POLITECNICO DI MILANO
DIPARTIMENTO DI SCIENZE E TECNOLOGIE AEROSPAZIALI (DAER)
DOCTORAL PROGRAMME IN AEROSPACE ENGINEERING

A UNIFIED GPU-CPU AEROELASTIC COMPRESSIBLE URANS SOLVER FOR AERONAUTICAL, TURBOMACHINERY AND OPEN ROTORS APPLICATIONS

Doctoral Dissertation of:
Andrea Gadda

Supervisor:
Prof. Paolo Mantegazza

Co-Supervisor:
Dr. Giulio Romanelli

Tutor:
Prof. Alberto Matteo Attilio Guardone

The Chair of the Doctoral Program:
Prof. Luigi Vigevano

Year 2016 – Cycle XXIX

I would like to thank my supervisor, Prof. Paolo Mantegazza for this opportunity. Thanks to him I could combine my passion for computer science to my work.

I would like to thank Dr. Giulio Romanelli for his help and support during these 3 years of PhD. I would like to thank him for the collaboration regarding AeroX development and for his advice.

I would like to thank Prof. Luca Mangani and Prof. Ernesto Casartelli from HSLU for their support and collaboration for what concerns AeroX development, especially regarding turbomachinery. I would like to thank them for the hospitality during my period abroad at HSLU.

I would like to thank Dr. Andrea Parrinello and Dr. Davide Prederi for their help and the collaboration concerning open rotors.

I would like to thank Prof. Marco Morandini for his help during these years at Politecnico, especially for what concerns programming and computer science.

I would like to thank all the friends met during these years at Politecnico di Milano, especially Marco, Michela, Fonte, Sara, Desa, Teo, Luca, Mattia, Zaga and all the new crazy friends at the office, Pietro, Davide, Simone, Mattia, Farooq, Aureliano, Malik, Paolo, Mirco, Alessia and all the others.

I would like to thank all my friends from the TAV (TAVerna) for all these years since high-school. Thank you Branza, Teo, Cislo, Giulia, Baro, Pianta, Devix, Mirio, Carbo and all the others! Too much memories!

Finally I would like to thank my parents and my sister, Alessia, for their support during these years. I would like to thank all the other members of my family for all their support.

Abstract

For the aerodynamic design of aeronautical components Computational Fluid Dynamics (CFD) plays a fundamental role. Pure CFD analyses are usually sufficiently accurate for a wide range of problems. However, when the deformability of the structure cannot be neglected or rigidly moving parts appear in the fluid domain, different disciplines (such as Fluid-Structure Interaction), methodologies (such as Finite Element Method) and strategies (such as Multibody System Dynamics) are also required.

Beside the usual aeronautical examples where an accurate study of the interaction between the fluid and the structure is a key part of the design process (e.g. wings, aircraft, helicopter blades), another important field is represented by turbomachinery, where in particular in the literature aeroelastic investigations are not widely performed yet. A recent research trend is also represented by open rotors and propfans.

Together with the availability of more and more powerful computing resources, current trends pursue the adoption of such high-fidelity tools and state-of-the-art technology even in the preliminary design phases. Within such a framework Graphical Processing Units (GPUs) yield further growth potential, allowing a significant reduction of CFD process turn-around times at relatively low costs.

The target of the present work is to illustrate the design and implementation of an explicit density-based URANS coupled aeroelastic solver, called AeroX, for the efficient and accurate numerical simulation of multi-dimensional time-dependent compressible fluid flows on polyhedral unstructured meshes. Turbomachinery and open rotors extensions are also implemented to handle complex compressor, turbine and propfan cases. The solver has been developed within the object-oriented OpenFOAM framework, using OpenCL for GPGPU programming and CPU-GPU interfacing. Different convergence acceleration techniques, such as Multi Grid and Local Time Stepping, are implemented and opportunely tuned for GPU executions in order to allow an implicit-like residuals convergence. Dual Time Stepping is also implemented to allow time-accurate simulations of unsteady cases of aeronautical interest, such as wings and blades flutter. For what concerns aeroelasticity, Radial Basis Functions are employed to interface the aerodynamic and the structural meshes. The modal representation of the structural behavior is adopted thanks to its accuracy and computational efficiency. Inverse Distance Weighting is used to update the aerodynamic mesh points knowing the wall displacements. The solver is specifically designed to exploit cheap gaming GPU archi-

tectures which exhibit high single precision computational power but a limited amount of global memory. Equations are solved in a non-dimensional form to reduce numerical errors. The solver is also natively compatible with more expensive HPC GPUs, allowing the exploitation of their high double precision computational power and their higher amount of memory. Thanks to OpenCL, AeroX is also natively compatible with multi-thread CPU executions.

The credibility of the proposed CFD solver is assessed by tackling a number of aeronautical, turbomachinery and open rotor benchmark test problems including the 2nd Drag Prediction Workhop, the 2nd Aeroelastic Prediction Workshop (AePW2), the HiReNASD wing, the AGARD 445 wing, the NASA's Rotor 67 blade, the 2D/3D Standard Configuration 10 blades, the Aachen turbine and the SR-5 propfan blade. The recent AePW2 benchmark case, in particular, proves that AeroX is capable to predict flutter with an accuracy level that is comparable with the state-of-the-art aeroelastic compressible URANS solvers, requiring just a cheap gaming GPU. In the literature it is difficult to find static aerelastic investigations of turbomachinery blades. Thus, the trim of the NASA's Rotor 67 fan blade is here investigated, showing that the high blade stiffness is responsible for the very small wall displacements. This is translated in negligible differences between the aeroelastic and the purely aerodynamic solutions for such configurations.

The focus of this work is also on computational aspects. With AeroX an average one order of magnitude speed-up factor is obtained when comparing CPUs and GPUs of the same price range.

Keywords: Aeroelasticity, Open Rotors, Turbomachinery, GPGPU, OpenCL

Sommario

La fluidodinamica computazionale (CFD) costituisce un ruolo fondamentale per la progettazione di componenti aeronautici. Solitamente analisi puramente aerodinamiche sono sufficienti per una vasta gamma di problemi. Tuttavia, quando la deformabilità della struttura non può essere trascurata oppure quando nel dominio fluido sono presenti parti rigide in movimento, altre discipline (come l'interazione fluido-struttura), metodi (come il metodo agli elementi finiti) e strategie (come la dinamica dei sistemi multi-corpo), risultano necessarie.

Accanto ai soliti esempi aeronautici dove un accurato studio dell'interazione tra il fluido e la struttura è un punto chiave nel processo di progettazione (ad es. ali, interi aerei, pale di elicottero), un altro campo è rappresentato dalle turbomacchine, dove in letteratura analisi aeroelastiche statiche non sono ancora ampiamente effettuate. Un trend recente è inoltre rappresentato dagli open rotor e dai propfan.

Assieme alla disponibilità di risorse di calcolo sempre più potenti, l'idea attuale è quella di adottare strumenti in grado di restituire soluzioni accurate già nelle fasi preliminari di progettazione. All'interno di questo concetto le schede grafiche (GPU) permettono una significante riduzione dei tempi di calcolo a costi relativamente bassi.

Lo scopo di questo lavoro è quello di illustrare la progettazione e implementazione di un solutore aeroelastico esplicito, comprimibile, viscoso (URANS), chiamato AeroX, adatto alla simulazione efficiente ed accurata di casi instazionari e multidimensionali, compatibile con mesh poliedriche non strutturate. Nel solutore sono anche implementate estensioni riguardanti turbomacchine e open rotor per poter gestire casi di compressori, turbine e propfan. Il solutore è stato sviluppato nel contesto dell'ambiente orientato ad oggetti OpenFOAM, usando OpenCL per la programmazione GPGPU e per interfaccia CPU-GPU. Diverse tecniche di accelerazione della convergenza, quali Multi Grid e Local Time Stepping, sono implementate e ottimizzate per esecuzioni su GPU in modo da ottenere andamenti di convergenza simili a un solutore implicito. Inoltre, il Dual Time Stepping è implementato poter sfruttare queste tecniche anche con casi instazionari di interesse aeronautico come il flutter di ali e palette. Per quanto riguarda l'aeroelasticità, le Radial Basis Function sono utilizzate per interfacciare mesh strutturali e aerodinamiche. La rappresentazione modale del comportamento strutturale è adottata per via della sua accuratezza ed efficienza computazionale. L'Inverse Distance Weighting è usato per aggiornare la posizione dei punti della mesh aerodinamica sulla base degli spostamenti della parete. Il solutore è progettato per

sfruttare le architetture delle GPU da gioco che sono caratterizzate da un'elevata potenza di calcolo in singola precisione ma una limitata quantità di memoria globale. Le equazioni sono risolte in forma adimensionale per ridurre gli errori numerici. Il solutore è inoltre nativamente compatibile con le più costose GPU da HPC, permettendo di sfruttarne l'elevata potenza di calcolo in doppia precisione e la maggiore quantità di memoria. Grazie a OpenCL il solutore è inoltre nativamente compatibile con l'esecuzione multi-thread su CPU.

Il solutore è stato validato con diversi casi aeronautici, di turbomacchine e open rotors come il 2nd Drag Prediction Workshop, il 2nd Aeroelastic Prediction Workshop (AePW2), l'ala HiReNASD, l'ala AGARD 445, la pala del Rotor 67 della NASA, la pala della Standard Configuration 10 (2D e 3D), la turbina Aachen e la pala del propfan SR-5. Il recente benchmark AePW2, in particolare, prova che AeroX è capace di completare analisi di flutter con un livello di accuratezza comparabile a quello fornito dallo stato dell'arte dei solutori comprimibili URANS aeroelastici, richiedendo semplicemente l'uso di un'economica GPU da gioco. In letteratura è difficile trovare analisi aeroelastiche statiche di palette di turbomacchine. In questo lavoro è quindi stata effettuata l'analisi di trim della pala del Rotor 67, mostrando che la sua elevata rigidezza è il motivo per cui i suoi spostamenti sono ridotti e di conseguenza le differenze tra soluzioni aeroelastiche e soluzioni puramente aerodinamiche sono trascurabili.

In questo lavoro l'attenzione è stata posta anche sugli aspetti computazionali. Uno speed-up medio di un ordine di grandezza è stato ottenuto confrontando CPU e GPU della stessa fascia di prezzo.

Contents

Nomenclature

AePW Aeroelastic Prediction Workshop

ALE Arbitrary Lagrangian Eulerian

AMI Arbitrary Mesh Interface

API Application Programming Interface

APU Accelerated Processing Unit

AVX Advanced Vector Extension

BC Boundary Conditions

BDF Backward Differentiation Formula

BSCW Benchmark Super-Critical Wing

BZT Bethe–Zel'dovich–Thompson

CC Cell Centered

CC-NUMA Cache Coherent Non Uniform Memory Access

CC-UMA Cache Coherent Uniform Memory Access

CFD Computational Fluid Dynamics

CFL Courant–Friedrichs–Lewy condition

CN Crank–Nicolson

CPU Central Processing Unit

CROR Counter Rotating Open Rotor

CU Compute Unit

DDES Delayed Detached Eddy Simulation

DDR Double Data Rate

DES Detached Eddy Simulation

DNS Direct Numerical Simulation

DP Double Precision

DPW Drag Prediction Workshop

DTS Dual Time Stepping

ECC Error Correcting Code

EE Explicit Euler

FEA Finite Element Analysis

FEM Finite Element Method

FFT Fast Fourier Transform

FLOPS FLoating point Operations Per Second

FPGA Field-Programmable Gate Array

FRF Frequency Response Function

FSI Fluid Structure Interaction

FV Finite Volume

FVM Finite Volume Method

GDDR Graphics Double Data Rate

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphical Processing Unit

GUI Graphical User Interface

GVT Ground Vibration Tests

HB Harmonic Balance

HBM High Bandwidth Memory

HiReNASD High Reynolds Number AeroStructural Dynamics

HPC High Performance Computing

HR High Resolution

IBPA Inter-Blade Phase Angle

IC Initial Conditions

IDE Integrated Development Environment

IDW Inverse Distance Weighting

IE Implicit Euler

LES Large Eddy Simulation

LS Least Square

LTS Local Time Stepping

MBS Multi-Body System

MG Multi-Grid

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Multiple Data

ML Mixing Length

MLS Moving Least Square

MP Mixing Plane

MRF Moving Reference of Frame

NLFP Non Linear Full Potential

NS Navier–Stokes

NUMA Non Uniform Memory Access

ODE Ordinary Differential Equation

ORC Organic Rankine Cycle

OTT Oscillating TurnTable

PAPA Pitch And Plunge Apparatus

PDE Partial Differential Equation

PE Processing Element

PIG Polytropic Ideal Gas

R67 Rotor 67

RAM Random Access Memory

RANS Reynolds Average Navier–Stokes

RBF Radial Basis Function

RK Runge–Kutta

ROM Reduced Order Model

RPM Revolutions Per Minute

RS Residual Smoothing

SA Spalart-Allmaras turbulence model

SC Standard Configuration

SDK Software Development Kit

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SISD Single Instruction Single Data

SM Streaming Multiprocessor

SMP Symmetric Multi-Processing

SP Single Precision

SPH Smooth Particle Hydrodynamics

SSE Streaming SIMD Extensions

SST Shear Stress Transport

SU Speed-Up

TDP Thermal Design Power

TDT Transonic Dynamics Tunnel

UMA Uniform Memory Access

URANS Unsteady Reynolds Average Navier–Stokes

USD US Dollars

CHAPTER 1

Introduction

The aim of this first chapter is to introduce the fundamental concepts that motivate this work. The reader will be provided with an overview of the current state-of-art approaches in the numerical aeroelastic analyses of typical aeronautical cases. An important part of this work is also dedicated to the role played by GPUs in accelerating the solver computations. Thus, a brief introduction of the modern technologies adopted to accelerate numerical simulations is showed. Exploiting General Purpose GPU (GPGPU) to reduce the simulation times is in fact a very relevant trend in a wide range of numerical applications, from CFD to finance and cryptography. One of the aim of this work is to build a general purpose solver, called AeroX, that is also capable of performing turbomachinery and open rotors simulations thanks to dedicated extensions. Finally the structure of the thesis is presented alongside a brief introduction of the most important concepts presented in each chapter. This work can be also viewed as a continuation of what started by Romanelli and Serioli [136] and Romanelli [127] with the AeroFoam solver, pursuing the goal of obtaining fast and accurate solutions at the very beginning of the design phases of the aeronautical component.

1.1 Background, CFD/FSI and HPC state-of-the-art

Computational Fluid Dynamics (CFD) is nowadays a fundamental tool for the aerodynamic design in the aeronautical field. CFD allows to simulate almost every kind of aeronautical component, from simple airfoils to entire jet fighters.

In CFD, like in every other kind of simulation, three fundamental aspects must be considered: the mathematical/physical modelization of the reality, the discretization of the problem through numerical formulations and the computational side.

Depending on which physical effects are required to be modeled in a particular

case, different formulations and algorithms can be used to obtain the final solution. Obviously as more and more physical effects are modeled, the simulation cost in term of computational time is increased. Thus, the usual approach in every engineering field is based on an iterative process. Computationally inexpensive formulations are adopted for the initial design of the aerodynamic component when high results accuracy is not necessary. As the development of the particular component progresses, more accurate numerical and experimental results are required in order to verify that the performance and efficiency will reach the prefixed target. Often, a final optimization loop is adopted in order to find the best parameters that satisfy all the project requirements.

As said, accurate formulations are computationally expensive, thus it is the engineer's job to figure out the perfect trade-off between results accuracy and simulation times. Results accuracy and computational power are always related. When CFD was born the computational power of a personal computer was orders of magnitude lower than what we can find today in a cheap smartphone. Nowadays it is basically possible to buy a true computer for $5\$$ (Raspberry Pi Zero [30]) with a $1\,GHz$ ARM CPU. Obviously the first approaches to CFD were largely restricted by the limited amount of available FLOPS (Floating Point Operations Per Second) and memory. Historically the first adopted methods were represented by Doublet Lattice Method (DLM) by Morino, exploiting a linearization of the aerodynamic problem under small disturbances hypotheis. Full potential formulations where implemented firstly in [57, 83] '70 eventually in a finite volume framework [84] (1977). These methods are now relatively inexpensive and can be adequate when it can be safely hypotized that strong nonlinear effects such as shocks and separations do not appear in the flow. It must be noted that using a Non Linear Full Potential (NLFP) formulation [66, 115, 116] it is easily possible to handle cases with weak shocks, when they are not strong enough to invalidate the isoentropic hypotesis. However, a state-of-the-art formulation [115, 122] for the NLFP can be adopted to handle this occurrence. These methods are still used to provide, today in matter of seconds/minutes [66] on desktop computers, a general idea of the performances that can be provided by an airfoil/wing/rotor/aircraft. After the initial design decisions, usually performed with a workstation, it is then possible to perform a round off by running more accurate and computationally expensive simulations on more powerful cluster computers.

Historically as more and more accurate results were required and higher computational power became available, potential methods were surpassed by the Euler formulation in 1980s. Again, viscous effects are neglected, however strong compressible non linear effects given by shock waves can be modeled. Euler methods are an order of magnitude faster with respect to more expensive compressible viscous simulations and usually provide enough accurate results when it is known that strong viscous effects are not likely to occur in the particular test case under analysis [130, 131, 136]. Thus, compressible Euler formulations can be viewed as an alternative to the NLFP or panel methods for the initial design phases of the aeronautical component. It must be noted however that in a compressible Euler approach we have to deal with all the 5 conservative variables (density, momentum vector, and total energy), while in a one-field or two-field NLFP approach it reduces down to only one or two variables (velocity potential or density and velocity potential) with obvious advantages in term of memory consumption and simulation times.

The next logical step, following both the computers evolution and the typical engineering work flow for an aeronautical component design, is the introduction of a way to model viscous effects [120]. Directly solving the full compressible Navier–Stokes (NS) equations in what is called Direct Numerical Simulation (DNS) is correct from a mathematical and physical point of view. However the computational cost of the DNS is today still prohibitive for a wide range of typical aeronautical cases of interests with high Reynolds numbers. The problem is not strictly related to the computational cost of solving the NS equations inside a single discretized entity of the continuum. The main drawback arises from the necessity of discretizing the fluid domain up to the smallest scales of the turbulence [120]. Nowadays with the available computational power DNS simulations are limited to relatively low Reynolds and peculiar cases. Of course the situation is likely to change in the future thanks to the research in both the mathematical/numerical and computational sides. Currently two main alternatives to DNS are available when the continuum hypothesis is considered and an eulerian or ALE (Arbitrary Lagrangian Eulerian) space formulations are adopted (thus excluding approaches like Boltzmann/Lattice-Boltzmann/SPH): (U)RANS and LES (and their combinations like DES/DDES). The idea behind (U)RANS, (Unsteady) Reynolds Average Navier–Stokes, resided in the "modelization" of the small scales turbulence effects, thus excluding the necessity of their effective "resolution". This is usually done using the Boussinesq hypothesis [120] and eventually by solving additional partial differential equations (i.e. the turbulence equations associated to the particular chosen model). Since (U)RANS represents a modelization of the reality, different models were developed in the last decades. Mixing Length [54], Spalart–Allamaras [140], $k-\omega$ [152], $k-\epsilon$ [53], SST [107, 108] are just few examples in a very rich literature. Some models were developed and opportunely tuned for specific problems. As an example, $k-\omega$ performs well in near wall regions, $k-\epsilon$ instead performs well far from the wall and in free shear layers, Spalart–Allmaras is specifically designed for aeronautical cases without boundary layer separations (wings, airfoils in normal conditions). Mixing Length models (e.g. Smagorinsky [139] and Baldwin-Lomax [43]) are usually less accurate and more dissipative than one or two equations models, requiring damping functions like Van Driest [148] for the near wall regions. However, the main advantages of ML models is their relatively low computational requirements since no additional PDEs are required to compute the turbulent viscosity. Some models are meant to be general purpose, providing quite accurate results on a wide range of different cases. As an example, $k-\omega$ SST was developed in order to exploit the advantages offered by both $k-\omega$ and $k-\epsilon$ models, allowing to simulate both near wall and far from the wall flows. The literature offers many papers describing corrections and optimiziations for existing (U)RANS models in order to better describe particular cases (e.g. on [1] it is possible to see 7 variants just for the SST model). Sometimes (U)RANS models can be opportunely tuned with experimental results. Another important aspect is represented by the so called wall functions. In fact, usually (U)RANS models require a fluid domain discretization up to the viscous sublayer (e.g. $k-\omega$, $k-\omega SST$ and SA), where the non-dimensional wall distance (y^+) is in the order of 1. This, of course, is directly translated in a mesh refinement near the wall regions that, together with the costs given by the discretization of viscous terms and the resolution of the the turbulence equations, is the main reason of the greater computational costs with respects to an Euler simula-

tion. Wall functions and automatic wall treatment [88, 121] are developed to asses this problem by lowering the near-wall discretization requirements. This way it is possible to perform (U)RANS simulations using a mesh where the wall distance of the first cell is such that $y^+ > 30$ obtaining at the same time accurate viscous effects. Automatic wall treatments can be also used to automatically switching on and off wall functions depending on the value of y^+. This is done by using near-wall formulations in regions where the near-wall discretization allows to resolve the viscous sublayer and at the same time by using the log-law formulations in regions where the near wall discretization is reduced. Another advantage of this approach is that it is not needed an iterative procedure of building meshes until the one that exactly matches the y^+ values for which the adopted turbulence model is supposed to produce accurate results, over all the discretized wall surface, is found. It must be noted, however, that turbulence is a strictly unsteady and 3D phenomena. Thus, using (U)RANS for steady and/or 2D cases means hiding further approximations. Furthermore (U)RANS represents a cheap modelization of the turbulence effects. Thus (U)RANS models have an intrinsic limited range of applicability. In particular, complex phenomena like separations, boundary layer-shock interactions and transition from laminar flows still represent a challenge for (U)RANS simulations. More complex models were also developed (e.g. RSM, Reynolds Stress Models) in order to improve results accuracy of (U)RANS simulations. In RSM models the idea is to discard the Boussinesq hypothesis and directly model each component of the Reynolds stress tensor. Of course this is translated in more expensive simulations due to the necessity to solve more turbulence equations than Spalart–Allmaras and SST models. Besides the fact that different turbulence models may perform well in some peculiar cases and poorly in other cases for which they are not opportunely tuned, when approaching a new case an important aspect is also given by user experience. For aeronautical components like airfoils, wings, airplanes, turbomachinery blades and open rotor blades usually models like SST and SA are the first choice for a good trade-off between results accuracy and computational requirements [127].

