Seventh International Conference on Computational Fluid Dynamics (ICCFD7), Big Island, Hawaii, July 9-13, 2012

ICCFD7-2603

Efficient Implementation of the CPR Formulation for the Navier-Stokes Equations on GPUs

Malte Hoffmann*, Claus-Dieter Munz* and Z. J. Wang**
Corresponding author: hoffmann@iag.uni-stuttgart.de

* Institute of Aerodynamics and Gas Dynamics, University of Stuttgart
Pfaffenwaldring 21, 70569, Germany
** Department of Aerospace Engineering and CFD Center, Iowa State University
50011 Ames, USA

Abstract: The correction procedure via reconstruction (CPR) formulation for the Euler and Navier-Stokes equations is implemented on a NVIDIA graphics processing unit (GPU) using CUDA C with both explicit and implicit time-stepping schemes for 2D unstructured triangular grids. For the implicit time integration, a first order time approximation with Newton iteration and Gauß elimination is used to solve the system of equations, while for explicit time-stepping a 3-stage Runge-Kutta scheme is used. For the implicit time-stepping on the GPU a preconditioned mesh coloring algorithm is developed, which is derived from the *Four Color Theorem* known from the graph theory.
For the speed-up, compared to a single core of an Intel Xeon CPU, a factor up to 112-130 for explicit time-stepping is achieved, varying on the polynomial degree k and the chosen numerical flow. For the implicit time-stepping the maximum speed-up is between 47 and 89. All calculations are made in double precision using a single NVIDIA Tesla C2050.

Keywords: GPU Computing, Euler and Navier-Stokes equations, 2D unstructured triangular grids, implicit time integration, double precision, correction procedure via reconstruction, high-order.

1 Introduction

Recently a unifying discontinuous formulation named the correction procedure via reconstruction (CPR) was proposed for 2D and 3D unstructured grids[1]. This high-order method uses an element-local operator and one with only element-to-element coupling. These make this high-order methods ideal suited to the architecture of a graphic processing unit (GPU). GPUs, over the last three decades, where mostly used for graphic acceleration to calculate images shown by a computer screen. The calculation of the pixel color is done by the many-core GPU in parallel. But during the last years the GPUs became capable of calculating general purpose problems, like CFD calculations. A GPU is much cheaper then a CPU cluster with the same performance of peak operations per second. This makes GPU computing interesting especially for small companies and research groups with a small financial budget.
In this study the two topics, the high-order CPR formulation and GPU computing are combined. The CPR formulation for Euler and Navier-Stokes equations is implemented on the GPU, with both an explicit and an implicit time integration scheme. For the implicit time integration a cell coloring, derived from the four color theorem, is used to split the mesh. The aim of this study is to show that the CPR formulation is suitable for GPU computing and can be implemented in a way easy to understand. The implementation for the GPU should reduce the computation time dramatically compared to a not parallelized CPU implementation.

This paper is ordered as follows: At first the CPR formulation for the Euler and Navier-Stokes equation on arbitrary high-order triangular elements is explained. After explaining the CPR formulation the imple-

mentation for this method is described including the four color theorem and the cell coloring. Then the numerical results and the discussion of these results are shown. A conclusion is given in the last part of this paper.

2 High-Order CPR Method

The high-order CFD method, which is used in this study, is the so called Correction Procedure via Reconstruction (CPR). The CPR method was developed to improve the efficiency or stability of several well-known high-order methods, including staggered grid multi-domain (SG), spectral volume (SV), spectral differences (SD) and nodal discontinuous Galerkin (DG) methods. As a matter of fact, it unifies all these methods into a simple nodal or collocation-type formulation. In the CPR method, the degrees-of-freedom (DOFs) are the state variables at a pre-defined nodal set named solution points (SPs), where the differential form of the governing equation is solved [1]. In the following we shortly review Ref. [1].

2.1 CPR for a Linear Triangle

In the first part, CPR for the Euler equations (inviscid flux) is described. In the second part, the discretization for the Navier-Stokes equations is presented (viscous flux).

2.1.1 Discretization of the Inviscid-Equation

The hyperbolic conservation law is given by

$$\frac{\partial Q}{\partial t} + \vec{\nabla} \cdot \vec{F}(Q) = 0, \tag{1}$$

with proper boundary- and initial-conditions. Q is the state vector and $\vec{F}(Q) = (F(Q), G(Q))$ is the flux vector. In two dimensions the solution vector Q is

$$Q = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ e \end{bmatrix}, \tag{2}$$

where ρ is the density of the fluid, u the velocity of the fluid in X-Direction, v the velocity of the fluid in Y-Direction and e the total energy per unit volume. So the number of variables is $N_v = 4$. The inviscid-flux vector $\vec{F}(Q)$ is

$$\vec{F}(Q) = (F(Q), G(Q)) = \left(\begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(e+p) \end{bmatrix}, \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(e+p) \end{bmatrix} \right), \tag{3}$$

with $p = (\gamma - 1)\left(e - \frac{1}{2}\rho\left(u^2 + v^2\right)\right)$ the pressure of the fluid (γ is the ratio of specific heats for ideal gas). If one splits the computational area Ω into N_C non-overlapping triangular cells $\{V_i\}_{i=1}^{N_\mathrm{C}}$ and let W be any weighting function, then the weighted residual formulation of Eq. (1) on an cell V_i can be written as

$$\int_{V_i} \left(\frac{\partial Q}{\partial t} + \vec{\nabla} \cdot \vec{F}(Q) \right) W \,\mathrm{d}V = \int_{V_i} \frac{\partial Q}{\partial t} W \,\mathrm{d}V + \int_{\partial V_i} W \vec{F}(Q) \cdot \vec{n} \,\mathrm{d}S - \int_{V_i} \vec{\nabla} W \cdot \vec{F}(Q) \,\mathrm{d}V = 0. \tag{4}$$

In order to approximate the exact solution Q on V_i, Q_i is introduced. The approximate solution Q_i belongs on each cell V_i to the space of polynomials of degree k or less: $Q_i \in P^k(V_i)$. These polynomials may be discontinuous across cell interfaces. The dimension of P^k is $K = (k+1)(k+2)/2$. For now the numerical solution Q_i must satisfy Eq. (4)

$$\int_{V_i} \frac{\partial Q_i}{\partial t} W \,\mathrm{d}V + \int_{\partial V_i} W \vec{F}(Q_i) \cdot \vec{n} \,\mathrm{d}S - \int_{V_i} \vec{\nabla} W \cdot \vec{F}(Q_i) \,\mathrm{d}V = 0. \tag{5}$$

In this equation the surface integral $\int_{\partial V_i} W\vec{F}(Q_i)\cdot\vec{n}$ has to be approximated in a proper way because the numerical solution is discontinuous over cell interfaces. For this the normal flux term $\vec{F}(Q_i)$ in Eq. (5) is replaced by a common Riemann flux,

$$F^n(Q_i) \equiv \vec{F}(Q_i)\cdot\vec{n} \approx F^n_{com}(Q_i, Q_{i+}, \vec{n}),$$

in which Q_{i+} stands for the solution in the adjacent cell of V_i. Now instead of Eq. (5), the approximate solution must satisfy

$$\int_{V_i} \frac{\partial Q_i}{\partial t} W \,\mathrm{d}V + \int_{\partial V_i} W F^n_{com}\,\mathrm{d}S - \int_{V_i} \vec{\nabla}W\cdot\vec{F}(Q_i)\,\mathrm{d}V = 0.$$

The last term of the above left hand side can be integrated by parts again

$$\int_{V_i} \vec{\nabla}W\cdot\vec{F}(Q_i)\,\mathrm{d}V = \int_{V_i} W\vec{\nabla}\cdot\vec{F}(Q_i)\,\mathrm{d}V - \int_{\partial V_i} W\,F^n(Q_i)\,\mathrm{d}S,$$

which leads to

$$\int_{V_i} \frac{\partial Q_i}{\partial t} W \,\mathrm{d}V + \int_{V_i} W\vec{\nabla}\cdot\vec{F}(Q_i)\,\mathrm{d}V + \int_{\partial V_i} W\left[F^n_{com} - F^n(Q_i)\right]\mathrm{d}S = 0. \tag{6}$$

The term $\vec{\nabla}\cdot\vec{F}(Q_i)$ contains no information about the data in the neighbor cells. The boundary integral $\int_{\partial V_i} W\left[F^n_{com} - F^n(Q_i)\right]\mathrm{d}S$ represents the influence of these data. To cast the above boundary integral into a volume integral a "correction field" on $V_i, \delta_i \in P^k(V_i)$ is introduced,

$$\int_{\partial V_i} W\left[F^n_{com} - F^n(Q_i)\right]\mathrm{d}S = \int_{V_i} W\,\delta_i\,\mathrm{d}V. \tag{7}$$

The above equation is sometimes called "lifting operator" [2], which gets the normal flux differences on the boundary as an input and produces a member of $P^k(V_i)$ as an output. Substituting Eq. (7) in Eq. (6) leads to

$$\int_{V_i} \left(\frac{\partial Q_i}{\partial t} + \vec{\nabla}\cdot\vec{F}(Q_i) + \delta_i\right) W\,\mathrm{d}V = 0. \tag{8}$$

Because of (3) the conservation law is nonlinear, $\vec{\nabla}\cdot\vec{F}(Q_i)$ is usually not an element of $P^k(V_i)$. To make it a member of $P^k(V_i)$ the most obviously choice is to project $\vec{\nabla}\cdot\vec{F}(Q_i)$ onto $P^k(V_i)$. By introducing $\Pi\left(\vec{\nabla}\cdot\vec{F}(Q_i)\right)$ as the projection of $\vec{\nabla}\cdot\vec{F}(Q_i)$ onto $P^k(V_i)$ one choice is

$$\int_{V_i} \left(\frac{\partial Q_i}{\partial t} + \Pi\left(\vec{\nabla}\cdot\vec{F}(Q_i)\right) + \delta_i\right) W\,\mathrm{d}V = 0. \tag{9}$$

Eq. (9) can easily be reduced to

$$\frac{\partial Q_i}{\partial t} + \Pi\left(\vec{\nabla}\cdot\vec{F}(Q_i)\right) + \delta_i = 0. \tag{10}$$

Introducing the correction field δ_i and a projection of $\vec{\nabla}\cdot\vec{F}(Q_i)$ for nonlinear conservation laws, the weighted residual formulation can be reduced to a differential formulation.
By introducing the degrees of freedom (DOFs), which are the solutions at a set of solution points $\{\vec{p}_{i,j}\}$ (j runs from 1 to K), Eq. (10) can be written as

$$\frac{\partial Q_{i,j}}{\partial t} + \Pi_j\left(\vec{\nabla}\cdot\vec{F}(Q_i)\right) + \delta_{i,j} = 0. \tag{11}$$

Here $\Pi_j\left(\vec{\nabla}\cdot\vec{F}(Q_i)\right)$ are the values of $\Pi\left(\vec{\nabla}\cdot\vec{F}(Q_i)\right)$ at the SP j. For the computation of the correction field $\delta_{i,j}$, $k+1$ flux points (FPs) are defined on each cell face. At these FPs the normal flux differences

