# Efficiency Improvement of Optimization of Ships

## Literature Survey

by

D. R. De Bruycker

TU Delft Delft University of Technology

# List of Figures

# List of Algorithms

# Contents

# 1 Introduction

Computational Fluid Dynamics (CFD) has evolved greatly over the past decades, resulting in a powerful tool for analyzing all kinds of problems involving fluid flows. Being used in many different fields, it now plays an essential role in engineering design and scientific research. As well for ship engineering, where ship design in the past mostly relied upon the design of previous ships and on empirical hull and propeller data. More rapid development and innovation therefore became possible with the advent of ship CFD, allowing for more insight into ship hydrodynamics and making it possible to analyze and predict ship performance with more certainty. At MARIN, the Maritime Research Institute Netherlands, CFD for ships is being used for a wide variety of applications; from consultancy in an early design stage to high-end calculations in order to accurately determine the ship's performance [2]. Generally, MARIN makes use of their in-house developed CFD tools. They have several panel codes available for non-viscous flow computations, as well as two different viscous flow solvers.

MARIN's two main CFD tools solve the Reynolds-Averaged Navier-Stokes (RANS) equations, taking into account friction and allowing for analyzing flow phenomena dominated by viscosity. Parnassos, their first RANS-solver, is very fast and a dedicated code for computing the steady flow around ship hulls. It is, however, limited to simple geometries such as ship hulls without appendages. ReFRESCO, on the other hand, is a more general code, applicable to steady or unsteady flows and complicated geometries. It makes use of unstructured grids, resulting in having to solve systems characterized by large matrices with an unknown structure. Therefore, ReFRESCO is computationally more heavy and a lot slower than Parnassos. The code is also newer and still under constant research in order to make it into a more efficient and robust solver. ReFRESCO is a very powerful tool due to its general applicability, however, when simple hull shapes need to be computed, it is favorable to use Parnassos. Development of Parnassos started in the 1980s and is by now optimized with respect to robustness, accuracy and efficiency. Parnassos is even an order of magnitude faster than other RANS-codes. It owes its fast computation rates to a combination of the use of structured grids, resulting in sparse diagonally structured matrices, and the memory efficient character of its solving strategy. Even for a grid of over a million grid points, Parnassos can simply be run on your home desktop. This is unique for a RANS-solver.

As a result of its capability for carrying out fast viscous flows computations, Parnassos is very well-suited for use in optimization. MARIN therefore frequently uses Parnassos for optimizing ship hull form design. This is done through a multi-objective optimization where ship hulls are optimized for (1) minimum required power, and (2) best wake field quality. New hull shapes are generated through an interpolation of different existing basic hull shapes. These systematic hull form variations are then analyzed using Parnassos. Many different hull shape variations are generally automatically generated. Parnassos therefore needs to compute the flow properties around all these ship hulls such that the results can be used for determining the Pareto front resulting from satisfying the two optimization requirements. Even for being a fast RANS-solver, it still takes Parnassos a few hours (or even days) for one hull form evaluation.

Automatic optimization with hundreds or thousands of calculations is thus impractically time consuming. It is therefore desired to significantly lower down computation times, which can be achieved by decreasing the time needed for the flow calculations. In Parnassos, more than 80% of CPU time is spent in the solution of large systems of linear equations $Ax = b$, in which the coefficient matrix is sparse. The current technique to solve these systems is by using an iterative method (GMRES) combined with an incomplete LU decomposition as preconditioner. Improving the performance of this linear solver can be obtained through the use of high performance computing. Especially using the Graphical Processing Unit (GPU) has great potential for speeding up scientific computations. The highly parallel structure of GPUs and their good floating-point performance makes them well-suited for problems that can make use of data parallelism. Meaning that GPUs are mainly efficient for handling algorithms where large blocks of data can be processed in parallel. This applies to the linear system solve in Parnassos. Furthermore, GPUs can carry out their fast parallel computations at a rather low cost.

Overall efficiency of parallel computations is strongly influenced by the problem to be solved, the discretization technique applied and the solution algorithms used, as well as on the characteristics of the hardware. Parnassos employs a combination of high-order finite difference schemes, resulting in a sparse diagonally structured matrix. Knowing the structure of the matrix has a great advantage in choosing an appropriate solving algorithm. The preconditioned Krylov solver, now used for sequential computations, unfortunately is not well-suited for parallel implementation. As will become clear from this report, parallel implementation of these solvers requires quite some effort, especially finding a good GPU implementation for the preconditioner is a real challenge.

The purpose of this work is therefore to investigate: *How to / Is it possible to achieve reasonable speedup of Parnassos' linear system solver by making use of GPU computing?* The main focus will be on the choice of preconditioner and how to efficiently implement it on the GPU. This report is the literature survey prior to the actual work, meant to gather all necessary information and to give an overview of the state-of-the-art. The structure of this report is as follows. Chapter 2 describes the overall solving strategy employed by Parnassos and will present some more detailed info on the linear system solve and the construction of its coefficient matrix $A$. This linear system is solved using a preconditioned Krylov solver, hence, a general overview on these solvers and preconditioning, will be given in chapter 3. Chapter 4 then introduces GPU computing and how it can be employed for speeding up scientific computing applications. Preconditioned Krylov solvers, suitable for solving Parnassos' linear solve on the GPU, as well as techniques for further improvement of these algorithms, will be the subject of chapter 5. The chapter will also introduce some topics relevant for carrying out a parallel performance analysis. The report will end in chapter 6 by stating the questions needing to be answered by the further research, as well as the experimental setup that will be used for (hopefully) finding those answers.

# 2

# Parnassos: a RANS solver for structured grids

Parnassos is the in-house CFD-tool at MARIN for solving the steady incompressible viscous flow around a ship hull. It does so at both model and full scale. Development of the software started in the 1980s and happened in collaboration with Instituto Superior Técnico (IST) in Lisbon, Portugal. By 1995 it was ready to be used for practical applications. It is now a commonly used tool at MARIN, providing detailed information about the velocity and pressure field around all of the ship hull (including the wake field in the propeller plane), essential for analyzing and optimizing ship hull design. Due to the use of structured grids the tool can only be used for relatively simple geometries, meaning ship hulls without appendages. As a more general tool, MARIN uses another in-house software named ReFRESCO. Through the years, Parnassos has evolved into a tool providing accurate solutions within a reasonable amount of time. An important design criterion has been to obtain a robust piece of code, and mainly to achieve this without sacrificing accuracy. Besides *accuracy*, *efficiency*, and *robustness*, there has been another key parameter in its development, namely *flexibility*. The main solving strategies upon which Parnassos is based are thoroughly explained in [33]. Below follows a brief overview of the solution strategy used in order to achieve the previously mentioned characteristics. The first section describes the overall solution strategy of which Parnassos makes use. Section 2.2 and 2.3 then go more in depth into the part of the routine that will be the focus of the remainder of this report.

## 2.1. Solution strategy

Parnassos solves the steady incompressible viscous Navier-Stokes equations. The flow around a ship hull is characterized by high Reynolds numbers and is turbulent in nature. Turbulence is simulated using the Reynolds-Averaged Navier-Stokes equations (RANS). Therefore Parnassos actually solves the following continuity and momentum equations [33]:

$$\overline{\mathrm{u}}_{i,i} = 0 \tag{2.1}$$

$$\rho\overline{\mathrm{u}}_j\overline{\mathrm{u}}_{i,j} - \mu\overline{\mathrm{u}}_{i,jj} + \overline{p}_{,i} + \rho(\overline{\mathrm{u}_i'\mathrm{u}_j'})_j = \bar{\mathrm{f}}_i \tag{2.2}$$

where $\rho\overline{\mathrm{u}_i'\mathrm{u}_j'}$ is the Reynolds stress, describing turbulence. The bars above the quantities in (2.1) and (2.2) indicate their time average, however, these will be dropped in the discussion that follows. The Reynolds stress requires the addition of a turbulence model. Parnassos allows the use of different models. This already illustrates its flexible character. There is the option of using the simple algebraic Cebeci-Smith model [18], not requiring any additional, so-called transport equations, to be solved. Another possibility is to use Menter's one-equation turbulence model [39] or the two-equation SST k-$\omega$ model [40]. Furthermore, Menter's turbulence model can be used with the Dacles-Mariani correction [19].

The physical domain will be referred to by the curvilinear coordinate system $(\xi, \eta, \zeta)$, where $\xi$ is roughly the mainstream direction taken positive downstream, $\eta$ is the wall-normal direction, and $\zeta$ is denoted as the girthwise direction. There has been chosen for a body-fitted grid in physical space. This allows for the use of largely stretched out cells near the wall of the hull. With these grid contractions, high gradients in the boundary layer are taken into account which omits the need for adding wall functions, not even when computing the viscous flow at full-scale. This is a strong, and very unique, feature of the RANS-solver. Furthermore the grid is structured and consists out of different blocks, fitting the needs for computing flow phenomena in certain parts of the domain. For reasons of numerical discretization, the physical space and its body-fitted grid, are transformed into a rectangular computational domain, $(x, y, z)$, in which a uniform Cartesian grid is used. This is illustrated in figure 2.2[1]. Parnassos then uses a mapping from the computational domain to the physical domain. Also the RANS-equations of (2.1) and (2.2) need to be rewritten according to the new coordinate system. For more details on the transformation equations and the resulting RANS-equations in curvilinear coordinates one may refer to [33].

Discretization of the curvilinear RANS-equations is done through a combination of high-order finite difference schemes (at least second-order) aiming for an accurate solution while keeping the discretization scheme stable. In [33] a full and detailed description is given of the discretization of each of the individual terms in the equations. Here, only a short summary is given according to [53]. As illustrated by the discretization stencils in figure 2.1:

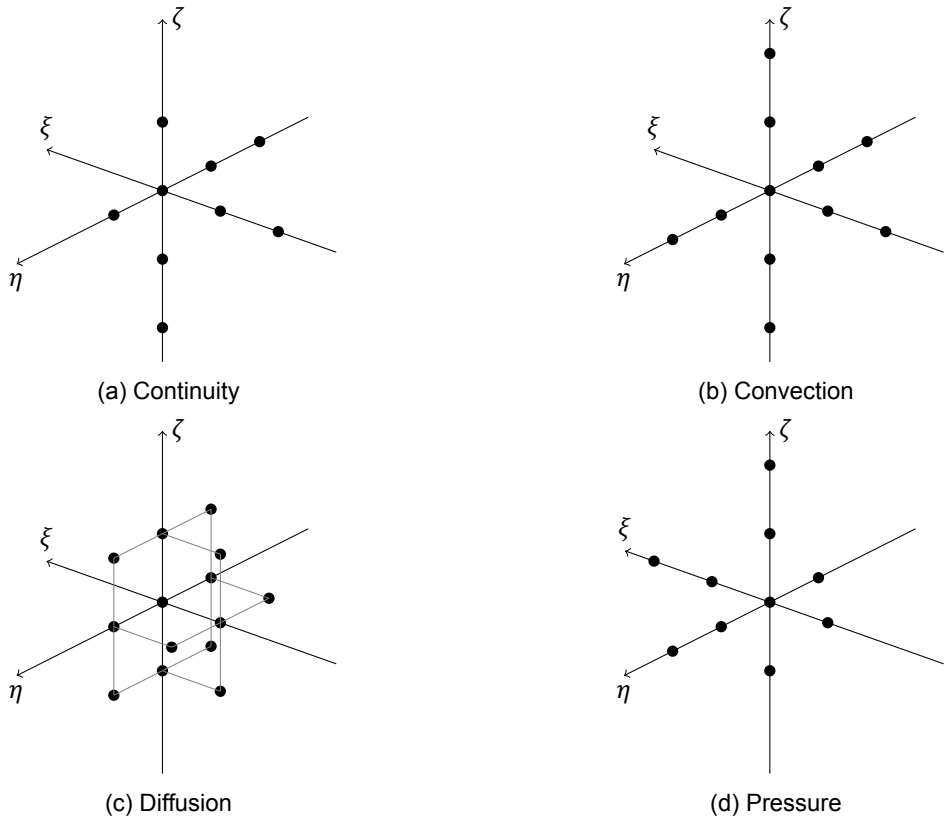
(a) In the continuity equation, a second-order three-point backward scheme is used for the terms differentiated in mainstream ($\xi$) direction, and for the derivatives in normal ($\eta$) and girthwise ($\zeta$) direction, a third-order four-point scheme is used.

(b) For the convective term in the momentum equations, a second-order and a third-order upwind scheme are used for the derivatives of the velocity in mainstream direction and in normal and girthwise direction, respectively.

(c) A second-order central-difference scheme is used for all second derivatives in the diffusion terms in the momentum equations.

(d) The pressure gradients in the momentum equations are discretized using a third-order scheme.

[1] Due to symmetry of the shape of a ship hull only the starboard side is taken into account in order to reduce computational time.




(a) Continuity (b) Convection

(c) Diffusion (d) Pressure

Figure 2.1: Discretization stencils for (a) the terms in the continuity equation and for the (b) convective, (c) diffusive and (d) pressure terms in the momentum equation [33]

The above discretization leads to a large system of nonlinear equations that needs to be solved for the unknowns at each grid point, represented by the grid indices $i$, $j$ and $k$:

$$F(\phi) = F(u_{ijk}, v_{ijk}, w_{ijk}, p_{ijk}, (\nu_t)_{ijk}) = 0 \tag{2.3}$$

where $u$, $v$ and $w$ are the three velocity components of the vector $\overline{\mathrm{u}}$, $p$ is the pressure and $(\nu_t)$ is the eddy viscosity coming from the turbulence model. The equations are linearized using the quasi-Newton linearization, implying that the nonlinearity is taken care of by solving the equations iteratively according to:

$$\phi^{m+1} = \phi^m - \frac{F(\phi^m)}{F'(\phi^m)} \tag{2.4}$$

where $m$ is the nonlinear iteration step. Rearranging the terms leads to the following:

$$A\phi = b \tag{2.5}$$

a linear system of equations that needs to be solved at each step of a Newton iteration. The coefficient matrix $A = F'(\phi)$ is the Jacobi matrix, a large sparse squared matrix. Due to the discretization scheme used, the structure of this matrix is entirely know, a major advantage for choosing the appropriate iterative algorithm for solving (2.5). This will be further discussed in section 2.2.

In developing Parnassos, an important question has been how to deal with such a large system of equations. The common approach is to use an iterative solution strategy which includes, in one way or another, a partitioning of the original system to reduce its size. Generally this size reduction is dealt with by uncoupling the equations (2.1) and (2.2). The Navier-Stokes are then solved by employing a pressure correction, or by adding an artificial compressibility constraint. However, for high Reynolds number flows, such iterative methods might experience problems with convergence due to the difficulty in restoring the coupling between the continuity and momentum equations. Due to the high level

of importance given to the development of a robust tool, there has been opted for a solution method maintaining the coupling of the equations. The transport equation(s), in case of choosing for a non-algebraic turbulence model, can be solved uncoupled in order to minimize computational complexity.

Solving the coupled equations adds to the robustness of the solver, however, the issue of wanting to reduce the size of the system remains. In Parnassos this is taken care of by dividing the computational domain into subdomains. The equations are then solved per subdomain which significantly reduces the amount of unknowns that need to be solved for simultaneously. This is what will be referred to as the space-marching method. The domain is divided in $\xi$=constant planes, perpendicular to the mainstream direction, as illustrated in figure 2.2. The planes are visited in downstream order. Using this space-marching method one literally sweeps through the computational domain. This sweeping is what will be denoted as the global iteration, i.e. where one iteration step equals one sweep in downstream direction. The moving in downstream direction was a rather obvious choice because high Reynolds number flows are convection-dominated, hence information transport mainly happens in mainstream direction. However, to maintain the elliptical character of the equations it is important to restore the communication of information happening in upstream direction. This is done by including a pressure correction in upstream direction into the global iteration loop. It basically consists of a simple algebraic operation, where a correction value for the pressure is determined based on the previously computed flow values. These corrections are then applied to update all pressure values by sweeping through the domain in upstream direction. Once this is finished, the next downstream sweep can begin. Each global iteration step therefore consists out of two main cycles: (1) a downstream sweep where all velocities and pressure are computed, and (2) an upstream sweep in order to update the pressure values. This is done until a sufficiently accurate solution is obtained.

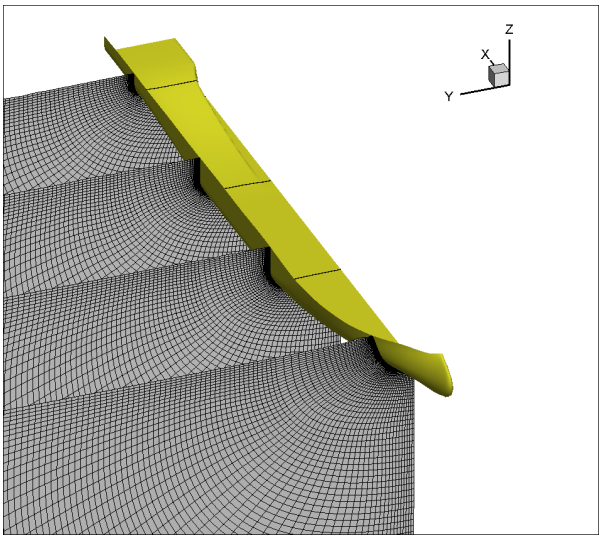



Figure 2.2: View of the computational domain employed in Parnassos

In the early days of Parnassos the global iteration was carried out by using the plane-by-plane approach as described in the above. The main advantage being that it results in a very memory efficient method. Due to this Gauss-Seidel approach of the global iteration loop, all values can be updated as soon as they have been computed. Therefore, only the values of one subdomain, i.e. one plane, need to be stored. Later on, when memory became less of a limitation, the effect of choosing larger subdomains has been studied. Each subdomain now consists out of $g$ different $\xi$=constant planes which requires to store the variables of all $g$ planes. It was observed that this multi-plane approach resulted in significant savings in computation time. In order to let the global iteration converge with the plane-by-plane approach, several steps of the Newton iteration are needed per plane before sweeping to the next one. When using larger subdomains only one Newton iteration suffices. This indicates that the global iteration itself takes care of the nonlinear character of the equations, increasing the efficiency of the solver.

From the above it can be summarized that one step of the global iteration loop consists out of the following:

1. A downstream sweep through the domain. The following steps are then carried out for each of the subdomains:

   1.a) At first, the eddy viscosity $(\nu_t^n)_{ijk}$ is determined by making use of the previously computed

velocities and pressure, i.e. solve:

$$F(u_{ijk}^{n-1}, v_{ijk}^{n-1}, w_{ijk}^{n-1}, p_{ijk}^{n-1}, (v_t^n)_{ijk}) = 0 \tag{2.6}$$

1.b) Next, the RANS-equations are linearized using the quasi-Newton method, and the linear system of coupled equations is solved for all velocities and pressure over the full subdomain.

2. The upstream pressure correction

where $n$ indicates the number of global iteration steps.

Further on in the development of Parnassos, a free-surface (FS) treatment has been included.This adds the (viscous) effects ships have on the wave pattern of the water surface, which in turn affects the flow around the ship hull, increasing the accuracy of the computed solutions. The FS method basically adds a more elaborate boundary condition to the equations which takes into account those free-surface effects. This boundary condition initially assumes a flat water surface, or any other water surface computed by one of the in-house panel codes. The boundary condition then gets updated using newly computed flow information. Note that this update adds an extra iteration loop around the so-called global iteration. It also updates the grid, making it computationally heavy. Therefore the update is only carried out every so often. Generally this is done after approximately 100 sweeps, i.e. 100 global iteration steps. For solving the extra equations that come with this approach, a steady iterative formulation is used [47][52] such that no time-dependent terms are added, which actually made it possible to solve the problem by using iteration rather than time stepping. This also allowed for the limited number of free-surface updates required.

Parnassos' solution strategy resulted from many years of research and, as can be concluded from the above discussion, succeeds at combining accuracy and efficiency while maintaining a robust RANS-solver. In order to preserve the efficient character of the solver it is essential to choose a suitable algorithm for solving the linear iteration loop at each of the subdomains. This part of the solver, indicated as 1.b) in the above, takes over 80% of the total CPU time spent on a computation [53]. Solving the systems of linear equations requires a significantly large amount of iterations and therefore becomes the main bottleneck when trying to further increase the rate with which a computation is executed in Parnassos. The aim of this thesis is to accelerate computations done in Parnassos, thus in other words, to speed up the linear iteration loop. The next section will therefore go a little more in depth into this part of the solver.

## 2.2. The linear system solver

The system of linear equations, given by equation (2.5), needs to be solved for each subdomain and up until a predefined accuracy. These systems, characterized by a large and sparse coefficient matrix, are generally solved by iterative Krylov methods, used in practice with a preconditioner to improve convergence. In Parnassos, further iterative improvement is achieved through the use of the defect-correction method. Hence, each new approximation is computed through solving a correction to the linear system based on the defect vector, being the residual associated to the previous approximation. The corrected linear system is then solved by a preconditioned Krylov solver, of which many different types exist, all with their strengths and weaknesses. Typically their performance is highly problem-dependent. One of the focus points of this thesis will therefore be to find the more suitable algorithm for solving the linear system characterized by the matrix $A$ given in (2.5). A major advantage of Parnassos is that, due to the discretization schemes used, the structure of $A$ is exactly known. A good understanding of this matrix structure is crucial for the remainder of the thesis work, and thus the build-up of $A$ will be explained in more detail below.

Let's first agree on the notation of some parameters in order to keep a clear discussion going. Let $NX$, $NY$ and $NZ$ be the number of grid points in mainstream, normal, and girthwise direction, respectively. As previously mentioned, $g$ refers to the number of planes taken per subdomain. Parnassos' space-marching approach therefore results in solving linear systems, as given by equation (2.5), with the size of $A$ equal to $(4 \times g \times NY \times NZ) \times (4 \times g \times NY \times NZ)$. Note that $4$ denotes the variables $\{u, v, w, p\}$ that

need to be solved for per grid point. The use of larger subdomains thus results in a matrix being $g$ times bigger as compared to the matrix resulting from the original plane-by-plane approach. Furthermore it also contains extra terms accounting for the coupling between the planes in the subdomain.

