Universität Trier

Discontinuous Galerkin Approaches for HPC Flow Simulations on Stream Processors

Dissertation

zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften (Dr. rer. nat.)

Dem Fachbereich IV der Universität Trier
vorgelegt von

Martin Siebenborn

Trier, im Oktober 2013

Gutachter: Prof. Dr. Volker Schulz
Prof. Dr. Nicolas Gauger

Zusammenfassung

Gegenstand dieser Arbeit ist die Untersuchung der industriellen Anwendbarkeit von Grafik- und Stream-Prozessoren auf dem Gebiet der Strömungssimulation. Zu diesem Zweck wird ein explizites Runge-Kutta diskontinuierliches Galerkin-Verfahren in beliebig hoher Ordnung komplett für die spezielle Architektur von GPUs implementiert. Parallel dazu wird eine Implementierung der selben Funktionalität für CPUs demonstriert und Vergleiche werden durchgeführt. Für die CPU werden sowohl explizite Verfahren als auch etablierte, implizite Verfahren betrachtet. Es werden dabei ebenfalls Code-Optimierungen für moderne Mehrkern-Prozessoren berücksichtigt, wie z.B. OPENMP Parallelisierung.

Der Fokus dieser Arbeit liegt dabei auf der Simulation von nicht-viskosen, transsonischen Strömungen über den ONERA M6 Testflügel. Für die Behandlung der daraus resultierenden Unstetigkeiten in der Lösung wird eine Methode basierend auf künstlicher Viskosität auf ihre Anwendbarkeit im Zeit-expliziten Verfahren untersucht. Besonders wird auf das Zusammenspiel mit der speziellen Hardware der GPU eingegangen. Da dieser Ansatz der künstlichen Viskosität sehr nah an der Simulation der Navier-Stokes Gleichungen ist, wird ebenfalls ein kurzer Exkurs zu diesem Thema gegeben, der den Einsatz von GPU-beschleunigten Verfahren für viskose Strömungen untersucht. Dabei baut diese Arbeit auf Untersuchen zum Einsatz von GPUs im nodalen, diskontinuierlichen Galerkin-Verfahren für lineare hyperbolische Probleme auf. Dieser Ansatz wird auf nicht-lineare Gleichungen erweitert, was den Einsatz von numerischer Quadratur verlangt. Darüber hinaus werden isoparametrisch gekrümmte Elemente zur exakten Darstellung komplexer Geometrien eingeführt. Ebenfalls werden Untersuchungen zur Implementierung der diskreten Adjungierten angestellt.

Verfahren hoher Ordnung weisen eine große Sensitivität bezüglich der Darstellung von Geometrien auf. Aus diesem Grund wird eine Methodik vorgestellt, die es ermöglicht, Objekte, deren Oberfläche durch Polynome höherer Ordnung dargestellt werden, in einem DG-Gitter aufzulösen. Dieser Ansatz beruht auf der Modellierung des Gitters als elastischer Festkörper mit Hilfe der Gleichungen der linearen Elastizität. Hierzu wird ein Kleinste-Quadrate Minimierungsproblem gelöst, um die Differenz zwischen geradlinigem Gitter und tatsächlicher Geometrie zu beschreiben. Dies wird dann als Randbedingung in der Gitterdeformation eingesetzt.

Die Sensitivität bezüglich der Darstellung der Geometrie wird am Ende der Arbeit nochmals aufgegriffen im Zusammenhang mit der Formoptimierung des ONERA M6 Flügels. Ziel dabei ist es, den Widerstand des Profils durch minimale Veränderungen der Form zu verkleinern. Hierzu wird demonstriert, wie der unstetige Shape Gradient innerhalb der Gitterdeformierung geglättet werden kann. Dabei wird ebenfalls ein Vergleich zur etablierten Laplace-Beltrami Glättung anhand der Stokes Gleichungen durchgeführt.

Acknowledgments

First of all, I would like to express my gratitude to Prof. Dr. Volker Schulz for the inspiring discussions and ongoing support. Not only the pleasant environment of his group at the University of Trier but also the chance to present my research on several international conferences was very helpful for me to finish this thesis.

I would also like to thank Prof. Dr. Nicolas Gauger for the attendance of being the second referee and for giving me many valuable advices for my research in the project meetings.

This work has been supported by the German Ministry for Education and Research (BMBF) within the collaborative project "DGHPOPT: Diskontinuierliche Galerkin-Verfahren für den robusten aerodynamischen Entwurf auf effizienten Rechnerarchitekturen in der Luftfahrt"[1].

Moreover, I would like to thank all my colleagues from the mathematical department of the University of Trier and especially the colleagues in the group of Prof. Schulz, namely Dr. Benjamin Rosenbaum, Christina Schenk, Dr. Claudia Schillings, Dr. Stephan Schmidt, Roland Stoffel, Dr. Christian Wagner and Heinz Zorn for creating such a nice work atmosphere, not only at the university but also on excursions and conferences. I would also like to thank Sonja Barth, Thomas Billen, Ulf Friedrich, David Schmitz, Marina Schneider, Dirk Thomas and Matthias Wagner for many interesting discussions during coffee breaks and for proofreading this thesis.

Finally, I would like to thank my family, especially my parents Jutta and Rainer, and my brother Bernhard, for supporting me in so many ways and encouraging me for this work.

Martin Siebenborn
Trier, 2013

[1] Discontinuous Galerkin Methods for Robust Aerodynamic Design on Efficient Hardware Architectures

Contents

Chapter 1

Introduction

1.1 Motivation

In computational fluid dynamics (CFD), which has shown to be one of the driving forces in numerical mathematics during the past few decades, there is a growing tendency towards higher order methods nowadays. While CFD in its beginning was dominated by first and second order accurate finite volume methods (FVM), the terminology "higher order" is commonly used for discretization methods which are of third or higher order accuracy. However, most of the production codes with industrial applications are still low order finite volume schemes. There is yet an increasing demand for higher order codes, visible in the popularity of discontinuous Galerkin (DG) methods, which are one famous representative of higher order methods. These methods were first proposed in the early 1970s for the neutron transport equation in [1]. Especially in the field of fluid dynamics the DG method promises to enable a new level of resolution of the underlying physical phenomena [2]. Direct numerical simulations (DNS) of turbulent flows, for instance, are often mentioned in this context.

One common misunderstanding connected to higher order methods is that they are computationally more expensive than classical finite volume methods. This might stem from the fact that the computational costs of discretizations are usually measured in the number of cells in the underlying mesh. Clearly, with respect to this measure the DG method with higher order polynomial reconstructions inside each cell is computationally more expensive than a finite volume discretization, which can be seen as a DG method with constant basis functions. However, due to the richer local ansatz spaces, a higher order DG discretization needs far less elements in the mesh to achieve the same resolution level. If the computational costs are thus measured in the number of degrees of freedom instead of the number of elements, higher order methods are not more expensive.

In [3] this is illustrated by an example. If the mesh size and time steps in a three dimensional first order method are halved, the computational effort increases by a factor of about 16. Thus, in order to reduce the discretization error by a factor of 4, the total computational costs are 256 times as high. In contrast, a second order accurate method shows only 16 times higher costs, which can be continued to higher order methods.

Beside the opportunities coming with higher order DG methods, there are still issues which are at least partially unsolved. In general, the numerical dissipation of higher order methods is lower than in FVM, which affects the robustness and the time to reach steady state solutions. The capturing of shocks, which likely arise in the solution of hyperbolic conservation laws, has thus to be treated more carefully in higher order methods. Thanks to the Godunov theorem it is known that higher order discretizations lead to problems in the vicinity of shocks. In finite

volume methods this issue is usually overcome by applying limiter methods to enforce stability despite shocks. However, the construction of limiters in higher order methods is challenging and computationally costly. Adding artificial viscosity to the hyperbolic equations, as investigated in [4], has thus proven to be an adequate instrument in order to deal with shocks in the solution, even if the shock is not aligned to element interfaces, but intersects the elements itself.

Another issue is connected directly to the high resolution of the spatial discretization, which makes the method very sensitive with respect to the representation of geometries. While in the low order FV method straight-sided grids are sufficient to resolve curved geometries, a higher order DG method is able to resolve the arising kinks in the boundary approximation. Thus, the geometry has to be resolved at least with polynomials of the same order as the DG scheme itself. This is problematic since higher order mesh generators do not yet show the necessary robustness for industrial applications. These two issues along with others make the DG method a vivid field of study.

Another feature and advantage of higher order DG methods is that they bring more structure into the problem. For a desired error tolerance it can be observed that increasing the polynomial degree and decreasing the number of elements leads to a natural clustering of the unknown variables. This is visible in the system matrix which shows a block-wise structure for higher order DG methods. Moreover, this feature suggests the application of stream processors, e.g. GPUs. In low order discretizations the unknowns are typically scattered throughout the memory which affects the speedup that can be gained by GPUs. However, the natural ordering of the degrees of freedom by the DG discretization allows GPUs to work in parallel on a very fine level, which enables considerable speedups against sequential algorithms. Moreover, as the terminology "discontinuous Galerkin" suggests, the loose coupling of elements through numerical flux functions increases the number of independent operations which also supports the parallelization on GPUs.

Especially the Runge-Kutta discontinuous Galerkin method (RKDG), as introduced in [5, 6, 7, 8, 9], is attractive for application on stream processors because of its explicit nature. First experiments combining DG and GPUs are shown in [10] for a nodal DG scheme, as introduced in [11], and the Maxwell equations. It is investigated how the performance gain of the GPU against conventional CPU codes increases with increasing polynomial order. One major aspect is that the memory alignment induced by the higher order, discontinuous basis functions leads to a memory layout that allows for 100% coalesced access patterns on the GPU. These results motivate the main topic of this work.

While there is much attention payed to the simulation of partial differential equations (PDE) in high performance computing (HPC), solving the corresponding adjoint problems is not that much in the focus. Especially in the field of PDE constraint optimal design and shape optimization, adjoints play a decisive role [12]. Adjoint methods open up the opportunity to compute sensitivities with respect to thousands of degrees of freedom, e.g. the triangulation of a shape, within the effort of only one PDE solution. Usually, these sensitivities are obtained in a discrete sense by applying automatic differentiation (AD) to the primal solver code. However, this is not yet practical for GPUs because most of the performance tuning is lost in the source code transformation of the AD tool.

In the field of shape optimization two major paradigms are followed. First, the shape can be seen as parametrized for instance by splines and the optimization is then conducted with respect

to the control points. This approach usually leads to a high level of robustness. However, the mapping of the boundary deformation into the volume discretization mesh also has to be derived, which can be complicated. Another approach is based on so-called shape calculus as presented in [13] which yields an analytical expression for the shape gradient and circumvents the problem of deriving a mesh deformation tool. By the Hadamard theorem the shape optimization problem can be restricted to gradient information which is based on the boundary of the shape only. Especially in higher order discretizations this is a promising approach, since it is independent of the discretization itself.

The analytical expression of the shape gradient in hand, the remaining question is how to deform the shape in order to obtain an iterative optimization strategy. Moreover, the discrete version of the shape gradient is not necessarily smooth such that it can be directly applied as a deformation to the discretized boundary. One common approach, which is followed for example in [14, 15], is to use a Laplace-Beltrami like operator to smooth the gradient information on the surface. After that a mesh deformation tool is applied in order to obtain the deformed volume discretization mesh. Here the sensitivity of higher order methods with respect to poorly resolved geometries comes full circle, which forces the mesh deformation to be payed more attention to. One major drawback of this approach is that there are parameters in the Laplace-Beltrami operator which have to be specified. However, these degrees of freedom depend on the mesh, which makes it a time consuming task to find appropriate smoothing parameters.

1.2 Aim and Scope of this Work

While both higher order discretization methods and graphics processing units are paid more and more attention it is a natural question to ask how these two fit together. Thus, the aim of this work is to investigate the potential of stream processors in the field of computational fluid dynamics. For this purpose, a GPU code for a Runge-Kutta discontinuous Galerkin discretization of the Euler and Navier-Stokes equations with arbitrary polynomial degrees is developed within this work.

A special focus is on comparisons to conventional algorithms which are optimized for x86 architectures using implicit time stepping methods. In particular, the speedup, which is gained by the application of different hardware architectures, should be measured against the additional programming effort. The underlying test case for the simulation is a transsonic flow over the ONERA M6 wing which requires also the investigation of shock capturing techniques within explicit time steppings. On the basis of the investigations of nodal discontinuous Galerkin methods together with an GPU implementation for Maxwell's equations [10], this work further shows a GPU approach of the RKDG method for non-linear PDEs of fluid dynamics and enables curved elements within the discretization. Also a new aspect is the treatment of the discrete adjoint approach on GPUs, which is investigated on basis of the GPU code developed within this work.

During the development of a GPU code for DG it turned out that the treatment of geometries is of great importance within higher order methods. Thus, it is also investigated how geometric information given by NURBS can be implemented into the GPU DG method. The final result of this work is the combination of these building blocks in the field of shape optimization. Here a drag reduction of the transsonic ONERA M6 test case is conducted using the higher order

mesh deformation tool for both the generation of new meshes and the smoothing of shape gradients.

1.3 Structure of this Work

This work has the following structure: Chapter 2 gives an overview on the mathematical background of the discontinuous Galerkin method. First, the spatial DG discretization is presented for the class of general hyperbolic conservation laws leading to a semidiscrete system of ODEs. In particular, the choice of a polynomial basis and quadrature rules is discussed. After that the equations of fluid dynamics, which are investigated within this work, are introduced and some simplifications are explained. This work concentrates on the pure hyperbolic case of the Euler equations but also one numerical test case is shown using the full set of Navier-Stokes equations.

Then two different time stepping approaches are presented. On the one hand, an explicit Runge-Kutta time stepping which will be implemented on the GPU and on the other hand a backward Euler method which will be implemented for the CPU as a reference. This chapter closes with a discussion of shock capturing techniques. In particular, it is shown how higher order derivatives for artificial viscosity and the Navier-Stokes equations come into the DG discretization.

Chapter 3 deals with the mesh generation for higher order DG methods. First, the problem of under-resolved geometries is demonstrated within a common test framework, which motivates the implementation of a higher order mesh generator for DG meshes. This mesh curvature approach is based on linear elasticity deformations. These differential equations are used in order to model the transport of curvature information given at the boundary into the discretization mesh. For this purpose, the shape is assumed to be described by NURBS and the straight-sided tetrahedra of the DG mesh are fitted to the spline representation by solving a non-linear least squares problem with a Gauss-Newton method. This solution of a closest point problem is then applied as a boundary condition in a finite element solver for the linear elasticity equations. The resulting solution, which is also computed in a higher order FE basis, is used as an isoparametric mapping for the DG elements.

After generating body fitted, curved grids with higher order boundary approximations, chapter 4 turns to the main topic of this work, namely the implementation of a DG code for GPUs. In the beginning of this chapter the GPU architecture and the CUDA programming model are presented and best practice implementation guidelines are reviewed. In particular, differences and similarities to conventional multicore CPUs are discussed. Then the parallelization of the Runge-Kutta discontinuous Galerkin method is investigated for the application on a cluster of multiple GPUs. This code is tested in some well-established fluid simulations with focus on challenging inviscid, transsonic flows and also viscous Navier-Stokes flows. This is followed by a performance analysis of the GPU code. For this purpose, a CPU code with the same functionality is implemented following the best practice performance guidelines for that architecture. Further, the explicit GPU code is compared to an implicit CPU code, which reflects state of the art solver techniques, where strengths and weaknesses of both approaches are pointed out.

In chapter 5 an explicit approach to compute the discrete adjoint of the Euler solver, presented in chapter 4, is discussed. Particularly, this chapter deals with an efficient implementation of

the discrete adjoint method on GPUs. It is further discussed why standard approaches like automatic differentiation are not applicable for GPUs. This chapter ends with a validation of the discrete adjoint algorithm. For this purpose, the sensitivity of the drag functional with respect to the angle of attack of the ONERA M6 wing is compared to finite difference approximations.

This work ends in chapter 6 with an outlook on shape optimization based on higher order discretizations. In this chapter all methods and techniques of the preceding work are applied in the field of shape optimization. First, the fundamentals of shape calculus and differential geometry are shortly reviewed. This is followed by the presentation of a novel shape optimization approach where the shape gradient is smoothed within the mesh deformation tool and not by a separate smoothing step. This approach is compared to the classical Laplace-Beltrami smoothing and advantages are pointed out in a simple shape optimization problem in a Stokes flow. After that these techniques are applied to the shape optimization of the ONERA M6 wing in an Euler flow which concludes this work.

In chapter 7 the essential facts and achievements of this work are shortly summarized and reviewed. After that an outlook is given on the topics which should be further investigated.

Chapter 2

Basic Concepts

In this chapter the discontinuous Galerkin method for hyperbolic problems is shortly introduced. Especially the choice of numerical fluxes, a polynomial basis and cubature rules is discussed. These building blocks are chosen such that they fit into the special programming model of stream processors. Then the equations of fluid dynamics, which are considered within this work, are shortly introduced. With the underlying PDEs in hand, the temporal discretization of the semidiscrete system, which is obtained from the DG scheme, is discussed. Explicit as well as implicit methods are presented and investigated with respect to GPU programming.

Since non-linear, hyperbolic PDEs likely have discontinuities in the solution, even with smooth initial conditions, it is necessary to deal with stabilization techniques. For the purpose of this work, a well-established artificial viscosity approach is reviewed at the end of this chapter.

2.1 General Discontinuous Galerkin Formulation

This works concentrates on the discontinuous Galerkin method for *hyperbolic conservation laws* of the following form

$$\frac{\partial U}{\partial t} + \frac{\partial F_1(U)}{\partial x_1} + \cdots + \frac{\partial F_d(U)}{\partial x_d} = 0 \qquad \text{in } (0,T] \times \Omega, \tag{2.1}$$

$$U(0,\boldsymbol{x}) = U_0(\boldsymbol{x}) \qquad \boldsymbol{x} \in \Omega. \tag{2.2}$$

Here $U : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^m$, $U(t,\boldsymbol{x}) = (U_1(t,\boldsymbol{x}), \ldots, U_m(t,\boldsymbol{x}))^T$ denotes the vector of m conserved variables at a point $\boldsymbol{x}$ in d-dimensional space and at time t and $U_0 : \mathbb{R}^d \to \mathbb{R}^m$ are initial conditions at time $t = 0$. The domain of interest is denoted by $\Omega \subset \mathbb{R}^d$ and could be arbitrarily shaped. Finally $[0,T]$ is the time horizon of the PDE. The vector fields $F_i : \mathbb{R}^m \to \mathbb{R}^m$, which are non-linear in general, are usually referred to as *flux vectors* and can be gathered introducing the tensor $F : \mathbb{R}^m \to \mathbb{R}^{m \times d}$, $F(U) = (F_1(U) \ldots F_d(U))$. In order to complete the system, spatial boundary conditions are needed, which will be discussed later since they depend on the physical background of the system (cf. section 2.2). The PDE can then be written in the widely used compact notation

$$\frac{\partial U}{\partial t} + \nabla \cdot F(U) = 0. \tag{2.3}$$

In the equation above the scalar product with the Nabla operator denotes the divergence of the tensor F. In many applications only the steady state solution is of interest which can be

obtained by solving system (2.3) in time until $\frac{\partial U}{\partial t}(t_\infty) \approx 0$ or by solving a discretized version of the system

$$\nabla \cdot F(U) = 0 \tag{2.4}$$

directly. In that case time does not have a physical meaning, but can be seen rather as a pseudo time.

The systems (2.1), (2.3) and (2.4) are said to be *hyperbolic* if the matrix

$$\mathcal{F}(U, W) = \sum_{i=1}^{d} W_i \frac{\partial F_i}{\partial U} \tag{2.5}$$

has m real eigenvalues and, moreover, a complete set of linearly independent eigenvectors for all $W = (W_1, \ldots, W_d)^T \in \mathbb{R}^d$.

The discontinuous Galerkin finite element discretization is obtained by deriving a weak formulation of (2.3) or (2.4), which is basically the same procedure as for continuous Galerkin methods. First, the domain Ω is subdivided into a finite set of K disjoint, conforming elements Ω_k with $\Omega = \bigcup_{k=1}^{K} \Omega_k$. Second, the solution U is approximated by U_h in a space $(\mathcal{V}_h)^m$ of element-wise defined polynomials $\psi_j : \Omega_k \to \mathbb{R}$ up to degree P. Thus, the global function space $\mathcal{V}_h$ is defined in the following way

$$\mathcal{V}_h = \bigoplus_{k=1}^{K} \mathcal{V}_h^k, \quad \mathcal{V}_h^k = \text{span}\{\psi_j : \Omega_k \to \mathbb{R},\, j = 1, \ldots, N_P\} \tag{2.6}$$

where $N_P = \frac{(P+d)!}{d!P!}$ is the number of polynomials needed to form a basis depending on the spatial dimension d and the degree P. The difference to continuous finite element methods is the double-valued state at the element interfaces, which will be addressed later in detail. Thus, the global function space $\mathcal{V}_h$ is often referred to as a *broken polynomial space*. Equation (2.3) is then multiplied by a local test function $\Phi \in (\mathcal{V}_h^k)^m$ and integrated over each element k leading to

$$\int_{\Omega_k} \left[\frac{\partial U_h}{\partial t} + \nabla \cdot F(U_h) \right] \Phi d\Omega = 0. \tag{2.7}$$

In order to obtain the weak discontinuous Galerkin formulation, integration by parts is applied to the flux term which yields

$$\int_{\Omega_k} \frac{\partial U_h}{\partial t} \Phi \, d\Omega = -\int_{\partial\Omega_k} \left(F(U_h^-, U_h^+)^* \cdot \vec{n} \right) \Phi \, dS + \int_{\Omega_k} F(U_h) \cdot \nabla\Phi d\Omega \,, \quad \forall 1 \le k \le K. \tag{2.8}$$

In the first integral on the right hand side in equation (2.8) $\partial\Omega_k$ denotes the surface of element number k and $\vec{n} : \partial\Omega_k \to \mathbb{R}^d$ the outward pointing normal vector field. $d\Omega$ indicates integration with respect to the three dimensional Lebesgue measure and dS two dimensional integration, respectively. This notation is chosen in order to clearly distinguish between volume and surface integrals. $F(U_h^-, U_h^+)^*$ is a so-called *numerical flux* function, which deals with the double-valued state at the element interfaces. These numerical fluxes are well known from finite

volume methods and will be discussed in section 2.2. Using the notation U_h^- and U_h^+, it is distinguished between the interior and exterior states at the interfaces of cell number k and its neighboring cells.

The next step is the representation of the function space $\mathcal{V}_h$. Here, it can be distinguished between two characteristic discontinuous Galerkin schemes, namely *modal* and *nodal* DG. In the first one the polynomials ψ_j in Ω_k are chosen to form an orthonormal basis with the following property

$$\int_{\Omega_k} \psi_i \psi_j d\Omega = \delta_{ij}\,, \quad \forall 1 \leq i,j \leq N_P \tag{2.9}$$

where δ_{ij} is the *Kronecker delta*. The advantage of this basis choice is that calculating inner products, as arising in the weak discontinuous Galerkin formulation (2.8), is straightforward as long as the integrands can be represented properly in the polynomial space. Moreover, this leads to diagonal mass matrices depending on the element shapes, which is attractive with respect to computational effort.
In nodal DG methods the basis of $\mathcal{V}_h$ is formed by Lagrange polynomials l_i. In order to fulfill the interpolation property, a set of N_P distinct interpolation points $\{\boldsymbol{x}_i \in \Omega_k,\ i = 1,\ldots,N_P\}$ is chosen with the property $l_i\left(\boldsymbol{x}_j\right) = \delta_{ij}$ for all $1 \leq i,j \leq N_P$. The connection between the two schemes can be established in the following sense. Let $f : \Omega_k \to \mathbb{R}$ be a polynomial with degree less or equal P and $\boldsymbol{x} \in \Omega_k$. Then $f(\boldsymbol{x})$ can be expressed in the two different bases as

$$f(\boldsymbol{x}) = \sum_{i=1}^{N_P} \hat{f}_i \psi_i(\boldsymbol{x}) = \sum_{i=1}^{N_P} f(\boldsymbol{x}_i)\, l_i(\boldsymbol{x})\,. \tag{2.10}$$

In the equation above the $\hat{\boldsymbol{f}} = \left(\hat{f}_1,\ldots,\hat{f}_{N_P}\right)^T$ are called the *modal expansion coefficients*, whereas the *nodal expansion coefficients* $\boldsymbol{f} = \left(f(\boldsymbol{x}_1),\ldots,f\left(\boldsymbol{x}_{N_p}\right)\right)^T$ are the function values themselves at the set of interpolation points $\boldsymbol{x}_i$. Transferring this to the unknown function U_h of the DG scheme means that the variables of the discretized equation (2.8) are either modal expansion coefficients or the function values of U_h.
Introducing a *Vandermonde matrix* of the form $V_{ij} = \psi_j(\boldsymbol{x}_i)$ for $1 \leq i,j \leq N_P$, these two formulations can be linked with the following relation

$$\boldsymbol{f} = V\hat{\boldsymbol{f}}. \tag{2.11}$$

Due to the interpolation property, V is regular. Moreover, the locality of the DG scheme, which stems from the discontinuous basis functions, leads to element-wise defined Vandermonde matrices, which are relatively small and dense operators. This feature allows to explicitly invert V and switch between modal and nodal representations efficiently. In this work features from both the nodal and the modal DG scheme will be relevant. Thus, it is obligatory to introduce an orthonormal set of polynomials as well as a set of interpolation points with special properties.

Since the aim of this work is the implementation of a 3D discontinuous Galerkin code on unstructured meshes, the dimension is fixed to $d = 3$ from now on. Furthermore, all Ω_k are chosen to be tetrahedra, curved or straight-sided. This choice is made because for domains Ω with complex shapes mesh generators offer more flexibility and robustness using simplices

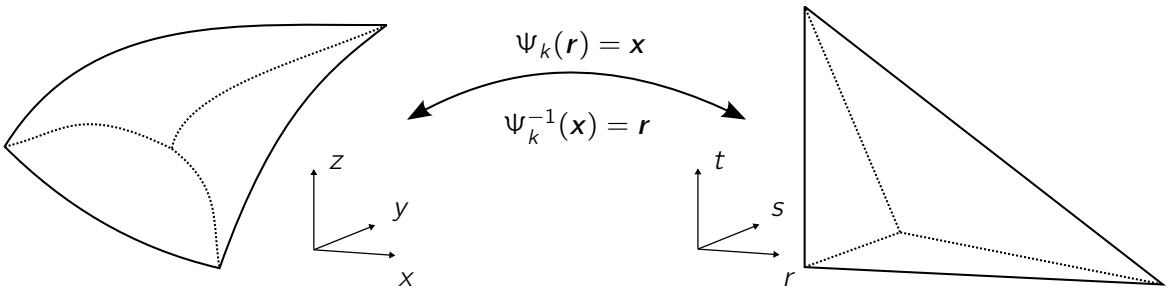


Figure 2.1: Mapping from reference element $\mathcal{T}$ to physical element Ω_k

as element type. It can now be proceeded as in standard FEM (finite element method) by introducing a *reference element*. The physical elements in the discretization mesh are then linked to the reference element by mapping functions. This concept offers the advantage that polynomial basis, interpolation points and quadrature nodes have to be defined only once and can be mapped into the individual elements.

For this purpose, it is useful to distinguish *computational coordinates* $\boldsymbol{r} = (r, s, t)^T$ in the reference element and *physical coordinates* $\boldsymbol{x} = (x, y, z)^T$ in the elements Ω_k. Assuming that the discretization of Ω is reasonable, bijective and continuously differentiable functions $\Psi_k : \mathcal{T} \to \Omega_k$ for all $1 \leq k \leq K$ can be introduced which establish a link to the reference tetrahedron

$$\mathcal{T} = \{-1 \leq r, s, t \leq 1\,,\; r + s + t \leq -1\}. \tag{2.12}$$

This is illustrated in figure 2.1. Together with the transformation formula, inner products of two functions $f, g : \Omega_k \to \mathbb{R}$ can then be traced back to integrations on the reference tetrahedron

$$(f, g)_{\Omega_k} = \int\limits_{\Omega_k = \Psi_k(\mathcal{T})} f g \; d\Omega = \int\limits_{\mathcal{T}} f\left(\Psi_k\right) g\left(\Psi_k\right) \left|\det\left(D\Psi_k\right)\right| d\Omega. \tag{2.13}$$

Here the Jacobi matrix of the mapping and its determinant often referred to as the *Jacobian* are given by

$$D\Psi = \begin{pmatrix} x_r & x_s & x_t \\ y_r & y_s & y_t \\ z_r & z_s & z_t \end{pmatrix} \quad \text{and} \quad J = \left|\det\left(D\Psi\right)\right|. \tag{2.14}$$

So far there are almost no restrictions on the mapping functions. However, it leads to interesting simplifications to represent them in the same polynomial basis as the discontinuous Galerkin scheme itself. This procedure is commonly referred to as *isoparametric mapping* and will be addressed later.

The next step is the generation of a polynomial basis on the reference element $\mathcal{T}$. This basis should fulfill two properties, which are important for the further results of this work. First, the basis polynomials should be orthonormal with respect to the inner product on the reference element (cf. equation (2.9)) and second, the basis should be hierarchical. The orthonormality

has great influence on the condition of the local mass and stiffness matrices and thereby affects the solvability of the global system (cf. [16]).

Further the polynomial basis is said to be *hierarchical* if the following condition holds. Let $\{\psi_1, \ldots, \psi_{N_{P_1}}\}$ and $\{\phi_1, \ldots, \phi_{N_{P_2}}\}$ be two bases on $\mathcal{T}$ for two polynomial degrees $P_1 < P_2$. Then it holds

$$\psi_i \in \{\phi_1, \ldots, \phi_{N_{P_2}}\} \quad \forall 1 \leq i \leq N_{P_1}. \tag{2.15}$$

This property will be used throughout this work for a P-enrichment procedure where calculations with higher polynomial degrees are preconditioned with lower ones in order to speed up convergence and save time. Another application is to measure oscillations in the solution by restricting it to the next lower polynomial degree and comparing these two functions. The process of restriction is straightforward for bases as introduced above.

While there is much literature dealing with a variety of polynomial bases in 1D, the extension to the multidimensional case is non-trivial. For elements like quadrilaterals in 2D or hexahedra in 3D the so-called *tensor product approach* could be used where the multivariate polynomials are formed as products of the 1D basis polynomials. For the tetrahedron the construction of a hierarchical, orthonormal polynomial basis is demonstrated in [17]. Further discussions and the proof of orthogonality can be found in [16]. At first, an orthonormal basis is set up for the hexahedron $[-1, 1]^3$ by forming a tensor product of 1D Jacobi polynomials. The hexahedron is then collapsed to the standard tetrahedron $\mathcal{T}$. This link is established by the following coordinate transformation

$$\mathcal{T} \to [-1, 1]^3,\ (\xi_1, \xi_2, \xi_3)^T \mapsto \left(\frac{2(1+\xi_1)}{-\xi_2 - \xi_3} - 1, \frac{2(1+\xi_2)}{-1-\xi_3} - 1, \xi_3\right)^T = (\eta_1, \eta_2, \eta_3)^T. \tag{2.16}$$

A basis of polynomials up to order P is finally formed by

$$\psi_{ijk}(\xi_1, \xi_2, \xi_3) = 2\sqrt{2}\Phi_i^{(0,0)}(\eta_1)(1-\eta_2)^i \Phi_j^{(2i+1,0)}(\eta_2)(1-\eta_3)^{i+j}\Phi_k^{(2i+2j+2,0)}(\eta_3) \tag{2.17}$$

for $0 \leq i, j, k \leq P$ and $i + j + k \leq P$. In the equation above Φ denotes the classical Jacobi polynomial which can be efficiently calculated by the recurrence relation

$$\xi\Phi_n^{(\alpha,\beta)}(\xi) = a_n\Phi_{n-1}^{(\alpha,\beta)}(\xi)b_n\Phi_n^{(\alpha,\beta)}(\xi) + a_{n+1}\Phi_{n+1}^{(\alpha,\beta)}(\xi) \tag{2.18}$$

for $\alpha, \beta > -1$ and $\xi \in [-1, 1]$. The coefficients a and b are given by

$$a_n = \frac{2}{2n+\alpha+\beta}\sqrt{\frac{n(n+\alpha+\beta)(n+\alpha)(n+\beta)}{(2n+\alpha+\beta-1)(2n+\alpha+\beta-1)}} \tag{2.19}$$

and

$$b_n = -\frac{\alpha^2 - \beta^2}{(2n+\alpha+\beta)(2n+\alpha+\beta+2)}. \tag{2.20}$$

In order to start the recurrence, the two first Jacobi polynomials are set as initial values

$$\Phi_0^{(\alpha,\beta)}(\xi) = \sqrt{2^{-\alpha-\beta-1}\frac{\Gamma(\alpha+\beta+2)}{\Gamma(\alpha+1)\Gamma(\beta+1)}}, \tag{2.21}$$

$$\Phi_1^{(\alpha,\beta)}(\xi) = \frac{1}{2}\Phi_0^{(\alpha,\beta)}(\xi)\sqrt{\frac{\alpha+\beta+3}{(\alpha+1)(\beta+1)}}((\alpha+\beta+2)\xi + \alpha - \beta). \tag{2.22}$$

A feature of the Jacobi polynomials is that their derivatives can also be represented in a recurrence

$$\frac{d}{d\xi}\Phi_n^{(\alpha,\beta)}(\xi) = \sqrt{n(n+\alpha+\beta+1)}\Phi_{n-1}^{(\alpha+1,\beta+1)}(\xi) \tag{2.23}$$

and thus computed efficiently. Further details on these polynomials can be found in [18].

The next step is to choose a set of interpolation points in $\mathcal{T}$ in order to define the basis of Lagrange polynomials. In literature there are several different approaches for constructing a set of N_P interpolation points for the tetrahedron. Further details can be found in [17, 19, 20]. Each set of N_P distinct points would in principle lead to a uniquely defined basis of Lagrange polynomials. However, the distribution has great influence on oscillations of the basis functions. One major property and also a measure for the quality of interpolation points is the so-called *Lebesgue constant* which is given by

$$\Lambda = \max_{\boldsymbol{r}\in\mathcal{T}} \sum_{i=1}^{N_P} |l_i(\boldsymbol{r})| \tag{2.24}$$

where l_i are Lagrange basis polynomials uniquely defined on a set of $\{\boldsymbol{r}_i \in \mathcal{T},\, 1 \leq i \leq N_P\}$ distinct points. Let $f : \mathcal{T} \to \mathbb{R}$ be a function on the reference element, f_h the interpolation with Lagrange polynomials and f^* the best approximating polynomial of degree P. Then the following estimation holds

$$\|f - f_h\|_\infty = \|f - f^* + f^* - f_h\|_\infty \leq \|f - f^*\|_\infty + \|f^* - f_h\|_\infty \leq (1+\Lambda)\,\|f - f^*\|_\infty\,. \tag{2.25}$$

This inequality illustrates the impact of the interpolation points on the approximation property of the polynomial basis. f is interpolated by f_h at the points $\boldsymbol{r}_i$, but what happens in between depends on how oscillatory the l_i are. Moreover, minimizing the Lebesgue constant is closely related to maximizing the determinant of the Vandermonde matrix V in equation (2.11) as described in [21]. This determinant plays a decisive role since explicit inversion of V by LU decomposition is an essential part of the DG method considered here.

