique solution, that we are going to compute approximated in space with finite elements and in time with finite differences. The needed weak formulation reads for $t \in (t^n, t^{n+1})$:
Find $u_{x_d}^{n+1/5} \in \mathcal{H}_0^1(\Omega)$*, such that* $\forall v \in \mathcal{H}_0^1(\Omega)$*:*

$$\int_\Omega \frac{\partial^2 u_{x_d}^{n+1/5}}{\partial t^2} v - (\mathbf{u}^n \cdot \nabla u_{x_d}^{n+1/5})(\mathbf{u}^n \cdot \nabla v)\,\mathrm{d}\mathbf{x} = 0,$$
$$u_{x_d}^{n+1/5}(t^n) = u_{x_d}^n, \quad \frac{\partial u_{x_d}^{n+1/5}}{\partial t}(t^n) = \mathbf{u}^n \cdot \nabla u_{x_d}^n.$$

For the time discretization, we follow [Glow03] further and set $\delta t' := \frac{\delta t}{Q}$, with Q being a positive integer, and $u_{x_d}^{n+0/(5Q)} := u_{x_d}^n$. We then compute $u_{x_d}^{n+1/(5Q)}$ and $u_{x_d}^{n-1/(5Q)}$ from

$$\int_\Omega \frac{u_{x_d}^{n+1/(5Q)} - u_{x_d}^{n-1/(5Q)}}{2\delta t'} v - \mathbf{u}^n \cdot \nabla u_{x_d}^n v\,\mathrm{d}\mathbf{x} = 0,$$
$$\int_\Omega \frac{u_{x_d}^{n+1/(5Q)} - 2u_{x_d}^{n+0/(5Q)} + u_{x_d}^{n-1/(5Q)}}{(\delta t')^2} v - (\mathbf{u}^n \cdot \nabla u_{x_d}^n)(\mathbf{u}^n \cdot \nabla v)\,\mathrm{d}\mathbf{x} = 0.$$

In order to facilitate computation we can manually solve for $u_{x_d}^{n-1/(5Q)}$ and remain:

$$\int_\Omega u_{x_d}^{n+1/(5Q)} v\,\mathrm{d}\mathbf{x} = \int_\Omega u_{x_d}^n v + \delta t' \mathbf{u}^n \cdot \nabla u_{x_d}^n v + \frac{(\delta t')^2}{2}(\mathbf{u}^n \cdot \nabla u_{x_d}^n)(\mathbf{u}^n \cdot \nabla v)\,\mathrm{d}\mathbf{x}. \tag{5.20}$$

For the other $q = 1, \ldots, Q-1$ we solve $\forall v \in \mathcal{H}_0^1(\Omega)$:

$$\int_\Omega \frac{u_{x_d}^{n+(q+1)/(5Q)} - 2u_{x_d}^{n+q/(5Q)} + u_{x_d}^{n+(q-1)/(5Q)}}{(\delta t')^2} v - (\mathbf{u}^n \cdot \nabla u_{x_d}^{n+q/(5Q)})(\mathbf{u}^n \cdot \nabla v)\,\mathrm{d}\mathbf{x} = 0,$$

or sorted for knowns and unknowns:

$$\int_\Omega u_{x_d}^{n+(q+1)/(5Q)} v\,\mathrm{d}\mathbf{x} =$$
$$\int_\Omega (2u_{x_d}^{n+q/(5Q)} - u_{x_d}^{n+(q-1)/(5Q)}) v + (\delta t')^2 (\mathbf{u}^n \cdot \nabla u_{x_d}^{n+q/(5Q)})(\mathbf{u}^n \cdot \nabla v)\,\mathrm{d}\mathbf{x}. \tag{5.21}$$

The fully discrete scheme based on (5.20) and (5.21) reads then:
Find $\left(u_{x_d,i}^{n+1/(5Q)}\right)_{d=1,\dots,3,\ i=1,\dots,N}$, *such that*

$$\begin{aligned}
&\int_\Omega \left(\sum_{i=1}^{N} u_{x_d,i}^{n+1/(5Q)}\psi_i\right)\psi_j \,\mathrm{d}\mathbf{x} \\
&= \int_\Omega \left(\sum_{i=1}^{N} u_{x_d,i}^{n}\psi_i\right)\psi_j + \delta t' \left(\sum_{i=1}^{N} u_{x_d,i}^{n}\left(\sum_{e=1}^{3}\sum_{k=1}^{N} u_{x_e,k}^{n}\psi_k\frac{\partial\psi_i}{\partial x_e}\right)\right)\psi_j \\
&+ \frac{(\delta t')^2}{2}\left(\sum_{i=1}^{N} u_{x_d,i}^{n}\left(\sum_{e=1}^{3}\sum_{k=1}^{N} u_{x_e,k}^{n}\psi_k\frac{\partial\psi_i}{\partial x_e}\right)\right)\left(\sum_{f=1}^{3}\sum_{l=1}^{N} u_{x_f,l}^{n}\psi_l\frac{\partial\psi_j}{\partial x_f}\right)\,\mathrm{d}\mathbf{x}, \\
&\quad \text{for } j=1,\dots,N, \qquad (5.22)
\end{aligned}$$

and $\left(u_{x_d,i}^{n+(q+1)/(5Q)}\right)_{d=1,\dots,3,\ i=1,\dots,N,\ q=1,\dots,Q-1}$, *such that*

$$\begin{aligned}
&\int_\Omega \left(\sum_{i=1}^{N}\left(u_{x_d,i}^{n+(q+1)/(5Q)}\right)\psi_i\right)\psi_j \,\mathrm{d}\mathbf{x} \\
&= \int_\Omega \left(\sum_{i=1}^{N}\left(2u_{x_d,i}^{n+q/(5Q)} - u_{x_d,i}^{n+(q-1)/(5Q)}\right)\psi_i\right)\psi_j \\
&+ (\delta t')^2\left(\sum_{i=1}^{N} u_{x_d,i}^{n+q/(5Q)}\left(\sum_{e=1}^{3}\sum_{k=1}^{N} u_{x_e,k}^{n}\psi_k\frac{\partial\psi_i}{\partial x_e}\right)\right)\left(\sum_{f=1}^{3}\sum_{l=1}^{N} u_{x_f,l}^{n}\psi_k\frac{\partial\psi_j}{\partial x_f}\right)\,\mathrm{d}\mathbf{x}, \\
&\quad \text{for } j=1,\dots,N. \qquad (5.23)
\end{aligned}$$

Diffusion sub-problem

By applying the described procedure, we transform (5.9) into the weak problem:
Find $\mathbf{u}^{n+2/5}\in\mathcal{H}_0^1(\Omega)^3$ *such that* $\forall\mathbf{v}\in\mathcal{H}_0^1(\Omega)^3$

$$\frac{\rho_f}{\delta t}\int_\Omega\left(\mathbf{u}^{n+2/5}-\mathbf{u}^{n+1/5}\right)\mathbf{v}\,\mathrm{d}\mathbf{x} = -\mu_1\int_\Omega\nabla\mathbf{u}^{n+2/5}\nabla\mathbf{v}\,\mathrm{d}\mathbf{x}.$$

Sorting the unknowns to the left and shifting the divergence in the right side to $\mathbf{v}$, we obtain

$$\int_\Omega\frac{\rho_f}{\delta t}\mathbf{u}^{n+2/5}\mathbf{v}+\mu\nabla\mathbf{u}^{n+2/5}\nabla\mathbf{v}\,\mathrm{d}\mathbf{x} = \int_\Omega\frac{\rho_f}{\delta t}\mathbf{u}^{n+1/5}\,\mathrm{d}\mathbf{x}.$$

This equation is discretized by inserting (5.18):

Find $\left(\mathbf{u}_i^{n+2/5}\right)_{i=1,\dots,3N}$ *(and thereby* $\mathbf{u}_h^{n+2/5}\in\mathcal{V}_h^2$*), such that*

$$\begin{aligned}
&\int_\Omega\frac{\rho_f}{\delta t}\left(\sum_{i=1}^{3N}\mathbf{u}_i^{n+2/5}:\boldsymbol{\psi}_i\right)\boldsymbol{\psi}_j+\mu\nabla\left(\sum_{i=1}^{3N}\mathbf{u}_i^{n+2/5}:\boldsymbol{\psi}_i\right)\nabla\boldsymbol{\psi}_j\,\mathrm{d}\mathbf{x} = \\
&\int_\Omega\frac{\rho_f}{\delta t}\left(\sum_{i=1}^{3N}\mathbf{u}_i^{n+1/5}\boldsymbol{\psi}_i\right)\,\mathrm{d}\mathbf{x}\quad\text{for } j=1,\dots,3N,
\end{aligned}$$

what can be rearranged to get

$$\sum_{i=1}^{3N}\left(\frac{\rho_f}{\delta t}\mathbf{u}_i^{n+2/5}\int_\Omega \boldsymbol{\psi}_i:\boldsymbol{\psi}_j\,\mathrm{d}\mathbf{x}+\mu\mathbf{u}_i^{n+2/5}\int_\Omega \nabla\boldsymbol{\psi}_i:\nabla\boldsymbol{\psi}_j\,\mathrm{d}\mathbf{x}\right)=$$
$$\sum_{i=1}^{3N}\frac{\rho_f}{\delta t}\mathbf{u}_i^{n+1/5}:\int_\Omega \boldsymbol{\psi}_i\,\mathrm{d}\mathbf{x}\quad\text{for } j=1,\dots,3N. \tag{5.24}$$

Currently, the integrals in this equations are to be understood component-wise, we will later treat the the three space-dimensions independently as done for the advection.

Projection sub-problem

The derivation of the weak problem is again straight forward:

Find $p^{n+1}\in\mathcal{H}^1(\Omega)$ *such that*

$$\int_\Omega \nabla p^{n+1}\nabla q\,\mathrm{d}\mathbf{x}=\frac{\rho_f}{\delta t}\int_\Omega \mathbf{u}^{n+2/5}\nabla q\,\mathrm{d}\mathbf{x}\quad\forall q\in\mathcal{H}^1(\Omega),$$

and $\mathbf{u}^{n+3/5}\in\mathcal{H}_0^1(\Omega)^3$ *such that*

$$\frac{\rho_f}{\delta t}\int_\Omega\left(\mathbf{u}^{n+3/5}-\mathbf{u}^{n+2/5}\right)\mathbf{v}\,\mathrm{d}\mathbf{x}=\int_\Omega p^{n+1}\left(\nabla\cdot\mathbf{v}\right)\,\mathrm{d}\mathbf{x}\quad\forall\mathbf{v}\in\mathcal{H}_0^1(\Omega)^3.$$

This is discretized into

Find $\left(p_i^{n+1}\right)_{i=1,\dots,M}$, *such that*

$$\sum_{i=1}^{M}p_i^{n+1}\int_\Omega\nabla\chi_i\nabla\chi_j\,\mathrm{d}\mathbf{x}=\sum_{i=1}^{3N}\frac{\rho_f}{\delta t}\mathbf{u}_i^{n+2/5}\int_\Omega\boldsymbol{\psi}_i:\nabla\chi_j\,\mathrm{d}\mathbf{x}\quad\text{for } j=1,\dots,M, \tag{5.25}$$

and $\left(\mathbf{u}_i^{n+3/5}\right)_{i=1,\dots,3N}$, *such that*

$$\sum_{i=1}^{3N}\frac{\rho_f}{\delta t}\left(\mathbf{u}_i^{n+3/5}-\mathbf{u}_i^{n+2/5}\right)\int_\Omega\boldsymbol{\psi}_i:\boldsymbol{\psi}_j\,\mathrm{d}\mathbf{x}=\sum_{i=1}^{M}p_i^{n+1}\int_\Omega\chi_i\left(\nabla\cdot\boldsymbol{\psi}_j\right)\,\mathrm{d}\mathbf{x}$$
$$\text{for } j=1,\dots,3N. \tag{5.26}$$

Rigid body motion sub-problem

A discretization of equations (5.15) and (5.16) is needed here. As we are only dealing with constant quantities, the correction can be done node wise, as soon as the integral values have been computed. Namely, these integrals were

$$\int_{\Omega_{2,i}(t^{n+1})}\mathbf{u}^{n+3/5}\,\mathrm{d}\mathbf{x}\quad\text{and}\quad\int_{\Omega_{2,i}(t^{n+1})}\nabla\times\mathbf{u}^{n+3/5}\,\mathrm{d}\mathbf{x}.$$

Let N_i be the number of velocity nodes in $\Omega_{2,i}(t^{n+1})$, then the discrete versions read:

$$\sum_{j=0}^{3N_i} \mathbf{u}_j^{n+3/5} : \int_{\Omega_{2,i}(t^{n+1})} \boldsymbol{\psi}_j \, \mathrm{d}\mathbf{x} \quad \text{and} \tag{5.27}$$

$$\begin{aligned} &\sum_{j=0}^{3N_i} \int_{\Omega_{2,i}(t^{n+1})} \nabla \times (\mathbf{u}_j^{n+3/5} : \boldsymbol{\psi}_j) \, \mathrm{d}\mathbf{x} \\ =&\sum_{j=0}^{3N_i} \begin{bmatrix} u_{x_3} \displaystyle\int_{\Omega_{2,i}(t^{n+1})} \frac{\partial [\boldsymbol{\psi}_j]_{x_3}}{\partial x_2} \, \mathrm{d}\mathbf{x} - u_{x_2} \int_{\Omega_{2,i}(t^{n+1})} \frac{\partial [\boldsymbol{\psi}_j]_{x_2}}{\partial x_3} \, \mathrm{d}\mathbf{x} \\ u_{x_1} \displaystyle\int_{\Omega_{2,i}(t^{n+1})} \frac{\partial [\boldsymbol{\psi}_j]_{x_1}}{\partial x_3} \, \mathrm{d}\mathbf{x} - u_{x_3} \int_{\Omega_{2,i}(t^{n+1})} \frac{\partial [\boldsymbol{\psi}_j]_{x_3}}{\partial x_1} \, \mathrm{d}\mathbf{x} \\ u_{x_2} \displaystyle\int_{\Omega_{2,i}(t^{n+1})} \frac{\partial [\boldsymbol{\psi}_j]_{x_2}}{\partial x_1} \, \mathrm{d}\mathbf{x} - u_{x_1} \int_{\Omega_{2,i}(t^{n+1})} \frac{\partial [\boldsymbol{\psi}_j]_{x_1}}{\partial x_2} \, \mathrm{d}\mathbf{x} \end{bmatrix} \end{aligned} \tag{5.28}$$

5.2.3.2 Element matrices

For the assembly of the matrices occurring in the weak formulations above, local element matrices are computed that are later assembled into a global matrix. A general computation scheme for any geometry is derived by affine mapping from a reference tetrahedron $\hat{T}$, as illustrated in figure 5.4. By computing the element contributions and mapping back to the actual element, we can easily assemble the global matrices.