The entries of $A$ are grouped in blocks, each block representing all coefficients that need to be multiplied with all variables belonging to a certain $\xi$=constant plane. The size of such a squared matrix block thus equals $(4 \times NY \times NZ)$. At this point it is useful to have a look again at the discretization stencils presented in figure 2.1 such that it can be understood how the discretization scheme results in the following penta-diagonal structure for $A$:

$$A = \begin{bmatrix} d_1 & e_1 & f_1 & 0 & 0 & 0 & \cdots \\ c_2 & d_2 & e_2 & f_2 & 0 & 0 & \cdots \\ b_3 & c_3 & d_3 & e_3 & f_3 & 0 & \cdots \\ 0 & b_4 & c_4 & d_4 & e_4 & f_4 & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & \cdots & 0 & 0 & b_g & c_g & d_g \end{bmatrix} \tag{2.7}$$

The result is a sparse matrix with the only non-zero blocks being:

- The $d$-blocks: containing the coefficients that need multiplying with the variables of the current $\xi$=constant plane. Note that with the plane-by-plane approach, $A = d$ for each of the subdomains.
- The $b$- and $c$-blocks: containing the coefficients that need multiplying with the variables of the two consecutive planes in upstream direction.
- The $e$- and $f$-blocks: containing the coefficients that need multiplying with the variables of the two consecutive planes in downstream direction.

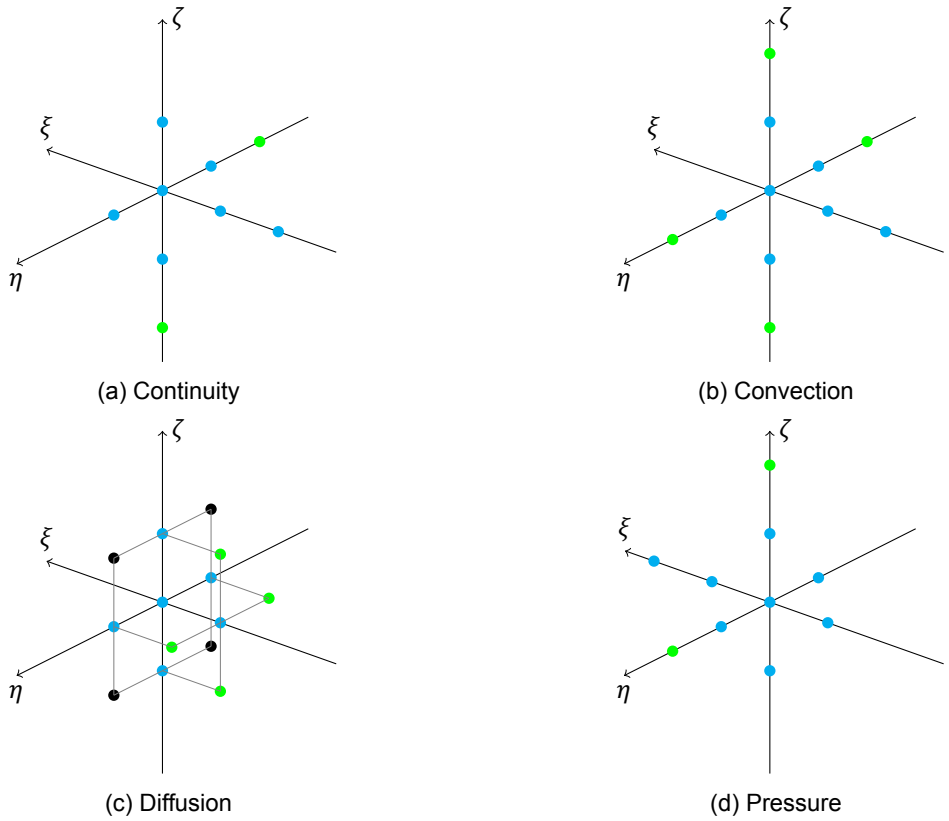
All blocks $b$, $c$, $d$, $e$ and $f$ are then build up out of small 4×4 block-structures representing the coefficients associated with the variables of one grid point in a $\xi$=constant plane.

## 2.3. Construction of the coefficient matrix $A$ in Parnassos

The previous sections outlined the theoretical principles upon which Parnassos' solution strategy is build, and the resulting coefficient matrix $A$ was introduced. This matrix characterizes the linear system of equations for which a more efficient solver strategy will be investigated in the remainder of this report. Therefore this section will further discuss the actual construction of $A$ in Parnassos, and present its structure in more detail. In Parnassos, the routine called `coefU.f` handles the discretization, and linearization, of the continuity and momentum equations, constructing the coefficient matrix $A$. This matrix is then passed to the routine called `preclinsol.f` which takes care of the linear solve of $Ax = b$. All notations used in the discussion below will be in accordance with the names given in the code itself.

In theory, the coefficient matrix $A$ should be build up according to the discretization stencils given in figure 2.1 of chapter 2. However, in solving the linear system $Ax = b$, robustness and speed are two main requirements, and for this it is beneficial for $A$ to be diagonally dominant. The use of higher order discretization schemes is harmful to this design of matrix structure, yet necessary to ensure sufficiently accurate solutions to the RANS-equations. Therefore, in order to maintain accuracy of the solution, while solving the linear system in a robust and efficient way, the points in these stencils are treated in three different ways. This is illustrated in figure 2.3 using the following color code:

- Points are taken into account explicitly: they are included into the right-hand-side of the system to be solved;
- Points are taken into account using deferred correction [34]: through an iterative solution method the points are taken into account for computing the solution without having any effect on the matrix $A$;
- Points are taken into account implicitly: they are included into the matrix $A$ itself.




(a) Continuity (b) Convection

(c) Diffusion (d) Pressure

Figure 2.3: Discretization stencils, as taken from theory, for (a) the terms in the continuity equation and for the (b) convective, (c) diffusive and (d) pressure terms in the momentum equation

Hence, the points indicated in light blue are the only ones directly used in constructing $A$. The green and black points do not affect its structure, however, they are used to compute the solution.
Remember that these finite difference discretization schemes result in the following penta-diagonal structure for the matrix $A$:

$$A = \begin{bmatrix} d & e & f & 0 & 0 & 0 & \cdots \\ c & d & e & f & 0 & 0 & \cdots \\ b & c & d & e & f & 0 & \cdots \\ 0 & b & c & d & e & f & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & \cdots & 0 & 0 & b & c & d \end{bmatrix} \tag{2.8}$$

Parnassos uses $4 \times 4$ structures to construct the above matrix, each block representing the coefficients that need multiplying with the variables of one grid point. All $4 \times 4$ blocks are constructed as follows:

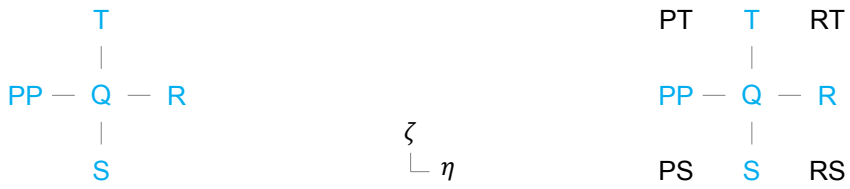
- Row 1 represents the momentum equation in $\xi$-direction;
- Row 2 represents the continuity equation;
- Row 3 represents the momentum equation in $\zeta$-direction;
- Row 4 represents the momentum equation in $\eta$-direction.

And:

- Coefficients in column 1 multiply with $u$ (i.e. velocity in $\xi$-direction);
- Coefficients in column 2 multiply with $v$ (i.e. velocity in $\eta$-direction);
- Coefficients in column 3 multiply with $w$ (i.e. velocity in $\zeta$-direction);

- Coefficients in column 4 multiply with $p$ (i.e. the pressure);

The $d$-blocks in the matrix given in (2.8) are then constructed using the $4 \times 4$ blocks named Q, PP, R, T and S, representing all coefficients that need multiplying with the variables in the current $\xi$=constant plane. This is represented by the 5-point stencil given below in figure 2.4a.




(a) 5-point stencil, including points in $\xi$=constant plane implicitly taken into account

(b) Stencil, also including the points in $\xi$=constant plane explicitly taken into account

Figure 2.4: Stencils illustrating matrix representation in Parnassos

Figure 2.4b also shows the blocks PT, PS, RT and RS, resulting from the diffusion terms in the momentum equations indicated by the black dots in figure 2.3c. These blocks will not be used for constructing $A$ but will be explicitly taken into account by including them in the right-hand-side of the system. From the 5-point stencil in figure 2.4a it can be understood that the $d$-blocks also have a penta-diagonal structure:

$$d = \begin{bmatrix}
\mathrm{Q} & \mathrm{R} & 0 & \cdots & \mathrm{T} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
\mathrm{PP} & \mathrm{Q} & \mathrm{R} & 0 & \cdots & \mathrm{T} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
0 & \mathrm{PP} & \mathrm{Q} & \mathrm{R} & 0 & \cdots & \mathrm{T} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & \cdots & \cdots & \mathrm{PP} & \mathrm{Q} & 0 & \cdots & \cdots & 0 & \mathrm{T} & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\
\mathrm{S} & 0 & \cdots & \cdots & 0 & \mathrm{Q} & \mathrm{R} & 0 & \cdots & 0 & \mathrm{T} & 0 & \cdots & \cdots & \cdots & 0 \\
0 & \mathrm{S} & 0 & \cdots & \cdots & \mathrm{PP} & \mathrm{Q} & \mathrm{R} & 0 & \cdots & 0 & \mathrm{T} & 0 & \cdots & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & \cdots & \cdots & 0 & \mathrm{S} & 0 & \cdots & 0 & \mathrm{PP} & \mathrm{Q} & \mathrm{R} & \cdots & \cdots & 0 & \mathrm{T} & 0 \\
0 & \cdots & \cdots & \cdots & 0 & \mathrm{S} & 0 & \cdots & 0 & \mathrm{PP} & \mathrm{Q} & 0 & \cdots & \cdots & 0 & \mathrm{T} \\
0 & \cdots & \cdots & \cdots & \cdots & 0 & \mathrm{S} & 0 & \cdots & \cdots & 0 & \mathrm{Q} & \mathrm{R} & \cdots & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & \mathrm{S} & 0 & \cdots & 0 & \mathrm{PP} & \mathrm{Q} & \mathrm{R} & 0 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & \mathrm{S} & 0 & \cdots & 0 & \mathrm{PP} & \mathrm{Q} & \mathrm{R} \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & \mathrm{S} & 0 & \cdots & 0 & \mathrm{PP} & \mathrm{Q}
\end{bmatrix} \tag{2.9}$$

with $d$ being a square matrix of size $(4 \times NY \times NZ)$. The other blocks in (2.8) all have a diagonal structure since they only contain coefficients that need multiplying with the variables in the consecutive planes at the same $\eta$ and $\zeta$ positions. Recall that the $b$- and $c$- blocks are being multiplied with the two consecutive planes in upstream directions, hence their coefficients are stored in $4 \times 4$ blocks named UPSTRa and UPSTRb, forming the following diagonal $b$- and $c$-matrices:

$$b = \begin{bmatrix}
\mathrm{UPSTRb} & 0 & \cdots \\
0 & \mathrm{UPSTRb} & \cdots \\
\vdots & \ddots & \ddots \\
0 & \cdots & \mathrm{UPSTRb}
\end{bmatrix} \tag{2.10}$$

$$c = \begin{bmatrix}
\mathrm{UPSTRa} & 0 & \cdots \\
0 & \mathrm{UPSTRa} & \cdots \\
\vdots & \ddots & \ddots \\
0 & \cdots & \mathrm{UPSTRa}
\end{bmatrix} \tag{2.11}$$

The $e$- and $f$-blocks, necessary for the multiplication with the two consecutive planes in downstream direction, are then likewise build by storing their coefficients in $4 \times 4$ blocks named DOWNSTRa and

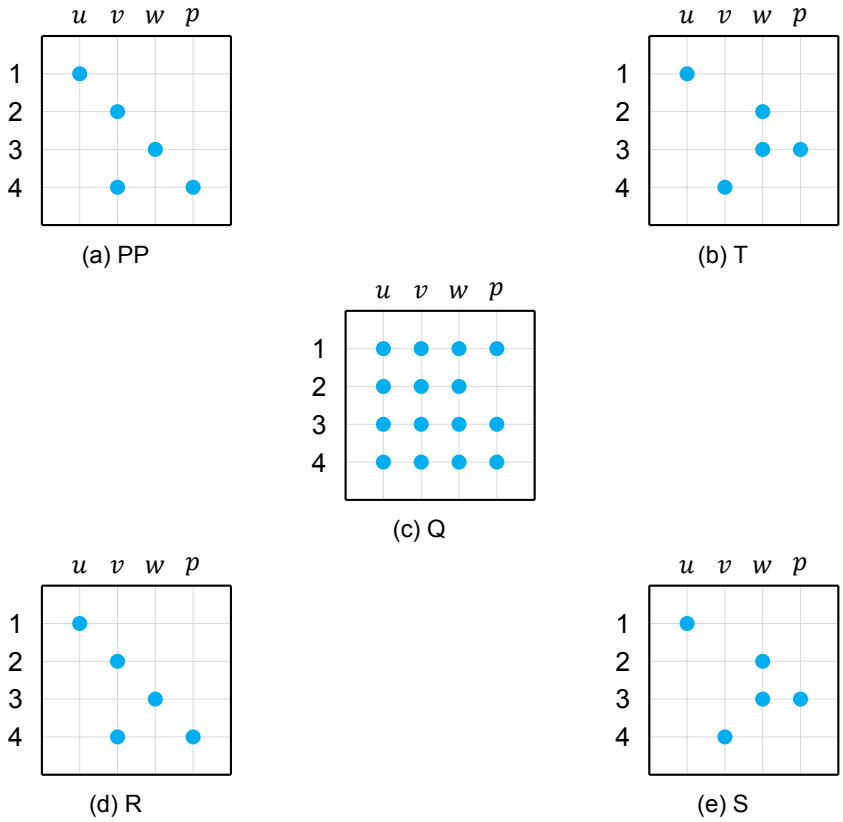
DOWNSTRb:

$$e = \begin{bmatrix} \text{DOWNSTRa} & 0 & \cdots \\ 0 & \text{DOWNSTRa} & \cdots \\ \vdots & \ddots & \ddots \\ 0 & \cdots & \text{DOWNSTRa} \end{bmatrix} \tag{2.12}$$

$$f = \begin{bmatrix} \text{DOWNSTRb} & 0 & \cdots \\ 0 & \text{DOWNSTRb} & \cdots \\ \vdots & \ddots & \ddots \\ 0 & \cdots & \text{DOWNSTRb} \end{bmatrix} \tag{2.13}$$

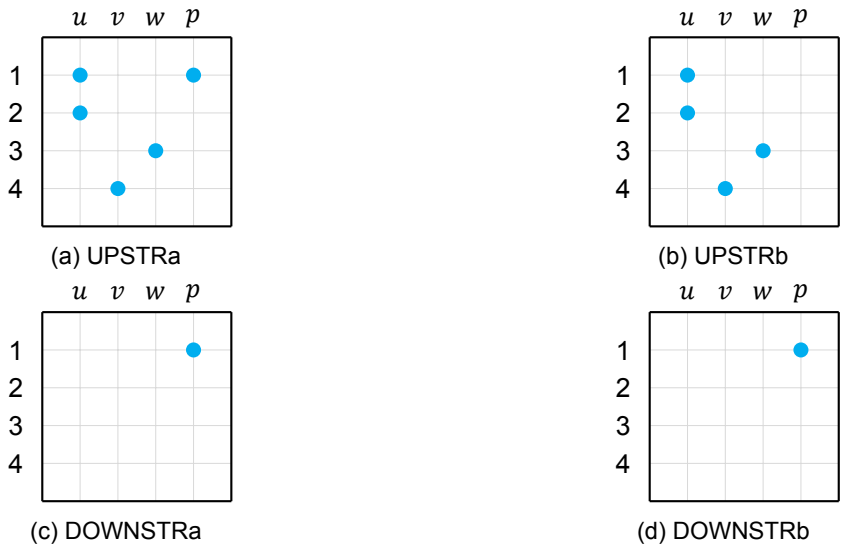
Naturally, all matrices presented in (2.10) to (2.13) are squared and of size $(4 \times NY \times NZ)$. Note that the DOWNSTRa/b blocks do not only contain the coefficients coming from the discretization of the pressure in the momentum equation in mainstream direction, but also contain small contributions of the derivative of the velocity in mainstream direction occurring in the convective terms. This of course only when flow separation occurs. Initially, these convection terms in the reversed flow regions were omitted, or explicitly added. However, it has been observed that adding them implicitly benefits convergence due its stabilizing effects.

It can now be concluded that the coefficient matrix $A$ is formed using the blocks PP, T, Q, R, S, UPSTRa/b and DOWNSTRa/b. Using the discretization stencils of figure 2.3, also the structure of each of these blocks can be understood. This is illustrated below in the figures 2.5 and 2.6.The entire structure of $A$, with all fill-in positions, is now known.




(a) PP (b) T (c) Q (d) R (e) S

Figure 2.5: 4×4 structure of the blocks corresponding to the 5-point stencil, with row 2 representing the continuity equation and rows 1, 3 and 4 the momentum equations in $\xi$-, $\zeta-$ and $\eta$-direction, respectively




(a) UPSTRa (b) UPSTRb

(c) DOWNSTRa (d) DOWNSTRb

Figure 2.6: 4×4 structure of the upstream and downstream blocks, with row 2 representing the continuity equation and rows 1, 3 and 4 the momentum equations in $\xi$-, $\zeta-$ and $\eta$-direction, respectively

# 3

# Iterative methods for sparse linear systems

The previous chapter described how MARIN's RANS solver Parnassos solves the flow around ships in a robust, accurate and efficient way. Most of the computation time is spent on the solution of the system of linear equations, i.e. $Ax = b$, required at all steps of the global iteration loop and for all of the subdomains. Linear systems, arising from the solution of fluid flow problems, are generally characterized by large sparse matrices. Due to the large size of the matrix $A$, direct solving methods are not well suited for this type of problems, as they become too expensive. Krylov methods, on the other hand, are considered to be among the most important iterative techniques available for solving large linear sparse systems, and they happen to be well suited for parallel implementation. Section 3.1 gives a brief overview of the basic principles from which these solvers are built. The most commonly used Krylov methods are described in section 3.2 and 3.3. Furthermore, preconditioning will be covered in section 3.4. Preconditioners are used to increase convergence of the iterative methods. Later on, in chapter 5, focus will be more on preconditioned Krylov solvers suitable for GPU computing.

## 3.1. Krylov subspace methods

Krylov subspace methods are all based on projection processes [48]. The basic principle of a projection process is to extract the approximate solution to a linear system from a subspace $\mathbb{R}^n$. Consider the linear system:

$$Ax = b \tag{3.1}$$

where $A$ is a real $n \times n$ matrix. Let both $\mathcal{K}$ and $\mathcal{L}$ be $m$-dimensional subspaces of $\mathbb{R}^n$, being the subspace of candidate approximates and the subspace of constraints, respectively. The approximate solution $\tilde{x}$ to (3.1) is then found by imposing the conditions that $\tilde{x}$ belongs to $\mathcal{K}$ and that the residual $r$ is orthogonal to $\mathcal{L}$. Using an initial guess $x_0$ this leads to the following:

$$\text{Find } \tilde{x} \in x_0 + \mathcal{K} \text{ , such that } r = b - A\tilde{x} \perp \mathcal{L} \tag{3.2}$$

The subspace $\mathcal{K}$ is represented by the $n \times m$ matrix $V = [v_1, v_2, \dots, v_m]$, whose column vectors form a basis for $\mathcal{K}$. Similarly, the column vectors of the $n \times m$ matrix $W = [w_1, w_2, \dots, w_m]$ form the basis for $\mathcal{L}$. The approximate solution $\tilde{x}$ can be written as:

$$\tilde{x} = x_0 + \delta \text{ , with } \delta \in \mathcal{K} \tag{3.3}$$

or, using the matrix representation for $\mathcal{K}$:

$$\tilde{x} = x_0 + Vy \tag{3.4}$$

Taking the condition imposed on the residual in (3.2), substituting it into (3.4), and using the matrix representation for $\mathcal{L}$, leads to the orthogonality condition:

$$(r_0 - AVy, W^T) = 0 \tag{3.5}$$

Condition (3.5) then results into the system of equations:

$$W^T r_0 = W^T AVy \tag{3.6}$$

which can be rewritten for $y$, and when substituted in (3.4) leads to the following expression for the approximate solution:

$$\tilde{x} = x_0 + V(W^T AV)^1 W^T r_0 \tag{3.7}$$

In deriving equation (3.7) it has been assumed that the matrix $W^T AV$ is nonsingular.

The above demonstrates how an algorithm, representing a projection process in its most general form, consists of 4 steps that need to be executed until convergence is reached: (1) choose $V$ and $W$, the bases for $\mathcal{K}$ and $\mathcal{L}$, respectively; (2) compute the residual $r = b - Ax$; (3) compute $y$ from (3.6); and (4) compute approximate solution $\tilde{x}$ by substituting $y$ into $\tilde{x} = x + Vy$, with $x$ the previous approximation. Different iterative methods then arise from the way the subspaces $\mathcal{K}$ and $\mathcal{L}$ are chosen. All iterative solvers utilizing the Krylov subspace for $\mathcal{K}$ are generally referred to as Krylov subspace methods. Below the definition of a Krylov subspace is given.

**Definition 3.1.** *Let $A \in \mathbb{R}^{n \times n}$ be a linear operator, and $r_0 \in \mathbb{R}^n$ be a vector, then:*

$$\mathcal{K}^m(A, r_0) = span(r_0, Ar_0, A^2 r_0, \dots, A^{m-1} r_0)$$

*is called the $m^{th}$ order Krylov subspace, being the linear subspace spanned by the images of $r_0$ under the first $m-1$ powers of $A$, starting from $A^0 = I$.*

It can be seen that the dimension of the subspace of approximates will increase at every iterative step. An elementary property of Krylov subspaces is that they form the subspace of all vectors in $\mathbb{R}^n$ which can be written as $x = p(A)r_0$, where $p$ is a polynomial of maximum degree $m-1$. Therefore, all Krylov subspace methods lead to the following expression for the $m^{th}$ approximate solution:

$$\tilde{x}_m = x_0 + p_{m-1}(A)r_0 \tag{3.8}$$

Although the above forms the basis of all Krylov methods, various solvers, with significant differences in the characteristics of the iterative techniques they employ, exist. The choice of $\mathcal{L}$, i.e. choosing the constraints used to build the approximations, has a great effect on the overall algorithm. Two major approaches exist for choosing $\mathcal{L}$. One can choose $\mathcal{L} = \mathcal{K}$, as well as $\mathcal{L} = A\mathcal{K}$. These methods are based on an *Arnoldi orthogonalization*. The second approach is to choose $\mathcal{L} = \mathcal{K}(A^T, r_0)$. This results in a class of methods based on a *Lanczos biorthogonalization*. A brief overview of the basic principles of both types of methods is given below. Note that different versions of Krylov subspace methods also arise from the way in which the linear system is preconditioned. Preconditioning will be discussed later in section 3.4.