The importance of the interpolation points should however not be overemphasized as the typical polynomial degree in this work will be $P = 1, 2, 3, 4, 5$. For this degrees the optimized points mentioned above show only a small advantage over an equidistant set of interpolation points for both the Lebesgue constant and the determinant of V. However, for degrees of 10 and greater it is demonstrated in [21] that V gets close to a singularity. For the purpose of this work, the approach in [17] is used since it is connected to an explicit procedure to generate point sets for arbitrary polynomial degrees. With the polynomial approximation of the PDE solution in hand, it remains an open question how to approximate the integrals arising in equation (2.8). The so-called nodal discontinuous Galerkin approach introduced in [21, 11] for the Maxwell equations on unstructured, tetrahedral grids takes advantage of the orthogonality of the basis functions and thus leads to a quadrature free calculation of the arising integrals. However, this simplification only works for PDEs with polynomial flux functions and on straight-sided elements which means that the transformation function (2.13) between the reference element and the physical element is an affine mapping. In this case each point $\boldsymbol{x} \in \Omega_k$ can be mapped to an $\boldsymbol{r} \in \mathcal{T}$ by

$$\Psi_k(\boldsymbol{x}) = A_k\boldsymbol{x} + \boldsymbol{b}_k = \boldsymbol{r},\; A_k \in \mathbb{R}^{3\times 3},\; \boldsymbol{b}_k \in \mathbb{R}^3. \tag{2.26}$$

This is an interesting simplification since the Jacobian $J_k = \det(A_k)$ of the transformation Ψ_k is element-wise constant and does not influence the integration. If the physical element is not of this form, in particular in case of an isoparametric mapping where the transformation is also a polynomial of degree P, the integration has to be dealt with carefully. The determinant of the Jacobi matrix of the mapping is then not in the range of the basis polynomials anymore.

Moreover, the flux function F in equation (2.3) is most often not a polynomial. In particular, the fluxes of the Euler equations, which are the subject of this work, are rational functions. Thus, the integration of these parts has to be of higher order to at least minimize the additional error.

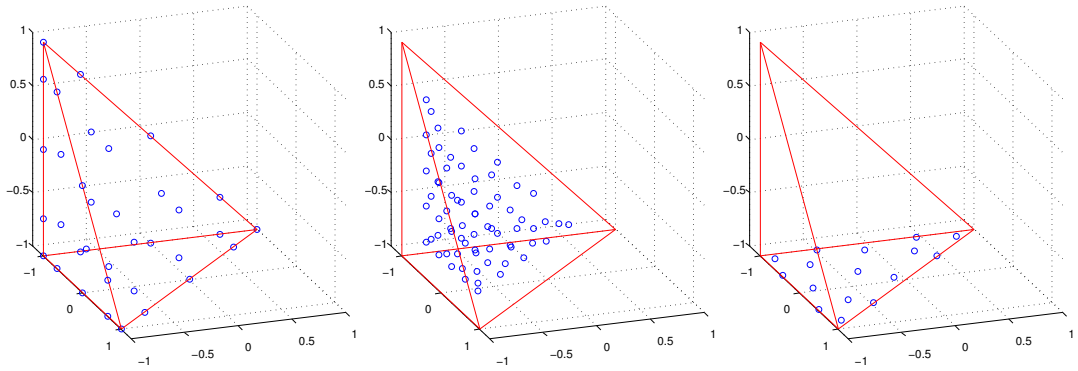
The application of *cubature rules* (multidimensional quadrature) overcomes this issue since the accuracy of integral approximations can be controlled by the number of evaluation nodes. In general, cubature rules are specific for one element type (here the tetrahedron $\mathcal{T}$) and are defined by a set of N_V cubature nodes $\{\boldsymbol{r}_i^\mathrm{V} \in \mathcal{T}, 1 \leq i \leq N_\mathrm{V}\}$ where the integrand is evaluated and a set of weights $\{w_i^\mathrm{V} \in \mathcal{T}, 1 \leq i \leq N_\mathrm{V}\}$. These rules are usually characterized by the maximum degree of polynomials they are able to integrate exactly. The integral over $f : \mathcal{T} \to \mathbb{R}$ is then approximated by a sum of the following form

$$\int_{\mathcal{T}} f \, d\Omega \approx \sum_{i=1}^{N_\mathrm{V}} f(\boldsymbol{r}_i^\mathrm{V}) w_i^\mathrm{V}. \tag{2.27}$$

Multidimensional cubature is different to one dimensional quadrature since there is no generalization of the one dimensional Gaussian quadrature known. In one dimension the interval is the only connected and compact subset, whereas in two and more dimensions there is an infinite variety of such shapes. This explains why cubature rules are specific for one element type. Moreover, in one dimension Gaussian quadrature is known to be optimal in the sense that a Gaussian rule with n points integrates polynomials of degree up to $2n-1$ exactly, whereas no n-point rule is exact for polynomials of degree $2n$. In multiple dimensions only upper and lower bounds are known. An upper bound for the number of cubature nodes needed can be found by considering tensor-product rules on structured elements. These are obtained by forming products of the one dimensional Gaussian rules for the multidimensional unit cube. A lower bound for cubature on simplices can be found in [22].

In order to distinguish between the three dimensional integration inside the tetrahedron and the cubature rule for its surface, nodes and weights will be denoted by "V" and "S", respectively. A survey of the cubature theory and analysis is given in [22] and an extensive list of cubature rules is given in [23]. In order to derive a discrete approximation to the weak DG formulation in equation (2.8), two different integrations have to be performed. On the one hand, two three dimensional volume integrals have to be computed, which are cubatures over a tetrahedron in this particular framework. For this purpose, the rules described in [24] are applied. These cubature rules are found by combinatorial methods and are invariant under all affine transformations of the tetrahedron onto itself.

On the other hand, the numerical flux function has to be integrated on the two dimensional surface of the elements. For tetrahedral elements this means a cubature over the four triangles forming its surface. For further considerations it is obligatory that the triangle cubature nodes are symmetric since the computation of the numerical flux between two adjacent tetrahedral



(a) interpolation points (b) volume cubature points (c) surface cubature points

Figure 2.2: Different set of nodes inside the reference tetrahedron $\mathcal{T}$

elements requires that the local U_h^- and remote state U_h^+ of the unknown function coincide at the cubature points. Ensuring this symmetry, it only remains to check in which orientation two neighboring elements are connected in order to define the assignment between local and remote variables. As it is necessary for the flux integration first to interpolate the unknown variables represented in the Lagrange basis to the surface cubature nodes, interpolation operators are needed. Here, the usage of a symmetrical cubature rule allows to apply the same interpolation for each element. This will be explained in more detail later in this section.

For the purpose of this work, the symmetric cubature rules for triangles described in [25] are used. These rules are found utilizing methods from group theory and numerical optimization. Nodes and weights are denoted by $\{\boldsymbol{r}_i^S \in \mathcal{T}, 1 \leq i \leq 4N_S\}$ and $\{w_i^S \in \mathcal{T}, 1 \leq i \leq 4N_S\}$. The total number of cubature nodes for the surface is $4N_S$ since the nodes and weights for all four surface triangles are collected in one vector.

Figure 2.2 visualizes the locations of interpolation, volume cubature and surface cubature points inside the reference tetrahedron. Later in chapter 4 it will be discussed why it is important to distinguish between these three sets of nodes with respect to computational efficiency on the GPU.

The next step is to derive a discrete version of equation (2.8). In this section the right hand side is discretized according to the discontinuous Galerkin scheme introduced so far. The discretization of the time derivative will be addressed later in detail. Note that the local, discrete operators are derived for a single PDE and not a system of equations, e.g. $m = 5$ in the Euler case. If there are more than one equation, then it is assumed that the discrete operators are extended and applied component-wise.

As mentioned earlier, it has to be decided how the unknown variable U is approximated by the broken polynomial space $(\mathcal{V}_h)^m$ defined in (2.6). Here, a vector $U_k \in \mathbb{R}^{m \cdot N_P}$ is chosen to state the function values at the interpolation points $\{\boldsymbol{r}_i \in \mathcal{T},\ 1 \leq i \leq N_P\}$ in element number k. The variables U_k can thus be seen as function values and as expansion coefficients in the basis of Lagrange polynomials.

Since the Lagrange basis is not available in an explicit form but only defined over the orthonor-

mal polynomials and relation (2.11), it will be an essential part to switch between these two basis representations. Due to the broken polynomial space the Vandermonde matrix $V \in \mathbb{R}^{N_P \times N_P}$ is a relatively small, dense operator. It is thus possible to explicitly invert V by LU decomposition in an initial stage of the solver. The considerations concerning the interpolation points ensure that V is regular. In order to interpolate U_h to the cubature points, the orthonormal polynomial basis is evaluated in the following way

$$V_{ij}^V = \psi_j(\boldsymbol{r}_i^V),\ 1 \leq i \leq N_V,\ 1 \leq j \leq N_P. \tag{2.28}$$

Together with V^{-1} the volume cubature interpolation operator is obtained by

$$\mathcal{I}_V = V^V V^{-1} \in \mathbb{R}^{N_V \times N_P}. \tag{2.29}$$

The same is done for the surface cubature points to obtain

$$V_{ij}^S = \psi_j(\boldsymbol{r}_i^S),\ 1 \leq i \leq 4N_S,\ 1 \leq j \leq N_P \tag{2.30}$$

and

$$\mathcal{I}_S = V^S V^{-1} \in \mathbb{R}^{4N_S \times N_P}. \tag{2.31}$$

With this interpolation operator the left hand side of equation (2.8) can be discretized leading to an element specific local mass matrix

$$M = \mathcal{I}_V^T \cdot \mathrm{diag}(J_i^V w_i^V) \cdot \mathcal{I}_V \in \mathbb{R}^{N_P \times N_P} \tag{2.32}$$

where $\mathrm{diag}(J_i^V w_i^V)$ is the matrix with entries $(J_i^V w_i^V)_i$ on the diagonal and zero otherwise. To be precise M and J_i^V should be indexed with the element number k yet this is left out for clarity.

Since the transformation mapping is assumed to be non-linear and in particular not in the range of the basis polynomials, the Jacobian J varies between the cubature nodes. The superscript V indicates that the Jacobian evaluated at the cubature nodes is meant. In the case of a straight-sided element, J is constant. As a consequence all non-curved elements share the same local mass matrix up to a constant factor. This is an interesting fact with respect to memory usage of the solver.

For the construction of the local stiffness matrices the partial derivatives of the polynomial basis are needed. This is done by differentiating equation (2.17) and evaluating it at the volume cubature points

$$(V_r)_{ij} = \left.\frac{\partial \psi_j(\boldsymbol{r})}{\partial r}\right|_{\boldsymbol{r}=\boldsymbol{r}_i^V},\ (V_s)_{ij} = \left.\frac{\partial \psi_j(\boldsymbol{r})}{\partial s}\right|_{\boldsymbol{r}=\boldsymbol{r}_i^V},\ (V_t)_{ij} = \left.\frac{\partial \psi_j(\boldsymbol{r})}{\partial t}\right|_{\boldsymbol{r}=\boldsymbol{r}_i^V} \tag{2.33}$$

for $1 \leq i \leq N_V$ and $1 \leq j \leq N_P$. Multiplying the Vandermonde matrices of the derivatives with V^{-1} leads to differentiation operators for the reference element with respect to computational coordinates

$$D_r = V_r V^{-1},\ D_s = V_s V^{-1},\ D_t = V_t V^{-1} \in \mathbb{R}^{N_V \times N_P}. \tag{2.34}$$

In order to obtain the differentiation operators for the physical elements, the composition with the element transformation mapping has to be considered. This is then derived with the chain rule yielding

$$\begin{aligned} S_x &= M^{-1}\left(D_r^T \operatorname{diag}(r_{x,i}^V) + D_s^T \operatorname{diag}(s_{x,i}^V) + D_t^T \operatorname{diag}(t_{x,i}^V)\right) \operatorname{diag}(J_i^V w_i^V) \in \mathbb{R}^{N_P \times N_V} \\ S_y &= M^{-1}\left(D_r^T \operatorname{diag}(r_{y,i}^V) + D_s^T \operatorname{diag}(s_{y,i}^V) + D_t^T \operatorname{diag}(t_{y,i}^V)\right) \operatorname{diag}(J_i^V w_i^V) \in \mathbb{R}^{N_P \times N_V} \\ S_z &= M^{-1}\left(D_r^T \operatorname{diag}(r_{z,i}^V) + D_s^T \operatorname{diag}(s_{z,i}^V) + D_t^T \operatorname{diag}(t_{z,i}^V)\right) \operatorname{diag}(J_i^V w_i^V) \in \mathbb{R}^{N_P \times N_V}. \end{aligned} \tag{2.35}$$

Note that these three operators are multiplied with the inverse element mass matrix from the left in order to separate the time derivative on the left hand side of equation (2.8). Finally, the local mass matrices for the surface integration are given by

$$M_{\partial\Omega} = M^{-1} \cdot \mathcal{I}_S^T \cdot \operatorname{diag}(J_i^S w_i^S) \in \mathbb{R}^{N_P \times 4N_S} \tag{2.36}$$

and again the multiplication with the inverse mass matrix is already performed. Here the superscript S indicates that the Jacobian is evaluated at the surface cubature nodes.

Up till now the Jacobian and with it the representation of the element curvature is not specified yet. As mentioned earlier, choosing the elemental transformation mapping to be representable within the same Lagrangian basis as the DG scheme itself offers a high amount of flexibility. For the isoparametric mapping it is assumed that the transformation for each element Ω_k is given pointwise for the set of interpolation nodes. In section 3.2 an approach is presented how this could be achieved. The inverse of the actual polynomial mapping Ψ_k (cf. figure 2.1) is defined by the assignment of interpolation points in the reference element and in the k-th physical element

$$\{\boldsymbol{r}_i \in \mathcal{T},\, 1 \leq i \leq N_P\} \mapsto \{\boldsymbol{x}_i \in \Omega_k,\, 1 \leq i \leq N_P\}. \tag{2.37}$$

Let $X, Y, Z \in \mathbb{R}^{N_P}$ be the vectors of the components of the nodes $\boldsymbol{x}_i$. Then the Jacobi matrix of the inverse transformation evaluated at interpolation point number i is given by

$$(D\Psi)_i = \begin{pmatrix} \tilde{D}_{r,i}X & \tilde{D}_{s,i}X & \tilde{D}_{t,i}X \\ \tilde{D}_{r,i}Y & \tilde{D}_{s,i}Y & \tilde{D}_{t,i}Y \\ \tilde{D}_{r,i}Z & \tilde{D}_{s,i}Z & \tilde{D}_{t,i}Z \end{pmatrix} = \begin{pmatrix} x_{r,i} & x_{s,i} & x_{t,i} \\ y_{r,i} & y_{s,i} & y_{t,i} \\ z_{r,i} & z_{s,i} & z_{t,i} \end{pmatrix}. \tag{2.38}$$

In the equation above $\tilde{D}_{\cdot,i}$ denotes the i-th row of the differentiation operator which evaluates derivatives at interpolation points. $\tilde{D}_r, \tilde{D}_s, \tilde{D}_t$ are obtained in the same way as D_r, D_s, D_t with the modification that in equations (2.33) and (2.34) derivatives are evaluated at interpolation points. The partial derivatives given in (2.38) can be interpolated to the volume and surface cubature nodes using the operators $\mathcal{I}_V$ and $\mathcal{I}_S$.

After this interpolation the Jacobi matrices are inverted in order to retrieve the partial derivatives of the inverse transformation mappings Ψ^{-1} needed in (2.35). Furthermore, the Jacobians J_i^V and J_i^S at the volume and surface cubature nodes are then calculated. Note that the order of interpolation and forming the determinant of the Jacobi matrices is not interchangeable since the Jacobian is not in the range of the Lagrange polynomials in general. Plugging the local operators introduced in this section into (2.8) finally leads to the semi discrete version of

the weak discontinuous Galerkin formulation

$$\frac{\partial U_k}{\partial t} = \sum_{m=1}^{3} S_{k,x_m} F_m \left(\mathcal{I}_V U_k\right) - M_{\partial\Omega_k} H(U_k^-, U_k^+, \vec{n}). \tag{2.39}$$

Here $H(U_k^-, U_k^+, \vec{n})$ is the implementation of a numerical flux function. With H, the flux between cell number k and its adjacent cells is approximated. This flux depends on the local variables interpolated to the surface cubature nodes $U_k^- = \mathcal{I}_S U_k$ and the remote variables from the adjacent tetrahedra U_k^+ coinciding with the local ones. Note that the symmetry of the surface cubature rule ensures that each component of U_k^- can be identified with one of U_k^+ resulting in a permutation of the remote variables before the flux is evaluated.

The convergence analysis of the spatial DG scheme, as presented within this chapter, was first conducted for linear PDEs in two dimensions. For the special case of the neutron transport equation

$$\mu \cdot \nabla u + \eta u = S \tag{2.40}$$

as introduced in [1] with $S, u : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$, $\mu \in \mathbb{R}^2$ and $\eta \in \mathbb{R}$ in [26] a convergence order of $\mathcal{O}(h^{P+1})$ was shown where P is the polynomial degree and h is the width of the elements. This analysis was conducted on a rectangular grid with a tensor-product basis leading to an estimate of the L^2 error

$$\|u - u_h\|_{L^2(\Omega)} \leq c_1 h^{P+1} \|u\|_{H^{P+2}(\Omega)} \tag{2.41}$$

assuming enough regularity in the solution u. Here the $H^n(\Omega)$-norm is given by

$$\|u\|_{H^n(\Omega)} = \sqrt{\sum_{|\alpha| \leq n} \left\| \frac{\partial^\alpha u}{\partial x^\alpha} \right\|_{L^2(\Omega)}^2} \tag{2.42}$$

in terms of a multi-index α. This result was extended in [27] to quasi-uniform triangulations, which means that the angles within the triangles are bounded, obtaining an order of convergence of $\mathcal{O}(h^{P+\frac{1}{2}})$ in the sense

$$\|u - u_h\|_{L^2(\Omega)} \leq c_2 h^{P+\frac{1}{2}} \|u\|_{H^{P+1}(\Omega)}. \tag{2.43}$$

The convergence order of $\mathcal{O}(h^{P+1})$ could be validated later in [28] again for quasi-uniform triangulations with the additional condition that element interfaces are not aligned with the characteristic direction, i.e. $|\langle \mu, \vec{n} \rangle| > 0$.

In the more recent article [29] the DG method is shown to have optimal convergence rates of $\mathcal{O}(h^{P+1})$ for general non-linear, scalar conservation laws in multiple dimensions assuming the use of upwind fluxes. In the case of non-smooth solutions, which may arise in general conservation laws, the concept of higher order accuracy conflicts with the Godunov theorem which will be discussed in section 2.4. This theorem states that a numerical method has to be first order in the vicinity of discontinuities in the solution in order to preserve monotonicity. Nevertheless, this case is further investigated in [7] and [8].

2.2 Equations of Fluid Dynamics

In this section some representative equations of fluid dynamics are briefly introduced, namely the Navier-Stokes equations and, as a special case, the Euler equations. Both of them are investigated numerically within this work. In order not to go too deeply into the physical meaning of these equations, a more mathematical point of view is chosen here. A detailed survey of the derivation of the conservation laws can be found for instance in [30] or [31].

In this work a simplified version of the Navier-Stokes equations is considered, leaving out the effect of heat diffusion and assuming a homogeneous viscosity, which does not depend on density. Also external forces, e.g. gravity, are neglected. This system of equations describes the motion of a compressible fluid in the three dimensional space and states the conservation of mass, momentum and energy. The fluid is characterized by the quantities density ρ, the vector of three velocities in terms of the Cartesian coordinates $\boldsymbol{u} = (u, v, w)$ and the total energy E. These are the so-called *primitive variables*, whereas the differential equations are formulated in terms of *conservative variables*. In the terminology of section 2.1 five conservative variables are distinguished such that $m = 5$ and $d = 3$ leading to the unknown function

$$U : (0, T] \times \Omega \to \mathbb{R}^5, \quad (t, \boldsymbol{x}) \mapsto (\rho, \rho u, \rho v, \rho w, \rho E). \tag{2.44}$$

Here a more general formulation of equation (2.1) is chosen with additional viscous fluxes

$$\frac{\partial U}{\partial t} + \nabla \cdot F^C(U) - \nabla \cdot F^D(U, \nabla U) = 0. \tag{2.45}$$

Yet, section 2.4 will show that the building elements for a discrete version are already derived. The superscripts C and D distinguish between fluxes describing the convective and viscous part of the PDE. This system is commonly referred to as a *convection-diffusion equation*. In the case of the Navier-Stokes equation as described above the *convective fluxes* are given by

$$F^C(U) = \begin{pmatrix} \rho u & \rho v & \rho w \\ \rho u^2 + p & \rho uv & \rho uw \\ \rho uv & \rho v^2 + p & \rho vw \\ \rho uw & \rho vw & \rho w^2 + p \\ u(\rho E + p) & v(\rho E + p) & w(\rho E + p) \end{pmatrix} \tag{2.46}$$

and the *diffusive fluxes*

$$F_j^D(U, \nabla U) = \begin{pmatrix} 0 \\ \tau_{1j} \\ \tau_{2j} \\ \tau_{3j} \\ u\tau_{j1} + v\tau_{j2} + w\tau_{j3} \end{pmatrix}, \quad j = 1, 2, 3. \tag{2.47}$$

These fluxes depend on the *stress tensor*

$$\tau = \mu \left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T - \frac{2}{3}(\nabla \cdot \boldsymbol{u}) I \right) \tag{2.48}$$

where $\nabla \boldsymbol{u}$ denotes the Jacobi matrix of the velocity field, $\nabla \cdot \boldsymbol{u}$ the divergence, I the 3×3 identity matrix and μ the viscosity of the fluid. If neglecting the diffusive fluxes or choosing the viscosity $\mu = 0$, the pure hyperbolic Euler equations are obtained which are the main component of this work. System (2.45) with the convective (2.46) and diffusive (2.47) fluxes is still underdetermined since there are six unknowns in five equations. Thus, some assumptions on the fluid have to be made for which the perfect gas assumption is chosen here linking the pressure to the conservative variables

$$p = (\gamma - 1)\left(\rho E - \frac{u^2 + v^2 + w^2}{2\rho}\right). \tag{2.49}$$

This assumption is reasonable for dry air at normal pressure and temperature and in that case the *adiabatic index* is given by $\gamma \approx 1.4$. In the case of a viscous flow the dimensionless *Reynolds number* can be used in order to characterize different situations. It is given by

$$\mathrm{Re} = \frac{\rho u L}{\mu} \tag{2.50}$$

where u denotes the mean velocity of the obstacle relative to the fluid, μ the viscosity and L a characteristic length which can be, e.g., the diameter of the obstacle. Roughly speaking, the Reynolds number describes the ratio of inertial and viscous forces in the fluid. The lower the Reynolds number the more viscous is the fluid. More details can be found in [30].

In section 2.1 the actual choice of spatial boundary conditions was left open since they depend on the physical equations. The test cases in this work depend on two different types of boundary conditions: on the one hand, a so-called *farfield condition* which models the fluid entering or leaving the domain and on the other hand, boundary conditions describing the interaction of the fluid with geometry, e.g. an airfoil. In the discontinuous Galerkin framework boundary conditions are commonly imposed in a weak sense. This means, if a discretization element is at a physical boundary and hence one face is not connected to a neighboring cell, the remote state U_h^+ is set to a value representing the boundary condition. For this purpose, the Jacobi matrices of the convection fluxes have to be introduced. These are given by

$$\frac{\partial F_1^C}{\partial U} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ (\gamma-1)H - u^2 - a^2 & (3-\gamma)u & -(\gamma-1)v & -(\gamma-1)w & \gamma-1 \\ -uv & v & u & 0 & 0 \\ -uw & w & 0 & u & 0 \\ u((\gamma-2)H - a^2) & H - (\gamma-1)u^2 & -(\gamma-1)uv & -(\gamma-1)uw & \gamma u \end{pmatrix}$$

$$\frac{\partial F_2^C}{\partial U} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ -vu & v & u & 0 & 0 \\ (\gamma-1)H - v^2 - a^2 & -(\gamma-1)u & (3-\gamma)v & -(\gamma-1)w & \gamma-1 \\ -vw & 0 & w & v & 0 \\ v((\gamma-2)H - a^2) & -(\gamma-1)uv & H - (\gamma-1)v^2 & -(\gamma-1)vw & \gamma v \end{pmatrix}$$

$$\frac{\partial F_3^C}{\partial U} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ -uw & w & 0 & u & 0 \\ -vw & 0 & w & v & 0 \\ (\gamma-1)H - w^2 - a^2 & -(\gamma-1)u & -(\gamma-1)v & -(\gamma-1)w & \gamma-1 \\ w((\gamma-2)H - a^2) & -(\gamma-1)uw & -(\gamma-1)vw & H - (\gamma-1)w^2 & \gamma w \end{pmatrix} \tag{2.51}$$

in terms of the *total enthalpy*

$$H = \frac{\rho E + p}{\rho} \tag{2.52}$$

and the *local speed of sound*

$$a = \sqrt{\frac{\gamma p}{\rho}} = \sqrt{(\gamma - 1)\left[H - \frac{1}{2}(u^2 + v^2 + w^2)\right]}. \tag{2.53}$$

Let $\vec{n}$ be the outward pointing normal vector at the farfield boundary of the flow domain and

$$F_{\vec{n}}^{C\prime} = \sum_{i=1}^{3} n_{x_i} \frac{\partial F_i^C}{\partial U} \tag{2.54}$$

the normal convective flux Jacobi matrix. Then four different flow situations can be distinguished depending on the eigenvalues of $F_{\vec{n}}^{C\prime}$ which are given by

$$\lambda_1 = \langle \boldsymbol{u}, \vec{n} \rangle - a,\ \lambda_2 = \lambda_3 = \lambda_4 = \langle \boldsymbol{u}, \vec{n} \rangle,\ \lambda_5 = \langle \boldsymbol{u}, \vec{n} \rangle + a, \tag{2.55}$$

namely

- *supersonic inflow* if $\lambda_i < 0,\ i = 1, \ldots 5$
- *subsonic inflow* if $\lambda_i < 0,\ i = 1, \ldots 4,\ \lambda_5 > 0$
- *subsonic outflow* if $\lambda_i > 0,\ i = 2, \ldots 5,\ \lambda_1 < 0$
- *supersonic outflow* if $\lambda_i > 0,\ i = 1, \ldots 5.$

In the case of a supersonic inflow, information is only entering the domain, thus U_h^+ is set to the freestream values $(\rho_\infty, \rho u_\infty, \rho v_\infty, \rho w_\infty, \rho E_\infty)$. Here freestream denotes the state of the fluid which is assumed to be in sufficiently large distance to the airfoil. At a supersonic outflow boundary information is only leaving the domain, therefore $U_h^+ = U_h^-$ is assumed.

For subsonic inflow conditions the first four conservative variables are taken from the freestream state and the pressure is taken from inside the domain which determines the fifth conservative variable. At a subsonic outflow boundary this situation reverses and four conservative variables are taken from inside and the fifth is determined by the pressure calculated with the freestream values according to (2.49).

The second boundary condition is the one on the obstacle in the flow field - the so-called *wall boundary*. It is necessary to distinguish between inviscid and viscous flows. In the first case, when the fluid is modeled by the Euler equations, there are no friction forces. This allows the fluid to slip over walls such that only velocities tangent to the boundary are possible leading to the condition $\langle \boldsymbol{u}, \vec{n} \rangle = 0$ on the surface of the obstacle. This condition is weakly imposed by setting U_h^+ according to U_h^- yet replacing the velocity such that the average velocity is zero in normal direction

$$\langle \frac{1}{2}(\boldsymbol{u}^+ + \boldsymbol{u}^-), \vec{n} \rangle = 0 \tag{2.56}$$

which leads to

$$\boldsymbol{u}^+ = \boldsymbol{u}^- - 2\langle \boldsymbol{u}^-, \vec{n}\rangle \vec{n}. \tag{2.57}$$

In the case of a viscous fluid modeled by the Navier-Stokes equations the fluid sticks to the wall due to friction forces resulting in a no-slip boundary condition. This is realized by setting $\boldsymbol{u}^+ = 0$ in U_h^+.

The last step is the actual definition of the numerical flux function $H(U_h^-, U_h^+, \vec{n})$ in equation (2.39). Since the solution space is discontinuous, the normal component of the flux which is integrated over the element surfaces (cf. equation (2.8)) is not uniquely defined. This problem is also known from finite volume methods and is solved by so-called *approximative Riemann solvers*. An extensive overview can be found in [32]. In [33] some of these numerical fluxes are investigated in the context of the Runge-Kutta discontinuous Galerkin framework.

Although there is actually freedom in choosing the numerical flux function, two major properties have to be fulfilled for the discretization scheme to be reasonable. First, the numerical flux H has to be consistent with the physical flux in the sense that

$$H(U_h^-, U_h^-, \vec{n}) = F^C(U_h^-) \cdot \vec{n}. \tag{2.58}$$

And second, it has to be conservative leading to

$$H(U_h^+, U_h^-, \vec{n}) = -H(U_h^-, U_h^+, -\vec{n}). \tag{2.59}$$

Consider two elements Ω_{k_1} and Ω_{k_2} with the common edge $e = \partial\Omega_{k_1} \cap \partial\Omega_{k_2}$ and the outer normal vector field of $\vec{n}$ in Ω_{k_1} on e. Then the outer normal vector field in Ω_{k_2} on e is given by $-\vec{n}$. In this context, property (2.59) has the meaning that exactly the amount of a quantity which is leaving Ω_{k_1} over e enters Ω_{k_2} and vice versa. Without this condition the conservativity of the global scheme could not be established.

For the purpose of this work, the *local Lax-Friedrichs flux* is chosen. It is given by

$$H(U_h^-, U_h^+, \vec{n}) = \frac{1}{2}\left(F^C(U_h^-) + F^C(U_h^+)\right) \cdot \vec{n} + \frac{1}{2} A (U_h^- - U_h^+) \tag{2.60}$$

with A the largest, absolute eigenvalue of the normal convective flux Jacobian matrix evaluated for U_h^+ and U_h^- obtained by

$$A = \max_{\{U_h^+, U_h^-\}} \left|\lambda(F_{\vec{n}}^{C\prime})\right|. \tag{2.61}$$

Consistency can be verified by

$$\begin{aligned} H(U_h^-, U_h^-, \vec{n}) &= \frac{1}{2}\left(F^C(U_h^-) + F^C(U_h^-)\right) \cdot \vec{n} + \frac{1}{2} A (U_h^- - U_h^-) \\ &= F^C(U_h^-) \cdot \vec{n} \end{aligned} \tag{2.62}$$

and conservativity follows from

$$\begin{aligned} H(U_h^+, U_h^-, \vec{n}) &= \frac{1}{2}\left(F^C(U_h^+) + F^C(U_h^-)\right) \cdot \vec{n} + \frac{1}{2} A (U_h^+ - U_h^-) \\ &= -\left[\frac{1}{2}\left(F^C(U_h^-) + F^C(U_h^+)\right) \cdot (-\vec{n}) + \frac{1}{2} A (U_h^- - U_h^+)\right] \\ &= -H(U_h^-, U_h^+, -\vec{n}). \end{aligned} \tag{2.63}$$

Roughly speaking, the flux (2.60) consists of two parts, namely a central flux which is stabilized by an upwind part. This Lax-Friedrichs flux is often referred to in literature and is known to be relatively stable due to its dissipative nature, which is important for the transsonic test cases in this work.

The discretization discussed so far does only include the convective part of equations (2.45). Extensions needed for the diffusive part are discussed in section 2.4 where the discretization of higher order derivatives in the DG scheme is presented.

2.3 Temporal Discretization

The discontinuous Galerkin spatial discretization, as introduced in section 2.1, results in a system of non-linear, coupled ODEs (*ordinary differential equations*). This procedure is referred to as the *method of lines*. An alternative approach is to treat the time in the same way as the spatial dimensions leading to the so-called *space-time discontinuous Galerkin method* introduced for the Euler equations in [34]. One advantage of the method of lines is that the theory of time integration schemes for ODEs can be applied to the resulting semi-discrete system (2.39) without paying further attention to where the spatial discretization came from. Since the DG method is designed to be of arbitrary order, the temporal discretization should also be of higher order to prevent the loss of accuracy. Here two major classes of time integration schemes can be distinguished. On the one hand, *explicit schemes* where the state in one time step only depends on the previous one. On the other hand, *implicit schemes* where, in contrast, the state in the following time step is computed by solving an implicit equation.

As discussed in section 2.1, the DG discretization leads to a very local method in the sense that the update for one cell only depends on the states of the direct neighbors. In particular, it only depends on the trace of the conserved variables on the common interface. Together with an explicit time stepping scheme this leads to a method which can efficiently be parallelized as investigated in [35]. This stems from the fact that the additional effort for the time integration is roughly speaking a vector addition. In contrast, in implicit methods a non-linear system has to be solved for each time step.

Since the aim of this work is to investigate the potential of DG methods on massively parallel streaming processors, it concentrates on the class of explicit Runge-Kutta time integration schemes. Denoting the spatial discretization including boundary conditions by an operator $\mathcal{N}_h$ leads to the following system of ODEs

$$\frac{\partial U_h}{\partial t} + \mathcal{N}_h(t, U_h) = 0 \tag{2.64}$$

where U_h can be thought of as the nodal values describing the solution in $\mathcal{V}_h$ (cf. section 2.1). Applying a finite difference approximation to the temporal derivative with a step size of Δt results in the following system

$$\frac{U_h^{n+1} - U_h^n}{\Delta t} + \mathcal{N}_h(t^n, U_h^n) = 0 \tag{2.65}$$

which yields the explicit time marching formula

$$U_h^{n+1} = U_h^n - \Delta t \mathcal{N}_h(t^n, U_h^n) \tag{2.66}$$

with the initialization $U_h^0 = U_h(t = 0)$. This is the so-called *explicit Euler time stepping* scheme which is known to be stable only for a small Δt. The range of feasible time steps can be significantly enlarged by applying a Runge-Kutta method where the final time step U_h^{n+1} not only depends on U_h^n but also on a number of intermediate steps. The idea of Runge-Kutta methods is to integrate (2.64) in time over the intervals $[t^n, t^{n+1}]$ and apply higher order integration rules to the integral

$$S_n = \int_{t^n}^{t^{n+1}} \mathcal{N}_h(t, U_h) dt, \tag{2.67}$$

whereas in the explicit Euler method S_n is approximated by multiplying the state at t^n with the interval length Δt. In general, explicit s-stage Runge-Kutta methods for the discretization of ODEs have the following form

$$\begin{aligned} U_h^{(1)} &= \mathcal{N}_h\left(t^n, U_h^n\right) \\ U_h^{(i)} &= \mathcal{N}_h\left(t^n + c_i \Delta t, U^n + \Delta t \sum_{j=1}^{i-1} a_{ij} U_h^{(j)}\right), \quad i = 2, \dots s \\ U_h^{n+1} &= U_h^n + \Delta t \sum_{j=1}^{s} b_j U_h^{(j)}. \end{aligned} \tag{2.68}$$

Here U_h^n denotes the current time step and U_h^{n+1} the successive one. These methods can be characterized by a so-called *Butcher tableau*

$$\begin{array}{c|ccccc} c_1 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s. \end{array} \tag{2.69}$$

This notation is introduced in [36], which gives a fundamental overview on this field. Usually the coefficients a,b and c are derived such that the method fulfills a given order k which means that the local truncation error is $\mathcal{O}(\Delta t^{k+1})$.

From the computational point of view such a method requires s times as much additional memory as the conservative variables U_h since the intermediate states $U^{(j)}$ are all needed in the final stage. For higher dimensional problems this memory consumption might be critical. A solution for this issue is presented in [37] leading to the class of so-called *low storage Runge-Kutta methods*. Here the algorithm is reformulated in the following way

$$\begin{aligned} dU_h^{(j)} &= A_j dU_h^{(j-1)} + \Delta t \mathcal{N}_h\left(U_h^{(j-1)}\right) \\ U_h^{(j)} &= U_h^{(j-1)} + B_j dU_h^{(j)} \qquad\qquad j = 1, \dots s \end{aligned} \tag{2.70}$$

where the coefficients A and B are derived from the original ones a,b and c in algorithm (2.68). Furthermore, $\mathcal{N}_h$ is assumed not to be time-dependent. In that work $A_1 = 0$ is chosen such that the resulting Runge-Kutta method is self-starting. Note that the additional memory consumption of (2.68) is independent of the number of stages s and is in particular only of the size of U_h. However, the feature of low storage requirements does not come for free. In [37] a five stage Runge-Kutta method is derived which is only fourth order accurate. Nevertheless, in section 4.1 it will be discussed that low storage is more crucial than computational effort since the stream processors taken into account are optimized that way.