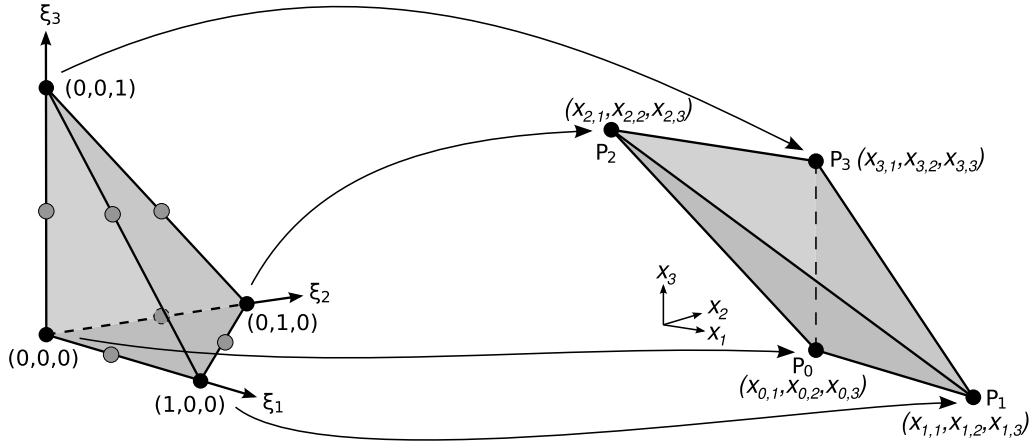

Figure 5.4: Mapping from the reference tetrahedron to an arbitrary $\mathcal{P}_1$ element.

Pressure stiffness matrix

We are going to denote the stiffness matrix of the pressure by B, defined as

$$\mathbf{B} := (b_{ij}) := \int_\Omega \nabla \chi_i \nabla \chi_j \, \mathrm{d}\mathbf{x}.$$

As mentioned before, we are going to assemble the whole matrix by adding up the contributions of all elements, which constitute a complete partition of Ω:

$$\int_\Omega \nabla\chi_i \nabla\chi_j \,\mathrm{d}\mathbf{x} = \sum_{T\in\mathcal{T}_h} \int_T \nabla\chi_i \nabla\chi_j \,\mathrm{d}\mathbf{x}. \tag{5.29}$$

In our case of $\mathcal{P}_1$-tetrahedra, we only have to consider the four *element basis functions*$\zeta_{T,i}$, that are non-zero on the respective element's nodes. These can be understood as the restrictions of the global basis functions χ_k on the element T and hence they form a basis set, such that within the element, the solution takes the form

$$p|_T = \sum_{i=1}^{4} p_{T,i}\zeta_{T,i}.$$

Localizing (5.29), we obtain a set of 4×4 matrices B_T, such that

$$\mathbf{B}_T = (b_{T,ij}) = \int_T \nabla\zeta_{T,i}\nabla\zeta_{T,j} \,\mathrm{d}\mathbf{x}.$$

These B_T are the *element stiffness matrices* associated to the $T\in\mathcal{T}_h$, we are looking for.

We can derive general equations for $\nabla\zeta_i$ by mapping from the reference tetrahedron, which is defined by

$$\hat{T} = \{(x_1, x_3, x_3) : 0 \le x_1, x_2, x_3 \le 1, x_1 + x_2 + x_3 \le 1\}. \tag{5.30}$$

We now consider the mapping from the reference tetrahedron to the actual element, as illustrated in figure 5.4. From this we get the local-global mapping defined for all points $P_i(x_1, x_2, x_3) = (x_{i,1}, x_{i,2}, x_{i,3}) \in T$ by

$$\begin{aligned} x_1(\boldsymbol{\xi}) &= x_{0,1}\tau_0(\boldsymbol{\xi}) + x_{1,1}\tau_1(\boldsymbol{\xi}) + x_{2,1}\tau_2(\boldsymbol{\xi}) + x_{3,1}\tau_3(\boldsymbol{\xi}),\\ x_2(\boldsymbol{\xi}) &= x_{0,2}\tau_0(\boldsymbol{\xi}) + x_{1,2}\tau_1(\boldsymbol{\xi}) + x_{2,2}\tau_2(\boldsymbol{\xi}) + x_{3,2}\tau_3(\boldsymbol{\xi}),\\ x_3(\boldsymbol{\xi}) &= x_{0,3}\tau_0(\boldsymbol{\xi}) + x_{1,3}\tau_1(\boldsymbol{\xi}) + x_{2,3}\tau_2(\boldsymbol{\xi}) + x_{3,3}\tau_3(\boldsymbol{\xi}), \end{aligned} \tag{5.31}$$

where

$$\begin{aligned} \tau_0(\xi_1,\xi_2,\xi_3) &= 1-\xi_1-\xi_2-\xi_3,\\ \tau_1(\xi_1,\xi_2,\xi_3) &= \xi_1,\\ \tau_2(\xi_1,\xi_2,\xi_3) &= \xi_2,\\ \tau_3(\xi_1,\xi_2,\xi_3) &= \xi_3. \end{aligned} \tag{5.32}$$

are the shape functions on the reference element, that are easily computed from the rule that all basis functions χ_i are linear, equal to 1 on node i and zero on all other nodes.

Clearly the mapping is differentiable, and we obtain for any differentiable function $\varphi(\xi_1, \xi_2, \xi_3)$ a transformation of derivatives by

$$\begin{bmatrix} \frac{\partial\varphi}{\partial\xi_1} \\ \frac{\partial\varphi}{\partial\xi_2} \\ \frac{\partial\varphi}{\partial\xi_3} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{\partial x_1}{\partial\xi_1} & \frac{\partial x_2}{\partial\xi_1} & \frac{\partial x_3}{\partial\xi_1} \\ \frac{\partial x_1}{\partial\xi_2} & \frac{\partial x_2}{\partial\xi_2} & \frac{\partial x_3}{\partial\xi_2} \\ \frac{\partial x_1}{\partial\xi_3} & \frac{\partial x_2}{\partial\xi_3} & \frac{\partial x_3}{\partial\xi_3} \end{bmatrix}}_{=:\mathbf{J}^T} \begin{bmatrix} \frac{\partial\varphi}{\partial x_1} \\ \frac{\partial\varphi}{\partial x_2} \\ \frac{\partial\varphi}{\partial x_3} \end{bmatrix}. \tag{5.33}$$

The *Jacobian matrix* $\mathbf{J}$ occurring here can be calculated by substituting (5.32) into (5.31) and differentiating:

$$\mathbf{J} = \frac{\partial(x_1, x_2, x_3)}{\partial(\xi_1, \xi_2, \xi_3)} = \begin{bmatrix} x_{1,1} - x_{0,1} & x_{2,1} - x_{0,1} & x_{3,1} - x_{0,1} \\ x_{1,2} - x_{0,2} & x_{2,2} - x_{0,2} & x_{3,2} - x_{0,2} \\ x_{1,3} - x_{0,3} & x_{2,3} - x_{0,3} & x_{3,3} - x_{0,3} \end{bmatrix}. \tag{5.34}$$

In this simple case, $\mathbf{J}$ is constant and we can explicitly compute the determinant by

$$\begin{aligned} |\det \mathbf{J}| = |\det \mathbf{J}^T| &= \left| \det \begin{pmatrix} x_{1,1} - x_{0,1} & x_{1,2} - x_{0,2} & x_{1,3} - x_{0,3} \\ x_{2,1} - x_{0,1} & x_{2,2} - x_{0,2} & x_{2,3} - x_{0,3} \\ x_{3,1} - x_{0,1} & x_{3,2} - x_{0,2} & x_{3,3} - x_{0,3} \end{pmatrix} \right| \\ &= \left| \det \begin{pmatrix} 1 & x_{0,1} & x_{0,2} & x_{0,3} \\ 1 & x_{1,1} & x_{1,2} & x_{1,3} \\ 1 & x_{2,1} & x_{2,2} & x_{2,3} \\ 1 & x_{3,1} & x_{3,2} & x_{3,3} \end{pmatrix} \right| = 6V_T, \end{aligned}$$

where V_T is the volume of the master tetrahedron T, in our case $\dfrac{h^3}{6}$.

In order to get the inverse global-local mapping, we can simply invert $\mathbf{J}^T$, as the determinant is non-zero for any point (ξ_1, ξ_2, ξ_3):

$$\begin{aligned} \mathbf{J}^{-T} &= \begin{bmatrix} x_{1,1} - x_{0,1} & x_{1,2} - x_{0,2} & x_{1,3} - x_{0,3} \\ x_{2,1} - x_{0,1} & x_{2,2} - x_{0,2} & x_{2,3} - x_{0,3} \\ x_{3,1} - x_{0,1} & x_{3,2} - x_{0,2} & x_{3,3} - x_{0,3} \end{bmatrix}^{-1} =: \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{-1} \\ &= \frac{1}{\det \mathbf{J}} \begin{bmatrix} -fh + ei & +ch - bi & -ce + bf \\ +fg - di & -cg + ai & -af + cd \\ -eg + dh & +bg - ah & -bd + ae \end{bmatrix} =: \frac{1}{\det \mathbf{J}} \begin{bmatrix} b_{11}^* & b_{12}^* & b_{13}^* \\ b_{21}^* & b_{22}^* & b_{23}^* \\ b_{31}^* & b_{32}^* & b_{33}^* \end{bmatrix}. \end{aligned}$$

Writing this inverse mapping similar to (5.33), we have

$$\begin{bmatrix} \frac{\partial\varphi}{\partial x_1} \\ \frac{\partial\varphi}{\partial x_2} \\ \frac{\partial\varphi}{\partial x_3} \end{bmatrix} = \frac{1}{\det \mathbf{J}} \underbrace{\begin{bmatrix} b_{11}^* & b_{12}^* & b_{13}^* \\ b_{21}^* & b_{22}^* & b_{23}^* \\ b_{31}^* & b_{32}^* & b_{33}^* \end{bmatrix}}_{=:\underbrace{\begin{bmatrix} b_1^* & b_2^* & b_3^* \end{bmatrix}}_{=:\mathbf{B}^*}} \begin{bmatrix} \frac{\partial\varphi}{\partial\xi_1} \\ \frac{\partial\varphi}{\partial\xi_2} \\ \frac{\partial\varphi}{\partial\xi_3} \end{bmatrix}. \tag{5.35}$$

We can proceed by giving an expression for $b_{T,ij}$:

$$\begin{aligned} b_{T,ij} &= \int_T \frac{\partial\zeta_{T,i}}{\partial x_1}\frac{\partial\zeta_{T,j}}{\partial x_1} + \frac{\partial\zeta_{T,i}}{\partial x_2}\frac{\partial\zeta_{T,j}}{\partial x_2} + \frac{\partial\zeta_{T,i}}{\partial x_3}\frac{\partial\zeta_{T,3}}{\partial x_1}\,\mathrm{d}\mathbf{x} \\ &= \int_{\hat{T}} \left\{ \frac{\partial\tau_i}{\partial x_1}\frac{\partial\tau_j}{\partial x_1} + \frac{\partial\tau_i}{\partial x_2}\frac{\partial\tau_j}{\partial x_2} + \frac{\partial\tau_i}{\partial x_3}\frac{\partial\tau_j}{\partial x_1} \right\} |\det \mathbf{J}|\,\mathrm{d}\xi \\ &= \frac{|\det \mathbf{J}|}{(\det \mathbf{J})^2} \int_{\hat{T}} \left(b_{11}^*\frac{\partial\tau_i}{\partial\xi_1} + b_{12}^*\frac{\partial\tau_i}{\partial\xi_2} + b_{13}^*\frac{\partial\tau_i}{\partial\xi_3} \right) \left(b_{11}^*\frac{\partial\tau_j}{\partial\xi_1} + b_{12}^*\frac{\partial\tau_j}{\partial\xi_2} + b_{13}^*\frac{\partial\tau_j}{\partial\xi_3} \right) \\ &\qquad + \left(b_{21}^*\frac{\partial\tau_i}{\partial\xi_1} + b_{22}^*\frac{\partial\tau_i}{\partial\xi_2} + b_{23}^*\frac{\partial\tau_i}{\partial\xi_3} \right) \left(b_{21}^*\frac{\partial\tau_j}{\partial\xi_1} + b_{22}^*\frac{\partial\tau_j}{\partial\xi_2} + b_{23}^*\frac{\partial\tau_j}{\partial\xi_3} \right) \\ &\qquad + \left(b_{31}^*\frac{\partial\tau_i}{\partial\xi_1} + b_{32}^*\frac{\partial\tau_i}{\partial\xi_2} + b_{33}^*\frac{\partial\tau_i}{\partial\xi_3} \right) \left(b_{31}^*\frac{\partial\tau_j}{\partial\xi_1} + b_{32}^*\frac{\partial\tau_j}{\partial\xi_2} + b_{33}^*\frac{\partial\tau_j}{\partial\xi_3} \right) \mathrm{d}\xi. \end{aligned}$$

Because of the simple nature of the τ_i we can explicitly compute the derivatives as

$$\frac{\partial\tau_0}{\partial\boldsymbol{\xi}} = \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}, \qquad \frac{\partial\tau_1}{\partial\boldsymbol{\xi}} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad \frac{\partial\tau_2}{\partial\boldsymbol{\xi}} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \qquad \frac{\partial\tau_3}{\partial\boldsymbol{\xi}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

As also the derivatives of the shape functions on the master element $\zeta_{T,i}$ are constant, we have in (5.35), that also $\mathbf{B}^*$ is constant and integration simplifies to multiplying with the volume of the reference tetrahedron. The volume is calculated by $\int_0^1 \int_0^{1-\xi_1} \int_0^{1-\xi_1-\xi_2} \mathrm{d}\xi_3\,\mathrm{d}\xi_2\,\mathrm{d}\xi_1 = \frac{1}{6}$ and inserting gives the following result:

$$\begin{aligned} b_{T,ij} = \frac{1}{6h^3} \Bigg[&\left(b_{11}^*\frac{\partial\tau_i}{\partial\xi_1} + b_{12}^*\frac{\partial\tau_i}{\partial\xi_2} + b_{13}^*\frac{\partial\tau_i}{\partial\xi_3} \right) \left(b_{11}^*\frac{\partial\tau_j}{\partial\xi_1} + b_{12}^*\frac{\partial\tau_j}{\partial\xi_2} + b_{13}^*\frac{\partial\tau_j}{\partial\xi_3} \right) \\ &+ \left(b_{21}^*\frac{\partial\tau_i}{\partial\xi_1} + b_{22}^*\frac{\partial\tau_i}{\partial\xi_2} + b_{23}^*\frac{\partial\tau_i}{\partial\xi_3} \right) \left(b_{21}^*\frac{\partial\tau_j}{\partial\xi_1} + b_{22}^*\frac{\partial\tau_j}{\partial\xi_2} + b_{23}^*\frac{\partial\tau_j}{\partial\xi_3} \right) \\ &+ \left(b_{31}^*\frac{\partial\tau_i}{\partial\xi_1} + b_{32}^*\frac{\partial\tau_i}{\partial\xi_2} + b_{33}^*\frac{\partial\tau_i}{\partial\xi_3} \right) \left(b_{31}^*\frac{\partial\tau_j}{\partial\xi_1} + b_{32}^*\frac{\partial\tau_j}{\partial\xi_2} + b_{33}^*\frac{\partial\tau_j}{\partial\xi_3} \right) \Bigg]. \end{aligned}$$

The actual computation for the six types of tetrahedra used (cf. figure 5.3) is quickly done with a computer algebra system, the resulting six element stiffness matrices are presented in appendix A.1. As elements of type IV ,V and VI are only flipped versions of elements of types I, II and III, B^* is the same for a pair of elements, only the sign of the determinant is different. As we transform both gradients, we obtain the squared determinant as factor and the element matrices of I and IV, II and V, and III and VI are identical.