### 3.1.1. Arnoldi based Krylov methods

Arnoldi based methods [11] compute an approximate solution by building an orthonormal basis for the Krylov subspace. The orthonormal basis is formed by employing the Gram-Schmidt orthogonalization process which transforms a finite, linearly independent set of vectors $S = (v_1, v_2, \dots, v_k)$ into an othonormal set of vectors $S' = (u_1, u_2, \dots, u_k)$, using a projection operator:

$$\mathrm{proj}_u(v) = \frac{\langle u, v\rangle}{\langle u, u\rangle} u \tag{3.9}$$

which projects the vector $v$ orthogonally onto the line spanned by the vector $u$. The Gram-Schmidt process then orthogonalizes every vector towards all previous vectors as follows:

$$v_k = v_k - \sum_{j=1}^{k-1} \mathrm{proj}_{u_j}(v_k) \tag{3.10}$$

followed by:

$$e_k = \frac{u_k}{||u_k||} \tag{3.11}$$

resulting in an orthonormal basis. The basic Arnoldi algorithm is given as follows:

**Algorithm 1** Basic Arnoldi

```
1: Choose a starting vector v_1, such that ||v_1|| = 1
2: for j = 1, 2, ..., m do
3:    for i = 1, 2, ..., j do
4:       h_ij = (A v_j, v_i)
5:       w_j = A v_j − h_ij v_i
6:    end for
7:    h_{j+1,j} = ||w_j||
8:    if h_{j+1,j} = 0 then
9:       stop
10:   end if
11:   v_{j+1} = w_j / h_{j+1,j}
12: end for
```

When applying the above method to the vectors spanning the Krylov subspace, an orthonormal basis is generated for the subspace from which the approximates will be extracted. Implementation of the orthogonalization process influences the way $y$ is computed. Remember from (3.6) that $y$ is computed by:

$$y = (W^T A V)^{-1} W^T r_0 \tag{3.12}$$

For Arnoldi based methods $\mathcal{L}$=$\mathcal{K}$, and therefore $W$=$V$. Furthermore the following relations hold:

$$AV = \bar{V}\bar{H} \tag{3.13}$$

$$V^T A V = H \tag{3.14}$$

where $V$ is a $n \times m$ matrix and $\bar{V}$ a $n \times (m+1)$ matrix, and where $H$ is the $m \times m$ matrix obtained from deleting the last row of the matrix $\bar{H}$, which is the $(m+1) \times m$ Hessenberg matrix whose nonzero entries $h_{ij}$ are defined by the Arnoldi iteration as can be seen from line 7 in algorithm 1. Furthermore, if in algorithm 1, $v_1 = r_0/||r_0||$ and $\beta = ||r_0||$, then:

$$V^T r_0 = V^T(\beta v_1) = \beta e_1 \tag{3.15}$$

Substituting (3.14) and (3.15) into (3.12), and taking into account $W$=$V$, results into the following expression for computing $y$:

$$y = H^{-1}(\beta e_1) \tag{3.16}$$

As before, the approximate solution $\tilde{x}$ is then computed by substituting $y$ into $\tilde{x} = x + Vy$. The algorithm, based on Arnoldi's iteration method, for solving linear systems then looks like:

**Algorithm 2** Arnoldi for Linear Systems

1: Compute $r_0 = b - Ax_0$, $\beta = ||r_0||$ and $v_1 = r_0/\beta$
2: Define the $m \times m$ matrix $H_m = \{h_{ij}\}_{i,j=1,\dots,m}$ and set $H_m = 0$
3: **for** $j = 1, 2, \dots, m$ **do**
4: $w_j = Av_j$
5: **for** $i = 1, \dots, j$ **do**
6: $h_{ij} = (w_j, v_i)$
7: $w_j = w_j - h_{ij}v_i$
8: **end for**
9: $h_{j+1,j} = ||w_j||$
10: **if** $h_{j+1,j} = 0$ **then**
11: $m = j$ and stop
12: **else**
13: $v_{j+1} = w_j/h_{j+1,j}$
14: **end if**
15: **end for**
16: $y_m = H_m^{-1}(\beta e_1)$
17: $x_m = x_0 + V_m y_m$

### 3.1.2. Lanczos based Krylov methods

The Lanczos algorithm [37] differs from the Arnoldi method because it relies on biorthogonal sequences instead of orthogonal ones. Moreover, the subspace $\mathcal{L}$ does not equal $\mathcal{K}$. While $\mathcal{K}$ still equals $\mathcal{K}(A, r_0)$, $\mathcal{L}$ now is the Krylov subspace associated with $A^T$, namely $\mathcal{L} = \mathcal{K}(A^T, r_0)$. The Lanczos algorithm then builds a pair of biorthogonal bases for the two subspaces, namely $\mathcal{K}(A, v_1)$ and $\mathcal{K}(A^T, w_1)$. This is done by the Lanczos biorthogonalization procedure, given by algorithm 3. The vectors $v_i$, formed in the last line of algorithm 3, form the basis of $\mathcal{K}(A, v_1)$ And the vectors $w_j$, build in line 12, form the basis of $\mathcal{K}(A^T, w_1)$. Together they form a biorthogonal system, i.e.:

$$\langle v_i, w_j \rangle = \delta_{ij} \tag{3.17}$$

and therefore the following relation holds:

$$W^T A V = T \tag{3.18}$$

The tridiagonal matrix $T$ represents the oblique projection of $A$ onto $\mathcal{K}(A, v_1)$ and the orthogonal projection onto $\mathcal{K}(A^T, w_1)$. Hence, $T^T$ is the oblique projection of $A^T$ onto $\mathcal{K}(A^T, w_1)$ and the orthogonal projection onto $\mathcal{K}(A, v_1)$. From (3.18) it follows that, using the Lanczos algorithm, $y$ is computed by:

$$y = T^{-1}(\beta e_1) \tag{3.19}$$

which can then be substituted into $\tilde{x} = x + Vy$ to compute the approximate solution. The algorithm, based on Lanczos' method, for solving linear systems is now given by algorithm 4.

**Algorithm 3** Lanczos Biorthogonalization

1: Choose two vectors $v_1$ and $w_1$, such that $(v_1, w_1) = 1$
2: Set $\beta_1 = \delta_1 \equiv 0$ and $w_0 = v_0 \equiv 0$
3: **for** $j = 1, 2, \dots, m$ **do**
4: $\alpha_j = (Av_j, w_j)$
5: $\hat{v}_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$
6: $\hat{w}_{j+1} = A^T w_j - \alpha_j w_j - \delta_j w_{j-1}$
7: $\delta_{j+1} = |(\hat{v}_{j+1}, \hat{w}_{j+1})|^{1/2}$
8: **if** $\delta_{j+1} = 0$ **then**
9: stop
10: **end if**
11: $\beta_{j+1} = (\hat{v}_{j+1}, \hat{w}_{j+1})/\delta_{j+1}$
12: $w_{j+1} = \hat{w}_{j+1}/\beta_{j+1}$
13: $v_{j+1} = \hat{v}_{j+1}/\delta_{j+1}$
14: **end for**

**Algorithm 4** Lanczos for Linear Systems

1: Compute $r_0 = b - Ax_0$ and $\beta = ||r_0||$
2: **for** $j = 1, 2, \dots, m$ **do**
3: Run nonsymmetric Lanczos algorithm 3, generating the Lanczos vectors $v_1, \dots, v_m, w_1, \dots, w_m$ and eventually forming the tridiagonal matrix $T_m$
4: **end for**
5: $y_m = T_m^{-1}(\beta e_1)$
6: $x_m = x_0 + V_m y_m$

The tridiagonal matrix $T$ is formed from the Lanczos vectors $v_i$ and $w_j$, generated using the Lanczos biorthogonalization as given by line 13 and 12. Furthermore it has to be mentioned that Lanczos algorithms may suffer from breakdown. There exist a number of approaches avoiding such breakdowns, as well as techniques that rather deal with it. However, this discussion is out of the scope of this report.

## 3.2. Methods for symmetric positive definite systems

The most well known iterative method for solving large sparse systems characterized by a symmetric (i.e. $A = A^T$) and positive definite (i.e. $x^T Ax > 0$ for $x \neq 0$) system matrix, is the Conjugate Gradient method [31], further referred to as CG. The idea behind CG is to construct an approximate solution $x_j \in \mathcal{K}^j(A, r_0)$ such that the A-norm of the error, i.e.$||x - x_j||_A$, is minimized. This minimization is accomplished by searching for the approximate solution using $n$ conjugate directions, meaning that the direction vectors, along which one searches for the new approximates, are subsequently orthogonal to each other. Let $\{p_0, p_1, \dots, p_n\}$ be $n$ A-orthogonal vectors referred to as the search direction vectors. The initial search direction $p_0$ is chosen to be equal to $r_0$. Each new iterate $x_{j+1}$ is then found by updating $x_j$ with this search direction through a scalar $\alpha_j$. At each iteration step, also the search direction $p_{j+1}$ needs to be updated. This is done through a scalar $\beta_j$. Both coefficients $\alpha$ and $\beta$ are found through orthogonality conditions. The full conjugate gradient algorithm is given in algorithm 5.

Although all problems dealt with in this thesis will be characterized by a non-symmetric matrix, the CG algorithm is given since, from a computational point of view, CG is a very efficient algorithm. This because it is characterized by the following three nice properties:

1. Each new approximation $x_j$ is an element of $\mathcal{K}^j(A, r_0)$, i.e. an element of the **Krylov subspace**.
2. The A-norm of the error is minimized, referred to as the **optimality property**, i.e. at each step the optimal approximation is computed.
3. The method possesses **short recurrences**, meaning that only the results of one foregoing step are needed, therefore the amount of work and memory do not increase at each iteration step.

**Algorithm 5** Conjugate Gradient

1: Set initial solution $x_0$
2: Compute $r_0 = b - Ax_0$ and set $p_0 = r_0$
3: **for** $j = 0, 1, \ldots$, *until convergence* **do**
4: $\alpha_j = (r_j, r_j)/(Ap_j, p_j)$
5: $x_{j+1} = x_j + \alpha_j p_j$
6: $r_{j+1} = r_j - \alpha_j A p_j$
7: $\beta_j = (r_{j+1}, r_{j+1})/(r_j, r_j)$
8: $p_{j+1} = r_{j+1} + \beta_j p_j$
9: **end for**

Also note that, in the unpreconditioned case, only two inner products are needed (and three in the preconditioned case) at each iteration, which makes CG very suitable for parallel implementation due to minimal global communication.

## 3.3. Methods for non-symmetric systems

The majority of problems occurring in computational science and engineering are characterized by a non-symmetric system matrix, as is the case for the fluid flow problems dealt with in this thesis. Unfortunately there does not exist an algorithm as efficient as CG for solving non-symmetric systems. In [24] it was stated that, for non-symmetric systems, it is not possible to compute a new Krylov approximate by satisfying the optimality criterion whilst having short recurrences. Either the algorithm computes an optimal approximation but suffers of long recurrences, or has short recurrences but no optimality. Thus, a large number of different methods exist for solving such systems and many surveys on general iterative solving methods for linear systems can be found in the literature; for example in [48], [28], [17] and [14]. Section 3.3.1 and 3.3.2 will give a brief overview of the most commonly used methods. However, there does not exist a universal ranking of which are the best performing methods, as has been pointed out in [41]. The success of each algorithm strongly depends on the problem it is applied to.

### 3.3.1. Optimal methods

Optimality in CG is obtained through minimization of the A-norm of the error at each iteration. This, however, is only possible when the system matrix $A$ is SPD (i.e. Symmetric Positive Definite). The equivalent for non-symmetric matrices is done through minimization of the residual. At each iteration $j$, the new approximate $x_j \in \mathcal{K}^j(A, r_0)$ is then chosen such that $||r_j||$ is minimal. Optimal methods are therefore built on the following theorem:

**Theorem 3.1.** *The vector $x_j \in \mathcal{K}^j(A, r_0)$ satisfies*

$$x_j = \arg \min_{x \in \mathcal{K}^j(A, r_0)} ||b - A(x_0 + x_j)||$$

*if and only if*

$$r_0 - Az_j \perp A\ \mathcal{K}^j(A, r_0) \leftrightarrow r_j \perp A\mathcal{K}^j(A, r_0)$$

Computing an optimal approximation thus involves orthogonalization over the whole Krylov subspace. This is done using the Arnoldi iteration, described in section 3.1.1. This orthogonalization process, however, makes optimal methods expensive, both in CPU time and in memory requirements, because the amount of work necessary for doing the orthogonalization increases with each iteration, as well as the amount of vectors that need to be stored. Therefore such methods suffer from long recurrences. Several approaches exist for dealing with this, such as the use of restarted or truncated versions of these methods. On the other hand, optimal methods, also referred to as orthogonalization methods, are very robust. Below the most commonly used orthogonalization method is briefly described.

### Generalized minimal residual method

The Generalized Minimal Residual Method [49], further referred to as GMRES, approximates the solution by the vector in a Krylov subspace with minimal residual, and finds this vector using the Arnoldi iteration. Thus, GMRES finds $x_j \in \mathcal{K}^j(A, r_0)$ such that:

$$x_j = \arg \min_{x \in \mathcal{K}^j(A,r_0)} ||b - A(x_0 + x_j)|| = \arg \min_{y \in \mathbb{R}^j} ||b - A(x_0 + V_i y)|| \tag{3.20}$$

Substituting (3.15) and (3.13) into (3.20) results in the following expression for minimizing the residual:

$$\begin{aligned} \arg \min_{y \in \mathbb{R}^j} ||b - A(x_0 + Vy)|| &= \arg \min_{y \in \mathbb{R}^j} ||r_0 - AV_i y|| \\ &= \arg \min_{y \in \mathbb{R}^j} ||\beta v_1 - \bar{V}_{j+1} \bar{H}_j y|| \\ &= \arg \min_{y \in \mathbb{R}^j} ||\bar{V}_{j+1}(\beta e_1 - \bar{H}_j y)|| \end{aligned} \tag{3.21}$$

Since the vectors of $\bar{V}$ are orthonormal, the above can be simplified, resulting in the following expression for computing $y_j$:

$$y_j = \arg \min_{y \in \mathbb{R}^j} ||\beta e_1 - \bar{H}_j y|| \tag{3.22}$$

Once again, the approximate solution $x_j$ can then be computed from $y$. The full GMRES algorithm then looks as follows:

**Algorithm 6** GMRES

1: Set initial solution $x_0$
2: Compute $r_0 = b - Ax_0$, $\beta = ||r_0||$ and $v_1 = r_0/\beta$
3: **for** $j = 1, 2, \ldots, m$ **do**
4: $\quad w_j = Av_j$
5: $\quad$ **for** $i = 1, \ldots, j$ **do**
6: $\quad\quad h_{ij} = (w_j, v_i)$
7: $\quad\quad w_j = w_j - h_{ij} v_i$
8: $\quad$ **end for**
9: $\quad h_{j+1,j} = ||w_j||$
10: $\quad$ **if** $h_{j+1,j} = 0$ **then**
11: $\quad\quad m = j$ and stop
12: $\quad$ **else**
13: $\quad\quad v_{j+1} = w_j / h_{j+1,j}$
14: $\quad$ **end if**
15: **end for**
16: Define $(m+1) \times m$ Hessenberg matrix $\bar{H}_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$
17: $y_m = \arg \min_{y \in \mathbb{R}^j} ||\beta e_1 - \bar{H}_m y||$
18: $x_m = x_0 + V_m y_m$

As mentioned before, these type of methods suffer from long recurrences. Due to orthogonalization the costs of GMRES iterations grow with $\mathcal{O}(m^2)$, with $m$ the iteration number. To reduce costs and memory requirements, a restarted version of GMRES, i.e. GMRES($k$), can be used. The method is then restarted after $k$ iterations and the approximation $x_k$ is then used as the initial guess for the restarted algorithm. By doing so, less vectors need to be stored but this comes at the cost of minimizing the residual only over a small part of the subspace, therefore losing the global optimality property.

## 3.3.2. Methods with short recurrences

Methods having the advantage of short recurrences unfortunately do not satisfy the optimality property. At each iteration $j$, non-optimal methods compute the new Krylov iterate $x_j \in \mathcal{K}^j(A, r_0)$ by constructing a basis for the Krylov subspace by a 3-term biorthogonality relation. This biorthogonalization is done using Lanczos iteration method, described in section 3.1.2. Thus, there where optimal methods ensure

orthogonality of the residuals to the space $A\mathcal{K}^j(A, r_0)$, non-optimal methods establish orthogonality to the space $\mathcal{K}^j(A^T, \tilde{r}_0)$, where $\tilde{r}_0$ is arbitrary but satisfying $\langle r_0, \tilde{r}_0 \rangle \neq 0$. Although these methods, also referred to as biorthogonalization methods, do not compute an optimal approximation, they do posses the important characteristic of having short recurrences. This makes them very efficient, at least at every iteration step. The most popular biorthogonalization method, the Biconjugate Gradient Stabilized Method, i.e. BiCGStab, is described below. BiCGStab is build from earlier developed algorithms, such as the Biconjugate Gradient, i.e. BCG, and the Conjugate Gradient Squared, i.e. CGS, algorithms. Therefore these methods will be briefly addressed at first.

## Biconjugate gradient method

The BiCG method [26] is deduced from the CG method. Remember that CG is based on the Lanczos iteration method. BiCG is then derived from applying the Lanczos algorithm to non-symmetric matrices. Lanczos applied to symmetric matrices, as is done in CG, resulted in the following biorthogonality condition:

$$r_j \perp r_{j-1} \quad \leftrightarrow \quad \langle r_j, r_{j-1} \rangle = 0 \tag{3.23}$$

$$r_j \perp r_{j-2} \quad \leftrightarrow \quad \langle r_j, r_{j-2} \rangle = 0 \tag{3.24}$$

For non-symmetric systems, one needs to choose an arbitrary vector $\tilde{r}_0$, such that $\langle r_0, \tilde{r}_0 \rangle \neq 0$. Doing so, two vector sequences, $r_j \in \mathcal{K}^{j+1}(A, r_0)$ and $\tilde{r}_j \in \mathcal{K}^{j+1}(A^T, \tilde{r}_0)$, are constructed to which the orthogonality condition is now applied. This translates the two biorthogonality conditions (3.23) and (3.24) into the following:

$$\langle r_j, \tilde{r}_{j-1} \rangle = 0 \quad \text{and} \quad \langle r_j, \tilde{r}_{j-2} \rangle = 0 \tag{3.25}$$

$$\langle \tilde{r}_j, r_{j-1} \rangle = 0 \quad \text{and} \quad \langle \tilde{r}_j, r_{j-2} \rangle = 0 \tag{3.26}$$

BiCG is then build analogous to the CG method, resulting in a method computing approximate solutions $x_j \in \mathcal{K}^j(A, r_0)$, through short recurrences, but without satisfying the optimality property. The full BiCG algorithm is displayed below in algorithm 7.

**Algorithm 7** Biconjugate Gradient

1: Set initial solution $x_0$
2: Compute $r_0 = b - Ax_0$ and set $p_0 = r_0$
3: Choose $\tilde{r}_0$, such that $(r_0, \tilde{r}_0) \neq 0$, and set $\tilde{p}_0 = \tilde{r}_0$
4: **for** $j = 0, 1, \ldots$, *until convergence* **do**
5: $\quad \alpha_j = (r_j, \tilde{r}_j)/(Ap_j, \tilde{p}_j)$
6: $\quad x_{j+1} = x_j + \alpha_j p_j$
7: $\quad r_{j+1} = r_j - \alpha_j A p_j$
8: $\quad \tilde{r}_{j+1} = \tilde{r}_j - \alpha_j A^T \tilde{p}_j$
9: $\quad \beta_j = (r_{j+1}, \tilde{r}_{j+1})/(r_j, \tilde{r}_j)$
10: $\quad p_{j+1} = r_{j+1} + \beta_j p_j$
11: $\quad \tilde{p}_{j+1} = \tilde{r}_{j+1} + \beta_j \tilde{p}_j$
12: **end for**

The BiCG algorithm, however, has quite some disadvantages. First of all, the method easily breaks down and is therefore numerically unstable. A second disadvantage is that BiCG requires two matrix-vector products per iteration, there where CG only needs one. This significantly increases the amount of work per iteration. Furthermore, each step of BiCG requires a matrix-vector product with $A^T$ which may be much less efficient than a multiplication with $A$. Also, all vectors generated with $A^T$ do not contribute directly to the solution. These vectors are only used for computing the necessary scalars. All of these inefficiencies in the BiCG algorithm thus gave rise to the development of better variants.

## Conjugate gradient squared method

An attempt to bypass the use of the transpose of A in the BiCG algorithm, lead to the existence of the CGS method [50]. CGS, besides avoiding the use of $A^T$, also gains faster convergence, as compared

to BiCG, for roughly the same computational cost. The derivation of the method is based on the observation that, in the BiCG algorithm, the residual vector $r_j$ and the direction vector $p_j$ are expressed as:

$$r_j = \phi_j(A)r_0 \tag{3.27}$$
$$p_j = \pi_j(A)r_0 \tag{3.28}$$

where $\phi_j$ and $\pi_j$ are polynomials of degree $j$. The vectors $\tilde{r}_j$ and $\tilde{p}_j$ are defined the same as $r_j$ and $p_j$, in which $A$ is replaced by $A^T$:

$$\tilde{r}_j = \phi_j(A^T)\tilde{r}_0 \tag{3.29}$$
$$\tilde{p}_j = \pi_j(A^T)\tilde{r}_0 \tag{3.30}$$

The further derivation of the method then relies on simple algebra only, and is based on squaring the polynomials. This squaring results in a more efficient computation of the inner products. There where BiCG needs $\phi_j(A)r_0$, $\phi_j(A^T)\tilde{r}_0$, $\pi_j(A)r_0$ and $\pi_j(A^T)\tilde{r}_0$, CGS only needs $\phi_j^2(A)r_0$ and $\pi_j^2(A)r_0$ for computing the inner products. This is how the matrix-vector product with $A^T$ is avoided. The final CGS algorithm is displayed below in algorithm 8.