$[F^n_{com} - F^n(Q_i)]$ are computed. For efficiency the solution points are chosen to coincide with the flux points along cell faces to avoid any polynomial evaluation. In Fig. 2 the positions of the solution points and flux points for $k = 1$ to $k = 5$ for the standard triangle are shown (FPs are only located on the cell edges). The normal flux difference $[F^n_{com} - F^n(Q_i)]$ is approximated with a degree k interpolation polynomial along each face,

$$[F^n_{com} - F^n(Q_i)]_f \approx \mathbf{I}_k [F^n_{com} - F^n(Q_i)]_f \equiv \sum_l [F^n_{com} - F^n(Q_i)]_{f,l} L^{FP}_l,$$

where f is a face index and l is the FP index. L^{FP}_l is the value of the Lagrange interpolation polynomial based on the FP in a local face coordinate. For linear triangles with straight edges, once the solution points and flux points are chosen, the correction at the SPs can be written as

$$\delta_{i,j} = \frac{1}{|V_i|} \sum_{f \in \partial V_i} \sum_l \alpha_{j,f,l} [F^n_{com} - F^n(Q_i)]_{f,l} S_f, \tag{12}$$

where $\alpha_{j,f,l}$ are lifting constants independent of the solution, S_f is the face length and $|V_i|$ is the cell area. $\delta_{i,j}$ for one SP j is a linear combination of all the normal flux differences on all the faces of the cell.
To compute $\Pi_j \left(\vec{\nabla} \cdot \vec{F}(Q_i)\right)$ efficiently the chain rule approach (CR) is chosen,

$$\begin{aligned} \vec{\nabla} \cdot \vec{F}(Q_{i,j}) &= \frac{\partial F(Q_{i,j})}{\partial x} + \frac{\partial G(Q_{i,j})}{\partial y} \\ &= \frac{\partial F(Q_{i,j})}{\partial Q} \frac{\partial Q_{i,j}}{\partial x} + \frac{\partial G(Q_{i,j})}{\partial Q} \frac{\partial Q_{i,j}}{\partial y} \\ &= \frac{\partial \vec{F}(Q_{i,j})}{\partial Q} \cdot \vec{\nabla} Q_{i,j}, \end{aligned} \tag{13}$$

where $\frac{\partial \vec{F}(Q_{i,j})}{\partial Q}$ can be computed analytically.
The derivative $\nabla Q_{i,j}$ can be calculated with the Lagrange interpolation polynomial

$$\vec{\nabla} Q_{i,j} = \sum_j Q_{i,j} \vec{\nabla} L^{SP}_j.$$

The projection is approximated by the values of the Lagrange interpolation polynomial and the flux vector divergence at the solution points, i.e.

$$\Pi_j \left(\vec{\nabla} \cdot \vec{F}(Q_i)\right) \approx \sum_j L^{SP}_j \vec{\nabla} \cdot \vec{F}(Q_{i,j}).$$

We note that the CR approach is not exactly conservative [3].
Substituting Eq. (12) in Eq. (11) the following CPR formulation is obtained

$$\frac{\partial Q_{i,j}}{\partial t} + \Pi_j \left(\vec{\nabla} \cdot \vec{F}(Q_i)\right) + \frac{1}{|V_i|} \sum_{f \in \partial V_i} \sum_l \alpha_{j,f,l} [F^n_{com} - F^n(Q_i)]_{f,l} S_f = 0.$$

2.1.2 Formulation for Viscous-Flux

The Navier-Stokes equations can be written as

$$\frac{\partial Q}{\partial t} + \vec{\nabla} \cdot \vec{F}(Q) - \vec{\nabla} \cdot \vec{F}^\nu(Q, \vec{\nabla} Q) = 0, \tag{14}$$

where $\vec{F}^{\nu}(Q,\vec{\nabla}Q)$ is the viscous flux vector

$$\vec{F}^{\nu}(Q,\vec{\nabla}Q)=\Big(F^{\nu}(Q,\vec{\nabla}Q),G^{\nu}(Q,\vec{\nabla}Q)\Big)=\left(\begin{bmatrix}0\\ \tau_{xx}\\ \tau_{xy}\\ u\tau_{xx}+v\tau_{xy}-q_x\end{bmatrix},\begin{bmatrix}0\\ \tau_{yx}\\ \tau_{yy}\\ u\tau_{yx}+v\tau_{yy}-q_y\end{bmatrix}\right). \tag{15}$$

The viscous stress tensor can be represented as

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

where μ is the molecular viscosity coefficient, I is the identity matrix and $\vec{u}=(u,v)$ is the ve;ocity vector. The derivatives of the viscous stress tensor τ are given by (see [4], page 599)

$$\begin{aligned}\tau_{xx}&=\frac{2}{3}\mu\left(2\frac{\partial u}{\partial x}-\frac{\partial v}{\partial y}\right),\\ \tau_{yy}&=\frac{2}{3}\mu\left(2\frac{\partial v}{\partial y}-\frac{\partial u}{\partial x}\right),\\ \tau_{xy}=\tau_{yx}&=\mu\left(\frac{\partial u}{\partial y}-\frac{\partial v}{\partial x}\right).\end{aligned}$$

The heat flux is given as

$$\vec{q}=-c_p\frac{\mu}{Pr}\vec{\nabla}T.$$

Here, c_p is the specific heat capacity at constant pressure, T is the temperature and Pr is the Prandtl number and the components for q_i are (see [4] page 599)

$$q_x=-c_p\frac{\mu}{Pr}\frac{\partial T}{\partial x}\quad\text{and}\quad q_y=-c_p\frac{\mu}{Pr}\frac{\partial T}{\partial y}.$$

The variable $\vec{R}$ is introduced and defined by

$$\vec{R}=\vec{\nabla}Q. \tag{16}$$

Let $\vec{R}_i$ be an approximation of $\vec{R}$ on a linear triangular cell V_i, and $\vec{R}_i\in\left(P^k,P^k\right)$. The obvious choice of $\vec{R}_i=\vec{\nabla}Q_i$ is not appropriate due to the jump at the interface. Instead, the computation of $\vec{R}_i$ needs to involve data from neighboring cells. The CPR formulations of Eq. (14) and Eq. (16) on a linear triangle V_i can be written as

$$\begin{aligned}&\frac{\partial Q_{i,j}}{\partial t}+\Pi_j\left(\nabla\cdot\vec{F}\left(Q_i\right)\right)-\Pi_j^{\nu}\left(\nabla\vec{F}^{\nu}(Q_i,\vec{R}_i)\right)\\ &\qquad+\frac{1}{|V_i|}\sum_{f\in\partial V_i}\sum_{l}\alpha_{j,f,l}\left(\left[F_{com}^{n}-F^{n}\left(Q_i\right)\right]_{f,l}-\left[F_{com}^{\nu,n}-F^{\nu,n}(Q_i,\vec{R}_i)\right]_{f,l}\right)S_f=0,\\ &\vec{R}_{i,j}=(\vec{\nabla}Q_i)_j+\frac{1}{|V_i|}\sum_{f\in\partial V_i}\sum_{l}\alpha_{j,f,l}\left[Q^{com}-Q_i\right]_{f,l}\vec{n}_f S_f,\end{aligned}$$

where Π^{ν} is the projection operator for the divergence of the viscous flux vector to P^k, and

$$\left[F_{com}^{\nu,n}-F^{\nu,n}(Q_i,\vec{R}_i)\right]_{f,l}\equiv\vec{F}^{\nu}(Q_{f,l}^{com},\vec{\nabla}Q_{f,l}^{com})\cdot\vec{n}_{f,l}-\vec{F}^{\nu}(Q_i,\vec{R}_i)\Big|_{f,l}\cdot\vec{n}_{f,l},$$

with Q_f^{com} and $\vec{\nabla}Q_f^{com}$ the common solution and gradient on face f respectively, and $Q_{i,f,l}$ is the solution within cell i on FP l of face f. The computation of $\Pi^{\nu}\left(\nabla\cdot\vec{F}^{\nu}(Q_i,\vec{R}_i)\right)$ follows the Lagrange polynomial

approach. First, the viscous flux vector at each solution point is evaluated using

$$\vec{F}^{\nu}_{i,j} = \vec{F}^{\nu}\left(Q_{i,j}, \vec{R}_{i,j}\right).$$

After that, a Lagrange polynomial for the viscous flux vector is built with the values at all the solution points, i.e.

$$\mathbf{I}_k\left(\vec{F}^{\nu}_i\right) = \sum_j \vec{F}^{\nu}_{i,j} L^{SP}_j.$$

Finally the divergence of this polynomial is used as the projection

$$\Pi^{\nu}_j\left(\nabla \cdot \vec{F}^{\nu}\left(Q_i, \vec{R}_i\right)\right) \approx \vec{\nabla} \cdot \mathbf{I}_k\left(\vec{F}^{\nu}_{i,j}\right) = \sum_j \vec{F}^{\nu}_{i,j} \cdot \vec{\nabla} L^{SP}_j. \tag{17}$$

Various schemes for viscous fluxes differ in how the common solution Q^{com}_f and the common gradient $\vec{\nabla} Q^{com}_f$ are defined [5, 6, 7, 8]. Here the Bassi-Rebay 2 method [5] (BR2) is described.
The common solution in BR2 is simply the average of the solutions at both sides of a FP

$$Q^{com}_{f,l} = \frac{Q_i\Big|_{f,l} + Q_{i+}\Big|_{f,l}}{2}. \tag{18}$$

The common gradient is computed with

$$\vec{\nabla} Q^{com}_{f,l} = \frac{1}{2}\left(\vec{\nabla} Q_i\Big|_{f,l} + \vec{r}_i\Big|_{f,l} + \vec{\nabla} Q_{i+}\Big|_{f,l} + \vec{r}_{i+}\Big|_{f,l}\right), \tag{19}$$

where $\vec{\nabla} Q_i\Big|_{f,l}$ and $\vec{\nabla} Q_{i+}\Big|_{f,l}$ are the gradients of the solution at the left and right cells without corrections, while $\vec{r}_i\Big|_{f,l}$ and $\vec{r}_{i+}\Big|_{f,l}$ are the corrections to the gradients due to the difference between the common solution and the solution at each side of the face f at flux point l. More specifically,

$$\vec{r}_i\Big|_{f,l} = \frac{1}{|V_i|}\sum_{m=1}^{N_{FP}} \beta_{l,m}\left[Q^{com}\Big|_{f,m} - Q_i\Big|_{f,m}\right]\vec{n}_{f,m} S_f,$$

$$\vec{r}_{i+}\Big|_{f,l} = \frac{1}{|V_{i+}|}\sum_{m=1}^{N_{FP}} \beta_{l,m}\left[Q^{com}\Big|_{f,m} - Q_{i+}\Big|_{f,m}\right](-\vec{n}_{f,m}) S_f,$$

where N_{FP} is the number of flux points on the face f which is $k+1$ in 2D, l is a flux point on face f and $\beta_{l,m}$ is the coefficient of correction due to face f. Because of the choice that the solution points and flux points are the same $\beta_{l,m} = \alpha_{j,f,m}$, where index j is the solution point corresponding to flux point l on face f. For triangular cells, all $\beta_{l,m}$ are identical for any face f with a fixed distribution of flux points.