Velocity stiffness matrix

The stiffness matrix of the velocity is going to be denoted by $\mathbf{A}$, with

$$\mathbf{A} := (a_{ij}) = \int_\Omega \nabla \boldsymbol{\psi}_i : \nabla \boldsymbol{\psi}_j \, \mathrm{d}\mathbf{x}.$$

This can be simplified by noting that we used the same scalar spaces of finite elements for each of the three components, we can instead calculate three identical $N \times N$ matrices $\mathbf{A}'$ by using $\{\boldsymbol{\psi}_i\}_{i=1}^{3N} = \{(\psi_1, 0, 0)^T, \ldots, (\psi_N, 0, 0)^T, (0, \psi_1, 0)^T, \ldots, (0, \psi_N, 0)^T, (0, 0, \psi_1)^T, \ldots, (0, 0, \psi_N)^T\}$ and computing

$$\mathbf{A}(\mathbf{u}_i)_{i=1,\ldots,3N} = \begin{bmatrix} \mathbf{A}' & 0 & 0 \\ 0 & \mathbf{A}' & 0 \\ 0 & 0 & \mathbf{A}' \end{bmatrix} \begin{bmatrix} (\mathbf{u}_i)_{i=1,\ldots,N} \\ (\mathbf{u}_i)_{i=N+1,\ldots,2N} \\ (\mathbf{u}_i)_{i=2N+1,\ldots,3N} \end{bmatrix},$$

with

$$\mathbf{A}' = \int_\Omega \nabla \psi_i \cdot \nabla \psi_j \, \mathrm{d}\mathbf{x}.$$

Again, we restrict the basis functions ψ_k on an element T and call these 10 local basis functions $\varsigma_{T,i} = \psi_k|_T$:

$$(\mathbf{u}_h)_k|_T = \sum_{i=1}^{10} u_i \cdot \varsigma_{T,i}.$$

Again we assemble the global stiffness matrix by computing the element stiffness matrices $\mathbf{A}'_T$ associated to T by

$$\mathbf{A}'_T = (a'_{T,ij}) = \int_T \nabla \varsigma_{T,i} \nabla \varsigma_{T,j} \, \mathrm{d}\mathbf{x}.$$

As we are still dealing with tetrahedra, the mappings we derived for the pressure in (5.31) and (5.33) still hold. We only have to derive new shape functions η_i on the $\mathcal{P}_2$ reference element in order to get expressions for the $\varsigma_{T,i}$. This can be done manually for the pressure or by using the fact, that the quadratic functions we are looking for are actually Lagrange interpolants and can be expressed with the help of the linear functions.

Either way, we derive the following 10 shape functions on the reference tetrahedron, where the numbering indicates, on which node a shape function equals one:

$$\begin{aligned}
&\eta_1 = 2(\xi_1^2+\xi_2^2+\xi_3^2) - 3(\xi_1+\xi_2+\xi_3) + 4(\xi_1\xi_2+\xi_1\xi_3+\xi_2\xi_3) + 1,\\
&\eta_3 = 2\xi_1^2 - \xi_1, \qquad \eta_2 = 4(-\xi_1^2+\xi_1-\xi_1\xi_2-\xi_1\xi_3), \qquad \eta_5 = 4\xi_1\xi_2,\\
&\eta_6 = 2\xi_2^2 - \xi_2, \qquad \eta_4 = 4(-\xi_2^2+\xi_2-\xi_2\xi_1-\xi_2\xi_3), \qquad \eta_8 = 4\xi_1\xi_2,\\
&\eta_{10} = 2\xi_3^2 - \xi_3, \qquad \eta_7 = 4(-\xi_3^2+\xi_3-\xi_3\xi_1-\xi_3\xi_2), \qquad \eta_9 = 4\xi_1\xi_2.
\end{aligned}$$

Here, η_0 is the shape function, that is one in the origin, η_1, η_2, η_3 are equal to one on the corner nodes, η_4, η_5, η_6 are mid-edge shape functions on the edges parallel to one coordinate axes and finally, η_7, η_8, η_9 are the remaining other three mid-edge shape functions.

Similarly to the linear case, we can compute the entries of the element stiffness matrix by

$$\begin{aligned}
a'_{T,ij} =& \frac{1}{|\det \mathbf{J}|} \int_{\hat{T}} \left(b_{11}^* \frac{\partial \eta_i}{\partial \xi_1} + b_{12}^* \frac{\partial \eta_i}{\partial \xi_2} + b_{13}^* \frac{\partial \eta_i}{\partial \xi_3} \right) \left(b_{11}^* \frac{\partial \eta_j}{\partial \xi_1} + b_{12}^* \frac{\partial \eta_j}{\partial \xi_2} + b_{13}^* \frac{\partial \eta_j}{\partial \xi_3} \right)\\
&+ \left(b_{21}^* \frac{\partial \eta_i}{\partial \xi_1} + b_{22}^* \frac{\partial \eta_i}{\partial \xi_2} + b_{23}^* \frac{\partial \eta_i}{\partial \xi_3} \right) \left(b_{21}^* \frac{\partial \eta_j}{\partial \xi_1} + b_{22}^* \frac{\partial \eta_j}{\partial \xi_2} + b_{23}^* \frac{\partial \eta_j}{\partial \xi_3} \right)\\
&+ \left(b_{31}^* \frac{\partial \eta_i}{\partial \xi_1} + b_{32}^* \frac{\partial \eta_i}{\partial \xi_2} + b_{33}^* \frac{\partial \eta_i}{\partial \xi_3} \right) \left(b_{31}^* \frac{\partial \eta_j}{\partial \xi_1} + b_{32}^* \frac{\partial \eta_j}{\partial \xi_2} + b_{33}^* \frac{\partial \eta_j}{\partial \xi_3} \right) \mathrm{d}\xi.
\end{aligned}$$

Now we do not have constant derivatives anymore, such that we have to compute the integrals on the reference tetrahedron. From (5.30), we have that the integral for an integrable function φ on $\hat{T}$ is computed by

$$\int_{\hat{T}} \varphi \,\mathrm{d}\xi = \int_0^1 \int_0^{1-\xi_1} \int_0^{1-\xi_1-\xi_2} \varphi \,\mathrm{d}\xi_3 \,\mathrm{d}\xi_2 \,\mathrm{d}\xi_1.$$

Using a computer algebra system again to evaluate the above expressions for the matrix entries, we get six 10×10 velocity elment stiffness matrices, that are printed as well in appendix A.1. Again we obtain pairs of elements with identical element matrices.

Velocity mass matrix

For the velocity mass matrix,

$$\mathbf{M}_{ij} = \int_\Omega \boldsymbol{\psi}_i : \boldsymbol{\psi}_j \,\mathrm{d}\mathbf{x},$$

we perform again a component splitting and compute

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}' & 0 & 0 \\ 0 & \mathbf{M}' & 0 \\ 0 & 0 & \mathbf{M}' \end{bmatrix}, \text{ with } \mathbf{M}' = \int_\Omega \psi_i \cdot \psi_j \,\mathrm{d}\mathbf{x}.$$

Also as before, we perform the computation via element matrices, that we derive by mapping to a reference tetrahedron. Using (5.34), we can re-write (5.32) to define the local-global mapping F by

$$F(\boldsymbol{\xi}) = \mathbf{J}^T\boldsymbol{\xi} + \mathbf{x}_0,$$

where the inverse mapping is clearly given by

$$F^{-1}(\mathbf{x}) = \mathbf{J}^{-T}(\mathbf{x} - \mathbf{x}_0).$$

This can now applied to compute the element mass matrix entries on the reference tetrahedron:

$$\begin{aligned}\int_T \psi_i \cdot \psi_j \,\mathrm{d}\mathbf{x} &= \int_T \left(F^{-1}(\psi_i)\right) \cdot \left(F^{-1}(\psi_j)\right) \,\mathrm{d}\mathbf{x} \\ &= \int_T \eta_i \cdot \eta_j \,\mathrm{d}\mathbf{x} \\ &= |\det \mathbf{J}| \int_{\hat{T}} \eta_i \cdot \eta_j \,\mathrm{d}\xi \\ &= h^3 \int_{\hat{T}} \eta_i \cdot \eta_j \,\mathrm{d}\xi\end{aligned}$$

The computed result is again printed in appendix A.1.

Velocity pressure-gradient matrix

$$\mathbf{C}_{ij} = \int_\Omega \boldsymbol{\psi}_i \nabla\chi_j \,\mathrm{d}\mathbf{x}, \quad i = 1, \ldots, 3N,\ j = 1, \ldots, M,$$

As we have already dealt with the two factors occurring here, i.e. the basis functions of the velocity and the gradient of the pressure basis functions, the derivation of element matrices for this matrix is easy.

We consider again an element $T \in \mathcal{T}_h$ and the restrictions of $\boldsymbol{\psi}$ and χ on it. We also decompose $\boldsymbol{\psi}$ again and compute

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}'_1 \\ \mathbf{C}'_2 \\ \mathbf{C}'_3 \end{bmatrix}, \text{ with } (\mathbf{C}'_d)_{ij} = \int_\Omega \psi_i \frac{\partial\chi_j}{\partial x_d} \,\mathrm{d}\mathbf{x} = \frac{\partial\chi_j}{\partial x_d} \int_\Omega \psi_i \,\mathrm{d}\mathbf{x}, \qquad \begin{matrix} i = 1, \ldots, N, \\ j = 1, \ldots, M. \end{matrix}$$

We then have

$$\mathbf{C}'_{d,T} = \frac{\partial\zeta_{T,j}}{\partial x_d} \int_T \varsigma_{T,i} \,\mathrm{d}\mathbf{x} = \frac{|\det \mathbf{J}|}{\det \mathbf{J}} \left(b^*_{d1} \frac{\partial\tau_j}{\partial\xi_1} + b^*_{d2} \frac{\partial\tau_j}{\partial\xi_2} + b^*_{d3} \frac{\partial\tau_j}{\partial\xi_3} \right) \int_{\hat{T}} \eta_i \,\mathrm{d}\xi.$$

As we now have the determinant only once as factor, we obtain unique element matrices for each of the occurring tetrahedra. The resulting $6 \cdot d = 18$ element matrices can be found in appendix A.1.

Velocity-divergence pressure matrix

The last matrix to be computed is the velocity-divergence pressure matrix D defined by

$$\begin{aligned}\mathbf{D}_{ij} &= \int_\Omega \chi_i \left(\nabla \cdot \boldsymbol{\psi}_j\right) \,\mathrm{d}\mathbf{x} \\ &= \int_\Omega \chi_i \left(\frac{\partial(\boldsymbol{\psi}_j)_1}{\partial x_1} + \frac{\partial(\boldsymbol{\psi}_j)_2}{\partial x_2} + \frac{\partial(\boldsymbol{\psi}_j)_3}{\partial x_3}\right) \mathrm{d}\mathbf{x}, \quad i = 1,\ldots,M,\ j = 1,\ldots,3N,\end{aligned}$$

that we divide by components as

$$\mathbf{D} = \left[\begin{array}{ccc} \mathbf{D}'_1 & \mathbf{D}'_2 & \mathbf{D}'_3 \end{array}\right], \text{ with } \mathbf{D}'_d = \int_\Omega \chi_i \frac{\partial \psi_j}{\partial x_d} \,\mathrm{d}\mathbf{x} \quad i = 1,\ldots,M,\ j = 1,\ldots,N.$$

Here we used, that we consider only basis functions $\boldsymbol{\psi}_j$ that are non-zero only in one component at a time.
The respective element matrix is found as usual via mapping from the reference tetrahedron:

$$\begin{aligned}\mathbf{D}'_{d,T} &= \int_T \zeta_{T,i} \frac{\partial \varsigma_{T,j}}{\partial x_d} \,\mathrm{d}\mathbf{x} \\ &= \frac{|\det \mathbf{J}|}{\det \mathbf{J}} \int_{\hat{T}} \tau_i \left(b^*_{d1} \frac{\partial \eta_j}{\partial \xi_1} + b^*_{d2} \frac{\partial \eta_j}{\partial \xi_2} + b^*_{d3} \frac{\partial \eta_j}{\partial \xi_3}\right) \mathrm{d}\xi.\end{aligned}$$

This results again in 18 element matrices that are as well printed in appendix A.1.

Advection component matrices

Finally, we deal with the vectors in the right-hand sides of (5.22) and (5.23):

$$\sum_{i=1}^{N} u^n_{x_d,i} \left(\sum_{e=1}^{3}\sum_{k=1}^{N} u^n_{x_e,k} \int_\Omega \psi_k \frac{\partial \psi_i}{\partial x_e} \psi_j \,\mathrm{d}\mathbf{x}\right), \quad j = 1,\ldots,N, \qquad \text{and,}$$

$$\sum_{i=1}^{N} u^{n+q/(5Q)}_{x_d,i} \left(\sum_{e,f=1}^{3}\sum_{k,l=1}^{N} u^n_{x_e,k} u^n_{x_f,l} \int_\Omega \psi_k \psi_l \frac{\partial \psi_i}{\partial x_e} \frac{\partial \psi_j}{\partial x_f} \,\mathrm{d}\mathbf{x}\right), \quad j = 1,\ldots,N,$$

where we have $q = 0$ in the second equation for (5.22).