**Algorithm 8** Conjugate Gradient Squared

1: Set initial solution $x_0$
2: Compute $r_0 = b - Ax_0$ and set $p_0 = u_0 = r_0$
3: Choose $\tilde{r}_0$, such that $(r_0, \tilde{r}_0) \neq 0$
4: **for** $j = 0, 1, \ldots$, *until convergence* **do**
5: $\quad \alpha_j = (r_j, \tilde{r}_0)/(Ap_j, \tilde{r}_0)$
6: $\quad q_j = u_j - \alpha_j Ap_j$
7: $\quad x_{j+1} = x_j + \alpha_j(u_j + q_j)$
8: $\quad r_{j+1} = r_j - \alpha_j A(u_j + q_j)$
9: $\quad \beta_j = (r_{j+1}, \tilde{r}_0)/(r_j, \tilde{r}_0)$
10: $\quad u_{j+1} = r_{j+1} + \beta_j q_j$
11: $\quad p_{j+1} = u_{j+1} + \beta_j(q_j + \beta_j p_j)$
12: **end for**

Note that the algorithm still requires two matrix-vector products at each iteration step. Only this time both products use $A$.

### Biconjugate gradient stabilized method

One major problem with the CGS algorithm is that, in case of irregular convergence (which often occurs), the squared terms may lead to a large build-up of rounding errors. To overcome this problem the BiCGStab algorithm [20] was developed. Basically, BiCGStab rewrites the squared polynomials as the product of the polynomial times another polynomial of the same degree, i.e. $\psi_j$. Hence, $\phi_j^2(A)r_0$ and $\pi_j^2(A)r_0$ now become $\psi_j(A)\phi_j(A)r_0$ and $\psi_j(A)\pi_j(A)r_0$, respectively. The goal of this polynomial multiplication is to stabilize the convergence behavior of the original algorithm. Further derivation then again solely relies upon simple algebra. The resulting BiCGStab algorithm is displayed below in algorithm 9. Due to its faster and smoother convergence, BiCGStab is preferred above the less stable BCG and CGS algorithm. Unless a dual system with $A^T$ needs to be solved, then BCG is the preferred method. Note however that BiCGStab is slightly more expensive per iteration due to inner products.

## 3.3.3. Hybrid methods

The most popular Krylov methods are the GMRES and BiCGStab algorithms. Unlike with CG, methods for solving systems with non-symmetric matrices seem to have to make a choice: display the optimality property and suffer from long recurrences or make use of short recurrences by sacrificing optimality. As described in sections 3.3.1 and 3.3.2, GMRES belongs to the first group and BiCGStab to the latter.

**Algorithm 9** Biconjugate Gradient Stabilized

Set initial solution $x_0$
Compute $r_0 = b - Ax_0$ and set $p_0 = r_0$
Choose $\tilde{r}_0$, such that $(r_0, \tilde{r}_0) \neq 0$
**for** $j = 0, 1, \ldots$, *until convergence* **do**
  $\alpha_j = (r_j, \tilde{r}_0)/(Ap_j, \tilde{r}_0)$
  $s_j = r_j - \alpha_j Ap_j$
  $\omega_j = (As_j, s_j)/(As_j, As_j)$
  $x_{j+1} = x_j + \alpha_j p_j + \omega_j s_j$
  $r_{j+1} = s_j - \omega_j As_j$
  $\beta_j = \frac{(r_{j+1}, \tilde{r}_0)}{(r_j, \tilde{r}_j)} \times \frac{\alpha_j}{\omega_j}$
  $p_{j+1} = r_{j+1} + \beta_j(p_j - \omega_j Ap_j)$
**end for**

There do, however, exist hybrid methods that aim at satisfying optimality while also having reasonable short recurrences by combining the properties of different Krylov methods. One such method is the Induced Dimension Reduction Method, further referred to as IDR(*s*). This algorithms seems to be a good candidate for solving the linear systems dealt with in this thesis. Furthermore the algorithm is well suited for parallel implementation. Hence, a brief description will be given below.

## Induced dimension reduction method

IDR(*s*) has been derived from the elderly IDR method, described in [59], which is the predecessor of the CGS method, and later of the BiCGStab method. Development of IDR, just as with CGS, was motivated by the idea of combining the BiCG iteration polynomial with another polynomial such as to avoid the matrix-vector multiplication with $A^T$. However, IDR is fundamentally different from the algorithms previously described in section 3.3.2. BiCG, CGS and BiCGStab all three make use of finite dimensional Krylov subspaces (biorthogonal) for computing the approximate solutions, while IDR, and thus IDR(*s*), generates residuals that are forced to be in a sequence of shrinking subspaces. This happens according to the Induced Dimension Reduction theorem [51]:

**Theorem 3.2.** *Let $\boldsymbol{A}$ be any matrix in $\mathbb{C}^{N\times N}$, let $\boldsymbol{v}_0$ be any nonzero vector in $\mathbb{C}^N$, and let $\mathcal{G}_0$ be the full Krylov space $\mathcal{K}^N(\boldsymbol{A}, \boldsymbol{v}_0)$. Let $\mathcal{S}$ denote any (proper) subspace of $\mathbb{C}^N$ such that $\mathcal{S}$ and $\mathcal{G}_0$ do not share a nontrivial invariant subspace of $\boldsymbol{A}$, and define the sequence $\mathcal{G}_j$, $j = 1, 2, \ldots$ as*

$$\mathcal{G}_j = (\boldsymbol{I} - \omega_j \boldsymbol{A})(\mathcal{G}_{j-1} \cap \mathcal{S}) \,, \tag{3.31}$$

*where the $\omega_j$'s are nonzero scalars. Then*
*(i) $\mathcal{G}_j \subset \mathcal{G}_{j-1}$ for all $\mathcal{G}_{j-1} \neq \{0\}$, $j > 0$.*
*(ii) $\mathcal{G}_j = \{0\}$ for some $j \leq N$.*

The idea is that in the end a zero-dimensional subspace remains, implying that the residual vector equals the zero vector. Note that IDR uses subspaces of dimension 1 whereas IDR(*s*) uses $s$-dimensional subspaces. The full derivation of the algorithm can be found in [51] and the result is given below in algorithm 10. Worth mentioning as well is the IDR(*s*)-variant mentioned in [27], which uses biorthogonality relations for computing the intermediate residuals in each subspace.

Thus, IDR(*s*) is in essence a BiCG-like method, resulting in short recurrences, while making use of shrinking subspaces in order to speedup minimization of the residual, hence displaying reasonable optimality characteristics. This makes it a memory efficient and fast converging algorithm, and therefore it is a true competitor to the BiCGStab and GMRES algorithms.

**Algorithm 10** IDR-based algorithm

**Require:** $A \in \mathbb{C}^{N\times N}$; $x, b \in \mathbb{C}^{N}$; $P \in \mathbb{C}^{N\times s}$; *TOL* $\in (0,1)$;
**Ensure:** *x* such that $||b - Ax|| \leq$ *TOL* $\cdot\, ||b||$

{*Initialisation*}
Calculate $r = b - Ax$
$G = O \in \mathbb{C}^{N\times s}$; $U = O \in \mathbb{C}^{N\times s}$
$M = I \in \mathbb{C}^{s\times s}$; $\omega = 1$
{*Loop over* $\mathcal{G}_j$ *spaces*}
**while** $||r|| >$ *TOL* **do**
  {*Compute s independent vectors* $g_k$ *in* $\mathcal{G}_j$ *space*}
  **for** $k = 1$ to $s$ **do**
    Compute $f = P^H r$
    {*Note:* $M = P^H G$}
    $v = r - Gc$
    $u_k = Uc + \omega v$
    $g_k = Au_k$
    {*Linear combinations of vectors* $\in \mathcal{G}_j$ *are still in* $\mathcal{G}_j$*:*}
    Select $\alpha_i$ and $\beta_i$, $i = 1, \ldots, k-1$
    $g_k = g_k - \sum_{i=1}^{k-1} \alpha_i g_i$; $u_k = u_k - \sum_{i=1}^{k-1} \alpha_i u_i$
    $r = r - \sum_{i=1}^{k} \beta_i g_i$; $x = x + \sum_{i=1}^{k} \beta_i u_i$
    {*Update of the k-th column of* $M$*:*}
    $M_{:,k} = P^H g_k$
    {*Overwrite k-th column of* $G$ *by* $g_k$*, and of* $U$ *by* $u_k$*:*}
    $G_{:,k} = g_k$; $U_{:,k} = u_k$
  **end for**
  {*Entering* $\mathcal{G}_{j+1}$}
  $f = P^H r$
  Solve $c$ from $Mc = f$
  $v = r - Gc$
  $t = Av$
  Select $\omega$
  $x = x + Uc + \omega v$
  {*Alternative computation:* $r = v - \omega t$}
  $r = r - Gc - \omega t$
**end while**

## 3.4. Preconditioning

Krylov methods, especially when applied to large systems arising from CFD problems, generally suffer from slow convergence, or even fail to converge. Preconditioning is a very powerful tool which transforms such an original linear system $Ax = b$ into a system with the same solution, but better convergence properties [48]. This transformation is established using a preconditioning matrix $P$, satisfying the following requirements:

- $P$ should be a good approximation to $A$
- $P$ should be nonsingular
- No excessive cost should be required for constructing the preconditioner
- The preconditioned system should be easier to solve than the original one

The preconditioner can then be implemented in three different ways. First, $P$ can be implemented through left-preconditioning:

$$P^{-1}Ax = P^{-1}b \tag{3.32}$$

This effects both the left- and right-hand side. Right-preconditioning, on the other hand, only affects the operator:

$$AP^{-1}y = b, \quad \text{then} \quad x \equiv P^{-1}y \tag{3.33}$$

Here, one first needs to solve for $y$ after which $x$ can be found. A third option is to apply the preconditioner in the factored form $P = P_L P_R$, with $P_L$ and $P_R$ typically being triangular matrices. This results into two-sided preconditioning:

$$P_L^{-1}AP_R^{-1}y = P_L^{-1}b, \quad \text{then} \quad x \equiv P_R^{-1}y \tag{3.34}$$

Note how the preconditioning matrix itself, $P$, is never explicitly used but only its inverse $P^{-1}$. In fact, $P^{-1}$ will also never be explicitly available since its application is commonly performed in a matrix-free fashion, meaning that instead of storing all matrix entries of $P^{-1}$ and doing a matrix multiplication, its application will be evaluated through matrix-vector products. This is common practice with Krylov solvers since it significantly reduces computational work. To give an example of what such a preconditioned system looks like, the left-preconditioned version of GMRES is given below in algorithm 11.

**Algorithm 11** Left-Preconditioned GMRES

Set initial solution $x_0$
Compute $r_0 = P^{-1}(b - Ax_0)$, $\beta = ||r_0||$ and $v_1 = r_0/\beta$
**for** $j = 1, 2, \ldots, m$ **do**
  $w_j = P^{-1}Av_j$
  **for** $i = 1, \ldots, j$ **do**
    $h_{ij} = (w_j, v_i)$
    $w_j = w_j - h_{ij}v_i$
  **end for**
  $h_{j+1,j} = ||w_j||$
  **if** $h_{j+1,j} = 0$ **then**
    $m = j$ and stop
  **else**
    $v_{j+1} = w_j/h_{j+1,j}$
  **end if**
**end for**
Define $(m+1) \times m$ Hessenberg matrix $\bar{H}_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$
$y_m = \arg\min_{y \in \mathbb{R}^j} ||\beta e_1 - \bar{H}_m y||$
$x_m = x_0 + V_m y_m$

It is important to notice that the algorithm now minimizes the preconditioned residual. This might be quite different from minimizing the original residual and since stopping criteria are commonly based on

the norm of the residual this might have significant consequences for those stopping criteria.

The main idea of implementing a preconditioner is that the matrix $P^{-1}A$ is better conditioned than $A$, such that the preconditioned system converges faster. Preconditioning thus aims at reducing the condition number, $\kappa(P^{-1}A) < \kappa(A)$, to obtain a well-conditioned problem. Well-conditioned problems allow for stable algorithms and are much less sensitive to round-off errors, benefiting the convergence rates. In theory this means that application of $P^{-1}$ should efficiently cluster all eigenvalues of $P^{-1}A$ near 1 as much as possible. Hence, the most efficient preconditioner would be $A$ itself, such that $P^{-1}A = A^{-1}A = I$, resulting in all eigenvalues being 1. Of course, construction costs are too high and do not way up against the reduction in number of iterations. The other extreme is to use $I$ as a preconditioner. While this would be the cheapest to construct, it would not sufficiently improve convergence rates. Finding a good preconditioner is a challenging task and therefore many different techniques exist. There are three main approaches to choosing $P$ [22]:

1. The preconditioning matrix is the result of pure *black box* algebraic techniques.
2. A problem dependent preconditioned matrix can be derived which exploits the features of a certain problem.
3. Another (or even the same) Krylov solver can be used as a preconditioner.

Furthermore, it matters in which way the preconditioner is implemented. The three different approaches, i.e. left, right and two-sided, result in the same eigenvalues for the preconditioned matrices. However, the eigenvectors will be different. Since convergence depends on both eigenvalues and eigenvectors, different ways of implementing $P$ might result in different convergence behavior. Two examples of preconditioners of the first type will be briefly discussed below. The first one, the Jacobi preconditioner, is mentioned because it is probably the most simple preconditioner one can use. Next, the ILU preconditioner is described since this is perhaps the most commonly used and most well know preconditioner out there. Note that preconditioners form the main bottleneck for parallel implementation of Krylov algorithms. Therefore, finding a suitable preconditioner will be one of the main challenges of this thesis. This will be further discussed in chapter 5.

### 3.4.1. Jacobi (or diagonal) preconditioner

The Jacobi preconditioner is merely a diagonal matrix $P$ who's entries are defined as follows:

$$p_{ij} = \begin{cases} a_{ii} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{3.35}$$

with $a_{ii}$ being the diagonal entries of the matrix $A$.

### 3.4.2. Incomplete LU preconditioner

The Incomplete LU factorization is a popular technique for setting up preconditioners, who will be further referred to as ILU preconditioners, and is derived from direct solution methods [22]. Direct solvers are known to be too expensive for solving large systems, but the techniques upon which those methods are based have proven to be very effective. The most common direct procedure is to decompose $A$ as $A = LU$, which sadly results in a matrix with significantly more entries than the sparse original matrix $A$. When using this technique as a basis for preconditioning, the entries will be kept sparse such as to save computing time and memory. This is done through *incomplete* LU factorizations. The preconditioner is therefore built by performing an incomplete factorization of the original matrix $A$ of the form $A = LU - R$, where $L$ is a sparse lower triangular matrix and $U$ a sparse upper triangular matrix. The elements of $L$ and $U$ are determined by taking the matrix product $LU$ and equating it to $A$. However, only the elements of the matrix product there where the matrix $A$ has elements are set equal, i.e. $(LU)_{ij} = a_{ij}$ for all $(i, j)$ where $a_{ij} \neq 0$. Note that $\mathrm{diag}(L) = \mathrm{diag}(U) = \mathrm{diag}(A)$. The elements appearing in the matrix product there where A has no elements (i.e. there where $a_{ij} = 0$) form the defect matrix R. Hence the name incomplete factorization. The matrices $L$ and $U$ are formed by performing a Gaussian elimination and

dropping some of the elements in the predetermined nondiagonal positions, as described in the above. Practically, the matrix $R$ does not need to be build. Once the preconditioner $P = LU$ is obtained, the inverse of the matrix, i.e. $P^{-1}$, can be used to precondition the system $Ax = b$ as explained previously. Note that one never explicitly needs the matrix $P$. For the ILU preconditioner this is done as follows: imagine one wants to compute $z = P^{-1}y$ with $P = LU$. The vector $z$ is then found through $LUz = y$, where one first computes a vector $w$ from $Lw = y$ and then finds $z$ by using $Uz = w$.

**ILU(0): zero fill-in ILU** Generally, it is impossible to exactly match $A$ with the product $LU$. This is due to the appearance of extra diagonals in the matrix product. The entries in these extra diagonals are called fill-in elements. One speaks of zero fill-in ILU, or ILU(0), when all fill-in elements are ignored when building $L$ and $U$ such that their product is equal to $A$ in the other diagonals. This is the standard ILU preconditioner and is rather easy and inexpensive to construct, however, it is not the most efficient Krylov subspace accelerator.

**ILU(*p*): level of fill ILU** To increase convergence rates, alternative incomplete factorizations have been developed by allowing more fill-in of the matrices $L$ and $U$. This is called level of fill ILU, or ILU(*p*). The cost for computing these preconditioners is significantly higher. Hence, a trade-off between increased convergence rates and implementation costs is necessary. There exist two different strategies for allowing the level of fill-in, and both then drop elements during the Gaussian elimination process based on this level of fill of the elements. In the first strategy, the level of fill is attributed to each element depending on their location. This approach is only suitable for structured matrices and therefore not extensible to general sparse matrices. A generalized technique is obtained by the second approach where the focus is on the magnitude of the elements, rather than on their position. This is referred to as ILUT, i.e. threshold ILU.

**ILUT: threshold ILU** Elements will now be dropped based on a threshold value, and the higher the level, the smaller the elements can be. A size of $\epsilon^k$ is assigned to any element $a_{ij}$ whose level of fill is $k$, where $\epsilon < 1$. Initially, a nonzero element has a level of fill of $0$ and a zero element has a level of fill of $\infty$:

$$\mathrm{lev}_{ij} = \begin{cases} 0 & \text{if } a_{ij} \neq 0, \text{ or } i = j \\ \infty & \text{otherwise} \end{cases} \tag{3.36}$$

The level of fill of an element is then updated every time the element $a_{ij}$ gets updated throughout the Gaussian elimination process This level update is calculated according to:

$$\mathrm{lev}_{ij} = \min\{\mathrm{lev}_{ij}, \mathrm{lev}_{ik} + \mathrm{lev}_{kj} + 1\} \tag{3.37}$$

where $\mathrm{lev}_{ij}$ is the current level of fill of the element $a_{ij}$ and $\mathrm{lev}_{ik} + \mathrm{lev}_{kj} + 1$ the *temporary* new level of fill. The new level of fill is then chosen to be the minimum value. Hence, the level of fill of an element can never increase during the process. Thus, if $a_{ij} \neq 0$ in the original matrix $A$ then the element on location $(i, j)$ will always have a level of fill equal to zero.

### 3.4.3. Preconditioning in Parnassos

Parnassos generally uses GMRES for doing the linear solve. A suitable preconditioner then has to be found for the matrix $A$, given by (2.7) in section 2.2 of chapter 2. The preconditioning matrix $P$ is taken as the lower triangular matrix $L$ multiplied by the upper triangular matrix $U$. Both $L$ and $U$ are constructed according to the block structure previously described in section 2.2. Hence:

$$L = \begin{bmatrix} p_1 & 0 & 0 & 0 & 0 & 0 & \cdots \\ c_2 & p_2 & 0 & 0 & 0 & 0 & \cdots \\ b_3 & c_3 & p_3 & 0 & 0 & 0 & \cdots \\ 0 & b_4 & c_4 & p_4 & 0 & 0 & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & \cdots & 0 & 0 & b_g & c_g & p_g \end{bmatrix} \tag{3.38}$$

and:

$$U = \begin{bmatrix} I & p_1^{-1}e_1 & p_1^{-1}f_1 & 0 & 0 & 0 & \cdots \\ 0 & I & p_2^{-1}e_2 & p_2^{-1}f_2 & 0 & 0 & \cdots \\ 0 & 0 & I & p_3^{-1}e_3 & p_3^{-1}f_3 & 0 & \cdots \\ 0 & 0 & 0 & I & p_4^{-1}e_4 & p_4^{-1}f_4 & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & \cdots & 0 & 0 & 0 & 0 & I \end{bmatrix} \tag{3.39}$$

where the blocks $p_i$ are approximations to the $d$-blocks of (2.7). They are constructed through an incomplete LU factorization (constructing $l_i$ and $u_i$) where are fill-in blocks are neglected. Remember that the penta-diagonal structure of the $d$-blocks resulted from the 5-point stencil given in figure 2.4a. The factors $l_i$ and $u_i$ are constructed such that the matrix $l_i + u_i$ has a block sparsity pattern according to the 5-point stencil given in the figure below [53].

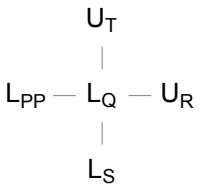



Figure 3.1: 5-point stencil illustrating LU factorization

The construction of all $p_i$-blocks happens independently such that the preconditioner is suited for parallel implementation on a vector machine. Furthermore all $l_i$- and $u_i$-blocks will be characterized once again by the famous 4×4 structure utilized in Parnassos.

## 3.5. Choice of preconditioned Krylov solver

It was already mentioned at the beginning of section 3.3 that it is not possible to make a ranking of iterative solvers according to their performance. There does not exist a universal perfect preconditioned Krylov solver. This is due to several reasons:

1. Mathematical: The mathematical formulation of a solver will only suit well with the discrete formulation of certain problems. Changing for example the Reynolds number or adjusting grid refinement influences the problem formulation such that another solver might become more suited.
2. Stability: Convergence of the solver strongly depends upon the problem.
3. Memory: Due to the long recurrences appearing in some algorithms, the available memory might be a limiting factor.

Regarding the problem to be solved, it is a great advantage to know the exact structure of the coefficient matrix of the system to be solved. The main challenge of this thesis however, is to find a suitable preconditioned Krylov solver which allows for efficient parallel implementation on the GPU. The real difficulty with this is the selection of the preconditioner since preconditioning is an inherently sequential process. Preconditioning techniques suited for GPU computing, and how to efficiently implement them, will be further discussed in chapter 5. For the solver itself, the three major algorithms presented in this chapter, being GMRES, BiCGStab and IDR($s$), are sufficiently fitted for solving the linear systems arising in Parnassos on the GPU.