The future represented by compressible DNS for all aeronautical cases is still far away. However an intermediate point between (U)RANS and and DNS is already available today and is represented by the Large Eddy Simulation (LES) and the Detached Eddy Simulation (DES, and eventually DDES for Delayed Detached Eddy Simulations). It must be noted however that LES represents currently a very active research field [147]. Roughly speaking in the LES approach the user is able to choose the particular scale that divides what is resolved (as in the DNS) and what is instead modeled (as in (U)RANS) in term of turbulence effects. Very small scales, that negligibly contribute to the final solution are just modeled, in a (U)RANS fashion. Bigger scales of an engineering interest are instead fully resolved. LES simulations are however at least one order of magnitude more expensive in term of simulation times with respect to (U)RANS ones. Although LES are not yet the standard in the aeronautical field, they surely represents the next future for viscous simulations. An important advantage of LES over (U)RANS is that the LES solution converges to the DNS solution by improving the mesh refinement. For (U)RANS equations instead, mesh convergence analyses should always be performed since from a mathematical point of view there is no guarantee to converge to the DNS solution. A less expensive alternative to LES is represented by the concept of DES. One of the main advantages provided by DES for-

mulations is that they can usually be implemented as simple modifications to an existing (U)RANS model [108, 142]. The so-called DDES (Delyaed DES) formulation was also introduced [141]. In near wall regions and zones where the turbulent scales are smaller than grid dimensions the model is switched to the (U)RANS mode. Where instead the turbulence scales are bigger than the grid dimensions, the model is switched to the LES mode. DES strategies can provide better results in strongly unsteady simulations than plain (U)RANS formulations with relatively lower computational requirements with respect to plain LES formulations.

LES surely represents the future but is still years away from an adoption as the default approach both in academic and industrial fields. (U)RANS and DES, opportunely tuned with experimental data, provide nowadays enough accuracy for a wide range of cases and operating regimes. Thus they will still represent the default approach for viscous cases in the next years.

Once the most suitable model is chosen, after considering both its accuracy and costs, the next problem is represented by the numerical aspects of its solutions. Of course there are problems for which the analytic solution can be easily found. However, when considering the solution of compressible Navier–Stokes equations for arbitrary geometries, with particular initial and boundary conditions, eventually with moving walls, some sort of numerical discretization is required.

A plethora of numerical schemes, algorithms and formulations can be found in literature to discretize the same particular problem. Let us consider the solution of the compressible URANS equations which are the main goal of this work. URANS represents a system of partial differential equations with temporal terms, convective terms, diffusion terms and source terms. They contain spatial and temporal derivatives of the unknowns and require consistent initial and boundary conditions to be specified in order for the problem to be well posed. The numerical discretization of the problem starting from its analytic representation is mandatory in order to be implemented as an algorithm that can be processed by a computer. Thus, analytic operators such as spatial and temporal derivatives have to be substituted by their numerical counterparts. The idea is to express the problem in a form that can be processed by a computer.

Different domain discretization approaches can be used. Usually the geometry of the problem is firstly specified (e.g. with an STL file) and then a mesh is generated using software like gmsh, gambit, icem, blockmesh, pointwise. However mesh-free approaches exist in which there is no particular connectivity between the numerical points on which the solution is defined. The Smoothed Particle Hydrodynamics (SPH) approach represents an example and is particularly used among Multi Body Systems (MBS) analyses. The main difference between the classical Finite Volume Method (FVM) and the SPH approach is that in the former case an Eulerian representation of the fluid is adopted (eventually an ALE formulation if considering mesh deformation), while in the latter a Lagrangian representation is adopted. In CFD, FEM is usually adopted for incompressible low Reynolds simulations. FVM is instead used usually for compressible and incompressible high Reynolds simulations. Both FEM and FVM formulations can be formally obtained from the representation of the (U)RANS equations in weak form. Usually a Petrov-Galerkin approach is adopted, in which the functional space of the test functions is the same adopted for the solution. However a Bubnov-Galerkin approach can be used as well, allowing to use two different functional spaces.

Within an FVM framework, a cell-centered formulation or a node-centered formulation can be adopted. In the former case usually the solution is considered uniform over the entire numerical cell [66, 127, 136]. In the latter, usually a linear interpolation (similar to a FEM approach) can be adopted to represent the solution [115].

Based on the chosen unknowns to describe the problem, two main families are usually adopted in CFD: pressure-based methods [102] and density-based methods [127, 136]. In particular, in density-based approaches [124] the unknowns are represented by the density, the momentum and the specific total energy, i.e. conservative variables. These would be the only unknowns in an Euler (inviscid) formulation. In (U)RANS formulations further variables, related to the turbulence model adopted, must be taken into account for the solution. Density-based schemes are usually well-suited for subsonic, transonic and supersonic cases. However when the Mach number approaches 0, they are usually characterized by convergence problems. Different strategies, based on preconditioning [58, 96, 150] can be adopted to obtain an all-Mach formulation. In general, if a density-based formulation is chosen, different convective numerical fluxes can be adopted, such as Roe [94, 95, 136], AUSM+ [97], CUSP [145] and Jameson [85,86]. These are different examples of upwind fluxes that guarantee stability during the convergence but have the main drawback to be only one order accurate. This problem can be tackled through the use of flux limiters [136] and automatically switch to a second order formulation wherever is possible in the computational domain. In particular, in near-shock regions the second order contribution is switched off, to avoid oscillations. while it is fully recovered in smooth regions. Another important aspect is represented by the entropy fix (e.g. by Harten and Hyman [117]), necessary to avoid non-physic results.

For what concerns viscous fluxes, different numerical schemes with different costs and accuracy levels can be adopted to compute the required gradients. One of the most simple and robust scheme is represented by the Gauss formulation, well suited for cell-centered approaches. In this case the cell gradient is assembled adding up the contributions of the faces, contributions that are computed directly from the cell and neighbor cells. Since no upwind-like concepts are required for the viscous terms, a simple and cheap weighted average between the cells values is enough. Another possible formulation is represented by the Least Square scheme (LS) [66] mostly used in node-centered approaches. This is less robust and more expensive but usually provide more accurate results.

For what concerns the temporal discretization of the problem, two main formulations exists: explicit and implicit schemes. Explicit schemes can be easily implemented, are easily parallelizable (since the solution at the new time depends only from the values stored at the previous times) and they require a small amount of memory since the matrix storage is avoided. Their main disadvantage is represented by the CFL constraint that limits the maximum value of the allowed time-step. On the other side, the solution of implicit iterations is more costly since it basically requires the solution of a linear system. The problem is also represented by the fact that Navier–Stokes equations are non-linear, meaning that some sort of linearization is required to compute the solution. Of course Newton–Raphson method can be employed to perform this task, however it basically requires to perform multiple factorizations at each physical time step (although the same factorized matrix can be used for multiple iterations [115]).

One of the main drawbacks from a computational point of view of implicit schemes is represented by the high memory requirements, since the system matrix has to be stored (although in a sparse format). The main advantage of implicit methods is represented by the possibility of using large time-steps (thus big CFL values) without severe stability problems. Segregated methods consists in a mixed approach between a fully implicit and a fully explicit scheme. Basically some equations are solved using linear systems, like a typical implicit scheme. However other variables are updated in an explicit-like manner, usually the turbulence equations. These strategies allow a reduction of the size of the system that would be required in a full implicit approach, at the cost of inferior convergence performances. A staggered approach [66] can be also implemented. As an example the solution of some equations is computed using the values previously obtained from the solution of other equations in the same iteration. That said, considering the allowed values of time-steps and the costs of each iteration, explicit method are usually preferred when small time-steps are required, e.g. when studying acoustics. Implicit schemes are instead preferred when large time-steps are required, e.g. to reach a steady-state solution. Different strategies can be adopted in order to speed up the convergence of explicit schemes, bypassing the CFL limit without actually violating it. The most important strategies are represented by the Local Time Stepping (LTS), Residual Smoothing (RS), Multi-Grid (MG) [48]. These schemes can be combined together to damp residuals and achieve convergence rates comparable to what provided by implicit formulations. These convergence acceleration techniques speed up the convergence of explicit methods to reach steady solution whith null residuals but cannot directly adopted in unsteady simulations. This is due to the fact that the time, at this point "pseudo time" loses its physical meaning. Dual Time Stepping (DTS) [48] can be adopted to perform unsteady simulation with explicit methods while maintaining convergence acceleration active. The idea behind DTS is basically to converge from one physical time to the next one by solving a steady problem with source terms representing the physical temporal derivatives. This way all the CFL problems are related to the pseudo time handled with LTS, while the physical time step can be chosen independently. With DTS it is possible to employ physical time steps that are not limited by CFL restrictions, allowing to reconstruct only the frequencies of interest, reducing the computational effort with respect to a global time stepping strategy.

The DTS technique can be used to solve the fully non-linear URANS equations in the time domain. This can be profitably used when unsteady complex non-linear phenomena are investigated on a generic computational domain. A subclass of aeronautical problem is represented by turbomachinery and recently by the renewed interest in open rotors/propfans, which are usually characterized with time and spatial periodicity. Thus, in this class of problem, when the hypothesis of spatial and/or time periodicity is valid, simplifications can be adopted to reduce the computational costs of the simulations. In particular, beside the non-linear time-domain schemes, also time-linearized [63] and the Harmonic Balance (HB) techniques [63] can be adopted. Time-linearized techniques allow a drastic computational effort reduction with respect to time domain strategies but cannot be adopted when strong non-linear phenomena such as separations and shocks occurs in the flow. However the Harmonic Balance technique can be adopted to provide a full non-linear frequency domain formulation. Generally, time-linearized and HB techniques reduce the total computational costs when a single

particular frequency has to be analyzed, since with a non-linear time domain approach an entire unsteady solution would be required. However as will be presented in this work, by exploiting an opportunely crafted aeroelastic system input it is possible to excite a wide range of frequencies in a single unsteady analysis. This supports the choice of a non-linear time-domain solver, especially for flutter analyses.

Alongside CFD simulations that involves purely aerodynamic effects, a primary research field is represented by aeroelasticity. Reality is intrinsically aeroelastic, there is no non-deformable structure. Some structures are such that displacements under aerodynamic loads are negligible from an engineering point of view. However there are important cases, like aircraft wings, where accounting for the structural response provides more accurate results [115, 127]. This is more and more relevant as the adoption of innovative materials (i.e. composite [39]) and technologies advances (i.e. 3D printing [149]). This is particularly true for static aeroelasticity. The most important aeroelastic phenomenon in the aeronatical world is represented by flutter. This is a dynamic instability of the aeroelastic system, when structural and aerodynamic subsystems are considered coupled. Flutter must be investigated as it is directly related to self-sustained vibrations, thus fatigue life, thus safety. At the beginning of the first century of flight aeroelasticity was ignored during the design process. However catastrophic failures occurred and therefore it was evident that aeroelastic analyses needed to be part of the safety-check procedures during the design of new components. Nowadays the trend is to introduce aeroelastic analysis since the very beginning of the design process, leading to the requirement of high efficiency solvers, capable to fully exploit the state-of-the-art computational hardware. However, aeroelasticity is still nowadays an open problem. This is confirmed by the fact that the most recent (2015/2016) effort to asses the the accuracy of state-of-the-art aeroelastic solvers is represented by NASA's Aeroelastic Prediction Workshop 2 (AePW2) [81], with the purpose of comparing results provided by different research groups from all over the world. In the past the AePW1 [80] was also adopted to pursue this goal with the HiReNASD and BSCW wings. The AGARD 445.6 wing flutter investigation [156] is another well know benchmark case. Benchmark cases for turbomachinery aeroelastic investigations also exist, like the SC (Standard Configuration) [69] cases, e.g. SC10. While in turbomachinery aeroelastic investigations the aerodynamic damping analysis seems to be the most important kind of investigation, in literature it is not common to find static aeroelastic analyses. This is probably due to the higher stiffness of blade configurations with respect to more classical aeronautical wings. Besides turbomachinery, open rotors/propfans represent a recent trend in the aeronautical field. The first studies for this kind of configurations are from 1975 by NASA. At their very beginning CROR (Counter Rotating Open Rotors) [78] configurations were affected by high noise levels. The current need to find new high efficiency solutions for aeronautical propulsion indicated open rotors as a possible candidate, renewing the interests over this kind of configurations [122]. Figures 1.1 show examples of pushing and pulling CROR configurations. As for turbomachinery and helicopter blades, open rotors represent a challenge from the numerical point of view. In fact with the need to account for compressibility and viscous effects from the purely CFD point of view, the structural deformability due to both aerodynamic and centrifugal effects should be taken into account. This clearly suggests that for this kind of rotors what is often called multi-physics approach is required. Figure 1.2 shows the Collar's

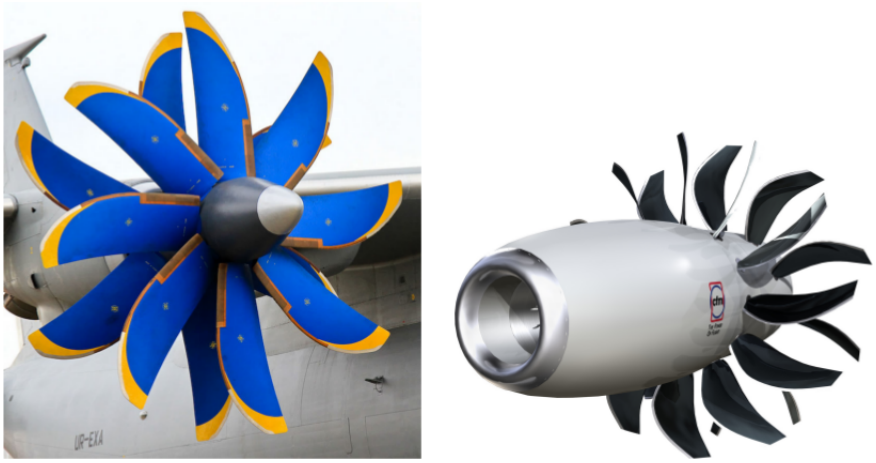


Figure 1.1: *Examples of CROR puller (on the left) and pushing (on the left) configurations [122], www.redstar.org, www.gfdiscovery.blogspot.it.*

triangle [47], highlighting the connection between different subsystems [127]. These triangle can also be modified adding the control subsystem [127]. The control subsystem is particularly important for different applications, e.g. gust alleviation [68]. These interactions could also trigger non-linearities [127]. Accurate aeroelastic simulations

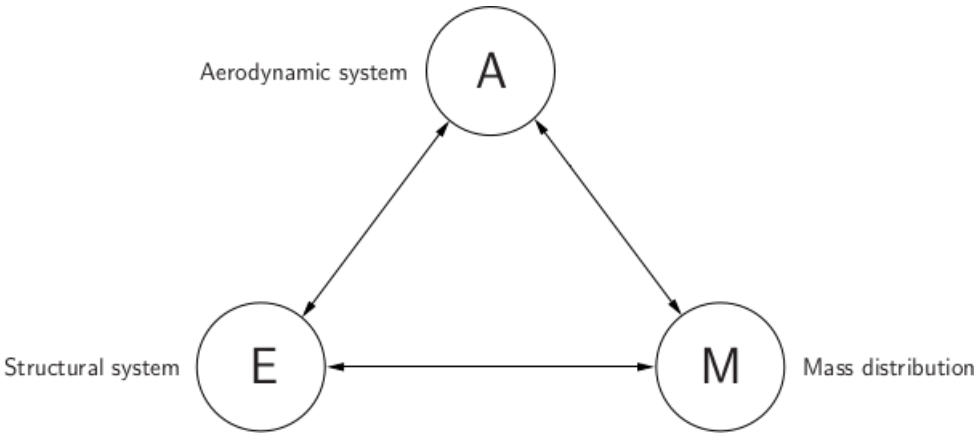


Figure 1.2: *Collar's triangle, interaction between subsystems.*

requires CFD formulations to be coupled with algorithms that allow mesh deformation. In particular, Radial Basis Functions (RBF) interpolation schemes can be adopted to build the so-called aeroelastic interface between the usually different structural mesh and aerodynamic mesh wall discretizations [52, 123]. An option is represented by Inverse Distance Weighting (IDW) algorithm [137, 154] that can be also used to update aerodynamic mesh internal nodes location by knowing wall displacements.

The schemes and algorithms chosen for the numerical discretization of the problem are strictly related to computational aspects. As previously said, even smartphones have nowadays orders of magnitude the computational power of the computers of few decades ago. When performing numerical simulations with a computer, at least a basic knowledge of how computers work, from a software and hardware point of view, is required. This is necessary in order to efficiently implement the numerical algorithms.

Roughly speaking, from the hardware point of view, when performing simulations, the two most important aspects are represented by the total amount of memory available and the Floating Point Operations Per Seconds (FLOPS) achievable. Memory is strictly related to the problem size that can be handled: more memory means bigger problem sizes. FLOPS are instead related to the speed at which the computations can be performed: more FLOPS means that more computations can be performed per unit of time. It must be noted that the computational time required to obtain the solution does not depend only by the available computational power. Fundamental aspects are also represented by the convergence properties of the implemented schemes. In fact, let us consider a scheme A that requires very small time per iteration, but it requires a lot of iterations to reach convergence, and scheme B that is more costly in term of time per iteration but requires much less iterations to reach convergence. Since the goal is to obtain the solution as soon as possible, the scheme B could be the candidate to be implemented. Another important aspect from the computational point of view is that the choice of the algorithm must take into account also the particular architecture of the machine over which it will be executed. As an example, if a quad-core CPU is available, in order to exploit the full available computational power, the chosen algorithm must be capable of split the work in chunks that will be distributed among cores. If instead the algorithm is intrinsically serial, 3/4 of the available computational power will be wasted.