Here it is not reasonable to pre-compile matrix-like structures for the element types and multiply with all the coefficient vectors directly before solving the equations systems. In this case we would be left with three- or four-dimensional objects that will occupy enormous memory resources when assembled later, albeit being sparse and identical for all d. As we only deal with known coefficients in the right-hand sides, we can already solve the sums and keep only vectors over j. For the first equation that is only needed once, this is possible reasonable solution. For the second, we would have to recompile the element vectors for each sub-time-step. In this case, it is more appropriate to keep the sum over i and hence a matrix only

depending on the values of $u^n_{x_d}$ from the previous major time step. Also for the other matrix, we do not gain performance by pre-computing a vector instead of a matrix, we can as well do a matrix-vector multiplication during run-time.

On this basis, we define the two component matrices $\mathbf{R}' = \sum_{T\in\mathcal{T}_h} \mathbf{R}'_T$ and $\mathbf{S}' = \sum_{T\in\mathcal{T}_h} \mathbf{S}'_T$, with the element matrices defined as

$$\mathbf{R}'_T = \sum_{e=1}^{3}\sum_{k=1}^{N} u^n_{x_e,k} \frac{|\det \mathbf{J}|}{\det \mathbf{J}} \int_{\hat{T}} \eta_k \left(b^*_{e1}\frac{\partial \eta_i}{\partial \xi_1} + b^*_{e2}\frac{\partial \eta_i}{\partial \xi_2} + b^*_{e3}\frac{\partial \eta_i}{\partial \xi_3}\right) \eta_j \,\mathrm{d}\xi, \quad \text{and}$$

$$\mathbf{S}'_T = \sum_{e,f=1}^{3}\sum_{k,l=1}^{N} \left[u^n_{x_e,k} u^n_{x_f,l} \frac{1}{|\det \mathbf{J}|} \cdot \int_{\hat{T}} \eta_k \eta_l \left(b^*_{e1}\frac{\partial \eta_i}{\partial \xi_1} + b^*_{e2}\frac{\partial \eta_i}{\partial \xi_2} + b^*_{e3}\frac{\partial \eta_i}{\partial \xi_3}\right) \left(b^*_{f1}\frac{\partial \eta_j}{\partial \xi_1} + b^*_{f2}\frac{\partial \eta_j}{\partial \xi_2} + b^*_{f3}\frac{\partial \eta_j}{\partial \xi_3}\right) \mathrm{d}\xi \right].$$

Vectors

As well as the matrices, the load vector $\mathbf{L}$ is computed by adding up element contributions:

$$\mathbf{L}'_i = \int_\Omega \psi_i \,\mathrm{d}\mathbf{x} = \sum_{T\in\mathcal{T}_h} \int_T \psi_i \,\mathrm{d}\mathbf{x} = \sum_{T\in\mathcal{T}_h} |\det \mathbf{J}| \int_{\hat{T}} \eta_i \,\mathrm{d}\xi, \quad i = 1,\ldots,3N.$$

The same applies to the component gradient vectors needed $\mathbf{G}_d$, that is needed in (5.16):

$$\begin{aligned}(\mathbf{G}_d)_i &= \int_\Omega \frac{\partial \psi_i}{\partial x_d} \,\mathrm{d}\mathbf{x} = \sum_{T\in\mathcal{T}_h} \int_T \frac{\partial \psi_i}{\partial x_d} \,\mathrm{d}\mathbf{x} \\ &= \sum_{T\in\mathcal{T}_h} \frac{|\det \mathbf{J}|}{\det \mathbf{J}} \int_{\hat{T}} b^*_{d1}\frac{\partial \eta_i}{\partial \xi_1} + b^*_{d2}\frac{\partial \eta_i}{\partial \xi_2} + b^*_{d3}\frac{\partial \eta_i}{\partial \xi_3} \,\mathrm{d}\xi, \quad i = 1,\ldots,N.\end{aligned}$$

5.2.4 Assembly of the Galerkin systems

For the generation of global matrices from element matrices we need some information of the triangulation. A matrix entry e_{ij} is only non-zero if there exists at least one element that includes point P_i and P_j. (As every point is at a corner or on an edge of an element, if there exists one such element, then there exists at least another).

The summing up of all contributions is done step-wise for each matrix entry of the final matrix $\mathbf{E}$:

1. Identify the octants, i.e. the cubes enclosing six tetrahedra each, that P_i and P_j are part of.
2. Identify the k tetrahedra and their type $t_k = I,\ldots,VI$ that includes P_i and P_j and the local point numbers i' and j' of the points.

3. Sum up $\mathbf{E}'_{t_k}(i', j')$.

We start out with the first task and use the naming conventions as shown in figure 5.5 for the pressure nodes.

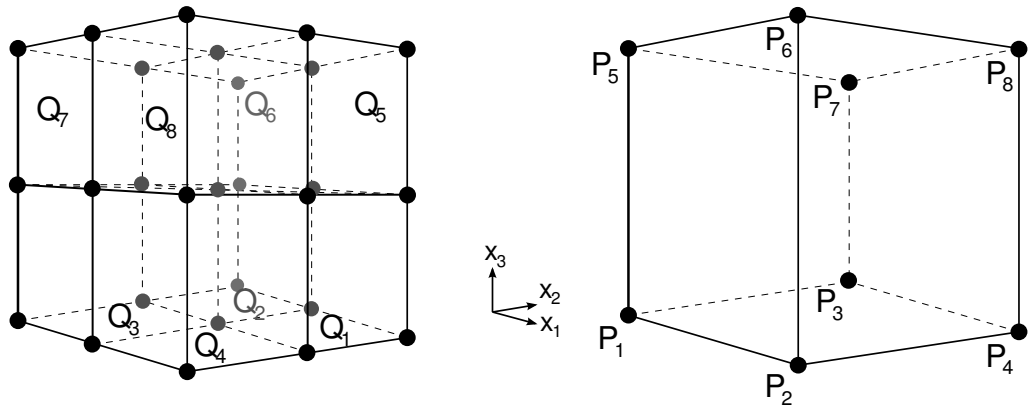

Figure 5.5: Local node and octants naming conventions.

Each point is part of eight quadrants and is assigned a local number therein, that is needed for picking the respective contribution in the elment matrices. Likewise, a pair of neighboring points is included in four quadrants, those that share the edge between these two points.

Secondly, for each quadrant with its nodes numbered locally from 1 to 8, we have for the six elements contained in it a mapping of node numbers with:

P	I	II	III	IV	V	VI
1	1			1		
2	2	1				
3				2	1	
4	3	2	1	3	2	1
5	4	3	2	4	3	2
6		4	3			
7					4	3
8			4			4

In order to proceed with the third step, we only have to check for each element type t_k, if there exists a mapping for both $P_i \to P_{i'}$ and $P_j \to P_{j'}$.

Now $\mathbf{E}'_{t_k}(i', j')$ can be added to e_{ij}.

As an example, figure 5.6 shows the resulting stiffness matrices for a triangulation of the unit cube with $h = 0.25$, i.e. the unit cube is divided in $4 \times 4 \times 4$ cubes, resulting in 384 elements consisting of 125 pressure nodes and 729 velocity nodes. By construction we get sparse symmetric matrices with block structures. The structure

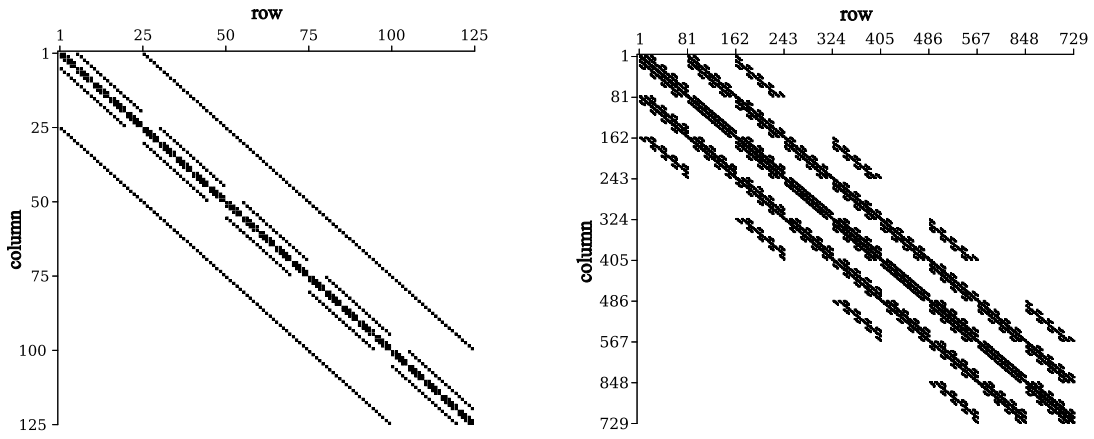

Figure 5.6: Structure plots of the pressure stiffness matrix $\mathbf{B}$ (left) and the velocity component stiffness matrix $\mathbf{A}'$ (right) for a triangulation of the unit cube with $h = 0.25$.

plots of figure 5.6 show a black dot for every matrix entry that is non-zero. All other structure plots can be found in appendix A.2.

The matrices/vectors occurring in the advection sub-problem depend on results of the previous time step, such that we cannot pre-compile them. Compilation at run-time is done the same way, via summing up of element contributions. Here we have two main possibilities, to iterate node- or element-wise. Element-wise calculation would mean, to evaluate the element matrix and distribute the results over the respective nodes. On the GPU this would make the handling of possible write conflicts necessary, when different processors evaluated neighboring elements and try to set the a value to the same node. Node-wise calculation avoids this problem but evaluates elements several times, as for each nodes all the octants shown in figure 5.5 have to be considered.

With all matrices compiled as described, we can rewrite the problems as

Advection: Find $\left(\mathbf{u}_i^{n+\frac{q+1}{5Q}}\right)_{i=1,\ldots,3N}$, for $q = 0, \ldots, Q-1$, such that

$$\begin{cases} \mathbf{M}_{bc} \cdot \left(\mathbf{u}_i^{n+\frac{1}{5Q}}\right)_{bc} = \mathbf{M}_{bc} \cdot (\mathbf{u}_i^n)_{bc} \\ \qquad\qquad + \left(\delta t' \mathbf{R} + \dfrac{(\delta t')^2}{2}\mathbf{S}\right)_{bc} \cdot (\mathbf{u}_i^n)_{bc} \qquad \text{for } q = 0 \qquad (5.36a) \\ \mathbf{M}_{bc} \cdot \left(\mathbf{u}_i^{n+\frac{q+1}{5Q}}\right)_{bc} = \mathbf{M}_{bc} \cdot \left(2\mathbf{u}_i^{n+\frac{q}{5Q}} - \mathbf{u}_i^{n+\frac{q-1}{5Q}}\right)_{bc} \\ \qquad\qquad + (\delta t')^2 \, \mathbf{S}_{bc} \cdot \left(\mathbf{u}_i^{n+\frac{q}{5Q}}\right)_{bc} \qquad \text{for } q > 0 \qquad (5.36b) \end{cases}$$

Diffusion: Find $\left(\mathbf{u}_i^{n+2/5}\right)_{i=1,\ldots,3N}$, such that

$$\left(\frac{\rho_f}{\delta t}\mathbf{M} + \mu\mathbf{A}\right)_{bc} \cdot \left(\mathbf{u}_i^{n+2/5}\right)_{bc} = \left(\frac{\rho_f}{\delta t}\mathbf{L}\right)_{bc} : \left(\mathbf{u}_i^{n+1/5}\right)_{bc} \qquad (5.37)$$

Projection: Find $\left(p_i^{n+1}\right)_{i=1,\dots,M}$ and $\left(\mathbf{u}_i^{n+3/5}\right)_{i=1,\dots,3N}$, such that

$$\begin{cases} \mathbf{B} \cdot \left(p_i^{n+1}\right) = \dfrac{\rho_f}{\delta t} \mathbf{C}^T \left(\mathbf{u}_i^{n+2/5}\right) & \text{and} \quad (5.38a) \\ \left(\dfrac{\rho_f}{\delta t}\mathbf{M}\right)_{bc} \cdot \left(\mathbf{u}_i^{n+3/5}\right)_{bc} = \left(\dfrac{\rho_f}{\delta t}\mathbf{M}\left(\mathbf{u}_i^{n+2/5}\right) + \mathbf{D}^T\left(p_i^{n+1}\right)\right)_{bc} & (5.38b) \end{cases}$$

The subscript *bc* denotes, that we still have to enforce the homogeneous Dirichlet boundary conditions for the fluid velocity. Fixing the velocity values on the boundary reduces the degrees of freedom by the number of these nodes. Instead of changing the matrix and vector entires, we can completely remove the ith row and/or ith column if a node P_i resides on the boundary and fill the solution with zeros afterwards. This way, the dimension of the equation system is reduced leading to faster computation.

Fulfilling the condition of the pressure integral to be 0 on Ω, can be achieved by at least two strategies. One is, to enlarge $\mathbf{B}$ with an additional row of 1's and the right side with a zero entry. This would be contradicting the idea, to keep the systems as sparse as possible. Thus the second approach is more favorable, which takes advantage of the fact, that the solution of the equation system will be computed in an iterative fashion. It is hence possible to correct the intermediate solutions for p after each step (or possibly several steps), to have integral zero, without disturbing the convergence of the overall scheme.

5.3 Summary

In this chapter, we motivated the choice of a operator splitting technique to divide the particulate flow problem into sub-problems that are individually easier to handle. The Navier-Stokes equations for the fluid flow problem were approached with wave-like/projection method and mixed $\mathcal{P}_2/\mathcal{P}_1$ finite elements in space and finite differences in time. The scheme is first-order accurate in time, as well as for the pressure in space. For the velocity, it is second-order accurate in space.

For each time step we execute the following algorithm:

1a Advection: Assemble the matrices $\mathbf{R}$ and $\mathbf{S}$ and solve the linear equations systems (5.36). As the individual velocity components in space are independent, we can instead solve three smaller problems (in parallel), using the component matrices $\mathbf{M}'$, $\mathbf{R}'$ and $\mathbf{S}'$

1b Prediction of particle velocities: By using the advected fluid velocity, we compute a first approximation of the particles' velocities by (5.7). The integrals occurring therein are evaluated similar to (5.27).

1c Particle position update: With the velocities predicted, also an estimation of the particles' positions can be given via (5.8).

2 Diffusion: As stated just above, we solve the linear equation system (5.37) in the diffusion sub-problem. As the velocity components are independent again, we use $\mathbf{M}'$, $\mathbf{A}'$ and vector $\mathbf{L}'$.

3 Projection: In this step, the two equation systems (5.38) have to be solved. As before, we can do this component-wise, such that the computation can be done in parallel. Before solving the second equations system, the pressure solution has to be corrected, such that the pressure integral over Ω equals zero.