An implicit time stepping scheme is obtained by evaluating $\mathcal{N}_h$ at the unknown state U_h^{n+1} in equation (2.66) which leads to the *backward or implicit Euler scheme*

$$U_h^{n+1} = U_h^n - \Delta t \mathcal{N}_h(t^{n+1}, U_h^{n+1}). \tag{2.71}$$

This scheme is known to be A-stable, which usually allows for much larger time steps Δt in practice than explicit methods. However, the computationally attractive characteristics of an explicit scheme are lost. In each time step the non-linear system (2.71) has to be solved, which is commonly done by Newton's method. Thus, it has to be dealt with several very large linear systems for each time integration. In the present application obtaining the Jacobi matrix $\frac{\partial \mathcal{N}_h}{\partial U_h}$, which is the key part of Newton's method, leads to additional programming and computational effort. Another feature of the explicit method (2.66) is to be matrix free. Thus, the memory requirement scales like $\mathcal{O}(N)$ where N is the number of interpolation points in the discretization. In contrast, an implicit method usually requires $\frac{\partial \mathcal{N}_h}{\partial U_h}$ to be in memory leading to a requirement which scales like $\mathcal{O}(N^2)$. If the arising linear systems are solved with an iterative solver like GMRES, the Jacobi matrix does not necessarily have to be in memory since only products with this matrix are needed. However, incomplete LU factorization (ILU), which is probably the most often applied family of preconditioners, is based on the matrix itself.

A third and different approach is based on the time-independent version of the hyperbolic equations already presented in 2.4

$$\nabla \cdot F(U) = 0. \tag{2.72}$$

Discretizing with the DG approach, presented in this chapter, leads to the non-linear equation

$$\tilde{\mathcal{N}}_h(U_h) = 0. \tag{2.73}$$

This can be seen as the limit $t \to \infty$ of the backward Euler scheme. The difference between $\tilde{\mathcal{N}}_h$ and $\mathcal{N}_h$ is that the multiplication with the inverse mass matrix is left out in the time-independent case. This system of non-linear equations (2.73) is usually solved using Newton's method.

2.4 Shock Capturing

A well known and intensively investigated phenomenon of hyperbolic equations such as (2.1) is the presence of discontinuities (also called shocks) in the solution. Even if the initial condition U_0 is smooth, regions with large gradients and also discontinuities may arise during time evolution. In fluid dynamics this typically happens for large Mach numbers or for transsonic flows.

In the case of a transsonic flow the fluid enters the domain with a Mach number $M < 1$ and then, due to interactions with obstacles, the Mach number increases to $M > 1$ and decreases

again. Two of such transsonic test cases are demonstrated in section 4.3.1. When representing the solution U with polynomials, like in the discontinuous Galerkin method, spurious oscillations may arise in the numerical approximation U_h in the regions of shocks. Due to the well known *Gibbs phenomenon*, this issue is not necessarily resolvable with a grid refinement. Furthermore, this becomes even worse for higher order approximations with increasing polynomial degree P. The presence of such spurious oscillations is particularly critical in the case of the Euler equations. In regions where there is low pressure or density in the solution, oscillations might lead to negative or zero values which are not feasible. This problem is illustrated by the Godunov theorem. Consider the one dimensional convection equation

$$\frac{\partial U}{\partial t} + c\frac{\partial U}{\partial x} = 0,\ x \in \mathbb{R} \tag{2.74}$$

with a convection speed $c \in \mathbb{R}$ and initial conditions $U(x,0) = U_0(x)$ which is fulfilled by $U(x,t) = U_0(x - ct)$. As a result the monotonicity of U_0 with respect to x is fixed in time. Roughly speaking, preserving monotonicity means that no new local maxima nor minima are produced in the solution. Thus, a numerical discretization scheme should fulfill this condition, which is in general not given due to the following theorem.

Godunov theorem: Linear one-step methods with second order spatial discretization for the convection equation cannot preserve monotonicity if not

$$|c|\frac{\Delta t}{\Delta x} \in \mathbb{N} \tag{2.75}$$

is satisfied.

A proof and further discussions can be found in [31]. The statement of the theorem conflicts with the concept of higher order polynomial basis functions inside the elements. Since the basis functions are chosen to be discontinuous across element interfaces, a possible solution could be to align shocks in the solution with inter-element boundaries in the discretization mesh. This is, however, not practical as shocks likely change their position during time. Since the DG method can be seen as a generalization of the finite volume discretization method, it is tempting to reuse stabilization techniques like slope limiters. These limiters are successfully applied in FVM to reconstruct gradients of the solution and limit them to reasonable values. For instance in [38, 9], slope limiters similar to the FVM ones are extended to the DG method. A major drawback of this approach is a loss of accuracy since these techniques only work for lower degrees of basis polynomials. In [39] limiters for higher order DG methods are presented however the extension to a three dimensional solver would be challenging. Another disadvantage of limiter methods is their dependency on the element geometry. Different element types need different limiting procedures.

A completely different approach is to modify the differential equations (2.1) by adding a small amount of *artificial viscosity* which yields

$$\frac{\partial U}{\partial t} + \nabla \cdot F(U) + \nabla \cdot \left(\epsilon^2 \nabla U\right) = 0. \tag{2.76}$$

Here $\epsilon : (0,T] \times \Omega \to [0, \epsilon_{\text{max}}]$ is a function determining where there are large gradients and shocks in the solution U. Due to the additional dissipation controlled by ϵ_{max}, discontinuities

are smeared out such that the solution is in the range of the polynomial basis functions again. Hereby spurious oscillations are suppressed. Furthermore, by spreading its width, it is possible to capture a shock within one element. In contrast, when decreasing the order of the numerical method in the vicinity of shocks, it is necessary to drastically refine the grid in those regions. This artificial viscosity approach is yet not for free. Since the basic DG scheme is not able to handle higher order derivatives, equation (2.76) has to be rewritten as a system of first order PDEs increasing the number of variables. This leads to

$$\begin{aligned} Q + \epsilon \nabla U &= 0 \\ \frac{\partial U}{\partial t} + \nabla \cdot (F(U) + \epsilon Q) &= 0 \end{aligned} \tag{2.77}$$

where $Q : (0, T] \times \Omega \to \mathbb{R}^{m \times 3}$ is an auxiliary variable approximating the gradient of the solution U. These equations are both discretized with the same DG scheme, introduced in section 2.1, yielding

$$\begin{aligned} Q_{k,x_m} =& S_{k,x_m} \mathcal{I}_V \epsilon_k U_k - M_{\partial\Omega_k} \frac{1}{2} \left(\epsilon_k^- U_k^- + \epsilon_k^+ U_k^+ \right) n_{x_m}, \quad m = 1, 2, 3 \\ \frac{\partial U_k}{\partial t} =& \sum_{m=1}^{3} S_{k,x_m} \left(F_m \left(\mathcal{I}_V U_k \right) + \mathcal{I}_V \epsilon_k Q_{k,x_m} \right) \\ & - M_{\partial\Omega_k} \left[H \left(U_k^-, U_k^+, \vec{n} \right) + \frac{1}{2} \left(\epsilon_k^- Q_k^- + \epsilon_k^+ Q_k^+ \right) \cdot \vec{n} \right]. \end{aligned} \tag{2.78}$$

In the first equation n_{x_m} denotes the m-th component of the normal vector $\vec{n}$ and the same applies for Q_{k,x_m}. Note that in (2.78) for the two new viscous fluxes a central flux is implemented which is represented by the arithmetic averages of local and remote state. This is a reasonable choice since diffusion does not have a preferred direction of propagation. Further discussions on numerical fluxes for elliptic problems can be found in [40].

Similar to the derivation of the discrete version of the purely hyperbolic equations in (2.39), the local states at surface cubature nodes are given by

$$\epsilon_k^- = \mathcal{I}_S \epsilon_k, \; Q_k^- = \mathcal{I}_S Q_k. \tag{2.79}$$

The corresponding remote variables ϵ_k^+ and Q_k^+ are obtained in the same way like in section 2.1. With these additional variables the DG scheme is capable of dealing with diffusive terms. Yet, ϵ has to be defined in some way. It is important that artificial diffusion is only added near shocks and the equations remain unchanged in smooth regions of the domain. For this purpose, a shock detector is presented in [4] which is slightly modified here in order to fit in the GPU framework presented in chapter 4. Based on this, an element-wise constant viscosity is assigned to each discretization cell. Like in this work, the authors use a representation of the numerical solution U_h in a hierarchical, orthogonal basis. Then they compare the ratio of modes with high and low frequencies in the solution.

Let u_h be one component of the unknown variables U_h inside one element and $\hat{u}$ the vector of modal expansion coefficients such that

$$u_h = \sum_{i=1}^{N_P} \hat{u}_i \psi_i, \; \tilde{u}_h = \sum_{i=1}^{N_{P-1}} \hat{u}_i \psi_i. \tag{2.80}$$

Here $\tilde{u}_h$ is the solution where the modes with the highest frequencies - namely the polynomials with degree P - are left out. This is straightforward due to the hierarchical basis polynomials. For each element Ω_k the following *smoothness indicator* is evaluated

$$S_k = \frac{(u_h - \tilde{u}_h, u_h - \tilde{u}_h)_{\Omega_k}}{(u_h, u_h)_{\Omega_k}}. \tag{2.81}$$

Roughly speaking, this indicator is close to zero in regions where the solution is smooth and increases where there are oscillations and large gradients. In order to obtain a discrete version of the shock detector, a reduced Vandermonde matrix is used. This means that the columns in V corresponding to the polynomials ψ with degree P are replaced by zero leading to a matrix V_{red}. The projection onto the space of the polynomials up to degree $P-1$ is then established by the interpolation operator $\mathcal{I}_{\text{red}} = V_{\text{red}} V^{-1}$. Finally, the scalar products in equation (2.81) are assembled by the matrices

$$M_{k,\text{red}} = \mathcal{I}_{\text{red}}^T M_k \mathcal{I}_{\text{red}} \tag{2.82}$$

for the nominator and M_k for the denominator. Based on S_k, the viscosity is chosen to be constant in element number k according to the following smooth function

$$\epsilon_k = \begin{cases} 0 & \text{if} \quad s_k < s_0 - \kappa \\ \frac{\epsilon_0}{2}\left(1 + \sin\frac{\pi(s_k - s_0)}{2\kappa}\right) & \text{if} \quad s_0 - \kappa \leq s_k \leq s_0 + \kappa \\ \epsilon_0 & \text{if} \quad s_k > s_0 + \kappa \end{cases} \tag{2.83}$$

where $s_k = |\log_{10} S_k|$. The constants s_0 and κ are chosen empirically which might be challenging in practice. However, this approach is one of the most popular throughout literature. One disadvantage of this method are the inter-element jumps in ϵ. The discontinuity of ϵ influences the stability of the explicit time stepping and leads to a significantly smaller Δt. This issue is for example investigated in [41] with respect to an explicit time stepping scheme and an approach is presented how to smooth viscosity and cancel out inter-element jumps. Another approach dealing with PDE based artificial viscosity is presented in [42]. Additional equations for the viscosity are added to the PDE system leading to a smooth viscosity, which results in a computationally more expensive method.

Chapter 3

Mesh Curvature Approach

3.1 Problems of Under-Resolved Geometries

As mentioned in the introduction and during the derivation of the discontinuous Galerkin scheme, one outstanding feature of this method is its arbitrarily high order in space. This results in a very high resolution of the method without extensive refinement of the discretization mesh. However, this leads also to a major drawback. Potential deficiencies in the representation of geometric entities in the flow field are resolved within the solution, which may lead to nonphysical behavior.

Moreover, since the state of the art discretization technique in computational fluid dynamics over the past decades is the low order finite volume method, most available meshes consist of straight-sided elements. It is thus tempting to reuse these grids in the context of higher order discretizations such as the DG method. When representing a physically curved geometry with straight-sided elements, there are kinks arising at the element interfaces on the surface. While this is a sufficient representation for the constant basis functions in a finite volume method, the higher order polynomials are able to resolve these kinks, which leads to a falsification of the solution.

In the worst case of transsonic and supersonic flows small non-physical shocks appear on each element interface, which probably crashes the solver. Thus, it is obligatory to introduce curved elements into the discontinuous Galerkin discretization in order to enable a higher order boundary representation. To be precise, the boundary should at least be resolved in the same order as the basis functions to obtain reasonable solutions.

In the formal derivation of the DG discretization at the beginning of section 2.1 almost no restrictions were made on the shape of the elements. Yet, for the purpose of this work, the elements are chosen to be logically tetrahedra in order to use standard mesh generation techniques. These tetrahedra are then curved with respect to polynomial mapping functions Ψ_k, which are within the local ansatz space $\mathcal{V}_h^k$.

This issue is visualized in figure 3.1. A flow past a sphere with freestream Mach number $M_\infty = 0.38$ is simulated on both a mesh with straight-sided elements in the top image and a mesh with isoparametric, curved elements in the bottom image. It can be seen that the solver converged to a non-physical solution in the first case, although the mesh is drastically refined (approximately 20 times as many elements), whereas in the second case a much better solution is achieved on a coarser grid. Originally, this was investigated in detail in [43] for the two dimensional flow over a sphere.

For many applications, and particularly industrial ones, the considered geometries are known in an analytic form, e.g., by NURBS surfaces. However, this curvature information on its own

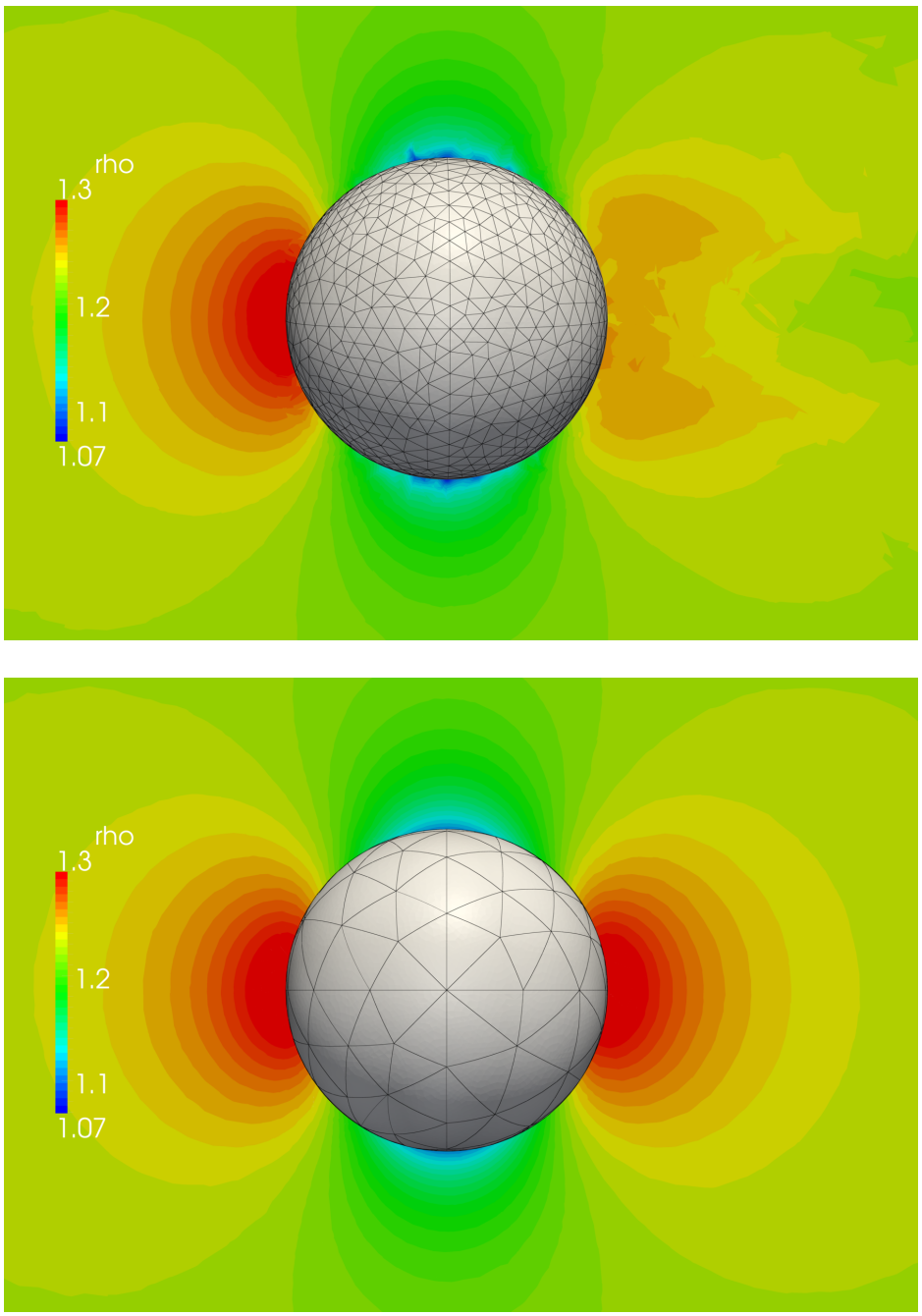


Figure 3.1: Density distribution of a subsonic Euler flow past a sphere with $P = 4$ basis functions. Disturbed solution on straight-sided elements (top) and isoparametric curved elements (bottom).

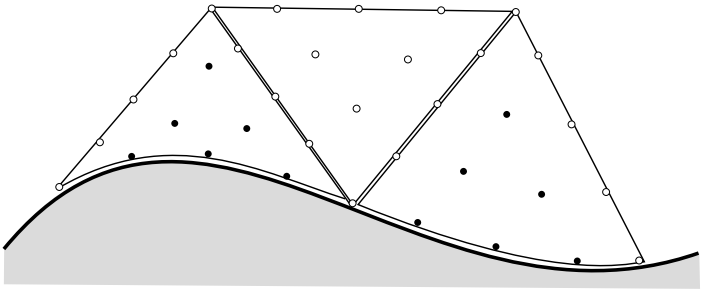


Figure 3.2: Two dimensional curved elements on a physical boundary with displaced (filled) and original (empty) interpolation nodes.

is not sufficient. In addition, the location of interpolation and cubature points in the layer of cells covering the curved surface has to be corrected. Assuming that the mesh generator is able to evaluate the analytical description of the surface, mesh nodes match up with the physical geometry. However, this is not necessary for the approach proposed in this work. This situation is visualized in figure 3.2 for a two dimensional curved discretization. The interpolation points in the corner of the left and right triangle remain untouched, since the mesh generator has placed them according to the curved boundary. Moreover, by only displacing the interpolation points denoted by the filled circles, all nodes in the triangle in the middle keep their original position. Thus, in two dimensions only the elements sharing an edge with the physical boundary have to be curved.

This is much more complicated in three dimensions for tetrahedra. Here three different cases can be distinguished. One tetrahedron could share a face with the boundary, or an edge or only one node. This ends up with several layers of elements which have to be curved. Even more attention has to be payed for anisotropic meshes, where curving the first layer of elements on the boundary could result in overlappings with the following layers.

In both two and three dimensions one has to be careful that the displacement of interpolation points is performed smoothly. It is crucial to avoid agglomerations of interpolation points, since this affects the determinant of the Jacobian of Ψ_k resulting in a poor integration quality.

3.2 Mesh Curvature Based on Linear Elasticity

For the purpose of this work, a bounding box is put around the geometry marking an area in which cells get curved instead of tracking cells adjacent to the boundary. This is visualized in figure 3.3. Here the red box distinguishes between cells that are close to the geometry which are allowed to be curved and the cells which may remain straight-sided.

The discretization mesh is modeled as embedded in some flexible material on which forces are performed. These forces are chosen in a way such that the straight-sided boundary matches the physical, curved geometry. By solving the linear elasticity equations with a continuous finite element approach (FEM), this deformation of the boundary shape is transported into the domain. Furthermore, deformations can then be evaluated at arbitrary points since they are available in some finite element polynomial basis. The finite element method for the linear

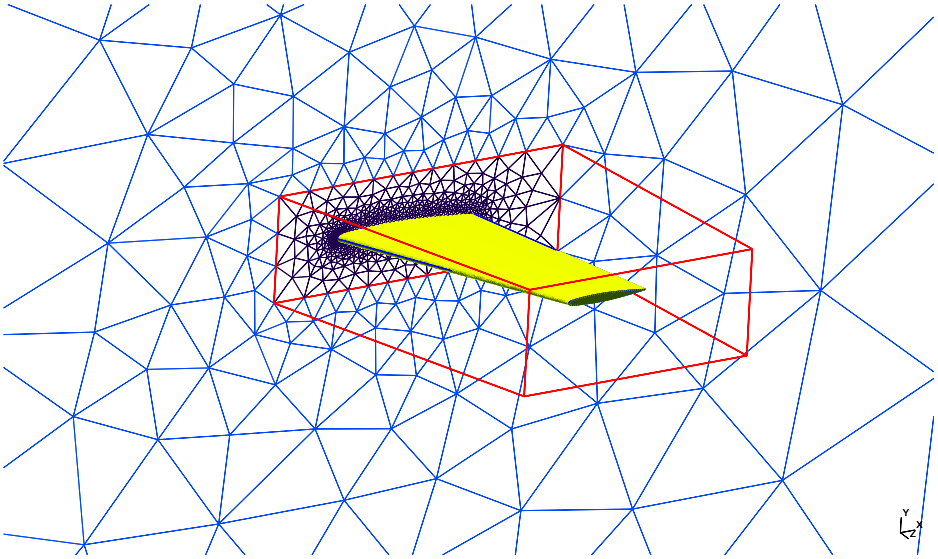


Figure 3.3: Bounding box (red) around ONERA M6 wing (yellow) marking the sub-mesh for the linear elasticity solver. The blue and violet mesh in the background is the triangulation of the symmetry wall where the airfoil is fixed.

elasticity and the discontinuous Galerkin scheme do thus not have to rely on the same set of interpolation points. They can even be of different order.

In order to keep this process as cheap as possible, the affected cells within the bounding box are extracted and the linear elasticity equations are solved on the resulting sub-mesh. A very similar approach is presented in [44] leading to comparable results.

The governing equations of *linear elasticity* are given by

$$\nabla \cdot \sigma = f \quad \text{on} \quad \Omega_c \tag{3.1}$$

where f denotes body forces acting on the solid, e.g. gravity, which are not present in this model. $\Omega_c \subset \Omega$ is the subset of cells within the DG mesh Ω that are near the geometry and have to be curved. In equation (3.1) σ denotes the *stress tensor* and is given by

$$\sigma = \lambda \mathrm{Tr}(\epsilon) I + 2\mu\epsilon. \tag{3.2}$$

The stress tensor is defined in terms of the *strain tensor*

$$\epsilon = \frac{1}{2}(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T). \tag{3.3}$$

Here λ and μ denote the *Lamé parameters*, which can be expressed in terms of *Young's modulus* $\mathcal{E}$ and *Poisson's ratio* ν as

$$\lambda = \frac{\nu\mathcal{E}}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{\mathcal{E}}{2(1+\nu)}. \tag{3.4}$$

Furthermore, in equation (3.1) $\mathrm{Tr}(\cdot)$ denotes the trace operator and $I \in \mathbb{R}^{3\times 3}$ is the identity matrix. The unknown function $\boldsymbol{u} : \Omega_c \to \mathbb{R}^3$ describes the deformation the material performs under given boundary conditions which will be discussed later.

For the purpose of curving the discretization mesh, it is not important that the parameters $\mathcal{E}$ and ν describe the properties of a physical material. However, it is crucial to understand how they influence the solution. Young's modulus is a measure for the stiffness of the material. Larger values for $\mathcal{E}$ result in smaller deformations. However, since $f = 0$ within this chapter, $\mathcal{E}$ has no influence on the solution. Poisson's ratio describes how much a material expands in two coordinate directions when compressed in the third.

Before discussing the actual deformation, the parametrized surface has to be defined. This surface is assumed to be based on *non-uniform, rational B-splines* (NURBS). A detailed survey on the theory and application of splines can be found in [45]. Let $(u_0, \ldots, u_m)$, $0 \le u_i \le 1$ be the so-called *knot vector* with $u_i \ge u_{i-1}$, $1 \le i \le m$ for a given number of control points n and $P \ge 0$. The i-th B-spline basis function of degree P with knot vector u is then given by

$$N_{i,0}(u) = \begin{cases} 1 & \text{if} \quad u_i \le u \le u_{i+1} \\ 0 & \text{else} \end{cases}, \tag{3.5}$$

$$N_{i,P}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,P-1}(u) + \frac{u_{i+P+1} - u}{u_{i+P+1} - u_{i+1}} N_{i+1,P-1}(u). \tag{3.6}$$

Further, the derivatives of the B-splines are obtained in a similar recursion

$$N'_{i,P}(u) = \frac{P}{u_{i+p} - u_i} N_{i,P-1}(u) + \frac{P}{u_{i+P+1} - u_{i+1}} N_{i+1,P-1}(u) \tag{3.7}$$

and can also be computed efficiently. In the evaluation of B-splines the term $\frac{0}{0}$ may arise and is typically set to 1 throughout literature.

If additionally n weights $w_i > 0$, $1 \le i \le n$ are given, the NURBS basis functions can be formed by

$$R_{i,P}(u) = \frac{N_{i,P}(u) w_i}{\sum_{j=0}^{n} N_{j,P}(u) w_j}. \tag{3.8}$$

Finally, for a set of control points $\boldsymbol{q}_i = (x_i, y_i, z_i) \in \mathbb{R}^3$ the NURBS curve $\mathcal{C} : [0, 1] \to \mathbb{R}^3$ is given by

$$\mathcal{C}(u) = \sum_{i=0}^{n} R_{i,P}(u) \boldsymbol{q}_i. \tag{3.9}$$

In a NURBS curve the partitioning of the knot vector determines which control point is influencing the curve for a given u. The weights determine the overall influence of a control point on the curve.

The two dimensional case of a NURBS surface is similar to the one dimensional case and is obtained by a tensor product approach. Let P_u and P_v be the degrees in each parametrized direction. Further n_u and n_v are the numbers of control points in each direction spanning the grid $\boldsymbol{q}_{i,j} \in \mathbb{R}^3$, $0 \le i \le n_u$ and $0 \le j \le n_v$. With this grid a matrix of weights $w_{i,j}$ is associated. The knot vectors can then be defined by $(u_0, \ldots, u_{m_u})$ and $(v_0, \ldots, v_{m_v})$ with $m_u = n_u + P_u + 1$

and $m_v = n_v + P_v + 1$ leading to the two dimensional NURBS surface $\mathcal{S} : [0,1]^2 \to \mathbb{R}^3$. This is obtained by

$$\mathcal{S}(u,v) = \sum_{i=0}^{n_u} \sum_{j=0}^{n_v} R_{i,j}(u,v)\boldsymbol{q}_{i,j} \tag{3.10}$$

where the tensor product of the NURBS is given by

$$R_{i,j}(u,v) = \frac{N_{i,P_u}(u)N_{j,P_v}(v)w_{i,j}}{\sum\limits_{k=0}^{n_u}\sum\limits_{l=0}^{n_v} N_{k,P_u}(u)N_{l,P_v}(v)w_{k,l}}. \tag{3.11}$$

The partial derivatives are obtained by applying the quotient rule which yields

$$\begin{aligned}\frac{\partial R_{i,j}(u,v)}{\partial u} = & \frac{N'_{i,P_u}(u)N_{j,P_v}(v)w_{i,j}\sum\limits_{k=0}^{n_u}\sum\limits_{l=0}^{n_v} N_{k,P_u}(u)N_{l,P_v}(v)w_{k,l}}{\left(\sum\limits_{k=0}^{n_u}\sum\limits_{l=0}^{n_v} N_{k,P_u}(u)N_{l,P_v}(v)w_{k,l}\right)^2} \\ & - \frac{N_{i,P_u}(u)N_{j,P_v}(v)w_{i,j}\sum\limits_{k=0}^{n_u}\sum\limits_{l=0}^{n_v} N'_{k,P_u}(u)N_{l,P_v}(v)w_{k,l}}{\left(\sum\limits_{k=0}^{n_u}\sum\limits_{l=0}^{n_v} N_{k,P_u}(u)N_{l,P_v}(v)w_{k,l}\right)^2}.\end{aligned} \tag{3.12}$$

The derivative with respect to v is obtained in the same way. This yields the Jacobi matrix of $\mathcal{S}$ as

$$J_{\mathcal{S}}(u,v) = \left[\sum_{i=0}^{n_u}\sum_{j=0}^{n_v} \frac{\partial R_{i,j}(u,v)}{\partial u}\boldsymbol{q}_{i,j} \quad \sum_{i=0}^{n_u}\sum_{j=0}^{n_v} \frac{\partial R_{i,j}(u,v)}{\partial v}\boldsymbol{q}_{i,j}\right]. \tag{3.13}$$

Since this is the standard framework in *computer-aided design* (CAD) and widely used in industrial applications, it can be assumed that a geometry is given as a NURBS surface.

In order to complete system (3.1), three different kinds of boundary conditions are added. Two Dirichlet and one Neumann type boundary condition are used

$$\begin{aligned}\boldsymbol{u} &= g \qquad \text{on } \Gamma_{D_1} \\ \boldsymbol{u} &= 0 \qquad \text{on } \Gamma_{D_2} \\ \frac{\partial \boldsymbol{u}}{\partial \vec{n}} &= 0 \qquad \text{on } \Gamma_N.\end{aligned} \tag{3.14}$$

In the situation of figure 3.3, Γ_{D_1} is the yellow surface of the airfoil. This is the part of the boundary which has to be curved in order to match with the physical geometry.

Γ_{D_2} are the five surfaces of the red box in the foreground. Here the curvature is forced to stop by the zero condition, which yields that all cells outside the red box remain untouched. Furthermore, there will be no overlappings due to this conditions. The remaining Neumann condition is active at the rear face of the bounding box. This condition only permits displacements in tangential directions. By this, points on Γ_N are able to slide within the plane, but cannot leave it.

The next step is to analyze the distance between the actual surface given by $\mathcal{S}(u, v)$ in equation (3.10) and the straight-sided tetrahedra in the discretization mesh. As mentioned earlier, the mesh generator does not necessarily have to place the nodes of a surface tetrahedron onto the spline S. The gap between the straight-sided tetrahedron and the spline is described by the function $g : \Gamma_{D_1} \to \mathbb{R}^3$ in (3.14). This function is chosen according to the solution of a closest point problem of the following form

$$\begin{aligned} \min_{(u,v)} \quad & f(u, v) := \frac{1}{2} \|\mathcal{S}(u, v) - \boldsymbol{x}\|_2^2 \\ \text{s.t.} \quad & 0 \leq u, v \leq 1. \end{aligned} \tag{3.15}$$

This non-linear least squares problem has to be solved for every $\boldsymbol{x} \in \Gamma_{D_1}$. To be precise, $\boldsymbol{x}$ plays the role of the degrees of freedom of the finite element method on the boundary Γ_{D_1}. Assume that (u^*, v^*) is the unique, globally optimal solution of problem (3.15) for a given $\boldsymbol{x}$. Then $g(\boldsymbol{x}) = \mathcal{S}(u^*, v^*) - \boldsymbol{x}$ describes the displacement the point $\boldsymbol{x}$ on the polygonal boundary Γ_{D_1} has to perform in order to match with the curved geometry. It will be discussed later why this very strong assumption of uniqueness in (3.15) is reasonable. Furthermore, it is essential that the resulting mapping described by g is bijective. Otherwise, singularities may arise in the curved mesh, which results in overlappings and zero-volume cells.

In order to solve the minimization problems in (3.15), the *Gauss-Newton* approach is applied. The techniques and notations presented in [46] are closely followed for this purpose. In a more general framework the function to be minimized is written in terms of a residual vector $r : \mathbb{R}^n \to \mathbb{R}^m$, $r(\boldsymbol{u}) = (r_1(\boldsymbol{u}), \ldots, r_m(\boldsymbol{u}))^T$ as

$$f(\boldsymbol{u}) = \frac{1}{2} \|r(\boldsymbol{u})\|_2^2 . \tag{3.16}$$

Further, the Jacobi matrix of r is denoted by

$$J_r(\boldsymbol{u}) = \begin{pmatrix} \nabla r_1(\boldsymbol{u})^T \\ \vdots \\ \nabla r_m(\boldsymbol{u})^T \end{pmatrix} . \tag{3.17}$$

Then the gradient of f can be formulated as

$$\nabla f(\boldsymbol{u}) = \sum_{j=1}^{m} r_j(\boldsymbol{u}) \nabla r_j(\boldsymbol{u}) = J_r(\boldsymbol{u})^T r(\boldsymbol{u}) \tag{3.18}$$

and the Hesse matrix as

$$\begin{aligned} \nabla^2 f(\boldsymbol{u}) &= \sum_{j=1}^{m} \left(\nabla r_j(\boldsymbol{u}) \nabla r_j(\boldsymbol{u})^T + r_j(\boldsymbol{u}) \nabla^2 r_j(\boldsymbol{u}) \right) \\ &= J_r(\boldsymbol{u})^T J_r(\boldsymbol{u}) + \sum_{j=1}^{m} r_j(\boldsymbol{u}) \nabla^2 r_j(\boldsymbol{u}). \end{aligned} \tag{3.19}$$

Due to the assumption of uniqueness, it is sufficient to consider the first-order necessary conditions for optimality, which are given by finding a point $\boldsymbol{u}^*$ which fulfills $\nabla f(\boldsymbol{u}^*) = 0$. This

can be achieved by choosing a start vector $\boldsymbol{u}_0$ which is sufficiently close to $\boldsymbol{u}^*$ and applying *Newton's method*

$$\begin{aligned} \nabla^2 f(\boldsymbol{u}_k)\Delta\boldsymbol{u}_k &= -\nabla f(\boldsymbol{u}_k) \\ \boldsymbol{u}_{k+1} &= \boldsymbol{u}_k + \Delta\boldsymbol{u}_k. \end{aligned} \tag{3.20}$$

However, the evaluation of the Hesse matrix $\nabla^2 f$ is computationally costly. This leads to the idea of the Gauss-Newton method where the second summand in (3.19) is omitted and $J_r(\boldsymbol{u})^T J_r(\boldsymbol{u})$ is used as an approximation. In the vicinity of the stationary point $\boldsymbol{u}^*$ this approximation is reasonable since the second order derivatives in (3.19) are weighted with the residual function r which is close to zero. The modified version of equation (3.20) is then given by

$$\begin{aligned} J_r(\boldsymbol{u}_k)^T J_r(\boldsymbol{u}_k)\Delta\boldsymbol{u}_k &= -J_r(\boldsymbol{u}_k) r(\boldsymbol{u}_k) \\ \boldsymbol{u}_{k+1} &= \boldsymbol{u}_k + \Delta\boldsymbol{u}_k. \end{aligned} \tag{3.21}$$

For the purpose of problem (3.15), computational efficiency is of great importance. Although the dimension of each minimization problem is low, there is a very large number of systems that have to be solved depending on the grid and the order of the finite element method. In this specific context the dimensions are given by $n = 2$ and $m = 3$. Furthermore, in algorithm (3.21) it holds $\boldsymbol{u} = (u, v)$, $r(\boldsymbol{u}) = \mathcal{S}(u, v) - \boldsymbol{x}$ and $J_r(\boldsymbol{u}) = J_{\mathcal{S}}(u, v)$. Because of the low dimension and the symmetry of the matrix in (3.21), these systems are explicitly solved by Cholesky decomposition.

Since the NURBS surface is defined in the interval $[0, 1]^2$, constraints on (u, v) have to be fulfilled in (3.15). These so-called *box constraints* can be implemented into the Gauss-Newton algorithm in the following way. Whenever $(u_k, v_k) \notin [0, 1]^2$ holds, it is projected back into the feasible space

$$(u_k, v_k) \leftarrow \begin{cases} \text{if} \quad u_k < 0 : \; u_k = 0 \\ \text{if} \quad u_k > 1 : \; u_k = 1 \\ \text{if} \quad v_k < 0 : \; v_k = 0 \\ \text{if} \quad v_k > 1 : \; v_k = 1 \end{cases} . \tag{3.22}$$

It should be remarked that there is no guarantee for this procedure to converge. Moreover, it is not hard to construct examples where this projection does not lead to descent directions. This is illustrated in [47] where a solution for this issue is discussed. However, in the special setting of the closest point problem to a NURBS surface and additionally the geometries discussed here, the projection is sufficient.