2.2 Extension to High Order Cells

To calculate arbitrary triangular cells, including high-order cells all cells are transformed from the physical domain (x, y) into a standard cell in the computational domain (ξ, η). The standard triangle is

$$\mathbf{T} = \left\{\vec{\xi} = (\xi, \eta)\,|\xi, \eta \geq 0; \xi + \eta \leq 1\right\},$$

shown in Fig. 1.

The transformation can be written as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \sum_j^K M_j(\xi, \eta) \begin{bmatrix} x_j \\ y_j \end{bmatrix},$$

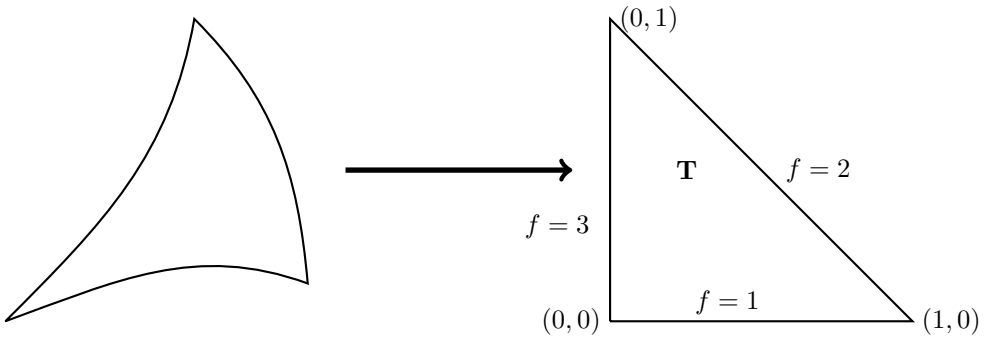

Figure 1: Transformation of general cells to the standard cell

where K equals the number of solution points defining the physical cell and $M_j(\xi, \eta)$ is the shape function which is based on a set of locations of nodes defining the shape of the standard cell. The Jacobian matrix J takes the following form

$$J = \frac{\partial(x, y)}{\partial(\xi, \eta)} = \begin{bmatrix} x_\xi & x_\eta \\ y_\xi & y_\eta \end{bmatrix}.$$

The metrics can be computed according to

$$\xi_x = \frac{y_\eta}{|J|}, \; \xi_y = -\frac{x_\eta}{|J|}, \; \eta_x = -\frac{y_\xi}{|J|}, \; \eta_y = \frac{x_\eta}{|J|},$$

and the inverse of the Jacobian is

$$J^{-1} = \begin{bmatrix} \xi_x & \xi_y \\ \eta_x & \eta_y \end{bmatrix}.$$

The transformed equation takes the following form for the inviscid flux

$$\frac{\partial \tilde{Q}}{\partial t} + \frac{\partial \tilde{F}}{\partial \xi} + \frac{\partial \tilde{G}}{\partial \eta} = 0, \tag{20}$$

and for the viscous flux

$$\frac{\partial \tilde{Q}}{\partial t} + \frac{\partial \tilde{F}}{\partial \xi} + \frac{\partial \tilde{G}}{\partial \eta} - \frac{\partial \tilde{F}^\nu}{\partial \xi} - \frac{\partial \tilde{G}^\nu}{\partial \eta} = 0, \tag{21}$$

where

$$\begin{aligned} \tilde{Q} &= |J| Q, \\ \tilde{F} &= |J| \left(\xi_x F + \xi_y G\right), \\ \tilde{G} &= |J| \left(\eta_x F + \eta_y G\right), \\ \tilde{F}^\nu &= |J| \left(\xi_x F^\nu + \xi_y G^\nu\right), \\ \tilde{G}^\nu &= |J| \left(\eta_x F^\nu + \eta_y G^\nu\right). \end{aligned} \tag{22}$$

The locations of the SPs and FPs on the standard triangle can be seen in Fig. 2 and are chosen to be the Gauss-Lobatto points.

2.2.1 Inviscid Flux

Let $\vec{S}_\xi = |J| \vec{\nabla} \xi$, $\vec{S}_\eta = |J| \vec{\nabla} \eta$, $\tilde{F} = \vec{F} \cdot \vec{S}_\xi$ and $\tilde{G} = \vec{F} \cdot \vec{S}_\eta$. Then Equation (20) can be written in the following form

$$\frac{\partial \tilde{Q}}{\partial t} + \vec{\nabla}^\xi \cdot \vec{\tilde{F}} = 0,$$

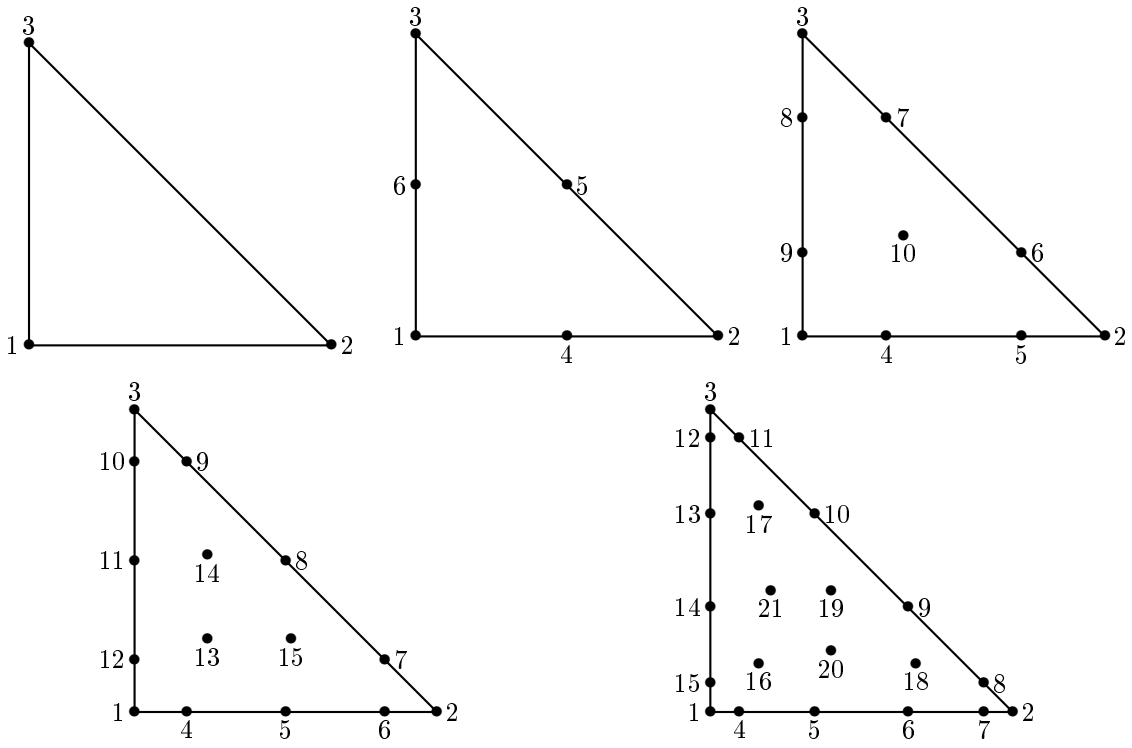

Figure 2: SPs ($\frac{(k+1)(k+2)}{2}$) and FPs $(k+1)$ for P^k ($k=1$ to $k=5$)

where $\vec{\tilde{F}} = \left(\tilde{F}, \tilde{G}\right)$ and $\vec{\nabla}^{\xi}$ is the the divergence operator in the computational domain. Since the standard cell (see Fig 1 on page 7) is a linear triangle, the CPR formulation can be directly applied

$$\frac{\partial \tilde{Q}_{i,j}}{\partial t} + \Pi_j \left(\vec{\nabla}^{\xi} \cdot \vec{\tilde{F}}(\tilde{Q}_i)\right) + \frac{1}{|V_i^{\xi}|} \sum_{f \in \partial V_i} \sum_{l} \alpha_{j,f,l} \left[\tilde{F}^n_{com} - \tilde{F}^n(\tilde{Q}_i)\right]_{f,l} S_f^{\xi} = 0, \tag{23}$$

where the superscript ξ means that the variables or operations are evaluated on the computational domain. By introducing a transformation from the physical to the computational domain we get

$$\begin{aligned}
\left[\vec{\tilde{F}}^n_{com} - \vec{\tilde{F}}^n(Q_i)\right]_{f,l} S_f^{\xi} &= \left(\left[\vec{\tilde{F}}_{com} - \vec{\tilde{F}}(\tilde{Q}_i)\right]_{f,l} \cdot \vec{n}_f^{\xi}\right) S_f^{\xi} \\
&= \left(\left[\vec{F}_{com} - \vec{F}(Q_i)\right]_{f,l} \cdot \vec{S}_{\xi}\Big|_{f,l} n_{\xi}\Big|_{f,l}\right) S_f^{\xi} + \left(\left[\vec{F}_{com} - \vec{F}(Q_i)\right]_{f,l} \cdot \vec{S}_{\eta}\Big|_{f,l} n_{\eta}\Big|_{f,l}\right) S_f^{\xi} \\
&= \left[\vec{F}_{com} - \vec{F}(Q_i)\right]_{f,l} \cdot \vec{S}^n_{f,l} \\
&= [F^n_{com} - F^n(Q_i)]_{f,l} |\vec{S}^n_{f,l}|,
\end{aligned} \tag{24}$$

where $\vec{n}^{\xi} = (n_{\xi}, n_{\eta})$ and $\vec{S}^n_{f,l} = \left[\vec{S}_{\xi} n_{\xi}\Big|_{f,l}, \vec{S}_{\eta} n_{\eta}\Big|_{f,l}\right]^T S_f^{\xi}$. The face length S_f^{ξ}, for the standard cell equals to $S_1^{\xi} = 1$, $S_2^{\xi} = \sqrt{2}$ and $S_3^{\xi} = 1$ as seen in Fig. 1 on page 7. The transformation $|\vec{S}^n_{f,l}|$ does not depend on the solution. Taking into account that for the standard triangle $|V_i^{\xi}| = 1/2$ and

$$\frac{1}{|J|} \vec{\nabla}^{\xi} \cdot \vec{\tilde{F}}(\tilde{Q}_i) = \vec{\nabla} \cdot \vec{F}(Q_i),$$

then with Eq. (24), the Eq. (23) can be further expressed as

$$\frac{\partial Q_{i,j}}{\partial t} + \Pi_j \left(\vec{\nabla} \cdot \vec{F}\left(Q_i\right) \right) + \frac{2}{|J|_{i,j}} \sum_{f \in \partial V_i} \sum_{l} \alpha_{j,f,l} \left[F^n_{com} - F^n\left(Q_i\right) \right]_{f,l} |\vec{S}^n_{f,l}| = 0. \tag{25}$$

$\alpha_{j,f,l}$ is the lifting constant in the computational domain.