4a Rigid body motion: Here, we have to identify the elements, occupied by the particles and compute the integrals (5.27) and (5.28) on these domains. Thereafter, we correct the velocities of both, the fluid and the particle via (5.15) and (5.16). For this first step, we leave out, the particle-wall collision terms.

4b Particle position update: Based on the newly computed particle velocities, we correct the particle positions, similar to (5.8).

5 Collisions: With the particle positions known, potential collisions are handled, i.e. forces occurring at particle-wall collisions are computed according to (2.21), as well as displacements resulting from particle-particle collisions, cf. (2.20). In the event of collisions, the updated values are taken into account in another run of the step 4a. This procedure is repeated until no more collisions appear.

6. Implementation, evaluation and visualization

This chapter shall give an insight into the solver functionality of the simulation. Important sub-problems are described and solutions are presented with respect to efficient hardware-usage. The respective sub-routines of the solver are thereafter benchmarked, before a final focus will be put on visualization..

6.1 Computational challenges

The sub-steps of the algorithm as summarized at the end of the previous chapter can be divided into two categories according to their working principle:

1. Node-wise: only information of a single particle and the nodes occupied by it are needed (Steps 1c, 4a, 4b, 5)

2. All nodes: information of the whole grid is necessary (Steps 1a, 2, 3)

Due to the time restrictions of this thesis, only part of the simulation could be implemented and tested. The matrix assembly for all necessary matrices has been implemented successfully with and without the boundary nodes set by homogeneous Dirichlet conditions. The assembly of the matrices $\mathbf{R}$ and $\mathbf{S}$ during run-time could unfortunately not be ported to CUDA in time, as well as the integration of velocity data over the particle domain.
But with the existing parts of the implementation it is already possible to test the linear equation solver intended for steps 1a, 2 and 3 on the exact problem. The algorithm will be explained in the following section.

Furthermore a parallel collision algorithm can already be presented with test results for different numbers particles.

6.1.1 An adapted conjugate gradient method

The efficient solution of the linear equation systems arising in the previous chapter is a complex problem on its own. Usually either a multi-grid solver is employed or a preconditioned conjugate gradients (CG) solver, as it is done here. The CG solver is simply faster to implement and fits thus the time frame of this thesis better.

The theoretical background of Krylov space methods and the CG solver as a basic example can be found in more or less any introduction to numerical mathematics. The resulting method as presented 1952 by M.R. Hestenes and E. Stiefel reads for $\mathbf{Ax} = \mathbf{b}$ with the initial guess being $\mathbf{x}_0$ as follows:

$$
\begin{aligned}
\mathbf{r}_0 &:= \mathbf{b} - \mathbf{A}\mathbf{x}_0 \\
\mathbf{p}_0 &:= \mathbf{r}_0 \\
k &:= 0 \\
&\textbf{do} \\
\alpha_k &:= \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k} \\
\mathbf{x}_{k+1} &:= \mathbf{x}_k + \alpha_k \mathbf{p}_k \\
\mathbf{r}_{k+1} &:= \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k \\
&\textbf{if } \mathbf{r}_{k+1} \text{ is sufficiently small } \textbf{then} \text{ exit loop} \\
\beta_k &:= \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k} \\
\mathbf{p}_{k+1} &:= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \\
k &:= k + 1 \\
&\textbf{end do}
\end{aligned}
$$

We note that in each step a matrix vector product is needed ($\mathbf{q}_{k+1} := \mathbf{A}\mathbf{p}_k$), and three scalar products: two for the computation of α and another for β. The remaining operations are vector updates with element wise multiplication and addition.

For the implementation on the GPU it would be preferable to combine as much operations as possible into a single kernel. If for example, each scalar product is computed by starting the same kernel three times, between each kernel launch all the GPU's registers would have to be flushed and re-set.

But in order to perform every scalar product operation in parallel and also all vector updates, the operation of the CG method have to be re-ordered. At first the problem arises, that $\mathbf{p}$ depends via β on the scalar product involving $\mathbf{r}$, while $\mathbf{r}$ depends on the just updated vectors.

A solution to this was presented in [Meur85] and re-used for FPGA-implementation in [StGö06]. The idea is to substitute $\rho_{k+1} := \mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}$ by σ_k, that depends only on values computed in the previous step:

$$\sigma_k := \alpha_k^2 \mathbf{q}_k^{\mathrm{T}}\mathbf{q}_k - \alpha_k \mathbf{p}_k^{\mathrm{T}}\mathbf{q}_k.$$

Algebraically, ρ_{k+1} and σ_k are identical as we have:

$$\begin{aligned}
(\mathbf{r}_{k+1}-\mathbf{r}_k)^T(\mathbf{r}_{k+1}-\mathbf{r}_k) &= (-\alpha_k^2\mathbf{A}\mathbf{p}_k)^T(-\alpha_k^2\mathbf{A}\mathbf{p}_k) && \Leftrightarrow\\
\mathbf{r}_{k+1}^T\mathbf{r}_{k+1} - 2\underbrace{\mathbf{r}_{k+1}^T\mathbf{r}_k}_{=0\ (\mathbf{r}\perp\mathbf{p})} + \mathbf{r}_k^T\mathbf{r}_k &= \alpha_k^2(\mathbf{q}_k)^T(\mathbf{q}_k) && \Leftrightarrow\\
\mathbf{r}_{k+1}^T\mathbf{r}_{k+1} &= \alpha_k^2(\mathbf{q}_k)^T(\mathbf{q}_k) - \mathbf{r}_k^T\mathbf{r}_k && \Leftrightarrow\\
\mathbf{r}_{k+1}^T\mathbf{r}_{k+1} &= \alpha_k^2(\mathbf{q}_k)^T(\mathbf{q}_k) - \alpha_k\mathbf{p}_k^T\mathbf{q}_k && \Leftrightarrow\\
\rho_{k+1} &= \sigma_k.
\end{aligned}$$

From the numerical point of view, we are of course introducing new round-off errors. In [StGö06] it was investigated that ρ should be computed right after this update procedure directly form $\mathbf{r}$. By doing so, the altered CG version keeps the original behavior concerning convergence and stability.

The new algorithm reads:

$$\begin{aligned}
\mathbf{r}_0 &:= \mathbf{b} - \mathbf{A}\mathbf{x}_0\\
\mathbf{p}_0 &:= \mathbf{r}_0\\
k &:= 0\\
&\textbf{do}\\
\mathbf{x}_{k+1} &:= \mathbf{x}_k + \alpha_k\mathbf{p}_k\\
\mathbf{r}_{k+1} &:= \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k\\
\mathbf{p}_{k+1} &:= \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k\\
\mathbf{q}_{k+1} &:= \mathbf{A}\mathbf{p}_{k+1}\\
&\quad \textbf{if } \mathbf{r}_{k+1} \text{ is sufficiently small } \textbf{then} \text{ exit loop}\\
\rho_{k+1} &:= \mathbf{r}_{k+1}^T\mathbf{r}_{k+1}\\
\alpha_{k+1} &:= \frac{\rho_k}{\mathbf{p}_{k+1}^{\mathrm{T}}\mathbf{q}_{k+1}}\\
\sigma_{k+1} &:= \alpha_{k+1}(\mathbf{q}_{k+1}^{\mathrm{T}}\mathbf{q}_{k+1} - \mathbf{p}_{k+1}^{\mathrm{T}}\mathbf{q}_{k+1})\\
\beta_{k+1} &:= \frac{\sigma_{k+1}}{\rho_{k+1}}\\
k &:= k+1\\
&\textbf{end do}
\end{aligned}$$

This can be perfectly split into three kernels:

1. Update the three vectors $\mathbf{x}_{k+1}$, $\mathbf{r}_{k+1}$ and $\mathbf{p}_{k+1}$ in parallel,
2. Compute the matrix-vector product $\mathbf{A}\mathbf{p}_{k+1}$ and
3. Compute the three scalar products $\mathbf{r}_{k+1}^T\mathbf{r}_{k+1}$, $\mathbf{p}_{k+1}^T\mathbf{q}_{k+1}$ and $\mathbf{q}_{k+1}^T\mathbf{q}_{k+1}$ in parallel.

```
__global__ void update(int DIM, double *x, double *r, double *p, double
    *q, double *alpha, double *beta)
{
  int i, idx=blockIdx.x*blockDim.x+threadIdx.x;
  for (i=idx; i<DIM; i+=gridDim.x*blockDim.x)
  {
    x[i] += alpha*p[i];          //x = x + alpha * p
    r[i] -= alpha*q[i];          //r = r - alpha * q
    p[i]  = r[i] + beta*p[i];    //p = r + beta * p
  }
}
```

Listing 6.1: Parallel vector update

Parallel vector update: This is the simplest case, when each thread can process one element update independently. This is done by looping over all elements such that consecutive elements are processed by consecutive threads. Listing 6.1 shows the respective CUDA kernel.

Matrix vector product: Here either a *gemm* (cf. 4.3.1) or *spmv* kernel (cf. 4.3.2) is employed, in our case of course a *spmv* version that uses textures for faster data accesses. As now the reads to the source vector and writes to the target vector are not random anymore, it is expected that using explicit prefetching to shared memory might be of benefit here. Because of the limited time frame of this thesis, this approach could unfortunately not be investigated, though the method is explicitly chosen to result in predictable data dependencies.

Parallel scalar product: Evaluating a scalar product is equal to performing an element-wise product of the two vectors and a sum reduction thereafter. The element-wise product can be done as described in the vector-update procedure above. For the reduction operation there are several possibilities of implementation, alone seven are presented in the CUDA SDK.

A main question is, if the final reduction shall be done on the CPU or GPU. Naturally, the GPU is only faster, if almost all of the streaming processors are in use, what cannot be satisfied if the number of remaining intermediate sums is smaller then the SP count. An advantage of using only the GPU for the whole reduction is the possibility to combine all three kernels in a single one and thus avoiding the execution overheads. But this would on the other hand require a complete inter-block synchronization that is currently hardly manageable with CUDA.

It has been tested to implement a full inter-block synchronization with the help of atomic functions. These experiments showed, that is was actually faster to start two kernels then a fused one with synchronization. For our problem this means, that it is more efficient to keep three separate kernels and also allow the CPU to do the final reduction.

6.1.2 Parallel handling of particle collisions

Depending on the number of particles, different work distribution methods promise best performance:

Low particle count: each thread handles one (possible) collision,

Medium particle count: each thread handles all collisions of a single particle,

High particle count: each thread handles the collisions occurring in a particular part of the domain.

Between n particles there are n^2 possible collisions. If n is significantly lower then the number of processors cores available, the best results in terms of performance are surely produced, if each core checks for one possible collision. This distribution can also be arranged in a way, such that the SP cores read consecutive particle data and no unnecessary delay is produced.

For an increasing number of particles, it is more reasonable to bind each particle to an individual SP, and let this one check for possible collisions.

For a high number of particles it is of course a waste of resources to check for improbable collisions, especially a second or third time when repeating the collision handling in the same time step of the simulation algorithm. Here, the most promising approach is to do a domain decomposition, e.g. according to the cube division of the fluid grid, sort the particles according to the cubes they reside (at least partially) in and let each processor core resolve the collision in a particular set of cubes.
As it is not to be expected that only particles with consecutive numbers collide, the data accesses will have random patterns, what will result in large performance losses. Implementing such a technique needs careful drafting, in order to achieve a high performance nevertheless.

6.2 Performance evaluation

Conjugate gradients algorithm

The pipelined CUDA implementation of the CG algorithm, as described above has been compared to implementations with CUBLAS and Intel MKL. As CUBLAS does not include a sparse matrix-vector multiplication routine, the same manual CUDA implementation of this kernel was used as for the CUDA-only version. The MKL version uses *csrmv* of the *SparseBLAS* package.

The first tests were conducted for matrices gained from the discretized fluid flow problem, namely the diffusion problem (step 2). When discretizing the unit cube with a uniform h for every dimension in space and excluding the boundary nodes in order to enforce the homogeneous Dirichlet condition, we obtain matrix properties, as shown in table 6.1.

When evaluating the results gained with the three CG implementations in figure 6.1, we notice different phenomena:

Matrix properties		
h	Dimension	Non-zeros
10	6859	151775
20	59319	1426595
30	205379	5072215
40	493039	12336635

Table 6.1: Resulting matrix properties for discretizations of the unit cube

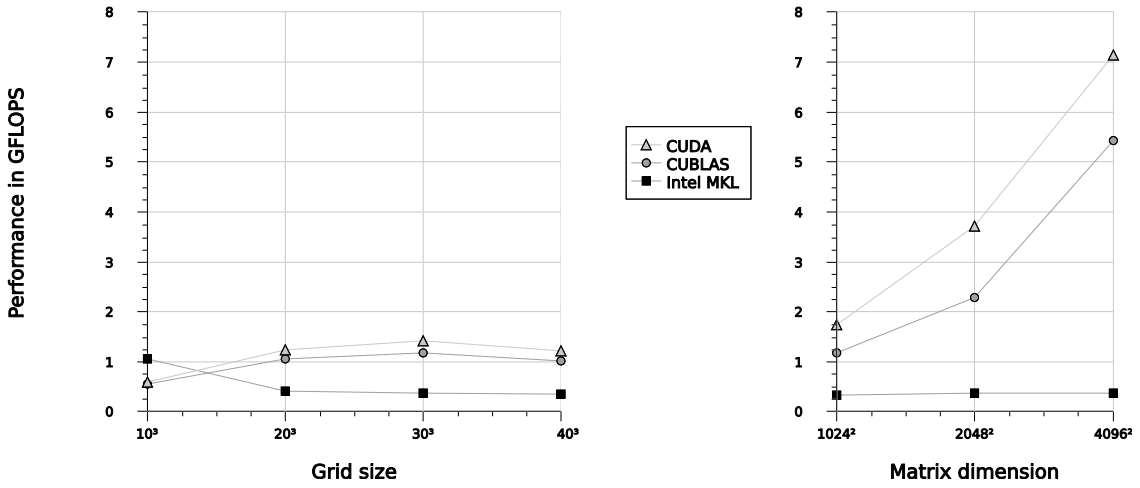

Figure 6.1: Test results of conjugate gradients implementations on the diffusion problem (left) and a dense matrix (right)

1. Increasing the problem size from 10^3 sub-cubes to 20^3 changes the situation completely, MKL breaks down to 50 % of the previous result while both GPU implementations gain 100 %, achieving speed-ups over MKL of 2.5 (CUBLAS) and 3 (CUDA).

2. Further increasing of the problem size to a tripled and octupled matrix dimension does not have a major influence on the performance result.

3. For the grid sizes 20^3 to 40^3 the CUDA-only implementation achieves $\approx$ 1.2 to 1.4 GFLOPS, resulting in a speed-up of 3 to 3.9 compared to MKL and 1.2 over CUBLAS.