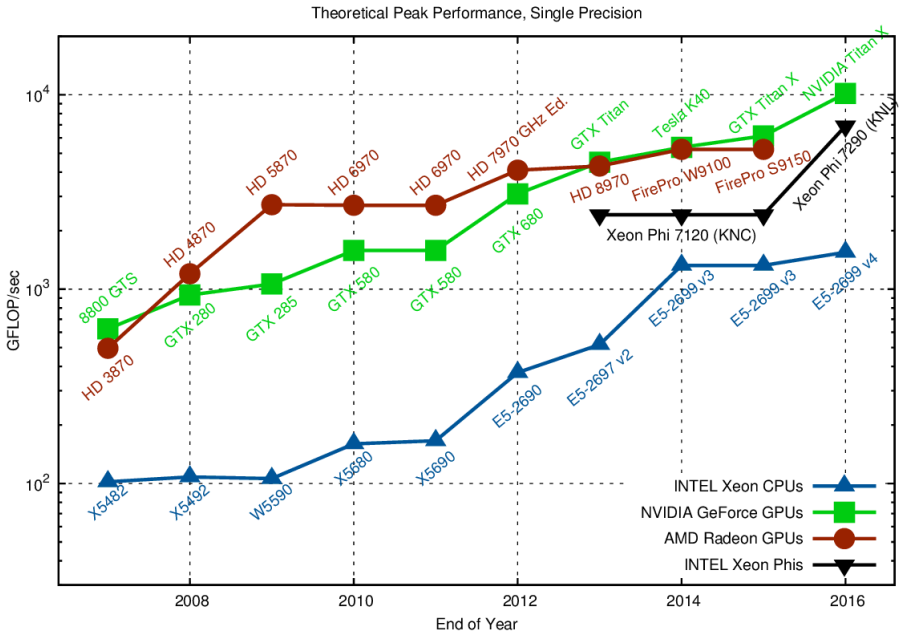
Multi-core CPUs are available for personal computers since a decade. Until the first years of 2000 CPUs were basically single core processors. From year to year new architectures were presented with the goal of improving serial performances, mainly by allowing higher frequencies to be reached. As an example, Intel was publicizing its Pentium 4 processors with their GHz-range frequencies. Transistor scales where reduced up to nanometers, approaching not only engineering limitations but also physical limitations. It was clear that the single-core performances, thus serial performances of processors were reaching their intrinsic physical limits with the available technology. Furthermore, increasing core frequencies to over 3 or 4 GHz became very difficult. Thus, in order to improve CPU performances, different strategies were adopted. First of all the multi-core concept. Basically the idea is to have multiple independent connected computational units, sharing memory, on the same socket. Modern operating systems like Windows, OSX and Linux distributions support multi-tasking thanks to concepts like processes, threads, time slices. When the operating system kernel is executed on a multi-core CPU, it is allowed to schedule processes/threads for execution over the available cores. This way, when performing numerical simulations, with the right algorithm, it is also possible to distribute the work among the available cores.

Besides the CPU, another powerful chip installed in basically every modern computer is the GPU, the Graphical Process Unit. As the name suggests, the main purpose of this device is to accelerate computations strictly related to graphics. When performing graphical computations, the same operation has to be performed on a large amount of pixel or vertexes, meaning that GPUs are intrinsically SIMD (Single Instruction Multiple Data) devices. As an example, in order to draw a triangle on the screen and then translate it, basically it is needed to translate each vertex that compose the triangle. This job can be directly performed by the GPU in a parallel way by translating each vertex that compose the triangle, effectively offloading the CPU from the computations. Let us

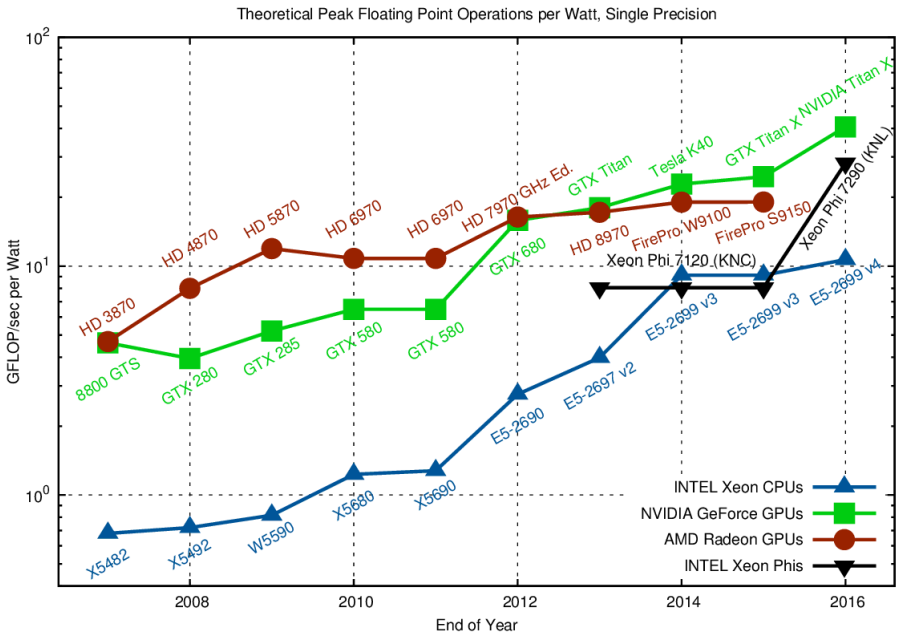
consider now the sum of two vectors: the same operation has to be performed on each couple of numbers. This is basically the same operation needed to compute the new vertexes positions given the initial positions and the translation vector. Since GPUs are specifically designed to perform this kind of computations, it easily understandable why they outperform CPUs of the same price level in specific SIMD computations. The idea behind GPGPU (General Purpose GPU) is, as the name suggests, to use GPUs to perform general purpose numerical computations. Of course, since the GPU architecture is inherently SIMD, only data-parallel algorithms can truly exploit its computational power. When this is possible, and the code is opportunely tuned, one order of magnitude speed-up can be potentially achieved by using a GPU instead of a CPU of the same price level. This might seem impressive, especially considering that almost every computer has nowadays a GPU that can be exploited to offload the CPU and accelerate specific type of computations. However it must be noted that GPGPU programming is somehow cumbersome and exposes numerous limitations with respect to classical CPU programming. Figure 1.3(a) clearly shows the differences between theoretical floating point computational power of CPUs and GPUs and in particular the trend of last years. It can be seen that AMD and NVIDIA GPUs exhibit higher theoretical FLOPS performances considering single precision. The comparison between Single Precision (SP) and Double Precision (DP) performances is a fundamental aspect that will be discussed in detail in this work. Figure 1.4 shows the advantages provided by GPU acceleration in term of performances to costs ratio. Besides the fact that the numbers showed in the figure are highly dependent from the particular chosen CPU/GPU combination, it is clear that adding GPUs to an HPC system contributes to reduce the performances to costs ratio. What is important to notice is not the numbers themselves, but the order of magnitude of the advantages given by using GPUs with respect to a CPU-only system. Obviously it is reminded that these are just theoretical numbers since when tackling a numerical problem it is not always possible to exploit the intrinsically GPU SIMD architectures. The first GPGPU approaches [77] were based on mapping the numerical problem to a graphical problem in order to exploit the graphical API (Application Programming Interfaces), communicate with the GPU and ask it to perform computations. Translating numerical algorithms into pixel/vertexes operations was very difficult at the beginning of GPGPU. Later, in 2007, NVIDIA launched CUDA, providing an easier way to access the computational power of GPUs for generic numerical computations. ATI (nowadays AMD), the main NVIDIA's competitor in GPU market, launched its SDK for GPGPU programming, called ATI Stream. Nowadays different modern SDK and languages can be adopted for GPGPU computing, such as CUDA (NVIDIA only) and OpenCL (multiple CPU, GPU and FPGA vendors). OpenCL and CUDA offers a low level GPGPU programming capabilities. Other languages, such as OpenACC, offer more high level interfaces to GPGPU.

GPGPU is exploited in very different numerical fields such as CFD, finance, cryptography, machine learning, medics, signal and image processing, etc. Different software from different software houses offer the possibility of GPU acceleration. Furthermore, wrappers and libraries for numerical computations are today available to exploit GPGPU by getting rid the user from low-level GPU programming. Examples are represented by ViennaCL [35] or MATLAB.

Despite the advantages given by GPGPU, the programmer must face different draw-



(a) Single Precision theoretical performances



(b) Single precision performances per Watt

Figure 1.3: *CPU vs. GPU performances trend, [2].*

backs and limitations when programming GPUs for general numerical computations. Concepts like branch divergence and coalesced memory access must be taken into account in every algorithm that has to be implemented. Furthermore, a typical GPU exhibit less memory than the typical amount of system memory (RAM) available on a workstation. This means that explicit algorithms are preferred over implicit algo-

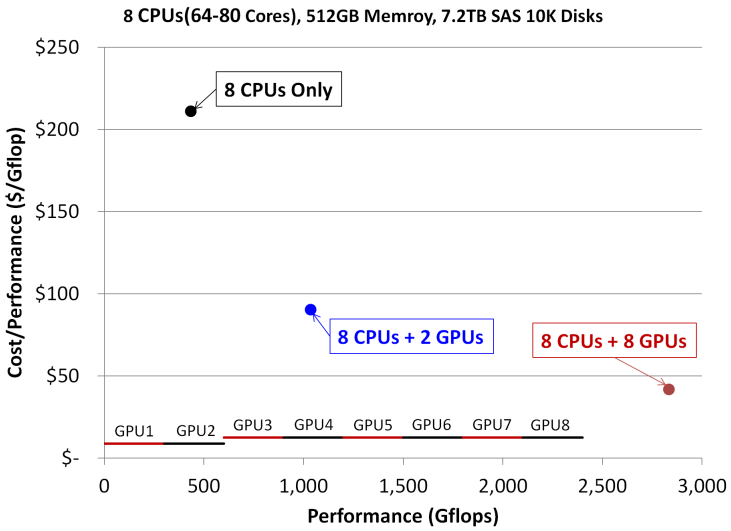


Figure 1.4: *Performances to costs ratio, GPGPU advantages [3].*

rithms, thanks to the avoided matrix storage and intrinsically local memory footprints. Alongside these problems, debugging and profiling GPU code is quite hard with respect to typical CPU code. Particular attention must be dedicated to memory accesses since buffer overflows could lead to unexpected crashes. Finding the exact line of code where buffer bounds are not respected can be very hard. Recently, a debugger was developed, called Oclgrind [27] that basically allows the programmer to obtain a Valgrind-like [34] debugging tool for OpenCL-based applications. Another feature of this great debugger is the possibility of checking for possible data-races.

An important aspect in GPGPU world is given by the differences between gaming GPUs (such as NVIDIA GeForce product line and AMD Radeon product line) and HPC GPUs (such as NVIDIA Tesla product line and AMD FirePro). Usually high-end gaming GPUs exhibit about the same single-precision (SP) computational power provided by HPC GPUs but just a fraction of their double-precision (DP) computational power [37, 38]. Furthermore HPC GPUs have nowadays more than $10\,GB$ of memory, while high-end gaming GPUs have only about $4 - 8\,GB$. Finally HPC GPUs features ECC (Error Correcting Code) compliant memory. One of the goal of this project is to develop a solver that is capable of exploiting the single-precision computational power of gaming GPUs since they are usually one order of magnitude cheaper than HPC GPUs with about the same SP computational power. Of course using SP for the solution of Navier–Stokes equations often requires particular tuning of the code when precision loss could be a problem.

Finally, it must be noted that GPUs have better performances per Watt than CPUs as can be seen in figure 1.3(b). Thus there are also important advantages in using them (when it is possible) from the point of view of energy consumption. This and other aspects regarding GPGPU will be discussed in details in this work.

Usually numerical solvers run on CPUs, eventually with the possibility to exploit multi-core architectures through multi-threading, or clusters with multiple nodes using multi-processing through message-passing strategies. Few programs have the possibility to offload part of the computations to GPUs in order to accelerate specific compu-

tations. Here the idea is instead to use OpenCL to parallelize almost all the solver's algorithms in order to exploit SP performances of modern cheap gaming GPUs. However, the idea is to obtain a solver that is natively compatible with both CPUs and GPUs thanks to the OpenCL runtime libraries and device drivers offered by the most important CPU and GPU vendors (Intel, AMD, NVIDIA). This way with a single source code set it is possible to achieve compatibility with the widest range of devices.

It is worth to note that despite the main target of this work is accelerating simulations through hardware/software-based techniques, other strategies like parametric computing and reduced order methods can be exploited to further reduce computational times and increase simulation complexity [133].

1.2 Overview of the thesis

Here an overview of the chapters is presented. First of all the numerical formulations implemented in the solver are showed alongside the explanations of the choice from both a numerical and a computational point of view, considering that the main goal is a GPU-optimized solver with turbomachinery and open rotors extensions. Then GPGPU main concepts are presented, considering in particular the choice of the OpenCL API and language. Next, the software architecture of the solver is presented, with a detailed view of how GPGPU concepts are translated in the parallelization of different CFD/FSI tasks. Computational benchmarks are then used to assess the speed-up advantages of using GPUs instead of CPUs for the solver execution. Finally numerical results are presented in order to validate the solver's numerical formulations. Different kind of test cases are adopted at this stage, raging from steady to unsteady, from aerodynamic to aeroelastic, from classical aeronautical to turbomachinery/open rotors cases.

Chapter 2

This chapter presents the numerical formulations implemented in the solver to solve the Navier–Stokes equations in an ALE framework. This is required to handle steady and unsteady aeronautical cases with fixed and deformable meshes. As previously said the OpenFOAM framework is adopted for the pre-processing phase, thus a finite-volume cell-centered formulation is adopted. The implemented convective numerical fluxes are here briefly presented, alongside strategies for high resolution, flux limiters and entropy fix. The convective fluxes formulations take into account also the ALE terms necessary for moving and deforming meshes. The implemented schemes for viscous fluxes are then briefly introduced with the implemented automatic wall treatment strategy. The most important boundary conditions for aeronautical cases are here introduced, considering also transpiration boundary conditions to emulate moving boundary effects without actually updating wall points positions. For what concerns aeroelastic simulations, the adopted RBF-based strategy is here described to handle the interface between the aerodynamic and structural meshes. IDW algorithm is described to update internal point positions. Convergence acceleration techniques such as Local Time Stepping, Multi-Grid and Residual Smoothing are here described as the strategies to speed up the convergence of the explicit solver. Dual Time Stepping is also showed for unsteady simulations. The procedures for steady, unsteady, trim, flutter and forced oscillations analyses are also described in this chapter.

Chapter 3

In this chapter an introduction to turbomachinery and open rotors problems and simulations is firstly provided. After the introduction of purely aeronautical cases formulations, this chapter is dedicated to the extensions implemented into the general-purpose solver that are required to handle rotating cases for turbomachinery and open rotors. The main purpose of these formulations is to speed up convergence of cases that exhibit spatial and temporal periodicity. MRF allows to simulate rotating domains without actually rotate the mesh. Cyclic boundary conditions are adopted to reduce the computational domain to an N-blade sector when spatial periodicity is supposed to be related to N blades. Strictly related to this, IBPA and delayed boundary conditions concepts are presented, useful for unsteady cases when adjacent blades vibrates with a particular phase angle. Mixing plane strategy is then introduced to simulate the interface between two communicating blade rows (e.g. one stator row and one rotor row). Again, this is useful when used in conjunction with cyclic BCs in steady simulations. A comparison with other strategies of Navier–Stokes equations solution specifically designed for temporal periodic cases (time-linearized and Harmonic Balance) is also briefly presented alongside the reasons supporting the implemented approach (non-linear time marching).

Chapter 4

This chapter is aimed to introduce the reader to GPGPU concepts. The most important advantages and limitations of using GPUs to perform numerical computations are here presented. Concepts and problems related to branch divergence and sequential memory access are here explained. The attention is focused on OpenCL since it is chosen in this work as the API and language for GPU programming. The most important abstractions (e.g. devices, platforms, kernels) provided by OpenCL are here explained. Examples are provided to better explain when GPUs can be used to accelerate computations. This is used to understand the programming choices adopted in the solver. This chapter is not aimed to be an OpenCL or GPGPU tutorial but anyway could be useful to understand if a particular algorithm can be efficiently ported on GPU architectures.

Chapter 5

After the introduction of the main concepts related to both the numerical and the computational sides of the problem, this chapter is aimed to explain the computational aspects of the solver architecture, how the different subsystems communicate and how the solver algorithms are implemented and tuned. The solver uses C, C++ and OpenCL languages. In particular, two different source sets have to be created, one for the "host" that enqueue work and one for the "device" that actually performs aeroelastic computations. In this chapter pieces of host and device code will be showed to provide a better feel of what really runs under the hood. Furthermore, this chapter better explain the possible bottlenecks with hybrid/unstructured meshes and why some algorithms are instead very efficient if run on a typical GPU architecture. The connection between the solver, AeroX, and the OpenFOAM framework is also showed. Furthermore strategies to improve numerical robustness and convergence are presented.

Chapter 6

This chapter shows the results for what concerns the purely computational side of the problem. The main purpose of using OpenCL is to obtain a solver that can exploit GPU acceleration with devices provided by the main GPU vendors, while retaining CPU multi-thread compatibility. In particular, an important advantage provided by OpenCL is the fact that there is no need to write different codes for different architectures: every CPU and GPU that is compatible with OpenCL can be immediately used by the solver (provided that the correct runtime is installed). This is translated into compatibility with a wide range of devices, and, for the purpose of this chapter, an easy way to compare the simulation times of CPU and GPU executions. In this chapter the speed-up obtained with different CPU and GPU architectures, provided by different vendors such ad AMD, NVIDIA and Intel is showed. As said AeroX is compatible with CPUs and GPUs but is tuned for GPU executions, so it is expected to perform better with this kind of devices. An APU (Accelerated Processing Unit) from AMD is also used for these benchmarks. The speed-ups are also analyzed considering isolated kernels in order to perform an accurate investigation of the computational efficiency of the different implemented algorithms. One of the main goals of this work is to exploit cheap gaming GPUs instead of more expensive, specifically designed HPC GPUs. Thus, the solver is here tested to check for possible problems due to the use of single precision instead of double precision and for possible differences due to the lack of ECC memory in gaming GPUs. This is done in order to respond to the most criticisms encountered by following the gaming GPUs choice. Obviously the solver is also natively compatible with HPC GPUs, double precision and ECC memory.

Chapter 7

In this chapter the solver is validated for the purely aerodynamic formulations side. Here, different test cases and benchmarks are used to show the capabilities of the solver to provide accurate inviscid and viscous solutions in a reasonable amount of time, considering the fact that it can be executed on a relatively cheap desktop computer instead of computer clusters and high-end workstations. In this chapter typical aeronautical geometries are adopted for the validation, such as the RAE and Onera M6 wings and the 2nd Drag Prediction Workshop. This kind of cases are used to show the capability of the solver to accurately predict viscous and compressible effects, with different range of Reynolds and Mach numbers. These cases are investigated before rotating (turbomachinery and open rotors) cases and unsteady cases with mesh deformation in order to validate the solver for what concerns purely aerodynamic formulations for aeronautical cases.

Chapter 8

After the analysis of the purely aerodynamic aeronautical cases, in this chapter the solver is validated for what concerns static and dynamic aeroelasticity. Static aeroelasticity is assessed using a steady test case with deformable structure. For this purpose the well-known HiReNASD wing is adopted for the validation. For what concerns unsteady cases with moving walls, the solver capabilities are investigated through forced oscillations and flutter analyses. Flutter investigations are performed with the AGARD

445.6 wing and the BSCW wing, the latter within the recently presented Aeroelastic Prediction Workshop 2. It is highlighted that the HiReNASD and AGARD 445.6 wings aeroelastic investigations are well-known test cases in the aeroelastic aeronautical world, created ad-hoc to investigate the aeroelastic solvers capabilities to reproduce experimental results with numerical simulations. The AePW2 is instead the most recent effort to asses the state-of-the-art in FSI numerical simulations and is here adopted to compare the results provided in this work to what currently produced by other research groups over the world using state-of-the-art aeroelastic solvers.

Chapter 9

After the validation of the aerodynamic/aeroelastic formulations with aeronautical test cases, the implemented turbomachinery/open rotors extensions are here investigated for what concerns steady cases. Numerical simulations are performed on cases with different levels of complexity. 2D and 3D cases with rotors and stators are investigated. Different formulations specifically designed for turbomachinery and open rotor cases are here validated, such as MRF, cyclic boundary conditions and mixing plane formulations. In particular, the validation is started with a simple 2D stator blade in which only cyclic boundary conditions are adopted. For this purpose the Goldman blade case is used. Then the mixing plane formulation is validated through the simulation of the 1.5 stages Aachen turbine with a stator-rotor-stator blades configuration. Finally an open rotor case from Stuermer is investigated.