2.2.2 Viscous Flux

Let

$$\begin{aligned}
\vec{S}_\xi &= |J| \vec{\nabla} \xi, \\
\vec{S}_\eta &= |J| \vec{\nabla} \eta, \\
\tilde{F} &= \vec{F} \cdot \vec{S}_\xi, \\
\tilde{G} &= \vec{F} \cdot \vec{S}_\eta, \\
\tilde{F}^\nu &= \vec{F}^\nu \cdot \vec{S}_\xi, \\
\tilde{G}^\nu &= \vec{F}^\nu \cdot \vec{S}_\eta,
\end{aligned}$$

then Equation (21) can be written in the following form

$$\frac{\partial \tilde{Q}}{\partial t} + \vec{\nabla}^\xi \cdot \vec{\tilde{F}} - \vec{\nabla}^\xi \cdot \vec{\tilde{F}}^\nu = 0,$$

where $\vec{\tilde{F}} = \left(\tilde{F}, \tilde{G} \right)$, $\vec{\tilde{F}}^\nu = \left(\tilde{F}^\nu, \tilde{G}^\nu \right)$ and $\vec{\nabla}^\xi$ is the divergence operator in the computational domain. Since the standard cell (see Fig. 1 on page 7) is a linear triangle, the CPR formulation can be directly applied

$$\begin{aligned}
&\frac{\partial \tilde{Q}_{i,j}}{\partial t} + \Pi_j \left(\vec{\nabla}^\xi \cdot \vec{\tilde{F}} \left(\tilde{Q}_i \right) \right) - \Pi^\nu_j \left(\vec{\nabla}^\xi \vec{\tilde{F}}^\nu (\tilde{Q}_i, \vec{\tilde{R}}_i) \right) \\
&\quad + \frac{1}{|V^\xi_i|} \sum_{f \in \partial V^\xi_i} \sum_{l} \alpha_{j,f,l} \left(\left[\tilde{F}^n_{com} - \tilde{F}^n \left(\tilde{Q}_i \right) \right]_{f,l} - \left[\tilde{F}^{\nu,n}_{com} - \tilde{F}^{\nu,n} \left(\tilde{Q}_i, \vec{\tilde{R}}_i \right) \right]_{f,l} \right) S^\xi_f = 0,
\end{aligned} \tag{26}$$

$$\vec{\tilde{R}}_{i,j} = (\vec{\nabla}^\xi \tilde{Q}_i)_j + \frac{1}{|V^\xi_i|} \sum_{f \in \partial V^\xi_i} \sum_{l} \alpha_{j,f,l} \left[\tilde{Q}^{com} - \tilde{Q}_i \right]_{f,l} \vec{n}^\xi_f S^\xi_f. \tag{27}$$

Transforming $[\tilde{F}^{\nu,n}_{com} - \tilde{F}^{\nu,n}(\tilde{Q}_i, \vec{\tilde{R}}_i)]_{f,l}$ into the physical domain using the same approach as in Eq. (24) and taking into account $|V^\xi_i| = 1/2$,

$$\frac{1}{|J|} \vec{\nabla}^\xi \cdot \vec{\tilde{F}}(\tilde{Q}_i) = \vec{\nabla} \cdot \vec{F}(Q_i) \quad \text{and} \quad \frac{1}{|J|} \vec{\nabla}^\xi \cdot \vec{\tilde{F}}^\nu(\tilde{Q}_i, \vec{\tilde{R}}_i) = \vec{\nabla} \cdot \vec{F}^\nu(Q_i, \vec{R}_i),$$

Equation (26) can be further expressed as

$$\begin{aligned}
&\frac{\partial Q_{i,j}}{\partial t} + \Pi_j \left(\nabla \cdot \vec{F}\left(Q_i\right) \right) - \Pi^\nu_j \left(\nabla \vec{F}^\nu (Q_i, \vec{R}_i) \right) \\
&\quad + \frac{2}{|J|_{i,j}} \sum_{f \in \partial V_i} \sum_{l} \alpha_{j,f,l} \left(\left[F^n_{com} - F^n\left(Q_i\right) \right]_{f,l} - \left[F^{\nu,n}_{com} - F^{\nu,n}(Q_i, \vec{R}_i) \right]_{f,l} \right) |\vec{S}^n_{f,l}| = 0,
\end{aligned} \tag{28}$$

with

$$\frac{1}{|J|_{i,j}} \left(\vec{\nabla}^\xi \tilde{Q}_{i,j} \cdot J^{-1}_{i,j} - Q_{i,j} \vec{\nabla} |J|_{i,j} \right) = \vec{\nabla} Q_{i,j}, \tag{29}$$

where

$$\nabla^{\xi}\bar{Q}_{i,j} = \begin{bmatrix} \frac{\partial \bar{Q}_{i,j}}{\partial \xi} \\ \frac{\partial \bar{Q}_{i,j}}{\partial \eta} \end{bmatrix} = \begin{bmatrix} \sum_m^K c_{j,m}^{\xi} Q_{i,m} |J|_{i,m} \\ \sum_m^K c_{j,m}^{\eta} Q_{i,m} |J|_{i,m} \end{bmatrix}.$$

The coefficients $c_{j,m}^{\xi}$ and $c_{j,m}^{\eta}$ can be calculated analytically. With Eq. (29) and the transformation in Eq. (24), the Equation (27) can be further expressed as

$$\vec{R}_{i,j} = \vec{\nabla} Q_{i,j} + \frac{2}{|J|_{i,j}} \sum_{f \in \partial V_i} \sum_l \alpha_{j,f,l} \left[Q^{com} - Q_i\right]_{f,l} \vec{n}_{f,l} |\vec{S}_{f,l}^n|.$$

According to the BR2 method $Q^{com} = (Q_i + Q_{i+})/2$, the previous equation becomes

$$\vec{R}_{i,j} = \vec{\nabla} Q_{i,j} + \frac{1}{|J|_{i,j}} \sum_{f \in \partial V_i} \sum_l \alpha_{j,f,l} \left[Q_{i+} - Q_i\right]_{f,l} \vec{n}_{f,l} |\vec{S}_{f,l}^n|. \tag{30}$$

Also the common gradient can be calculated in the computational domain and transformed to the physical domain. With $|V_i| = |V_{i+}| = 1/2$ Eq. (19) can be further expressed

$$\vec{\nabla} Q_i^{com}\Big|_{f,l} = \frac{1}{2}\left(\vec{\nabla} Q_i\Big|_{f,l} + \vec{\nabla} Q_{i+}\Big|_{f,l}\right) + \frac{1}{|J|_{f,l}} \sum_{m=1}^{N_{FP}} \beta_{l,m} \left[Q_{i+} - Q_i\right]_{f,m} \vec{n}_{f,m} |\vec{S}_{f,m}^n|. \tag{31}$$

3 Time-Stepping

In the last sections, the focus was on how the space discretization is calculated. To complete the fully discrete scheme the focus is on the time discretization in this section. The CPR formulation for the inviscid flux can also be written as (see Eq. (25))

$$\frac{\partial Q_{i,j}}{\partial t} = -\Pi_j \left(\vec{\nabla} \cdot \vec{F}(Q_i)\right) - \frac{2}{|J|_{i,j}} \sum_{f \in \partial V_i} \sum_l \alpha_{j,f,l} [F_{com}^n - F^n(Q_i)]_{f,l} |\vec{S}_{f,l}^n|, \tag{32}$$

and the viscous flux (see Eq. (28))

$$\begin{aligned} \frac{\partial Q_{i,j}}{\partial t} =& -\Pi_j \left(\vec{\nabla} \cdot \vec{F}\left(Q_i\right)\right) + \Pi_j^{\nu} \left(\vec{\nabla} \cdot \vec{F}^{\nu}(Q_i, \vec{R}_i)\right) \\ & - \frac{2}{|J|_{i,j}} \sum_{f \in \partial V_i} \sum_l \alpha_{j,f,l} \left(\left[F_{com}^n - F^n\left(Q_i\right)\right]_{f,l} - \left[F_{com}^{\nu,n} - F^{\nu,n}(Q_i, \vec{R}_i)\right]_{f,l} \right) |\vec{S}_{f,l}^n|. \end{aligned} \tag{33}$$

The vector $\frac{\partial Q}{\partial t}$ from all solution points at every cell is also known as the residual $\mathrm{Res}(Q)$. In the next subsections an explicit and an implicit way to integrate with respect to time are described.

3.1 Explicit

For the explicit time integration, a 3-stage Runge-Kutta scheme from Ref. [9] is used. To step from Q_i^t (the solution at the old time), to Q_i^{t+1} (the solution at the new time), the following equations must be solved

$$\begin{aligned} Q_i^{(1)} &= Q_i^t + \Delta t \, \mathrm{Res}_i(Q^t), \\ Q_i^{(2)} &= \frac{3}{4} Q_i^t + \frac{1}{4} Q_i^{(1)} + \frac{1}{4} \Delta t \, \mathrm{Res}_i(Q^{(1)}), \\ Q_i^{t+1} &= \frac{1}{3} Q_i^t + \frac{2}{3} Q_i^{(2)} + \frac{2}{3} \Delta t \, \mathrm{Res}_i(Q^{(2)}), \end{aligned}$$

where Δt is the chosen time step. The explicit time discretization has to satisfy a time step restriction and is appropriate for unsteady problems.

3.2 Implicit

In Ref. [10] is shown that for an implicit time integration the following equation

$$\left(\frac{I}{\Delta t} - \frac{\partial \mathrm{Res}_i}{\partial Q_i}\right)\left(Q_i^{(w+1)} - Q_i^w\right) = \mathrm{Res}_i(Q^*) - \frac{\Delta Q_i^*}{\Delta t}, \tag{34}$$

must be solved. The superscript w is an iteration index and the superscript $*$ indicates the most recently updated solution.
The matrix

$$D_i = \left(\frac{I}{\Delta t} - \frac{\partial \mathrm{Res}_i}{\partial Q_i}\right),$$

is the cell matrix. Since the residual for one solution point is calculated by including the solutions of all the other solution points, $\frac{\partial \mathrm{Res}_i}{\partial Q_i}$ is a $K * N_v \times K * N_v$ matrix and I is the $K * N_v \times K * N_v$ identity matrix.

$$\frac{\partial \mathrm{Res}_i}{\partial Q_i} = \begin{bmatrix} \frac{\partial \mathrm{Res}_{i,1,1}}{\partial Q_i} & \frac{\partial \mathrm{Res}_{i,1,2}}{\partial Q_i} & \frac{\partial \mathrm{Res}_{i,1,3}}{\partial Q_i} & \cdots & \frac{\partial \mathrm{Res}_{i,1,m}}{\partial Q_i} \\ \frac{\partial \mathrm{Res}_{i,2,1}}{\partial Q_i} & \frac{\partial \mathrm{Res}_{i,2,2}}{\partial Q_i} & \frac{\partial \mathrm{Res}_{i,2,3}}{\partial Q_i} & \cdots & \frac{\partial \mathrm{Res}_{i,2,m}}{\partial Q_i} \\ \frac{\partial \mathrm{Res}_{i,3,1}}{\partial Q_i} & \frac{\partial \mathrm{Res}_{i,3,2}}{\partial Q_i} & \frac{\partial \mathrm{Res}_{i,3,3}}{\partial Q_i} & \cdots & \frac{\partial \mathrm{Res}_{i,3,m}}{\partial Q_i} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathrm{Res}_{i,j,1}}{\partial Q_i} & \frac{\partial \mathrm{Res}_{i,j,2}}{\partial Q_i} & \frac{\partial \mathrm{Res}_{i,j,3}}{\partial Q_i} & \cdots & \frac{\partial \mathrm{Res}_{i,j,m}}{\partial Q_i} \end{bmatrix},$$

where $\frac{\partial \mathrm{Res}_{i,j,m}}{\partial Q_i}$ is calculated by using the following numerical approach based on the definition

$$\frac{\partial \mathrm{Res}_{i,j,m}}{\partial Q_i} \approx \frac{\mathrm{Res}_{i,j,m}(\{q_{i,m} + \epsilon\}, Q \setminus q_{i,m}) - \mathrm{Res}_{i,j}(Q)}{\epsilon},$$

where $\epsilon = 10^{-4}$ and m runs from 1 to K. In the above equation $q_{i,m} \in [\rho, \rho u, \rho v, e]_{i,m}$ are the solutions of the cell i and SP m, and is changed by the amount of ϵ, and all the other solutions are not changed. The residual $\mathrm{Res}_{i,j}(Q)$ is the residual without any changes made to the solution vector Q. This makes $\partial \mathrm{Res}_{i,j,m}/\partial Q_i$ a $N_v \times N_v$ matrix (4×4 in 2D).
For steady state problems the last term of Eq. (34) can often be dropped resulting in a faster convergence rate.