As mentioned earlier, it is a strong assumption that each optimization problem for a fixed $\boldsymbol{x}$ is uniquely solvable and that algorithm (3.21) converges to the global minimum. At the following examples it is presented how the global optimality is achieved. Closely connected to this is the assumption that the resulting mapping, described by g, is bijective. Here the theoretical background will not be discussed, yet it should be mentioned that for the test cases, presented within this work, it is sufficient to place the nodes of the tetrahedral mesh on the spline $\mathcal{S}$. A further mesh refinement in the vicinity of large curvatures of $\mathcal{S}$ ensures the two conditions to be fulfilled. This functionality is implemented in the open-source mesh generator GMSH [48] which is used for all meshes within this work.

The resulting linear elasticity system is discretized and solved with the finite element toolbox GETFEM++ [49]. For a reasonable solution the toolbox is configured with the same polynomial

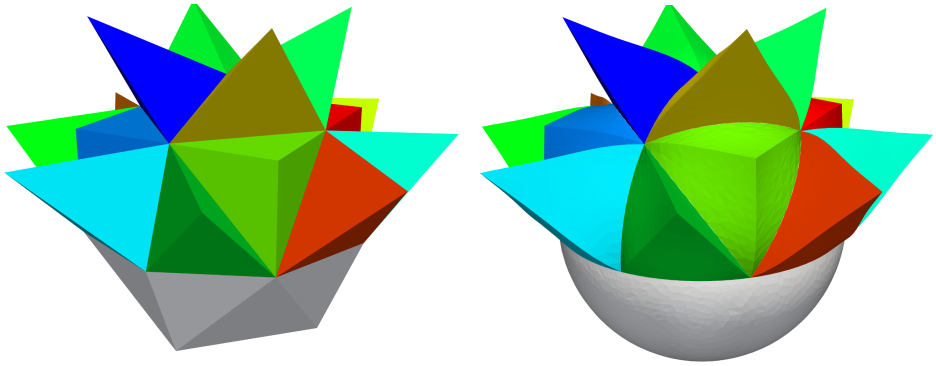


Figure 3.4: Influence of linear elasticity curvature on surface-elements (colored) of the sphere (gray). Initial mesh on the left and curved mesh on the right.

order like the DG scheme. A boundary approximation in more detail would not benefit lower order DG discretizations. This solution is then exported and stored together with the mesh for the DG flow solver. The GETFEM++ toolbox uses the Gauss-Newton solver as interpolation function on Γ_{D_1} and evaluates g at interpolation points on the boundary. Since the finite element tool operates on the same tetrahedra like the DG scheme, the evaluation of the solution is relatively cheap. For each element Ω_k the finite element solution has to be evaluated at the position of the interpolation points of the DG scheme $\{\boldsymbol{x}_i \in \Omega_k,\ 1 \leq i \leq N_P\}$. If the finite element solution was computed on a different mesh, the identification of $\boldsymbol{x}_i$ with an element Ω_k would result in an expensive operation.

The lower image in figure 3.1 shows the curvature on the surface of the sphere with radius r and centroid $\boldsymbol{x}_0$. Here the displacement can be described without the need of a parametrization by

$$g(\boldsymbol{x}) = \boldsymbol{x}_0 - \boldsymbol{x} + \frac{r}{\|\boldsymbol{x} - \boldsymbol{x}_0\|}\boldsymbol{x}. \tag{3.23}$$

In this case $\nu \approx 0.5$ is chosen for the linear elasticity equations. This leads to a smooth deformation of the discretization mesh in all coordinate directions.

The visualization of the deformation inside the three dimensional mesh is difficult (cf. figure 3.4). Here the curvature process of the sphere is demonstrated. The straight-sided surface triangulation of the sphere can be seen in the left figure painted in gray (similar to figure 3.1 but coarser). This surface is surrounded by a tetrahedral volume discretization. Some of these elements are visualized as the straight-sided, colored tetrahedra on the top of the sphere. In the figure on the right hand side it can be seen how the solution of the least squares fitting and the elasticity equations deform the mesh. First, the surface does not have kinks anymore and represents the actual sphere now. Second, the colored tetrahedra, which surround the sphere, are also slightly curved. Moreover, this figure indicates that not only the first layer of tetrahedra but also some elements in the neighborhood are affected by the curvature. This is the major difference compared to the two dimensional situation in figure 3.2.

The next case gives more insight in the curvature process. Figure 3.5 shows the NACA0012 test case. Since the geometries in the NACA series are two dimensional, this grid is stretched

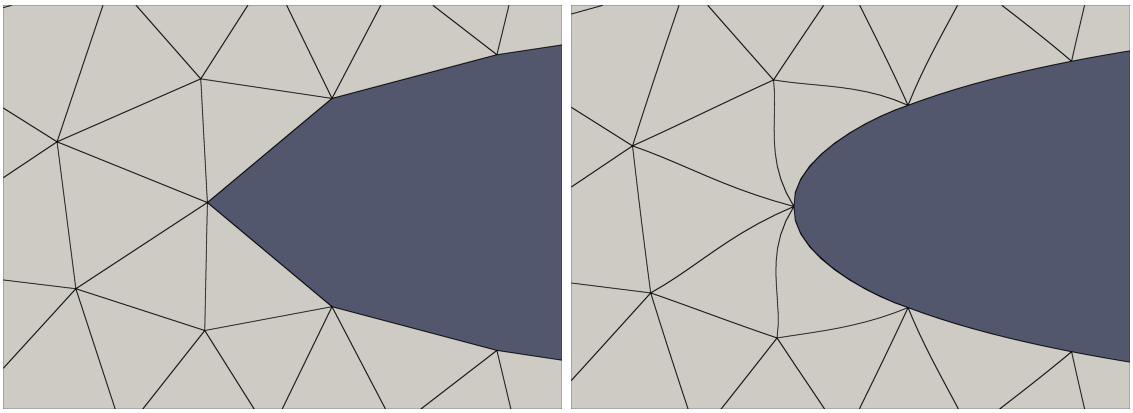


Figure 3.5: NACA0012 airfoil, leading edge. Initial grid on the left, $P = 4$ curved elements on the right.

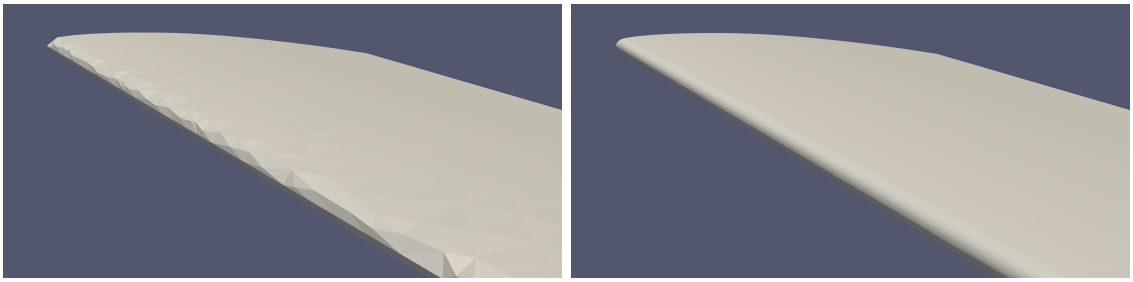


Figure 3.6: ONERA M6 airfoil, top surface with leading edge. Initial geometry produced by the mesh generator on the left. $P = 4$ curved cells around the physical geometry on the right.

into the third coordinate direction as the solver developed for this work only supports tetrahedra. The extend in z-direction is chosen to be 25% of the chord length of the airfoil. However, this test case visualizes the deformation inside the grid at least partly. Here $\nu \approx 0$ is set such that a deformation in the xy-plane does not result in displacements in z-direction. The front and back plane of the mesh are chosen to be Γ_N such that these walls are not deformed. In the three test cases discussed so far, the degree of basis polynomials is chosen to be $P = 4$ in the finite element solver, since the DG flow simulation throughout this work will mostly be of that order.

Reviewing the NACA0012 and ONERA M6 case, it should be noted that the Gauss-Newton algorithm for problem (3.15) does not necessarily converge to the global minimum. Assuming a point $\boldsymbol{x}$ in the center of the upper surface of the ONERA M6 geometry and start values (u_0, v_0) corresponding to a point $\mathcal{S}(u_0, v_0)$ near the center of the lower surface, the algorithm (3.21) would converge to a $\mathcal{S}(u^*, v^*)$ near $\mathcal{S}(u_0, v_0)$. Due to the geometric simplicity of these

two test cases, it is sufficient to solve two least squares problems per node $\boldsymbol{x}$: one with start parameter corresponding approximately to a point in the center of the upper surface and one with parameters referring to the center of the lower surface.

Although this process seems to be computationally expensive, the linear elasticity equations only have to be solved once. As mentioned earlier, the solution of the finite element solver can then be stored together with the straight-sided mesh and evaluated by the DG solver.

For discretizations based on quadrilaterals and hexahedra a different approach is chosen in [50]. Here the curvature information is retrieved from additional points which are taken from a refined mesh. This means that, e.g., every second point in the mesh is omitted and used for a polynomial representation of the curvature. However, this approach needs a hierarchy of grids and is not straightforward in unstructured discretizations.

Chapter 4

GPU Implementation of Discontinuous Galerkin

At the beginning of this chapter the functionality of modern graphics processing units (GPUs) is shortly introduced. For this purpose, the presentations in [51, 52, 53] are followed where additional information can be found. In particular, GPUs are compared to classical central processing units (CPUs) and the differences in the corresponding program paradigms are illustrated. The strengths and weaknesses of each hardware model are also outlined and it is discussed when to use GPUs or CPUs depending on the underlying problem. For a better understanding of GPU programming, the hardware and particularly the memory model is presented. It is then explained why the discontinuous Galerkin method, as described in chapter 2, is attractive for the implementation on GPUs and necessary building blocks for such an algorithm are derived. With a parallel GPU/MPI algorithm in hand, several standard numerical test cases are performed and investigated.

At the end of this chapter the discontinuous Galerkin GPU algorithm is compared to a classical CPU code with the same functionality. In this comparison a special focus is on the performance gain that can be achieved by applying GPUs. Furthermore, in a second test the GPU algorithm is compared to a well-established, implicit approach for the CPU. Parts of this chapter are already published in [54].

4.1 Stream Processors and GPUs

4.1.1 GPU Hardware Layout

The graphics processing unit (GPU), as the name suggests, was originally designed in the 1980's to assist the central processing unit (CPU) with two dimensional graphical computations. In the 1990's GPUs where extended by the ability to accelerate tasks like three dimensional polygon rendering, texture mapping and lighting. Throughout this evolution the computer games industry among others was the driving force for the development of faster GPUs. Since there is a high degree of parallelism in graphical computations, the GPUs are conceptually laid out with a large number of small computing cores and significantly larger bandwidth than the CPU. The GPU faces millions of relatively small computations of the same type on huge data sets. Thus, GPUs are designed to evaluate one single instruction on many data points simultaneously. This concept is commonly referred to as the *single instruction, multiple data* (SIMD) paradigm. In contrast, modern CPUs follow a *multiple instruction, multiple data* (MIMD) approach yet on a much coarser level. This will be discussed later in more detail.

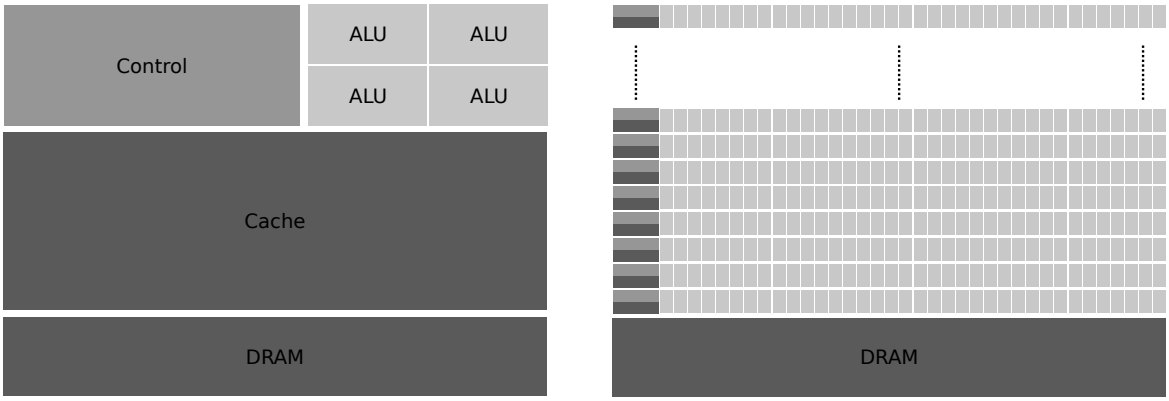


Figure 4.1: Schematic comparison of CPU (left) and GPU (right) hardware layout according to [53]

For the first two decades the concept of GPUs was very static. Rendering graphics on the GPU was divided in several stages which where dedicated to specific parts of the hardware. Given a triangular surface which is to be rendered, the fixed-function graphics pipeline was passed through containing tasks like vertex shading, transformation, lighting, rasterization and finally putting the computed pixel into the frame buffer. The interaction between the system and GPU was realized completely through graphics interfaces like Direct3D and OPENGL and the control of the graphics pipeline was limited.

In 2001 the DirectX 8.0 standard was released which coincided with the development of a programmable vertex and pixel shader on the GPU. The abilities to implement arbitrary code were yet limited. All instructions had to be ported through a graphics application programming interface (API) like OPENGL to the GPU. However, for the first time researchers from various fields became interested in the enormous parallelism and computational power of the GPUs. Especially in the fields of partial differential equations, attention was payed to these developments.

The final step towards a universally applicable computing device was made in 2006 when NVIDIA released the G80 chip. This hardware realizes the concept of unified shaders which means that the stages of the graphics pipeline, as described above, are mapped to an array of unified processors. Simultaneously NVIDIA released the CUDA (*compute unified device architecture*) driver and toolkit, which is basically a C/C++ compiler and runtime environment. This enables the developer to program the GPU in almost plain C/C++ code with some extensions. Through these extensions, the programmer is able to exploit his knowledge about the underlying algorithm and define in which way parallelism of data and executions is handled. Moreover, in the CUDA model arbitrary read and write accesses to the GPU memory are possible unlike through graphics APIs. Another important step is the ability of double precision arithmetic supported in the CUDA architecture. Since for graphics calculations precision is not of importance, the GPU manufacturers did not support double precision in the first place. Still in the latest chip generation called Fermi there is a significant gap between single and double precision performance.

The differences between CPUs and GPUs are schematically depicted in figure 4.1. Since these two processors are designed for different tasks, the hardware architecture follows contrary design philosophies. The CPU chip on the left is laid out for sequential code performance. Most parts of the chip are dedicated to the control of the execution flow. In a program mainly consisting of sequential operations, the CPU is able to break the order of executions whenever it is possible in order to exploit parallelisms. However, this is usually limited to some extend for which reason the number of execution units is significantly smaller than on the GPU side. Another design consideration is the relatively large processor cache buffering data from the global RAM for computations. The idea behind that is to exploit locality of data. On the one hand, data is hold in cache over time for a potential reuse and on the other hand, spatial locality of data is made use of by caching regions of memory. It could be summarized that, since CPUs have to be applicable for all-purpose computations, they are not tuned for one specific kind of task.

On the right hand side of figure 4.1 the hardware of a GPU is outlined. Reviewing their original purpose, GPUs are designed to load a large set of triangles from the RAM, do some independent operations on them, and store the resulting pixel back into the RAM. Thus, the GPU is laid out for small, independent operations. This can be found in the chip design where the area dedicated to execution flow control is much smaller than on the CPU side since operations are expected to be independent by definition. In contrast, most of the chip area is dedicated to floating point operations and only small caches are provided. Another important fact is the connection to the RAM which is much faster on the GPU side resulting in higher memory bandwidth. The link between the global system RAM and the CPU has to satisfy requirements for e.g. backward compatibility of operation systems or the input/output of plugged devices, whereas the GPU is, to some extend, uncoupled from the main system and has to serve only one purpose. Thus, it can be optimized for high bandwidth without regarding compatibility and universality.

This classical point of view does yet not strictly hold for the presence and the future. Modern CPUs are more and more turning into multi core processors since physical barriers do not allow one core to run at arbitrary frequencies. Thus, it rather has to be distinguished between multi core and many core hardware than between classical CPUs and GPUs and here the transition is a smooth one.

To be more specific: The underlying hardware used throughout this work is the NVIDIA Fermi GPU architecture. This design provides up to 16 so-called *streaming multiprocessors* (SM) each featuring 32 cores, which leads to a total number of 512 CUDA processors. Each of these processors has its own *arithmetic logic unit* (ALU) and a *floating point unit* (FPU). A total amount of up to 6 GB RAM is supported. This architecture further provides 64 KB memory per SM which can be configured to be both shared memory or L1 cache [52].

4.1.2 GPU Programming Model

In this section the CUDA architecture is briefly introduced. It has to be seen as one example for a many core or stream processor computing approach, and most of the considerations within this work would also apply for other architectures designed for massively parallel floating point operations.

The CUDA architecture model on the coarsest level distinguishes between a *host*, denoting

the CPU, with the global system RAM and multiple *devices* denoting the GPUs with their attached RAM. Device and host are connected through the PCI express bus. From the programming point of view, the devices are controlled from within threads that are running on the host. In a typical CUDA program each host thread executes the following steps:

- Identify a device with this host and initialize the GPU.
- Allocate memory on the device and upload necessary data to the device.
- Make calls to CUDA kernel functions which execute on the device.
- Perform some asynchronous computations on the host.
- Wait for the device to finish and download the computed results to the host.
- Free the allocated memories on the device.

A major concept in the CUDA programming model are so-called *kernel functions*. They are declared using the CUDA-specific keyword `__global__`. This function defines what all CUDA threads are executing in parallel on the device. The keyword implies that the function is only callable from the host using a special syntax. In contrast, the keyword `__device__` declares a function which is executed on the device and is only callable from within other devices or kernel functions. The call of a kernel function uses an extension to the standard C language in order to declare how many parallel threads have to be started and how they have to be organized. Such calls typically have the following syntax: `NameOfKernel <<< gridDim, blockDim >>> (arguments)`.

The arguments between "<<<" and ">>>" symbols declare the numbering of the parallel threads. In CUDA threads are organized in a hierarchic structure. On top there is a three dimensional *grid* structure where each grid cell contains a *block* structure which is also three dimensional. The elements of such a block are then the threads to be executed. The launch of one CUDA kernel constructs one grid, and the elements (blocks) of the grid are all copies of this kernel. Within the kernel function, or every other device function called in this context, there are variables telling one thread its number and the number of the block it runs in. The variables `threadIdx.x`, `threadIdx.y` and `threadIdx.z` give the location of a thread within its block and `blockIdx.x`, `blockIdx.y` and `blockIdx.z` the location of the block within the grid. The intent of this construction is the distribution of threads over the SMs. Threads of one block are able to share data within the cache efficiently. This will be discussed later in more detail. Another feature is the ability to synchronize the threads of a block by the command `_syncthreads()`, which forces all calling threads to wait for the remaining ones. These aspects of the CUDA model show some similarities to the MPI (*massage passing interface*) model, yet on a much finer level. In CUDA the identification of threads is similar to the "rank" variable in MPI.

From the hardware point of view, the SM gathers its threads in groups of 32 which are called *warps*. A warp executes one common instruction at a time. Whenever the threads within one warp are branched due to a data-dependent conditional statement in the code, branches are executed serialized. This is important to keep in mind since it is crucial with respect to performance. It should be remarked that the CUDA architecture follows a so-called

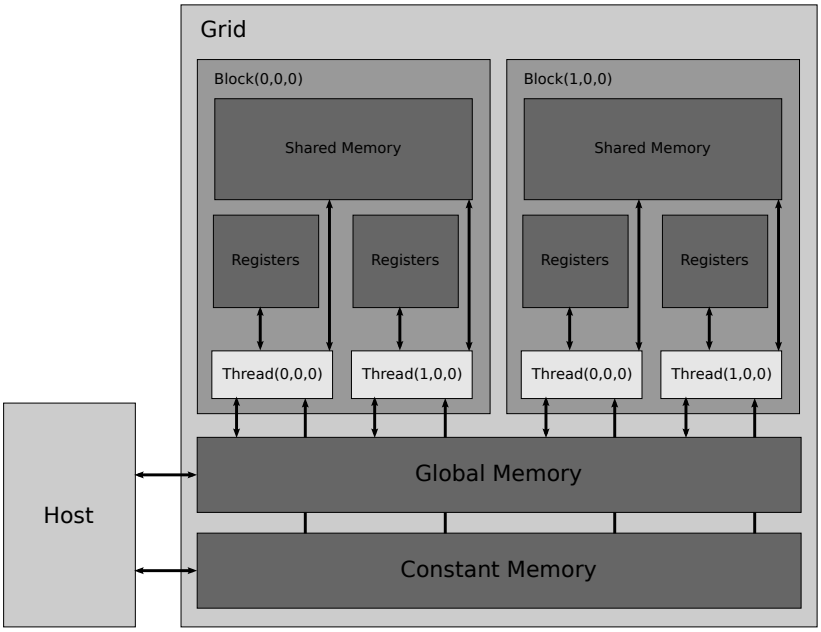


Figure 4.2: CUDA memory model according to [51]

SIMT (*single instruction, multiple threads*) model which slightly differs from the SIMD model. While in SIMD models the width of the vector is exposed to the software, in the SIMT model instructions specify the execution and branching behavior for a single thread.

Besides organization of parallel threads and optimization of instruction throughput, the memory management is very important in terms of high performance computing. Especially in the world of GPUs and stream processors with their very high floating point performance, it is crucial to feed the hundreds of computing cores with data. Without avoiding high latencies and idle times due to misuse of the device memory, the GPU code will likely not perform better than a conventional CPU code or even worse. Here two problem classes can be distinguished. On the one hand, problems where arithmetic computations are dominant, and on the other hand, problems which are dominated by memory operations. As an example for the first class the rendering of fractal images or the Monte Carlo simulations could be seen. Problems where there are pointwise independent, possibly arithmetic expensive operations with only few data input and output. Here the GPU with its hundreds of lightweight processor cores, optimized for floating point operations, shows very high speedups compared to a CPU. Moreover, in these situations parallelism is straightforward and a programmer does not have to go too deeply into the hardware specific optimization techniques. In contrast, solving PDEs in non-trivial domains usually leads to discretizations on unstructured grids, like in this work, resulting in a mostly memory dominated problem. In that case, the gained speedup against conventional CPU codes crucially depends on the utilization of the different types of memory and the caching strategy.

In the CUDA architecture there are several different types of memories with different properties (cf. figure 4.2). The major one is the DRAM of the GPU, which is called the *global memory*. In the NVIDIA Fermi architecture DRAM offers up to 6 GB space. This memory is

comparable to the CPUs DRAM and is used in a similar fashion. The global memory is both readable and writable to the CPU and the CUDA threads, but reading data from within CUDA threads is connected to high latencies. It is common practice in GPU programs that the host thread on the CPU allocates space in the global device memory and then copies all needed data from the host memory to the device global DRAM. This is usually executed in an initial phase of the application. The data transport is conduced over the PCI express bus and could be the bottle neck for the performance which will be discussed later. A pointer to this allocated space is then transferred to the GPU as an argument in the kernel call.

Similar to the global memory is the *constant memory* on the device. It is both readable from the host and device but only writable from the host, as the name says. Constant memory features lower latencies than global memory and is thus interesting for data which does not change through the whole application and is used frequently. These two memories are persistent for the live-time of the application.

In order to overcome the high latencies of global memory, the GPU features very fast on-chip memory, namely *registers* and *shared memory*. Variables hold in the registers are visible only to the owner thread and can be accessed very efficiently. This is where temporal variables, used within the computations of one thread, are usually placed. On the Fermi architecture there is additionally 64KB of fast memory which can either be configured as 48KB shared memory and 16KB L1 cache or 16KB shared memory and 48KB L1 cache.

The extensive use of this shared memory is essential for many applications. It is allocated from within a device function and all threads of the owner block have read and write permission. The shared memory is not only attractive because of the possibility to share and reuse data within a thread block but also because of low latencies resulting in efficient data accesses. Alternatively this on-chip memory is configured as L1 cache, which offers a great performance gain against direct accesses to the DRAM. If one thread block spreads over more than one warp, accesses to shared memory are handled by the warp scheduler and are not predictable for the programmer. In that case the use of the barrier function `_syncthreads()` is obligatory in order to guarantee proper behavior of the code. Both the shared memory and the registers are persistent for the life-time of the owner block and thread, respectively.

4.1.3 Performance Considerations

With the methods presented in this section so far and a naive C code on the GPU there will hardly be any performance gain against conventional CPU algorithms. As mentioned earlier, performance heavily depends on the utilization of memory and caching strategies. There are two critical aspects which have to be considered. First, the connection between host and device. Since there are high latencies and a low bandwidth in these transactions through the PCI express bus, they should be avoided whenever possible. In the ideal case a large block of data is transferred to the device in an initial stage of the application. Arithmetic intensive operations are performed on the device and at the end of the application a large block of resulting data is transferred back to the host.

Otherwise it can be made use of so-called *page-locked memory* on the host. This is a special type of memory the host thread allocates. It enables asynchronous data copies between host and device, which means that transferring data can be overlapped with kernel calls and

computations on the device.

Another critical performance consideration is the connection to global memory. Since the bandwidth to the DRAM of the device is comparatively low, global memory accesses should be avoided whenever possible by applying the fast on-chip memory or the registers. Yet, at least once data has to be loaded and stored to the DRAM. On Fermi architectures global memory accesses are cached in L1 cache of the SM, which has a cache line width of 128 byte. Concurrent accesses of the threads of one warp (32 threads) are split into the number of cache lines necessary to serve all accesses. If special alignment conditions are fulfilled, the memory transactions of all threads of a warp are *coalesced* into one operation. Due to high latencies to the global memory, coalescing is essential - not to say obligatory - in order to gain a performance superior to conventional CPU codes. The choice of 128 byte cache line width and the number of 32 threads indicates that GPUs are tuned for operations in single precision resulting in 4-byte words. For the case of scattered data accesses the Fermi architecture also allows to cache 32 byte segments into L2 cache. Yet, for the purpose of this work, the default strategy is sufficient.

As mentioned earlier the alignment of memory accesses is important in order to get them coalesced. The best layout is achieved when the threads within a warp request a block of memory such that the starting address is a multiple of 128 byte. Misaligned or strided accesses may result in a significant higher number of memory transactions. However, it is not necessary that the threads follow a specific order. The mapping between the threads and the accessed memory addresses can be arbitrarily permuted. For such aligned access patterns the vendor reports bandwidth of up to 130 GB/s on the Fermi architecture which is significantly larger than the bandwidth obtained between CPU and RAM. It is common practice to load data from the global memory to shared memory in a block-wise fashion and then exploit the high bandwidth and low latencies of the on-chip memory.

In order to enable simultaneous accesses of the threads of one block, shared memory is divided into equally sized *memory banks*. Thus, for loads and stores, whose addresses fall into n distinct memory banks, the achieved effective bandwidth is n times higher than the bandwidth of one bank. If two threads request addresses within one bank, this leads to a so-called *bank conflict* which is overcome by serialization of the accesses. In the underlying hardware there are 32 banks, and shared memory is organized such that successive 32-bit words fall into successive memory banks. Usually memory is requested as 32-bit words (e.g. `float` or `int`) or larger, however, if e.g. 8-bit words (e.g. `char`) are requested per thread, one would expect bank conflicts. Yet, the bank conflict is avoided by a so-called *broadcast mechanism*. Since multiple threads access addresses within one word in one memory bank, this word is broadcasted to all of these threads resulting in a single transaction. The case of 64-bit data types (e.g. `double`) is treated specifically by the hardware. If 64-bit words are accessed such that successive threads access successive words, there are no bank conflicts.

It is also a common situation that all threads of a block request the same address at a time. This does not lead to bank conflicts due to the broadcast mechanism mentioned earlier and results in only one transaction.

When dealing with problems arising in numerical mathematics, most of the involved operations are of linear algebra type. The parallelization of operations on dense matrices and vectors perfectly fits into the CUDA programming model and high speedups can be achieved. Since

most operations of this type have as many input as output variables, distributing work to the CUDA threads mostly follows the one thread per input and output principle. This is more complicated for *reduction type operations* such as summing up the elements of a vector or forming L_2 and L_∞ norms, for instance.

The reduction is heavily memory dominated since there is usually only one operation per element such as addition, maximum or minimum. Here typically a long vector of input data is given, but the output is a single scalar, which makes it hard to keep as many threads of the GPU busy as possible. Thus, this is a good example to see how parallelization on stream processors works. Since the code developed for this work is based on an explicit time stepping scheme, the calculation of norms, for instance the norm of the residual, is an essential building block.

```
1  float *B;
2  int SIZE = ...;
3
4  // do something
5
6  float sum = 0;
7  for(int i=0; i<SIZE; i++)
8  {
9      sum += B[i];
10 }
```

Listing 4.1: Serial code example for reduction operation

Code example 4.1 shows a very simple and straightforward approach to compute the sum of a vector written in the C language. This is the standard procedure on a single processor core. Since the value of `sum` in each iteration depends on its predecessor, it is non-trivial to distribute work over hundreds of threads. The best practice approach is illustrated in code example 4.2. Here it is assumed that the length of the vector `B_host` is a multiple of the maximum CUDA block size 512.

Lines 13-25 show the kernel function for the reduction with summation. First, a block of 512 elements is loaded from the global memory of the GPU into the shared memory since these values will be read and updated multiple times. This operation is most efficient on the GPU due to coalesced memory accesses. Then half of the threads of this block load the second half of the values from shared memory and update the first half by summation. In the next iteration this is performed by a quarter of the threads and so on until there is only one value left. For this approach, it is assumed that the blocksize is a power of two, e.g. 512 in the example. If not, the input vector could be padded with zeros up to the next larger power of two. The operation >>= is a bitwise shift which is the most efficient variant to divide by two. Shared memory accesses arising in this kernel are completely free of bank conflicts. Although in this approach there are many idle threads due to the tree-like structure, in [53] bandwidths are reported for a slightly optimized version of this kernel which are close to the theoretical peak performance of the hardware.

At the end of the iteration `u[0]` contains the sum over the 512 elements. This sum is then

```
1  float *B_host,*B_dev;
2  int SIZE = 2^N;
3  int BLOCKSIZE = 512;
4
5  // do something
6
7  cudaMemcpy(B_dev, B_host, SIZE*sizeof(float), cudaMemcpyHostToDevice);
8  dim3 grid(1,1,1);
9  dim3 block(BLOCKSIZE,1,1);
10 reduce_kernel <<<grid, block>>> (B_dev);
11 cudaMemcpy(B_host, output, 1*sizeof(float), cudaMemcpyDeviceToHost);
12
13 __global__ void reduce_kernel(float* B, float* output)
14 {
15     int tx = threadIdx.x;
16     __shared__ float u[BLOCKSIZE];
17     u[tx] = B[tx];
18
19     for(int stride = BLOCKSIZE/2; stride>0; stride>>=1)
20     {
21         if(tx < stride) u[tx] += u[tx + stride];
22         __syncthreads();
23     }
24     if(tx == 0) output[blockIDx.x] = u[0];
25 }
```

Listing 4.2: Example for a reduction operation on GPUs

written into the output field `output` at the position of the block index. This procedure enables to apply the reduce kernel recursively in order to work on larger fields. Lines 7-11 illustrate the preparation for the kernel invocation in line 10. Here the input field is copied to the GPU and a grid containing one block is generated. After the kernel terminates, the first element in `output` is copied back to the host memory.

4.2 Discontinuous Galerkin on Stream Processors

Reviewing the discontinuous Galerkin method as introduced in section 2.1, it turns out that there is a lot of parallelism in the executions. The concept of higher order DG discretizations offers structure to the discrete problem which is not present in lower order methods. Even on unstructured grids a block-structured system matrix can be obtained. Yet, it is not straightforward how to compare a lower order discretization and a higher order one with respect to the quality of the approximation. On the same underlying mesh the higher order method obviously offers a richer ansatz space resulting in a higher resolution. However, especially in the unstruc-

tured higher dimensional case, there are no precise estimations known of how many elements are needed in a higher order discretization in order to reflect the approximation quality of a lower order one on a finer grid. It is thus reasonable to compare different methods by the degrees of freedom in the resulting discrete system and not by the number of elements.

Due to the higher order elements, the discontinuous Galerkin method defines an arrangement and alignment of the degrees of freedom which is very close to the considerations made in the previous section concerning the programming model for stream processors. The unknown variables, namely the nodal values at the interpolation and cubature points belonging to one higher order element, are naturally agglomerated in the memory. In contrast, a lower order discretization, like the finite volume method with constant basis functions (according to $P = 0$ DG), typically has only one variable per cell leading to a scattered memory layout on unstructured grids, which hardly fits in the concept of stream processors. For example in [55] a case study with the Euler equations on a GPU cluster is presented. However, it has to be kept in mind that increasing the polynomial order and decreasing the number of discretization elements leads to the boundary representation problem discussed in section 3.1.

In the DG method several levels of parallelism can be distinguished. On the coarsest level the spatial domain can be partitioned like in the finite volume or finite element method. Partitioning the mesh means that all elements are divided in groups and attached to the single compute nodes. A typical procedure for this is to apply algorithms from graph theory, which is described in [56] and gathered in the ParMetis toolbox. This tool is applied in the code underlying to this work. In a first step the mesh is interpreted as a graph where discretization elements are nodes and the adjacency list defines the edges in the graph. Then a *p-way graph partitioning* is evaluated where p is the number of compute nodes in the cluster. The ParMetis toolbox optimizes this partitioning with respect to load balancing and minimal cuts. This means that each compute node is assigned an almost equal number of elements in order to avoid idle times. Further, cuts between partitions are chosen such that the number of cut edges is as small as possible since this minimizes communications between compute nodes, which is typically an expensive operation.

Clearly the partitioning is not for free. Fluxes between elements attached to different compute nodes have to be evaluated. With respect to this communication, the DG method is well suited due to the double-valued states at element interfaces. As discussed earlier for the flux computation and integration, only the traces of the unknown on common interfaces with adjacent cells have to be known. As a consequence, only the values interpolated to the surface cubature nodes have to be communicated to adjacent compute nodes and not the full set of nodal values. This data transfer is processed through the message passing interface (MPI) which is the standard approach. In [35] the parallelism on this level is intensively investigated.

On a second level of parallelism the update calculation for the cells can be processed simultaneously. In the explicit Runge-Kutta method applied here, the calculation of the non-linear operator $\mathcal{N}_h$ in equation (2.64) and particularly in (2.39) and (2.78) is the most expensive task. Due to the explicit nature of the time stepping, this calculation depends only on the function state from the previous time step. $\mathcal{N}_h$ consists mainly of products with element-specific, dense matrices allowing to handle all elements in parallel.

Furthermore, a third and very fine level of parallelism can be distinguished. Within one element the non-linear flux functions F^C and F^D (cf. (2.46) and (2.47)) have to be evaluated

per cubature-node. These arithmetic expensive computations, due to their rational nature, can also be handled simultaneously. Moreover, this level of parallelism is the most interesting one since with increasing polynomial order P the number of interpolation points increases with $\mathcal{O}(P^3)$. Also the number of cubature points per element and per element-face drastically increases due to the necessarily curved elements in the vicinity of boundaries.

Putting this together shows that the concept of the discontinuous Galerkin method perfectly fits into the CUDA hardware and programming model as illustrated in section 4.1.2. Assuming a cluster of p nodes each equipped with one GPU a p-way mesh partition is evaluated and distributed to the GPUs. On each GPU a CUDA kernel is started which computes the right hand side of equation (2.39) or (2.78). Roughly speaking, the kernel invocation generates a grid containing as many blocks as there are elements in the mesh, and each block contains as many threads as there are cubature nodes in the elements.