# 4

# Scientific computing with GPUs

Solving large and complex problems and doing so within a reasonable amount of time requires high computational rates. Processor speed significantly increased over time, however, due to physical limitations (mainly excessive heat dissipation, as well as limits on transmission times of signals, current leakage issues, etc.), it's impossible to keep increasing the speed of serial computers. To overcome this saturation point parallel computing came into play. Simply said, parallel computing is a computation in which many calculations, or the execution of processes, are carried out simultaneously [5]. In other words, it allows for dividing large problems into subproblems such that different processors can work on the different smaller problems. The goal is to complete the large problem in less time than it would take when done in one large chunk. Different forms of parallelism exist, as well as many different types of parallel computers depending on the level to which the hardware allows parallelism. The work done in this thesis deals with the parallel implementation of algorithms on the Graphics Processing Unit, i.e. the GPU. GPU computing is a rather recent way of doing computations in parallel and is rapidly growing in use and popularity due its ability of significantly speeding up computations at a low cost. It came in use about 10 years ago and according to the last publication (November 2016), already 17% of the TOP500 supercomputers are using GPUs [3]. Furthermore, GPUs have a great potential for being used in scientific computing, as will be explained in section 4.1. Section 4.3 till 4.5 will then give a general introduction such that one has a good understanding of GPU computing by the end of the chapter.

## 4.1. Why GPUs for scientific computing?

This question is best answered by having a look at the difference between the GPU and CPU. Remember that GPUs are originally designed to be efficient at manipulating computer graphics and do fast image processing. Graphics rendering is very computational intensive and is a highly parallel computation. Therefore, the GPU is built in such that more transistors are devoted to data processing. This in contrary to the CPU where data caching and flow control are given as much attention. Figure 4.1 illustrates their different build-up, clearly showing the parallel structure of the GPU.

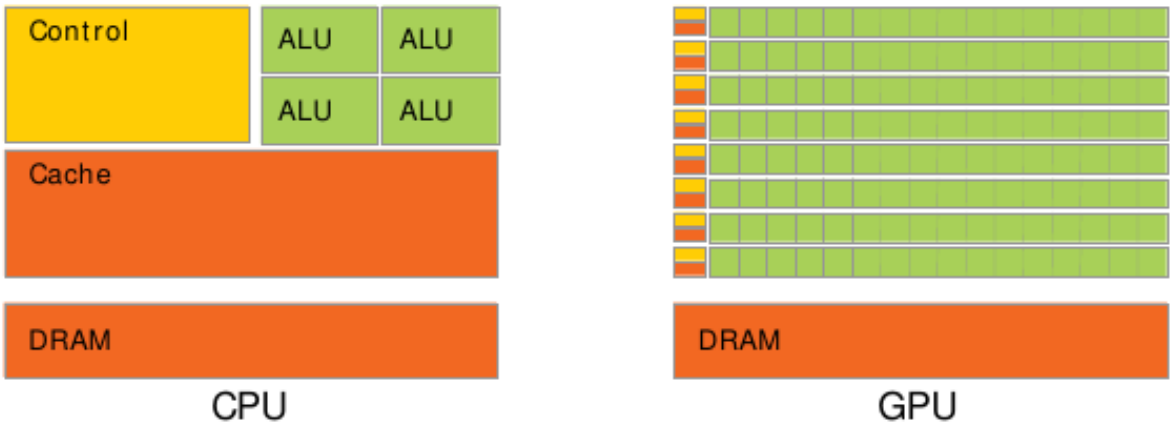



Figure 4.1: CPU vs GPU [43]

Because GPUs employ such a large amount of cores, with the number of cores doubling with each new generation, they posses high floating-point performance. Figure 4.2 shows how GPUs have been exceeding CPUs in floating-point capability since 2003 [35]. The graph also illustrates a slow down in performance improvement of CPUs while GPUs fiercely continue to get better.

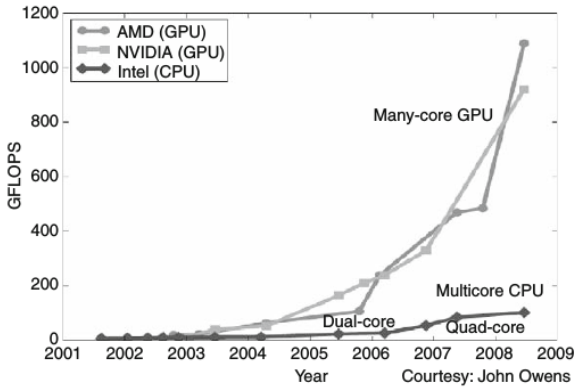



Figure 4.2: Perfromance gap between CPUs and GPUs [35]

The highly parallel structure of GPUs and their good floating-point performance makes them well-suited for problems that can make use of data parallelism. Meaning that GPUs are mainly efficient for handling algorithms where large blocks of data can be processed in parallel. In Flynn's taxonomy , a classification of computer architectures, GPUs are therefore denoted as SIMD machines, i.e. Single Instruction Multiple Data. Furthermore they serve well for problems with high arithmetic intensity, meaning that the number of operations performed should be high with respect to the number of memory operations. This because memory access is the main bottleneck with GPU computing, as will become clear later in this chapter. The linear solve in Parnassos, i.e. solving $Ax = b$ with $A$ very large, is therefore a suitable candidate for GPU implementation. In fact, in 2001, the scientific computing community began experimenting with the new hardware by implementing a matrix multiplication routine, and in 2005, one of the first common scientific programs to run faster on GPUs than CPUs was an implementation of LU factorization [23].

With this army of processors available to tackle the problem, GPUs these days promise up to 1 teraflop (single precision) compute power [57]. More advantages are:

- Readily available and (relatively) cheap. Every computer comes with a GPU card on which one can already program in parallel. Development of scientific GPU cards, such as the ones produced by NVIDIA, lead to better performance but come of course at a higher cost. Yet, they are more economical than workstation clusters. Though be aware that NVIDIA's last generation cards came to the market for a steep increase in price.
- Scalable, with better scalability for large problems than MPI due to faster communication.
- High performance-to-Watt ratio. With a lower power consumption per flop than CPUs, GPUs are also referred to as *green machines*.
- Besides high floating-point performance, GPUs also possess high throughput (the actual amount of data sent per time unit).
- High level libraries available such as BLAS, FFTW, etc.

Naturally they also come with some disadvantages:

- Single precision chips. This because of their graphics background and the accuracy for pixels can be rather low, i.e. $\mathcal{O}(10^{-5})$. It is possible to achieve double precision accuracy, however, this is rather complicated. Double precision GPUs recently became available as well but they are very expensive.
- No error checking through IEEE standards.
- IO not supported. GPUs will therefore always be connected to a CPU and memory transfer between these two causes lots of overhead and should therefore be minimized.
- Limited amount of memory available (2-12 GByte) and latencies can be high. As mentioned before, memory issues will be the main challenge in efficiently implementing algorithms on the GPU.
- A cluster of multiple GPUs (often with OpenMP or MPI) becomes complex.

In what follows, the GPU will be referred to as the *device* and the CPU as the *host*. The remainder of this chapter will mainly focus on some general aspects of the GPU and how to efficiently program on it. Chapter 5 will then elaborate more on the specifics of implementing iterative linear solvers on the GPU.

## 4.2. NVIDIA and the CUDA programming environment

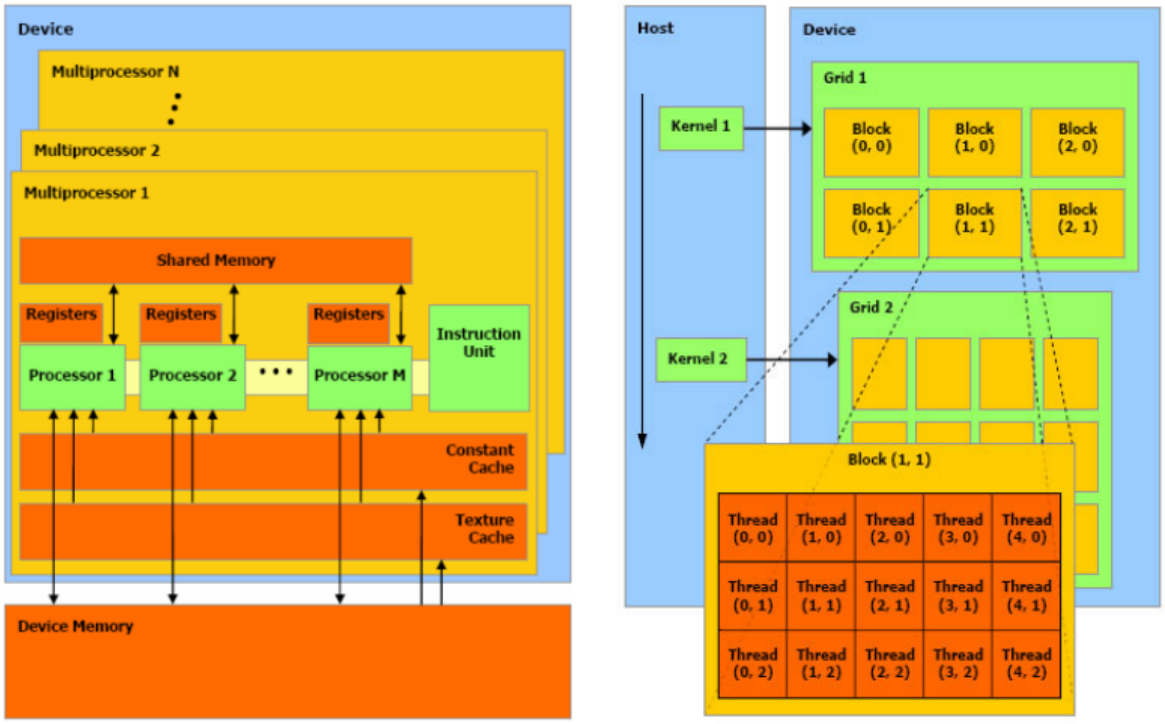
The two main manufacturers and suppliers of GPU cards are Nvidia Corporation, further referred to as NVIDIA, and ATI Technologies, or simply said ATI. In the early days, programming on these cards was very cumbersome. The programmer had to precisely understand how a GPU functioned, it being a rendering environment. Applications therefore had to be modified as if they were rendering operations, basically translating scientific code into pixels. GPU implementation became a lot easier with the advent of API's such as CUDA and OpenCL. These are programming interfaces allowing for parallelization of an application on the GPU without having to get into the details of how to map to the graphics hardware. OpenCL is independent of the hardware vendor and can be used on any platform, while CUDA is specific to Nvidia cards. A detailed comparison of OpenCL and CUDA can be found in [23] and [25].

The work done in this thesis will deal with NVIDIA cards, hence CUDA is a rather obvious choice. It is, however, also a desired choice since CUDA has already evolved a lot, and it keeps on doing so rapidly. Furthermore it has a large user community and it is supported by a great amount of existing

libraries. CUDA stands for Compute Unified Device Architecture and is an extension to the C programming language. It was introduced by NVIDIA in 2006, and really launched the use of GPUs for scientific computing.

## 4.3. Device architecture and programming model

Even when making use of CUDA, programming on the GPU requires some knowledge on the hardware, as well as a good understanding of how instructions are executed on the GPU. The first will be briefly discussed in section 4.3.1 and the latter, referred to as the programming model, will be described in section 4.3.2. In describing the functioning of the GPU it will become clear that memory is an important topic to elaborate on, which will be the subject of section 4.3.3. The figures below will help in describing all these topics.




(a) Layout of a GPU card

(b) Thread configuration

Figure 4.3: Device architecture and programming model

### 4.3.1. Device architecture

Everything a GPU card is equipped with is shown in figure 4.3a. Before describing the different components it is useful to know that every CUDA compatible device is tagged with a 2 digit number, indicating the compute capability (notation: $c.x$). The first number, called the major revision number ($c = 1$ or $2$), marks the core architecture. The minor revision number, i.e. $x$, then indicates small improvements made to the core architecture, available resources and newly added features. When programming on a GPU it is of great importance to know the exact compute capability you have access to.

A NVIDIA GPU card can have 1 or more devices (e.g. Tesla C2070 has 1, Tesla S1070 has 4). Such a device is what is illustrated in figure 4.3a. It contains a number of *streaming multiprocessors* (SM), simply called multiprocessors in the figure. The number of SMs present varies for each device (e.g. Tesla C2070 has 14, Tesla S1070 has 4x30). Each SM on its turn contains several *streaming processors* (SP). These are the green processors in the figure and are the actual ALUs, i.e. Arithmetic and

Local Units, that will carry out the computations (in parallel). This is equivalent to the cores present on a CPU, but while a CPU generally has about 4 cores, a GPU of computing capability $1.x$ has 8 SPs, $2.0$ has 32, $2.x$ has 48, $3.x$ has 192 and $5.x$ has 132. Beside the SPs, SMs have space allocated to different kinds of memory. This will be the topic of section 4.3.3. Furthermore, each SM is equipped with an *instruction unit* (IU). The IU provides instructions to the SPs and schedules the warps. Old architectures, $1.x$, have 1 IU but the newer ones come with 2. One to process the even thread IDs and the other one for the odd thread IDs. What threads and warps are will become clear in the next section.

### 4.3.2. Programming model: execution of threads

This section will explain how a computation can be executed on the GPU. Figure 4.3b will be used for this, as well as the example of a vector addition taken from [43]. A program basically runs on the host (sequentially) and invokes a so-called *kernel* that will be executed on the device. A kernel is a C function that essentially defines the instruction one wants to be executed $N$ times in parallel, with $N$ being the number of threads. A CUDA *thread* is what actually executes the kernel, with each thread having its own data. Remember that the GPU is classified as a SIMD processor, the instruction being the kernel which is executed $N$ times, in parallel, on multiple data through the use of threads. Each thread is given a unique *thread index*. To give an idea of what this actually looks like, the example of a vector addition is given. First the kernel is defined using the `_global_` declaration specifier:

```
_global_ void VecAdd(float* A, float* B, float*C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
```

The kernel is then invoked with $N$ threads. This is done from the main program (running on the host) as follows:

```
int main()
{
...
VecAdd <<< 1, N >>> (A, B, C);
...
}
```

In generic CUDA this invocation looks as follows:

```
kernel <<< dimGrid, dimBlock >>> (...);
```

with `dimGrid`,being the number of blocks per grid, and `dimBlock` the number of threads per block. Note that there is a limit, depending on the device architecture, on the amount of threads that can be launched simultaneously.

Threads can be grouped into one-, two-, or three-dimensional *blocks* and can be identified by a 3-component vector, i.e. `threadIdx={threadIdx.x, threadIdx.y, threadIdx.z}`, the thread index. This provides a natural way to invoke computations across the elements in a domain such as a vector, matrix or volume. The number of threads per block, given by `dimBlock`, is limited because all threads in a block reside on the same SP and must therefore share the limited amount of memory available to that core. This does not however determine the limit of how many times the kernel can be executed in parallel. Equally-shaped thread blocks can be formed, residing on other SPs, and these blocks can be run simultaneously. These blocks are then organized into a *grid*, with `dimGrid` the number of blocks in the grid. Note that the blocks are divided among the various SMs. In principle, the number of blocks per grid is dictated by the amount of data the kernel needs to be executed upon, or by the number of cores available on the device.

An important point in designing thread blocks is that they need to be able to be executed independently such that they can be run in any order; in parallel or in series. This allows for scheduling the blocks in various arrangements and across any number of cores. This is illustrated in figure 4.4 and clearly proves the scalable characteristic of CUDA programs.

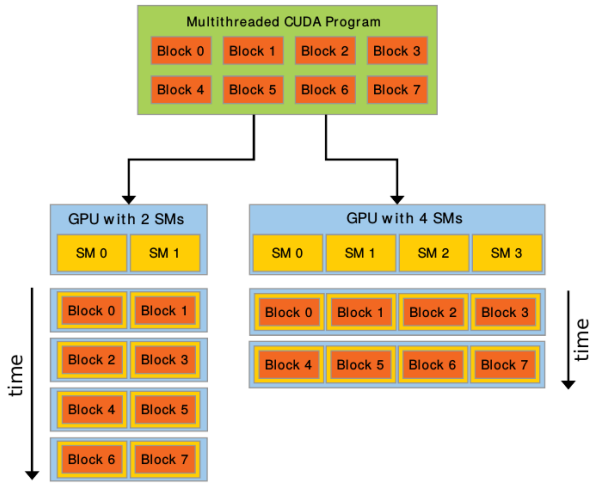



Figure 4.4: Scalability of CUDA programs

It is now known how threads, running the kernel simultaneously on different data, are organized which is basically the execution configuration. Now remains the issue of how the threads are activated for execution. This is done through *warps*. The IU of each SM, also referred to as the warp scheduler, picks up threads for execution and does so in the granularity of a warp. For example, if the warp size is 32, the IU will pick 32 threads with consecutive thread indexes and schedule them for execution in the next cycle. SMs are capable of executing a number of warps simultaneously, depending on the computing capability.

### 4.3.3. Memory hierarchy

GPUs come with various memory types which all come with a different set of rules for access. This memory hierarchy is what makes programming on the GPU a true challenge. It is therefore important to have a good understanding of these different types, their characteristics, and how to use them. Consider once again figure 4.3a. The main memory, in the figure referred to as the device memory, is:

- *Global memory*
  As the name suggests, this is available to all SMs (and to all devices in case of multiple ones) on the GPU, meaning that all SPs can read and write from and to this memory. Global memory is the biggest in size (up to several GB), but it also has the highest access time, i.e. high latency. This is the memory that is the farthest away from the threads where the actual computations happen.

Each SM is then equipped with the following:

- *Constant cache and texture cache*
  These are available to all threads across all blocks on a SM. They are, however, read-only memories.

- *Shared memory*
  It might not be very clear from figure 4.3a but this memory is shared among all threads in a single block. Each block thus has its own shared memory. It is a lot faster to access than the global memory, but they also are a lot smaller. Shared memory is very powerful for efficient implementation of algorithms on the GPU. This will be discussed in more detail in section 4.4.

- *Registers*
  The closest to the actual computation are the registers. These are thread-exclusive memories. Due to their vicinity they are the fastest to access. In turn they are also the smallest, and although they come in large numbers (devices 1.0 and 1.1 have a total of 8192 registers), they tend to be the limiting factor in designing a GPU program.

### 4.3.4. Designing a CUDA program

As mentioned before, it is of great importance to know the device characteristics when designing a CUDA program. Many things are namely decided by the compute capability of your device: threads/warp, max threads/block, max warps/SM, max blocks/SM, max threads/SM, max registers/thread, max registers/block, etc. Taking these limits into account, the programmer is still left with a lot of freedom in designing his/her program. This is best illustrated by using a simple example. Assume you are programming on a device with a maximum of 768 threads per SM, and which has a warp size equal to 32 threads. This leaves us with a maximum of 24 warps (which can be run simultaneously if this does not exceed the max warps/SM limit set to the device). The threads can be configured in many different ways, such as: (1) 256 threads per block $\times$ 3 blocks, (2) 128 threads per block $\times$ 6 blocks, (3) 16 threads per block $\times$ 48 blocks, etc. However, for the $2.x$ and $3.x$ architectures, the maximum number of blocks per SM equals 8 and 16, respectively. Thread configuration (3) is then no longer an option. Furthermore, the availability of memory plays an important role in designing these configurations. Especially the registers do. For example, suppose you have 8192 registers available. Blocks of threads of 16×16 are used and each thread requires the usage of 10 registers. Since $10 * 16 * 16 = 2560$ and $8192/2560 = 3$, a maximum of three blocks can be executed per SM. Notice how increasing the required number of registers per thread by 1, reduces the number of blocks by 1 as well. Remember that it was mentioned before that registers are often the limiting factor in designing CUDA programs.

## 4.4. Strategies for efficient code implementation

Different approaches are known to have a beneficial influence on the performance of GPU programs. This section is written in response to a GPU workshop the author attended, where strategies for efficient code implementation were extensively treated. First, some general guidelines will be given, after which more advanced coding tricks will be introduced. These techniques will merely be listed, foreseen of a small description, as it is not the purpose of this section to be a guide for programming on the GPU. For this, the reader is referred to the NVIDIA CUDA C programming guide [43] and their CUDA C best practices guide [44]. Also the book on programming massively parallel processors [35] is a good reference for this.

In order to carry out a computation on a computer, two main operations are involved: (1) the computing itself (done by functional units), and (2) memory reads/writes. The first is relatively easy and fast. The latter, communication, is what makes it challenging to design an efficient program. A simple model for the time it takes to move $n$ data items from one place to another is:

$$t_{comm} = \alpha + \beta n \tag{4.1}$$

with $\alpha$ being the start-up time, also called latency, and $\beta$ the reciprocal of the bandwidth, usually expressed in (bits/second)$^{-1}$. These values are different for different memory types and generally $\beta << \alpha$. Below follow a few general guidelines that should be taken into account, especially when programming on the GPU with all its different memories:

- Aim for hiding latency as much as possible by overlapping communication with computation. This significantly reduces overall wall-clock time.
- Communication between host and device is the most expensive one; latency is very high and also the actual sending of the data happens rather slowly. Therefore one should minimize data transfer between host and device, even if this means running kernels on the GPU that do not show any speedup compared with running them on the CPU.
- Also copying data inside the GPU is quite time consuming. Know which memory to use and immediately transfer data to the correct place.
- In GPU computing good use of registers is very important since they are the fastest memory available. However, they are very scarce. Be aware to not overload them. Once this happens, data is sent to other memory which significantly slows down the program.

- More threads per block give a better speedup, as well as enough blocks to keep all SMs occupied. It is a general understanding that keeping the GPU full and busy is beneficial to the performance of the program running on it.
- Carefully design the thread configuration. This is related to the above. The goal is to set up the execution configuration such that the GPU's resources are used in the most optimal way. This requires good insight in both the problem and the device architecture.
- Once the data is available, do as much as possible with it. This is another way to overcome the latency issue. It basically states to always maximize arithmetic intensity. In other words, maximize the number of floating point operations (computation) with respect to the number of memory reads.