4 Implementation

It is important to note, that the pre- and post-processing for the solver itself, for example, reading the grid and calculating the values for the transformation from the physical domain into the computational domain, as well as setting up the initial data to the SPs and writing the output files is done by the CPU and was not coded by the author.
The implementation was done in a way that for polynomial degrees from P^1 to P^5 the code only has to be compiled once. Then the degree can be set in an input file. For this paper a NVIDIA C2050 was used to perform the calculations. The block size for a kernel call was set to be a multiply of 14 to fit the numbers of streaming multiprocessors on the C2050. In each block multiple cells were calculated. Since the polynomial degree can vary the maximum for the thread grid, $\vec{t}$, in X direction must be determined. So $t^x = \mathbf{max}(N_{SP}, N_{FP} * N_f)$. The complete thread grid is then $\vec{t} = [t^x, 64/t^x]$. Here a total thread number around 64 was chosen which gives a good calculation speed for all P^k. In other words the number of cells calculated per SM in parallel vary with the chosen degree k. The number of cells goes from ($k = 5$) 3 cells to ($k = 1$) 10 cells.
The following points were important in this study to get a good performance

- The needed amount of shared memory is kept as low as possible by reusing it
- The synchronization of threads is done at a good code position

- Use of textured memory
- Only one global memory write per thread
- Multiple cells have to be calculated on one SM

4.1 Explicit Time Stepping

For the explicit time stepping one kernel to calculate the residual is used. This kernel is solving Eq. (32) or Eq. (33). For the viscous flux a additional kernel is implemented to calculate the derivatives of Q. For time integration the 3-Stage-Runge-Kutta-scheme from subsection 3.1 is used, where a kernel solves one stage of the Runge-Kutta-scheme on the GPU in parallel.

4.2 Implicit Time-Stepping

In this section the way how to solve Eq. (34) (p. 11) with the GPU is described. The algorithm to solve this equation is different with the CPU and with the GPU. For the CPU a lower-upper symmetric Gauss-Seidel (LU-SGS) scheme is implemented which is described in Ref. [10] and will not be further reviewed here. For the GPU implementation Eq. (34) is modified and the modification is described in the following subsection.

4.2.1 Implicit Time-Stepping on a GPU

For the implementation on the GPU only steady state problems are calculated in this study. This reduces Eq. (34) to

$$\left(\frac{I}{\Delta t} - \frac{\partial \mathrm{Res}_i}{\partial Q_i}\right)\left(Q_i^{(w+1)} - Q_i^{(w)}\right) = \mathrm{Res}_i(Q^*). \tag{35}$$

Lets keep in mind that the cell matrix is

$$D_i = \left(\frac{I}{\Delta t} - \frac{\partial \mathrm{Res}_i}{\partial Q_i}\right). \tag{36}$$

Now Eq. (35) can be written as

$$Q_i^{(w+1)} = D_i^{-1}\mathrm{Res}_i(Q^*) + Q_i^{(w)}, \tag{37}$$

where, as mentioned before, w is an iteration index and the superscript $*$ stands for the most recently updated solution. If calculated in a serial way, i is the index for a cell and runs from 0 to $(N_C - 1)$, and the Res_i is a function containing the neighbor cells solutions. This means that for the first cell the old solutions are used to calculate the residual for the first cell. The solution for the first cell is then updated as shown in the equation (37). This updated solution is included into the calculation of the new solution for the second cell and this is then included to calculate the new solution for the third and so on, until all solutions for each cell are updated.
The advantage of using a GPU to calculate the solutions in parallel makes it impossible to calculate the solution of one cell and update the overall solution Q, to calculate then a solution of another cell with the updated overall solution. For the GPU implementation a different approach must be found.
Since the Res_i, for cell i, needs only the solutions from cell i and the solutions from the neighbor cells Q_{nb}, the $\mathrm{Res}_i(Q^*)$ can be reduced to $\mathrm{Res}_i(Q_i, Q_{nb}^*)$. By including this to Eq. (37), the equation becomes

$$Q_i^{(w+1)} = D_i^{-1}\mathrm{Res}_i(Q_i, Q_{nb}^*) + Q_i^{(w)}. \tag{38}$$

This means for the GPU implementation, it must be assured, that neighbor cells are not updated at the same time. For the GPU implementation it is also important to calculate as many cells at the same time as possible. This means that a way must be found to split the mesh into as few segments as possible, but ensure that neighbor cells are not in the same segment. Next a way is described how this can be done.

4.2.2 Mesh Coloring

The solution how to split the mesh into as few segments as possible, with no neighbor cells in the same segment, can be found in the Graph-Theory.

Four Color Theorem: *Every planar graph can be colored with maximal four colors, in a way that no connected vertices share the same color*[11, 12, 13].

A graph contains vertices which are connected by edges. A planar graph is a graph that can be drawn on the plane in such a way that no edges cross each other.[14] An example for a planar graph can be seen in Fig. 3 and one for a non planar graph in Fig. 4.

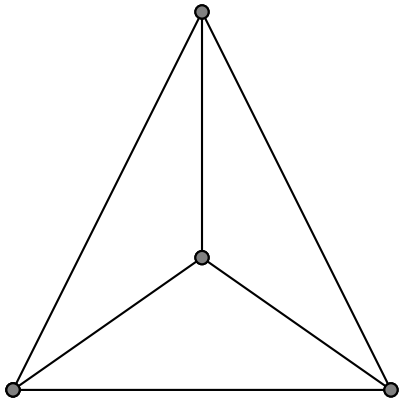

Figure 3: Planar graph

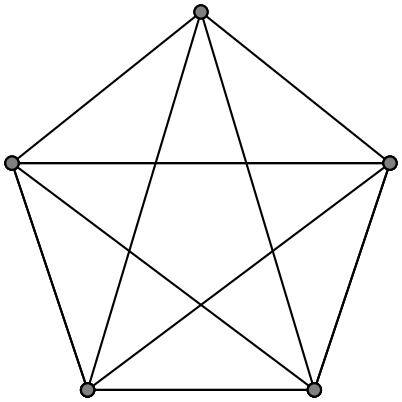

Figure 4: Non-Planar graph

Now every 2D mesh can be transformed to a planar graph. The cell center is representing a vertex, and the face, which connected cells share, can be seen as an edge (see Fig. 5).

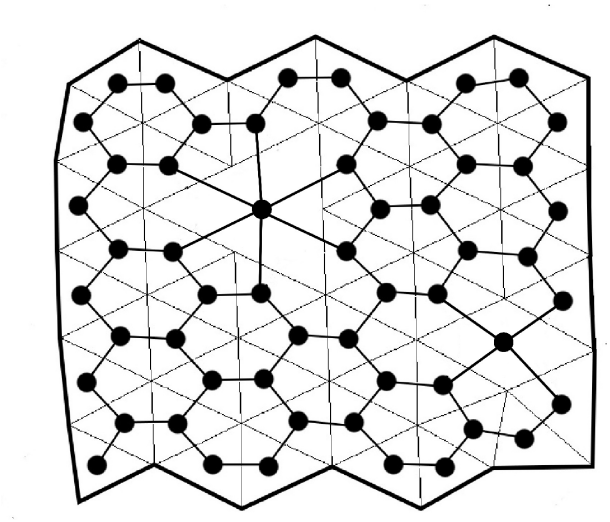

Figure 5: 2D mesh overlapped with its planar graph

This implies that every 2D mesh (structured, unstructured, any shape and hanging nodes) can also be colored with maximal four colors, with neighbor cells not having the same color. An example can be seen in Fig. 6. The Eq. (38) p. 12 can now be solved with the symmetric forward and backward sweeps. The sweeps can

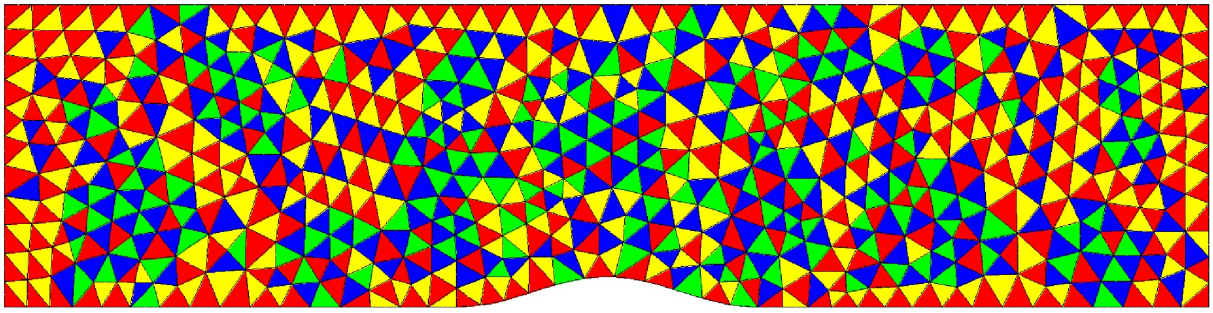

Figure 6: 2D triangular mesh colored with four colors (994 cells: 313 Red, 257 Yellow, 252 Blue and 172 Green)

be written in a row as

$$\text{Red} \rightarrow \text{Yellow} \rightarrow \text{Blue} \rightarrow \text{Green} \rightarrow \text{Green} \rightarrow \text{Blue} \rightarrow \text{Yellow} \rightarrow \text{Red}.$$

This must be done to make sure that all the solutions are calculated with the updated solutions from the neighbor cells.