In order to maximize the bandwidth and the instruction throughput, the calculation of the right hand side in element number k is split into volume contributions

$$Q_{k,x_m} \leftarrow S_{k,x_m} \mathcal{I}_V \epsilon_k U_k \tag{4.1}$$

$$U_k \leftarrow \sum_{m=1}^{3} S_{k,x_m} \left(F_m \left(\mathcal{I}_V U_k \right) + \mathcal{I}_V \epsilon_k Q_{k,x_m} \right) \tag{4.2}$$

and surface contributions

$$Q_{k,x_m} \leftarrow M_{\partial\Omega_k} \frac{1}{2} \left(\epsilon_k^- U_k^- + \epsilon_k^+ U_k^+ \right) n_{x_m} \tag{4.3}$$

$$U_k \leftarrow M_{\partial\Omega_k} \left[H\left(U_k^-, U_k^+, \vec{n} \right) + \frac{1}{2} \left(\epsilon_k^- Q_k^- + \epsilon_k^+ Q_k^+ \right) \cdot \vec{n} \right] \tag{4.4}$$

resulting in four kernel functions. A fifth kernel interpolates the nodal values to the surface quadrature nodes by evaluating the product of $\mathcal{I}_S U_k$. Finally, a sixth kernel function performs the inner products, needed for the evaluation of ϵ_k in (2.81), which works as illustrated in code example 4.2. In order to coalesce the global memory accesses on the GPU, the memory has to be aligned such that the address of the first value corresponding to each element is aligned to a 128-byte segment. Thus, it is required that the unknown variables are stored in blocks of 16 in double precision and 32 in single precision, which is achieved by padding zeros to the end of each field

$$U_k = \begin{bmatrix} \rho_0, & \rho_1, & \dots, & \rho_{N_P-1}, & \underbrace{0, & \dots, & 0,}_{\text{padding}} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \rho E_0, & \rho E_1, & \dots, & \rho E_{N_P-1}, & 0, & \dots, & 0 \end{bmatrix}. \tag{4.5}$$

The resulting blocksize for nodal values at interpolation points and volume/surface cubature nodes in double precision is given by

$$b_P = \left\lceil \frac{N_P + 15}{16} \right\rceil, \quad b_S = \left\lceil \frac{4N_S + 15}{16} \right\rceil, \quad b_V = \left\lceil \frac{N_V + 15}{16} \right\rceil. \tag{4.6}$$

P	1	2	3	4	5
interpolation points (N_P)	4	10	20	35	56
volume cubature points (N_V)	5	15	35	70	126
surface cubature points ($4 \cdot N_S$)	$4 \cdot 4$	$4 \cdot 7$	$4 \cdot 12$	$4 \cdot 16$	$4 \cdot 25$

Table 4.1: Number of interpolation and cubature nodes for different polynomial degrees

Table 4.1 shows the number of interpolation and cubature nodes which are used within this work. The element-specific stiffness matrices S_x, S_y and S_z are also padded to a blocksize of b_P resulting in

$$S_x = \left[\begin{array}{cccccc} s^x_{(0,0)}, & s^x_{(1,0)}, & \dots, & s^x_{(N_P-1,0)}, & \underbrace{0, \dots, 0,}_{\text{padding}} \\ s^x_{(0,1)}, & s^x_{(1,1)}, & \dots, & s^x_{(N_P-1,1)}, & 0, \dots, 0, \\ \vdots & \vdots & & \vdots & \vdots \\ s^x_{(0,N_V-1)}, & s^x_{(1,N_V-1)}, & \dots, & s^x_{(N_P-1,N_V-1)}, & 0, \dots, 0 \end{array} \right] \tag{4.7}$$

and similarly the interpolation operator

$$\mathcal{I}_V = \left[\begin{array}{cccccc} \iota_{(0,0)}, & \iota_{(1,0)}, & \dots, & \iota_{(N_V-1,0)}, & \underbrace{0, \dots, 0,}_{\text{padding}} \\ \iota_{(0,1)}, & \iota_{(1,1)}, & \dots, & \iota_{(N_V-1,1)}, & 0, \dots, 0, \\ \vdots & \vdots & & \vdots & \vdots \\ \iota_{(0,N_P-1)}, & \iota_{(1,N_P-1)}, & \dots, & \iota_{(N_V-1,N_P-1)}, & 0, \dots, 0 \end{array} \right]. \tag{4.8}$$

These matrices are transposed in the global memory, which is important for the matrix vector products and will be discussed later. Due to the quadrature approach usually $N_V > N_P$ and $4N_S > N_P$ holds, meaning that there are more volume and surface cubature than interpolation nodes. Thus, one thread per input and output cannot be applied. Here the calculation of the pure hyperbolic volume contribution $\sum_{l=1}^{3} S_{k,x_l} F_l \left(\mathcal{I}_V U_k\right)$ will be discussed as an example. The other kernels and the extensions to the viscous fluxes work very similar and use mainly the techniques presented here.

The kernel is launched generating a grid with a block for each element k with a total number of b_V threads which is the number of non-linear flux evaluations F. Let these threads be denoted by $\{t_0, \dots, t_{N_V-1}\}$. Further, two arrays of shared memory are allocated: `U` of size $m \cdot b_V$ where m is the number of equations (5 in the Euler case) and an array `sum` of size $m \cdot b_P$. In a first step the unknown variables U_k are cached into the shared memory array `U`. These memory accesses are coalesced due to the padding and the thread alignment which is given by $t_i \rightsquigarrow \rho_i$ for $i = 0, \dots, b_P - 1$. The other unknown fields are loaded in the same manner. Then each thread t_i requests register variables `I` and `S_1`,...,`S_m`. In `I`, t_i successively

loads $\iota_{(j,i)}$, which is the j-th column of $\mathcal{I}_V$, multiplies with $\rho_j, \ldots, \rho E_j$ and adds the result to `S_1,...,S_m`. Hereby it is ensured that $\mathcal{I}$ is loaded coalesced from the global memory. These values are kept in registers since they are used only once. Thus, the threads t_i hold one column of $\mathcal{I}_V$ in registers and scale it with the unknown variables from shared memory. Note that all threads simultaneously request the same shared memory address, which does not result in bank conflicts and serialization due to the broadcast mechanism. At the end of this multiplication the shared memory, holding the variables U_k, is overwritten with the values from the registers of the threads holding $\mathcal{I}_V U_k$.

The next step is the evaluation of the divergence consisting of applying the non-linear flux functions F_l and multiplying with the element-specific stiffness matrices S_l. Evaluating the non-linear flux function is one of the key skills of stream processors where the gained speedup of the GPU code mainly comes from. The multiplication with the stiffness matrices is similar to the interpolation operation to the volume cubature nodes. However, each thread t_i is equipped with three registers, one for each of the three stiffness operators. This avoids that unknown variables $\mathcal{I}_V U_k$ have to be loaded from the shared memory three times. Since the number of volume cubature nodes is in general much larger than the number of interpolation points, there are too many threads for the matrix-multiplication scheme as described above. Thus, the threads t_i are distributed over $\left\lfloor \frac{b_V}{b_P} \right\rfloor$ columns of S_x, S_y and S_z which can then be handled in parallel. This results in $\left\lfloor \frac{b_V}{b_P} \right\rfloor$ groups of b_P threads holding partial sums of $\sum_{l=1}^{3} S_{k,x_l} F_l(\mathcal{I}_V U_k)$. Again the shared memory field `U` is used to store these partial sums which are then summed up by the first $t_0, \ldots, t_{b_P-1}$ threads and finally stored back to the global memory into a field holding the right hand side of (2.39).

The evaluation of the surface kernel is only different with respect to two aspects. First, data is scattered through the global memory. The solution trace at surface cubature nodes from adjacent elements is most likely located in other regions of the memory due to the unstructured mesh. However, the unknown values, interpolated to surface cubature nodes, can be organized in a field which is padded according to b_S. By this it can be ensured that the values corresponding to one face of the tetrahedron fall within one 128-byte segment and the memory transaction is coalesced. Another difference to the volume kernel is the additional information which is needed. For each cubature node on the surface it has to be known which node is the coinciding partner. Due to the symmetric cubature rule for triangles, described in section 2.1, it is ensured that this partner exists. Yet, the unstructured nature of the mesh allows a twist in the connection between two adjacent tetrahedra. This results in several possible permutations in the assignment of cubature nodes between two tetrahedral surface triangles. Further, for each surface node the normal vector has to be known for the flux evaluation. Because of the curved elements, the normal vector field varies between the nodes of one face of the tetrahedron. In order to maintain coalesced memory patterns, this information is packed into one field which is also padded according to b_S.

The inner products for the shock detector (2.81) can be seen as a special case in which the reduction operation follows a matrix-vector product. First, one component of the solution (here ρ) is cached into a shared memory field `U1`. The size of the field is chosen to be the next greater power of two and it is initialized with zeros for the reduction operation to work most efficiently. This seems to be a wasting of memory, however N_P is most often relatively small such that

the next power of two is close. From within this field a copy into another shared memory field `U2` is performed. Then the matrix product with $M_{k,\text{red}}$ is performed as described above and the result is written back from the registers to `U1`. After that `U1` is component-wise multiplied with the ρ values from `U2` and the reduction operation is performed on `U1`. The nominator of S_k is then extracted from `U1[0]`. Again the ρ values from `U2` are copied into `U1` and in `U2` the product with M_k is evaluated. Finally, the resulting values in `U2` are multiplied component-wise with `U1` and by the sum-reduction the denominator for S_k is obtained. This procedure seems not to be straightforward in order to evaluate a mathematically simple expression like S_k, yet data transfers to the device's global memory are minimized by this.

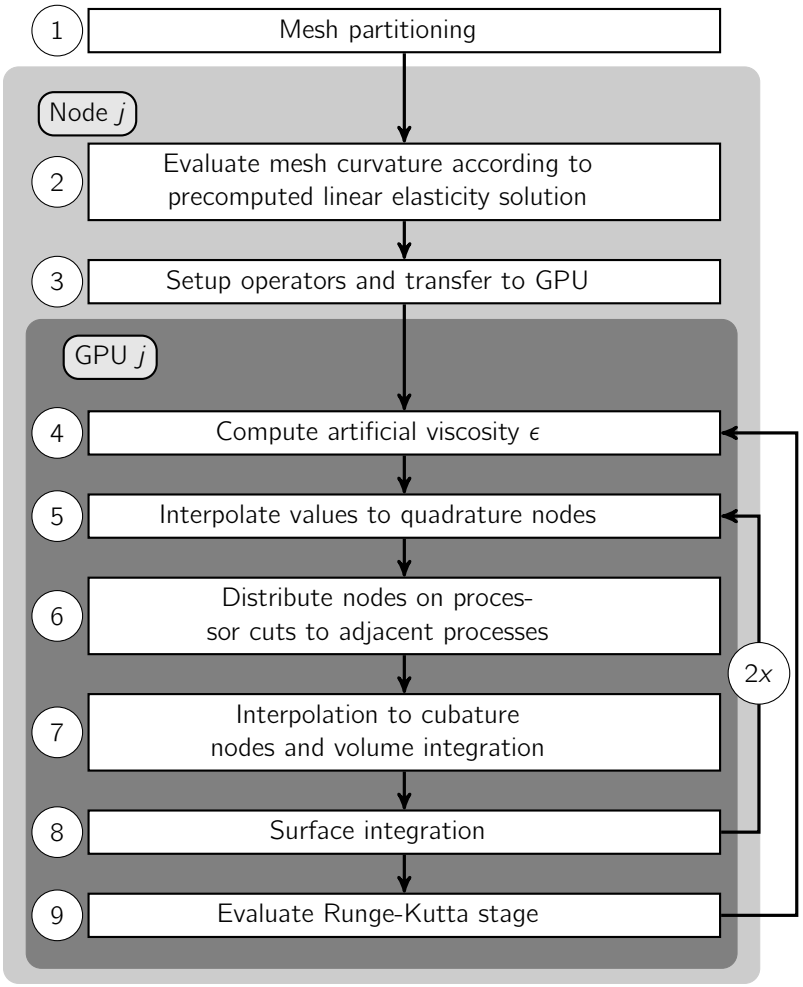


Figure 4.3: Algorithm execution on one compute node

Figure 4.3 illustrates the overall execution cycle on a compute node with rank j to which GPU number j is attached. The whole program mainly follows the paradigms presented in section 4.1.2. Since data transfers from and to the device are critical for the performance, the time stepping is designed in order to completely work on the GPU. In step one each processor reads in

a part of the mesh file and the ParMetis library swaps elements between the compute nodes for a better load balancing and minimal cuts. After that the straight-sided mesh is curved according to the linear elasticity solution. In step three the element specific operators are composed as derived in section 2.1. These operators are then transferred to the GPU and a workspace in the global device memory is allocated. The steps 4-9 contain the Runge-Kutta time stepping and run completely on the GPU without interaction to the main system. Yet, if the algorithm runs on a cluster of multiple GPUs, unknown values, interpolated to the surface cubature nodes on the processor cuts, have to be communicated in step 8. For this purpose, these values have to be downloaded from the GPU, passed through MPI and uploaded to the device of the adjacent process. This MPI communication does not consume additional time since the kernel invocation in step 7 can be performed simultaneously. Thus, the MPI communication time is hidden by the evaluation of the volume integrals. The feature of asynchronous data copies was not yet available in CUDA when this work was started, which will be illustrated in the performance analysis at the end of this chapter.

In the case of the artificial viscosity approach or the Navier-Stokes equations, which are both of the form (2.78), the steps 4-8 have to be executed twice: once for the auxiliary variable Q with kernel functions according to (4.1) and (4.3) and once for the actual solution U according to equations (4.2) and (4.4). Finally, in step 9 the Runge-Kutta time step is evaluated which is essentially done by adding two vectors leading to high efficiency on GPUs. Steps 4-9 are repeated until the time integration has reached a desired final time or the norm of the computed right hand side is below a specified tolerance. After that the computed solution is copied back from the device to the RAM and can be post-processed. As mentioned earlier, the evaluation of a norm is a critical task in a parallel application. In the situation of a cluster of GPUs a norm has to be evaluated in two steps. First, each single GPU has to perform a reduction as described in code example 4.2 and than MPI has to perform the reduction over each process. This makes the evaluation of norms to a computationally expensive task. In order to overcome this issue, the norm of the residual is not evaluated after each Runge-Kutta time step which will be discussed in the next section.

4.3 Numerical Tests of the GPU Algorithm

4.3.1 Flow Simulations

In this section some representative numerical test cases are discussed. The techniques presented in chapter 2 are applied within the GPU framework described in section 4.2. For this purpose, the boundary and freestream conditions $(\rho_\infty, \rho u_\infty, \rho v_\infty, \rho w_\infty, \rho E_\infty)$ have to be specified. The ∞ subscripts indicate that this state is assumed in an sufficiently large distance to the obstacle in the flow. "Sufficiently large" in this context means that the numerical solution in the vicinity of the obstacle is not disturbed by influences of the boundaries.

The flow condition is chosen according to the standard atmosphere at sea level published by the *International Civil Aviation Organization* (ICAO) in 1976 which states a temperature of 15 °C, a pressure of $p_\infty = 101{,}325$ Pa and a density of $\rho_\infty = 1.225\ \frac{\text{kg}}{\text{m}^3}$. From these values the freestream condition can be derived by specifying a freestream Mach number M_∞ and two angles of attack α in the xy-plane and β in the xz-plane. Together with the adiabatic index

(cf. section 2.2) the local speed of sound

$$a_\infty = \sqrt{\gamma \frac{p_\infty}{\rho_\infty}} \tag{4.9}$$

is obtained. From this the velocities in the three coordinate directions are given by

$$\begin{pmatrix} u_\infty \\ v_\infty \\ w_\infty \end{pmatrix} = M_\infty a_\infty \begin{pmatrix} \cos(\alpha)\cos(\beta) \\ \sin(\alpha) \\ \cos(\beta) \end{pmatrix}. \tag{4.10}$$

With these velocities a total enthalpy can be formulated as

$$H_\infty = \frac{a_\infty^2}{\gamma - 1} + \frac{1}{2}\left(u_\infty^2 + v_\infty^2 + w_\infty^2\right) \tag{4.11}$$

which leads to the total energy

$$\rho E_\infty = \rho_\infty H_\infty - p_\infty \tag{4.12}$$

completing the vector of freestream conserved variables. The initial condition to the hyperbolic system (2.45) is then chosen constantly as $U_0 = (\rho_\infty, \rho u_\infty, \rho v_\infty, \rho w_\infty, \rho E_\infty)$. However, this initialization leads to large gradients in the first iterations of the time stepping scheme in the vicinity of the obstacle. Yet, it is the only information that can be predisposed.

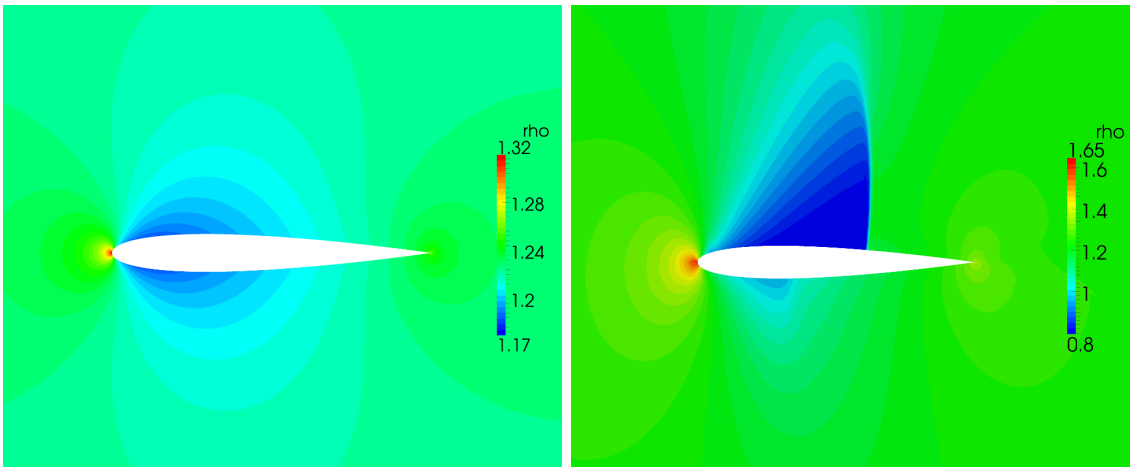
Another building block of a computer-based simulation is the definition of a tolerance when the simulation should stop. Since all test cases within this work, except one, are steady state calculations, it is obligatory to define a point where no improvement towards the solution is possible anymore. There are two different choices. First, the L_2-norm of the right hand side in equations (2.39) or (2.78) is considered. This scalar value is normalized by the value obtained with the initial conditions U_h^0 and it is then compared to a specified tolerance. A second kind of measure is to observe a representative target functional. In CFD the drag functional is commonly used. This is an integrated quantity over the surface of an obstacle in a fluid flow which will be specified in equation (5.19). If this value only differs up to a given tolerance between two iterations, the simulation is stopped.

The first test case is the so-called NACA0012 airfoil which is a two dimensional shape out of the NACA four digit series [57]. Since the element type chosen for the solver in this work are tetrahedra, the two dimensional profile placed in the xy-plane is stretched in z-direction resulting in a so-called staggered grid. The underlying geometry and mesh curvature is described in section 3.2. On the left hand side of figure 4.4 a subsonic flow past the NACA0012 is depicted. Here the freestream Mach number is chosen to be $M_\infty = 0.38$ and the angle of attack is $\alpha = 0^\circ$ resulting in a symmetric density distribution due to the symmetric profile. Both, the curvature and the DG scheme, are chosen to be $P = 5$ which leads to a total number of 143,752 nodes in the discretization. The number of degrees of freedom may appear surprisingly large for a subsonic flow, however this is a three dimensional mesh and the polynomial degree is globally constant. In general, the best numerical results are achieved using the same polynomial degree in the linear elasticity solver as in the DG scheme. These computations are performed using equation (2.39) for pure hyperbolic PDEs. The time stepping is executed until the normalized

L_2-norm of the residual reached a tolerance of 10^{-8} after 81,000 time steps with an increment of $4 \cdot 10^{-4}$.

Figure 4.4 (b) shows a transsonic flow on the same NACA0012 geometry. The freestream Mach number is chosen to be $M_\infty = 0.8$ and an angle of attack $\alpha = 1.25°$ leading to a strong shock on the upper surface and a weak shock on the lower surface. The time stepping converged like in the subsonic case with a tolerance of 10^{-8} of the normalized $L2$-norm of the residual after 209,000 iterations with $\Delta t = 8 \cdot 10^{-5}$. For these computations the artificial viscosity model in equation (2.78) is applied together with the shock detector (2.83). The detector is configured with $\epsilon_0 = 1.5 \cdot 10^{-3}$, $\kappa = 1.1 \cdot 10^{-5}$ and $s_0 = -8 \cdot 10^{-6} \log_{10}\left(\frac{1}{P^4}\right)$. It should be remarked that these constants where found experimentally. Moreover, this artificial viscosity model is very sensitive with respect to the configuration of the shock detector. Whenever the mesh is slightly modified in the test cases, a different set of constants has to be chosen. Yet, this model is widely used throughout literature.

Figure 4.5 shows the underlying grid and the values of the shock detector on the front plane of the staggered, three dimensional mesh. Together with $P = 3$ elements this leads to a total number of 1,671,840 interpolation nodes. In this test case the artificial viscosity approach was not sufficient to produce a sharp shock profile like in the original work [4]. A drastically refined mesh and a lower degree of basis polynomials had to be chosen in order to avoid stability problems which might stem from the explicit time stepping used within this work. Figure 4.6 shows the graph of the pressure coefficient C_p over the NACA0012 profile. Since the mesh extends over 25% of the chord length in the third coordinate direction, the C_p curve is evaluated at a slice through the center at 12.5%. Here it can be seen that despite the artificial viscosity small oscillations arise in the vicinity of the strong shock on the upper surface of the profile. These oscillations drastically increase on coarser meshes and lead to instabilities in the explicit time stepping.



(a) Subsonic flow, $M_\infty = 0.38$, $\alpha = 0°$ (b) Transsonic flow, $M_\infty = 0.8$, $\alpha = 1.25°$

Figure 4.4: Flow past NACA0012 airfoil with $P = 5$ elements on the left and $P = 3$ elements on the right, density distribution

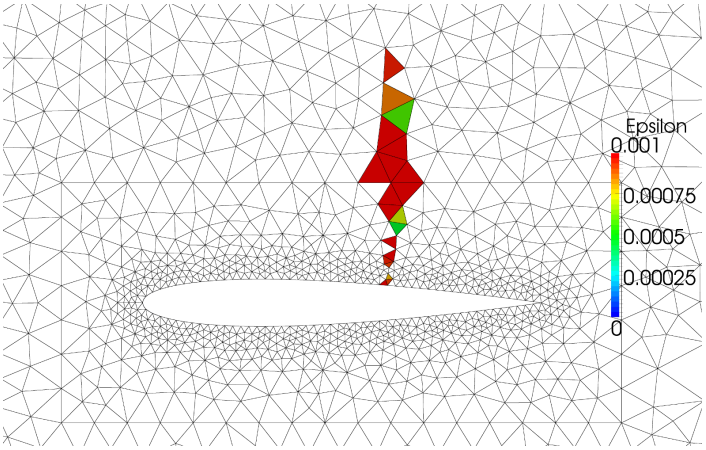


Figure 4.5: Artificial viscosity at front surface of three dimensional, stretched NACA0012 profile, transsonic test case from figure 4.4(b), $t \to \infty$

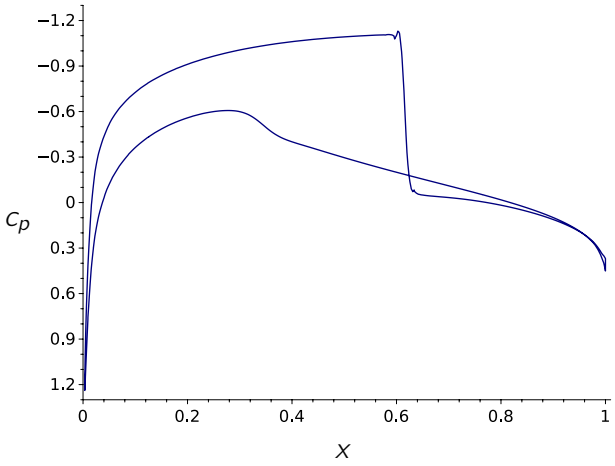


Figure 4.6: c_p distribution over the NACA0012 airfoil

The next test case is the ONERA M6 wing which is a well-known and established test for various numerical methods throughout literature. The original experiment and the geometry is described in [58]. This is a transsonic flow with Mach number $M_\infty = 0.8395$ and an angle of attack $\alpha = 3.06°$ leading to a strong, so-called λ-shock wave (based on its shape) on the upper surface. In the original experiment, the tip of the wing is a round closure, whereas here the wing is simply cut off. This allows a straightforward NURBS representation of the geometry and leads to much better results in the mesh generation process.

Figure 4.7 shows the density distribution on the upper surface of the ONERA M6 wing. The black lines in the background visualize 40 equidistant isolines of the density in the interval $[0.65, 1.5]$. In the computations this boundary is modeled as a symmetry wall which is realized by $U_h^+ = U_h^-$. The artificial viscosity model is configured with $\epsilon_0 = 3 \cdot 10^{-3}$, $\kappa = 1.5 \cdot 10^{-5}$

and $s_0 = -8 \cdot 10^{-6} \log_{10}\left(\frac{1}{P^4}\right)$. The necessary amount of viscosity is significantly larger than in the NACA0012 case which could be explained with the sharp edge at the tip of the wing. In the numerical tests the shock detector usually had its maximum value at this position where large gradients arise in the solution. The computations for this test case converged to a relative residual of 10^{-8} after 512,000 iterations and a time increment of $\Delta t = 1 \cdot 10^{-4}$. The underlying mesh is sketched in figure 3.3 and consists of 74,997 elements leading to a total number of 1,499,940 interpolation points in the discretization.

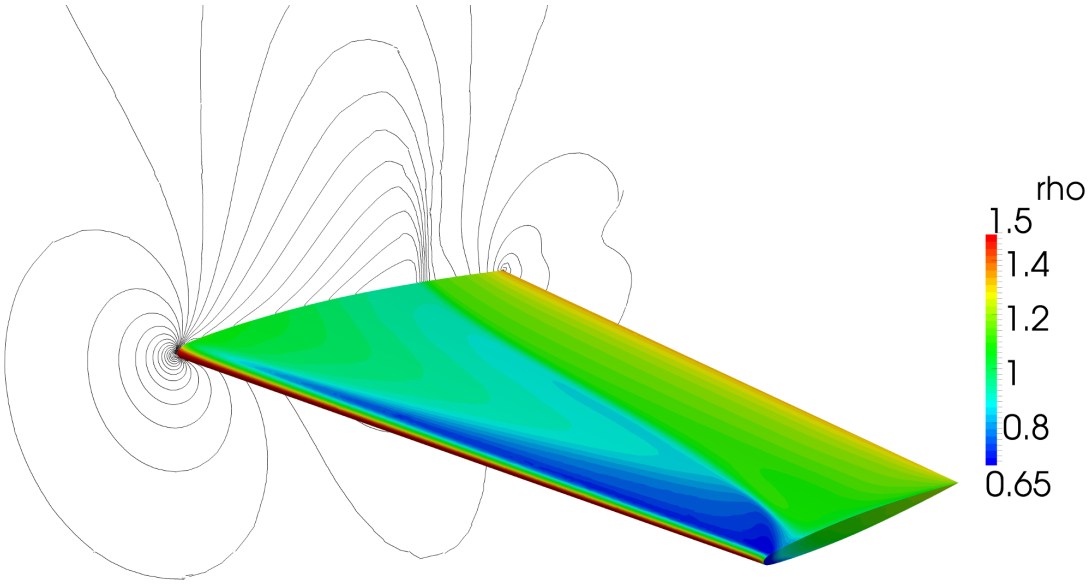


Figure 4.7: Transsonic flow over ONERA M6 airfoil with $P = 3$ elements, density distribution, 40 isosurfaces at symmetry plane

The computations in this section can be accelerated by a so-called *P-refinement* or *P-enrichment* technique. This means that the time stepping is started with low order polynomials and the ansatz space is subsequently refined or enriched. The first iterations are performed in a polynomial space with basis functions of degree P_1, e.g. $P_1 = 1$. On this level the time stepping is executed until the norm of the residual has reached a specified tolerance. Then the approximate solution is included into the next finer space with $P_2 = P_1 + 1$ polynomials and it is again iterated until the tolerance is reached. This process is repeated up to a desired order P_n. The transfer operation between two ansatz spaces with degrees P_l and P_{l+1} is straightforward since the polynomial bases $\{\psi_1, \dots \psi_{N_{P_l}}\}$ and $\{\phi_1, \dots \phi_{N_{P_{l+1}}}\}$, given in equation (2.17), are chosen to be hierarchical as described in section 2.1. Thus, the solution on level P_l can be directly embedded into a space with P_{l+1} polynomials. Let $\{\boldsymbol{r}_i^{P_l} \in \mathcal{T},\ i = 1, \dots, N_{P_l}\}$ and $\{\boldsymbol{r}_i^{P_{l+1}} \in \mathcal{T},\ i = 1, \dots, N_{P_{l+1}}\}$ be two sets of interpolation points, one corresponding to the Lagrange basis in the P_l space and one corresponding to the P_{l+1} space. Further, let U_k

denote the unknown, nodal variables in element number k and let

$$\begin{aligned}(V_1)_{i,j} &= \psi_j(\boldsymbol{r}_i^{P_l}), \qquad 1 \leq i,j \leq N_{P_l} \\ (V_2)_{i,j} &= \psi_j(\boldsymbol{r}_i^{P_{l+1}}), \quad 1 \leq j \leq N_{P_l},\, 1 \leq i \leq N_{P_{l+1}}\end{aligned} \tag{4.13}$$

denote two Vandermonde matrices. The approximated solution on the finer level $\tilde{U}_k$ is given by

$$\tilde{U}_k = V_2 V_1^{-1} U_k. \tag{4.14}$$

This procedure offers two advantages in terms of acceleration. On the one hand, evaluations on a coarser P-level are computationally cheaper than on the finer one since the number of unknown values increases like $\mathcal{O}(P^3)$ in three spatial dimensions. With this the dimension of the operators is smaller leading to less non-linear flux evaluations per element. Furthermore, the accuracy of the cubature formulas does not have to be as precise since the degree of polynomials under consideration is smaller. On the other hand, the time step for the Runge-Kutta time integration scales like $\mathcal{O}(P^{-2})$ (cf. [21]). Thus, on a coarser polynomial level a larger, stable time step can be chosen, which leads to a faster propagation in time since in the Euler flow examples considered in this work only the steady state solution is of interest. The P-refinement is demonstrated using the sphere test case presented in chapter 3. The underlying grid has a total number of 25,956 tetrahedra of which 4,457 are designated to become curved elements. The curvature is based on a $P = 4$ linear elasticity solution. This solution can then be evaluated at the interpolation points of every DG scheme with degree lower or equal 4.

Table 4.2 shows the results for computations with a successive refinement from $P = 2$ to $P = 3$ and $P = 4$ basis elements. Based on that relatively coarse grid, the $P = 0$ and $P = 1$ computations did not lead to reasonable results. Here two aspects can be observed. First, the total number of iterations on the left hand side is significantly smaller than in the pure $P = 4$ computation. Especially the number of iterations on the computationally most expensive level is only a third as large. Moreover, a decay in the maximum stable time steps between the P-level can be observed. These two factors result in an overall run time for the P-refinement strategy which is only halve of the run time for the plain algorithm. On each level it is iterated until the normalized residual reaches a tolerance of 10^{-8}. This residual is evaluated every 1,000 iterations to use the GPU most efficiently. The iteration counts and time steps in the example of table 4.2 where found experimentally and turned out to be the best practice for that particular test case.

P	# iterations	time step	time
2	8,000	6e-3	105.8 s
3	10,000	2e-3	196.5 s
4	20,000	5e-4	650.0 s
	38,000		952.3 s

P	# iterations	time step	time
2			
3			
4	61,000	5e-4	1,982 s
	61,000		1,982 s

Table 4.2: Subsonic flow past a sphere - steady state computation with (left) and without p-refinement (right)

The last test case is a turbulent Navier-Stokes flow past a sphere as described in section 3.1 at $M_\infty = 0.3$ with a low Reynolds number of $\mathrm{Re} = 300$. In this computation the time-dependent

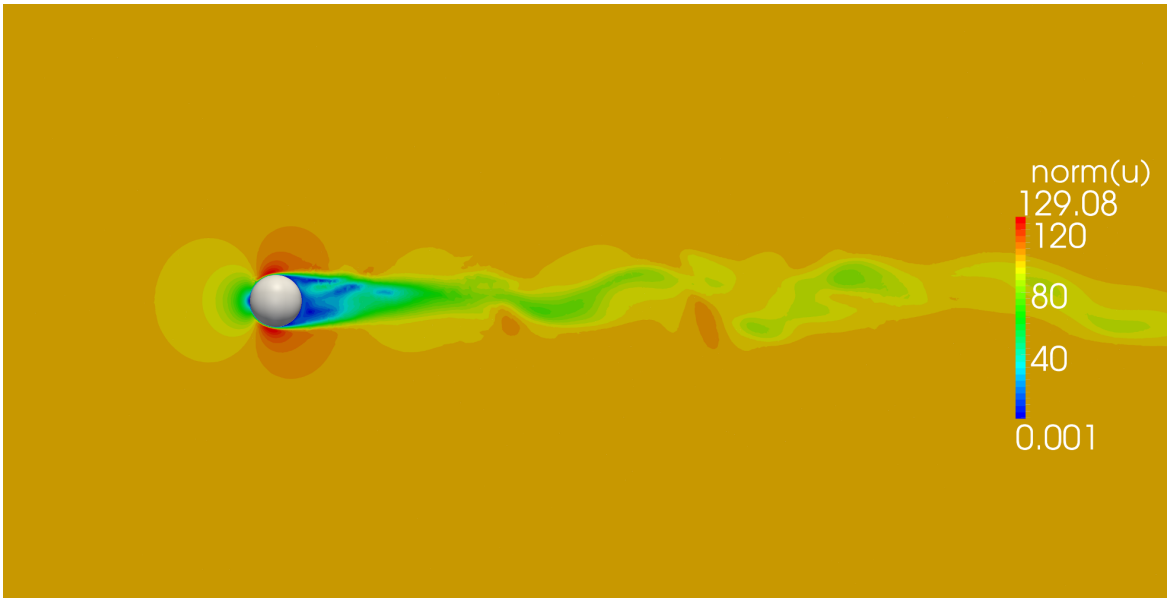


Figure 4.8: Magnitude of velocity, turbulent flow past a sphere, $M_\infty = 0.3$, $\mathrm{Re} = 300$

Navier-Stokes equations are solved directly for which reason this approach is called *direct numerical simulation* (DNS). However, this approach is only possible for low Reynolds numbers resulting in a moderate degree of turbulence. For flows with higher Reynolds numbers and more turbulence, techniques like turbulence modeling have to be applied. Even the application of super computers cannot resolve turbulence in complex flows with the DNS approach.

The underlying mesh in this test case consists of approximately 180,000 tetrahedra and is refined in the wake region of the sphere. Together with a $P = 3$ polynomial discretization, the total number of interpolation points is about 3,600,000. Figure 4.8 shows the magnitude of the velocity on a slice of the xy-plane and figure 4.9 visualizes the generated turbulence. Here an iso-surface in the density distribution of $\rho = 1.213$ is drawn in the wake of the sphere.

4.3.2 Performance Analysis

Whereas the previous sections show a proof of concept, this section concentrates on the performance analysis. For a software developer it is important to know whether the additional programming effort for the GPU is compensated by a speedup gained through the different hardware architecture. The algorithm presented within this work is optimized for parallel computing architectures rather than CPUs. Thus, a comparison of that algorithm implemented for a GPU, on the one hand, and on the other for a CPU, is not representative. It gives yet an estimation for which kind of numerical methods stream processors would fit and what speedup could be gained. In contrast, in the next section the GPU-accelerated code is compared to a conventional CPU code with optimized numerical methods.