Using the above guidelines is basically essential to GPU programming. Furthermore, one could choose to make use of some (more advanced) coding strategies. A few will be listed below.

1. *Libraries*
There is no need to reinvent the wheel; use library functions rather than designing your own kernels. When new cards come to the market, these libraries update their algorithms with respect to these new architectures. Once the code is recompiled, users of the libraries can immediately benefit from the resulting increase in performance. The most interesting library for this work will be BLAS. Also the GSL library should be kept in mind.

2. *Page locked memory, or Pinned memory*
A technique to improve transfer of data from host to device and vice versa. The use of page locked, or pinned, memory stores the data in physical memory, instead of using virtual memory, such that the GPU can fetch this data without help of the host. Note that allocating too much pinned memory will eventually slow down the process.

3. *Packing*
Another attempt to improve bandwidth between host and device is by packing non-contiguous regions of memory into a contiguous buffer, sending it all at once, and then unpack the data after the transfer. Bringing together many small transfers into one larger transfer performs better than doing each transfer separately. Packing can be done in combination with page locked memory in order to further improve communication between host and device.

4. *Memory coalescing for global memory access*
One of the most important performance considerations is to ensure that global memory access is coalesced. It basically means that global memory bandwidth is optimized by accessing data items in a consecutive manner. Generally, coalescing happens within a half warp, such that 16 threads, accessing global memory at the same time, do it such that consecutive threads access consecutive memory addresses. Different access patterns exists to achieve this. Note that coalescing depends on the compute capability of the device.

5. *Avoid bank conflicts for shared memory access*
Shared memory is divided into equally sized memory modules, called banks, that can be accessed simultaneously. This allows for high bandwidth, as long as consecutive threads access consecutive banks. Bank conflicts occur when multiple threads request data from the same memory bank. The memory access is then automatically serialized, significantly decreasing bandwidth. Through clever design of the CUDA program, this should be avoided. There is one exception though, when multiple threads in a warp address the same shared memory location, resulting in a broadcast.

6. *Use shared memory*
Shared memory is very fast and is a powerful tool for problems where results need to be used more than once. When running a kernel, one can reserve a desired amount of shared memory. This basically corresponds to manually caching your data. Be aware that there is no point in doing so when data is

only used once. It is most effective when the reuse of results happens within the same block. As with pinned memory, never take more as needed since reserving too much shared memory will eventually slow down the performance.

7. *Pointers*
Memory layout of an array in computer memory differs from the layout we have in mind. Marching through the elements of an array by indexing is a results of how a matrix/vector is displayed in human memory. Indexing refers to the position of an element in the matrix/vector. The use of pointers might be slightly faster. Pointers instead, actually point to where the value is stored in memory through the memory address.

8. *Sum reduction algorithm*
The reduction algorithm for summing elements can be best explained through figure 4.5 as it is pretty self-explanatory. Efficiently summing elements in parallel is crucial as it is also part of computing inner products.

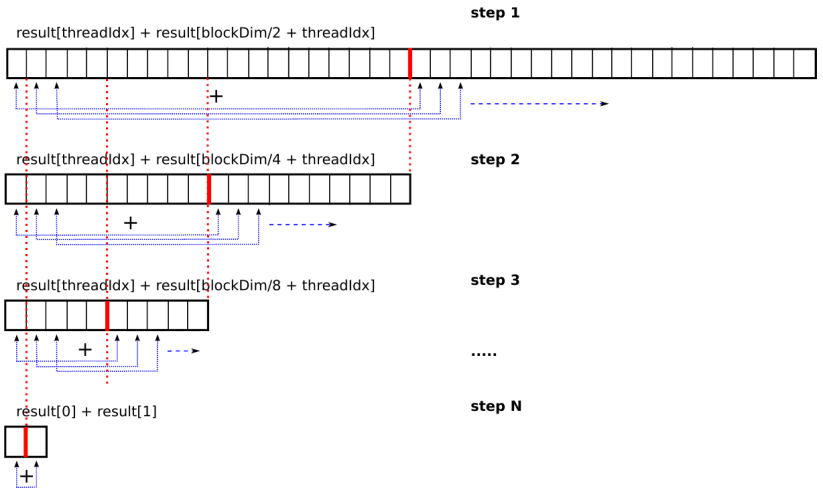
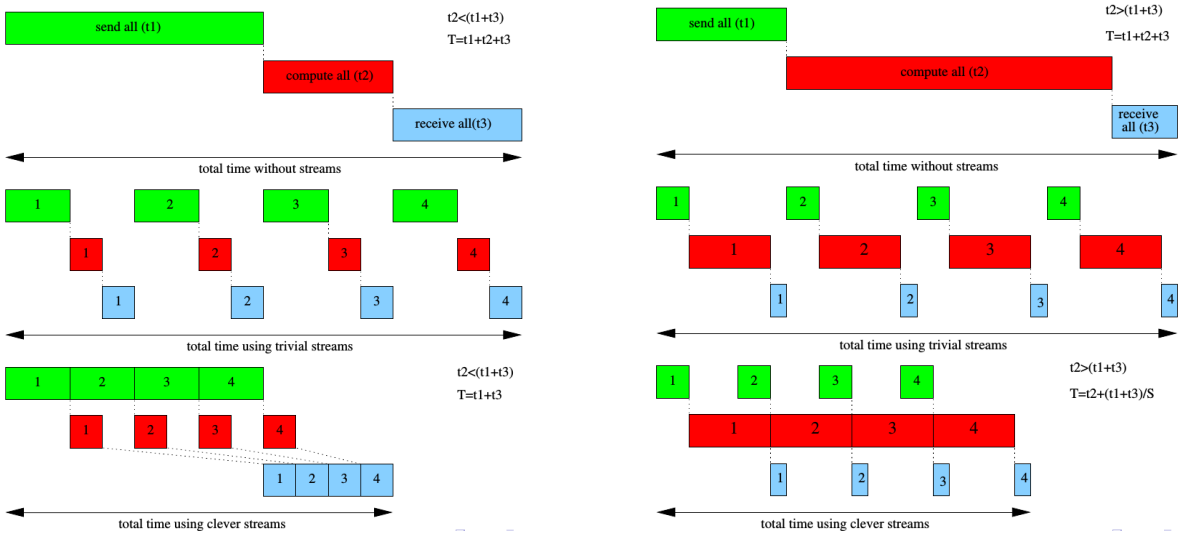



Figure 4.5: Sum reduction algorithm

9. *CUDA streams*
Using CUDA streams is a means of overlapping transfers (between host and device) with computation and is therefore an important tool in optimizing performance. It is also best explained by figure. In figure 4.6a the use of streams is illustrated for memory bound problems, while 4.6b shows the same for compute bound problems. Memory bound problems, as is Parnassos' linear solve, clearly benefit more from using streams.

10. *Atomic operations*
CUDA allows for the use of atomic functions. They perform a read-modify-write operation on one element in memory. For example, the atomic adding function reads a word at some address in memory, adds a number to it, and writes the result back to the same address. This operation is inherently sequential and is guaranteed to be done without the interference from other threads. Hence, no memory conflicts. Atomic operations can be used in global memory and can better be avoided in shared memory.

11. *Unified memory*
A new feature since architecture 3.0 is the use of unified memory. It creates a single virtual memory space for all device and host memories and automatically copies data if accessed from a place where it is not physically located. This makes the code a lot easier, however, it is not clear anymore what is stored where. The use of a profiler can make up for this.




(a) CUDA streams for memory bound problems

(b) CUDA streams for compute bound problems

Figure 4.6: CUDA streams for overlapping communication with computation

12. *Dynamic parallelism*
An even newer feature ,since architecture 5.0, is dynamic parallelism which makes it possible to call a kernel from another kernel on the device and could, for example, allow you to run most of the host code on a GPU thread. This avoids the need to move data between host and device and is therefore a very powerful new future. It does, however, require quite some effort to employ.

13. *Fast math*
CUDA can be compiled with the `-use_fast_math` flag which forces math functions to use the special function unit in each multiprocessor, taking one instruction, whereas the normal implementations can take many instructions. It makes the code run faster at the cost of diminished precision and accuracy.

Many more tricks exist and can be found in NVIDIA's CUDA programming guides, [43] and [44]. Be aware that all the above cannot offer a guarantee to a predefined speedup. The result is strongly dependent on your hardware, card architecture and problem characteristics. A certain approach might be beneficial on a particular device, while it does not have a significant influence on another one. Or, the advantage might only become noticeable once the problem is increased in size. It is therefore recommended to first make an estimation through a simple speedup analysis in order to verify whether it is worth it to implement a certain strategy.

## 4.5. Multi-GPU

Performance can be boosted even further by making use of multiple connected GPUs. In 2010 multi-GPU systems started appearing. In its simplest form these are multi-core systems which can have 4 to 8 GPUs. All devices are then connected to the same host. More complex are the large GPU clusters accommodating thousands of GPUs. These need to be used in combination with OpenMP or OpenMPI, which can become rather complex. When implementing algorithms on multi-GPU systems, communication is once again the biggest bottleneck. It is even more killing to the performance than with single-GPUs. Especially synchronization routines are. A crucial concern therefore becomes the division of data among the different devices. This also greatly affects scalability and will only make certain matrices scale well on multi-GPU systems.

# 5

# Parallel iterative methods on the GPU

Today many different parallel machines are available, offering the opportunity for carrying out fast computations. Yet, to program them such that the performance is in the vicinity of the machine's peak performance is a challenging and tedious task. Efficient parallel algorithms are therefore of uttermost importance because it will determine the feasible range of problem sizes and problem complexity. It has been observed that there is a tendency to modify existing sequential algorithms and thus, that there are not that many *truly parallel* algorithms. This because the existing algorithms are already known and generally are robust and reliable. However, some new algorithms did emerge because it is impossible to make a good parallel program from every sequential one. Also, recall that an algorithm is the result of many design decisions; being robustness, numerical stability, application range, etc. When programming on parallel machines, careful evaluation is required as to how much of those qualities one wants to sacrifice to gain better performance? Nevertheless, always remember that the correctness of the results can never be sacrificed. A lot also depends on the type of problem one wants to solve and the machine this problem needs to be solved on. This chapter will begin by discussing some aspects particular to the GPU implementation of preconditioned Krylov solvers. This will be the subject of section 5.1, after which section 5.2 will introduce some free open-source libraries that take care of the implementation for you. Section 5.3 to 5.5 will then present some promising candidates, and optimization techniques, for handling the linear solve in Parnassos on a GPU. An important concern will also be how to asses parallel performance in an honest way which will be discussed in section 5.6, ending the chapter.

## 5.1. Considerations for algorithm implementation

Before starting to implement an algorithm on a parallel machine, one should verify whether the algorithm is even suitable for this, since not all of them are. It should be possible to extract a sufficient amount of parallelism from it. Of course, implementation considerations highly depend on the type of algorithm, and mainly on the type of machine one wishes to program on. In order to exploit maximum parallelism on the GPU, algorithms should allow for; (1) fine-grained parallelism, and (2) a high flop/byte ratio. An algorithm displays fine-grained parallelism when it is possible to write it such that there will be different units of work that can be carried out independently. Matrix addition is a good example of a computational kernel exhibiting fine-grained parallelism:

$$C[i][j] = A[i][j] + B[i][j] \tag{5.1}$$

with $i$ and $j$ denoting the specific elements of each of the matrices. Each summation can be done individually, without sharing any information, meaning that these different units of work can be run in parallel. In CUDA, the matrix addition is the *kernel* which is executed by a bunch of threads, in parallel, each thread relating to the summation done for the elements at $[i][j]$. The second characteristic, high flop/byte ratio, was already mentioned in chapter 4, stating that high arithmetic intensity helps in overcoming latency issues. A good ratio can be found for the dense matrix-matrix multiplication of two $N \times N$ matrices. Here, the number of memory operations is of the order $\mathcal{O}(N^2)$, i.e. $\mathcal{O}(2N^2)$ reads and $\mathcal{O}(N^2)$ writes. The number of flops is of the order $\mathcal{O}(N^3)$, i.e. $N$ multiplication and $N-1$ summations, and this $N \times N$ times. This results in a ratio $\mathcal{O}(N)$. A worse example is the sparse matrix-vector product. Suppose a sparse matrix with only $5$ elements per row. In order to multiply a row of the matrix by the vector, it requires $11$ memory operations, i.e. reading $5$ matrix elements and $5$ vector elements, and $1$ write. The amount of flops now equals $9$, i.e. $5$ multiplications and $4$ additions, resulting in a flop/byte ratio $(9/11)$ smaller than 1. From this, it becomes clear that GPUs are best suited for dense matrix computations. Due to their indirect and irregular memory accessing, sparse matrices impose extra challenges on GPU computing. The work done in this thesis deals with sparse matrices, however, has the advantage of having to work with structured matrices which is beneficial for GPU computing, making it easier to carry out memory reads and writes, as well as to avoid bank conflicts. Another peculiar observation is that for GPUs, the level-1 (scalar-vector multiplication) and level-2 (matrix-vector multiplication) BLAS routines are rather cumbersome to implement, while level-3 (matrix-matrix multiplication) is fairly easy. Generally, one is used to the opposite.

Krylov methods generally consist out of five different computational kernels:

1. Preconditioner setup,
2. Matrix-vector multiplications,
3. Vector updates or SAXPY operations (e.g. $x = x + \alpha y$),
4. Dot products (e.g. $x^T y$), and
5. Preconditioning operations, i.e. applying the preconditioning matrix $P^{-1}$.

As mentioned before, the computational tasks involving the preconditioner are the most challenging to implement on GPUs since they are inherently sequential operations. Section 5.4 will elaborate more on this and will propose some candidates for investigation on the GPU. The most straightforward to implement are the vector updates since they exhibit the highest degree of fine-grained parallelism, allowing for running all threads simultaneously and without the need for any communication. Note however, that the flop/byte ratio only becomes sufficiently high for large-sized vectors and/or if more computations are carried out (e.g. $x = x + sin(\sqrt{\alpha y})$). Simple vector updates of small-sized vectors are done faster on the CPU. Dot products, i.e. inner products of vectors and vector norms, can be implemented relatively efficiently on the GPU. The multiplication of the different vector elements can be done in parallel. Next, the different outcomes have to be summed. The sum reduction algorithm, explained in section 4.4, is used to do so most efficiently. Eventually remains a synchronization over all processors of the resulting scalar, being the main bottleneck of these computational kernels. The sparse matrix-vector product, as mentioned above, suffers from a low flop/byte ratio, therefore making it the more difficult kernel to

implement in an efficient manner. Sparse matrix-vector products are the central building block of all Krylov solvers since they are needed to generate the Krylov subspace. The GPU implementation of this sparse matrix-vector product, further referred to as SpMV, has therefore been heavily researched. More details on this are given below in section 5.1.1.

It can thus be concluded that plain Krylov solvers, without preconditioning, are well suited for GPU implementation. The two kernels needing most attention when programming a Krylov solver in CUDA are; (1) summing elements, and (2) the SpMV. The first is done using the sum reduction algorithm. The latter is further discussed below.

### 5.1.1. Sparse matrix-vector product

The difficulty with sparse matrices is the arbitrariness with which the nonzero elements are spread over the matrix. Hence the elements are taken from memory in an indirect and irregular manner. This is detrimental to the desire of maximizing memory bandwidth on the GPU. Sparse matrices can be stored in many different formats and choosing the right format is key to getting best locality of data and optimizing the use of memory bandwidth. In 2009, NVIDIA released a study [15], focused on optimizing SpMVs. Herein they discuss different storage formats and their resulting performance for the SpMV. Storage formats discussed are; compressed sparse row (CSR), Co-ordinate (COO), diagonal (DIA), Ellpack (ELL) and a hybrid format (HYB) combining the benefits of ELL and COO. Which storage format to use depends on the structure of the matrix. Here, only the diagonal storage format will be discussed as it is most suited for the matrix characterizing the linear system solve in Parnassos.

The DIA format is best used for the GPU implementation of diagonally structured matrices, where all nonzero elements are confined to lie on a small number of diagonals. The elements of each diagonal are stored in the columns of an array and a second array indicates the offset between the different diagonals. For the matrix given below:

$$\begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 5 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 & 10 \\ 11 & 0 & 0 & 0 & 12 & 0 & 0 \\ 0 & 13 & 0 & 0 & 14 & 15 & 0 \\ 0 & 0 & 16 & 0 & 0 & 17 & 18 \end{pmatrix}$$

this would result in the DIAG array, containing all nonzero elements:

DIAG =

| | | | |
|---|---|---|---|
| * | * | 1 | 2 |
| * | 3 | 4 | 5 |
| * | 6 | 7 | 8 |
| * | * | 9 | 10 |
| 11 | * | 12 | * |
| 13 | 14 | 15 | * |
| 16 | 17 | 18 | * |

and one smaller array containing the offset values for all columns in DIAG:

IOFF =

| | | | |
|---|---|---|---|
| -4 | -1 | 0 | 3 |

The offset diagonals have lesser nonzero elements, because they are shorter or because they may also contain some zero elements. This is indicated by the * symbols in the DIAG array. In practice, those are filled with zeros. In the DIA format, this padding of non-full diagonals is necessary, however, it essentially just wastes storage. Nevertheless, the DIA format results in the more efficient SpMV. Unfortunately it cannot be extended to the use for general sparse matrices. The good news is that

the coefficient matrix $A$, characterizing the linear system solve in Parnassos, has all nonzero elements restricted to a small number (relative to the large size of the matrix) of diagonals (i.e. 27), as was explained in section of 2.3 chapter 2. The matrix can thus be stored in the DIA format.

In CUDA, the SpMV kernel can then be executed simultaneously by a number of threads, each thread working on a separate matrix row. The DIA format allows consecutive threads to access contiguous memory addresses. To further improve the good use of memory bandwidth, it is suggested in [15] to place the vector $x$, of the SpMV $y = Ax$, in the cached texture memory. The vector $x$ only needs to be read in order to carry out the product and its data might be reused later. Caching therefore improves throughput, according to [15], by 30% (single precision).

## 5.2. Libraries

Chapter 4, particularly section 4.4, and the above discussion clearly demonstrate that their are quite some difficulties when it comes to the efficient implementation of Krylov solvers on the GPU. One is required to learn about new programming models and often should acquire knowledge of a new programming language. A good understanding of the hardware is needed as well, since optimization techniques are hardware-specific. It is therefore safe to say that designing a CUDA implementation of a preconditioned Krylov solvers demands a great effort. This led to the development of linear algebra software libraries, offering a large variety of Krylov solvers. The GPU implementation is taken care of by the library itself and can in this way be hidden from the user. Such libraries are easy to use, and generally are portable and extensible. What follows below is only a selection of the large number of libraries available.

### 5.2.1. PARALUTION

PARALUTION is a library allowing the use of various sparse iterative solvers and preconditioners on multi/many-core CPU and GPU devices. Based on C++, it provides a generic and flexible design that allows seamless integration with other scientific software packages [45]. Figure 5.1 illustrates how PARALUTION forms the layer connecting hardware and problem specific packages.

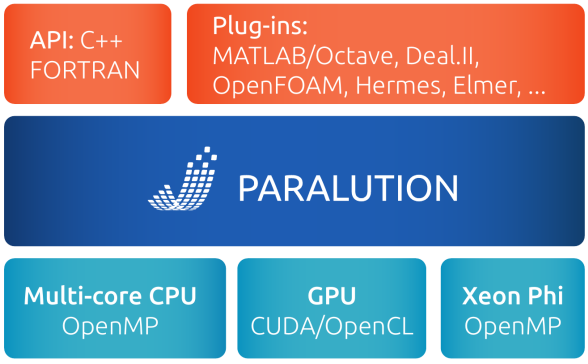



Figure 5.1: PARALUTION as a middle-ware between hardware and problem specific packages [46]

The library is easy in use and does not require any knowledge on the hardware or CUDA. The user can simply select the iterative method suitable for his/her problem and run it on the GPU (or CPU)[1]. The following solvers are included in PARALUTION:

- Standard Krylov subspace solvers: CG, CR, BiCGStab, GMRES, IDR
- Multigrid, both algebraic (AMG) and geometric (GMG)

[1] The library has an internal mechanism to check if a routine can be performed on the device or not. If not, the routine is moved to the host and is performed there. A warning will be printed to inform the user.

- Deflated PCG
- Fixed-point iteration schemes
- Mixed-precision schemes and defect-correction

together with a large collection of fine-grained parallel preconditioners:

- ILU/IC/MDIC, as well as multi-elimination ILU
- MC-GS/SGS/SOR/SSOR and power(q)-pattern ILU
- Block-Jacobi
- CPR
- FSAI/SPAI and Chebyshev
- Saddle-point preconditioners

Furthermore, every solver can be used as preconditioner. Parameters such as stopping criteria, maximum number of iterations, etc., can all be set by the user. PARALUTION also supports the use of different storage formats. This allows the user to experiment and determine which format results in the best performance of the SpMV for his/her specific problem.

### 5.2.2. AmgX

NVIDIA itself also has a library providing a number of iterative linear solvers, named AmgX. It is build upon customizable structures such that different solvers and preconditioners can be easily combined. This also allows for the use of nested solvers. AmgX contains the following [42]:

- Krylov methods: PCG, GMRES, BiCGStab, and flexible variants
- Smoothers: Block-Jacobi, Gauss-Seidel, incomplete LU, Polynomial, dense LU
- Algebraic multigrid

NVIDIA's library is clearly not as extensive as the PARALUTION library, however, it has the advantage of being highly optimized towards NVIDIA GPUs.