Chapter 10

This is the last chapter regarding numerical results. Here the solver is validated with steady and unsteady aeroelastic turbomachinery and open rotor cases. In particular the steady aeroelastic solution of the NASA Rotor 67 is computed. This test case is used to demonstrate the capabilities of the solver to compute the blade deformations of a rotor blade under aerodynamic loads. This is a particularly important result since in the literature it is difficult to find this kind of investigations. However, as the results will show, the steady aeroelastic solution is basically identical to the steady aerodynamic solution, hence it is possible to say that with this kind and similar geometries the steady aeroelastic simulation can be avoided. However, with the efficient implemented strategies, the aeroelastic solution has basically the same computational costs of a classical steady aerodynamic solution, thus it is worth to try to perform a trim simulation when facing a new geometry in order to verify if static blade deformations are negligible from an engineering point of view. Next, the attention is focused on the computation of the aerodynamic damping of the well-known two-dimensional and three-dimensional Standard Configuration 10 (SC10) test cases. These are used to validate the solver when performing unsteady turbomachinery simulations with forced oscillations. In particular, delayed boundary conditions are here exploited in order to simulate different IBPAs while retaining the single-blade domain reduction to reduce the overall computational costs when the spatial periodicity is not related to a single blade domain anymore. Finally a propfan blade, SR-5, is investigated for flutter conditions. In this case a single-blade spatial periodicity is supposed and the same strategy for flutter computation adopted for classical aeronautical cases is adopted. This completes the aeroelastic investigations.

CHAPTER 2

Fluid dynamics and aeroelastic system formulations

The target of this work is the implementation of a finite-volume, cell centered, explicit aeroelastic compressible URANS solver. This means that AeroX is a general purpose solver. Formulations regarding turbomachinery and open rotors will be added as extensions later. Thus, it is necessary to firstly introduce the equations that physically and analytically model the problem to be solved. After the presentation of the equations for the aerodynamic and structural system, the focus will be posed on the numerical discretization schemes. For what concerns the aerodynamic system the implemented schemes to discretize convective and viscous fluxes will be briefly introduced. This is also valid for turbulence models and the adopted wall treatment strategy. After the description of the purely aerodynamic aspects, the schemes and algorithms required to solve aeroelastic problem will be investigated. In fact, a connection between the aerodynamic and structural meshes is required, alongside a strategy to update the aerodynamic mesh based on the displacements of the moving walls. These are required both for static and dynamic aeroelastic analyses. The focus will be also posed on the description of the procedures behind the solution of trim, free and forced oscillations and flutter problems.

2.1 Aerodynamics formulations

First of all, the equations of the purely aerodynamic system are presented and discussed. With the aim of obtaining an aeroelastic solver capable of handling unsteady cases with mesh deformation, the ALE (Arbitrary Lagrangian Eulerian) framework is then presented and the equations are consistently modified with the introduction of the material velocity. The discretization schemes adopted to numerically solve the Navier–

Stokes equations are then briefly showed. Numerical convective fluxes are firstly introduced, which are sufficient for an inviscid analysis (Euler equations). Then, viscous fluxes are discussed, allowing to effectively solve compressible viscous cases. The implemented turbulence models are then presented to complete the RANS equations. The focus is then posed on boundary conditions and wall treatment. Finally Local Time Stepping (LTS), Residual Smoothing (RS) and Multi-Grid (MG) are briefly discussed alongside the Dual Time Stepping (DTS) strategy that allows the exploitation of such convergence acceleration techniques also for unsteady cases. Most of the formulations discussed here, in particular convective fluxes, boundary conditions and cell-centered discretization, are discusses in a more detailed way in [48, 127, 136].

2.1.1 Navier–Stokes equations

With the aim of implementing a compressible URANS solver, the Navier–Stokes equations are here introduced and briefly discussed. Next, the Unsteady Reynolds Average Navier–Stokes formulation are presented as a modification of the original Navier–Stokes system, allowing the adoption of different turbulence models. More in detail, for compressible systems the Favre equations are adopted [48].

From a mathematical point of view, Navier–Stokes equations consist in a system of mixed non-linear Partial Differential Equations (PDE). The system of equations must be coupled with consistent boundary and initial conditions. However, as will be showed, some sort of modelization for what concerns the viscous terms, the equation of state of the fluid, the conduction term, and turbulence effects is introduced to close the problem and to ease the solution from a computational point of view. Navier–Stokes equations can be used to describe the behavior of the fluid flow when the continuum hypothesis holds. This is strictly related to the Knudsen number, $Kn = \frac{\lambda}{L}$, where λ is the mean free path (the average distance traveled by a moving particle between two successive collisions), L is the representative physical length scale. The continuum hypothesis holds when $Kn << 1$, thus when the length scales typical of the investigated phenomenon are larger than the mean free path. From the engineer point of view this is translated in the possibility of using the approach adopted in this work for the majority of aeronautical cases (such as wings, planes, turbomachinery and open rotors blades in subsonic, transonic and supersonic regimes). However, other important study fields related in particular to space applications (such as the reentry of an hypersonic vehicle in the atmosphere) usually requires different formulations based on statistical methods.

Equation 2.1 shows the well-known integral form of the Navier–Stokes equations, expressed in an Eulerian framework, using conservative form variables, for the description of the dynamics of the flow of a compressible, viscous and conductive fluid:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_V \mathbf{U}\,\mathrm{d}V + \oint_S \mathbf{f}(\mathbf{U})\cdot\widehat{\mathbf{n}}\,\mathrm{d}S - \oint_S \mathbf{g}(\mathbf{U})\cdot\widehat{\mathbf{n}}\,\mathrm{d}S = \int_V \mathbf{h}(\mathbf{U})\,\mathrm{d}V \tag{2.1}$$

In these equations the control volume V and its boundary S are considered fixed in space, thus the equations describe the evolution of the variables $\mathbf{U}(\mathbf{x},t)$ inside it. $\mathbf{U}(\mathbf{x},t)$ is the vector of the unknown conservative form variables [124] and depends from the time t and the considered space location $\mathbf{x}$. In this work the conservative form variables, the density $\rho(\mathbf{x},t)$, the momentum $\mathbf{m}(\mathbf{x},t) = \rho(\mathbf{x},t)\mathbf{u}(\mathbf{x},t)$, and the total energy $E^t(\mathbf{x},t) = \rho(\mathbf{x},t)e^t(\mathbf{x},t)$, are adopted as the unknowns: $\mathbf{U} = \{\rho, \mathbf{m}, E^t\}$. $\mathbf{u}$

represents the velocity, $\widehat{\mathbf{n}}(\mathbf{x})$ the boundary surface normal unit vector. $\mathbf{f}(\mathbf{U})$ represents convective fluxes, $\mathbf{g}(\mathbf{U})$ represents viscous fluxes and $\mathbf{h}(\mathbf{U})$ finally represents source terms.

As shown in equations 2.2, Navier–Stokes equations can be also expressed in differential form, again in an Eulerian framework:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{f}(\mathbf{U}) = \nabla \cdot \mathbf{g}(\mathbf{U}) + \mathbf{h}(\mathbf{U}) \tag{2.2}$$

The arrays of convective and viscous fluxes are defined as follows

$$\begin{aligned} \mathbf{f}(\mathbf{U}) &= \{\rho\mathbf{u}, \rho\mathbf{u} \otimes \mathbf{u} + P[\mathbf{I}], \mathbf{u}(E^t + P)\} \\ \mathbf{g}(\mathbf{U}) &= \{0, \tau, \tau \cdot \mathbf{u} + \mathbf{q}\} \end{aligned} \tag{2.3}$$

where P is the pressure field, τ is the stress tensor, $\mathbf{q}$ is the power exchanged by the conduction, and $[\mathbf{I}]$ is the 3x3 identity tensor. For what concerns the source term $\mathbf{h}(\mathbf{U})$, it can be used to account for the gravitational field effects and other kind of volume forces. However, it must be noted that since gravitational effects are negligible, the gravitational terms can be neglected. As shown in 3.6, a source term is also needed for the MRF formulation [46].

As previously introduced, some sort of modelization and hypotheses are necessary to close the problem. In particular, the pressure field $P(\mathbf{x}, t)$ must be expressed as a function of the chosen unknown ($\mathbf{U}$). For this work the gas is modeled as a Polytropic Ideal Gas (PIG) [125]. This is true for the majority of aeronautical cases and turbomachinery cases with the exception of particular applications, like ORCs (Organic Rankine Cycles) [59, 76, 79], that require different gas models in order to correctly describe complex phenomena like expansion shocks and mixed waves [66]. Usually this kind of phenomena are investigated using gas models like Van Der Walls [125] that can, from a mathematical point of view, catch phenomena typical of the dense gas region [66].

In this work the PIG model is implemented for what concerns the thermodynamics relations. This allows the pressure P and the temperature T fields to be expressed as function of the unknowns $\mathbf{U}$ as follows:

$$\begin{aligned} P &= (\gamma - 1)\left(E^t - \frac{1}{2}\rho|\mathbf{u}|^2\right) \\ T &= \frac{\left(E^t - \frac{1}{2}\rho|\mathbf{u}|^2\right)}{\rho C_V} \end{aligned} \tag{2.4}$$

where γ is the specific heats ratio $\gamma = \frac{C_P}{C_V}$, C_P and C_V are the molar heat capacity at constant pressure and at constant volume respectively.

Now, after the brief introduction of the thermodynamics model, the discussion is focused on the definition of the stress tensor τ. Using the well-known Newtonian fluid hypothesis [125] it is possible to express the stress tensor as proportional to the deformation velocity. Again, this is true for most of the aeronautical, turbomachinery and open rotors cases, in which the considered fluid is usually air or other kind of Newtonian gas (e.g. freon). However, there exists specific cases in which the fluid behavior is fully non-Newtonian (e.g. ketchup), requiring different kind of modelization of the

stress tensor. In this work the stress tensor is expressed as a function of the velocity field as follows:

$$\tau = \frac{\mu}{2}\left[(\nabla \otimes \mathbf{u}) + (\nabla \otimes \mathbf{u})^T\right] + \lambda \nabla \cdot \mathbf{u}\,[\mathbf{I}] \tag{2.5}$$

where $\mu(T)$ is the molecular dynamic viscosity and $\lambda(T)$ is the bulk viscosity. These two viscosity coefficients depend from the temperature T, however they can be kept constant during the simulation if the temperature range is limited. As an example, the value of the molecular dynamic viscosity for the air at a reference temperature of $T_0 = 291.15\,K$ is $\mu_0 = 1.827 \cdot 10^{-5}\,Pa\,s$. In this work it is also implemented a model that describes the molecular viscosity change with the local temperature. In particular, the Sutherland's formula is adopted:

$$\mu(T) = \mu_0 \frac{T_0 + C}{T + C}\left(\frac{T}{T_0}\right)^{\frac{3}{2}} \tag{2.6}$$

where C is a constant that has a particular value for the considered gas, e.g. for the air is $120\,K$. For what concerns the bulk viscosity λ, this is related to the molecular dynamic viscosity μ through the Stokes' condition [124]:

$$\lambda = \frac{3}{2}\,\mu \tag{2.7}$$

Another important modelization introduced in the equations is related to the power exchanged by conduction term, $\mathbf{q}$. Using the Fourier relation it is possible to express this term as a function of the temperature gradient, which, in turn, can be expressed as a function of the unknowns, as seen in 2.4. Using Fourier relation $\mathbf{q}$ is expressed as proportional to the temperature gradient through a constant K:

$$\mathbf{q}(T) = K\,\nabla T \tag{2.8}$$

K is a constant that is expressed as a function of the Prandtl number (Pr, which is usually 0.72), the molecular dynamic viscosity (μ), and the molar heat capacity at constant pressure (C_P):

$$K = \frac{\mu C_P}{Pr} \tag{2.9}$$

In order to completely close the Navier–Stokes equations system it is also necessary to introduce boundary conditions ($\mathbf{U}(\mathbf{x},t)|_{\mathbf{x}\in S} = \mathbf{U_S}(t)$) over the entire domain boundary. The implemented boundary conditions are briefly presented in 2.1.8. Furthermore, initial conditions ($\mathbf{U}(\mathbf{x},t)|_{t=0} = \mathbf{U_0}(\mathbf{x})$), consistent with boundary conditions, have to be specified inside the whole domain for the initial time. It is worth to say that, from a numerical point of view in particular, when performing steady-state simulations employing convergence acceleration techniques, initial conditions assume the meaning of an initial guess solution, without any particular physical meaning. Furthermore, time loses its physical meaning, and it's often called pseudo time at this point. This concepts are described in detail in 2.1.11, 2.1.10, 2.1.12.

At this point, the Navier–Stokes PDE system is completely closed and could be potentially solved from a mathematical point of view. From the numerical and computational point of view these are the equations that have to be solved when employing the DNS formulation. However, as briefly introduced before, solving DNS nowadays

is confined to particular problems that are still far from the typical aeronautical, turbomachinery and open rotors cases. This is basically due to the mesh discretization levels that DNS requires and thus the huge computational power that is necessary to complete the simulations. Different formulations are nowadays available to bypass this problem, such as RANS, LES, DES and DDES. In all of the listed strategies, the research field is still very active. Of course, bypassing the direct solution of Navier–Stokes equations introduces some kind of errors in the results. However, from an engineering point of view, a trade-off between simulations times and accuracy is mandatory. Furthermore, it must be noted that when simulating aeronautical components it is perfectly legit to neglect the detailed simulation of some kind of localized phenomena, provided that they don't affect the searched results. Nowadays RANS are the default approach for the analysis of both simple and complex aeronautical components such as wings, airplanes, and turbomachinery blades. (U)RANS provide both a good level of results accuracy and computational effort. LES represents the future for compressible viscous simulations. However, today LES are still a very active research field and LES simulations requires orders of magnitude more computational power to investigate the flow-field around a typical aeronautical component. With all of these aspects in mind, (U)RANS is the chosen formulation in this work to handle turbulence effects. Different turbulence models are implemented in the AeroX, as better shown in 2.1.7. Furthermore DES and DDES formulations are also implemented as extensions to improve results accuracy for unsteady cases.

Here the mathematical procedure required to obtain the (U)RANS equations from the original Navier–Stokes equations is not described and the reader is referred to [48, 120]. However, it is necessary to briefly discuss the main differences between (U)RANS and the original equations system. Different kind of (U)RANS approaches can be adopted, here we are using the turbulence models coupled with the Bussinesq hypothesis. With the Bussinesq hypothesis, all the modelization of the turbulence appear in RANS equations through a coefficient called turbulent viscosity or eddy viscosity μ_T. Without discussing all the mathematical details, the idea is basically to express the Reynolds stress tensor as function of the trace-less mean strain tensor and a term related to the turbulent kinetic energy. Despite the Bussinesq-based turbulence models here adopted, other kinds of (U)RANS approaches that don't use this hypothesis are available, e.g. Reynolds-Stress Models (RSM) that directly model the tress tensor. In any case the original Navier–Stokes system is not only modified in its terms since, with the exceptions of Mixing-Length based models, additional PDEs are required. As an example, Spalart–Allmaras model requires one additional PDE while SST model requires two additional PDEs. The implemented turbulence models are briefly introduced in 2.1.7.

Extending the Navier–Stokes solver to the (U)RANS formulation with the Bussinesq approach is straightforward, since after the averaging procedure, all the equations terms are basically the same except for some coefficients. In particular, the molecular dynamic viscosity is substituted by the effective dynamic viscosity μ_{EFF} that is computed as the sum of the molecular dynamic viscosity and the eddy viscosity:

$$\mu_{EFF}(\mathbf{x}, t) = \mu(\mathbf{x}, t) + \mu_T(\mathbf{x}, t) \tag{2.10}$$

The same concept is also extended to compute the effective bulk viscosity and the

effective conductivity:

$$\lambda_{EFF}(\mathbf{x},t) = \lambda(\mathbf{x},t) + \lambda_T(\mathbf{x},t) \tag{2.11}$$

$$K_{EFF} = K(\mathbf{x},t) + K_T(\mathbf{x},t) \tag{2.12}$$

where K_T, is computed with the same formula 2.9, but using the turbulent Prandtl number (usually $Pr_T = 0.8$). It is worth to notice that using the Bussinesq hypothesis and models like SA and SST, the additional equations are used to compute $\mu_T(\mathbf{x},t)$ from the additional turbulence model variables. Once a Navier–Stokes solver is implemented, the extension to (U)RANS formulation is straightforward. Thus the implemented solver has the ability to solve both the original Navier–Stokes system and the (U)RANS system with a chosen turbulence model. Anyway, a simplification to the Navier–Stokes system can be employed, the Euler system, when viscous effects are negligible for the purpose of a particular investigation, as shown in 2.1.2. This way, the solver can be used to solve Navier–Stokes equations, (U)RANS equations and Euler equations in a trade-off between results accuracy and computational costs.

2.1.2 Euler equations

Euler equations are obtained from Navier–Stokes equations by removing specific terms related to viscosity ($\lambda = 0$, $\mu = 0$) and conductivity ($k = 0$). This assumption is valid when simulating high Reynolds flow-fields where complex viscous phenomena like separations are not expected to happen. This is true for an aerodynamic body with a small angle of attack α. In fact, with such kind of flow-field, the bulk of viscous and conduction effects is confined in the small boundary layer attached to the wall.

Within an Eulerian framework the Euler equations are represented in integral conservative form as follows:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_V \mathbf{U}\,\mathrm{d}V + \oint_S \mathbf{f}(\mathbf{U})\cdot\widehat{\mathbf{n}}\,\mathrm{d}S = \int_V \mathbf{h}(\mathbf{U})\,\mathrm{d}V \tag{2.13}$$

or in the corresponding differential conservative form as follows:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla\cdot\mathbf{f}(\mathbf{U}) = \mathbf{h}(\mathbf{U}) \tag{2.14}$$

Basically, viscous terms $\mathbf{g}(\mathbf{U})$ disappear from the system. It is worth to notice that Euler equations represent a system of non-linear hyperbolic partial differential equations employed to model a compressible, non-viscous and non-conductive fluid. Another important differences between Euler and Navier–Stokes system regards boundary conditions. In fact with Navier–Stokes equations, Dirichlet boundary conditions for velocity are employed on solid walls in order to enforce the same velocity between the solid and the fluid (which is 0 for fixed walls or a specified vector when using MRF 3.6 or moving walls 2.2.5 in unsteady simulations). With Euler system, instead, only the normal component of the wall velocity has to be specified. In steady simulations without MRF, this is null and the corresponding boundary condition is often called "non-penetration" or "slip" boundary condition since no restrictions are enforced on the tangential component of the velocity vector over the wall. Anyway, when using MRF, unsteady cases with moving walls 2.2.5 or transpiration boundary conditions

2.2.7, a non-null normal component of the velocity vector can be prescribed. This is still representing a consistent non-penetration boundary conditions but accounting for wall velocity. Transpiration boundary conditions, discussed more in details in 2.2.7, represent a computationally cheap alternative to mesh deformation since it is possible to simulate wall displacements effects without actually change mesh points positions, thus without recomputing mesh metrics.

Euler formulation is implemented in the solver as an alternative to RANS equations. This is done for several reasons. First it is simple from the programmer point of view: given the implemented architecture of the solver (see 5), all is need is to avoid the computation of the neglected RANS terms. This has also the advantage of a reduced memory requirement since the allocation of many arrays (e.g. gradients and turbulence variables) can be avoided. This is translated in bigger cases that can be executed with the same available GPU memory and smaller time per iteration per cell (a measure of algorithm efficiency related to the underlying hardware architecture, see 6.2).

Of course when the flow-field exhibit complex viscous effects such as separations and interactions between the boundary layer and shocks, Euler equations must be discarded in favor of the more expensive but more accurate RANS equations. Furthermore Euler equations are capable to predict pressure distribution on the surface of the body quite well, allowing to compute lift, pressure and induced drag. However when viscous drag is required, even without complex viscous effects happening in the fluid domain, RANS equations have to be employed.