As described in section 4.2, the developed GPU algorithm assumes a cluster where each compute node is equipped with one GPU. Whether the system is a shared or distributed memory

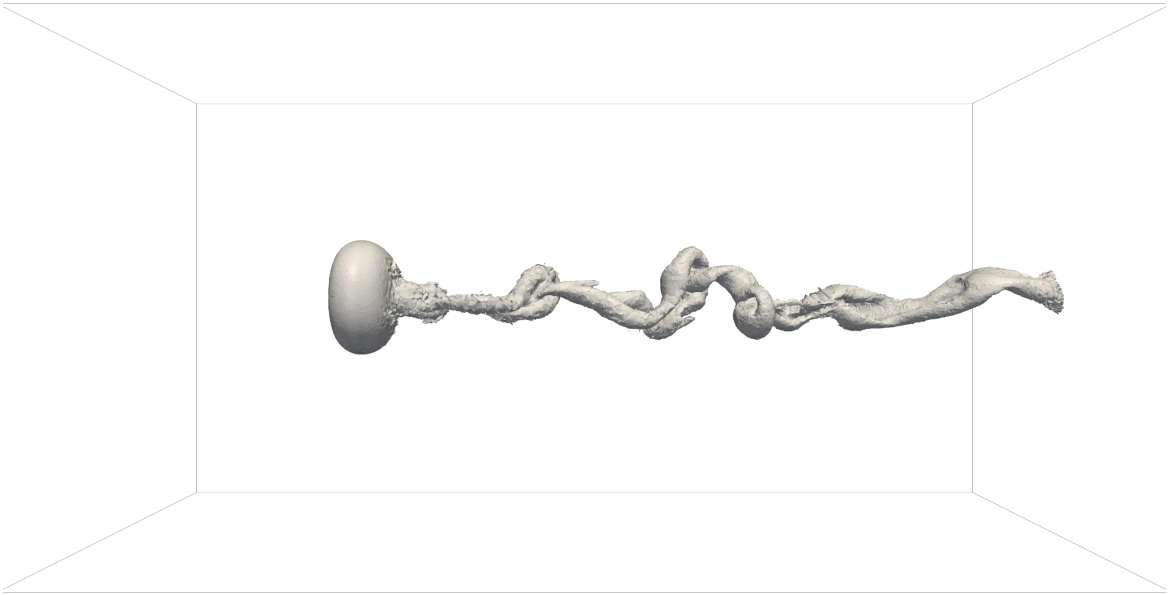


Figure 4.9: $\rho = 1.213$ iso-surface, Navier-Stokes flow past a sphere, $M_\infty = 0.3$, $\mathrm{Re} = 300$

model does not matter since the communication on top of the algorithm is performed through MPI. In order not to leave resources unused, it turned out to be the best setting to have one GPU per CPU core and not one per full CPU in the cluster since in a highly efficient GPU algorithm the CPU should only carry out small tasks connected to the control flow.

For the purpose of this work, a system equipped with 8 Intel Xeon E5620 CPU cores clocked at 2.4 GHz and 8 NVIDIA Tesla M2050 GPUs with CUDA 4.0 driver is utilized. Each of the GPUs has 3 GB of global RAM. In order to measure the performance of the GPU-accelerated code compared to a conventional CPU code, the same numerical scheme as described in chapter 2 is implemented for both hardware architectures. On the CPU side most of the matrix-vector multiplications are implemented using optimized BLAS (*basic linear algebra system*) routines and the flux evaluations are serialized in contrast to the GPU algorithm. In order to optimize the code for the underlying Intel hardware, it is compiled with the Intel C compiler 12.0 and the Intel Math Kernel Library (MKL) for the algebra. The maximum performance was achieved by applying BLAS routines to the products with $\mathcal{I}_V$, $\mathcal{I}_S$ and $M_{\partial\Omega}$ (cf. equation (2.39) and (2.78)). In contrast, like in the GPU algorithm, the volume contribution $\sum_{l=1}^{3} S_{k,x_l} F_l \left(\mathcal{I}_V U_k\right)$ is merged into one matrix-vector product, which turned out to be significantly faster than three separate BLAS calls. This could possibly be explained by a better CPU cache utilization and data reuse. Whereas the GPU code generates one grid per kernel call in which each CUDA block corresponds to one DG element, in the CPU algorithm there is an outer loop iterating over all K elements. Inside this loop the CPU code performs steps 4-9 in figure 4.3.

As seen in the derivation of the GPU algorithm, the computations of the Runge-Kutta updates of the DG elements do not depend on each other which also holds in the CPU version. Thus, on the level of this outer loop a OPENMP parallelization can be applied in order to

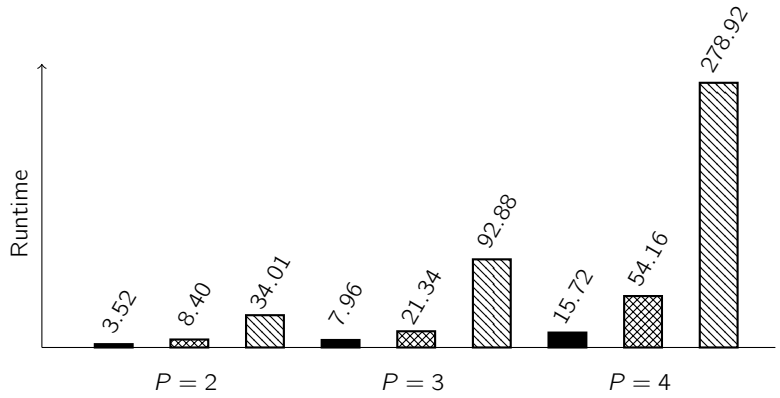


Figure 4.10: GPU and CPU runtimes on approximately 26,000 cells, 200 time steps, black: GPU single precision, crosshatched: GPU double precision, hatched: OPENMP parallel CPU

fully utilize the available hardware. Many articles report impressive GPU speedups, however the authors are comparing their optimized parallel code with an non-optimal serial code only running on one core of a CPU, which is not representative since almost all modern CPUs have multiple cores. In the present case including the OPENMP directive `#pragma omp parallel for ordered schedule(dynamic)` in front of the outer loop, iterating over all elements, is sufficient to enable parallelization over all cores of the underlying CPU.

Figure 4.10 visualizes the runtimes needed for 200 Runge-Kutta time steps on the sphere test case presented in section 3.1 with approximately 26,000 cells on one CPU and one GPU, respectively. The Mach number is chosen as $M_\infty = 0.8$ such that the shock detector and artificial viscosity can be applied. This test inspects the most interesting polynomial degrees with respect to CFD, $P = 2$, $P = 3$ and $P = 4$ where there are approximately $260k$, $520k$ and $910k$ interpolation points in the scheme. Due to the strong non-linear nature of the Euler flux, polynomial reconstructions of higher degrees are problematic and were often divergent in the test cases.

The GPU algorithm is run in two settings: once with single precision arithmetic represented by the black bars and once in double precision represented by the crosshatched bars. The hatched bars show the runtimes for the CPU algorithm. Since the mesh is not changed, the runtimes increase with higher polynomial degrees for the CPU and the GPU as well. This stems from the larger number of nodal and cubature points in the scheme leading to more flux evaluations. It can further be observed that the speedup of the GPU also increases with higher polynomial degrees. Whereas the speedup for $P = 2$ is 4 in double precision and about 9.7 in single precision, for $P = 4$ it grows to 5.2 and 18.3, respectively.

It is still an open question why the observed increment in single precision is significantly larger than the gained speedup in double precision. It can possibly explained be by the limited amount of shared memory which allows the GPU hardware to execute less CUDA blocks in parallel in the double precision case. However, the gap between a speedup of factor 18.3 and 5.2 is caused due to the hardware design. In the present GPU architecture the single precision instruction

throughput is still significantly larger than in double precision. Yet, the manufacturers are improving this issue. In the setting of figure 4.10 the CPU algorithm achieved a runtime of 1016.36 s without the OPENMP parallelization. Compared to 278.92 s in the parallel version, this reflects the theoretical speedup that can be gained on a four core CPU like the Intel Xeon E5620.

A natural question, which arises in this context, is how the GPU performs if the CPU code is adapted in a one-to-one fashion using the cuBLAS library. In [59] the manufacturer reports high speedups of the cuBLAS library against the Intel MKL for the most common BLAS calls. However, these speedups are achieved for much larger matrices than the DG operators within this work. The speedup moreover shrinks if a large number of small matrix-vector products is considered like in the DG scheme. Thus, the GPU algorithm, as described above, shows to be superior compared to cuBLAS since the latencies of a large number of kernel calls is overcome. Additionally data reuse is enabled by exploiting the structure of the DG scheme which heavily influences the GPU performance.

# GPUs	1	2	4	6	8
# cells	5,043	10,083	20,073	30,051	40,012
time	23.49 s	24.48 s	24.91 s	25.60 s	25.83 s
efficiency	100%	95.96%	94.30%	91.76%	90.94%

Table 4.3: Weak scaling over a cluster of 8 GPUs, $P = 4$ elements, 200 iterations

The next test deals with the weak scaling of the GPU algorithm on the level of MPI parallelism. For this purpose, the number of cells in the mesh is increased proportionally to the number of GPUs in the parallel MPI framework. Table 4.3 shows the runtime results from 1 to 8 GPUs and the achieved efficiency. Here it can be observed that the weak scaling efficiency obtained in this test is significantly worse than expected although the number of nodes in the cluster is relatively small.

There might be two reasons for that. First, the GPU code does not yet utilize the feature of computation and data overlapping available since CUDA 4.0. When executing on multiple GPUs, in each stage of the Runge-Kutta time stepping unknown values from the process interfaces have to be downloaded from the GPU, sent to adjacent nodes and uploaded to the GPU again. In the present algorithm this means 5 data transfers per time step. The weak scaling efficiency suggests that data transfers through the PCI express bus are a bottleneck in this algorithm.

A second reason could be that the ratio between computational time on the GPU and data transfer to other nodes in the cluster is not optimal for a GPU environment. This means that the GPU evaluates its computations too fast compared to the time needed for data exchanges through MPI.

4.4 Comparison to Conventional Algorithms

In this section the explicit Runge-Kutta GPU algorithm is compared to classical, implicit discontinuous Galerkin approaches used in industrial applications. One major drawback of an explicit time stepping is the small stability region which enforces small time steps. Steady state simulations thus need significantly more iterations. In contrast, the weakness of implicit methods is the larger amount of memory required to store Jacobi matrices or at least parts of them. It is thus tempting to investigate whether the computational power of GPUs is able to overcome the issue of smaller time steps. For this purpose, an implicit solver for the Euler equations is developed which reflects the techniques used within the PADGE code [50] of the DLR (Deutsches Zentrum für Luft und Raumfahrt).

One major building block of an implicit solver is an efficient procedure which sets up Jacobi matrices of the discretization $\mathcal{N}_h$ and $\tilde{\mathcal{N}}_h$ (cf. section 2.3). These are notations for the right hand side of equation (2.39), once multiplied with the inverse mass matrix and once not. The first one is needed for a backward Euler time stepping and the second one for Newton's method. Note that in this chapter the pure hyperbolic case is investigated neglecting the viscous fluxes. The routine for the Jacobi matrix works similarly to the techniques which will be presented in the next chapter and is not described in detail at this point.

Reviewing the backward Euler time stepping given in equation (2.71) the following non-linear system

$$U_h^{n+1} - U_h^n + \Delta t \mathcal{N}_h(U_h^{n+1}) = 0 \tag{4.15}$$

has to be solved for each time step by Newton's method. Thus, matrices of the following type have to be assembled

$$J_{BE} = I + \Delta t \frac{\partial \mathcal{N}_h}{\partial U_h}(U_h). \tag{4.16}$$

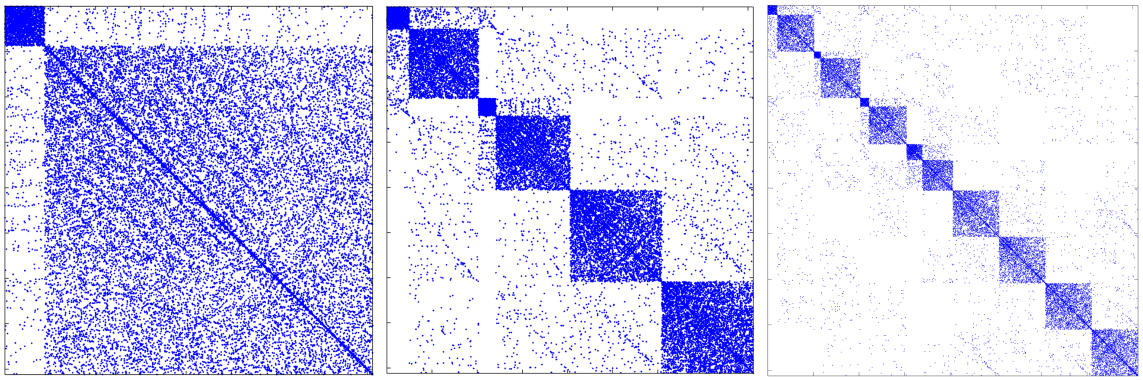
In the case that the time-independent equations are solved with Newton's method, the required matrix is given by

$$J_N = \frac{\partial \tilde{\mathcal{N}}_h}{\partial U_h}(U_h) = M \frac{\partial \mathcal{N}_h}{\partial U_h}(U_h) \tag{4.17}$$

where M is the blockdiagonal mass matrix which is given in terms of the local mass matrices (2.32).

In order to guarantee scalability, the matrix assembly is designed for distributed memory systems. As mentioned earlier in this chapter, a comparison between conventional hardware and GPUs can only be representative if all cores of the CPU are busy and not only one. Thus, the assembly routine utilizes graph partitioning and load balancing performed by the ParMetis library like the explicit GPU code. This results in each core holding a partition of rows of the Jacobi matrix (4.16) or (4.17). Communication is performed over the same MPI channels like in the explicit code. The distributed linear systems are then solved with the PETSc library [60]. This library is one of the most widely used linear system solvers in HPC and is known for good performance and scalability. In contrast to the previous section where the full computational power of the four CPU cores is enabled by an OPENMP parallelization, in this section on each core of the CPU one MPI process is launched.

Again, the test setting is a subsonic flow past a sphere like in figure 3.1 with $M_\infty = 0.38$. The underlying mesh consists of 8,117 tetrahedra of which 875 are curved according to chapter



(a) 1 core (b) 4 cores (c) 8 cores

Figure 4.11: Non zero pattern of Jacobian matrix $\frac{\partial \mathcal{N}_h}{\partial U_h}$ influenced by the ParMetis reordering and partitioning

3. This is a relatively coarse discretization however it has to be ensured that on the polynomial level of $P = 4$ there is enough memory for the system matrix available.

Figure 4.11 visualizes the influence of the graph partitioning and reordering on the Jacobi matrix $\frac{\partial \mathcal{N}_h}{\partial U_h}$ and similarly $\frac{\partial \tilde{\mathcal{N}}_h}{\partial U_h}(U_h)$. Each dot in the figure is a DG block of size $35 \cdot 5 \times 35 \cdot 5$ since this is a $P = 4$ computation. In (a) the matrix is assembled in a serial code and the numbering is the same as the mesh generator has chosen. The agglomeration of blocks in the first rows and columns indicates the region of curved cells, which is triangulated first by the mesh generator. It can also be seen that in these rows and columns of the matrix there are only a few off-diagonal blocks, which reflects the locality of the curved cells. In (b) and (c) the mesh is distributed between 4 and 8 cores, respectively. Each process owns approximately 1/4 of the rows in (b) and 1/8 in (c) which stems from the load balancing. The simple renumbering of cells in the mesh leads to a computationally much more attractive structure in the matrix for multicore architectures.

The essential part of a Krylov subspace method is the efficient evaluation of matrix-vector products and the evaluation of scalar products. Thus, in a distributed, parallel algorithm like in the PETSc library, the fraction of elements, which are not in the diagonal blocks, heavily influences the overall performance. Since the vector of unknowns is also distributed, each off-diagonal block in the Jacobi matrix requires data exchanges through the message passing interface, which is a potential bottleneck in parallel, linear solvers. This makes the application of a graph partitioning algorithm mandatory.

Following the presentations in [50] the backward Euler time stepping is conducted by a Newton-GMRES approach. Thus, the non-linear system is treated with Newton's method and the arising linear subproblems are solved with the standard GMRES solver in the PETSc toolbox. The included restarted-GMRES algorithm is set up to store 50 vectors, which is found to be a suitable choice. The preconditioner is chosen to be of ILU type. Although many articles report a good performance of block-Jacobi preconditioners in the context of DG methods, ILU leads to significantly smaller run times within the test case of this section. Yet, block-Jacobi

P	1	2	3	4
Δt explicit	$2e-2$	$1e-2$	$8e-3$	$5e-3$
Δt implicit	$2e+2$	$6e-2$	$2e-2$	$1e-2$
Δt implicit (preconditioned)		$2e-1$	$8e-2$	$6e-2$

Table 4.4: Time steps in the first iterations for the explicit GPU algorithm and the implicit Euler method

is straightforward in terms of programming since the DG discretization naturally brings a block structure into the system matrix.

Parallelization of ILU preconditioners is not a simple task since it is conceptually a serial problem. However, the parallel ILU(p) preconditioner in the HYPRE library [61] which is derived in [62] shows high efficiency and scalability even on very large linear systems. Within the present test case the best practice setting for ILU is a level of fill of $p = 1$. More detailed information on iterative methods and preconditioners can be found in [63].

Table 4.4 shows the maximum time step for the first iterations in the explicit GPU algorithm and the backward Euler implicit method for different polynomial degrees. The explicit time steps are chosen to be the largest feasible step. In contrast, the implicit Δt is chosen such that the arising systems (4.17) are solvable in reasonable time compared to the explicit time steps. The condition number of J_{BE} strongly depends on Δt which is reflected in the number of required GMRES iterations. Here it should be remarked that the number of iterations also depends on other factors like the chosen preconditioner, the restart width of GMRES and of course the tolerance in the residual. Thus, a comparison of explicit and implicit time stepping can only be conducted in terms of best practice settings for the implicit solver.

In the present test case a relative tolerance of 10^{-5} and a maximum number of 1,000 iterations in the GMRES algorithm seems to be a reasonable choice. Moreover, the outer Newton iteration is chosen to have a maximum number of 8 iterations and also a relative tolerance of 10^{-5}. The time steps in table 4.4 are chosen sufficiently small in order to fulfill the conditions given above. This leads to the class of inexact Newton methods which is analyzed for example in [64].

It can be further seen in table 4.4 that, apart from the $P = 1$ level, the implicit time steps are not significantly larger than the explicit time steps. Thus, if one is interested in a $P = 4$ solution, the only reasonable approach seems to be a successive P-refinement starting with the $P = 1$ level. The third row of table 4.4 shows the maximum implicit time steps after a P-preconditioning. This means that the backward Euler method on level P is started with a converged solution on level $P - 1$ which is included into the richer polynomial space thanks to the hierarchical basis.

This issue is known since the application of implicit time integration schemes became popular in fluid dynamics. It is already investigated in [65] where a strategy is proposed which controls the magnitude of the time steps relative to the residual. The time steps shown in table 4.4 are certainly only valid for the first iterations. When the integration scheme proceeds towards the solution, the implicit time steps greatly increase. Finally, in the vicinity of the solution the time integration scheme switches to Newton's methods and benefits from the locally quadratic

convergence. In [65] it is thus suggested that the first iterations should mimic explicit time steps with small Δt. Then the backward Euler time integration starts increasing the time steps and finally a few Newton steps lead to the solution.

Comparing the backward Euler method on four parallel processes with the explicit algorithm on one GPU, it turns out that on each P-level approximately 6,000 explicit iterations could be performed in the run time of one implicit step. Apart from the $P = 1$ level, where the implicit time step is significantly larger than the explicit one, this is a considerable speedup. Of course, this does only hold for the first iterations. While the implicit scheme allows larger time steps after a few iterations, the explicit time step cannot be enlarged as much. Especially in the vicinity of the solution the implicit scheme is superior to the explicit one. It thus turns out to be the best strategy for this test case to start directly on the finest P-level with explicit time steps until the full Newton step is computable.

In summary, it can be said that for this type of PDEs the explicit GPU time stepping can be applied as an excellent preconditioner. One advantage over the implicit scheme is that the simulation can be started directly on the finest P-level. Moreover, in the first few iterations, where the implicit scheme has to act like an explicit time stepping, the computational performance of the GPU can be utilized in order to reach the point where Newton steps are possible much faster.

Chapter 5

Discrete Adjoint GPU Implementation

This chapter deals with discrete adjoint approaches providing a basis for optimizations of many kinds. Here PDE constrained optimization is considered. This means that an objective function is to be minimized which takes the solution of a PDE and control variables as input. The control variables may have an impact on the geometry of the domain or the physical phenomena modeled by the PDE. Thus, in order to minimize the objective function, its sensitivities with respect to the control variables and the whole solution process of the PDE have to be analyzed. Even one of the most simple and natural optimization techniques, the steepest descent method, makes use of the gradient of the objective function with respect to the control variables. For a small number of control variables this gradient can be obtained by finite difference approximation. However, this procedure leads to as many PDE evaluations as there are control variables, which can be very problematic if for example the shape of the domain is considered to be variable. In many cases this approach leads to a numerical effort which is absolutely unrealistic. This issue can be overcome by so-called *adjoint calculus*.

For this purpose, the discrete adjoint approach is shortly introduced in this chapter. Then it is discussed how the GPU algorithm developed so far can be applied in order to solve the arising adjoint system, which is partly published in [66]. An introductory overview of the use of adjoint approaches can be found for example in [67].

5.1 Introduction to Discrete Adjoint Approaches

Assuming the situation of chapter 2 where the solution of a system of m hyperbolic PDEs is approximated on a mesh discretizing a domain $\Omega = \bigcup_{k=1}^{K} \Omega_k$ with K discontinuous Galerkin elements. In each of these elements the solution is represented in a polynomial space up to degree P leading to $m \cdot N_P$ degrees of freedom in U_k and a total of $N = K \cdot m \cdot N_P$ degrees of freedom in U_h. Further, a control variable $\alpha = (\alpha_1, \ldots, \alpha_M)$ is introduced. Then a non-linear function

$$\mathcal{N}_h : \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^N \tag{5.1}$$

is formulated which represents the right hand side of the discontinuous Galerkin discretization of the PDE and the boundary conditions. In contrast to section 2.3, here $\mathcal{N}_h$ has an additional dependency on the control variable α. It is assumed that the discretization of the PDE is fulfilled by a solution vector U_h which yields

$$\mathcal{N}_h(U_h, \alpha) = 0. \tag{5.2}$$

This could for example be achieved by iterating a time stepping scheme until the steady state solution is reached. Further, let

$$\mathcal{J}_h : \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R} \tag{5.3}$$

be an objective function depending on the solution of the PDE and the design variable. The associated optimization problem then reads as

$$\begin{aligned} \min_{(U_h,\alpha)} \quad & \mathcal{J}_h(U_h, \alpha) \\ \text{s.t.} \quad & \mathcal{N}_h(U_h, \alpha) = 0. \end{aligned} \tag{5.4}$$

The gradient of $\mathcal{J}$ is given by

$$\frac{d\mathcal{J}}{d\alpha} = \left(\frac{d\mathcal{J}}{d\alpha_1}, \ldots, \frac{d\mathcal{J}}{d\alpha_M} \right) \tag{5.5}$$

and is approximated by component-wise finite difference evaluations

$$\frac{d\mathcal{J}}{d\alpha_i} = \frac{\mathcal{J}\left(U_h(\tilde{\alpha}), \tilde{\alpha}\right) - \mathcal{J}\left(U_h(\alpha), \alpha\right)}{\delta}, \quad i = 1, \ldots, M. \tag{5.6}$$

In the equation above $\tilde{\alpha} = \alpha + \delta e_i$ with $e_i = (0, \ldots, 0, 1, 0, \ldots, 0)^T$ being the i-th unit vector and $\delta > 0$. Here $U_h(\alpha)$ clarifies the dependency of the solution on the design parameter. Thus, for each component of the gradient one solution of the PDE has to be evaluated resulting in a total number of $M + 1$ runs of the flow solver, which is impractical.

Another approach to evaluate sensitivities of the objective function with respect to the solution is the discrete adjoint approach, which is for example derived in [12]. Partial derivatives in the following presentations are evaluated at the approximated solution U_h, which is omitted for clarity. First, the right hand side equation of the PDE discretization (5.2) is differentiated with respect to α_i for $i = 1, \ldots, M$ which yields

$$\frac{\partial \mathcal{N}_h}{\partial U_h} \frac{dU_h}{d\alpha_i} + \frac{\partial \mathcal{N}_h}{\partial \alpha_i} = 0. \tag{5.7}$$

Assuming that $\frac{\partial \mathcal{N}_h}{\partial U_h}$ is regular it follows that

$$-\left(\frac{\partial \mathcal{N}_h}{\partial U_h} \right)^{-1} \frac{\partial \mathcal{N}_h}{\partial \alpha_i} = \frac{dU_h}{d\alpha_i}. \tag{5.8}$$

Differentiating the objective function with respect to α leads to

$$\begin{aligned} \frac{d\mathcal{J}_h}{d\alpha_i} &= \frac{\partial \mathcal{J}_h}{\partial U_h} \frac{dU_h}{d\alpha_i} + \frac{\partial \mathcal{J}_h}{\partial \alpha_i} \\ &= -\frac{\partial \mathcal{J}_h}{\partial U_h} \left(\frac{\partial \mathcal{N}_h}{\partial U_h} \right)^{-1} \frac{\partial \mathcal{N}_h}{\partial \alpha_i} + \frac{\partial \mathcal{J}_h}{\partial \alpha_i} \\ &= \lambda_h^T \frac{\partial \mathcal{N}_h}{\partial \alpha_i} + \frac{\partial \mathcal{J}_h}{\partial \alpha_i}. \end{aligned} \tag{5.9}$$

Here λ_h is the solution of the so-called *adjoint equation*

$$\left(\frac{\partial \mathcal{N}_h}{\partial U_h}\right)^T \lambda_h = -\left(\frac{\partial \mathcal{J}_h}{\partial U_h}\right)^T . \tag{5.10}$$

Once the linear system (5.10) is solved, each component evaluation of the gradient (5.9) comes at the cost of one scalar product, which is very cheap compared to multiple flow solutions needed for the finite difference approach.

The key point of this discrete adjoint approach is the evaluation of $\frac{\partial \mathcal{N}_h}{\partial U_h}$ or alternatively the evaluation of products like $\left(\frac{\partial \mathcal{N}_h}{\partial U_h}\right)^T \lambda_h$. In practice this task includes the differentiation of a computer program, consisting of thousands of lines of code, which is in general unrealistic to do by hand. Thus, a whole field of research deals with a technique called *automatic* or *algorithmic differentiation* (AD). An extensive overview can be found for example in [68]. AD is based on the idea that even the most complicated computer code is a concatenation of elementary arithmetic operations. Thus, roughly speaking an iterated application of the chain rule yields the differentiated code. This differentiation is realized by either a source code transformation or operator overloading if supported by the underlying language. In general, the outcome of an AD tool is a automatically generated computer code which represents the derivative. An interesting result in this field is published in [69] where upper bounds for the number of floating point operations in the automatic differentiated code compared to the original non-linear code are given.

However, for the purpose of this work, techniques like AD cannot be applied. In the special case of GPU programming it is not guaranteed that source code transformed via AD maintains the performance optimization implemented in the primal solver. As presented in chapter 4, the performance on GPUs is very sensitive for example with respect to memory patterns and the optimized memory accesses are probably not maintained after applying a AD tool. Thus, in the successive section a different approach is followed.

5.2 GPU Implementation

This section presents an approach to implement a discrete adjoint algorithm for the Euler equations on GPUs. The underlying primal algorithm is the one derived in chapter 2 and 4. For this purpose, the focus is on maintaining most parts of the primal algorithm and the performance optimizations therein. Without loss of generality $\alpha \in \mathbb{R}$ is chosen to be a scalar design parameter in this section. According to the primal variable U_h and the cell-wise variable U_k, adjoint variables λ_h and λ_k of the same dimension are introduced. Furthermore, a pseudo time is introduced like in the hyperbolic system (2.3) such that the same Runge-Kutta time stepping can be applied in order to solve the adjoint equation

$$\frac{\partial \lambda_h}{\partial t} = -\left(\frac{\partial \mathcal{N}_h}{\partial U_h}\right)^T \lambda_h - \left(\frac{\partial \mathcal{J}_h}{\partial \alpha}\right)^T . \tag{5.11}$$

A slightly different approach is described in [70] where also an adjoint version of the Runge-Kutta time stepping is considered. This procedure ensures that the gradient of the objective functional is obtained after as many steps as in the primal time stepping.

Comparing system (5.11) to the primal one

$$\frac{\partial U_h}{\partial t} = -\mathcal{N}_h(U_h) \tag{5.12}$$

the key point is to replace the evaluation of the non-linear DG discretization with the evaluation of the product with the transposed Jacobi matrix of $\mathcal{N}_h$. Note that $\frac{\partial \mathcal{N}_h}{\partial U_h}$ is evaluated at the steady state solution and is thus constant. Yet, for the purpose of a GPU application storing the full Jacobi matrix has to be prevented in any case, not only because of the limited device memory, but also to maintain the algorithm to be arithmetic dominated rather than memory dominated. Thus, in the following the product of the transposed Jacobi matrix $\left(\frac{\partial \mathcal{N}_h}{\partial U_h}\right)^T$ and the adjoint variable λ_h is derived and implemented on the GPU. As seen in the previous sections, most of the underlying, explicit discontinuous Galerkin scheme consists of small and dense matrix vector multiplications and is thus uncritical with respect to differentiation. Further, the viscous fluxes are also straightforward due to the linearity. Only the Lax-Friedrichs numerical flux, which is given in (2.60), has to be revised. Replacing the maximum function in equation (2.61) by the following expression

$$A = \left|\lambda\left(F_{\vec{n}}^{C\prime}\left(\frac{1}{2}\left(U_h^- + U_h^+\right)\right)\right)\right| = \left|\frac{1}{2}\langle \boldsymbol{u}^- + \boldsymbol{u}^+, \vec{n}\rangle\right| + a\left(\frac{1}{2}\left(U_h^- + U_h^+\right)\right) \tag{5.13}$$

brings more symmetry to the flux, which is essential for the GPU code to maintain its original form. However, this simplification does not lead to noticeable effects on the convergence of the numerical method. It does not even impact the maximum time step in the test cases of section 4.3.1. Conservativity and consistency is clearly maintained.

A further simplification is to keep the viscosity parameter ϵ_h fixed throughout the pseudo time stepping in (5.11) which spares the programmer the implementation of the derivative of the shock detector (2.81). Furthermore, this seems to be a reasonable decision since the steady state solution U_h is fixed during the adjoint time stepping and thus ϵ_h marks the troubled cells where discontinuities arise in U_h. For a deeper insight it is worth repeating the cell-wise independent operations for cell number k in the primal algorithm (2.78) which are given by

$$\begin{aligned} Q_{k,x_m} =& S_{k,x_m}\mathcal{I}_V\epsilon_k U_k - M_{\partial\Omega_k}\frac{1}{2}\left(\epsilon_k^- U_k^- + \epsilon_k^+ U_k^+\right) n_{x_m}, \quad m = 1,2,3 \\ \frac{\partial U_k}{\partial t} =& \sum_{m=1}^{3} S_{k,x_m}\left(F_m\left(\mathcal{I}_V U_k\right) + \mathcal{I}_V\epsilon_k Q_{k,x_m}\right) \\ & - M_{\partial\Omega_k}\left[H\left(U_k^-, U_k^+, \vec{n}\right) + \frac{1}{2}\left(\epsilon_k^- Q_k^- + \epsilon_k^+ Q_k^+\right)\cdot\vec{n}\right]. \end{aligned} \tag{5.14}$$

Differentiation with respect to U_k, transposing and multiplying with λ_k leads to the following

expression

$$\begin{aligned}
\Lambda_{k,x_m} =& \mathcal{I}_V^T \epsilon_k U_k S_{k,x_m}^T \lambda_k - \frac{1}{2}\mathcal{I}_S^T \left(\epsilon_k^- \lambda_k^- - \epsilon_k^+ \lambda_k^+\right) n_{x_m} , \quad m = 1,2,3 \\
\frac{\partial \lambda_k}{\partial t} =& -\mathcal{I}_V^T \sum_{m=1}^{3} \left[\frac{\partial F_m}{\partial U_k}^T \left(\mathcal{I}_V U_k\right) S_{k,x_m}^T \lambda_k + S_{k,x_m}^T \epsilon_k \Lambda_{k,x_m} \right] \\
&+ \mathcal{I}_S^T \left[\frac{\partial H}{\partial U_k^-}^T (U_k^-, U_k^+, \vec{n})(\lambda_k^- - \lambda_k^+) \right. \\
&\left. + \frac{1}{2}\left(\epsilon_k^- \Lambda_k^- - \epsilon_k^+ \Lambda_k^+\right) \cdot \vec{n} \right] - \frac{\partial \mathcal{J}_k}{\partial \alpha}^T (U_k).
\end{aligned} \tag{5.15}$$

In the system above Λ_{k,x_m} are the auxiliary variables for the adjoint gradient according to Q_{k,x_m}. $\frac{\partial \mathcal{J}_k}{\partial \alpha}^T (U_k)$ denotes the subvector of the gradient of the objective function with respect to the variables belonging to the k-th element. This vector can be precomputed and transferred to the device memory since it does not change throughout the time stepping. Further, the interpolation to surface quadrature nodes of the adjoint gradient Λ and the adjoint variable λ is not performed by multiplying with $\mathcal{I}_S$ like in the primal system (2.78). The transposition of the Jacobian matrix leads to a role reversal between interpolation and mass/stiffness matrices. Thus, $\Lambda_k^- = M_{\partial\Omega_k}^T \Lambda_k$ and $\lambda_k^- = M_{\partial\Omega_k}^T \lambda_k$ replace $Q_k^- = \mathcal{I}_S Q_k$ and $U_k^- = \mathcal{I}_S U_k$. Variables coming in from adjacent cells are treated in the same way. Moreover, the communication channels from the primal system are untouched. This means the identification of surface quadrature nodes from adjacent cells also holds for the adjoint algorithm. Most parts of the GPU algorithm and particularly the MPI framework can thus be reused. This stems from the fact that $M_{\partial\Omega_k}^T$ and $\mathcal{I}_S$ and further S_{k,x_m}^T and $\mathcal{I}_S$ have the same dimension and can be exchanged in the GPU kernels. The variables U_k^- and U_k^+ are not affected and are still interpolated by multiplying them with $\mathcal{I}_S$. Yet, a major difference between the two systems is that the signs of the normal vector fields switch. In the primal system the update of cell number k depends on the outward pointing normal vector field. However, in the adjoint system the normal vector fields of the adjacent cells are involved in the update calculation. The terms $\epsilon_k \Lambda_k^+$ and $\epsilon_k^+ \lambda_k^+$ are weighted with -1 in order to invert the normal vectors of cell number k which is then equivalent to the normal vector from the adjacent cells.

The conservative nature of the numerical flux function H inducing symmetry to the computations between adjacent elements, as discussed in section 2.2, states

$$H(V, W, \vec{n}) = -H(W, V, -\vec{n}). \tag{5.16}$$

This property simplifies the evaluation of the transposed Jacobi matrix in (5.15). The evaluation of the non-linear fluxes F_m in the primal system is replaced by the evaluation of the Jacobi matrices of the fluxes, which are given in (2.46), and the multiplication with the adjoint variable. Since this is a point-wise operation, it does not affect the overall structure of the algorithm.

Thus, the GPU algorithm for (5.15) mostly follows the execution cycle as described in figure 4.3 with the adjoint variable instead of the primal one. Since the steady state solution U_h does not change throughout the adjoint time stepping it can be distributed through the MPI interface in an initial stage of the adjoint simulation.

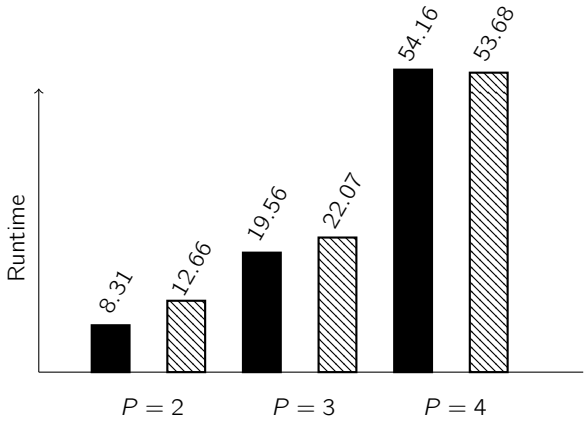


Figure 5.1: Comparison of primal (black) and adjoint (hatched) execution times on one GPU, 200 time steps on approximately 26,000 cells

One very critical issue with respect to GPU performance is the data alignment of the element-wise DG operators. Although in the adjoint system (5.15) products with transposed operators are involved, the matrix-vector operations as described in section 4.2 should be maintained. Thus, the operators have to be transposed in the device memory as well. Without this step memory accesses are not coalesced any more which critically affects the performance of the GPU.