### 5.2.3. ViennaCL

The Vienna Computing Library is a free open-source scientific computing library written in C++ and provides CUDA, OpenCL and OpenMP computing backends [4]. The library is primarily focused on common sparse and dense linear algebra operations. It also provides iterative solvers with optional preconditioners for large systems of equations. The following can be found in the library:

- Iterative solvers: CG, mixed-precision CG, GMRES and BiCGStab
- Preconditioners: incomplete Cholesky, ILU, threshold ILU, block ILU, algebraic multigrid, row normalization and Jacobi

### 5.2.4. MAGMA

MAGMA, standing for Matrix Algebra on GPU and Multi-core Architectures, is a collection of next generation linear algebra libraries for heterogeneous architectures [1]. It is designed and implemented by the team that developed LAPACK and ScaLAPACK, incorporating the latest developments in hybrid synchronization- and communication-avoiding algorithms, as well as dynamic runtime systems. The library was originally developed for dense linear algebra routines. It now also contains a variety of solvers and preconditioners for solving sparse linear systems. An overview is given below:

- Sparse solving routines: BiCG, BiCGSTAB, Block-Asynchronous Jacobi, CG, CGS, GMRES, IDR, Iterative refinement, LOBPCG, LSQR, QMR, TFQMR
- Preconditioners: ILU / IC, Jacobi, ParILU, ParILUT, Block Jacobi, ISAI

It also supports:

- Data formats: CSR, ELL, SELL-P, CSR5, HYB

Unfortunately it seems the DIA format, most attractive for implementing Parnassos' linear solve on the GPU, is not available.

For the work needed to be done during this thesis, PARALUTION seems the most attractive choice in case of wanting to use a library. It has a rather large variety of solver options and also contains the DIA storage format. Especially the amount of preconditioners makes it stand out from the other libraries. Furthermore, PARALUTION has a quite large user community and the library is nicely documented.

## 5.3. Krylov solvers

The content of this chapter is focused on finding good candidates for efficient implementation on the GPU. The objective is to speedup computations by improving the time needed per iteration step. Generally, the number one concern when choosing a preconditioned Krylov solver, is its convergence properties. Minimizing the number of iterations might not be the main requirement with GPU implementation. This because the strong point of GPUs is that they are able to carry out these iterations very fast such that they might be faster, even for higher iteration counts. This section discusses plain Krylov solvers and their suitability for GPU implementation. Preconditioning techniques will be addressed in section 5.4. When it comes to basic Krylov solvers most has been previously discussed, i.e in section 5.1. Their main computational kernels were discussed, the SpMV and summing elements of an array being the ones requiring most attention w.r.t. GPU implementation. Overall, Krylov solves are well-suited for GPU computing. The following discussion will therefore be oriented towards the availability of solvers in libraries such as the ones presented above, as well as on recent work done in the field.

**GMRES (restarted)**
The GMRES algorithm is the classical method for solving systems characterized by non-symmetric matrices. There have therefore been many attempts for parallelization of GMRES, on many different platforms. A good reference for a GPU implementation, on a Tesla T10P card, using CUDA, can be found in [58]. In this paper the main focus lies on the SpMV and the implementation of the preconditioning operations. The final result is a speed up of over 20× for sparse linear systems going from a few thousands to a few millions of unknowns. Because GMRES suffers from long recurrences, truncation needs to be applied to it. In [9] it is stated that, in order to obtain smooth convergence, the restart parameter should be larger than 20. With GPU memory being smaller than main memory of the host, the value of this parameter becomes an even more important setting to take into account. The restart parameter should be matched to the characteristics of both the linear system, and the hardware.

**BiCGStab**
A popular solver avoiding the long recurrences is the BiCGStab method. While having the advantage of short recurrences, the algorithm does not possess the optimality property. This means, however, that BiCGStab does not make use of the orthogonalization process which involves a large number of innerproducts, hence requiring several synchronization steps and thus resulting in significant communication overhead on the GPU. In that sense, BiCGStab is a more efficient algorithm, and also the short recurrences is a beneficial property for GPU implementation. This makes the method an interesting candidate for running it on the GPU. Surely, also with BiCGStab, sufficient attention should be paid towards optimizing the SpMV. In [10], it is suggested to further optimize GPU implementation of BiCGStab by reformulating the method in order to reduce data-communication through application-specific kernels. This instead of using generic BLAS kernels. Reformulating is done through two main approaches; (1) gathering similar operations (mainly dot products) in order to design algorithm-specific kernels with higher computational intensity, and (2) merging multiple arithmetic operations into one

kernel such that the number of kernel calls and data transfers are being reduced. Comparing their new implementation towards one build upon cuBLAS functions resulted in performance improvements going from 20% to 90% for a large range of diverse matrices. It has to be noted that in [10] it is strongly suggested that this reformulation in order to reduce communication overhead should be applied to the development of all GPU-accelerated Krylov methods.

**IDR(*s*)**
Minimizing recurrences while aiming for optimality can be found in the IDR(*s*) method. Most of what has been stated before should also be applied when implementing IDR(*s*) on a GPU; optimize the SpMV and try to further accelerate the algorithm by investigating communication reducing approaches. With this method, the dimension of the shadow space, *s*, is an important parameter. Results, w.r.t. both convergence and performance, will highly depend on the size of *s*. Reformulation of the algorithm, according to the same objectives as was done for BiCGStab in [10], has been investigated in [8] where the same authors optimize GPU-accelerated IDR(*s*) through the use of kernel fusion and kernel overlap. The concept of kernel fusion refers to the merging of routines into one kernel in order to improve memory bandwidth. Kernel overlap concerns the concurrent execution of multiple kernels in order to overlap communication with computation. Note that both approaches can be in conflict with one another, hence, a careful interplay of the two optimization techniques should be established. From the results in [8] it was observed that kernel fusion works particularly well for small shadow spaces, while larger ones benefit more from kernel overlap. Combining both succeeded in reducing overall runtime by one third.

The above communication reducing kernels, presented in the work of [10] and [8], are implemented in the linear algebra routines present in the MAGMA library. The same authors of those works also published an experimental study on the efficiency of general Krylov methods on GPUs [9]. In order to conduct this study, the MAGMA library was used. The methods investigated are BiCGStab, CGS, QMR and IDR(*s*) with *s* equal to 2, 4 and 8. The algorithms were tested on a large amount of different matrices and they were evaluated by analyzing both their robustness and efficiency. In terms of execution time, BiCGStab, CGS and QMR were usually more fast for the cases where they converged. When it comes to robustness, IDR(8) was the clear winner, converging in 96% of all test cases. When compared to the respectively fastest solver, IDR(8) was on average less than twice slower. The solver thus demonstrated a good balance between robustness and performance. Convergence properties of all of these solvers are generally enhanced by applying a preconditioner. Parallel implementation of preconditioning techniques is where the real challenge starts.

## 5.4. Parallel preconditioning

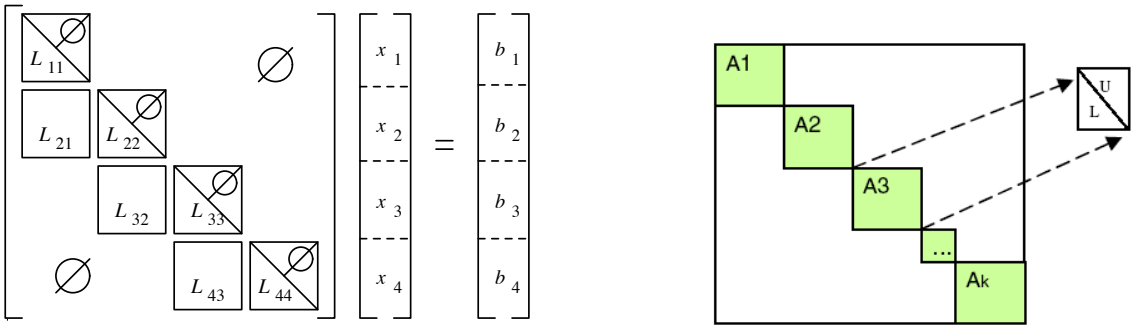
It was stated several times before that the true challenge in obtaining an efficient parallel solver is finding an appropriate preconditioner. Convergence of Krylov solvers is mainly determined by the preconditioning technique applied to it. Unfortunately, preconditioning is an essentially sequential operation because it generally operates on the whole domain. As is clear by now, parallelism does not like global operations as it benefits mostly from data locality. Preconditioners thus require some modification in order to extract parallelism from them. This, however, generally results in degrading numerical properties. In establishing efficient parallel preconditioners, one therefore has to make an essential trade-off between parallelism, which prefers locality, and fast convergence rates, which is stimulated by global dependence [56]. This is the fundamental difficulty of parallel preconditioning and can be illustrated using two of the most well-known preconditioners, ILU and diagonal preconditioning. ILU is one of the most frequently used techniques due to its good numerical properties which result in fast convergence. Unfortunately, due to the triangular backsolves involved, it is highly sequential and not well-suited for parallel implementation. Diagonal preconditioning on the other hand, is the easiest to implement in parallel, but generally does not sufficiently increase convergence rates.

Many different preconditioning techniques exist and various parallel variants have been been developed over the past decades. All these preconditioners come with their own particular computational complexities. Furthermore, their performance is problem-dependent and also varies significantly on

different computing platforms. It is therefore almost impossible to give a complete overview of suitable preconditioning techniques. From literature it also becomes clear that there is no consensus on what is the best preconditioner for GPU implementation, not even for solving systems arising from specific problem types. Here, the advantage of libraries, such as the ones presented in section 5.2, can be seen clearly; with a large collection of preconditioners already available for GPU implementation one can easily play around using different solvers, analyze their performance, and draw conclusion from there onwards. This methodology will be explained further in chapter 6, section 6.3. Hence, below will follow a selection of preconditioning techniques that seem to be good candidates for solving Parnassos' linear system on the GPU. This selection is also motivated by the availability of preconditioners in the above mentioned libraries.

**Jacobi (or diagonal) preconditioner**
The diagonal preconditioner, $P = diag(A)$, a matrix containing the elements of the main diagonal of $A$, has been introduced in section 3.4.1. This preconditioner is highly parallelizable, both in its construction and its application to a vector. The preconditioner is implemented in parallel through the block-Jacobi scheme. It basically is the only preconditioner maintaining its original global character through local parallel implementation. Unfortunately, it is not the most effective one in increasing convergence.

**Incomplete LU preconditioner**
Although not being a good candidate for parallel implementation, ILU deserves some attention on how it can be implemented in parallel due to the fact that it is the most widely used preconditioner for solving non-symmetric linear systems. Incomplete LU preconditioning has been clearly explained in section 3.4.2. Note that ILU is a highly sequential, both in its constructions and application. The LU factorizations are build using the Gaussian elimination process which makes use of the majority of the matrix rows and columns, making it very difficult to partition this process over different threads. This, however, is a minor obstacle since it only has to happen once at the beginning of the algorithm. The major bottleneck are the matrix-vector multiplications with the triangular L and U matrices. These back- and forward-substitutions make use of recurrences and can therefore not be parallelized. This is illustrated by a simple example in figure 5.2a, where it can be clearly seen that $x_2$ can only be computed if $x_1$ is known due to the coupling $L_{21}$ between the blocks $L_{11}$ and $L_{22}$.




(a) Triangular solve with $L$ for $x$, i.e. $L^{-1}b$

(b) Block ILU preconditioner

Figure 5.2: Incomplete LU preconditioning

ILU can be implemented in parallel through the block-Jacobi scheme, as illustrated in figure 5.2b. The L and U factorizations are then constructed, in parallel, for the separate blocks and the triangular solves can also be done simultaneously for the different blocks. This removes the coupling between the blocks, removing the global dependence upon which the preconditioners was originally developed. This allows for parallelization, however, significantly decreases the convergence properties. As the number of blocks increases, the rate of convergence becomes worse. One may also opt for overlapping the Jacobi blocks, i.e. additive Schwarz. This is beneficial to convergence of the solver but comes at a higher communication cost, increasing computation times. Note that these techniques for parallel implementation are applicable to the different variants of the preconditioner; ILU(*p*) and ILUT.

**Sparse approximate inverse preconditioner**
The main idea of this preconditioning technique is to explicitly compute a sparse appoximation to the inverse of the preconditioning matrix, i.e. $P^{-1}$, and use it as a preconditioner. Different techniques exist for this and a comparative study is given in [16]. The approach, suitable for solving non-symmetric systems, referred to as SPAI, is described in [29]. A sparse matrix $M$ is computed by minimizing $||AM - I||$ in the Frobenius norm. Since:

$$||AM - I||_F^2 = \sum_{k=1}^{N} ||Am_k - e_k||_2^2 \tag{5.2}$$

computing $M$ reduces to solving $k$ independent least squares problems. This preconditioning technique is therefore highly parallelizable. Note that the SPAI preconditioner is included in the PARALUTION library, however, the implementation is still experimental.

**Approximate inverse of ILU**
Combining the above, one could attempt to approximate the inverses of $\hat{L}$ and $\hat{U}$, with $A \approx \hat{L}\hat{U}$, by solving the following triangular systems:

$$\hat{L}x_i = e_i \tag{5.3}$$

$$\hat{U}y_i = e_i \tag{5.4}$$

with $1 \geq i \leq N$. Since the linear systems in (5.3) and (5.4) can be solved in parallel, this preconditioning technique is suitable for parallelization. More details can be found in [55].

**Truncated Neumann series preconditioner**
This approach is based on one of the oldest approaches for preconditioning, being polynomial preconditioning. These methods approximate $P^{-1}$ by a polynomial expansion in the matrix $P$, i.e. $p(P)$. For the truncated Neumann series based preconditioner, the inverse of the preconditioning matrix $P = (I + LD^{-1})D(I + (LD^{-1})^T)$ is approximated through the Neumann series expansion as follows:

$$P_{TNS1}^{-1} = (I - D^{-1}L^T)D^{-1}(I - LD^{-1}) \tag{5.5}$$

$$P_{TNS2}^{-1} = (I - D^{-1}L^T + (D^{-1}L^T)^2)D^{-1}(I - LD^{-1} + (LD^{-1})^2) \tag{5.6}$$

where the series has been truncated after 1 term for $P_{TNS1}^{-1}$, and after 2 terms for $P_{TNS2}^{-1}$. All expansion terms in (5.5) and (5.6) contain sparse matrices, hence, applying these preconditioners reduces to a number of SpMVs only. Therefore this preconditioner is well-suited for parallel implementation.

**Other**
Other parallel preconditioning techniques, available in the PARALUTION library, that could be worth testing are multi-elimination ILU [48] and ILU(*p*,*q*) [38].

## 5.5. Further improvements for GPU-accelerated Krylov solvers

In this section a few more techniques will be highlighted which can be used for improving further the parallel performance on the GPU, as well as the convergence properties, of preconditioned Krylov solvers.

### 5.5.1. Mixed precision techniques

Remember that GPUs use single precision floating point operations resulting in fast computations, however, at the cost of less accuracy. The CPU, on the other hand, obtains results of higher accuracy due to double precision floating point operations, but does so more slowly. Mixed precision techniques combine the use of single and double precision floating point operations in order to compute solutions with the speed associated to single precision while obtaining a final result with double precision accuracy. A common technique is to achieve this precision improvement through the use of the defect-correction method, resulting in an inner-outer iteration where the inner loop is done in single precision

and the outer iterations in double precision. The linear system solver in Parnassos already uses defect-correction for computing the approximations. It would therefore be interesting to investigate the linear solve with mixed precision for its GPU implementation.

Mixed precision, applied through defect-correction, has been done in [7] and [12], among others. The defect-correction method computes the solution to the linear system $Ax = b$ through the following iteration scheme:

$$x^{k+1} = x^k + A^{-1}r^k \tag{5.7}$$

meaning the new iterate is found by solving an error correction to the linear system. This error correction, the defect $d^k = A^{-1}r^k$ is based on the residual of the previous approximation, $r_k = b - Ax^k$, hence improving accuracy of the solution. The inner iteration, i.e. the error correction solver, can now be computed on the GPU in single precision. The vector $d^k$ is then sent to the host where the solution update, $x^{k+1}$, is computed in double precision. The host generally also computes the resulting residual, sends it back to the GPU, after which the GPU can start a new inner iteration approximating double precision solution through fast single precision arithmetic. Notice how $A^{-1}$ in equation (5.7) works as a preconditioner of the outer iteration loop. A seperate preconditioned solver can then be chosen for the inner iteration. In [12], results are presented for a non-symmetric linear solve, using mixed precision, where flexible GMRES is used for the outer iteration and GMRES for the inner loop. Mixed precision through defect-correction is also available in the PARALUTION library.

### 5.5.2. Deflation

A technique which can be used to further increase convergence rates is deflation. Essentially, deflation improves the condition number of the system to be solved and can therefore be seen as some sort of preconditioning technique. Generally, it is used in combination with a more generic preconditioner, resulting in a two-level preconditioned Krylov solver. Only a brief introduction will be given in order to get a basic idea of what deflation does. The mathematical description is out of the scope of this literature survey and will only be given once the author decides to actually implement this technique.

Remember that for the linear system $Ax = b$, with $A \in \mathbb{R}^{N\times N}$, the spectrum of $A$ is given by its eigenvalues $\lambda_1, \lambda_2, \cdots, \lambda_N$, its spectral condition number is defined by:

$$\kappa = \frac{\lambda_{max}}{\lambda_{min}} \tag{5.8}$$

and the smaller $\kappa$, the better the convergence. Deflation aims at lowering $\kappa$ by increasing the value of $\lambda_{min}$. Simply said, it does this by *removing* the bad eigenvalues, with bad referring to the eigenvalues with the lowest value. Strictly spoken, those eigenvalues are not removed, but set equal to zero. This done by multiplying $A$, from the left by a matrix $D$ such that the eigenvalue spectrum of $DA$ is given by:

$$\sigma(DA) = \{0, 0, \cdots, 0, \lambda_{k+1}, \lambda_{k+2}, \cdots, \lambda_N\} \tag{5.9}$$

This is the result of wanting to *remove* the first $k$ smallest eigenvalues of $A$ in order to improve its convergence properties. Applying the deflation matrix thus results in having to solve following linear system:

$$DA\tilde{x} = Db \tag{5.10}$$

In combination with generic preconditioning, indicated by the preconditioning matrix $P$, this becomes:

$$P^{-1}DA\tilde{x} = P^{-1}Db \tag{5.11}$$

Note that the matrix $DA$ is singular. Therefore, the system in (5.11) does not have a unique solution, hence indicating the deflated solution vector by $\tilde{x}$ rather than $x$. How to obtain a unique solution $\tilde{x}$ such that $\tilde{x}$ is the solution to the system $Ax = b$, as well as how to carefully choose the deflation vectors making up the deflation matrix $D$, is out of the scope of this report. It has to be mentioned, however, that choosing these vectors might become a difficult task for solving the linear system arising in Parnassos. Remember how Parnassos solves the coupled Navier-Stokes equations and exactly this coupling (of quantities of different orders) will make it hard to define a space where the deflations vectors should be taken from.

### 5.5.3. Multigrid method

Another way of improving convergence rates could be by employing the multigrid method as a preconditioner. Multigrid as a preconditioner to Krylov methods generally adds stability and increases convergence rates. This section will briefly introduce the concepts upon which multigrid is build. Iterative solvers basically compute every new iterate by decreasing the error of the previous approximation. The faster these errors are being decreased, the better convergence. Many iterative solvers suffer from the so-called smoothing property, i.e. high frequency modes of the error are rapidly damped, whereas low frequency modes are damped slowly. This slow decrease of low frequency errors slows down convergence. Multigrid tackles this problem by applying a coarse grid correction step. It has been observed that low frequency errors, when treated on coarser grids, behave as high frequency errors allowing for effectively reducing them. An added advantage of the coarser grid computations is that they are cheaper as the ones done on finer grids. As a standalone method, multigrid computes every new approximate solution by employing a basic iterative method as a smoother applied to a finer grid, and a coarse grid correction, computed on the coarser grid and added to the fine grid solution. As a preconditioner, it means that the preconditioning step $Py = r$, is solved using multigrid.

### 5.5.4. Other

Another interesting alternative for further improving computation rates is the implementation of the iterative solvers on multi-GPU systems. Multi-GPU was briefly addressed in chapter 4, section 4.5.

## 5.6. Analyzing parallel performance

The chapter will end by discussing how to evaluate the performance of solvers when implemented on a GPU. Different performance measures will be presented and some critical notes will be given on how to carry out a fair performance analysis. As indicated by the paper called *Twelve ways to fool the masses when giving performance results on parallel computers* [13], this is a rather difficult and challenging task on itself. Before continuing with presenting measures for evaluation, let's summarize the main features one requires from a good GPU implementation of a preconditioned solver:

- **Efficiency**: reduce execution time per iteration, resulting in overall faster computations
- **Effective**: a robust solver minimizing the number of iterations, resulting in fast convergence
- **Scalable**: increasing computational power should improve solver performance, preferably without having to reformulate algorithm implementation

The first and last are associated to parallel performance of the algorithm, while the second is a general requirement for Krylov solvers. However, when assessing parallel performance, one should continue to verify the algorithm's convergence properties. This will be done by keeping count of the number of iterations, needed till convergence, for the different algorithm implementations. A nice measure for comparing different algorithms (for example different preconditioning techniques) towards each other is by measuring their reduction in number of iterations, relative to a benchmark result.

In order to analyze and evaluate parallel performance, the following measures will be used:

**Wall clock time (or execution time)**
This is the real elapsed times, i.e. as humans perceive it, between the beginning and the completion of a task, and will be further referred to as execution time. One can measure the total execution time of a computation, or just the time spent on certain computational kernels. CUDA also allows for timing certain CUDA kernels. Note that in order to increase timing accuracy of task only taking a few milliseconds, these tasks should be timed multiple times (through a for-loop) and the average should be taken. Generally, in order to compare different solver implementations to one another, times for setting up the preconditioner and the time per iteration are measured. Execution times, given in second or microseconds, are also commonly used for calculating other performance measures.

**Floprate**
The floprate, generally referred to as flops, is the number of floating point operations per second:

$$\text{flops} = \frac{\text{\# floating point operations}}{\text{execution time}}$$

Hence, the higher the faster the algorithm. With GPUs, it is stated for every card architecture what the maximum amount of flops is the device can deliver. Therefore it is interesting to see how much of this theoretical peak is achieved by your algorithm. One should be aware, however, that reaching 10% of the peak is already a satisfying outcome.

**Throughput**
This measure is particularly interesting for assessing parallel performance on GPUs and should not be confused with bandwidth. Throughput is the memory rate for a specific kernel. In other words, it is the actual amount of bits that are being transferred per time unit. It is therefore dependent on the algorithm, while bandwidth is the theoretical memory rate and depends on the hardware. Throughput for a specific kernel is computed as follows:

$$\text{throughput} = \frac{b_{read} + b_{write}}{10^9 \cdot \text{execution time kernel}}$$

with $b_{read}$ and $b_{write}$ the number of bytes read from and written to memory during execution of the kernel. The results are generally of the order $10^9$ bytes per second, hence $10^9$ appears in the denominator such that the final outcome is given in GB/s. Since maximum parallelism is exploited on the GPU through high arithmetic intensity, one wants algorithm implementations displaying large throughput.