4.2.3 Inverting the cell matrix D

Since the LU-SGS approach, which is used for a CPU version of the code, is not suitable for a GPU parallelization a simple Gauss-Jordan elimination is used instead. The method is modified to avoid that the D matrix and its inverse is stored. Normally to invert a matrix we have a system that looks like the following example

$$[DI] = \begin{bmatrix} 2 & -1 & 0 & 1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 1 & 0 \\ 0 & -1 & 2 & 0 & 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{3}{2} & -1 & \frac{1}{2} & 1 & 0 \\ 0 & -1 & 2 & 0 & 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & \frac{3}{4} & \frac{1}{2} & \frac{1}{4} \\ 0 & 1 & 0 & \frac{1}{2} & 1 & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{4} & \frac{1}{2} & \frac{3}{4} \end{bmatrix} = [ID^{-1}]. \tag{39}$$

As said before this needs double the amount of memory than the D matrix alone. On the GPU the shared memory is used to avoid double data storage. The algorithm ,which has D as an input and D^{-1} as an output at the same memory address, is shown in the following example

$$D = \left[\begin{array}{ccc|c} 2 & -1 & 0 & 1 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & 0 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 1 & -\frac{1}{2} & 0 & \frac{1}{2} \\ 0 & \frac{3}{2} & -1 & \frac{1}{2} \\ 0 & -1 & 2 & 0 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{3}{2} & -1 & 1 \\ 0 & -1 & 2 & 0 \end{array}\right] \Rightarrow \tag{40}$$

$$\left[\begin{array}{ccc|c} \frac{2}{3} & 0 & -\frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & 1 & -\frac{2}{3} & \frac{2}{3} \\ \frac{1}{3} & 0 & \frac{4}{3} & \frac{2}{3} \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} \frac{2}{3} & \frac{1}{3} & -\frac{1}{3} & 0 \\ \frac{1}{3} & \frac{2}{3} & -\frac{2}{3} & 0 \\ \frac{1}{3} & \frac{2}{3} & \frac{4}{3} & 1 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} \frac{3}{4} & \frac{1}{2} & 0 & \frac{1}{4} \\ \frac{1}{2} & 1 & 0 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & 1 & \frac{3}{4} \end{array}\right] \Rightarrow \left[\begin{array}{ccc} \frac{3}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{3}{4} \end{array}\right] = D^{-1}. \tag{41}$$

The one column from inverse of each matrix is stored in shared memory and the rest of the matrix is stored in global memory. The row and columns operation are done in parallel. The size of the element matrix depends on the degree of the polynomial for P^1 it is 12×12 and for P^5 it is 84×84. The matrix vector multiplication in Eq. (38) can be easy implemented for a GPU.

5 Numerical Results and Discussions

In this chapter the first section shows the results of the GPU code, compared to the CPU version. Results for the CPU version, which is coded at the Iowa State University, and the proof that it is working correctly can be seen in Ref. [1]. The second section shows performance tests comparing the Tesla C2050 and one

core from a Intel Xeon CPU (X5650@2.67GHz). The CPU version is compiled with the *gcc*-Compiler and the optimization flag "-O3".

5.1 Verification

This sections shows the results of the GPU and the CPU version for different test cases. Since the CPU version is proofed to be correct, the GPU version should be correct when its results are the same. The first subsection shows calculations with explicit time-stepping for both flows, and the second subsection shows results for implicit time-stepping.

5.1.1 Explicit Time-Stepping

For the explicit time-stepping a small test case is chosen. The mesh of the test case (sine bump) can be seen in Fig. 7 page 15. The length of the mesh in X-Direction is 4 and in the height in Y-Direction is 1. The highest point of the bump is located in the middle with a height of 0.1 and the bump has a length of 1. To

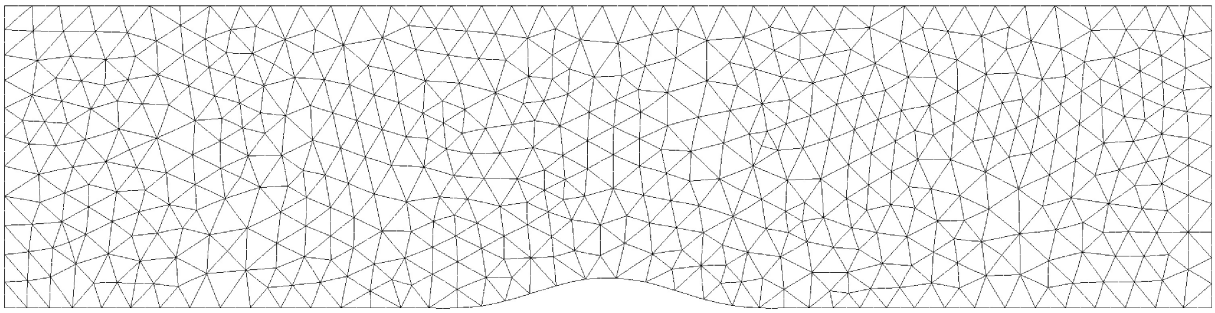

Figure 7: 2D triangular mesh with 994 cells

show that the GPU version is working correctly the residuals from both versions are compared.

Inviscid Flow Simulation

For the inviscid flow the Mach number in both coordinate directions are $u = 0.5$ Ma and $v = 0$. For ideal gas $\gamma = 1.4$. To proof that the GPU version works correctly it is not necessary that the calculation converges to machine zero, instead the calculation is aborted after the residual is smaller or equal to $1 * 10^{-3}$. The Rusanov flux is used for solving the Riemann problem and Δt is the chosen time step for the Runge-Kutta scheme (see Subsection 3.1). The results for P^1 to P^5 are shown in Tab. 1. This shows that the residual

Table 1: Sine bump test case on the CPU and GPU for inviscid flow with explicit time-stepping (ΔRes = |Residual CPU-Residual GPU|)

	Δt	Residual CPU	Residual GPU	ΔRes
P^1	$9 * 10^{-3}$	$9.8885479621984998 * 10^{-4}$	$9.8885479621338749 * 10^{-4}$	$6.46 * 10^{-15}$
P^2	$4.9 * 10^{-3}$	$1.0000527084353045 * 10^{-3}$	$1.0000527083855000 * 10^{-3}$	$4.98 * 10^{-14}$
P^3	$3.265 * 10^{-3}$	$9.7753546418898420 * 10^{-4}$	$9.7753546406587760 * 10^{-4}$	$1.23 * 10^{-13}$
P^4	$2.2 * 10^{-3}$	$9.9807573426047768 * 10^{-4}$	$9.9807573401956796 * 10^{-4}$	$2.41 * 10^{-13}$
P^5	$1.6 * 10^{-3}$	$1.0000958713101862 * 10^{-3}$	$1.0000958710186937 * 10^{-3}$	$2.91 * 10^{-13}$

for the CPU and GPU version are exactly the same except for a difference around machine zero.

Viscous Flow Simulation

For the viscous flow the same parameters are chosen: the velocity in X-Direction $u = 0.5$ Ma, $v = 0$ and $\gamma = 1.4$. The Reynolds number is $Re = 50$. For this test-case it is again not necessary that the calculation

converges to machine zero, instead in this case the calculation is aborted after 2000 iterations. The results for P^1 to P^5 are shown in Tab. 2. This shows that the residual of the CPU and GPU version are exactly

Table 2: Sine bump test case on the CPU and GPU for viscous flow with explicit time-stepping(ΔRes = |Residual CPU-Residual GPU|)

	Δt	Residual CPU	Residual GPU	ΔRes
P^1	$4 * 10^{-3}$	$5.1013428220563111 * 10^{-3}$	$5.1013428220572244 * 10^{-3}$	$9.0 * 10^{-16}$
P^2	$1 * 10^{-3}$	$3.4752654702877042 * 10^{-2}$	$3.4752654702867425 * 10^{-2}$	$1.0 * 10^{-14}$
P^3	$5 * 10^{-4}$	$6.5738716205786585 * 10^{-2}$	$6.5738716205772041 * 10^{-2}$	$1.4 * 10^{-14}$
P^4	$3 * 10^{-4}$	$8.4765734239744359 * 10^{-2}$	$8.4765734239742596 * 10^{-2}$	$2.0 * 10^{-15}$
P^5	$8 * 10^{-5}$	$8.6143727408424978 * 10^{-2}$	$8.6143727408467388 * 10^{-2}$	$4.3 * 10^{-14}$

the same except of a difference around machine zero.

For both flows the GPU version works correctly, compared to the results from the CPU. The small difference in values of the residual could be caused by different rounding on the CPU and GPU. If the Roe flux is used for solving the Riemann problem the result is the same.

5.1.2 Implicit Time-Stepping

Inviscid Flow Simulation

The inviscid flow calculation with implicit time-stepping uses the same mesh as seen in Fig. 7 (p. 15). The coloring of the mesh can be seen in Fig. 6 (p. 14). The test-case variables are the same as before: $u = 0.5$ Ma, $v = 0$ and for ideal gas $\gamma = 1.4$. Here Δt is the chosen time-step for the first iteration, the ratio r is the factor the time-step grows at each iteration and $\Delta t_{\max}$ is the maximum time step allowed for this calculation. For this test-case the residual must converge to a value smaller than 10^{-11}, to show that both versions would converge to machine zero. The results for P^1 to P^5 are shown in Tab. 3. Both calculations

Table 3: Sine bump test case on the CPU and GPU for inviscid flow with implicit time-stepping

	Δt	Ratio r	$\Delta t_{\max}$	Iterations CPU	Iterations GPU
P^1	$9 * 10^{-3}$	1.05	$1 * 10^{-1}$	1320	1760
P^2	$4.9 * 10^{-3}$	1.05	$1 * 10^{-1}$	640	740
P^3	$3.265 * 10^{-3}$	1.05	$1 * 10^{-1}$	1060	1330
P^4	$2.2 * 10^{-3}$	1.05	$1 * 10^{-1}$	770	890
P^5	$1.6 * 10^{-3}$	1.05	$1 * 10^{-1}$	1090	1320

compared visually show exactly the same result, even the values of the solution vectors are identical except of a difference around machine zero.

Viscous Flow Simulation

For the viscous flow simulation a different mesh is chosen. The test-case is the flow around a sphere (see Fig. 8) with $u = 0.5$ Ma, $v = 0$ and $\gamma = 1.4$. The Reynolds number is $Re = 50$. The radius of the sphere is 1 and the radius for the far field is 20. The coloring of the cells can be seen in Fig. 9. The residual again must converge to a value smaller than 10^{-11}. The results for P^1 to P^5 are shown in Tab. 4. Also in this case, both calculations compared by eye show exactly the same result, even the values of the solution vectors are identical except of a difference around machine zero.