2.1.3 ALE formulation

Here, Navier–Stokes/RANS and Euler equations systems are modified with the introduction of the Arbitrary Lagrangian Eulerian (ALE) formulation [48, 127] required to handle unsteady cases with mesh deformation and steady/unsteady cases with moving reference of frame (see 3.6). Up to this point the Eulerian formulation was adopted to describe Navier–Stokes equations 2.1 and Euler equations 2.13, thanks to the use of a fixed control volume.

Consider a field $a(\mathbf{x}, t)$ within an Eulerian formulation. Here, $\mathbf{x}$ represents a fixed location in space, while t represents the time. If we chose a particular value for the location $\mathbf{x_1}$ and then we plot the value of $a(\mathbf{x}, t)|_{\mathbf{x}=\mathbf{x_1}}$, we would see the values of a of different fluid particles that are moving through $\mathbf{x_1}$ while time is passing. This is the usual approach in CFD when a steady simulation or an unsteady simulation with fixed boundaries is performed. Lagrangian formulation, instead, is mainly adopted for structural analyses. Within a Lagrangian description of the continuum, we consider a field $A(\mathbf{X}, t)$, here $\mathbf{X}$ is used to recognize a particular continuum particle that was at location $\mathbf{X}$ at $t = t_0$. Thus, if we want to see the evolution of the value of A for a particle $\mathbf{X}$ through time, we have to keep $\mathbf{X}$ constant and let the time t change. Obviously Eulerian and Lagrangian are nothing but different choice of the meaning of the spatial coordinates when defining a field. Thus, it is possible to interface the two descriptions when the particle trajectory function $\Phi(\mathbf{X}, t)$ is defined. $\Phi(\mathbf{X}, t)$ represents a reversible transformation. This function gives the spatial location of the particle $\mathbf{X}$ at time t. It is straightforward that the following relation holds:

$$a(\mathbf{x}, t)|_{\mathbf{x}=\Phi(\mathbf{X},t)} = A(\mathbf{X}, t) \tag{2.15}$$

The main advantages of the Lagrangian description are exploited in structural analyses where small displacements of the material particles have to be handled. In these kind of analyses particle tracking is important, alongside tracking of free surfaces and interfaces between different materials. The drawbacks of a Lagrangian formulation are represented by the inability to follow large distortions of the computational domain that would require re-meshing. Large displacements of the continuum are instead typical of fluid dynamics, where Eulerian description of the continuum is instead the default approach. It is straightforward that the main drawback of the Eulerian description within CFD is represented by an accuracy loss when tracking particles or investigating flow details. Finally, ALE formulation tries to exploit the main advantages of both Eulerian and Lagrangian Framework, minimizing their drawbacks.

With ALE formulation, the considered control volume, when mathematically describing the continuum, is not represented by a fixed control volume, neither by a control volume that is moving with the local flow velocity. Instead, a velocity vector $\mathbf{v}(\mathbf{x}, t)$, called material velocity, is defined all over the volume boundary and represents the velocity with which the volume boundary is changing. This velocity vector is in no way related to the velocity of the flow $\mathbf{u}$, since the control volume is allowed to evolve independently from the fluid that flows through it. The reader is referred to [48, 124, 127] for a detailed description of the mathematical point of view and the procedure required to obtain the Navier–Stokes equations for an arbitrary moving control volume. However, here it is necessary to highlight some particular findings related to the ALE framework.

Equation 2.16 represents the integral form of the Navier–Stokes equations system introduced in 2.1.1 modified for an arbitrary evolving control volume, for a compressible, viscous and conductive fluid:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{V(t)} \mathbf{U}\,\mathrm{d}V + \oint_{S(t)} [\mathbf{f}(\mathbf{U}) - \mathbf{U}\mathbf{v}]\cdot\widehat{\mathbf{n}}\,\mathrm{d}S - \oint_{S(t)} \mathbf{g}(\mathbf{U})\cdot\widehat{\mathbf{n}}\,\mathrm{d}S = \int_{V(t)} \mathbf{h}(\mathbf{U})\,\mathrm{d}V \quad (2.16)$$

It is easy to see that the main differences are represented by the control volume and control volume boundary dependency from time t and an additional term $\mathbf{U}\mathbf{v}\cdot\widehat{\mathbf{n}}$ inside the surface integral. Another important difference is that when considering a fixed control volume the time derivative of the first term of 2.16 can be directly applied to the argument of the integral, meaning:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{V} \mathbf{U}\,\mathrm{d}V = \int_{V} \frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\,\mathrm{d}V \quad (2.17)$$

but:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{V(t)} \mathbf{U}\,\mathrm{d}V \neq \int_{V(t)} \frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\,\mathrm{d}V \quad (2.18)$$

this is not possible anymore with ALE formulation since the integration domain is now a function of time itself. This aspect will be important when discretizing Navier–Stokes/RANS/Euler equations, for what concerns unsteady simulations in particular, as shown in 2.1.11 and 2.1.4.

As previously said, ALE formulation is implemented in the solver in order to handle two different kind of simulations: unsteady cases with moving and deforming grids, and cases with MRF. Of course ALE is also used in unsteady cases when MRF and mesh

deformation are coupled. MRF formulation and its exploitation of the ALE framework is described in 3.6, while the formulations required for mesh deformations in unsteady cases are described in 2.2.

2.1.4 Numerical discretization

Here, the aspects related to the numerical discretization of the spatial domain are briefly introduced. The choices are strictly related to the underlying OpenFOAM software framework for the mesh handling and pre-processing. The strategies for the numerical discretization of the Euler and RANS equations are basically the same adopted for the AeroFoam solver [127,136]. The Cell-Centered (CC) Finite-Volume (FV) formulation implemented in the solver is well known and the literature is full of material describing it in detail from both mathematical and numerical point of view. For a detailed description see [48, 94, 95, 127, 136]. Here just the fundamental concepts required for the next sections are showed.

The computational domain is decomposed in N_V polyhedral cells of volume Ω_i, delimited by a set of N_f faces $\Gamma_{ij} = \Omega_i \cap \Omega_j$, where i and j are the indexes of the i-th and j-th cells respectively. Thanks to the OpenFOAM pre-processing API and mesh conversion tools, hybrid unstructured meshes can be easily handled, allowing the solver to be compatible with the mesh generated by different software (e.g. snappyHexMesh, HyperMesh, GAMBIT, ICEM). There are different kind of grids. In structured grids the hexaedra cells are identified by a unique triplet (i, j, k) allowing to easily identify neighbor data. Faces are quadrilateral. Unstructured grids are more general purpose, and there is no such kind of simple connection between cells. Thus, in unstructured grids cells and faces have no particular ordering. Structured grids have advantages from computational point of view since the cell neighbor addressing intrinsically characterize the mesh, while in unstructured grids this addressing has to be saved in memory and recalled every time numerical cell residuals are assembled. However, unstructured grids can easily handle complex geometries. Since the aim of this work is to obtain a general purpose solver, compatible with complex geometries and different mesh generation software, unstructured grids are supported. Obviously, a CFD solver capable to handle unstructured grids is also directly capable to compute solutions over structured grids, eventually without computational efficiency advantages. Another kind of meshes are represented by hybrid meshes, where different cells feature different number of faces, thus different number of cells neighbors. AeroX is capable to handle also this kind of mesh. Hybrid meshes are useful especially in RANS cases where the internal domain could be discretized with tetrahedra, while the boundary layer near the wall is refined with hexahedra. With hybrid meshes, however, the computational domain could contain a mix of hexahedra, prisms, pyramids and thetraedra without any restriction. Unstructured and hybrid meshes allow the solver the be employed for very different kind of demands, however they pose performance issues, especially for GPU execution due to reduced memory coalescing and branch divergence. This is discussed in detail in 4.3.

Within a Finite-Volume Cell-Centered framework, the solution field $\mathbf{U}(\mathbf{x}, t)$ of the unknown conservative form variables is translated in an array of cell's solutions $\mathbf{U}_i^h$. With this notation, i refers the i-th cell, while h represents the h-th time index. As explained in [48] temporal discretization and spatial discretization can be carried out

separately. This way it is possible to combine different temporal and spatial discretization schemes without any particular problem.

By applying the numerical spatial discretization to the Navier–Stokes/RANS equations 2.16 the initial Partial Differential Equations (PDE) system is translated to a simpler Ordinary Differential Equation (ODE) system. The following equations shows the system after the numerical discretization:

$$\frac{\mathrm{d}(\Omega_i(t)\mathbf{U}_i(t))}{\mathrm{d}t} = -\sum_{j=1}^{n} \mathbf{\Gamma}_{ij}(t)(\mathbf{F_{ij}}(t) - \mathbf{G}_{ij}(t)) + \Omega_i \mathbf{H}_{ij}(t) = \mathbf{R}_i(t) \tag{2.19}$$

where, since the temporal numerical discretization has not been employed yet, the explicit time t dependency is highlighted. For each face ij that cell i has in common with its neighbor cell j, convective fluxes $\mathbf{F}_{ij}(t)$ and viscous fluxes $\mathbf{G}_{ij}(t)$ have to be computed in order to assembly residuals. $\Gamma_{ij}(t)$ represents the product between the ij face area $S_{ij}(t)$ and the $\widehat{\mathbf{n}}_{ij}(t)$ unit normal vector of the face ij. $\Omega_i(t)$ is the cell volume. It must be underlined that geometrical properties, like cell volume and faces depend from time t when unsteady simulations with mesh deformation are performed, employing ALE framework. Furthermore, as explained in 2.1.3, in the first term of 2.19 the temporal derivative is applied not on the conservative form unknowns $\mathbf{U}_i(t)$ but on the product between the unknowns and the volume. This is required in order to obtain a consistent formulation valid for unsteady simulations with mesh deformation.

When inviscid simulations, i.e. Euler simulations, are performed, equation 2.19 loses the viscous fluxes term leading to:

$$\frac{\mathrm{d}(\Omega_i(t)\mathbf{U}_i(t))}{\mathrm{d}t} = -\sum_{j=1}^{n} \mathbf{\Gamma}_{ij}(t)\mathbf{F_{ij}}(t) + \Omega_i(t) \mathbf{H}_{ij}(t) = \mathbf{R}_i(t) \tag{2.20}$$

this way it is easy to understand that if a NS/(U)RANS solver is implemented, performing inviscid simulations just means avoiding the computation of $\mathbf{G}_{ij}(t)$ (and of course choosing consistent boundary and initial conditions).

The next step is the discretization of the time derivative. This problem is handled in 2.1.10 and 2.1.11. As mentioned, the aim of this work is the implementation of an explicit solver, thus an explicit time discretization has to be employed. Euler discretization can be easily implemented. This is the most simple time discretization and the less robust one. However, as presented later in 2.1.10, more complex Runge–Kutta (RK) [48] time schemes can be easily implemented, improving the robustness of the formulation and allowing better damping properties, which are crucial for steady-state simulations. Furthermore, as presented later in 2.1.10 and 2.1.11, due to CFL limitations, unsteady cases are handled through convergence acceleration techniques and Dual Time Stepping (DTS) formulation. Also, it is understood that when a (U)RANS simulation is performed, the turbulence model equations (if required) have to be discretized too, obtaining a set of equations similar to 2.19, with temporal term, convective and diffusive fluxes and source terms. The implemented turbulence models and their equations are briefly described in 2.1.7. Finally, it is reminded that Euler/NS/(U)RANS equations have to be coupled with opportunely defined initial and boundary conditions. Boundary conditions are briefly described in 2.1.8. For what concerns initial conditions they could be just guess solution fields if a steady-state solution is the aim of the

simulation or opportunely defined true initial conditions if a time accurate simulation is performed.

2.1.5 Convective fluxes

Different well-known numerical schemes for convective fluxes are implemented in the solver, such as Roe [94, 95, 136], AUSM+ [97] and CUSP [145] fluxes. They represent first order upwind schemes that are required to guarantee stability. However, this comes at the price of a low-order spatial discretization. Upwind schemes help prevent oscillations near shocks, but could degrade the overall solution accuracy where not required. Thus, in order to increase the spatial discretization order of the solution up to the second order wherever is possible inside the fluid domain, an high resolution strategy is also implemented and coupled with the numerical fluxes schemes. The main idea is to blend the first order upwind flux with a second order centered flux through the use of a limiter. Entropy fix is also required for Roe fluxes in order to avoid non-physical solutions. In this work the entropy fix by Harten and Hyman [117] is adopted. All the schemes are opportunely modified in order to take into account ALE velocity to correctly handle unsteady cases with mesh deformation and with moving reference of frames (MRF). The strategy adopted to achieve second order accuracy is based on the work [136] and uses the concept of extended cells. Basically for each face a set of 4 cells is adopted to compute the convective fluxes. Two cells are directly related to the face. The two additional cells are found alongside the face normal direction and are neighbor of the two primary cells. In this work a Van Leer limiter [95] is adopted.

For a more detailed view of the analytic aspects of the implemented strategies see [136] as here this is beyond the purposes of the work. Details regarding the implementation of convective fluxes can be found in 5.3.2.

2.1.6 Gradients computation

These schemes are required only for the solution of the NS/(U)RANS equations since Euler equations only require convective fluxes discussed in 2.1.5. A simple and robust Green–Gauss scheme is adopted to compute the gradients of different quantities required for both original NS equations and additional turbulence models equations. Let's consider the computation of the gradient of a scalar quantity Φ on cell k. This can be computed as follows:

$$\nabla\Phi_k \simeq \frac{1}{\Omega_k}\sum_{j}^{neigh}\left(W_{kj}\Phi_k + (1-W_{kj})\Phi_j\right)S_{kj}\widehat{\mathbf{n}}_{kj} \tag{2.21}$$

where Ω_k is the k-th cell volume, S_{kj} is the area of face kj belonging to both cells k and j, $\widehat{\mathbf{n}}_{kj}$ is the face unitary normal vector. W_{kj} is the weight related to the couple of cells that shares the face kj. The weight can be computed in different ways. In this work this is computed considering the distance between the cell center and the face.

Details regarding the implementation of automatic wall treatment can be found in 5.3.5.

2.1.7 Turbulence models

The implemented turbulence models are here briefly presented. These models are required when solving Navier–Stokes equations in the form of (U)RANS equations, thus introducing the modelization of turbulence effects. In particular, the implemented models require the Bussinesq approximation [120]. This basically means that turbulence effects appear in (U)RANS equations in the form of an additional viscosity coefficient, the turbulent (or eddy) viscosity, that, added to the molecular viscosity, gives the effective viscosity as explained with equation 2.10. The detailed description of the implemented turbulence models is beyond the purposes of this work as it can be easily found on publicly available papers. The two implemented turbulence models are represented by Spalart–Allmaras and $k-\omega$ SST. Details regarding Spalart–Allamaras model can be found in [140], while for SST see [107, 108]. Furthermore, in order to improve results accuracy for unsteady cases when the mesh is particularly refined, the so-called DES and DDES variants of the models are also implemented [108,142]. A great source of information regarding the possible modifications of these turbulence models is also [33]. As previously mentioned Spalart–Allmaras turbulence model is specifically designed for the simulation of attached flows when investigating aerodynamic bodies. SST is obtained blending $k-\omega$ and $k-\varepsilon$ turbulence models and opportunely tuned to be a general purpose model, accurate for both near-wall and free shear simulations. AeroX couples these turbulence models with an automatic wall treatment (see 2.1.9) in order to handle different levels of near-wall discretizations.

2.1.8 Boundary conditions

The detailed description of the implemented boundary conditions is beyond the purposes of this work. As this is a general purpose Eluer/(U)RANS solver with turbomachinery and open rotors extensions, specific boundary conditions are required. Furthermore, the solver is designed such that the user prescribes the values of velocity, temperature and pressure over the boundary and it's the solver job to convert them into conservative form variables and to decide which of the prescribed values use, based on the physical properties of the boundary. In classical aeronautical cases with open domain a far-field is present. In this case different boundary conditions can be adopted based on the Mach number and the fact that a particular boundary is considered inlet or outlet. Over a subsonic-inlet boundary, zero-gradient conditions are enforced on pressure while temperature and velocity are interpolated using user-prescribed and internal values. On subsonic-outlet boundaries instead the pressure is enforced using interpolation with user-prescribed data and internal cells solutions, while zero-gradient conditions are enforced for what concerns the velocity and temperature. Finally a strategy that can automatically switch between inlet on outlet conditions is also implemented, based on local characteristics variables values, as explained in [75, 136]. The advantages of this strategy are represented by the fact that it is intrinsically based on the physical aspects of the boundary and the fact that it leverages the user the decision of what is effectively inlet and outlet. Far-field boundary conditions are also adopted for open rotors cases. Instead, for turbomachinery simulations it is usually needed to enforce total pressure, total temperature and velocity direction on the inlet boundary. This is useful when investigating turbomachinery performances. This boundary condition is discussed in

details in 3.9. Wall boundary conditions for viscous simulations are briefly discussed in 2.1.9. For inviscid simulations slip (or non-penetration) boundary conditions have to be enforced on walls. In this work boundary conditions are enforced using the ghost cell approach, meaning that a virtual cell is build with conservative form variables such to enforce the correct boundary conditions on the boundary face of the aerodynamic mesh. One important advantage of the ghost cell approach is the possibility to directly use the same algorithms for fluxes computations adopted for internal faces once the virtual cell is built. A detailed description of the implemented ghost cell approach can be found in [136].

Details regarding the implementation of boundary conditions can be found in 5.3.4.

2.1.9 Wall treatment

Here the description of the implemented wall treatment is provided. When performing viscous simulations with (U)RANS turbulence models, usually two different approaches for the near-wall discretization can be adopted. The first approach relies on the capability of the turbulence model to accurately predict the flow properties inside the viscous sublayer, which is the nearest turbulence layer to the true wall boundary. This means that the turbulence models have to consistently reconstruct the flow velocity up to very small y^+ values, in the order of 1 ($y^+ < 5$ [120]). Otherwise it is possible to use wall functions to predict the flow properties inside the logarithmic region, for higher values of y^+, usually for $y^+ > 30$ [120]. Instead, in this work an "automatic" wall treatment based on a blended approach is implemented. Basically both viscous sublayer and log region formulations are implemented and the two contributions are opportunely blended based on the y^+ estimation. Thanks to an opportunely defined blending function it is possible to obtain high accuracy also inside the buffer layer (for y^+ values between 5 and 30), between the viscous sublayer and the log region. The approach implemented is based on [88, 121]. Using a Newton–Raphson algorithm, opportunely optimized for GPU executions, for each wall face at each pseudo time iteration, the wall values, like y^+ and u_τ, are estimated. These values are then used in the next pseudo time iteration to compute viscous fluxes when enforcing boundary conditions on wall boundaries. This way when pseudo time convergence is reached the wall boundary condition is enforced with the correct y^+ value, raging from y^+ in viscous sublayer to log region, without requiring the user to actually performing the switch between the two formulations.

Details regarding the implementation of automatic wall treatment can be found in 5.3.3.

2.1.10 Convergence acceleration techniques

There are advantages and drawbacks in using explicit algorithms for the numerical solution of a problem, both regarding computational aspects. One of the main advantages of explicit formulations is the fact that it is easily implementable and parallelizable. In fact, since the solution update depends only upon the previously stored solution, each domain cell can be updated independently and concurrently. Thus, this job can be easily distributed among hundreds or thousands of GPU cores. The same advantage is also exploitable in CPU-based shared and distributed memory architectures. As a practical example, a for loop over the cells can be easily parallelized on a multi-core CPU with

multi-threading using OpenMP, or in the case of this work using OpenCL and by choosing the CPU as the computational device (see 4). In distributed memory systems, where multiple computational nodes are connected together with high-bandwidth low-latency connections, the easiest way to distribute work is by decomposing the computational domain in multiple sub-domains, distributed among the nodes, and by reducing communications to sub-domain boundaries only. An important trend is the exploitation of hybrid CPU-based parallel architectures A.4.4 where each computational node is part of a cluster architecture in a distributed memory fashion, while inside each node the work is distributed using a shared-memory work-flow among cores. Anyway, it is easier to reach high computationally efficiency in parallel architectures by implementing explicit algorithms rather than parallelizing linear system solution algorithms required by implicit formulations. Another advantage of explicit algorithms is represented by the lower memory requirements due to the fact that there is no matrix to be stored like in implicit formulations. Obviously all of the advantages of explicit algorithms come with a price. The main drawback in fact, is represented by their slow convergence rate, especially if a steady-state solution is searched. This is due to the fact that when using explicit algorithms, CFL represents a quite restrictive limitation for the majority of aeronautical aeroelastic simulations due to the frequency content of interest. Basically the physical time-step is limited to values that are orders of magnitude lower than what is required for the majority of cases. However, it must be noted that this situation is inverted with peculiar unsteady simulations in which very small physical time-steps are mandatory, leading explicit algorithms to be preferable over implicit algorithms. This is the case of acoustics and high speed impacts where with explicit algorithms the maximum allowed physical time step that satisfy the CFL requirements is also nearly in the order of the time scales of interests. This leads to smaller simulation times with respect to implicit algorithms thanks to smaller time per iteration per cell due to the avoided linear system solution. However, usually in aeronautical aerodynamic and aeroelastic cases a steady-state solution is required, and in unsteady simulations the time scales of interest are orders of magnitude higher than the maximum allowable time step.