It certainly appears not to be intuitive to evaluate the Jacobi matrix in each time step although this operator is constant and does not change. This approach however safes memory which is limited on GPUs. In particular, the evaluations of $\frac{\partial H}{\partial U_k^-}^T$ and $\frac{\partial F_m}{\partial U}^T$ are constant, yet they can be computed very efficiently in the cache of the GPU. Since computations on the GPU are cheap compared to memory transactions, this approach is superior with respect to performance. Storing these Jacobi matrices would result in an additional memory usage of 5×5 values for each of H,F_1,F_2 and F_3 per quadrature node.

Figure 5.1 shows a comparison of run times of the primal and the adjoint algorithm for 200 time integration steps. The underlying test case is the flow past a sphere as described in section 3.1 with approximately 26,000 cells. The run times for the primal algorithm are the same as in chart 4.10. The subsonic flow in this situation does not affect the run times of the primal and adjoint algorithm since shock detector and artificial viscosity have to be globally applied no matter if there are discontinuities in the solution or not. Here it can be observed that the run times of the adjoint algorithm are slightly slower for low order polynomials yet approximately as fast as in the primal algorithm for $P = 4$ elements.

In order to verify the adjoint GPU algorithm, a generic objective functional is once approximated with finite differences and once with the discrete adjoint approach. For this purpose, some non-dimensional characteristics are used in order to describe flow situations and airfoils. The non-dimensional pressure coefficient is given by

$$C_p = \frac{2(p - p_\infty)}{\gamma M_\infty^2 p_\infty} \tag{5.17}$$

in terms of the freestream conditions denoted by the subscript ∞. Then the lift coefficient C_L and drag coefficient C_D are given by

$$C_D = \frac{1}{C_{\text{ref}}} \int_{\Gamma_W} C_p \left\langle (\cos(\alpha), 0, \sin(\alpha))^T, \vec{n} \right\rangle dS \tag{5.18}$$

and similarly by

$$C_L = \frac{1}{C_{\text{ref}}} \int_{\Gamma_W} C_p \left\langle (-\sin(\alpha), 0, \cos(\alpha))^T, \vec{n} \right\rangle dS. \tag{5.19}$$

In the equations above α denotes the angle of attack of the airfoil and C_{ref} is a reference length which is chosen to be the chord length in the case of an airfoil. Further, Γ_W is the part of the boundary of the domain where the solid wall condition is chosen. This is the airfoil itself. In section 3.2 this part of the boundary is denoted by Γ_{D_1} and is the region which is curved according to a NURBS description. For the further computations it is an useful fact that

$$C_D'(\alpha) = C_L(\alpha) \quad \text{and} \quad C_L'(\alpha) = -C_D(\alpha). \tag{5.20}$$

In order to verify the discrete adjoint GPU code, the ONERA M6 test case, as described in section 4.3.1, is used. The design parameter is chosen to be the angle of attack and the objective functional is the drag of the airfoil. Then three computations are performed. First, the original test setting is executed with an angle of attack of $\alpha = 3.06°$ which converges after 512,000 iterations with a relative norm of the residual of 10^{-8} resulting in a solution U_h. Further, using U_h as an initial guess a second solution $\tilde{U}_h$ is computed with an angle of attack of $\tilde{\alpha} = 3.07°$.

Then the discrete adjoint time stepping is performed to obtain λ_h, which also converges to a relative norm of the residual of 10^{-8} after 485,000 iterations. Plugging this into formula (5.9) yields a derivative of $\frac{dC_D}{d\alpha} = 0.150427$. The term $\frac{\partial \mathcal{N}_h}{\partial \alpha}$ can be evaluated easily since in the primal solver only the free stream boundary conditions depend on the angle of attack. Moreover, $\frac{\partial C_D}{\partial \alpha}$ is given by the lift coefficient of the airfoil and does not require further computations.

The second solution $\tilde{U}_h$ is then used to compute a finite difference approximation to the derivatives which yields $\frac{dC_D}{d\alpha} = 0.150414$. This shows that the discrete adjoint approximation is close to the finite different approximation of the derivative of the drag coefficient. Further, it can be seen as a validation of the GPU adjoint algorithm.

Chapter 6

Shape Optimization Based on DG

In this chapter the tools developed so far are combined and applied in the field of shape optimization. At the end of this chapter the transsonic ONERA M6 test case is reviewed. The aim is to apply very small modifications to the geometry of the wing in order to minimize the drag, whereas the overall shape is mostly maintained. For this purpose, the primal and adjoint Euler GPU solver, developed within this work, are used. Moreover, the mesh deformation tool, described in chapter 3, is applied again to deform the shape and the whole discretization mesh. The theory of shape calculus within this chapter mostly follows the two fundamental works in this field [13, 71]. Additionally [72] gives an overview. The application of shape calculus to aerodynamic problems can be found in [14, 15, 73]. First, the basics of shape optimization needed for this work are shortly reviewed. After that the focus is on the application of higher order discretization methods and particularly the discontinuous Galerkin method. It is discussed how to deal with the difficulties arising from discontinuous basis functions and non-smooth gradients.

6.1 A Short Introduction to Shape Calculus

In what follows shape optimization problems of the form

$$\begin{aligned} &\min_{(U,\Omega)} \mathcal{J}(U,\Omega) \\ &\text{s.t. } \; c(U,\Omega) = 0 \end{aligned} \tag{6.1}$$

are examined. According to the previous chapters Ω is a domain in which the underlying, physical phenomena are simulated, c is a discretization of the PDE and U is the corresponding solution. In the system above the domain Ω also plays the role of the control variable α from the previous chapter.

In order to discuss a gradient-based optimization technique for problem (6.1), some basics from differential geometry have to be introduced. The nomenclature in this chapter conflicts with the one used in the preceding chapters in some points. While in the first part of this work the dimension of the surrounding space is denoted by d (e.g. $d = 3$ for three dimensional flow domains), in this chapter the surrounding space is denoted by $\mathbb{R}^n$ for the purpose of consistency with the literature.

In the following let $\emptyset \neq M \subset \mathbb{R}^n$. If for each $x \in M$ an open subset $U(x) \subset \mathbb{R}^n$ exists with $x \in U(x)$ and an injective C^∞-mapping $\phi : \tilde{U} \to \mathbb{R}^n$ with $\tilde{U} \subset \mathbb{R}^d$ open and with a continuous inverse mapping $g_x := \phi^{-1} : \phi(\tilde{U}) \to \tilde{U}$ such that

$$\phi(\tilde{U}) \subset U \cap M, \tag{6.2}$$

M is called a d-dimensional C^∞-*submanifold* of $\mathbb{R}^n$.

ϕ is called a local *parametrization* describing the mapping of the parameter space in $\mathbb{R}^n$ to a neighborhood of the point x in the submanifold M. Further, the inverse of the parametrization together with the image of the parametrization $(g_x, \phi(\tilde{U}))$ is called a *chart*.

Let $\Omega \subset \mathbb{R}^n$ be a smooth domain of class C^k and let $\Gamma := \bar{\Omega} \setminus \Omega$ be the surface of Ω. Then Γ is an $(n-1)$-dimensional C^k-submanifold of $\mathbb{R}^n$ (cf. [13]). Assuming a point $x \in \Gamma$ on the surface of Ω, the structure of the submanifold yields a local parametrization ϕ and a parameter $\xi \in \mathbb{R}^{n-1}$ such that $x = \phi(\xi)$. Then the *tangent space* in x to Γ is denoted by $T_x\Gamma$ and is given by

$$T_x\Gamma := \operatorname{span}\{D\phi(\xi)e^i, i = 1, \ldots, n-1\} \tag{6.3}$$

where $e^i = (\overbrace{0, \ldots, 0, 1}^{i}, 0, \ldots, 0)^T$ denotes the i-th vector of the canonical basis in $\mathbb{R}^n$. Further, the unit normal vector on Γ in x is given by

$$\vec{n}(x) = \frac{D\phi(\xi)^{-T} e^n}{\|D\phi(\xi)^{-T} e^n\|_2}, \tag{6.4}$$

which can be checked by forming the scalar product with an basis element of the tangent space

$$\left\langle D\phi(\xi)e^i, D\phi(\xi)^{-T} e^n \right\rangle = \left\langle e^i, e^n \right\rangle = 0. \tag{6.5}$$

With the geometric description of the shape in hand, the next step is to define differential calculus within submanifolds as a preparation for shape optimization.

Let $f : \Omega \to \mathbb{R}$ be a continuously differentiable function for an $(n-1)$-dimensional C^k-submanifold $\Omega \subset \mathbb{R}^n$, then the *tangential gradient* is defined by the orthogonal projection of the classical gradient onto the tangent space of the submanifold. For this purpose, it is assumed that $\tau_1, \ldots, \tau_{n-1}$ form an orthonormal basis of the tangent space. Then the tangential gradient is given by

$$\nabla_\Omega f = \sum_{i=1}^{n-1} \frac{\partial f}{\partial \tau_i} \tau_i \in \mathbb{R}^{n-1}. \tag{6.6}$$

Let further $V : \Omega \to \mathbb{R}^n$ be a differentiable vector field. Then the *tangential divergence* of V is similarly to the tangential gradient given by

$$\operatorname{div}_\Omega V = \sum_{i=1}^{n-1} \left\langle \frac{\partial V}{\partial \tau_i}, \tau_i \right\rangle \in \mathbb{R}. \tag{6.7}$$

Later in this chapter, the following property of the tangential divergence will be used

$$\operatorname{div}_\Omega V = \operatorname{div} V - \left\langle DV\vec{n}, \vec{n} \right\rangle. \tag{6.8}$$

Furthermore, the curvature κ of the submanifold Ω is defined by the tangential divergence of the unit normal vector field

$$\kappa := \operatorname{div}_\Omega \vec{n}. \tag{6.9}$$

The next step is the definition of deformations on the submanifold for the purpose of shape optimization. Thus, in the following Ω is assumed to be a d-dimensional C^k-submanifold of

$\mathbb{R}^n$. It should be remarked that in the practical examples in this chapter the dimension of the submanifold Ω is also given by $d = n$ and the C^k-submanifold Γ is of dimension $n-1$. Like in the previous chapters, Ω corresponds to the domain of the PDE, which is the surrounding space of the airfoil. Thus, the boundary of Ω and especially the surface of the airfoil is denoted by Γ and is assumed to be an $n-1$-dimensional submanifold. Let $T_t : \mathbb{R}^n \to \mathbb{R}^n$ with $t \in \mathbb{R}$ be a family of bijective mappings. Then the *deformation of the submanifold* Ω is defined by

$$\Omega_t := T_t(\Omega) = \{T_t(x) : x \in \Omega\}. \tag{6.10}$$

The most natural choice of the deformation T_t is the so-called *perturbation of identity*. Let $V : \mathbb{R}^n \to \mathbb{R}^n$ be a vector field, which should also be C^k. Then the deformation is given by

$$T_t(V) := \mathrm{Id} + tV\ , \quad T_t(V)(x) = x + tV(x) \tag{6.11}$$

where Id is the identity mapping in $\mathbb{R}^n$. The notation $T_t(V)(x)$ clarifies the dependence of the deformation on the vector field V and is used within this chapter in a comparable way to directional derivatives.

An alternative approach of defining the deformation, often encountered in literature, is the so-called *speed method* where $V : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is not constant in time, but describes a velocity field for the deformation such that $x_0 \in \Omega$ moves according to

$$\frac{\partial x}{\partial t} = V(t, x)\ , \quad x(0) = x_0. \tag{6.12}$$

The advantage of this approach is that the direction of the deformation offers more flexibility (cf. [71]). However, in this chapter the perturbation of identity is considered.

Before a gradient depending on a shape can be formulated, it has to be defined what the differentiability of a shape means. The definition given in [13] is slightly technical and most of the requirements therein only play a role in the derivation of the shape gradients used at the end of this chapter. Yet, for the sake of completeness this formal definition is presented here.

Let $D \subset \mathbb{R}^n$ be open and $\Omega \subset D$ measurable. Further, let V be a vector field, as defined in the speed method, such that $V(\cdot, x)$ is continuous for a small interval $[0, \epsilon]$ and $V(t, \cdot) \in \mathcal{D}^k(D, \mathbb{R}^n)$, where $\mathcal{D}^k$ denotes the set of k times continuously differentiable functions with compact support in D. V is additionally defined to fulfill $\langle V, \vec{n} \rangle = 0$ on ∂D, which states that no deformations over the boundary of D are allowed. Let $\mathcal{P}$ denote the power set and $\mathcal{J} : \mathcal{P}(D) \to \mathbb{R}$ be a scalar shape functional. The *Eulerian derivative* of $\mathcal{J}$ in direction V evaluated in Ω is then defined as

$$d\mathcal{J}(\Omega)(V) := \lim_{t \downarrow 0} \frac{\mathcal{J}(\Omega_t) - \mathcal{J}(\Omega)}{t} \tag{6.13}$$

with the deformed domain

$$\Omega_t = T_t(V)(\Omega). \tag{6.14}$$

If there exists an Eulerian derivative $d\mathcal{J}(\Omega)(V)$ for all directions V and moreover the mapping $V \mapsto d\mathcal{J}(\Omega, V)$ is linear and continuous, $\mathcal{J}$ is called *shape differentiable* in Ω.

With these definitions a version of the so-called *Hadamard theorem* can be formulated. Let $\mathcal{J}$ be shape differentiable and $V \in C^k(\bar{D}, \mathbb{R}^n)$ be given similarly to the definitions above. Then it holds

$$d\mathcal{J}(\Omega)(V) = d\mathcal{J}(\Gamma)(\langle V, \vec{n} \rangle\, \vec{n}). \tag{6.15}$$

A proof can be found in [13].

More precisely, the theorem states that equation (6.15) holds for all vector fields V that are in the same equivalence class with respect to the quotient space

$$Q(\bar{D}, \Omega) = C^k(\bar{D}, \mathbb{R}^n)/F(\Omega) \tag{6.16}$$

with

$$F(\Omega) = \left\{ V \in C^k(\bar{D}, \mathbb{R}^n) : \langle V, \vec{n} \rangle = 0 \text{ on } \partial\Omega \right\}. \tag{6.17}$$

This states that the tangential portion of the deformation vector field on the surface $\partial\Omega$ has no impact on the derivative of the shape functional $d\mathcal{J}(\Omega)(V)$ in equation 6.15. Thus, on the right hand side of the equation only the normal component $\langle V, \vec{n} \rangle \vec{n}$ appears, which is a simplification of the original statement given in [13]. The presentations in [13, 72] are followed further. A generic shape functional of the form

$$\mathcal{J}(\Omega) = \int_{\Omega} f \, d\Omega \tag{6.18}$$

is considered.

Let again $\Omega \subset \mathbb{R}^n$ be measurable, $V \in \mathcal{D}^k(\mathbb{R}^n, \mathbb{R}^n)$ a vector field and assume the shape functional to be $f \in W^{1,1}(\mathbb{R}^n)$. Then it can be proven that the shape derivative of $\mathcal{J}$ is given by

$$d\mathcal{J}(\Omega)(V) = \int_{\Gamma} f \, \langle V, \vec{n} \rangle \, dS. \tag{6.19}$$

Here $W^{p,q}(\Omega)$ with $p \in \mathbb{N}_0$ and $q \in \mathbb{N}$ is the Sobolev space defined as the closure of $C^\infty(\bar{\Omega})$ with respect to the following norm

$$\|f\|_{W^{p,q}(\Omega)} = \sqrt[q]{\sum_{|\alpha| \leq p} \int_{\Omega} \left| \frac{\partial^\alpha f}{\partial x^\alpha} \right|^q d\Omega}. \tag{6.20}$$

Note that this is an extension to the definition given in (2.42). For shape functionals of the form

$$\mathcal{J}(\Omega) = \int_{\Gamma} h dS, \tag{6.21}$$

i.e. only depending on the boundary, the shape derivative is given by

$$d\mathcal{J}(\Omega)(V) = \int_{\Gamma} \langle V, \vec{n} \rangle \left(\frac{\partial h}{\partial \vec{n}} + \kappa h \right) dS. \tag{6.22}$$

In order to prove the equation above, some additional assumptions have to be made. Let Γ be the boundary of Ω such that Γ is a C^k-submanifold of dimension $n-1$ of $\mathbb{R}^n$. Further, it is assumed that $h \in W^{2,1}(\mathbb{R}^n)$ and that $\frac{\partial h}{\partial \vec{n}}$ exists.

Together with this generic volume and surface shape functional, two different shape optimization problems are considered in the subsequent sections. First, the minimization of energy

dissipation is investigated for Stokes flows. Here the objective functional is given by a volume integral leading to a gradient of the form (6.19). This simple test case is used to acquire some basic experiences with PDE constrained shape optimization based on higher order discretization methods. As observed in chapter 3, it is essential that the smoothness of the boundary is maintained during an optimization process in order to guarantee stability and reasonable solutions of the PDE.

With the experiences gained in the simple Stokes case a higher order shape optimization in an Euler flow is investigated. The objective is the drag functional given in (5.19), which is a surface integral. Thus, a gradient of the form (6.22) is applied. The shape gradients arising in the subsequent sections are slightly more complicated than (6.22) and (6.19) since the PDE constraint also has to be considered in form of the adjoint PDE solution. A derivation of the shape sensitivities for Stokes and Euler equations can be found in [72].

6.2 Higher Order Shape Optimization in Stokes Fluids

In this section the Stokes equations are discussed. This simplified flow model is chosen in order to investigate a method of smoothing the shape gradient such that it is applicable as a deformation to the underlying discretization mesh. The mesh deformation tool presented in chapter 3 takes the displacement information of the boundary as Dirichlet condition and deforms the interior mesh according to a linear elasticity solution. However, the curvature information comes from a spline interpolation and it can be assumed that it is smooth enough. The shape gradient, in contrast, is not necessarily smooth and especially in the case of discontinuous basis functions in the DG method an additional smoothing step is obligatory. Thus, an approach is investigated in which the shape gradient is applied as Neumann boundary condition in the mesh deformation tool. This means to interpret the shape gradient as a force on the boundary and not as a prescribed deformation of the shape.

In literature the problem of non-smooth shape gradients is commonly treated by applying the Laplace-Beltrami operator on the gradient in order to obtain smooth deformation information, e.g. in [74]. This smoothed vector field is then applied as a prescribed deformation to the boundary. After that the mesh is either deformed according to the techniques described in chapter 3 or a mesh generator is used to obtain a new mesh for the deformed surface. In the following presentations these two approaches are compared on the basis of a shape optimization in a Stokes flow.

The Stokes equations are simplifications of the Navier-Stokes equations under the assumption that inertial forces are negligible and viscous forces are dominant. This is a good approximation for fluids at low velocities with large viscosity. Like in the Euler and Navier-Stokes flow in chapter 4 let Ω be a subset of $\mathbb{R}^d$ with boundary Γ. First, two dimensional optimizations are considered and then the three dimensional case is discussed, thus $d = 2$ and $d = 3$, respectively. Again let $\boldsymbol{u} : \Omega \to \mathbb{R}^d$ denote the velocity field, $p : \Omega \to \mathbb{R}$ the pressure of the fluid and μ the

viscosity. Then the Stokes equations are given by

$$\begin{aligned} \nabla p - \mu \Delta \boldsymbol{u} &= 0 && \text{in } \Omega \\ \nabla \cdot \boldsymbol{u} &= 0 && \text{in } \Omega \\ \boldsymbol{u} &= 0 && \text{on } \Gamma_{\text{wall}} \text{ and } \Gamma_{\text{obs}} \\ \boldsymbol{u} &= \boldsymbol{u}^{+} && \text{on } \Gamma_{\text{in}} \\ p\vec{n} - \mu \frac{\partial \boldsymbol{u}}{\partial \vec{n}} &= 0 && \text{on } \Gamma_{\text{out}} \end{aligned} \tag{6.23}$$

where Γ_{wall} is the part of the boundary describing the walls of the flow field, Γ_{obs} denotes the boundary of the obstacle in the flow field and Γ_{in} and Γ_{out} is the inflow and outflow boundary, respectively. The corresponding energy dissipation minimization problem is then given by

$$\min_{(\boldsymbol{u},p,\Omega)} \mathcal{J}(\boldsymbol{u}, p, \Omega) = \mu \int_{\Omega} \sum_{i,j=1}^{d} \left(\frac{\partial u_i}{\partial x_j} \right)^2 d\Omega. \tag{6.24}$$

Additionally, for the optimization to be reasonable it is required that the volume of the obstacle remains unchanged for all deformed shapes Ω_t, which is given by

$$\text{Vol}(\Omega_t) = \int_{\Omega_t} 1 \, d\Omega \overset{!}{=} \int_{\Omega_0} 1 \, d\Omega = \text{Vol}(\Omega_0) \quad \forall t \geq 0. \tag{6.25}$$

This condition actually states that the volume of the flow field is constant, however inflow, outflow and wall boundaries are considered as fixed in the optimization. Thus, the volume of the obstacle represented by a hole in the discretization mesh is also forced to be constant.

This is an attractive test case in terms of shape optimization since it is intensively investigated in [75] and also in [74] where theoretical results are presented. These results indicate that in a two dimensional Stokes flow (6.23) the optimal shape with respect to condition (6.25) and objective functional (6.24) must have a conical shaped leading and trailing edge with an angle of 60°.

The shape derivative of the energy dissipation functional subject to the Stokes equations is then given by

$$d\mathcal{J}(\boldsymbol{u}, p, \Omega)(V) = -\mu \int_{\Gamma_{\text{obs}}} \langle V, \vec{n} \rangle \sum_{i=1}^{d} \left(\frac{\partial \boldsymbol{u}_i}{\partial \vec{n}} \right)^2 dS. \tag{6.26}$$

Due to the self-adjoint nature of the Stokes PDE, this derivative does not depend on an adjoint variable but only on the primal solution $\boldsymbol{u}$. In the following let the shape gradient be denoted by

$$s(\boldsymbol{u}) = -\mu \sum_{i=1}^{d} \left(\frac{\partial \boldsymbol{u}_i}{\partial \vec{n}} \right)^2. \tag{6.27}$$

Then, like in classical gradient-based optimization approaches, the steepest descent direction is chosen as the negative gradient. Thus, the deformation vector field V can be chosen as

$$V = -s \cdot \vec{n} \quad \text{on } \Gamma_{\text{obs}}. \tag{6.28}$$

The values of V in $\Omega \setminus \Gamma_{\text{obs}}$ are then given implicitly by the mesh deformation technique. The next step is to include the volume constraint into the optimization. For this purpose, it is useful to formulate the Lagrange function with multipliers λ_1 and λ_2

$$L(U, \Omega, \lambda_1, \lambda_2) = \mathcal{J}(U, \Omega) + \langle \lambda_1, c(U, \Omega) \rangle + \langle \lambda_2, h(\Omega) \rangle \tag{6.29}$$

where $U = (\boldsymbol{u}, p)$ denotes the vector of unknowns, $c(U, \Omega) = 0$ denotes the PDE constraint and $h(\Omega) = \text{Vol}(\Omega) - \text{Vol}(\Omega_0) = 0$ the volume constraint. Then the necessary optimality condition connected to the volume constraint is given by

$$dL(U, \Omega, \lambda_2) = \int_{\Gamma_{\text{obs}}} \langle V, \vec{n} \rangle \, s(\boldsymbol{u}) dS + \lambda_2 \int_{\Gamma_{\text{obs}}} \langle V, \vec{n} \rangle \, dS = 0, \tag{6.30}$$

since the shape derivative of $\text{Vol}(\Omega)$ is obtained by applying (6.19), which results in the second integral on the right hand side. From this it follows that after each gradient step in the optimization the volume of the original obstacle has to be recovered by scaling the deformed shape with a factor of λ_2. This scaling factor is obtained from the Lagrange multiplier

$$\lambda_2 = -\frac{\int_{\Gamma_{\text{obs}}} \langle V, \vec{n} \rangle \, s(\boldsymbol{u}) dS}{\int_{\Gamma_{\text{obs}}} \langle V, \vec{n} \rangle \, dS}. \tag{6.31}$$

A Lagrange multiplier λ_1 which is connected to the state variable of the PDE is not visible in this formulation due to the self-adjoint nature of the Stokes problem.

In most situations, especially in higher order discontinuous Galerkin discretizations, the shape gradient s is not smooth enough for a direct application to the shape. This would result in kinks in the geometry leading to the effects visualized in figure 3.1. Thus, s has to be smoothed in some way before deforming the shape. The classical approach is to apply the tangential Laplace operator Δ_Γ, often referred to as the Laplace-Beltrami operator, on s. In the terminology of this chapter the Laplace-Beltrami operator can be obtained by applying the tangential divergence to the tangential gradient

$$\Delta_\Gamma = \text{div}_\Gamma \cdot \nabla_\Gamma. \tag{6.32}$$

The actual smoothing is then conducted by inserting the shape gradient s as a source term into the following PDE

$$(\text{Id} + \eta_{\text{LB}} \Delta_\Gamma) \tilde{s} = s \tag{6.33}$$

where Id is the identity. The parameter $\eta_{\text{LB}} > 0$ is chosen in order to control how much the shape gradient is diffused. In [72] it is moreover discussed that the application of (6.33) to the shape gradient results in an approximation to the shape Hessian and thus brings higher order derivative information into the optimization. This technique is shown to speed up convergence significantly. In many applications the smoothed shape gradient $\tilde{s}$ is then used together with the normal vector field in order to deform the surface of the obstacle. After that a mesh generator is invoked which produces a discretization for the deformed domain Ω_t. However, a mesh deformation tool in hand as described in chapter 3, it is tempting to include the smoothing into the mesh deformation and execute all at once.

For this purpose, a deformation vector field $\boldsymbol{u}_D$ is introduced, which defines the displacement of the mesh nodes in Ω. The shape gradient s is then interpreted as a force on the boundary Γ_{obs} of the mesh and either the linear elasticity equations, already mentioned in (3.1),

$$\nabla \cdot \sigma = 0 \quad \text{in } \Omega \tag{6.34}$$

or a component-wise Laplacian

$$\eta \Delta \boldsymbol{u}_D = 0 \quad \text{in } \Omega \tag{6.35}$$

are solved with the following boundary conditions

$$\begin{aligned} \boldsymbol{u}_D &= 0 \qquad && \text{on } \Gamma_{\text{in}}, \Gamma_{\text{out}}, \Gamma_{\text{wall}} \\ \frac{\partial \boldsymbol{u}_D}{\partial \vec{n}} &= s\vec{n} \qquad && \text{on } \Gamma_{\text{obs}}. \end{aligned} \tag{6.36}$$

The major difference between (6.34) and (6.35) is that the component-wise Laplacian mesh deformation does not include a coupling of the spatial dimensions. This is especially noticeable in three dimensions where the linear elasticity approach leads to significantly better deformed meshes.

In the following the Laplace-Beltrami and the Neumann approach are compared numerically in the Stokes test case. The initial geometry is chosen to be a circle with diameter 1 around the origin and the dimensions of the flow tunnel are set as $x \in [-4, 6]$ and $y \in [-3, 3]$. In the three dimensional case the flow tunnel is a cylinder with the same dimensions. Further, the viscosity μ is set to 1. The inflow velocity in x-direction is chosen as

$$u^+(\boldsymbol{x}) = \cos\left(\frac{\|\boldsymbol{x}\|\pi}{\text{diam}}\right) \ , \ \boldsymbol{x} \in \Gamma_{\text{in}} \tag{6.37}$$

where diam denotes the diameter of the flow tunnel. The Stokes, linear elasticity, Laplace and Laplace-Beltrami PDEs are assembled using the GETFEM toolbox like in chapter 3 with a higher order finite element discretization. The resulting linear systems are solved using the parallel LU solver MUMPS [76]. The results in the following presentations are obtained by a third order finite element discretization.

The first test setting is the Laplace-Beltrami smoothing on two dimensional grids. One iteration of this algorithm is then given by the following steps:

- Solve Stokes equations
- Evaluate the shape gradient s
- Solve Laplace-Beltrami equation and obtain smoothed gradient $\tilde{s}$
- Evaluate $\tilde{s} \cdot \vec{n}$ and place this as Dirichlet boundary in the mesh deformation and obtain a deformation vector field D_1
- Conduct a mesh deformation with $1 \cdot \vec{n}$ as Dirichlet boundary on Γ_{obs} in order to obtain λ_2 and a second deformation field D_2
- The actual mesh deformation is then given by $D = D_1 + \lambda_2 D_2$

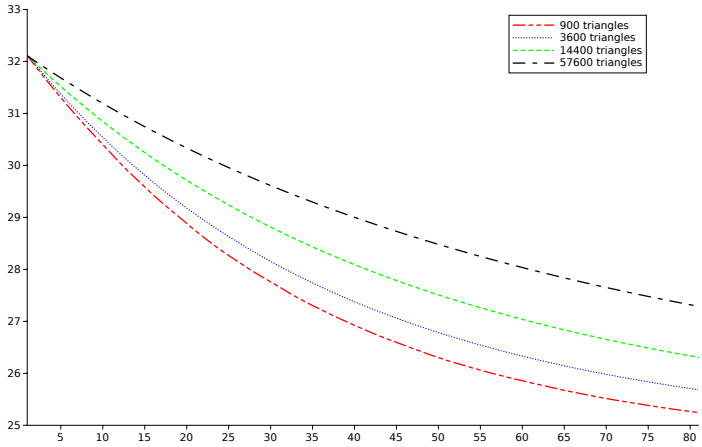


Figure 6.1: First iterations of shape optimization based on Laplace-Beltrami smoothing and Dirichlet boundary condition in linear elasticity mesh deformation on refined grids for fixed η_{LB}

This approach is tested on four grids starting with 900 triangles which is successively refined up to a grid with 57,600 cells. For the Laplace-Beltrami smoothing to work properly two constants have to be chosen. First, the diffusion coefficient η_{LB} which controls the smoothness of the gradient and second, the step size of $\tilde{s}\vec{n}$ has to be damped by a factor d_{LB}. In the present situation $d_{\text{LB}} = 10^{-2}$ shows to be sufficient for all grids in order not to introduce kinks or discontinuities into the shape. For the test in figure 6.1 $\eta_{\text{LB}} = 5$ is chosen constantly. This parameter shows to be sufficiently large on the coarse grid, but does not oversmooth the gradient on the fine grid. The figure depicts the objective functional on four successively refined grids. It can be seen that its development during the optimization depends on the grid.

Figure 6.2 visualizes the situation for the grid with 14,400 cells and a smoothing parameter $\eta_{\text{LB}} \in [1, 10]$. Again, the convergence is mesh-dependent and the converged shapes vary depending on η_{LB}, which cannot be seen in this figure.

The optimization is even more sensitive with respect to η_{LB} and d_{LB} when switching from two dimensions to three dimensions leading to a totally different set of adequate parameters. Here the discretization is chosen such that the element density on the surface of the obstacle is approximately comparable to the two dimensional case evaluated in figure 6.2. This leads to a discretization with approximately 40,200 elements and a set of parameters of $\eta_{\text{LB}} = 10^2$ and $d_{\text{LB}} = 1$. The optimization also works for other parameters, but these ones are chosen in order to mimic a comparable convergence history.

The mesh deformation in the Laplace-Beltrami setting is conducted using either the linear elasticity or the component-wise Laplacian approach. In this situation it makes no difference for the shape but only for the interior of the mesh since the displacement described by the smoothed shape gradient $\tilde{s} \cdot \vec{n}$ is applied as a Dirichlet boundary condition. Thus, the objective function is not influenced by the choice of the mesh deformation approach.

One major drawback of the method described and tested above is the dependence of the involved parameters on the mesh. This is even more visible in the Euler test case in the next

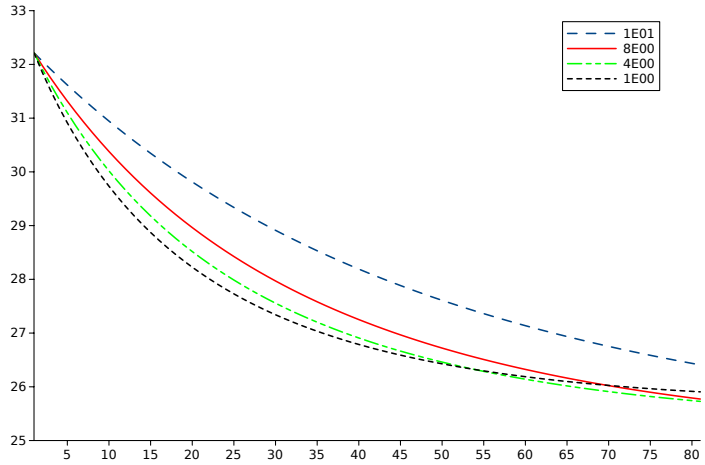


Figure 6.2: Same situation like in figure 6.1, mesh with 14,400 triangles and different Laplace-Beltrami coefficients η_{LB}

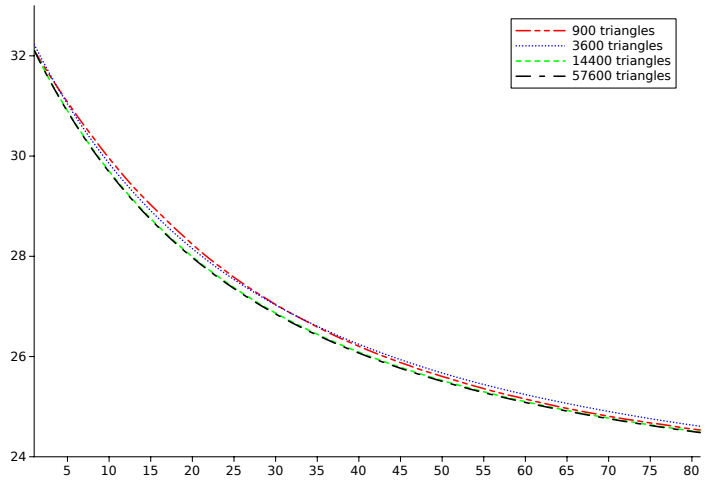


Figure 6.3: Mesh-independent trend of objective functional in shape optimization based on Neumann boundary condition in component-wise Laplacian mesh deformation

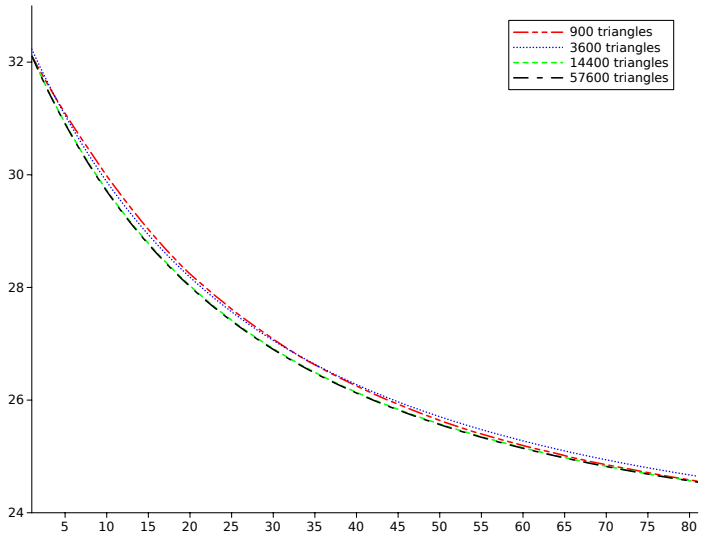


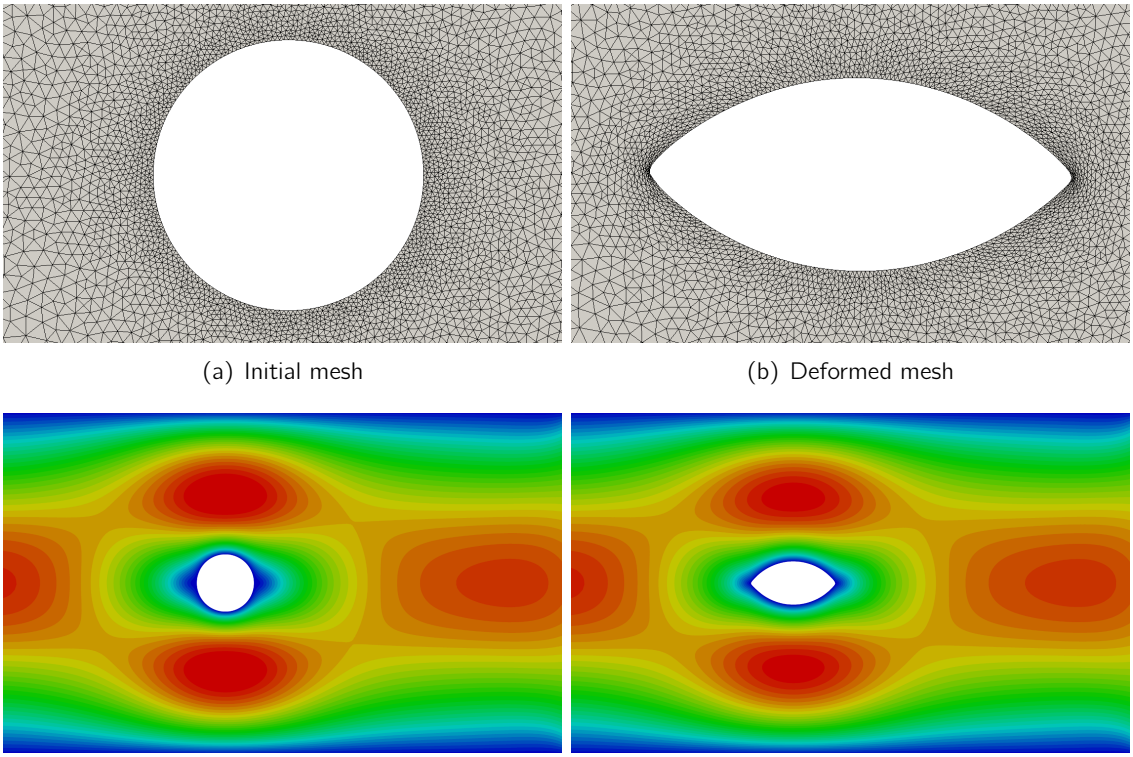
Figure 6.4: Mesh-independent trend of objective functional in shape optimization based on Neumann boundary condition in linear elasticity mesh deformation

section.