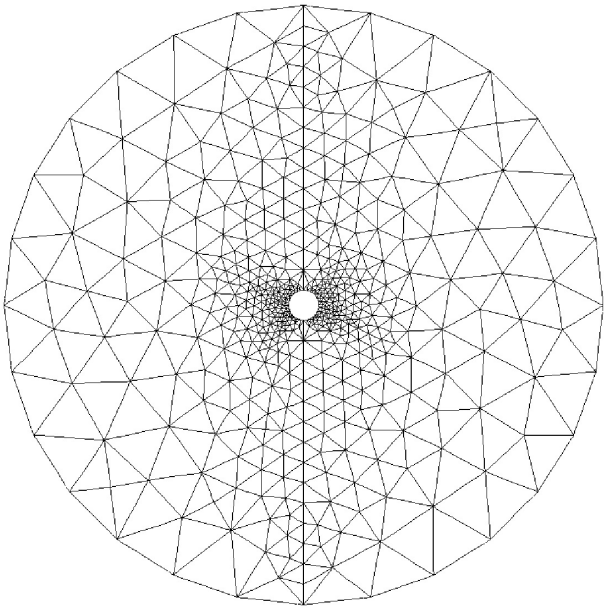

Figure 8: Mesh for the sphere test case with 992 cells

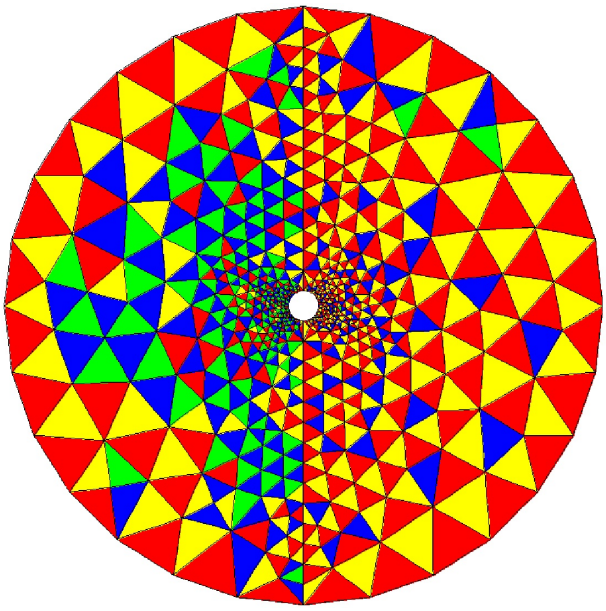

Figure 9: Coloring of the mesh

Table 4: Sphere test case on the CPU and GPU for viscous flow with implicit time-stepping

	Δt	Ratio r	$\Delta t_{\max}$	Iterations CPU	Iterations GPU
P^1	$9 * 10^{-3}$	1.05	$1 * 10^{-1}$	1040	1480
P^2	$4.9 * 10^{-3}$	1.05	$1 * 10^{-1}$	1830	2670
P^3	$6 * 10^{-4}$	1.05	$1 * 10^{-1}$	4030	5090
P^4	$5 * 10^{-4}$	1.05	$1 * 10^{-1}$	5220	6410
P^5	$1 * 10^{-4}$	1.05	$1 * 10^{-2}$	10110	12620

5.2 Visual Verification

To show that the GPU version calculates the correct results, a test-case with a flow around a NACA0012 profile is calculated. The flow speed is $|\vec{u}| = 0.5$ Ma under an angle of attack of one degree. For the viscous flow the Reynolds number is $Re = 2500$. The calculation is made with P^2 and this leads to 411840 DOFs. In Fig. 10 and Fig. 11 the Mach contour for inviscid and viscous flow, respectively, are shown. Dark red means a higher Mach number and dark blue means a lower Mach number. The GPU version works correctly for steady state problems compared to the results of the CPU. There is a difference in the iteration numbers because two different implementations are used on the CPU and on the GPU.

5.3 Performance

In this section the performance of the GPU version compared to the CPU version is described. At first the speed-up for the GPU code is shown on very small and very large mesh's for explicit and implicit time-stepping. For this tests the convergence is not important, only the same amount of calculations needed are compared. In the second subsection the convergence speed-up for implicit time-stepping is shown. The third subsection shows the speed-up for the Tesla C2050 compared to a Tesla C1060 and the performance of the GPU code in single precision.

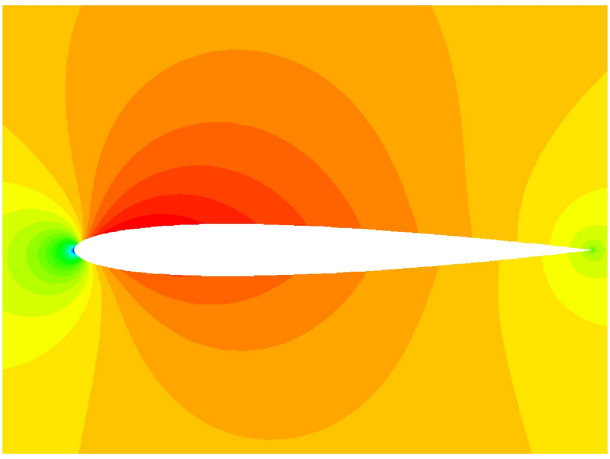

Figure 10: Inviscid Flow around a NACA0012 with an angle of attack of one degree (red=higher Mach number, blue=lower Mach number)

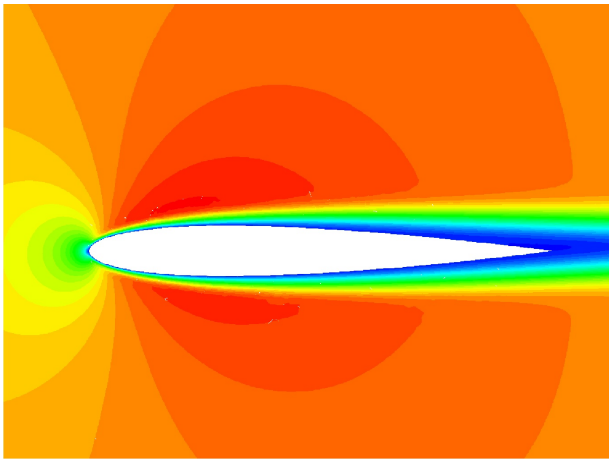

Figure 11: Viscous Flow around a NACA0012 with an angle of attack of one degree (red=higher Mach number, blue=lower Mach number)

5.3.1 Speed-Up for the GPU

For the performance tests simple test cases are chosen. The flow around a cylinder is calculated on different mesh sizes. In these tests the convergence is not important because only the calculation time is measured. The cylinder has a radius of 1 and the far field has a radius of 20.
1000 steps are calculated to get a good average time per step. The time step for all calculations is $\Delta t = 1{*}10^{-9}$ and for the implicit time-stepping the ratio is set to 1.0. This is necessary to avoid divergence. All unnecessary data output is avoided at both the GPU and the CPU calculations. The time between start of the first step and end of the last step is measured.
At first the results for explicit time-stepping are shown and later the ones for implicit time-stepping.

Inviscid Flow Simulation with Explicit Time-Stepping

In Tab. 5 the speed-up for the inviscid flux is shown. The mesh size vary from 64 cells to 1088830 cells. Some P^k can not be calculated on the biggest mesh because the memory on the GPU is not enough. As it

Table 5: Speed-Up between CPU and GPU for inviscid flux with explicit time-stepping

$P^k \backslash N_C$	64	412	992	1620	2804	5308	10304	22652	43732	50022	447944	1088830
P^1	5	23	48	65	72	84	108	104	110	109	111	**112**
P^2	9	33	63	73	84	92	100	106	109	110	111	**112**
P^3	14	49	80	89	96	103	110	114	112	**116**	113	114
P^4	13	59	86	101	105	117	121	128	**130**	129	121	–
P^5	19	76	104	108	112	118	121	124	**124**	124	117	–

is seen, the speed-up depends on the mesh size, but after a certain amount of cells the speed-up is for all P^k a factor above 100. For big mesh sizes the speed-up between one CPU and the GPU for all P^k is a factor around 110-130 for the explicit time-stepping inviscid flux.

Viscous Flow Simulation with Explicit Time-Stepping
The same mesh sizes are used to calculate the speed-up for the viscous flux (64-1088830 cells). The results can be seen in Tab. 6. Again the speed-up depends on the mesh size. For bigger meshes the speed-up is

Table 6: Speed-Up between CPU and GPU for viscous flow with explicit time-stepping

$P^k \backslash N_C$	64	412	992	1620	2804	5308	10304	22652	43732	50022	447944	1088830
P^1	8	29	49	67	77	87	88	104	102	**113**	107	111
P^2	11	37	63	69	80	90	97	104	107	109	**117**	116
P^3	17	49	77	83	87	98	104	110	112	111	**119**	118
P^4	14	59	82	92	97	108	115	119	125	127	**127**	–
P^5	18	74	98	100	104	114	119	123	125	125	**129**	–

between 109 and 129 for the viscous flux with explicit time-stepping.

Inviscid Flow Simulation with Implicit Time-Stepping
In this part the speed-up of the implicit time-stepping is shown for the case that the two versions have to carry out the same amount of calculations. This means the same number of steps are solved for both the CPU and the GPU version. The results can be seen in Tab. 7. In this test the higher polynomial degrees

Table 7: Speed-Up between CPU and GPU for inviscid flow with implicit time-stepping

$P^k \backslash N_C$	64	412	992	1620	2804	5308	10304	22652	43732	50022	183648	447944
P^1	1	8	14	18	27	44	62	74	84	85	85	**89**
P^2	3	9	17	26	38	51	60	66	69	70	**70**	69
P^3	3	10	22	32	41	48	53	56	57	58	**60**	–
P^4	5	14	29	38	44	51	52	53	54	**57**	–	–
P^5	4	19	34	40	43	45	46	47	47	**47**	–	–

$(P^3 - P^5)$ do not have a performance as good as the lower degrees $(P^1 - P^2)$ because the implementation of updating the solution is more time consuming for higher degrees. But still the speed-up for bigger meshes is between 47 and 89.

Viscous Flow Simulation with Implicit Time-Stepping
The speed-up for the viscous flux can be seen in Tab. 8. This time the speed-up for all P^k is higher

Table 8: Speed-Up between CPU and GPU for inviscid flux with implicit time-stepping

$P^k \backslash N_C$	64	412	992	1620	2804	5308	10304	22652	43732	50022	183648	447944
P^1	2	11	17	27	35	52	60	76	79	79	83	**88**
P^2	3	11	21	31	44	56	64	69	75	74	79	**84**
P^3	4	14	28	40	49	59	63	67	71	**74**	71	–
P^4	5	16	34	43	51	57	63	63	65	**65**	–	–
P^5	5	21	38	45	52	56	60	62	63	**63**	–	–

compared to the inviscid flow, because the updating of the solutions does not take so much percentage of the overall calculation, since the viscous flux calculation is more expensive under the aspect of calculation time than the inviscid flux calculation. The speed-up is between 60 and 88 for bigger meshes.

5.3.2 Implicit Convergence Speed-Up
In this subsection the convergence speed-up is calculated for the inviscid and the viscous flow. This time the flow around a NACA0012 with an angle of attack at one degree is chosen. The chord length of the airfoil is one and the far-field radius is 50. The number of cells are chosen from Tab. 7 and Tab. 8 in a way that the speed-up difference between the chosen mesh size and the highest speed-up are around 10. For this test the time to reach a residual lower than 10^{-11} was measured. The setup ($\Delta t, r$ and $\Delta t_{\max}$) for both, the CPU

and the GPU version, where the same. The beginning time step is $\Delta t = 1 * 10^{-3}$, the ratio is $r = 1.05$ and the maximal time step is $\Delta t_{\max} = 1 * 10^{-1}$. The flow speed under one degree is $|\vec{u}| = 0.5$ Ma.

Inviscid Flow Simulation
The results can be seen in Tab. 9. The convergence speed-up is mostly not as high as the speed-up with

Table 9: Convergence speed-up between CPU and GPU for inviscid flow with implicit time-stepping

	N_C	Iterations CPU	Time CPU (s)	Iterations GPU	Time GPU (s)	Speed-up
P^1	43768	3195	2909.235	3736	38.753	75
P^2	21576	2775	2523.489	3090	37.166	68
P^3	11270	2365	2156.122	2604	36.279	59
P^4	5434	2405	1780.75	2426	33.968	52
P^5	3820	2235	1964.954	2240	41.308	47

the same amount of calculations. This is because the CPU version for implicit time-stepping convergences with a lower amount of iterations (see Fig. 5.3.2).