As previously mentioned, in this work an explicit formulation is chosen because of its efficient parallelization on GPU architectures and because of its limited memory requirements that allows the solver to be executed on cheap gaming GPUs. However, in order to reach implict-like residuals convergence rates, convergence acceleration techniques are required. CFL requirements must be satisfied in order to have a stable formulation but with some techniques they can be somehow bypassed.

Different convergence acceleration techniques are implemented in the solver and combined together with the aim of the total computational time reduction by increasing the residual damping rate. It must be noted that these techniques are implemented as algorithms that have to be executed at each explicit iteration. Thus they obviously increase the time per iteration per cell (see 6.2) with respect to a simple explicit aerodynamic solver. The point here is the reduction of the product between the iteration cost and the total number of iterations required to reach convergence, i.e. the reduction of the total computational time required to reach a steady-state converged solution.

First of all, Local Time Stepping (LTS) [48] is described. This is the core technique around which the other strategies are built. The idea behind LTS is very simple: CFL requirements apply to each cell and not to the entire domain. Thus, since different cells

have different geometrical and flow properties, different cells could potentially advance with different time step values. Obviously having different cell advancing with different values of time step would break the physical meaning of the time. However, from the CFL point of view, no rules are violated. Usually in unsteady simulations with explicit formulations what is done is computing the time step value from each cell and then take the most conservative over the discretized domain and use it to advance the solution in time. This way CFL is still respected everywhere and different cells advance in time consistently, eventually with a time step smaller than what they could potentially offer. With LTS instead, different cells advance in time with their maximum CFL-allowed time step. Thus, the physical meaning of the "time" is lost. At this point, the time step computed with LTS is usually called "pseudo time step" and indicated with τ instead of t to underline that it has only a numerical meaning and not a physical meaning anymore. Inside each cell, the pseudo time-step is computed as follows:

$$\Delta\tau = C\frac{\Delta x_i}{|\mathbf{u}_i| + c_i + K\nu_i^{EFF}/\Delta x_i} \tag{2.22}$$

where i represents the i-th cell index, C is a user-defined constant, Δx_i is a measure of the cell length (usually the cubic root of the cell volume or the cell volume to surface ratio), $\mathbf{u}_i$ is the local flow velocity, c_i is the local sound speed, K is a constant and ν_i^{EFF} is the local effective kinematic viscosity. As mentioned, LTS cannot be used for unsteady simulations because of the time step value inconsistency between different cells. However, when the goal is a steady simulation, LTS can be adopted with no problem. In fact, considering equation 2.19, if the aim of the simulation is to reach steady state, the real goal is to damp residuals $\mathbf{R}_i(\tau)$ in each cell until $\frac{\Delta\mathbf{U}_i}{\Delta\tau} = 0$ that is the definition of steady-state. When, at convergence, this condition is satisfied in every cell, the steady state is reached. Thus, with LTS the idea is to reach a steady-state condition that satisfy the steady form of Euler/NS/RANS discretized equations without regarding of the (pseudo) time step value of each cell. LTS provides convergence advantages since each cell is allowed to advance with its maximum allowed time step value. Despite turbomachinery cases, in a typical aeronautical case (wings, airplanes), usually the aerodynamic object is immersed in the fluid domain, with an external far-field. The farfield is located numerous airfoil/wing/airplane length away from the object. The goal is to investigate the flow field around the object and on its walls, while what happens near the farfield has negligible interest. Thus, the mesh is well discretized near the object and much coarser near the farfield. Given the geometrical and flow properties, the cells near the object require smaller time step values with respect to the cells near the farfield. Thus, with LTS farfield cells are allowed to advance with greater pseudo time steps, effectively damping residual faster, favoring steady-state convergence.

LTS has another important advantage, especially with GPU architectures. By allowing the cells to just use their pseudo time step, a global reduction operation to find the most restrictive value is avoided. Pseudo time step computation is easily distributed on GPU cores since the same expression 2.22 has to be evaluated independently (see 5.3.1.

With LTS pseudo time has no physical meaning but time discretization is anyway required. This can be performed using the simplest explicit formulation, i.e. Explicit Euler (EE). However, first and second order multi-stage Runge–Kutta (RK) schemes can also be employed, as showed in [48]. With respect to EE, RK requires multiple stages. This practically means intermediate residual evaluations inside each pseudo

time step. This leads to greater computational requirements with respect to EE. However, RK schemes guarantee higher CFL values, greater than 1, as explained in detail in [48]. At each stage of the RK scheme, intermediate residuals are evaluated with different weights, opportunely tuned to improve the damping properties and thus convergence and robustness. RK schemes can be also extended to second order. It is worth to remind that both EE and RK are directly applied to the solution of the single cell equations, thus they can be directly coupled with LTS to further help damping residuals and accelerate convergence to steady-state while maintaining the efficient work distribution (parallelization) of an explicit scheme. The general RK scheme is here presented:

$$\begin{aligned}
\mathbf{U}_i^{k+1,(0)} &= \mathbf{U}_i^k \\
\mathbf{U}_i^{k+1,(1)} &= \mathbf{U}_i^{k+1,(0)} - \alpha_1 \frac{\Delta\tau_i}{\Omega_i}\mathbf{R}_i^{k,(0)} \\
\mathbf{U}_i^{k+1,(2)} &= \mathbf{U}_i^{k+1,(0)} - \alpha_2 \frac{\Delta\tau_i}{\Omega_i}\mathbf{R}_i^{k,(1)} \\
&\vdots \\
\mathbf{U}_i^{k+1} &= \mathbf{U}_i^{k+1,(0)} - \alpha_m \frac{\Delta\tau_i}{\Omega_i}\mathbf{R}_i^{k,(m-1)}
\end{aligned} \tag{2.23}$$

where in $\mathbf{U}_i^{k+1,(z)}$, i represents the i-th cell index, k the pseudo time iteration and z the z-th RK stage. The RK stage coefficients α_g are opportunely tuned for accuracy and damping, as showed in [48]. Finally the residuals R_i^{z-1} are evaluated at each RK stage. RK schemes are also memory efficient since from stage to stage residuals are overwritten and only the the solution of the previous pseudo time has to be kept stored and available at each RK stage.

The previous described techniques help damping residuals with algorithms that applies on the single computational cell. Both LTS and RK schemes are implemented in AeroX since they are well suited for the GPU multi-core SIMD parallelism. However, two other convergence acceleration techniques are implemented and directly applied to residuals. These are represented by a modified Residual Smoothing (RS) and a simplified Multi-Grid (MG) approach, an adaptation to unstructured grids of the formulation originally proposed by Denton. During each explicit aerodynamic iteration, before the cell solution update, residuals are assembled considering, as presented in equation 2.19, faces convective fluxes, viscous convective fluxes and finally source terms if present. At this point, when cells residuals are computed and ready to be used to update the cells solution, MG and RS are applied on them. After the execution of RS and MG algorithms, residuals can be used to update cells solutions as usual. Roughly speaking, the idea behind the CFL restriction is that in each cell the time step value has to be small enough that from one time step to the next one, the information is not allowed to travel for an amount of space bigger than the cell size itself. The idea behind both RS and MG is "bypassing" CFL restriction by smoothing each cell residuals with the residuals of the surrounding cells. This way, residuals are damped faster thanks to the smoothing effects provided by the residuals information propagation through the domain. What happens to the residuals when RS and MG algorithm are employed is

basically represented by the following expression:

$$\mathbf{R}_i^{k+1,smooth} = \alpha_i \mathbf{R}_i^{k+1,orig} + \beta_i \mathbf{C}_i(\mathbf{R}_0^{k+1,orig}, ..., \mathbf{R}_{NV}^{k+1,orig}) \tag{2.24}$$

Basically the smoothed residuals $\mathbf{R}_i^{k+1,smooth}$ are used to update the cells solutions instead of the original residuals $\mathbf{R}_i^{k+1,orig}$ obtained after the other LTS and RK acceleration techniques have been applied. The term $\mathbf{C}_i(\mathbf{R}_0^{k+1,orig}, ..., \mathbf{R}_{NV}^{k+1,orig})$ is computed by applying the smoothing algorithm behind the RS or MG strategy. Depending from the underlying algorithm, coefficients α_i and β_i are opportunely tuned to weight the contribution of the original cell residuals and the smoothed residuals.

The implemented RS algorithm is quite simple and easily parallelizzable, especially for the GPU architecture. The computation of $\mathbf{C}_i$ is performed in a loop of multiple stages. First of all for each cell, neighbor cells residuals are agglomerated and stored. Thus, $\mathbf{C}_i$ depends only upon the immediately cell neighbors residuals. Next, expression 2.24 is evaluated providing the smoothed cells residuals that are used to update the original cells residuals. At this point one stage of the RS algorithm is performed. However, multiple RS stages can be performed, by simply applying again the just presented RS algorithm, evaluating $\mathbf{C}_i$ with the just smoothed residuals. With just one RS stage smoothing effects are confined to the cell immediately neighbor region. More RS stages means more smoothing effect but also an increase on the computational effort for each explicit iteration. Usually 2-3 stages provide a good trade-off between smoothing effects and computational requirements. This is all repeated the next explicit iteration.

The implemented MG algorithm is more effective than the adopted RS strategy thanks to the fact that cells residuals are somehow propagated to a more extensive region of the computational domain. This also means that each cell residual is modified with the residual contribution of a large number of cells located around the cell itself. This not only include the cell's neighbor but also, in smaller magnitude, more distant cells. Thus, despite the RS strategy, with a single stage of MG, the cell residual is smoothed thanks to a term $\mathbf{C}_i$ that depends from an higher number of surrounding cells. With MG, from the original "finer" mesh multiple levels of "coarser" meshes are obtained. Each level is obtained agglomerating the cells of the underlying level. However, each level is not saved as a complete mesh metrics structure, since the implemented simplified MG algorithm works just on residuals. This is different from the usual MG approach where at each MG level the Euler/NS/(U)RANS equations are effectively solved. With the implemented approach instead, only the addressing between each level coarse cell and the underlying level finer cells that form the coarse cell is required. What happens is basically that residuals from the finer mesh cells are agglomerated to compute the residuals of the cells of the next mesh level. This is done sequentially for all the available mesh levels. Then, for each of the original finer mesh cell i, the term $\mathbf{C}_i$ is computed by opportunely weighting the residuals of the coarser cells that includes the cell i at the different levels. The main idea is that transients are dispersed more quickly on the coarser levels, as larger time steps are allowed, while retaining the spatial accuracy of the finest [49, 91].

It is reminded that all the briefly presented convergence acceleration techniques are useful only to accelerate steady state convergence. However thanks to Dual Time Stepping (DTS) formulation described in 2.1.11 it is possible to exploit these techniques also for unsteady simulations.

The computational aspects behind the GPU implementation of the RK and MG convergence acceleration techniques are discussed in 5.3.8.

2.1.11 Temporal discretization for unsteady simulations

Here the Dual Time Stepping [48] formulation is described. This formulation allows the solver to perform unsteady simulations while retaining all the convergence acceleration techniques previously described in 2.1.10. As previously mentioned, a global physical time stepping formulation could be implemented to perform unsteady simulations, but since the very restrictive CFL requirements it would not be efficient for the aeronautical aeroelastic cases of interest of this work.

As the name suggest, the idea behind DTS is to have two different concepts of time inside the formulation. One is the pseudo time already presented when describing LTS in 2.1.10, and the other is of course the physical time used to describe the evolution of the unsteady phenomenon. With DTS both the physical time derivative and the pseudo time derivative are present in the Euler/NS/(U)RANS/Turbulence equations. Basically, with DTS, converging to the next physical time step is translated to converging to a steady-state solution characterized by a new term in the Euler/NS/(U)RANS/Turbulence equations related to the physical time derivative of the conservative form unknown. This way, the goal of the simulation is again to damp residuals, thus reaching a steady-state solution with respect to the pseudo time derivative. However, with respect to an effective steady-state simulation, this time residuals are defined differently, with the addition of the new physical unsteady term. Equation 2.25 shows the modified version of the equation 2.19 with the new additional term added to recovery the physical time derivative:

$$\frac{\Delta(\Omega_i \mathbf{U}_i)^{(k,h)}}{\Delta\tau_i^k} + \frac{\Delta(\Omega_i \mathbf{U}_i)^{(k,h)}}{\Delta t} = \mathbf{R}_i^{(k,h)} \tag{2.25}$$

Indexes require explanation. i represents, as usual, the i-th cell index. Index k represents the pseudo time iteration, used for aerodynamic convergence to the next physical time step. Index h represents the physical time iteration. Δt is kept fixed during the simulation, thus there is no dependency from h and of course, differently from the pseudo time step, all the cells have the same user-prescribed value of Δt. It is underlined that the pseudo time step $\Delta\tau_i^k$ depends both from the geometrical properties of the i-th cell and flow properties at the k-th iteration, due to LTS algorithm (see 2.22. At each physical time step h the aim is to find the new value of the solution, i.e. $\mathbf{U}_i^{h+1}$, knowing the solution of the previous physical time step $\mathbf{U}_i^h$ with an iterative procedure in pseudo time that produces different potential solutions $\mathbf{U}_i^{(k+1,h)}$ before convergence. It must be noted that when performing unsteady simulations with mesh deformation Ω_i^{h+1} and Ω_i^h are usually different. Each of the potential solution is obtained by assembling residuals $\mathbf{R}_i^{(k,h)}$ that are computed from the solution at the previous iteration $\mathbf{U}_i^{(k,h)}$. Equation 2.25 can be expressed with all the explicit terms (supposing for simplicity the use of Explicit Euler for both physical time and pseudo time discretization) as follows:

$$\frac{\Omega_i^{h+1}\mathbf{U}_i^{(k+1,h+1)}}{\Delta\tau_i^{k+1}} - \frac{\Omega_i^{h+1}\mathbf{U}_i^{(k,h+1)}}{\Delta\tau_i^{k+1}} + \frac{\Omega_i^{h+1}\mathbf{U}_i^{(k+1,h+1)}}{\Delta t} - \frac{\Omega_i^{h}\mathbf{U}_i^{h}}{\Delta t} = \mathbf{R}_i^{(k,h)} \tag{2.26}$$

then, the equation is rewritten in order to explicitly identify the searched term at each pseudo time iteration:

$$\frac{\Omega_i^{h+1}\mathbf{U}_i^{(k+1,h+1)}}{\Delta\tau_i^{k+1}}\left(1+\frac{\Delta\tau_i^{k+1}}{\Delta t}\right)=\mathbf{R}_i^{(k,h)}+\frac{\Omega_i^{h+1}\mathbf{U}_i^{(k,h+1)}}{\Delta\tau_i^{k+1}}+\frac{\Omega_i^{h}\mathbf{U}_i^{h}}{\Delta t} \tag{2.27}$$

now subtracting the term $\frac{\Omega_i^{h+1}\mathbf{U}_i^{(k,h+1)}}{\Delta\tau_i^{k+1}}\left(1+\frac{\Delta\tau_i^{k+1}}{\Delta t}\right)$ to both the left hand side and the right hand side it is possible to write:

$$\frac{\Omega_i^{h+1}\mathbf{U}_i^{(k+1,h+1)}-\Omega_i^{h+1}\mathbf{U}_i^{(k,h+1)}}{\Delta\tau_i^{k+1}}\left(1+\frac{\Delta\tau_i^{k+1}}{\Delta t}\right)=\mathbf{R}_i^{(k,h)}+\frac{\Omega_i^{h+1}\mathbf{U}_i^{(k,h+1)}}{\Delta t}+\frac{\Omega_i^{h}\mathbf{U}_i^{h}}{\Delta t} \tag{2.28}$$

$$\frac{\Omega_i^{h+1}\mathbf{U}_i^{(k+1,h+1)}-\Omega_i^{h+1}\mathbf{U}_i^{(k,h+1)}}{\Delta\tau_i^{k+1}}\left(1+\frac{\Delta\tau_i^{k+1}}{\Delta t}\right)=\hat{\mathbf{R}}_i^{(k,h)}$$

this way a new definition of residual $\hat{\mathbf{R}}_i^{(k,h)}$ is provided. The advantage of the DTS formulation is that the same convergence acceleration techniques used to damp residuals for steady-state simulations, can now be exploited to converge to the next physical time step by damping the just defined residuals. A new source term is however added that takes into account the solution of the previous physical time step. Furthermore, the value of the physical time step Δt is independent from the CFL constraints since this is a problem handled with LTS regarding the pseudo time step value. This way, the user can choose the ad-hoc physical time step based on the time scales of the phenomenon under investigation, in a similar way with implicit time stepping schemes. Another important aspects is that, as showed in [106], the here presented DTS formulation is not afflicted by convergence problems when the physical time step and the pseudo time step are of the same order of magnitude. Thus, small time steps can also be employed without particular convergence problems. In the unsteady test case under investigation in this work, usually about $100-2000$ pseudo time step iterations are required to guarantee aerodynamic convergence between two physical time steps.

2.1.12 Aerodynamic steady analyses

Here the general aspects of the procedure implemented to perform steady-state simulations without mesh deformation is described. Steady analyses are very important for academic and industrial cases both when the flow fields around the aerodynamic component is supposed to be steady and when, instead, the solution is intrinsically unsteady (e.g. oscillatory) but the aim is to obtain an average solution. In fact, it must be noted that turbulence is intrinsically a 3D unsteady phenomena, however, thanks to (U)RANS formulation, an averaging operation is performed, allowing to obtain reasonable results for both 2D and 3D steady cases. Beside the case when a steady solution with fixed geometry is the aim of the simulation, steady computations are also required to initialize trim simulations (see 2.2.8 or to provide initial conditions for subsequent unsteady analyses with fixed walls (see 2.1.13 or moving walls (see 2.2.9 and 2.2.10). In fact, thanks to the previously described formulations and strategies like DTS and

convergence acceleration techniques, the steady-state solver provides also the core for the purely aerodynamic convergence required by the other mentioned kind of analyses.

Here the procedure adopted to perform a steady analysis with fixed geometry is briefly presented with the focus on the connection between the previously described Euler/NS/(U)RANS numerical formulations.

1. First of all, the user-provided initial guess solution is read alongside the mesh;
2. Mesh pre-processing is performed to compute mesh metrics (cell volumes, cell center, face areas and normals, etc);
3. The first step in the aerodynamic convergence algorithm is the computation of the cells pseudo time steps, using the LTS formulation 2.1.10 and code 5.3.1;
4. Internal faces convective fluxes are computed using one of the described numerical fluxes (see 2.1.5 and code 5.3.2). If it is a (U)RANS simulations this involves also turbulence models equations convective fluxes computations, see 2.1.7. If it is a rotating case using MRF (see 3.6 and code 5.3.10), ALE formulation is also employed to adjust convective fluxes with consistent faces velocities;
5. Now it's the turn of viscous fluxes, see 5.3.5. If it is an inviscid (Euler) simulation, viscous fluxes are not computed, saving computational time. If it is a laminar Navier–Stokes simulation viscous fluxes are computed considering just the molecular dynamic viscosity μ. If instead is a RANS simulation, the effective dynamic viscosity μ_{EFF} is computed from the sum of the molecular viscosity μ and the turbulent viscosity μ_T, thanks to the Bussinesq hypothesis. The latter is computed from the turbulence models described in 2.1.7. Eventually, Sutherland formula, described in 2.1.1 can be employed to update the value of μ from the temperature field;
6. Boundary conditions convective and viscous fluxes are computed (see 2.1.8 and code 5.3.4), based on the particular kind of boundary. If it is a RANS simulation, this is performed also for turbulence models equations;
7. Convective and viscous fluxes are added together in order to provide a single numerical flux to be used for the cell solution residuals update (see strategy 5.3.5);
8. Cells residuals are computed by adding up pseudo time derivative terms and numerical fluxes (see code 5.3.6);
9. Source terms are computed. This is usually the case with turbulence models and when MRF is employed for rotating cases;
10. At this point residuals computation is complete and it would be possible to update the cells solutions. However, convergence acceleration techniques, such as MG and RS algorithms (see 2.1.10 and 5.3.8), if active, are now invoked in order to help dissipating residuals;
11. Knowing residuals, pseudo time steps, cells volumes and previous iteration solution it is finally possible to update the cell solution (see code 5.3.9). This involves both conservative form variables and, if not an inviscid simulation, turbulence models variables;

It is worth to notice that since the aim of this simulation is obtaining a steady-state flow field, the "initial" conditions are in fact user-prescribed initial guess conditions. For typical aeronautical cases, values of pressure, temperature and velocity have to be specified over the farfield, using one of the described boundary conditions in 2.1.8. On walls, depending if an inviscid or viscous analysis has to be performed, slip conditions or non-slip conditions have to be employed, eventually with automatic wall treatment described in 2.1.9 (see also code 5.3.3).