**Speedup and the notion of scalability**
Speedup, $S_p$, is a measure for quantifying the relative improvement in performance from the sequential program to the parallel one. The fairest comparison is done when it is defined as the ratio of the serial execution time obtained by the best sequential algorithm, $t_1$, to the parallel execution time, on $p$ processors, using the best parallel algorithm:

$$S_p = \frac{t_1}{t_p} \tag{5.12}$$

An alternative approach, if one wishes to avoid using serial execution times, is to employ a certain number of processors $n$ as a benchmark for computing relative speedup, $S_p^{'}$:

$$S_p^{'} = \frac{t_n}{t_p} \tag{5.13}$$

for which $n < p$. Ideally, $S_p$ equals $n$ (and $S_p^{'} = (n/p)^{-1}$), however, in practice this value will be lower.

Besides good speedup, algorithms should be scalable. As pointed out in [32], a paper addressing the question *What is scalability?*, there does not exist a generally-accepted definition, but only an intuitive notion of this term. Intuitively, scalability is typically related to speedup in the sense that good speedup indicates good scalability of a system. For the purpose of this work, scalability of a parallel system will be defined as done in [36], being a measure of its capacity to increase speedup in proportion to the number of processors, reflecting the ability of the system to efficiently utilize increasing processing resources, without the need for redesigning it. Two different approaches exist to carry out a scalability analysis; (1) strong scaling: the system is considered scalable if the speedup on $n$ processors is close to $n$, an analysis done by gradually increasing the number of processors for a fixed problem, and (2) weak scaling: the system is said to be scalable if, when the number of processors and the problem size are increased by a factor $n$, i.e. maintaining a fixed amount of work per processor, the execution time remains the same.

Parallel implementation of an algorithm on a parallel architecture does not automatically imply high performance. Several limitations exist to parallel computing. A first, theoretical, limitation is given by

Amdahl in his work [6], where he defines an upper bound to speedup. Amdahl's law states that the speedup and efficiency of a parallel system is limited by the time needed for the sequential fraction of the program. Assume an algorithm where a fraction $f$ can be carried out in parallel, the speedup on $p$ processors, is then given by:

$$S_p = \frac{t_1}{t_p} = \frac{t_1}{f\frac{t_1}{p} + (1-f)t_1} = \frac{1}{\frac{f}{p} + (1-f)} \leq \frac{1}{1-f} \tag{5.14}$$

Considering most parallel applications have a significant serial part in their computations, Amdahl's law indicates his pessimistic view on parallel computing as it clearly implies that efficiency will necessarily drop with increasing number of processors. Note, however, that Amdahl's law assumes away things such as memory limitations, cache effects, etc, therefore implicitly assuming that the sequential fraction is independent of all other computational parameters. This is not the case since one such parameter which has significant influence is problem size, as was first indicated by Gustafson [30], providing a counterpoint to Amdahl's law. Gustafson showed how the sequential fraction decreases with increasing problem size, implying that parallel computing is suitable for solving sufficiently larger problems as increasing problem size increases efficiency. Still, Amdahl's law is important to keep in mind in order to maintain realistic expectations towards efficiency improvements that can be made through parallel implementation of an algorithm. Suppose 90% of an algorithm can be done in parallel, i.e. $f = 0.9$. According to (5.14), the speedup that can be achieved w.r.t. the sequential algorithm is maximum $S_p = 10$. Reality is not as pessimistic as Amdahl stated, however, parallel performance will always be negatively influence by the fact that part of the algorithm needs to be done sequentially.

**Profiling**

To yield a better understanding in what may cause an increase or decrease in the above mentioned performance measures, one can use profiling. It a means of analyzing the program by obtaining info on memory distribution, communication patterns, etc. Using this info, one should be able to identify major bottlenecks in the algorithm. Therefore, profiling information usually allows for program optimization. It has to be noted, however, that profiling comes at a certain cost. Measuring performance data itself needs time and thus may lower the overall performance, i.e. intrusion overhead. Furthermore, those measurements cause a perturbation which may alter the program behavior. Therefore, a careful trade-off needs to be made between accuracy and expressiveness of data. Simple profiling can be done manually by timing separate CUDA events, or more info can be obtained by, for example, making use of NVIDIA's profiler, named nvvp.

# 6

# Research question, experimental setup and planning

This chapter concludes the literature survey, conducted as the first step of the MSc thesis. The survey touched upon three different subjects; (1) Parnassos, the RANS solver for which the performance of the linear solve should be improved, (2) a general overview was given of preconditioned iterative methods suitable for solving large sparse linear systems, such as the ones occurring in Parnassos, and (3) GPUs, and their suitability for use in scientific computing, were introduced. The survey then continued by bringing together the last two subjects; how to implement the aforementioned iterative algorithms on GPUs in order to improve performance. Now, the research question can be formulated, the experimental setup can be decided and a further research plan can be designed. These will be the topics discussed in what follows.

## 6.1. Research question(s)

The main aim of this work will be to give an answer to the following question:

*How to / Is it possible to achieve reasonable speedup of Parnassos' linear system solver by making use of GPU computing?*

In answering this question, the following subquestions will have to be addressed as well:

- Which preconditioned Krylov solver to use? Here, the main question is which preconditioning technique should be selected? More specific, which preconditioner satisfies the following three requirements; (1) has good convergence properties, (2) is suitable for efficient GPU implementation, and (3) gives the solver good scalability?
- What are the strategies for a fast GPU implementation?
- Can the CUDA program be optimized further? How?

Furthermore, one should also wonder about:

- What will be the eventual speedup for Parnassos as a whole?

And if time allows, also the following can be investigated:

- How to reach speedup when using multi-GPU?

## 6.2. Experimental setup

The above questions will be investigated using the problems presented in section 6.2.1. The test environment will be summarized in section 6.2.2.

### 6.2.1. Test problems

The aim of this work is to investigate efficient implementations of preconditioned Krylov solvers in order to speedup the linear system solver in Parnassos. Test problems will therefore be taken directly from Parnassos. Below follows a brief summary of the matrices characterizing the linear systems needed to be solved.

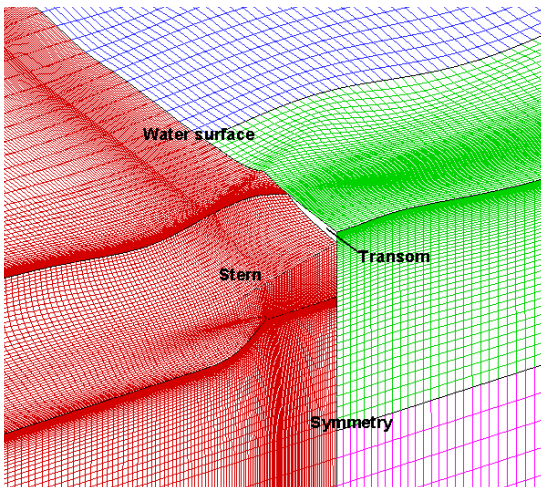



Figure 6.1: Detail of the block-structured grid, employed in Parnassos, near the stern [54]

Parnassos uses a special block topology, dividing the overall domain in 4 blocks with different grid structures. In this way it is possible to handle computations on both wetted and dry transoms, as well as on transoms that are partly dry and partly wetted [54]. A detail of this block structured grid near the stern is given in figure 6.1. The red-colored block will be referred to as domain 111, and is the main block, upstream of the transom containing the complete ship hull and a significant part of the flow upstream of the bow. This block consists of 321 × 201 × 53 cells in mainstream, normal, and girthwise direction respectively. The green, purple and blue blocks behind the transom will be further referred to as domains 211, 212 and 221, respectively. Their number of cells in mainstream, normal and girthwise direction are; 209 × 51 × 71 for domain 211, 209 × 81 × 51 for domain 212 and 209 × 81 × 25 for domain 221. The total grid thus consists of approximately 5.4 million cells. Remember from chapter 2 that Parnassos makes use of an outer and inner iteration for computing the flow around the ship hull. The outer iterations sweep through the overall domain in downstream direction. As an initial solution for the test problems, the solution computed after 1344 sweeps is taken. It can then be decided how many sweeps to compute, starting from this initial solution. For initial testing purposes, one sweep should be sufficient for identifying the better preconditioning options. During one sweep, the solution is then computed on the different subdomains. These subdomains consist out of a number, $g$, of $\xi$=constant planes. These are the inner iterations, where the linear system $Ax = b$ needs to be solved.

The matrix $A$, resulting on the subdomains of the different blocks, is characterized by the following properties:

- Large and sparse
- Squared
- Non-symmetric
- Diagonally structured

Refer back to chapter 2, section 2.3, for a thorough explanation of the structure of the matrix. The size of $A$ and the number of nonzero elements depend on the amount of planes taken per subdomain. Remember from chapter 2 that the size is given by $(4 \times g \times NY \times NZ) \times (4 \times g \times NY \times NZ)$. The parameter $g$, i.e. the number of planes, can be easily changed in Parnassos. Table 6.1 summarizes some numbers for varying number of planes per subdomain, and for subdomains appearing in the different blocks of the grid. The size of $A$ is given by $n$, hence $A \in \mathbb{R}^{n \times n}$. Furthermore, the table lists an approximation to the number of nonzero elements, $nnz$, determined approximately using the details given on the matrix structure in section 2.3.

Table 6.1: Size of matrix $A$ for different sizes of the subdomain and for the different blocks appearing in the grid

| | | $g$ =1 | $g$ =4 | $g$ =16 | $g = NX$ |
|---|---|---|---|---|---|
| domain 111 | $n$ | 42,612 | 170,448 | 681,792 | 13,678,452 |
| | $nnz$ | 371,585 | 1,788,434 | 7,668,890 | 157,130,480 |
| domain 211 | $n$ | 14,484 | 57,936 | 231,744 | 3,027,156 |
| | $nnz$ | 126,125 | 607,718 | 2,606,510 | 34,753,748 |
| domain 212 | $n$ | 16,524 | 66,096 | 264,384 | 3,453,516 |
| | $nnz$ | 143,925 | 693,348 | 3,335,460 | 39,648,678 |
| domain 221 | $n$ | 8,100 | 32,400 | 129,600 | 1,692,900 |
| | $nnz$ | 70,345 | 339,670 | 1,457,470 | 19,435,420 |

The above table clearly indicates that the linear solve is done for very big matrices with a large sparsity pattern. These matrices are stored by means of direct addressing, as was also explained in chapter 2. One can investigate the performance of different solvers and preconditioning techniques, for subdomains of varying sizes. Different metrics can be measured using Parnassos to get an indication of the number of iterations needed for convergence and the time it takes.

### 6.2.2. Test environment

Computations will be carried out on MARIN's cluster named Marclus3. It consists of the following hardware:

- 2 head nodes: used for cluster management
- 2 login nodes: used for user logins, interactive jobs
- 1 login nodes: used for external users (not available from inside MARIN)
- 5 preprocessing nodes: used for jobs with large memory requirements
- 204 compute nodes: used for batch nodes
- Gigabit ethernet connections
- High speed DDR/QDR Infiniband network for MPI applications

Small computations can be ran sequentially on a login node, consisting of:

- 2 Intel E5-2617 (2.9GHz) Hex core CPUs, 256 GB RAM memory, 280 GB local disk, Gigabit ethernet, QDR Infiniband HBA, nVidia Quadro 2200 GPU

Larger problems are executed on a compute node, consisting of:

- 2 Intel E5530 (2.4GHz) Quad core CPUs, 12 GB RAM memory, 320 GB local disk, Gigabit ethernet, DDR Infiniband HBA

Furthermore, there are heavy nodes consisting of:

- 4 Intel E5-2617 (2.9GHz) Hex core CPUs, 256 GB RAM memory, 280 GB local disk, Gigabit ethernet, QDR Infiniband HBA, nVidia Quadro 2200 GPU

GPU computations will be done on a NVIDIA Quadro K2200 card, which has the following characteristics:

- 640 CUDA cores
- 4GB RAM
- 80 GB/s memory bandwidth
- computing capability 5.0 (Maxwell based)

## 6.3. Planning

A brief overview will be given of the methodology that will be used in order to find an answer to the questions posed in section 6.1.

**Benchmark results**
A first step should be the generation of benchmark results. For this the solving strategy currently used at MARIN should be used. Parnassos is run sequentially at the moment and there is a choice for using GMRES or IDR($s$) as a solver, and ILU or Jacobi as a preconditioner. The test problem of section 6.2.1 should be employed and the different solver options can be tested for varying sizes of the subdomains. Initially this can be done without doing multiple outer iterations. The number of iterations needed for convergence and the execution times should be measured. Also the times for specific routines can be measured separately. Analyzing these results, the best performing sequential algorithm can be determined. Later, when analyzing the performance of parallel implementations, this can be done w.r.t. to these benchmark results, especially w.r.t. to the best performing sequential solver.

**Performance tuning cycle**
From the previous chapters is became clear that optimizing performance of parallel systems is a challenging task; it requires knowledge on both the algorithm and features of the parallel architecture you want to implement it on. The search for an efficient preconditioner for solving Parnassos' linear solve on the GPU will be done following the performance tuning cycle procedure, presented in [21], consisting out of the following steps:

1. Instrumentation: decide on what info to gather and how to measure it. One may opt to directly insert measurement probes in the application code and/or make use of certain profiling software.
2. Execution of the instrumented application and performance data.
3. Analysis of the captured performance data: identify possible performance impediments and from there decide what can be optimized.
4. Apply modifications and rerun with the aim of obtaining improved/optimized performance.

**Comparative study: initial iteration**
This refers to step 1 and 2 of the above mentioned performance tuning cycle; test the performance of different iterative solvers on the GPU and compare it towards each other and w.r.t. to the best sequential performance obtained. Some choices have to be made for doing so:

- *Self-implementation or library?*
  There has been chosen for making use of the libraries presented in section 5.2. In first instance, the PARALUTION library will be used. This choice resulted from two main observations made during the literature survey; (1) as explained through chapter 4 it is not too difficult to make an algorithm run on a GPU, however, it requires a lot of effort and time to make it efficient, and (2) from literature, of which a summary is presented in chapter 5, it became clear that there is no consensus on what is a good parallel preconditioner. If a lot of parallelism can be extracted from it, it generally has worse convergence properties, and vice versa. Performance is also highly problem-dependent. It is therefore also not that clear to decide which preconditioner to spend all your time on implementing. Here, libraries come in handy. Due to the significant amount of solvers and preconditioners already implemented , it allows for easily playing around with different solvers, and comparing their performances.
- *Solvers and preconditioners to test?*
  The most promising candidates were summarized in chapter 5; the solvers were presented in section 5.3 and different preconditioning techniques in 5.4.
- *Performance measures to measure and how?*
  Section 5.6 of chapter 5 gave a good overview on what is needed in order to carry out a descent performance analysis. In an early stage of the comparative study, simple measurement can be taken by inserting timers directly in the code to measure execution times for specific routines, as well as counters for keeping track of the required number of iterations for convergence. In a later stage, profilers can be used to better analyze and understand the results.

**Comparative study: further iterations**
Analyzing the results from the first comparative study, it can be decided which preconditioned solvers to investigate further. Through more testing of different setting, further performance improvements should try to be made.

**Conclusions**
It is expected that a best performing algorithm can be identified and implemented for use by the MARIN. The next step could then be to make an own implementation of this solver, further improving its performance.

# Bibliography


[1] MAGMA version 2.0.

[2] MARIN website.

[3] Highlights - November 2016 | TOP500 Supercomputer Sites.

[4] ViennaCL - Linear Algebra Library using CUDA, OpenCL, and OpenMP.

[5] GS Almasi and A Gottlieb. Highly parallel computing. 1988.

[6] GM Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, spring joint*, 1967.

[7] H Antz, V Heuveline, and B Rocker. An error correction solver for linear systems: evaluation of mixed precision implementations. 2010.

[8] H Anzt, E Ponce, GD Peterson, and J Dongarra. GPU-accelerated co-design of induced dimension reduction: algorithmic fusion and kernel overlap. *Proceedings of the 2nd*, 2015.

[9] H Anzt, J Dongarra, and M Kreutzer. Efficiency of General Krylov Methods on GPUs–An Experimental Study. *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, 2016.

[10] Hartwig Anzt, Stanimire Tomov, Piotr Luszczek, William Sawyer, and Jack Dongarra. Acceleration of GPU-based Krylov solvers via data transfer reduction. *The International Journal of High Performance Computing Applications*, 29(3):366–383, aug 2015.

[11] W.E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 1951.

[12] M Baboulin, A Buttari, J Dongarra, and J Kurzak. Accelerating scientific computations with mixed precision algorithms. *Computer Physics*, 2009.

[13] David H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, pages 54–55, 1991.

[14] R Barrett, M Berry, TF Chan, J Demmel, and J Donato. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[15] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, (1):1, 2009.

[16] M Benzi and M Tuma. A comparative study of sparse approximate inverse preconditioners. *Applied Numerical Mathematics*, 1999.

[17] AM Bruaset. *A survey of preconditioned iterative methods*. CRC Press, 1995.

[18] T. Cebeci and A. M. O. Smith. Analysis of turbulent boundary layers. *New York*, 1974.

[19] Matthew J. Churchfield and Gregory a. Blaisdell. Numerical/Experimental Study of a Wingtip Vortex in the Near Field. *Aiaa Journal*, 33(9):1561–1566, 1995.

[20] HA Van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 1992.

[21] Jack Dongarra, I a N Foster, Geoffrey Fox, William Gropp, K E N Kennedy, and Linda Torczon. *of Parallel Computing*. 2003.

[22] JJ Dongarra, IS Duff, DC Sorensen, and HA Van der Vorst. *Numerical linear algebra for high-performance computers*. 1998.

[23] P Du, R Weber, P Luszczek, S Tomov, and G Peterson. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 2012.

[24] V Faber and T Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM Journal on Numerical Analysis*, 1984.

[25] J Fang, AL Varbanescu, and H Sips. A comprehensive performance comparison of CUDA and OpenCL. *Parallel Processing (ICPP),*, 2011.

[26] R Fletcher. Conjugate gradient methods for indefinite systems. *Numerical analysis*, 1976.

[27] MB Van Gijzen and P Sonneveld. An elegant IDR (s) variant that efficiently exploits bi-orthogonality properties. 2010.

[28] A Greenbaum. *Iterative methods for solving linear systems*. SIAM, 1997.

[29] MJ Grote and T Huckle. Effective Parallel Preconditioning with Sparse Approximate Inverses. *PPSC*, 1995.

[30] JL Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 1988.

[31] MR Hestenes and E Stiefel. *Methods of conjugate gradients for solving linear systems*. 1952.

[32] Mark D. Hill and Mark D. What is scalability? *ACM SIGARCH Computer Architecture News*, 18 (4):18–21, dec 1990.

[33] M. Hoestra. *Numerical simulation of ship stern flows with a space-marching Navier Stokes method*. PhD thesis, 1999.

[34] P. K. Khosla and S. G. Rubin. A diagonally dominant second-order accurate implicit scheme. *Computers and Fluids*, 2(2):207–209, 1974.

[35] David B. Kirk and Wen Mei W Hwu. *Programming massively parallel processors: A hands-on approach, second edition*. 2013.

[36] V Kumar, A Grama, A Gupta, and G Karypis. *Introduction to parallel computing: design and analysis of algorithms*. 1994.

[37] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45(4):255, 1950.

[38] MSD Lukarski. Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms. 2012.

[39] F R Menter. Eddy Viscosity Transport Equations and Their Relation to the k-epsilon Model. *Journal of Fluids Engineering*, 119(December):876–884, 1997.

[40] Florian R Menter. Zonal Two Equation k-w, Turbulence Models for Aerodynamic Flows. *AIAA paper*, page 2906, 1993.

[41] NM Nachtigal, SC Reddy, and LN Trefethen. How fast are nonsymmetric matrix iterations? *SIAM Journal on Matrix Analysis and*, 1992.

[42] NVIDIA. AmgX | NVIDIA Developer.

[43] NVIDIA. CUDA C Programming Guide, v8.0. Technical Report January, NVIDIA Corporation, 2017.

[44] NVIDIA. CUDA C Best Practices Guide, v8.0. Technical Report January, NVIDIA Corporation, 2017.

[45] PARALUTION Labs. PARALUTION - The Library for Iterative Sparse Methods on CPU and GPU.

[46] PARALUTION Labs. Paralution User manual Version 1.1.0. 2016.

[47] Hoyte C Raven, Auke Van Der Ploeg, and Bram Starke. Computation of free-surface viscous flows at model and full scale by a steady iterative approach. *Starke*, (August):8–13, 2004.

[48] Y Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[49] Y Saad and MH Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 1986.

[50] P Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM journal on scientific and statistical computing*, 1989.

[51] P Sonneveld and MB Van Gijzen. IDR (s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM Journal on Scientific Computing*, 2008.

[52] Bram Starke, Hoyte C Raven, and Auke van der Ploeg. Computation of transom-stern flows using a steady free-surface fitting RANS method. *International Conference on Numerical Ship Hydrodynamics*, 2007.

[53] A Van der Ploeg, L Eça, and M Hoekstra. Combining Accuracy and Efficiency with Robustness in Ship Stern Flow Computation, 2000.

[54] Auke van der Ploeg, Bram Starke, and Christian Veldhuis. Optimization of a Chemical Tanker with Free-surface Viscous Flow Computations. *Prads 2013*, pages 716–723, 2013.

[55] Arno C. N. van Duin. Scalable Parallel Preconditioning with the Sparse Approximate Inverse of Triangular Matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(4):987–1006, jan 1999.

[56] HA Van Der Vorst and TF Chan. Parallel preconditioning for sparse linear equations. 2001.

[57] C W J Vuik, C and Lemmens. Programming on the GPU with CUDA (version 6.5). Technical report, 2015.

[58] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan. Solving Sparse Linear Systems on NVIDIA Tesla GPUs. pages 864–873. 2009.

[59] P Wesseling and P Sonneveld. Numerical experiments with a multiple grid and a preconditioned Lanczos type method. *Approximation methods for Navier-Stokes*, 1980.