Viscous Flow Simulation

The results can be seen in Tab. 10. In the Fig. 12 the convergence rate is plotted over iteration number. In

Table 10: Convergence speed-up between CPU and GPU for viscous flow with implicit time-stepping

	N_C	Iterations CPU	Time CPU (s)	Iterations GPU	Time GPU (s)	Speed-up
P^1	43768	3345	7597.375	3891	103.689	73
P^2	43768	3425	13597.59	3871	187.444	72
P^3	21576	3135	9148.825	3476	153.713	59
P^4	11270	2595	6573.779	2688	110.685	59
P^5	5434	2555	5065.542	2580	92.594	55

the case of the iterations the CPU version converges faster, but under the aspect of time the GPU converges much faster.

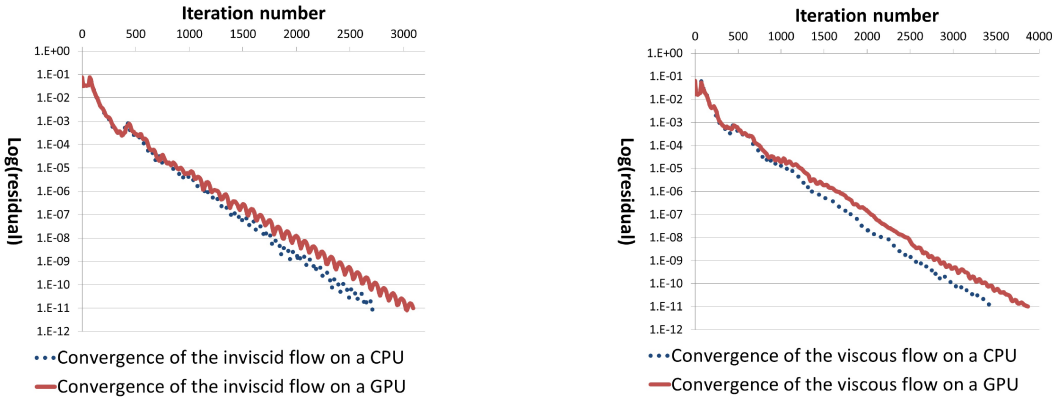

Figure 12: The convergence plotted for the inviscid and viscous flow for the GPU and CPU version

5.3.3 Single Precision Calculation and Calculation using C1060

In this subsection a couple of different tests are performed. At first the calculation accuracy is reduced to single precision. In the last part of this subsection an older generation of a NVIDIA-GPU is compared with

a newer one, to compare the calculation times. For all these calculations the number of cells are chosen from the tables 1 - 4. The mesh with the highest speed up is chosen for every P^k. This leads to the mesh sizes seen in Tab.11.

Table 11: Mesh-Sizes for the calculation

	Explicit, Inviscid	Explicit, Viscous	Implicit, Inviscid	Implicit, Viscous
P^1	1088830	50022	447944	447944
P^2	1088830	447944	183648	447944
P^3	50022	447944	183648	50022
P^4	43732	447944	50022	50022
P^5	43732	447944	50022	50022

Single Precision

For this section the code is changed in a way that only single precision arrays are used in the code. The code is compiled with the *nvcc* compiler, the flag "-arch=sm_12" is set to make sure that all operations are single precision floating point operations. With the mesh sizes seen in Tab. 11 the calculation in single precision is made. The speed-up is calculated between the CPU version (keep in mind that the CPU always calculates in double precision) and the double precision version of the GPU implementation. The results are seen in Tab. 12. The speed-up for the NVIDIA Tesla C2050 in single precision compared to the double precision

Table 12: Speed-Up between CPU with DP and GPU with SP (and in brackets the speed-up between SP and DP version on the GPU)

	Explicit, Inviscid	Explicit, Viscous	Implicit, Inviscid	Implicit, Viscous
P^1	192 (1.7)	228 (2.0)	144 (1.6)	176 (2.0)
P^2	184 (1.6)	214 (1.8)	113 (1.6)	154 (1.8)
P^3	184 (1.6)	203 (1.7)	93 (1.5)	120 (1.8)
P^4	184 (1.4)	196 (1.5)	89 (1.6)	102 (1.6)
P^5	174 (1.4)	193 (1.5)	62 (1.3)	87 (1.4)

calculations on this GPU is around the factor of 1.3 to 2.0. This speed-up gain is bought with a big loss of accuracy.

C1060

In this subsection two generations of NVIDIA-GPUs are compared. The C2050 as used before and one of one generation older, named C1060. The calculations are made in double and single precision and afterwards compared to the CPU version. Also the speed-up between these two cards is shown. The mesh sizes are the same as in the section above (Tab. 11 on page 21). First the calculation is done with double precision and the results can be seen in Tab. 13. This shows that for double precision the C2050 is around 3 times faster than the C1060. When the theoretical maximum double precision floating point calculation capability is compared the C2050 is at the most 4.2 times faster [15]. But since in this calculation a lot of memory reads and writes are needed the speed-up is only 3.7 and lower.
Also the calculation with single precision is done and the results can be seen in Tab. 14 (keep in mind that the CPU version is calculating in double precision). For this calculations in SP the C2050 is around 1.6 to 2 times faster than the C1060.

Table 13: Speed-Up between CPU and the C1060 with double precision (and in brackets the speed-up between the C2050 and the C1060 in DP)

	Explicit, Inviscid	Explicit, Viscous	Implicit, Inviscid	Implicit, Viscous
P^1	31 (3.6)	41 (2.7)	29 (3.1)	36 (2.4)
P^2	31 (3.6)	41 (2.8)	27 (2.6)	35 (2.4)
P^3	32 (3.6)	41 (2.9)	25 (2.4)	30 (2.5)
P^4	35 (3.7)	41 (3.1)	26 (2.2)	27 (2.4)
P^5	33 (3.7)	41 (3.1)	23 (2.0)	28 (2.2)

Table 14: Speed-Up between CPU with DP and the C1060 with SP (and in brackets the speed-up between the C2050 and the C1060 in SP)

	Explicit, Inviscid	Explicit, Viscous	Implicit, Inviscid	Implicit, Viscous
P^1	125 (1.5)	123 (1.8)	73 (2.0)	94 (1.9)
P^2	118 (1.5)	124 (1.7)	57 (2.0)	84 (1.8)
P^3	117 (1.6)	122 (1.7)	48 (1.9)	66 (1.8)
P^4	120 (1.5)	121 (1.6)	46 (1.9)	56 (1.8)
P^5	116 (1.5)	121 (1.6)	41 (1.5)	56 (1.5)

6 Conclusion and Future Work

This study of the implementation of the CPR formulation on GPUs shows, that a high-order method can get a huge performance gain running on one GPU instead of CPUs. With the Euler and Navier-Stokes equations solved with an explicit and an implicit time integration scheme for different polynomial of degree k, the implementation on a GPU is flexible and can be used for a wide range of 2D CFD calculations. The adaption from a CPU version to a GPU version is not difficult and the version for the GPU is understandable without much background information of GPU computing. The high parallel computation performance of a GPU shows speed-up factors of two orders of magnitude for the explicit time integration compared to a non parallelized CPU code. The capability of the GPU implementation to calculate all cells of a mesh apparently *at once* is the reason for this high performance gain. Using the implicit time-stepping scheme the speed-up factor is between 47 and 89 depending on the degree k of the polynomial and the used numerical flux. It must be said, that the developing of the cell coloring approach was the breakthrough to gain these results. With this concept a GPU is able to calculate each color apparently *at once* and so can use its high capability of parallel computing.
The comparison between two GPU generations and also the comparison between DP- and *SP*-implementation shows, that this GPU implementation has a memory bound, which means, that the ratio between global memory data access and kernel-instructions reaches not the theoretical optimum. The study shows that the CPR formulation can be implemented on a GPU in an efficient and flexible way because this formulation is very well suited for the GPU architecture.
Since the implementation of the CPR formulation on one GPU works so well for 2D triangular meshes the extension for quadrangular cells should give a similar performance speed-up. The extension to 3D grids is of course more complicated, but the results from this study can help for the 3D implementation. It should be considered to run these calculations on multiple GPUs because one might not have enough memory to store a reasonable sized 3D calculation. The coloring of course must be adapted to 3D if possible and if not, a similar way could be used.
The coloring algorithm that is implemented now could be the reason for the slower convergence (in terms of iteration numbers) on the GPU. A new algorithm could be implemented that trys to get always the same color pattern for the cells, which means red in the middle and the other colors equally around it. Of course this only works for triangular cells.

References

[1] T. Haga Z.J. Wang, H. Gao. A unifying discontinuous formulation for hybrid meshes. *World Scientific Review Volume*, (15):423–453, 2010.

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

[3] H. Gao Z.J. Wang. A unifying lifting collecation penalty formulation including the discontinuous galerkin, spectral volume/difference methods for conservation laws on mixed grids. *J. Comput. Phys.*, (228):8161–8186, 2009.

[4] C. Hirsch. *Numerical Computations of Internal and External Flows: Volume 2 Computational Methods for Inviscid and Viscous Flows*. John Wiley and Sons, 1st edition, 1990.

[5] S. Rebay F. Bassi. GMRES discontinuous galerkin solution of the compressible Navier-Stokes equations. In *Discontinuous Galerkin methods: Theory, Computations and Apllications*, pages 197–208. Springer-Verlag Berlin, 2000.

[6] H. T. Huynh. A reconstruction approach to high-order schemes including discontinuous Galerkin methods. *AIAA-2009-403*, 2009.

[7] Gregor Gassner, Frieder Lörcher, and Claus-Dieter Munz. A contribution to the construction of diffusion fluxes for finite volume and discontinuous galerkin schemes. *Journal of Computational Physics*, 224(2):1049 – 1063, 2007.

[8] P.-O. Perrson J. Peraire. The compact discontinuous galerkin (CDG) method for elliptic problems. *SIAM J. Sci. Comput.*, (30):1806–1824, 2008.

[9] C. W. Shu S. Gottlieb. Total variation diminishing Runge-Kutta schemes. *Math. Comput.*, (67):73–85, 1998.

[10] Y. Liu Y. Sun, Z.J. Wang. Efficient implicit non-linear LU-SGS approach for compressible flow computation using high-order spectral difference method. *Commun. Comput. Phys.*, (5):760–778, 2009.

[11] W. Haken K. Appel. Every map is four colourable. *Bulletin of the American Mathematical Society*, (82):711–712, 1976.

[12] P. Seymour R. Thomas N. Robertson, D. Sanders. The four-colour theorem. *J. Combinatorial Theory, Series B*, (70):2–44, 1997.

[13] G. Gonthier. Formal proof - the four-color theorem. *Notices of the American Mathematical Society*, (55):1382–1393, 2008.

[14] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 2nd edition, 2000.

[15] NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Fermi. Whitepaper, NVIDIA, Santa Clara, 2009.