2.1.13 Aerodynamic unsteady analyses

Unsteady analyses with fixed geometry are performed when the focus is posed on the evolution of some peculiar aerodynamic phenomena, like the development of complex shocks structures or complex viscous phenomena. As explained in the previous sections, thanks to the implementation of LTS, convergence acceleration techniques and DTS, converging to the next physical time step basically means performing a steady-state simulation with opportunely modified Euler/NS/(U)RANS equations. Thus, the same algorithm showed in 2.1.12 is employed, but it is repeated each physical time step until the user-prescribed final time. Usually $100 - 2000$ pseudo time steps are performed between two physical time steps depending on the test case under investigation. According to DTS techniques (2.1.11), the only other difference with respect to steady-state algorithm is given by the temporal derivative that appears in the equations as a source term and that must be taken into account when assembling cells residuals;.

2.2 Aeroelasticity

After the discussion of the purely aerodynamics formulations, it is time to introduce the schemes adopted to handle steady and unsteady cases that involve structure deformability. First of all the aeroelastic equations system is presented, The implemented strategies for mesh deformation and structural-aerodynamic grid interfacing are then introduced. Next, the transpiration boundary conditions are showed in order to represent a computationally potentially cheaper alternative to mesh deformation. Finally the previously introduced formulations are combined together and the strategies for trim analyses, forced oscillations analyses, free oscillations analyses and flutter analyses are presented.

2.2.1 Aeroelastic system

Since this work covers both numerical and computational aspects, in order to provide the reader a balanced view, the mathematical and numerical formulations behind the aeroelastic system are introduced. These formulations are based on the works [127,136], thus the reader is referred to these works for a more detailed view, especially for what concerns the structural subsystem. Furthormore, when considering turbomachinery and open rotors problems, details regarding the analytical and numerical point of view can be found in [122], especially for what concerns structural modelization.

Euler/NS/(U)RANS equations are employed to study the aerodynamic point of view of the problem, since they are used to investigate what happens inside the fluid domain. However, when considering the coupling between the fluid and the structure a new set of equations is required in order to represent the solid domain point of view. The

structural domain is usually discretized using a Finite Element Method (FEM) and by running software that allow to perform Finite Element Analyses (FEA). In aeronautical field usually different type of FEM elements can be used, like beams, plates, shells, and solid elements, considering a proper trade-off between results accuracy and computational costs. Wings, thanks to their high aspect ratio and internal structure can be usually discretized with beams characterized with particular sectional properties. This is usually not true for turbomachinery and open rotors blades [122] where shell or solid elements are usually preferred. For example, a beam model is adopted to discretize the structural properties of the HiReNASD wing (8.1, [127]), providing good results for trim analyses, i.e. static aeroelastic analyses. At the same time solid elements are used to discretize the Rotor 67 blade (10.3) while shell elements are adopted for the structural model of the SR-5 propfan (10.4, [122] and the AGARD 445.6 wing (8.2, [127]). In any case, when discretizing the solid continuum within a FEM framework, the system 2.29 is obtained:

$$\left[\bar{M}\right]\{\ddot{u}_s(t)\} + \left[\bar{C}\right]\{\dot{u}_s(t)\} + \left[\bar{K}\right]\{u_s(t)\} = \{f_s(t)\} \tag{2.29}$$

where $\left[\bar{M}\right]$ is the mass matrix, $\left[\bar{C}\right]$ is the damping matrix, $\left[\bar{K}\right]$ is the stiffness matrix and $\{u_s(t)\}$ is the vector of structural degrees of freedom. Mass and stiffness matrices can be directly assembled from the FEM model, eventually with some tuning like mass lumping. For what concerns the damping matrix, the situation is more complicated, as usually it is not straightforward its computation. Different strategies can be used to obtain the damping matrix (e.g. from experimental data or by opportunely combining mass and stiffness matrices), however this is beyond the scope of this work. Within a FEM framework, usually $\{u_s(t)\}$ array is represented by structural nodal degrees of freedom and they represent localized displacements of the deformable solid. $\{f_s(t)\}$ is the vector of external loads, aerodynamic loads in this particular work, and again they are defined on structural nodes. Thus, when performing aeroelastic simulations, from the aerodynamic point of view we have the Cell-Centered Finite-Volume discretized Euler/NS/(U)RANS system opportunely coupled with boundary conditions that take into account wall displacements and velocities due to mesh deformation (due to structural properties). Instead, from a structural point of view, we have system 2.29 that responds under the external aerodynamic loads $\{f_s(t)\}$. For the sake of trim and flutter analyses the aerodynamic and structural subsystems coupling is investigated from the structural point of view, i.e. considering system 2.29. This means that from the point of view of the aeroelastic analyses, the focus is on the structural system and aerodynamics is considered the external loads. When performing unsteady aeroelastic analyses, the unsteady variation of the aerodynamic loads due to structural motion is computed. This means that aerodynamic loads become function of the structural motion itself, thus function of $u_s(t)$. At this point it could be also possible to translate aerodynamic forces into mass, damping and stiffness contributions to assemble the final aeroelastic system, expressing the aeroelastic mass, damping and stiffness matrices.

Using the structural nodal degrees of freedom, directly obtained from the FEM model is not the only way to describe the behavior of the elastic solid domain. In fact from these d.o.f, it is possible to obtain other kind of representations thanks to opportunely defined transformations. For the sake of flutter and trim analysis covered in this work, the modal representation of the structural behavior is here presented alongside with the motivations behind this choice. The structural behavior of the elastic solid

is represented using modal degrees of freedom as follows:

$$[M]\{\ddot{q}(t)\} + [C]\{\dot{q}(t)\} + [K]\{q(t)\} = \{Q(t)\} \tag{2.30}$$

where $[M]$ is the modal mass matrix, $[C]$ is the modal damping matrix, $[K]$ is the stiffness matrix, $\{q(t)\}$ is the array of displacements in modal coordinates, also called generalized displacements. Finally, $\{Q(t)\}$ is the array of external loads in modal coordinates, also called array of generalized forces. It must be underlined that system 2.29 and system 2.30 have the same information content. Modal displacements can be obtained from nodal displacements using the following transformation:

$$\{q(t)\} = [U]^T \{u_s(t)\} \tag{2.31}$$

where $[U]$ is the modal shape matrix. At the same time, structural nodal coordinates can be obtained from modal coordinates using the following expression:

$$\{u_s(t)\} = [U]\{q(t)\} \tag{2.32}$$

Besides displacements transformation, the relation applies also to external loads. Thus it is possible to compute generalized forces from nodal forces as follows:

$$\{Q(t)\} = [U]^T \{f_s(t)\} \tag{2.33}$$

and similarly to 2.32 the following relation also holds:

$$\{f_s(t)\} = [U]\{Q(t)\} \tag{2.34}$$

The previous transformations are important when performing steady and unsteady aeroelastic analyses since structural behavior is considered in the solver through modal representation. In order to obtain modal properties from the initial FEM model an eigenanalysis has to be performed on the free system version of 2.29:

$$\left[\bar{M}\right]\{\ddot{u}_s(t)\} + \left[\bar{K}\right]\{u_s(t)\} = \{0\} \tag{2.35}$$

where for simplicity the damping matrix is not showed. This is often the case since damping matrix is both difficult to assembly and usually provides negligible effects on final results. Anyway, the modal analysis can be performed using different FEA solvers like NASTRAN, Code_Aster, Abaqus. Modal analyses provide modal shape matrix (eigenvectors) and modal frequencies (eigenvalues, eventually complex). By applying the transformation 2.32 and by projecting into modal space (pre-multiplying by $[U]^T$) it is possible to rewrite system 2.35 as:

$$[M]\{\ddot{q}(t)\} + [K]\{q(t)\} = \{0\} \tag{2.36}$$

An important point when performing modal analyses with rotating components is that the rotation effects have to be taken into account. This is explained in details in [122]. In any case, modern FEA software like the ones previously mentioned provide this capability. It is highlighted that the modal representation here presented is obtained from the structural system alone, without external loads. Thus modal shapes and frequencies represent just structural properties. In order to obtain aeroelastic modal properties, different techniques can be adopted. As mentioned, to obtain the aeroelastic system,

a way to compute aerodynamic loads as function of the displacements is required in order to close the aerodynamics-structure loop. This is discussed in 2.2.2.

The modal representation of the structural behavior is here showed and implemented in AeroX for different reasons, mainly in search of a trade-of between accuracy and computational efficiency, as explained also in 2.2.8. As previously mentioned, the aeroelastic system represented using FEM nodal degrees of freedom is completely equivalent to the same system expressed in modal coordinates. However, the main advantage of using the modal representation is that it can be used to perform a model reduction. This means that it is possible to accurately represent structural properties with a reduced number on modal degrees of freedom. This is immediately translated in computational efficiency advantages since the number of d.o.f. required for the solution of the forced system 2.30 becomes orders of magnitude smaller than system 2.29 if the model reduction is employed. In particular, as explained in [127] and later in this chapter in 2.2.8 few modal shapes and frequencies are usually sufficient to accurately perform both static and dynamic aeroelastic analyses. Since modal analyses algorithms (using FEA solvers) are able to extract a user defined number of low-frequency modes, modal analysis is usually efficient. Then, using the aeroelastic solver developed in this work, the solution of the structural system is performed on CPU, both when a CPU or a GPU is used to perform aerodynamic computations. Since this requires a CPU-GPU data transfer through PCI-Express bus (as explained in chapter 4) it is important to keep the GPU fed with available work, reducing the time spent on the CPU. Thus it is preferable an efficient but nonetheless accurate solution of the structural system. A reduced model based on the modal representation is the perfect choice for this purpose. Furthermore, the modal analysis is performed only once before any aeroelastic simulations, i.e. trim and flutter analyses, thus the computational effort is basically negligible with respect to the time spent for the next aeroelastic solver runs.

It must be noted that when performing steady and unsteady analyses the small perturbations hypothesis holds, thus it is possible to consider the generalized matrices constant. This is also true when considering rotating cases like in turbomachinery and open rotors [122].

2.2.2 Aerodynamic transfer function matrix

Flutter analyses can be performed by just checking the response of the system after the introduction of a small perturbation, checking the evolution of the interaction between the fluid and the structure. This is correct, but can be computationally inefficient. In fact if the aim is to find the flutter dynamic pressure, multiple expensive unsteady Fluid-Structure Interaction simulations have to the performed in an iterative manner until the conditions of self-sustained oscillations are found. This is basically what is done for the BSCW wing of the AePW2 (see 8.3) where starting from a guess dynamic pressure, the damping related to the wing pitch degree of freedom is checked. Computational flutter conditions are found when the amplitude of the oscillations is constant in time, meaning that the damping g is null and thus oscillations are self-sustained.

Here a more efficient strategy is presented, following [115, 122, 127, 136]. The idea is to use CFD simulations as a sort of experimental procedure to identify a Reduced Order Model (ROM) of the aerodynamic loads. This is done by performing a series of unsteady simulations in which the unsteady variation of the aerodynamic loads, with

respect to a reference steady-state solution due to the structural motion, is computed. It is highlighted that the steady-state solution could feature highly non-linear phenomena, like shocks, separations and complex interactions between boundary layer and shocks. However, even if the steady-state solution is characterized by such high non-linear phenomena, this is not necessarily true for the unsteady variations of the aerodynamic loads due to the structural motion. Structural motion have to be both large enough to provide results above the numerical error threshold but small enough to guarantee that the small perturbations hypothesis is still satisfied [127].

Considering the frequency domain, the so called aerodynamic transfer function relates generalized displacements and generalized aerodynamic forces as follows:

$$\{Q(s)\} = [H_{am}(p, M_\infty, Re)] \{q(s)\} \tag{2.37}$$

where $s = \sigma + i\omega$ is the complex frequency (i.e. in Laplace domain), $p = sL_a/V_\infty$ is the complex reduced frequency, L_a is the aerodynamic reference length, V_∞ is the reference air speed. M_∞ is the Mach number, Re is the Reynolds number and finally $[H_{am}(p, M_\infty, Re)]$ is the aerodynamic transfer function matrix. It is underlined that H_{am} is defined in Laplace domain and not directly in time domain. H_{am} matrix is defined in Laplace domain, however it is usually numerically computed in Fourier domain (through FFT) as $H_{am}(k)$, where $k = i\omega$ is the reduced frequency, thus computed along the imaginary axis. Numerical methods can be employed to extrapolate H_{am} for non general values of s with non null real part σ. How the aeroelastic solver is used to compute input and output from which the aerodynamic transfer function is computed is showed in 2.2.9, but here it is briefly presented. The first step is the computation of the reference equilibrium steady solution. This means finding the so called trim solution, explained in detail in 2.2.8, which is basically the steady-state aeroelastic solution characterized by certain values of generalized displacements due to structural deformations under aerodynamic loads. The next step is the already mentioned use of CFD as a sort of experimental setup where the unsteady variations of generalized aerodynamic forces $\{Q(t)\}$ are computed due to opportunely prescribed excitation time history of generalized displacements $\{q(t)\}$. In particular one unsteady simulation is performed for each considered generalized displacements, i.e. for each opportunely normalized modal shape. After the j-th simulation, exciting the j-th generalized displacement, the j-th column of the aerodynamic transfer function matrix can be computed:

$$[H_{am}(k)|_j] = \frac{F(\{Q(t)\})}{F(q_j(t))} \tag{2.38}$$

Thanks to the Fast Fourier Transform (FFT) algorithm, performing the transformation of the numerator and denominator is very fast, in the order of seconds. Furthermore it is understood that the computation of H_{am} is basically a post-processing operation and its assembly can be performed in parallel by computing each column independently [127], by exciting the structure with different generalized displacements.

At this point, the problem is to find a suitable time law for the structural displacements $q_j(t)$ in order to properly obtain the column $\{Q(t)\}$ and perform the FFT. Different kind of time laws could be employed, each one has advantages and drawbacks. The harmonic input could be used. The problem is that this kind of input only excites one frequency, thus one entire unsteady simulation for each desired frequency has to

be performed. Another option is the impulsive input: theoretically it is able to excite all frequencies simultaneously, but obviously from a numerical point of view it is impossible to introduce an ideal impulse and, if using a so called real impulse, very small physical time steps are required, degrading computational efficiency. A step input could be employed, with the advantage of a more accurate identification of the static gain and low frequency range. However, the derivative of this input is again an impulse, and it is easy to understand that the kinematic contribution to boundary conditions given by wall is afflicted by numerical problems. Thus, the best choice is represented by a blended step that combines the advantages of the step input for the static gain and low frequencies but at the same time bypasses the numerical problem of discretizing an impulse for the kinematic contribution. The blended step input is modeled as follows:

$$q(\tau) = \begin{cases} \frac{A_q}{2}\left[1 - cos(k_q \tau)\right] & if\ \tau < \tau_q \\ A_q & if\ \tau \geq \tau_q \end{cases} \tag{2.39}$$

where $q(\tau)$ represents the generalized displacement amplitude as a function of the dimensionless time $\tau = tV_\infty/L_a$ (not to be confused with pseudo time) that basically represents the number of aerodynamic reference length L_a traveled at the asymptotic speed V_∞. The other parameters are described in a moment. From expression 2.39, the time derivative $q(\tau)$ of the generalized displacement amplitude can also be computed:

$$\dot{q}(\tau) = \begin{cases} \frac{A_q k_q}{2} sin(k_q \tau) & if\ \tau < \tau_q \\ 0 & if\ \tau \geq \tau_q \end{cases} \tag{2.40}$$

As mentioned, no particular numerical problems are faced during the discretization of the time laws 2.39 and 2.40. The problem is now the choice of the various parameters. $k_q = k_\infty/2$ and $\tau_q = 2\pi/k_\infty$ are opportunely chosen in order to excite the low frequencies of interest in the interval $k \in [0, k_\infty]$. For what concerns A_q, the final amplitude of the step, as previously mentioned, should be large enough to provide a solution over the numerical error threshold but small enough to guarantee the small perturbations hypothesis. The choice of A_q is not straightforward, however one strategy that can be adopted is using the value computed with the following expression:

$$A_q = \frac{4 L_a \epsilon}{max\left(\left[U|_j\right]\right) k_\infty} \tag{2.41}$$

where $[U|_j]$ is the j-th column of the modal shape matrix showed in 2.32, ε is defined as follows:

$$\varepsilon = \frac{max\left(\{\dot{u}_s\}\right)}{V_\infty} = tan(1^\circ) \tag{2.42}$$

in order to have an opportunely small maximum nodal velocity with respect to the flight velocity.

After the aeroelastic unsteady simulation, a post-processing operation has to be performed in order to compute the j-th column of the aerodynamic transfer function matrix using 2.38. However, as explained in [127] it is better to modify the numerator and denominator of the expression and to consider the so-called "deficiency" of the

generalized aerodynamic forces and the "deficiency" of the generalized displacements:

$$[H_{am}(k)|_j] = \frac{\{Q_\infty\} + ikF(\{Q(t)\} - \{Q_\infty\})}{F(q_j(t) - q_{j,\infty})} \tag{2.43}$$

The previous expressions are related to the Fourier frequency k domain, strictly related to the FFT algorithm applied to the solver results. Assuming that the aerodynamic subsystem is asymptotically stable (this is experimentally verified for the aerodynamic subsystem alone, but the situation may change when coupled with the structural subsystem to form the aeroelastic system), and is excited with casual input signals, it is possible to obtain the generalized forces in time domain:

$$\{Q(t)\} = \int_0^\infty [h_{am}(t-\tau, M_\infty, Re)]\,\{q(\tau)\}\,d\tau \tag{2.44}$$

where $[h_{am}(t-\tau, M_\infty, Re)]$ is the so called aerodynamic impulsive response matrix. Again, the dependency from the Mach number and the Reynolds number is highlighted. System 2.30 can be rewritten considering that external aerodynamic forces are now expressed as function of the structural displacements themselves, leading to a free aeroelastic system:

$$[M]\,\{\ddot{q}(t)\} + [C]\,\{\dot{q}(t)\} + [K]\,\{q(t)\} - \int_0^\infty [h_{am}(t-\tau)]\,\{q(\tau)\}\,d\tau = \{0\} \tag{2.45}$$

The aeroelastic system 2.45 in Laplace domain assumes the following expression:

$$\left(s^2\,[M] + s\,[C] + [K] - [H_{am}(p)]\right)\{q\} = \{0\} \tag{2.46}$$

using definition 2.37. If the characteristic time $T_a = L_a/V_\infty$ that characterize the aerodynamic side of the problem is very small with respect to the structural characteristic time $T_s = 2\pi/\omega$, the so-called quasi-steady approach can be employed, significantly reducing the mathematical complexity of the problem. This is due to the fact that when considering constant flight conditions and the fact that the aerodynamic transfer function matrix is constant in $p = 0$, H_{am} can be expanded in Taylor series near $p = 0$ providing mass, damping and stiffness terms as follows:

$$\left(s^2\left([M] - \frac{L_a^2}{V_\infty^2}\,[M_a]\right) + s\left([C] - \frac{L_a}{V_\infty}\,[C_a]\right) + ([K] - [K_a])\right)\{q\} = \{0\} \tag{2.47}$$

that in time domain is translated in:

$$\left([M] - \frac{L_a^2}{V_\infty^2}\,[M_a]\right)\{\ddot{q}\} + \left([C] - \frac{L_a}{V_\infty}\,[C_a]\right)\{\dot{q}\} + ([K] - [K_a])\,\{q\} = \{0\} \tag{2.48}$$

with the understandable simplifications with respect to the original system 2.45. An important aspect is that the aerodynamic transfer function matrix is analytic, allowing to compute its derivatives in any direction.