For comparison, also tests with dense matrices have been conducted in order to eliminate the influence of the sparse matrix-vector multiplication. Before discussing the results it is to be further notices, that for the problems considered also the vector sizes are much smaller, resulting in a lower influence of the vector-only operations. Opposite to the sparse matrix tests, now the CUDA version is equipped with the CUBLAS *gemm* implementation.

The results can be found in figure 6.1 as well and show again interesting figures:

1. The MKL implementation can again only achieve between 0.32 and 0.36 GFLOPS, no benefit is gained from the dense data structure.
2. CUDA and CUBLAS can increase their performance by a factor of 2, when the problem size is quadrupled. (Note, the vector operations are done only on doubled dimensions).
3. The influence of the pipelined vector operations is much higher, resulting in a speed-up of CUDA compared to CUBLAS of 1.3 to 1.6.
4. The maximum speed-up of CUDA over MKL is 19.6 and achieved for the largest problem size.

We, can conclude again, that the main performance obstacle of the GPU is non-uniform memory access, what is clearly to be seen when comparing sparse an dense CG results. Furthermore it has been again of value to invest time to develop a hardware-fitted version, thereby achieving 20 to 60 % better results then the provided library, though only using standard kernels.

Particle collision algorithm

The collision sub-step can be divided into two phases, the resolution of particle-particle collisions and particle-wall collisions. The basic implementation used here, assumes a medium number of particles and thus assigns every particle to specific thread. The sequential CPU version though uses only one thread and loops over all particles instead.

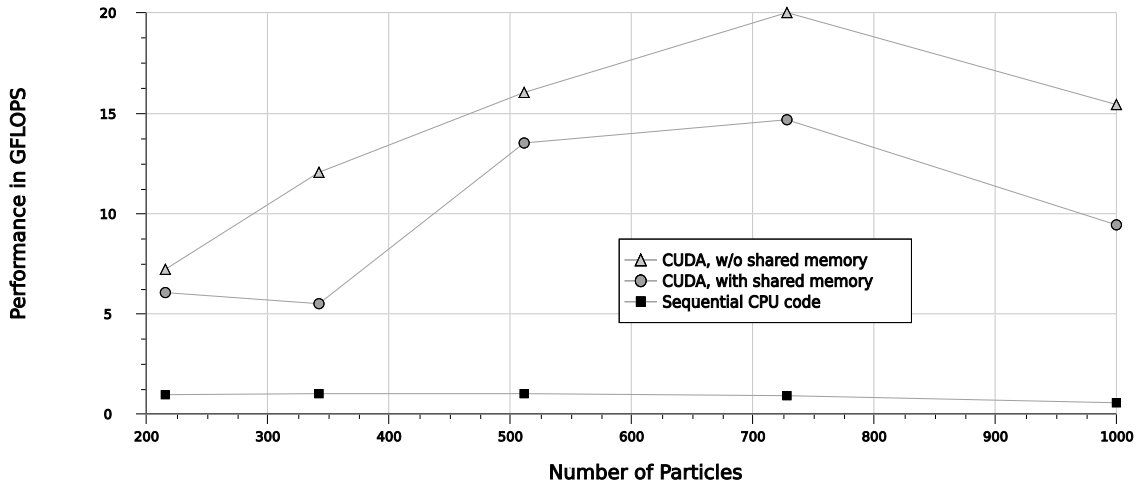

Figure 6.2: Test results for particle collision implementations for different particle counts

As expected the CPU performance is really low for this non-optimized code, again only about 1 GFLOPS. Far more interesting are the results gained with the CUDA implementation. Again it becomes visible, how important coalesced reads and uniform thread behavior is for this architecture. The peak performance of 20 GFLOPS

can be considered as quite high, compared to the standard kernels benchmarked in chapter 4

Figure 6.2 shows besides the mentioned CUDA and plain C versions also another one that uses shared memory in order to cache the particle positions of all particles prior to starting the collision check. Obviously this creates an overhead that leads to performance break downs of up to 60 %. The reads from global memory seem to be already perfectly aligned, such that saving and loading values from shared memory is a futile detour.

6.3 Visualization

Visualizing fluids in three dimensions, in a way that all important information is clearly visible is an elaborate task, that is complicated even more when particles are involved. Though much research has been conducted in the field of 3D flow visualization, examples of particulate flow visualization are rare. The consequence was to develop an own visualization system that displays not only the particles within the three-dimensional domain, but gives also information on the fluid behavior.

In order not to loose time by low-level graphics programming, the OpenGL based Visualization Toolkit (VTK) was employed. VTK is an open source C++ class library that is widely used in science for tasks in 2D and 3D visualization.

The sphere-shaped particles considered in this thesis can be created in a simple way with the built-in features of VTK. A number of *vtkSphereSource* objects is created that represent spheres discretized by polygons. These spheres can be simply varied in size and color and rendered on the screen. In order to emphasize the domain geometry, small tubes can for example be drawn along the edges (*vtkCubeSource, vtkExtractEdges, vtkTubeFilter*).

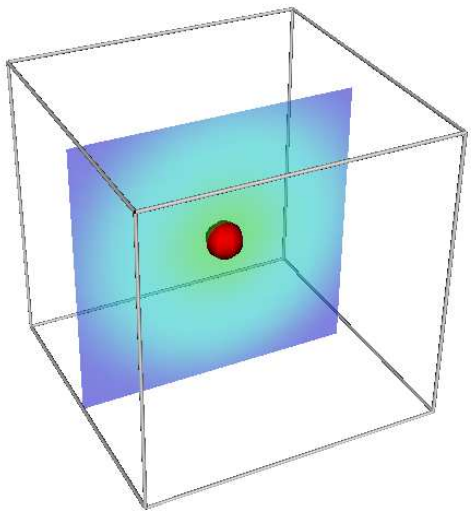

Figure 6.3: Particle and a cut-plane visualizing the fluid flow speed

Standard techniques for visualizing flows in three-dimensions include

- Cut-planes that visualize one slice of the fluid velocity or pressure field (cf. figure 6.3),
- Glyphs that indicate the flow directions and magnitude by arrows or other symbols(cf. figure 6.4),
- Stream lines, indicating the trajectory of a virtual fluid particle,
- Indicator particles, that flow along the fluid and follow stream lines,
- Iso-surface, that combine points of the same value in the fluid field into a surface.

From the above mentioned methods several could automatically be excluded, such as iso-surfaces and indicator particles. These would clearly not support the presentiveness of the particulate flow visualization. Stream-lines can be of high interest, when only a few particles are involved, but is not the case this thesis focuses on.

The remaining techniques are glyphs and cut-planes. When thinking of a high particle density, such as shown in figure 6.5 also glyphs are not eligible anymore and it is doubtful what information could be drawn from a fixed cut-plane during run-time.

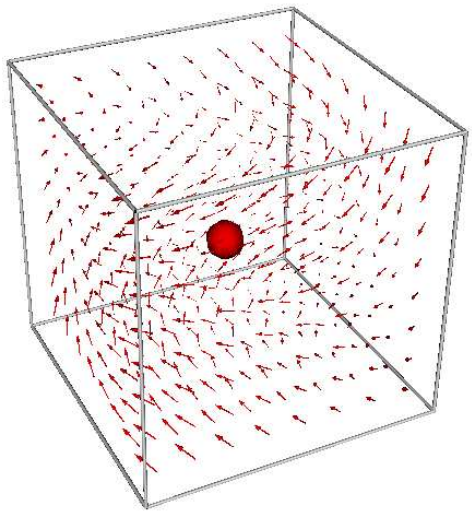

Figure 6.4: Flow field visualization using glyphs

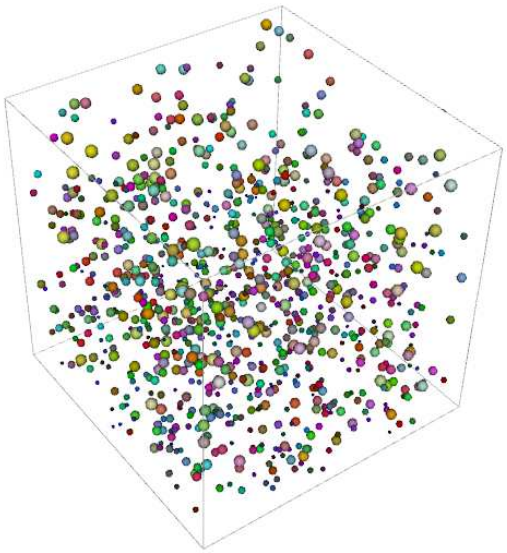

Figure 6.5: One thousand particles visualized using VTK

6.4 Summary

The results gained from the partial implementation of particulate flow simulation backup and complement the findings from chapter 4. GPUs are well suited to accelerate problems of computational fluid dynamics, not as much as advertisements promise, but with considerable speed-ups. These improvements compared to the already optimized MKL library already be achieved partially with comparably low effort, by simply redirecting calls to CUBLAS.

It proved further useful to hand-optimize the kernel for particular sub-routines, however the benefit is hard to predict in general. The same applies to the usage of shared memory: being very useful in some cases, we got significant performance losses for our particle collision algorithm.

In order to survey the collision tests, a visualization tool was implemented that was equipped with basic add-ons for fluid visualization. As long as the particle density is not too high, common visualization techniques for fluids can be used. In the case of many or clustered particles, it remains an open questions how to maintain presentiveness.

7. Conclusion and future directions

7.1 Performance of GPUs in scientific computing

The tests conducted in chapter 4 have shown that for single kernels out of the BLAS collection, the speed-ups achieved in comparison to the also highly optimized Intel MKL library are up to 20 and mostly at least 10, when transfers are not taken into account. For an algorithm that can hide host-device memory transfers by calculations, this improvement is impressive, especially with regard to the low financial effort to be undertaken.

The assessment turns out different for algorithms that are not as computationally intensive or cannot provide coalesced reads and writes, or use indirect addressing. An example thoroughly studied, was the *spmv* kernel, that has been benchmarked individually with different implementations and within a CG algorithm. Here the measured speed-ups were only 2, without transfer overheads and 3 as part of the CG solver. At this point it is to be considered, whether to invest into a graphics card or more CPU core.

When also taking the results into account, that were obtained with tests based on particle interaction, we can give a positive evaluation for the usage of GPUs for the particulate flow problem. Here the results achieved were up to 60 times higher then for a sequential CPU code.

As it has been discussed in depth in this thesis, an answer whether or not GPUs are qualified for tasks in scientific computing cannot be given in general, as the absolute performance and possible speed-up is highly problem dependent. The architectural properties and the restrictions resulting from them, do not favor highly coupled problems with irregular data dependencies. Only algorithms with low data dependencies and high computational intensity can make full use of the SIMD model and achieve the theoretical speed-ups between 50 and 100.

7.2 Expenditure and benefit of hardware-aware numerical mathematics

The example of CUDA shows, that a sophisticated hardware architecture does not necessarily require complicated low-level programming. In the meantime between starting and finishing this paper, CUDA has developed towards a stable programming tool, allowing even directly debugging GPU code. At the beginning the development of CUDA programs was slowed down by major bugs in memory management and also the provided libraries. These issues seem to be resolved, such that effective code generation is possible with fairly low additional effort. Practice in CUDA programming is gained quickly and even complicated algorithms are not harder to develop then with e.g. MPI.

The results in chapter 4 show, that programming on CUDA level is not even necessary when standard CUBLAS or CUFFT routines can be applied. In the case of CUBLAS, the results are fairly good and already adapted to the specific graphics cards. In this case, it is surely not advisable to invest further effort. The *spmv* example showed in addition, that with simple and quickly integrated texture operations even better results can be achieved then with more complicated shared memory management.

In the case of CUFFT, counterexamples on the other hand showed that up to three times higher performance is possible with even more optimized implementations. In this case, dealing with the particular hardware possibilities did result in a significant speed-up.

From the experience gained in this thesis, we can conclude, that it is possible to achieve considerable performance improvements in short time by outsourcing computational intensive tasks to the GPU with the help of CUBLAS or CUFFT. Manual CUDA programming can be of benefit if data accesses are structured and at best continuously. In this case the usage of textures most probably further increases the performance. Working with the GPUs shared memories can lead to improvements in particular situations, but creates new difficulties e.g. in terms of concurrent writes.

7.3 Future Directions

7.3.1 Particulate flow modeling and accuracy improvements

The current simulation method is not yet optimized for accuracy. It is to be clarified, that the method is not expected to produce wrong results, but with increasing problem complexity, the deviations from data gained in laboratory experiments might increase.

First of all, we have to remind of the model improvements concerning fluid-structure interaction given in 2.3.5. When more complex problems are to be analyzed, the model has to be extended respectively. An open question remains, whether the continuum hypothesis, that constituted the basis of our work and the resulting Navier-Stokes equations correctly model the physical processes and if not in general, on

what scales they hold. At this point we can only rely on the fact, that data gained by solving the Navier-Stokes equations has proven to map real-world phenomena sufficiently well in the past.

Another source of possible errors is the very basic collision model. Here, further research has to be conducted, in order to accurately model the processes happening in the fluid film between two approaching particles and derive methods that can be applied without introducing a much smaller scale.

Concerning the discretization, second order in time accuracy is an improvement that can be achieved with fairly few extra work, further improvements will need substantially more effort.

7.3.2 Outlook on future hardware and programming models

The development of CUDA and the preparation of GPUs to take over more general computational tasks has only been the first big step towards effectively combining the advantages of compute cores and classical x86 processors. In the long run, accelerators might even evolve in direction of a fusion between GPU and CPU architectures. Currently all three major GPU fabricators, AMD, Intel and NVIDIA are working on solution that address this hybrid chip in different ways. Meanwhile an open source framework aims to facilitate programming these new types of hardware.

MIMD/x86 GPUs and G/CPUs

- **AMD Fusion:** This project is currently in an early stage, such that not much reliable information is available. Fusions is based on a modular many-core concept, including general processor cores as well as GPU-like cores for computational intensive tasks on the same chip. This way, the problem of the bandwidth bottleneck PCIe is avoided, such that data can be transferred into the compute cores with high throughput. It will be interesting to see how the work division on the cores will be arranged, if (semi-)automatically or manually as it is done with Cell.

- **Intel Larrabee:** The Larrabee project approaches the CPU/GPU hybrid from the CPU side. A many-core CPU is built by combining 32+ cores of an earlier architecture (Intel Pentium) that has been enriched with more SIMD units to catch up with the performance achieved by current GPUs. This concept allows high performance, when pipelining and/or vector processing is possible, while other conveniences of x86 multi-core CPUs are preserved, such as cache coherency over the whole chip.