As mentioned earlier, a different approach is to use the unmodified shape gradient s and interpret it as a force $s \cdot \vec{n}$ to the obstacles boundary. Hereby the shape gradient is implicitly smoothed, which makes the additional step of solving the Laplace-Beltrami equations obsolete. Figure 6.3 and 6.4 visualize this situation. The first figure shows a mesh deformation based on a component-wise Laplacian and the second covers the linear elasticity based deformation. In both situations the shape gradient is included as described in equation (6.36). For the Laplacian $\eta = 5 \cdot 10^2$ is chosen and in the linear elasticity equation Young's modulus is chosen as $\mathcal{E} = 10^3$ and Poisson's ratio as $\nu = 0.3$.

It can be seen that in both tests the convergence on all meshes is comparable without adapting the parameters. Moreover, the optimal shapes on all meshes are comparable. It should be remarked that the number of optimization steps is chosen to be about 80 in all tests since in both the Neumann and the Laplace-Beltrami approach the step-size is a degree of freedom. In the Neumann approach it is controlled implicitly by the material coefficients in the linear elasticity equations or by the diffusion parameter in the Laplace equation. The most interesting fact is that switching from two to three dimensions does not require changes in the parameters or the code and does also lead to the same convergence after the same number of iterations. Thus, it can be said, that this approach is truly mesh-independent.

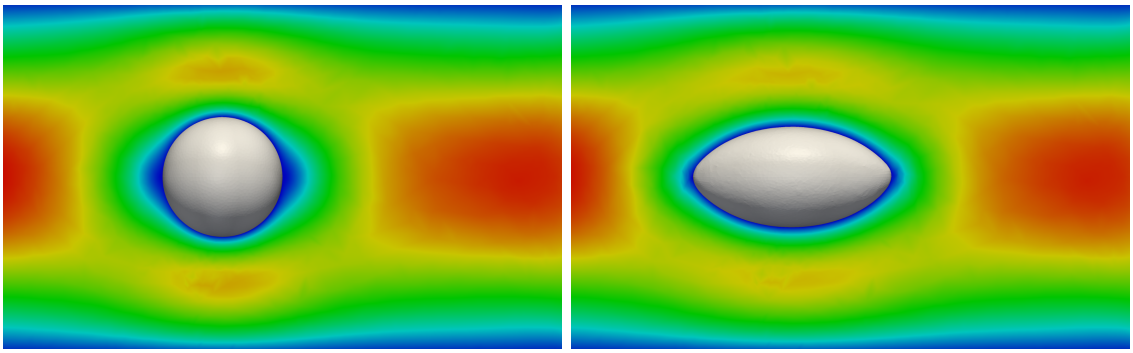
The results of the optimizations can be seen in figure 6.5. Here the two dimensional shape optimization is depicted, starting with the initial geometry on the left hand side and the optimal shape on the right hand side. It can be seen that this shape is close to the optimal shape which is theoretically investigated in [75]. Figure (a) and (b) show the effect of the linear elasticity mesh deformation and (c) and (d) show the shapes with the Stokes solution. The three dimensional case is covered in figure 6.6 which leads to very similar results.



(a) Initial mesh (b) Deformed mesh

(c) Initial geometry (d) Optimized geometry

Figure 6.5: Two dimensional shape optimization with respect to energy dissipation in a Stokes flow, (a) and (b) visualize the mesh deformation, in (c) and (d) color denotes the norm of the velocity field



(a) Initial geometry (b) Optimized geometry

Figure 6.6: Three dimensional shape optimization with respect to energy dissipation in a Stokes flow, color denotes the norm of the velocity field on a slice through the center of the flow field

In summary, it can be observed that applying the shape gradient as a boundary force in the mesh deformation brings two advantages. On the one hand, there is no need for an additional smoothing step as outline with the Laplace-Beltrami operator. On the other hand, the coefficients in the Laplace-Beltrami setting are mesh-dependent and have to be chosen empirically, which can be a time consuming process. In contrast, the mesh deformation based on Neumann boundary conditions showed to perform comparably on different meshes and even in two and three dimensions in the present higher order FEM discretization of the Stokes equations.

6.3 DG One Shot Shape Optimization in Euler Flows

In this section the techniques tested with Stokes flows in the previous section are applied to a higher order Euler shape optimization. The aim is to introduce small deformations to the ONERA M6 wing presented in chapter 4 in order to reduce the drag. Since this is a transsonic flow configuration, on the upper surface of the airfoil there is a large shock in the solution which induces drag. Thus, the overall drag can be reduced by smoothing out the shock which is conducted by geometric deformations in these regions. The mesh deformation is again based on the linear elasticity solver presented in chapter 3.

Since lift and drag functional, as introduced in equations (5.18) and (5.19), only differ in the rotation vector which is multiplied with the pressure coefficient, a shape derivative for both can be given in one formula. Moreover, due to the linear dependence of the pressure coefficient on the pressure itself (cf. (5.17)), for an optimization with respect to lift or drag it is sufficient to derive a gradient with respect to the following objective functional

$$\mathcal{J}(U,\Omega) = \int_{\Gamma_{\text{obs}}} \langle p_\alpha, \vec{n} \rangle \, dS. \tag{6.38}$$

Here p_α is given by one of the following expressions

$$p_\alpha^{\text{drag}} = p \cdot \begin{pmatrix} \cos(\alpha) \\ 0 \\ \sin(\alpha) \end{pmatrix} \quad \text{or} \quad p_\alpha^{\text{lift}} = p \cdot \begin{pmatrix} -\sin(\alpha) \\ 0 \\ \cos(\alpha) \end{pmatrix}. \tag{6.39}$$

The shape derivative depending on a vector field V is then given by

$$\begin{aligned} d\mathcal{J}(U,\Omega)(V) &= \int_{\Gamma_{\text{obs}}} \langle V, \vec{n} \rangle \left[\frac{\partial p_\alpha}{\partial \vec{n}} \vec{n} - \lambda U_H \left\langle \frac{\partial \boldsymbol{u}}{\partial \vec{n}}, \vec{n} \right\rangle + \operatorname{div}_\Gamma \left(p_\alpha - \lambda U_H \boldsymbol{u} \right) \right] dS \\ &= \int_{\Gamma_{\text{obs}}} \langle V, \vec{n} \rangle \left[\frac{\partial p_\alpha}{\partial \vec{n}} \vec{n} - \lambda U_H \left\langle \frac{\partial \boldsymbol{u}}{\partial \vec{n}}, \vec{n} \right\rangle + \kappa \langle p_\alpha, \vec{n} \rangle \right] \\ &\qquad + \langle p_\alpha - \lambda U_H \boldsymbol{u}, d\vec{n}(V) \rangle \, dS. \end{aligned} \tag{6.40}$$

A derivation and proof can be found in [14].

In equation (6.40) U_H denotes the vector of conserved variables $(\rho, \rho u, \rho v, \rho w, \rho E)$ where the last component ρE is replaced by ρH in terms of the enthalpy H. λ denotes the adjoint variable which is computed according to the presentations in chapter 5 and the curvature of the surface is given by κ. Finally, $d\vec{n}(V)$ denotes the shape derivative of the normal vector field in the direction of V. Like in the previous section Γ_{obs} denotes the mesh boundary representing the surface of the obstacle. In the mesh deformation this is handled as a Neumann boundary with the shape gradient as a force. Moreover, the symmetry wall where the wing is fixed is also modeled by a Neumann boundary condition with zero flux. This allows discretization elements to slide within the plane, but not to leave it. In the terminology of chapter 3 this boundary is denoted by Γ_N.

In contrast to the Stokes case here the artificial boundary, introduced by the bounding box of the curved cells (cf. figure 3.3), plays the role of Γ_{D_2} and states a zero Dirichlet condition in the mesh deformation. Thus, the shape deformation only affects cells which are curved in the original discretization mesh. This ensures that non-curved cells will remain straight-sided throughout the whole optimization. Moreover, the PDE of the mesh deformation only has to be solved within the marked volume which greatly reduces the costs.

For what follows in this chapter, the first one of the two equivalent formulations (6.40) is used as the calculation of the tangential divergence as it is more convenient in the present framework of the DG solver. Since the expression $p_\alpha - \lambda U_H \boldsymbol{u}$ not only exists on the boundary Γ_{obs}, the classical divergence can be computed according to the DG scheme (2.8) by replacing the Euler fluxes with central differences and then projecting onto the submanifold Γ_{obs} according to formula (6.8). Thus, in the following the shape gradient is given by

$$s = \left[\frac{\partial p_\alpha}{\partial \vec{n}} \vec{n} - \lambda U_H \left\langle \frac{\partial \boldsymbol{u}}{\partial \vec{n}}, \vec{n} \right\rangle + \operatorname{div}_\Gamma \left(p_\alpha - \lambda U_H \boldsymbol{u} \right) \right]. \tag{6.41}$$

Like in the Stokes case, the optimization is again constrained by a fixed volume of the obstacle.

The optimization is then conducted in a one-shot sense. This terminology was first used in [77] and describes an optimization approach where the intermediate state and adjoint variable are not feasible during the iterations, but only in the converged solution. In the case that the PDE and its adjoint are solved by iterative techniques, the solver is stopped when a desired residual is reached and not after convergence to machine precision. Thus, state and adjoint variable are not feasible during the iterations. By this approach the convergence of the optimization can be greatly accelerated.

In the present case a similar strategy is applied. First, the discretized PDE and its adjoint are once solved up to the residual described in sections 4.3.1 and 5.2. A relative residual of 10^{-8} is reached after 512,000 iterations in the primal case and 485,000 in the adjoint case, respectively. Then the first design update is conducted. After that the primal and adjoint solver are iterated up to a relative residual of 10^{-3} which is reached after approximately 20,000 iterations in each case. It should be remarked that both the primal and adjoint approximation, which were calculated before the design update, are used as initial conditions after the update. Thus, the solvers start with a much smaller absolute residual than in the case of a homogeneously initialized flow field.

In order to maintain the outline of the wing, the shape gradient is slightly modified. Since it is the aim to minimize the drag with respect to the shock on the upper surface, the shape

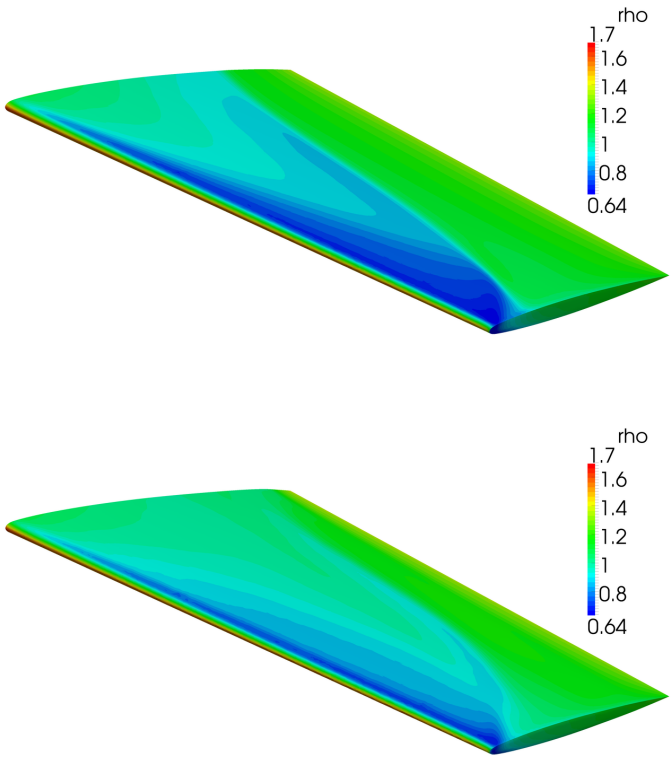


Figure 6.7: Shape optimization of ONERA M6 wing with respect to drag, 10 % drag reduction due to smoothing of the shock with minimal geometrical deformations and volume constraints

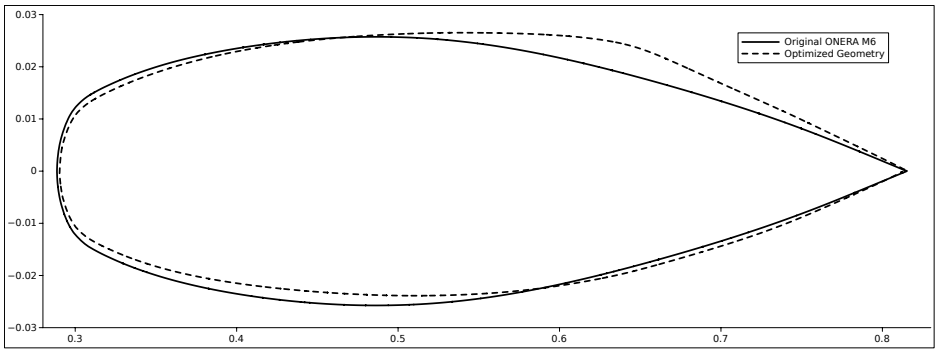


Figure 6.8: Initial and optimized geometry of ONERA M6, cross-section at 0.5 of the span width

gradient is weighted in the vicinity of the leading and trailing edge with zero. For this purpose, a set of m planes

$$P^i = \left\{\bar{x}^i, \vec{n}^i\right\} \ , \ \bar{x}^i, \vec{n}^i \in \mathbb{R}^3 \quad \forall 1 \leq i \leq m \tag{6.42}$$

is assumed which describes the outline of the geometry. Here $\bar{x}^i$ is a point on the i-th plane and $\vec{n}^i$ the corresponding normal vector. For each node x on the surface Γ_{obs} the distance to the closest plane can be measured in the Euclidean norm by

$$\delta(x) = \min_{1 \leq i \leq m} \left\langle x - \bar{x}^i, \vec{n}^i \right\rangle . \tag{6.43}$$

The normal vectors $\vec{n}^i$ are chosen such that $\delta(x)$ is negative if x is outside the region which is to be deformed and positive if it is inside. The shape gradient is now smoothly weighted if the corresponding node is within a specified range r of one plane which is done by

$$\tilde{s}(x) = s(x) \cdot \frac{1}{2}\left[\sin\left(\frac{\delta(x)\cdot\pi}{r} - \frac{\pi}{2}\right) + \frac{1}{2}\right] \ , \ \forall x \in \Gamma_{\text{obs}} : 0 \leq \delta(x) \leq r. \tag{6.44}$$

By applying equation (6.44) to the shape gradient s in a point $x \in \Gamma_{\text{obs}}$, the shape gradient is weighted with a factor in $[0, 1]$. This factor depends on how close the point x is to the closest plane P^i. If x is inside the bounding box defined by the planes P^i and if the minimum distance is smaller than r, then the shape gradient is weighted by $\frac{1}{2}\left[\sin\left(\frac{\delta(x)\cdot\pi}{r} - \frac{\pi}{2}\right) + \frac{1}{2}\right]$. This term is chosen as a smooth transition between 0 and 1 depending on the distance $\delta(x)$.

In the present case two planes are defined: one through the leading edge given by $\bar{x}^1 = (0, 0, 0)^T$, $\vec{n}^1 = (0.866, 0, -0.5)^T$ and one through the trailing edge given by $\bar{x}^1 = (0.65, 0, 0)^T$, $\vec{n}^1 = (-0.966, 0, 0.2588)^T$. Furthermore, the range of the modification is chosen to be $r = 0.01$.

Figure 6.7 shows the effect of the shape deformation on the upper surface of the ONERA M6 wing. Moreover, figure 6.8 visualizes a cross-section of the wing at 50% of the span width where the y-dimension is stretched in order to visualize the geometry deformation. It can be seen that the shock is smoothed out which leads to a reduction of the drag by 10% after 15 optimization steps. This reductions might appear slightly small, however it can be explained due to the geometric restrictions that the leading and trailing edge are fixed. The shape gradient shows its maximum values typically at the leading and trailing edge which indicates that the drag is very sensitive with respect to these regions.

The computations are all performed with $P = 3$ basis functions. This includes the linear elasticity solver for the initial curved geometry as well as the solver for the mesh deformation, which is also based on the linear elasticity equations as described in the Stokes case in the previous section. Although the GPU solver works in parallel on mesh partitions, it is not ensured that the surface Γ_{obs} and the region of curved cells Ω_c are also partitioned properly. In the worst case the graph partitioning algorithm attaches all cells of Ω_c and Γ_{obs} to one processor. This issue is not further investigated within this work and the mesh curvature is thus serialized on one processor, which is not critical with respect to runtime since Ω_c only contains a small fraction of the whole mesh. An example can be seen in the right picture of figure 4.11 where the surface is distributed only over 4 out of 8 processors.

The shape optimization is conducted in the following steps:

1. The DG solver is started with the straight-sided grid which is then partitioned.
2. The least squares problem is solved in order to fit the mesh to the NURBS representation.
3. Solution of the least squares problem is applied as Dirichlet boundary condition in the elasticity FEM solver, which results in a deformation field D_0.
4. An interpolation operator between the DG and FEM solution is set up based on the straight-sided mesh.
5. Primal/adjoint solution and shape gradient are computed by the DG solver and the gradient is then gathered in one process through MPI.
6. The shape gradient is interpolated at the nodes of the FEM discretization. The elements of the FEM solver are then curved according to D_0 and the linear elasticity equations are solved with the interpolated shape gradient as Neumann force on Γ_{obs} which yields a deformation field D_1.
7. For the volume constraint the linear elasticity equations are solved once more with $1 \cdot \vec{n}$ as Neumann boundary condition on Γ_{obs} resulting in a field D_2.
8. The Lagrange multiplier λ_2 (cf. section 6.2) is then obtained from the condition
$$\int_{\Gamma_{\text{obs}}} D1 \cdot \vec{n}\, dS - \lambda_2 \int_{\Gamma_{\text{obs}}} D_2 \cdot \vec{n}\, dS = 0. \tag{6.45}$$
9. The new geometry is then described by the deformation field $D_0 \leftarrow D_1 - \lambda_2 D_2 + D_0$ and it is proceeded with step 5.

It might sound surprising that the DG and the FEM solver are based on the straight-sided mesh which is curved according to the accumulated deformation field in each iteration. However, this procedure shows to be more stable with respect to round off errors. If the grid itself is modified in each optimization step, the discretization error of the linear elasticity FEM solution leads to overlappings and diverging elements in the DG scheme after a few iterations.

Summarizing this novel shape optimization approach presented in sections 6.2 and 6.3, the gradient smoothing and mesh deformation based on Neumann boundary conditions in the linear elasticity equations lead to a very robust shape update even in higher order discretizations. Clearly, this approach is much too expensive when applied in the simplified Stokes flow situation. This stems from the fact, that in the solver framework used in section 6.2, the solution of the Stokes equations is only halve as expensive as the shape update, since for the update the linear elasticity equations have to be solved twice.

The situation is very different when applied in the non-linear Euler case. Despite the application of optimized solver techniques, like the combination of explicit GPU time integration and implicit methods, the additional costs for the mesh deformation do not significantly appear in the overall runtime. This is moreover amplified by the fact that the mesh deformation tool only acts on a small subset of cells $\Omega_c \subset \Omega$. Thus, the shape optimization based on linear elasticity mesh deformations with an gradient smoothing through Neumann boundary conditions is also a computationally attractive approach.

Chapter 7

Conclusion and Outlook

7.1 Summary

The aim of this work was to investigate if GPU accelerated higher order discontinuous Galerkin methods could make their way into industrial applications connected to computational fluid dynamics. For this purpose, an explicit Runge-Kutta DG method was completely implemented for the GPU. Simultaneously, an explicit and implicit DG code was implemented for the CPU following well-established guidelines in order to estimate the speedup that can be gained by the different hardware architecture. This includes an OPENMP parallelization for multicore CPUs, which fully enables the computational abilities of the multiple cores of modern CPUs. Comparing the explicit codes on both the GPU and the CPU, a GPU-speedup of up to 5 in double precision and 18 in single precision arithmetic was achieved over the CPU. Whereas, it turned out that the explicit RKDG method is not comparable to the backward Euler DG method in a general framework. It showed to be best practice to apply the GPU-accelerated RKDG code as a preconditioner within a backward Euler or Newton method for the present subsonic Euler test case.

This work was focused on the simulation of a transsonic Euler flow over the ONERA M6 wing. Thus, shock capturing techniques had also to be investigated within an explicit time stepping. Here an artificial viscosity approach showed to be robust and fits seamlessly in the GPU framework. The implementation of this shock capturing tool is very close to a full Navier-Stokes solver which was also implemented and shown within this work.

Since there are no algorithmic differentiation tools available yet, which maintain the code optimizations made for the GPU hardware architecture, the RKDG method was derived by hand in order to obtain a solver for the discrete adjoint problem of the Euler equations. The most interesting result was that in the present DG scheme the discrete adjoint solver is very close to the primal one from an algorithmic point of view. This allowed for comparable run times of the adjoint and the primal code.

Another focus was on the treatment of curved geometries within the higher order DG discretization, which was not planned to investigate in the first place. However, simple test cases had already shown that straight-sided elements induce major errors in the scheme due to artificial kinks arising in the geometry. In order to overcome this issue, techniques which are known from mesh deformation approaches in shape optimization were adopted and used in order to generate body-fitted higher order DG meshes.

At the end of this work all the techniques and codes derived so far where combined in order to conduct a shape optimization of the ONERA M6 wing. Here the overall drag should be reduced by smoothing out the shock on the upper surface of the wing. It turned out that the

shape gradient information obtained from the higher order DG discretization is not smooth enough to apply directly to the shape as a deformation. For this purpose, the well-established Laplace-Beltrami smoothing approach was compared to a shape optimization approach where the shape gradient is interpreted as a force in the linear elasticity mesh deformation tool. This was tested in a simplified Stokes flow and finally applied to the transsonic Euler case.

Summarizing the results of this work, it can be said that the use of higher order discretization methods introduce some technical problems as shock capturing and geometry representation. However, this work showed that the computational attractiveness, especially with respect to stream processors, and high resolution potential outweighs the technical issues.

7.2 Future Work

This work mainly focused on the pure hyperbolic case of the Euler equations. Thus, it is a natural question to ask how explicit schemes perform in viscous flows and if the speedups can be maintained. As indicated in this work, the role of the GPU-accelerated explicit scheme could be seen as a preconditioner in an implicit time stepping which could be further investigated.

Another field of vivid studies is connected to the isoparametric mappings within the discretization scheme. Curved elements in a higher order method are a powerful tool to resolve complicated geometries. However, it has to be further investigated how the known curvature of the physical geometry can be brought into the discretization mesh.

Finally, this work only covered empirical studies dealing with the smoothing approach of the shape gradient by applying it as a force in the mesh deformation step. This should be further investigated and especially some theoretical inside should be gained.

List of Figures

List of Tables

Bibliography

[1] W. H. Reed and T. R. Hill. Triangular mesh methods for the neutron transport equation. *Los Alamos Report LA-UR-73-479*, 1973.

[2] F. Bassi and S. Rebay. A high-order accurate discontinuous finite element method for the numerical solution of the compressible Navier–Stokes equations. *Journal of computational physics*, 131(2):267–279, 1997.

[3] Z.J. Wang, K. Fidkowski, R. Abgrall, F. Bassi, D. Caraeni, A. Cary, H. Deconinck, R. Hartmann, K. Hillewaert, H.T. Huynh, N. Kroll, G. May, P.-O. Persson, B. van Leer, and M. Visbal. High-order CFD methods: current status and perspective. *International Journal for Numerical Methods in Fluids*, 2013.

[4] P. O. Persson and J. Peraire. Sub-cell shock capturing for discontinuous Galerkin methods. *Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit*, 112, 2006.

[5] B. Cockburn and C. W. Shu. The Runge-Kutta local projection P1-discontinuous Galerkin finite element method for scalar conservation laws. *RAIRO, Modélisation Mathématique et Analyse Numérique*, 25:337–361, 1991.

[6] B. Cockburn and C. W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws II: general framework. *Mathematics of Computation*, 52:411–435, 1989.

[7] B. Cockburn, S. Y. Lin, and C. W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: one-dimensional systems. *Journal of Computational Physics*, 84:90–113, 1989.

[8] B. Cockburn, S. Hou, and C. W. Shu. The Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws IV: the multidimensional case. *Mathematics of Computation*, 54:545–581, 1990.

[9] B. Cockburn and C. W. Shu. The Runge-Kutta discontinuous Galerkin method for conservation laws V: multidimensional systems. *Journal of Computational Physics*, 141:199–224, 1998.

[10] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228:7863–7882, 2009.

[11] J. S. Hesthaven and T. Warburton. Nodal high-order methods on unstructured grids: I. time-domain solution of Maxwell's equations. *Journal of Computational Physics*, 181:186–221, 2002.

[12] M. B. Giles and N. A. Pierce. Adjoint equations in CFD: duality, boundary conditions and solution behaviour. *AIAA*, 97:1850, 1997.

[13] J. Sokolowski and J.-P. Zolesio. *Introduction to shape optimization*. Springer, 1992.

[14] S. Schmidt, C. Ilic, V. Schulz, and N. Gauger. Airfoil design for compressible inviscid flow based on shape calculus. *Optimization and Engineering*, 12(3):349–369, 2011.

[15] S. Schmidt, C. Ilic, V. Schulz, and N. Gauger. Three dimensional large scale aerodynamic shape optimization based on shape calculus. In *41st AIAA Fluid Dynamics Conference and Excibit*, number AIAA 2011-3718, 2011.

[16] G. E. Karniadakis and S. J. Sherwin. *Spectral/hp element methods for CFD*. Oxford University Press, USA, 1999.

[17] T. Warburton. An explicit construction of interpolation nodes on the simplex. *Journal of engineering mathematics*, 56(3):247–262, 2006.

[18] G. Szegö. *Orthogonal Polynomials*, volume 23. American Mathematical Society, 1939.

[19] Q. Chen and I. Babuška. The optimal symmetrical points for polynomial interpolation of real functions in the tetrahedron. *Computer methods in applied mechanics and engineering*, 137(1):89–94, 1996.

[20] J. S. Hesthaven and C. H. Teng. Stable spectral methods on tetrahedral elements. *SIAM Journal on Scientific Computing*, 21(6):2352–2380, 2000.

[21] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer, 2008.

[22] A. H. Stroud. *Approximate calculation of multiple integrals*, volume 431. Prentice-Hall Englewood Cliffs, NJ, 1971.

[23] R. Cools and P. Rabinowitz. Monomial cubature rules since Stroud: A compilation. *Journal of Computational and Applied Mathematics*, 48(3):309–326, 1993.

[24] A. Grundmann and H. M. Möller. Invariant integration formulas for the N-simplex by combinatorial methods. *SIAM Journal on Numerical Analysis*, pages 282–290, 1978.

[25] S. Wandzura and H. Xiao. Symmetric quadrature rules on a triangle. *Computers & Mathematics with Applications*, 45(12):1829–1840, 2003.

[26] P. Lesaint and P.A. Raviart. On a finite element method for solving the neutron transport equation. In *Mathematical Aspects of Finite Elements in Partial Differential Equations*, pages 89–145. Academic Press, New York, 1974.

[27] C. Johnson and J. Pitkäranta. An analysis of the discontinuous Galerkin method for a scalar hyperbolic equation. *Mathematics of computation*, 46(173):1–26, 1986.

[28] G.R. Richter. An optimal-order error estimate for the discontinuous Galerkin method. *Mathematics of Computation*, 50(181):75–88, 1988.

[29] Q. Zhang and C.W. Shu. Error estimates to smooth solutions of Runge–Kutta discontinuous Galerkin methods for scalar conservation laws. *SIAM Journal on Numerical Analysis*, 42(2):641–666, 2004.

[30] J. Blazek. *Computational Fluid Dynamics: Principles and Applications*. Elsevier, 2001.

[31] P. Wesseling. *Principles of computational fluid dynamics*, volume 29. Springer, 2009.

[32] E. F. Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer, 2009.

[33] J. Qiu, B. C. Khoo, and C. W. Shu. A numerical study for the performance of the Runge–Kutta discontinuous Galerkin method based on different numerical fluxes. *Journal of Computational Physics*, 212(2):540–565, 2006.

[34] J. J. W. van der Vegt and H. van der Ven. Space–time discontinuous Galerkin finite element method with dynamic grid motion for inviscid compressible flows: I. General formulation. *Journal of Computational Physics*, 182(2):546–585, 2002.

[35] R. Biswas, K. D. Devine, and J. E. Flaherty. Parallel, adaptive finite element methods for conservation laws. *Applied Numerical Mathematics*, 14:255 – 283, 1994.

[36] J. C. Butcher. *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. Wiley-Interscience, 1987.

[37] M. H. Carpenter and C. A. Kennedy. Fourth-order 2n-storage Runge-Kutta schemes. *NASA Report TM*, 109112, 1994.

[38] S. Tu and S. Aliabadi. A slope limiting procedure in discontinuous Galerkin finite element method for gasdynamics applications. *International Journal of Numerical Analysis and Modeling*, 2:163–178, 2005.

[39] L. Krivodonova. Limiters for high-order discontinuous Galerkin methods. *Journal of Computational Physics*, 226(1):879–896, 2007.

[40] D. N. Arnold, F. Brezzi, B. Cockburn, and D. Marini. Discontinuous Galerkin methods for elliptic problems. *Lecture Notes in Computational Science and Engineering*, 11:89–102, 2000.

[41] A. Klöckner, T. Warburton, and J. S. Hesthaven. Viscous shock capturing in a time-explicit discontinuous Galerkin method. *Mathematical Modelling of Natural Phenomena*, 6(03):57–83, 2011.

[42] G. E. Barter and D. L. Darmofal. Shock capturing with PDE-based artificial viscosity for DGFEM: Part I. formulation. *Journal of Computational Physics*, 229(5):1810–1827, 2010.

[43] F.Bassi and S. Rebay. A high-order discontinuous Galerkin finite element method solution of the 2d Euler equations. *Journal of Computational Physics*, 138:251–285, 1997.

[44] P. O. Persson and J. Peraire. Curved mesh generation and mesh refinement using Lagrangian solid mechanics. *Proceedings of the 47th AIAA Aerospace Sciences Meeting and Exhibit*, 2009.

[45] L. A. Piegl and W. Tiller. *The NURBS book*. Springer Verlag, 1997.

[46] J. Nocedal and S. J. Wright. *Numerical optimization*. Springer, 2006.

[47] D. P. Bertsekas. Projected Newton methods for optimization problems with simple constraints. *SIAM Journal on Control and Optimization*, 20(2):221–246, 1982.

[48] C. Geuzaine and J. F. Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[49] Y. Renard and J. Pommier. Getfem finite element library. http://download.gna.org/getfem/html/homepage/, June 2011.

[50] R. Hartmann, J. Held, T. Leicht, and F. Prill. Discontinuous Galerkin methods for computational aerodynamics – 3D adaptive flow simulation with the DLR PADGE code. *Aerospace Science and Technology*, 14(7):512–519, October 2010.

[51] D. B. Kirk and W. W. Hwu. *Programming massively parallel processors: A hands-on approach*. Morgan Kaufmann, 2010.

[52] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. *NVIDIA whitepaper*, 2009.

[53] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, October 2012.

[54] M. Siebenborn, V. Schulz, and S. Schmidt. A curved-element unstructured discontinuous Galerkin method on GPUs for the Euler equations. *Computing and Visualization in Science*, 2013 (in print).

[55] M. Liebmann, C. Douglas, G. Haase, and Z. Horvath. Large scale simulations of the Euler equations on GPU clusters. In *Proceedings of DCABES 2010, Hongkong*, pages 50–54. IEEE Computer Society, 2010.

[56] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[57] E. N. Jacobs, K. E. Ward, and R. M. Pinkerton. The characteristics of 78 related airfoil sections from tests in the variable-density wind tunnel. 1933.

[58] V. Schmitt and F. Charpin. Pressure distributions on the ONERA-M6-wing at transonic mach numbers. *Report of the Fluid Dynamics Panel Working Group 04, AGARD AR 138*, 1979.

[59] NVIDIA. CUDA 5.0 Performance Report. http://developer.nvidia.com/, January 2013.

[60] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang. *PETSc Users Manual Revision 3.4*. Argonne National Laboratory, 2013.

[61] R.D. Falgout and U.M. Yang. HYPRE: A library of high performance preconditioners. In *Computational Science—ICCS 2002*, pages 632–641. Springer, 2002.

[62] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal on Scientific Computing*, 22(6):2194–2215, 2001.

[63] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.

[64] R.S. Dembo, S.C. Eisenstat, and T. Steihaug. Inexact Newton methods. *SIAM Journal on Numerical analysis*, 19(2):400–408, 1982.

[65] W. A. Mulder and B. van Leer. Experiments with implicit upwind methods for the Euler equations. *Journal of Computational Physics*, 59(2):232–246, 1985.

[66] M. Siebenborn and V. Schulz. GPU accelerated discontinuous Galerkin for Euler equation and its adjoint. In *Simulation Series*, volume 45, pages 15–21, 2013.

[67] M. B. Giles and N. A. Pierce. An introduction to the adjoint approach to design. *Flow, turbulence and combustion*, 65(3-4):393–415, 2000.

[68] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Society for Industrial and Applied Mathematics, 2008.

[69] A. Griewank. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6:83–107, 1989.

[70] M. B. Giles. On the use of Runge-Kutta time-marching and multigrid for the solution of steady adjoint equation. *Oxford University Computing Laboratory*, 2000.

[71] J.-P. Zolesio and M. C. Delfour. Shapes and geometries. *Analysis, differential calculus and optimization, SIAM, Philadelphia*, 2001.

[72] S. Schmidt. *Efficient Large Scale Aerodynamic Design Based on Shape Calculus*. PhD thesis, University of Trier, Germany, 2010.

[73] A. Borzì and V. Schulz. *Computational optimization of systems governed by partial differential equations*, volume 8. SIAM, 2012.

[74] B. Mohammadi and O. Pironneau. *Applied shape optimization for fluids*, volume 28. Oxford University Press Oxford, 2001.

[75] O. Pironneau. On optimal shapes for Stokes flow. *Journal of Fluid Mechanics*, 70(2):331–340, 1973.

[76] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer methods in applied mechanics and engineering*, 184(2):501–520, 2000.

[77] S. Ta'asan, G. Kuruvila, and M. D. Salas. Aerodynamic design and optimization in one shot. *30th Aerospace Sciences Meeting, Reno, NV, AIAA paper 92-0025*, 1992.