  From the current information available, Larrabee will firstly only be available as add-on PCIe card, thus not solving the bus bandwidth issue. There is not much information on the programming method available, only that besides the x86 instructions, Larrabee specific instructions will be introduced. It will be possible to use Larrabee as GPU, such that also programming via DirectX and OpenGL will be an option as well.

- **NVIDIA GT300:** As announced by NVIDIA, the next generation GT300 GPU series will further leave traditional GPU stream processing and rather constitute a *cGPU* design, where *c* stands for *compute* and refers to the claim to outsource more and more tasks to the graphics card. This represents a CPU/GPU hybrid approach from the GPU side. An improvement will also be the possibility to use MIMD processing on 512 cores instead of SIMD on 240 as before. As reliable details are missing the impact on the programming style and the new possibilities cannot yet be estimated.

OpenCL: One of the biggest problems with using accelerators so far is, that every architecture depends on a different programming model, that is not compatible with another. An answer to this might be given by *OpenCL (Open Computing Language)*, a framework for developing programs across heterogeneous platforms. As well as CUDA, OpenCL is C99 based and differentiates between standard host code and *kernels* that execute on OpenCL devices. Particular APIs will then define and control the platforms. It will be interesting how OpenCL will facilitate the development for different architectures and how fast the manufacturers of accelerator hardware adopt to the new open standard and if the performance of dedicated programming tools can be maintained.

It is to be concluded, that in the short and medium term, accelerators will continuously play an important role in scientific computing, achieving significant to enormous performance improvements over commodity hardware.

In the long run, the developments are hard to predict, but it is for certain that numerical methods and algorithm design will have to evolve alongside the trends in hardware if satisfactory performance result are to be achieved.

A. Appendix

A.1 Element matrices of the discrete particulate flow problem

For the tetrahedra shown in figure A.1, we derived general expressions for the element matrices in section 5.2.3.2.

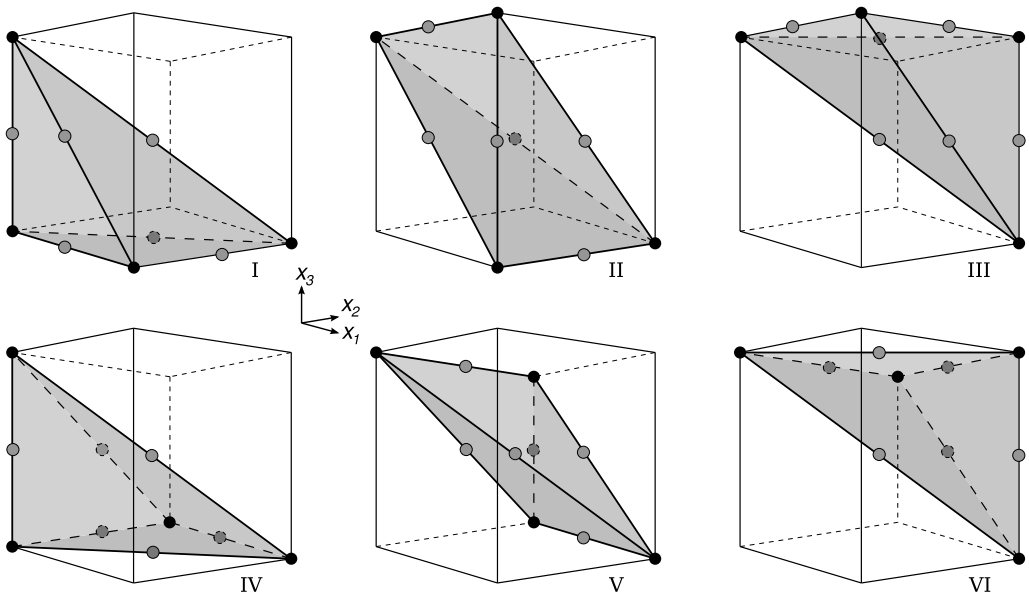

Figure A.1: Tetrahedra used for triangulation of Ω.

Specifically, for the tetrahedra embedded into cubes of edge length h, we obtain the following element matrices for the tetrahedra I to VI.

Pressure element stiffness matrix

$$B_I = B_{IV} = \frac{h}{6}\begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix}$$

$$B_{II} = B_V = \frac{h}{6}\begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

$$B_{III} = B_{VI} = \frac{h}{6}\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

Velocity (component) element stiffness matrices

$$A'_I = A'_{IV} = \frac{h}{30}\begin{bmatrix} 6 & -5 & 1 & -2 & 1 & 0 & -5 & 2 & 1 & 1 \\ -5 & 24 & -5 & -4 & 0 & 1 & 0 & -4 & -8 & 1 \\ 1 & -5 & 6 & 2 & -5 & 1 & 1 & -2 & 1 & 0 \\ -2 & -4 & 2 & 24 & -8 & -1 & 4 & -12 & -4 & 1 \\ 1 & 0 & -5 & -8 & 16 & -4 & -4 & 4 & 0 & 0 \\ 0 & 1 & 1 & -1 & -4 & 3 & 0 & 1 & -1 & 0 \\ -5 & 0 & 1 & 4 & -4 & 0 & 16 & -8 & 0 & -4 \\ 2 & -4 & -2 & -12 & 4 & 1 & -8 & 24 & -4 & -1 \\ 1 & -8 & 1 & -4 & 0 & -1 & 0 & -4 & 16 & -1 \\ 1 & 1 & 0 & 1 & 0 & 0 & -4 & -1 & -1 & 3 \end{bmatrix}$$

$$A'_{II} = A'_V = \frac{h}{30}\begin{bmatrix} 6 & -5 & 1 & -2 & 1 & 0 & -5 & 2 & 1 & 1 \\ -5 & 16 & -4 & 4 & 0 & 0 & 0 & -8 & -4 & 1 \\ 1 & -4 & 3 & 1 & -1 & 0 & 1 & -1 & 0 & 0 \\ -2 & 4 & 1 & 24 & -4 & -1 & -4 & -12 & -8 & 2 \\ 1 & 0 & -1 & -4 & 16 & -1 & -8 & -4 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 & 3 & 1 & 1 & -4 & 1 \\ -5 & 0 & 1 & -4 & -8 & 1 & 24 & -4 & 0 & -5 \\ 2 & -8 & -1 & -12 & -4 & 1 & -4 & 24 & 4 & -2 \\ 1 & -4 & 0 & -8 & 0 & -4 & 0 & 4 & 16 & -5 \\ 1 & 1 & 0 & 2 & 1 & 1 & -5 & -2 & -5 & 6 \end{bmatrix}$$

$$A'_{III} = A'_{VI} = \frac{h}{30}\begin{bmatrix} 3 & -1 & 0 & -1 & 0 & 0 & -4 & 1 & 1 & 1 \\ -1 & 16 & -1 & -4 & 0 & 1 & 0 & -4 & -8 & 1 \\ 0 & -1 & 3 & 1 & -4 & 1 & 0 & -1 & 1 & 0 \\ -1 & -4 & 1 & 24 & 4 & -2 & -8 & -12 & -4 & 2 \\ 0 & 0 & -4 & 4 & 16 & -5 & -4 & -8 & 0 & 1 \\ 0 & 1 & 1 & -2 & -5 & 6 & 1 & 2 & -5 & 1 \\ -4 & 0 & 0 & -8 & -4 & 1 & 16 & 4 & 0 & -5 \\ 1 & -4 & -1 & -12 & -8 & 2 & 4 & 24 & -4 & -2 \\ 1 & -8 & 1 & -4 & 0 & -5 & 0 & -4 & 24 & -5 \\ 1 & 1 & 0 & 2 & 1 & 1 & -5 & -2 & -5 & 6 \end{bmatrix}$$

Velocity (component) element mass matrix

$$M'_T = \frac{h^3}{2520}\begin{bmatrix} 6 & -4 & 1 & -4 & -6 & 1 & -4 & -6 & -6 & 1 \\ -4 & 32 & -4 & 16 & 16 & -6 & 16 & 16 & 8 & -6 \\ 1 & -4 & 6 & -6 & -4 & 1 & -6 & -4 & -6 & 1 \\ -4 & 16 & -6 & 32 & 16 & -4 & 16 & 8 & 16 & -6 \\ -6 & 16 & -4 & 16 & 32 & -4 & 8 & 16 & 16 & -6 \\ 1 & -6 & 1 & -4 & -4 & 6 & -6 & -6 & -4 & 1 \\ -4 & 16 & -6 & 16 & 8 & -6 & 32 & 16 & 16 & -4 \\ -6 & 16 & -4 & 8 & 16 & -6 & 16 & 32 & 16 & -4 \\ -6 & 8 & -6 & 16 & 16 & -4 & 16 & 16 & 32 & -4 \\ 1 & -6 & 1 & -6 & -6 & 1 & -4 & -4 & -4 & 6 \end{bmatrix}$$

Velocity (component) pressure-gradient element matrices

$$C'_{1,I} = C'_{2,IV} = C'_{2,II} = C'_{1,V} = \frac{h^2}{120}\begin{bmatrix} 1 & -1 & 0 & 0 \\ -4 & 4 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ -4 & 4 & 0 & 0 \\ -4 & 4 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ -4 & 4 & 0 & 0 \\ -4 & 4 & 0 & 0 \\ -4 & 4 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}$$

$$C'_{2,I} = C'_{1,IV} = C'_{1,III} = C'_{2,VI} = \frac{h^2}{120}\begin{bmatrix} 0 & 1 & -1 & 0 \\ 0 & -4 & 4 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & -4 & 4 & 0 \\ 0 & -4 & 4 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & -4 & 4 & 0 \\ 0 & -4 & 4 & 0 \\ 0 & -4 & 4 & 0 \\ 0 & 1 & -1 & 0 \end{bmatrix}$$

$$C'_{1,II} = C'_{2,V} = C'_{2,III} = C'_{1,VI} = \qquad C'_{3,t}|_{t=I,\ldots,VI} =$$

$$\frac{h^2}{120}\begin{bmatrix} 0 & 0 & 1 & -1 \\ 0 & 0 & -4 & 4 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -4 & 4 \\ 0 & 0 & -4 & 4 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -4 & 4 \\ 0 & 0 & -4 & 4 \\ 0 & 0 & -4 & 4 \\ 0 & 0 & 1 & -1 \end{bmatrix} \qquad \frac{h^2}{120}\begin{bmatrix} 1 & 0 & 0 & -1 \\ -4 & 0 & 0 & 4 \\ 1 & 0 & 0 & -1 \\ -4 & 0 & 0 & 4 \\ -4 & 0 & 0 & 4 \\ 1 & 0 & 0 & -1 \\ -4 & 0 & 0 & 4 \\ -4 & 0 & 0 & 4 \\ -4 & 0 & 0 & 4 \\ 1 & 0 & 0 & -1 \end{bmatrix}$$

Velocity-divergence (component) pressure element matrices

$$D'_{1,I} = D'_{2,IV} = D'_{2,II} = D'_{1,V}$$

$$\frac{h^2}{120}\begin{bmatrix} -3 & 4 & -1 & -4 & 4 & 0 & -4 & 4 & 0 & 0 \\ 1 & -4 & 3 & -4 & 4 & 0 & -4 & 4 & 0 & 0 \\ 1 & 0 & -1 & -8 & 8 & 0 & -4 & 4 & 0 & 0 \\ 1 & 0 & -1 & -4 & 4 & 0 & -8 & 8 & 0 & 0 \end{bmatrix}$$

$$D'_{2,I} = D'_{1,IV} = D'_{1,III} = D'_{2,VI} =$$

$$\frac{h^2}{120}\begin{bmatrix} 0 & -8 & 1 & 8 & 0 & -1 & 0 & -4 & 4 & 0 \\ 0 & -4 & -3 & 4 & 4 & -1 & 0 & -4 & 4 & 0 \\ 0 & -4 & 1 & 4 & -4 & 3 & 0 & -4 & 4 & 0 \\ 0 & -4 & 1 & 4 & 0 & -1 & 0 & -8 & 8 & 0 \end{bmatrix}$$

$$D'_{1,II} = D'_{2,V} = D'_{2,III} = D'_{1,VI} =$$

$$\frac{h^2}{120}\begin{bmatrix} 0 & 0 & 0 & -8 & -4 & 1 & 8 & 4 & 0 & -1 \\ 0 & 0 & 0 & -4 & -8 & 1 & 4 & 8 & 0 & -1 \\ 0 & 0 & 0 & -4 & -4 & -3 & 4 & 4 & 4 & -1 \\ 0 & 0 & 0 & -4 & -4 & 1 & 4 & 4 & -4 & 3 \end{bmatrix}$$

$$D'_{3,I} = D'_{3,II} = D'_{3,III} = D'_{3,IV} = D'_{3,V} = D'_{3,VI}$$

$$\frac{h^2}{120}\begin{bmatrix} -3 & -4 & 0 & -4 & 0 & 0 & 4 & 4 & 4 & -1 \\ 1 & -8 & 0 & -4 & 0 & 0 & 0 & 8 & 4 & -1 \\ 1 & -4 & 0 & -8 & 0 & 0 & 0 & 4 & 8 & -1 \\ 1 & -4 & 0 & -4 & 0 & 0 & -4 & 4 & 4 & 3 \end{bmatrix}$$

A.2 Structure plots of matrices

A discretization into $4 \times 4 \times 4$ cubes yields the following matrices.

Number of elements:	$4 \times 4 \times 4 \times 6$	$= 384$
Number of pressure nodes:	$5 \times 5 \times 5$	$= 125$
Number of velocity nodes:	$9 \times 9 \times 9$	$= 729$

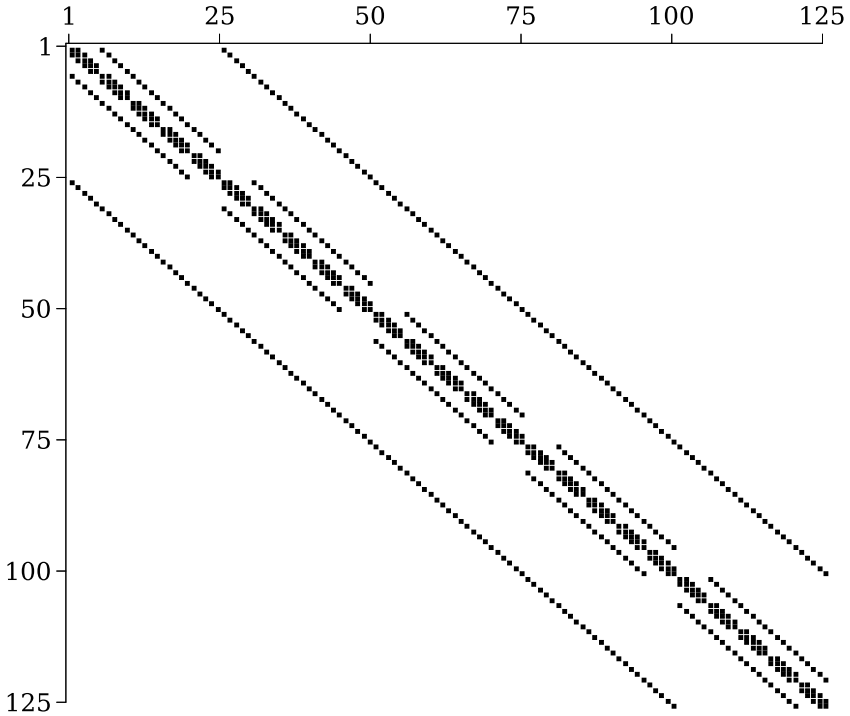

Figure A.2: Pressure stiffness matrix B

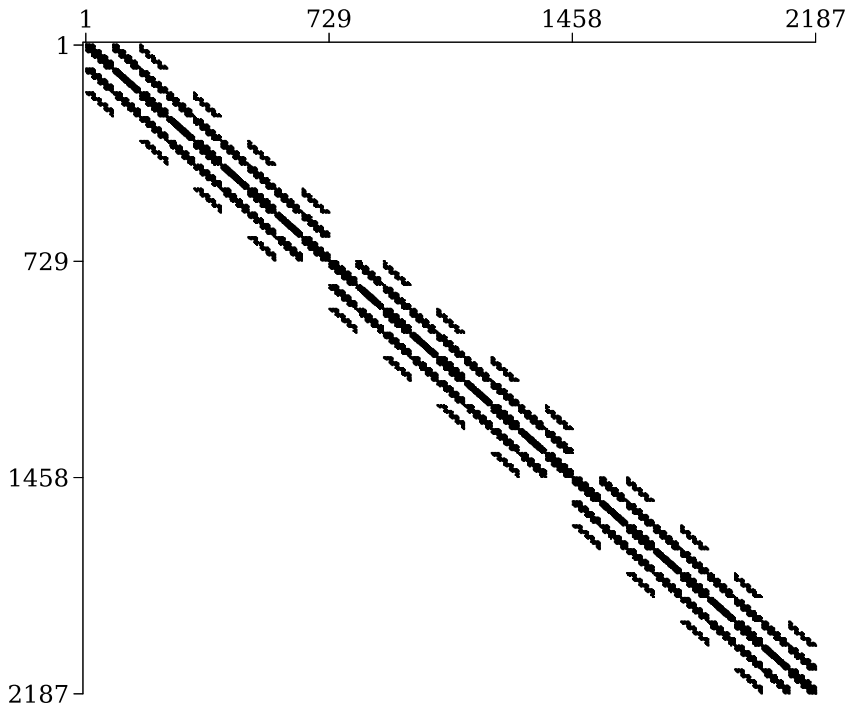

Figure A.3: Velocity stiffness matrix A

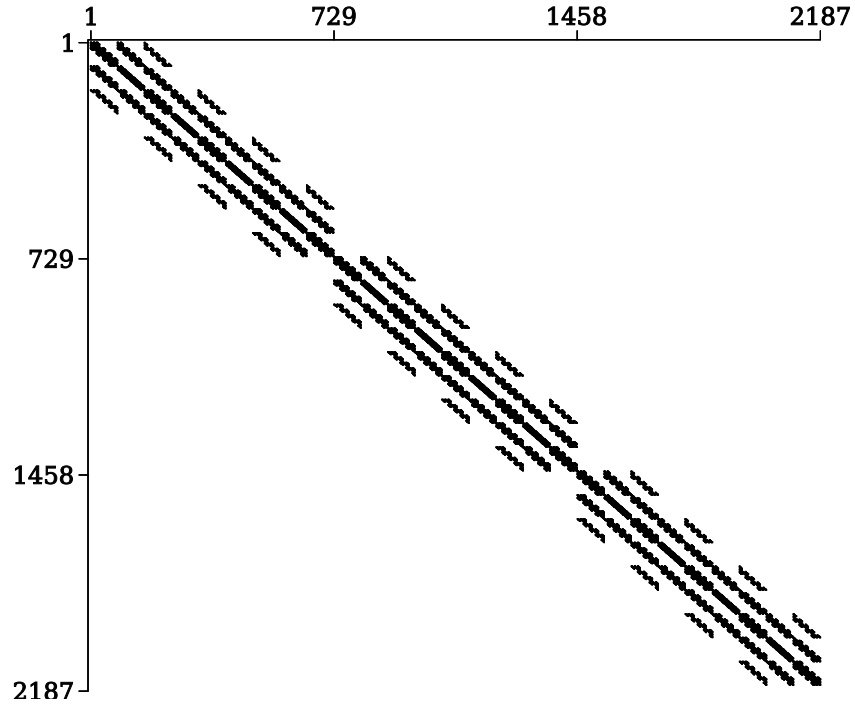

Figure A.4: Velocity mass matrix M

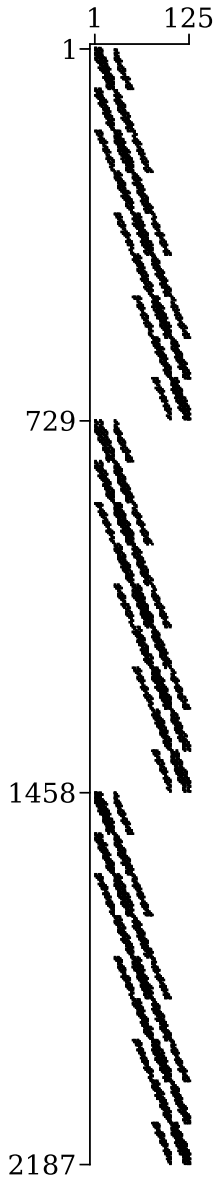

Figure A.5: Velocity pressure-gradient matrix C

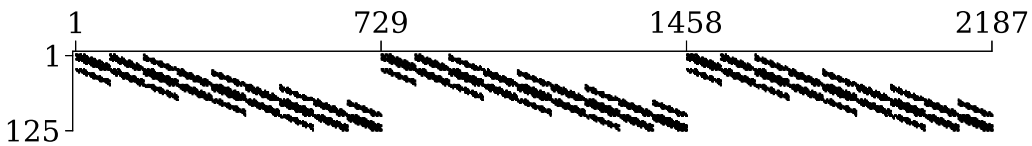

Figure A.6: Velocity-divergence pressure matrix D

Bibliography

[Adva08] Advanced Micro Devices, Inc. *Technical Overview: AMD Stream Computing*, August 2008.

[Batr06] Romesh C. Batra. *Elements of Continuum Mechanics*. AIAA Education Series. American Institute of Aeronautics and Astronautics, Inc., Reston, Virginia. 2006.

[BBCD+94] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA. 1994.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun and Peter Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics*, vol. 22, 2003, pp. 917–924.

[BuGN70] B.L. Buzbee, G.H. Golub and C.W. Nielson. On direct methods for solving Poisson's equations. *SIAM J. Numer. Anal*, vol. 7, 1970, pp. 627–656.

[Clea07] ClearSpeed Technology plc. *ClearSpeed Whitepaper: CSX Processor Architecture*, February 2007.

[Clea08] ClearSpeed Technology plc. *Product Brief: Advance^{TM}e710*, August 2008.

[CoTu65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90), 1965, pp. 297–301.

[CRDI07] Thomas Chen, Ram Raghavan, Jason N. Dale and Eiji Iwata. Cell Broadband Engine Architecture and its first implementation - A performance view. *IBM Journal of Research and Development*, 51(5), 2007, pp. 559–572.

[DGMN03] C. Diaz-Goano, P. Minev and K. Nandakumar. A fictitious domain/finite element method for particulate flows. *Journal of Computational Physics*, vol. 192, 2003, pp. 105–123.

[Flyn66] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12), December 1966, pp. 1901–1909.

[fSta96] International Organization for Standardization. *ISO/IEC 14977:1996: Information technology - Syntactic metalanguage - Extended BNF*. International Organization for Standardization, Geneva, Switzerland. 1996.

[GaHe07] G.P. Galdi and V. Heuveline. Lift and sedimentation of particles in the flow of a viscoelastic liquid in a channel. In *Lecture Notes in Pure and Applied Mathematics*, vol. 252, pp. 75–110. Chapman and Hall, 2007.

[GLDS+08] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith and John Manferdelli. *High performance discrete Fourier transforms on graphics processors*. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, 2008. IEEE Press, pp. 1–12.

[GLLS03] Nolan Goodnight, Gregory Lewin, David Luebke and Kevin Skadron. *A multigrid solver for boundary-value problems using programmable graphics hardware*. In *In Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2003, pp. 102–111.

[Glow03] R. Glowinski. *Handbook of Numerical Analysis*, vol. IX. North-Holland, Amsterdam. 2003.

[GPHJ99] R. Glowinski, T.W. Pan, T.I. Hesla and D.D. Joseph. A distributed Lagrange multiplier/fictitious domain method for particulate flows. *International Journal of Multiphase Flow*, vol. 25, 1999, pp. 755–794.

[GuHP91] E. Guyon, J.P. Hulin and L. Petit. *Hydrodynamique physique*, chapter 4. EDP Sciences / Editions du CNRS, Paris. 1991.

[Half08] Tom R. Halfhill. Parallel Processing with CUDA. InStat Microprocessor Report, January 28 2008.

[HMZZ+08] P. Harvey, R. Mandrekar, Y. Zhou, J. Zheng, J. Maloney, S. Cain, K. Kawasaki, G. Lafontant, H. Noma, K. Imming, T. Plachy and D. Questad. *Packaging the Cell Broadband Engine microprocessor for supercomputer applications*. In *Procedings of Electronic Components and Technology Conference*, vol. 58, May 2008, pp. 1368–1371.

[HuJC92] H.H. Hu, D.D. Joseph and M.J. Crochet. Direct simulation of fluid particle motions. *Theor. Comput. Fluid Dyn. 3*, vol. 169, 1992, pp. 285–306.

[HuPZ01] H.H. Hu, N.A. Patankar and M.Y. Zhu. Direct numerical simulations of fluid-solid systems using the arbitrary Lagrangian–Eulerian techniques. *Journal of Computational Physics*, vol. 169, 2001, pp. 427–462.

[IBM 08] IBM Corporation. *Cell BE Programming Handbook Including PowerXCell 8i 1.11*, May 2008.

[IEEE85] IEEE, New York. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, September 1985.

[KrWe03] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*, vol. 22, 2003, pp. 908–916.

[LaMc01] E. Scott Larsen and David K. McAllister. *Fast Matrix Multiplies using Graphics Hardware*. In *Proceedings of the 2001 ACM/IEEE International Conference on Supercomputing*, November 2001, pp. 55.

[LRDG90] Jed Lengyel, Mark Reichert, Bruce Randall Donald and Donald P. Greenberg. *Real-time robot motion planning using rasterizing computer graphics hardware*. In *Computer Graphics (Proceedings of ACM SIGGRAPH 1990)*, vol. 24, August 1990, pp. 327–335.

[Malv69] Lawrence E. Malvern. *Introduction to the Mechanics of a Continuous Medium*. Series in Engineering of the Physical Sciences. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1969.

[Marc75] G.I. Marchuk. *Methods of Numerical Mathematics*. Springer, New York. 1975.

[Maur99] B. Maury. Direct simulations of 2D fluid-particle flows in biperiodic domains. *Journal Computational Physics*, vol. 156, 1999, pp. 325–351.

[Meur85] Gerard A. Meurant. Multitasking the conjugate gradient on the CRAY X-MP/48. Technischer Bericht, Stanford, CA, USA, 1985.

[Micr08] Microsoft Corporation. *DirectX SDK Documentation: Direct 3D 10 Programming Guide*, August 2008.

[NBGS08] John Nickolls, Ian Buck, Michael Garland and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2), March/April 2008, pp. 40–53.

[NVID08a] NVIDIA Corporation. *The CUDA compiler driver NVCC*, January 04 2008.

[NVID08b] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 2.0*, June 7 2008.

[NVID08c] NVIDIA Corporation. *NVIDIA GeForce® GTX 200 GPU Architectural Overview*, May 2008.

[NVID08d] NVIDIA Corporation. *PTX: Parallel Thread Execution, ISA Version 1.2*, June 17 2008.

[OLGH+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn and Tim Purcell. *A Survey of General-Purpose Computation on Graphics Hardware*. In *Computer Graphics Forum*, vol. 26, March 2007, pp. 80–113.

[PeRa55] D.H. Peaceman and H.H. Rachford. The numerical solution of parabolic and elliptic differential eqations. *Journal of the Society for Industrial and Applied Mathematics*, vol. 3, 1955, pp. 28–41.

[Pope00] S.B. Pope. *Turbulent Flows*. Cambridge University, New York. 2000.

[ScSt71] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, vol. 7, 1971, pp. 281–292.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang and John D. Owens. *Scan primitives for GPU computing*. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Aire-la-Ville, Switzerland, 2007. Eurographics Association, pp. 97–106.

[StGö06] Robert Strzodka and Dominik Göddeke. *Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components*. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, 2006, pp. 259–270.

[Supe07] Supercomputing 2007. *Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms*, 11 2007.

[SuPS08] Junqing Sun, Gregory D. Peterson and Olaf Storaasli. High Performance Mixed-Precision Linear Solver for FPGAs. *IEEE Transactions on Computers*, 57(12), 2008, pp. 1614–1623.

[Taij03] Makoto Taiji. *MDGRAPE-3 chip: A 165-Gflops application-specific LSI for molecular dynamics simulations*. In *Proceedings of Hot Chips 16*, Stanford University, CA, USA, 2003.

[VeMN07] C. Veeramani, P.D. Minev and K. Nandakumar. A fictitious domain formulation for flows with rigid particles: A non-Lagrange multiplier version. *Journal of Computational Physics*, vol. 224, 2007, pp. 867–879.

[Vudu03] Richard W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. Dissertation, University of California, Berkeley, December 2003.

[WaRT02] D. Wan, L. Rivkind and S. Turek. *The Fictitious Boundary Method for the Implicit Treatment of Dirichlet Boundary Conditions with Applications to Incompressible Flow Simulations*. In *Proceedings of the CISC2002*, vol. LNCSE 35, Berlin, 2002. Springer Verlag.

[WaTu07] D. Wan and S. Turek. Fictitious boundary and moving mesh methods for the numerical simulation of rigid particulate flows. *Journal Computational Physics*, 222(1), 2007, pp. 28–56.

[Yane71] N.N. Yanenko. *The Method of Fractional Steps*. Springer, Berlin. 1971.

Index

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 30. April 2009

Tobias Hahn