As previously mentioned, flutter analyses can be performed by checking the aeroelastic responses of the system when a perturbation is introduced in the equilibrium configuration, as explained in 2.2.10. However, using all the presented aeroelastic mathematical framework it is possible to perform more computationally efficient flutter analysis simulations as explained in 2.2.9 and 2.2.3.

2.2.3 Aeroelastic system stability and flutter

Flutter is the instability of the aeroelastic system 2.45 which represents the linearization of the aerodynamic and structural problem for small perturbations around a reference equilibrium condition. This is an important kind of numerical and experimental analysis since oscillations are self-sustained and could lead to fatigue life reduction of the flexible aerodynamic components with catastrophic consequences. The problem with flutter is that the aerodynamic subsystem and the structural subsystem, when considering non null structural damping, are both asymptotically stable. However, by closing the loop between the two subsystems, there is no guarantee that the final aeroelastic system is asymptotically stable for certain conditions. In fact, due to the fluid-structure interaction, by increasing flight speed from 0, the original structural eigenvalues of the structural subsystem 2.36 may be shifted up to unstable conditions, by passing through the imaginary axis at some conditions, flutter conditions. Obviously more than one aeroelastic eigenvalue may become unstable, however the investigation is focused on the minimum flight speed V_∞ for which the system becomes simply stable. This is often studied by considering the aeroelastic frequency ω and damping g, defined as:

$$g = 2\sigma/\sqrt{\omega^2 + \sigma^2} \tag{2.49}$$

where σ is the real part of the considered aeroelastic eigenvalue $s = \sigma + i\omega$, and ω is its imaginary part. At flutter conditions, since $\sigma = 0$ also $g = 0$ holds. The aeroelastic system 2.45 can be represented in a more convenient way as follows:

$$[A(s, V_\infty)]\{q\} = \{0\} \tag{2.50}$$

remembering the dependency of H_{am} from flight conditions. In order to find flutter conditions it is possible to enforce that the real and imaginary parts of the determinant of matrix $[A(s, V_\infty)]$ are null. This way, two scalar equations are obtained and the solution provides flutter speed V_F and frequency ω_F. Then, these parameters can be used to extract the flutter eigenvector $\{q(\omega_F, V_F)\}$ in order to see its particular shape. However, usually it is useful to study the behavior of the aeroelastic system eigenvalues with respect to the flight speed V_∞, using a root-tracking procedure, starting from $V_\infty = 0$ when the aeroelastic system eigenvalues are identical to the purely structural eigenvalues.

Now, the numerical strategies adopted for flutter analyses will be briefly discussed. As previously mentioned, there is a problem in computing the aerodynamic transfer function matrix in Laplace space. This is due to the fact that the numerical procedure allows to evaluate the aerodynamic transfer function matrix over just the imaginary axis, providing $H_{am}(k)$, and not $H_{am}(p)$ over the whole complex domain. Thus, a strategy to extrapolate the values of H_{am} for generic values of p is required. Two strategies are usually adopted to handle the problem: a non-linear method and the so called PK method. The idea behind the PK method is that by considering small damping values, thus considering $p = \varepsilon + ik$ with $\varepsilon << 1$, and considering that H_{am} is analytical in $p = 0$, it is possible to expand $H_{am}(p)$ using a Taylor series around $p_0 = ik_0$ as follows:

$$[H_{am}(p)] \simeq [H_{am}(k)|_{k=k_0}] + [H'_{am}(k)|_{k=k_0}](p - ik_0) + \frac{1}{2}[H''_{am}(k)|_{k=k_0}](p - ik_0)^2 \tag{2.51}$$

this way it is easily possible to directly perform the eigenanalysis of the aeroelastic system 2.50 near-by the flutter velocity V_F.

Another strategy is represented by a non-linear method. The idea is to solve the free aeroelastic system of non-linear homogeneous equations 2.50 for the eigenvector $\{q\}$ and for the eigenvalue s, alongside with an eigenvectors normalization equation used to close the problem. This is done as follows:

$$\begin{cases} [A(s, V_\infty)] \{q\} & = 0 \\ \frac{1}{2} \{q\}^T [W] \{q\} & = 1 \end{cases} \tag{2.52}$$

where $[W]$ is a weighting matrix and it's highlighted the dependency of the system from the flight speed V_∞. The solution procedure of the system 2.52 is based on the well known Newton–Raphson method that requires the linearized form of the previous system (see [127]). With the non-linear strategy it is possible to obtain eigenvectors $\{q\}$ and eigenvalues s for any input velocity V_∞. The usual strategy is to start from $V_\infty = 0$, thus with the eigensolution provided by the purely structural subsystem 2.36 and then start the root-tracking procedure in order to draw the $V_\infty - \omega$ and $V_\infty - g$ diagrams. This last strategy is adopted for the flutter investigation of the AGARD 445.6 wing, see 8.2.

2.2.4 Aeroelastic interface

Usually, in aeroelastic simulations, the discretization of the solid and the fluid domains is performed independently. This is due to the fact that CFD and FEA formulations have different continuum discretization requirements that depend on the particular phenomena happening in the fluid and solid domain. Thus the wall boundaries in common between the aerodynamic and structural meshes usually feature different discretizations, meaning that the set of aerodynamic mesh wall nodes $\{x_a\}$ differs from the set of structural mesh wall nodes $\{x_s\}$. As an example, when performing (U)RANS simulations over a wing, the near-wall domain is opportunely discretized in order to reconstruct boundary layer effects and/or accurately predict shocks location. At the same time, the wing could be structurally modeled with beam elements. It is evident that this way an interface between structural mesh wall points and aerodynamic mesh wall points is mandatory. In this section the strategy adopted in the solver to compute the interface between the aerodynamic and structural meshes is briefly described, based on [127, 132].

When performing aeroelastic simulations it is required a way to project aerodynamic loads from aerodynamic (mesh) wall nodes $\{f_a(t)\}$ to structural (mesh) wall nodes $\{f_s(t)\}$ and a way to project structural wall nodes displacements $\{u_s(t)\}$ to aerodynamic wall nodes displacements $\{u_a(t)\}$. This is due to the fact that aerodynamic loads are computed by the CFD solver using the aerodynamic mesh and thus the aerodynamic mesh wall discretization, while the structural properties are computed from the FEM model that features its own wall discretization. In fact if the structural properties are represented with a full FEM model the structural response has to be computed by solving system 2.29 that requires aerodynamic loads in structural nodal coordinates $\{f_s(t)\}$ and gives solid point displacements $\{u_s(t)\}$ again in structural nodal coordinates. At

this point the projection of aerodynamic loads from aerodynamic wall nodes to structural wall nodes is required. Then, after the system solution, in order to compute the aerodynamic solution around the updated solid shape, the aerodynamic mesh has to be updated. To perform this update wall displacements defined on aerodynamic mesh wall nodes are required and have to be computed from structural wall nodes displacements obtained from the solution of the forced structural system. The same is also true when, like in this work, the structural behavior is represented by a modal reduction. In this case in fact system 2.30 has to be solved by providing generalized aerodynamic forces $\{Q(t)\}$ and obtaining then generalized displacements $\{q(t)\}$. However generalized forces and displacements are still related to structural nodal counterparts through relations 2.31 and 2.34. Thus, the aforementioned problem of projecting forces and displacements between aerodynamic and structural meshes arises also when employing modal reduction.

Considering the formulation explained in [132], the aeroelastic interface between aerodynamic and structural wall nodes displacements is represented by means of a linear operator $[I]$:

$$\{u_a(t)\} = [I]\,\{u_s(t)\} \tag{2.53}$$

thus the relation between the two sets of wall nodes displacements can be represented through a rectangular matrix. This relation is used after the structural system solution, when the just computed wall displacements have to be used to update the aerodynamic mesh. It must be noted that relation 2.53 binds only aerodynamic mesh wall nodes and structural mesh wall nodes, while the rest of internal nodes of the aerodynamic mesh have to be updated with a different strategy, as explained in 2.2.6.

As explained in [132], a fundamental requirement for the aeroelastic interface is the conservation of momentum and energy exchanged between the aerodynamic and the structural subsystems. This means that the virtual work made by the aerodynamic forces, $\{f_a(t)\}$, for the structural displacements interpolated on the aerodynamic nodes $\{u_a(t)\}$, must be equivalent to the virtual work made by the aerodynamic forces interpolated on the structural nodes, $\{f_s(t)\}$, for the structural displacements, $\{u_s(t)\}$. This has a very handful consequence since the transpose of the aeroelastic interface operator can be used to obtain the aerodynamic forces on structural nodes from aerodynamic forces on aerodynamic nodes:

$$\{f_s(t)\} = [I]^T\,\{f_a(t)\} \tag{2.54}$$

The interface matrix $[I]$ can be assembled using different strategies in order to weight the contribution of the set of aerodynamic wall nodes over each structural wall node. As an example, an Inverse Distance Weighting (IDW) approach can be used, eventually by neglecting the contribution of too distant nodes. In this work a different strategy is employed, however IDW is adopted for the aerodynamic mesh internal nodes position update described in 2.2.6. The approach here adopted is based on a Moving Least Squares (MLS) technique and the use of Radial Basis Functions. The detailed description of the numerical aspects is beyond the purpose of this work, for more details the reader is referred to [132]. When using RBF, the influence between the structural and aerodynamic nodes is weighted using functions that depend only by the relative distance between the two type of nodes:

$$\Phi(\mathbf{x_s}, \mathbf{x_a}) = \Phi\left(||\mathbf{x_s} - \mathbf{x_a}||\right) \tag{2.55}$$

where $\mathbf{x_a}$ is the aerodynamic mesh wall node position and $\mathbf{x_s}$ is the structural mesh wall node position. Different kinds of function can be used to compute each element of the interface matrix, but they all satisfy the form of 2.55.

It is again highlighted that the interface matrix only binds the nodes from the aerodynamic mesh and the structural mesh that reside on the surface of the deformable geometry. This means that the interface matrix is only useful to update the aerodynamic mesh wall shape, but another kind of strategy is required to fully update aerodynamic mesh internal nodes, as will be explained in 2.2.6.

There are some important computational aspects that must be discussed, in particular since the aeroelastic solver is GPU-accelerated. The computation of the interface matrix can be quite costly, however it is computed only once before any steady or unsteady simulation. In particular the interface matrix is computed from aerodynamic wall nodes and structural wall nodes locations of the original undeformed geometry. The interface matrix is stored in system memory and read every time the shape of the object under investigation has to be updated due to aerodynamic loads or to enforce oscillations. This is performed at each physical time step in unsteady simulations, as described in 2.2.9 and 2.2.10 for forced and free oscillations analyses respectively. The interface matrix is also required at each structural iteration when performing static aeroelastic analyses, i.e. trim analyses, as explained in 2.2.8. In any case, the aeroelastic interface is usually represented by a full matrix, thus its storage could be quite costly when handling thousands of wall nodes. Interface matrix is stored on system RAM and the computations related to expression 2.53 and 2.54 are performed on CPU. This reduces GPU memory consumption allowing to perform aeroelastic simulations on bigger cases on gaming GPUs that feature few GB of memory. However, this choice does not represent a particular bottleneck for GPU executions since, as mentioned, between two shape updates hundreds or thousands of purely aerodynamic explicit iterations are required. Furthermore, the CPU-GPU data exchange through PCI-Express bus is limited to just the wall nodes data, since, as will be showed in 2.2.6, aerodynamic mesh internal nodes update and mesh metrics update are performed instead on GPU. Considering all of these aspects, the time spent to perform mesh interface operations on CPU and data transfer between the CPU and the GPU is negligible with respect to purely aerodynamic convergence iterations performed on GPU.

2.2.5 Moving Boundaries

Excluding temporarily cases with rotating components, a key difference between steady-state analyses with mesh deformation and unsteady analyses with mesh deformation is represented by the fact that in the latter case not only ALE formulation is used when computing convective fluxes on internal and boundary faces, but also by the fact that wall boundary conditions must be opportunely modified in order to take into account the wall velocity. Furthermore, when considering also MRF formulation (see 3.6) the contributions of the rotation and the deformation have to be added up. As said the approach adopted in this work to enforce boundary conditions is based on the ghost cell concept. Thus, enforcing a non-null boundary velocity basically means enforcing particular values of ghost cell conservative form variables. It must be noted that when performing inviscid simulations the only important component is the velocity normal to the boundary face as it is the only information required to enforce slip boundary

conditions. Boundary faces velocities are computed alongside displacements on CPU and then transferred to the GPU. It is reminded that this does not particularly reduce computational efficiency since boundary faces velocities and displacements are computed once every physical time step. Thus all the pseudo time iterations between two wall faces boundary conditions updates can be performed without data-transfers, leaving the GPU continuously loaded. Boundary wall faces velocities are computed using relations 2.53 and 2.32 but using $\{\dot{q}(t)\}$ instead of $\{q(t)\}$, $\{\dot{u}_s(t)\}$ instead of $\{u_s(t)\}$ and $\{\dot{u}_a(t)\}$ instead of $\{u_a(t)\}$. In this work aerodynamic mesh wall nodes displacements are always referred to the undeformed configuration rather than being defined as incremental values between two different wall displacements.

2.2.6 Aerodynamic mesh internal nodes update

As said in 2.2.4 the interface matrix only binds aerodynamic mesh wall nodes and structural mesh wall nodes. Thus a way to update the location of the rest of the aerodynamic mesh internal nodes is required. This can be performed through different formulations like Laplacian Smoothing [87]. Other strategies are based on the analogy between the continuum and an elastic solid [52]. Mesh deformation is particularly important also for what concerns shape optimization problems [92]. Here, a strategy based on the Inverse Distance Weighting (IDW) [153] formulation is instead adopted, thanks to its combination of results quality and high computational efficiency when exploiting GPUs. The implemented strategy is based on the formulation presented with more details in [132]. Here the strategy is briefly presented, with the focus on computational aspects.

The idea behind IDW is to compute the displacements of each aerodynamic mesh internal node $\left\{u_a^k\right\}$ by opportunely weighting the displacements contributions of the moving wall nodes $\{u_a^i\}$ as follows:

$$\left\{u_a^k\right\} = \frac{\sum_{i=0}^{Nwall} W_{k-i} \left\{u_a^i\right\}}{\sum_{i=0}^{Nwall} W_{k-i}} \tag{2.56}$$

Following the definition of "IDW", the weights W_{k-i} are computed from a power of the inverse of the distance between the internal node k and the wall node i:

$$W_{k-i} = \frac{1}{|\mathbf{x}_k - \mathbf{x}_i|^p} \tag{2.57}$$

where the exponent p can be used to adjust the smoothness of the results. Usually $p = 2$ or $p = 3$ provides good results. Expression 2.56 has to be evaluated for each internal aerodynamic mesh point. For each aerodynamic mesh internal point a loop over the aerodynamic mesh wall nodes is required. Here we have two different operations to be performed: weights computation and displacements computations. Considering that thousands of wall nodes are usually used to discretize the shape of a deformable aerodynamic object and millions of nodes are instead used for the internal mesh, it is understood that both represent heavy computational operations. However this is basically a data-parallel operation, perfectly suited for the GPU architecture. As explained in 2.2.5, wall nodes displacements are always computed as absolute displacements from original wall nodes positions. This is true also for internal nodes displacements. This way the weights that are required at each physical time step, in unsteady simulations, or

at each mesh update step in trim simulations could be computed just once before starting the aeroelastic simulations. Next, the weights could be stored and reused whenever is required to update the aerodynamic mesh. However this is not how the strategy of updating internal mesh node positions is implemented. In fact, what is done is that each time the mesh requires an update, weights are all recomputed and displacements are then computed accordingly, even if the weights are kept constant through the entire simulation. Furthermore, this is all performed on the GPU. The explanation of this choice is based on the advantages and drawbacks related to CPUs and GPUs architectures. As said, the whole operation of computing weights can be performed in a data-parallel fashion since all the internal nodes have to loop an all the wall nodes to compute the weights. This operation can be efficiently performed by the GPU, exploiting the sequential memory access to load nodes locations, and the capabilities of its hundreds/thousands of cores to perform the floating point operations required by expression 2.57. Similarly, GPU architectures are also well suited to compute the displacements using expression 2.56 by reloading weights (that, as said, are kept constant during the simulation) from memory and computing internal nodes displacements by spreading the work on thousand of cores. The problem in this strategy is that with the aim of exploiting low-cost gaming GPUs that exhibit few GB of GPU memory, this in unfeasible due to the large amount of memory required by the storage of all weights. As an example, if an aerodynamic mesh features $10k$ wall nodes and $1M$ internal nodes, considering $4\,bytes$ to store each weight, a total amount of over $37\,GB$ would be required. At the same time this operation could be performed by the CPU, by computing the weights once before the aeroelastic simulation and recalling them when updating internal nodes position. This would theoretically solve the problem with memory consumption provided that enough RAM is installed in the system. However, this is again not feasible since each time the mesh is updated the CPU has to compute displacements using a nested loop formed by the internal nodes loop and wall nodes loop. Performing this operation on the CPU is orders of magnitude slower than performing it on the GPU. By using the CPU to update the aerodynamic mesh internal points positions the advantages provided by the GPU to speed-up the aerodynamics convergence would be avoided by the time spent waiting the CPU for the internal nodes update. Thus, the strategy that allows to both speed up IDW computations and at the same time allows low memory requirements is represented by performing on GPU, each time the mesh is deformed, the re-computation of the weights and, of course, the internal nodes displacements. Thanks to the efficient implementation of IDW and the fact that, as for the aeroelastic interface related computations (see 2.2.4), this operations is performed only once every few hundreds/thousand of purely aerodynamic iterations, the time spent by the GPU to perform IDW operations is negligible with respect to the time required for purely aerodynamics convergence. This is better shown in 6.2.2 when discussing about benchmark results and profiling the different kernels.

After the update of the aerodynamic mesh internal nodes, mesh metrics have to be recomputed, alongside, for unsteady simulations, the faces ALE velocities. It is understood that both operations can be efficiently parallelized on a typical GPU architecture since basically the same operations have to be performed over thousand/millions of elementary mesh objects like cells volumes, faces areas, face material velocities. It is highlighted that in this work mesh metrics update just means updating cells shapes

and cells/faces related quantities while mesh connectivity is preserved. Consequently after each mesh update all the addressing arrays are preserved. The discussion about how to compute such mesh metrics is beyond the scope of this work. The strategy here adopted to compute faces material velocities for the ALE framework when performing simulations with mesh deformation is described in details in [127]. Roughly speaking the ALE velocity at each physical time step of each face is computed as the ratio between the volume crossed by the face in the unit of time in the face normal direction, divided by the face area.

The implementation of the IDW algorithm is presented in 5.3.11.

2.2.7 Transpiration boundary conditions

Transpiration boundary conditions are implemented in the solver as a computationally cheaper alternative to the effective mesh deformation when simulating unsteady cases with wall movements. The main idea of transpiration boundary condition is to emulate the effects of wall displacements and velocities by employing an opportunely crafted non-penetration boundary condition, bypassing the need of true mesh deformation. The main advantage given by this strategy is the fact that the costs of updating mesh points positions and mesh metrics is avoided, thus simulations times are reduced. However, this comes at the cost of a limited range of applicability and a reduction of results accuracy. As showed by the results in [136], far small wall movements, transpiration boundary conditions can provide good results, comparable with the more accurate but expensive mesh deformation with ALE formulation. Anyway, if some kind of complex wall movements, particular flow-field phenomena, or just bigger displacements have to be taken into account, mesh deformation could be the only available choice to provide enough accurate results. A complete description of transpiration boundary conditions is available in [136]. Here just the final expression of the transpiration speed is showed, considering the non-linear finite-difference approach. The following expression shows the implemented transpiration speed:

$$V_n = -\mathbf{V}_\infty \cdot \boldsymbol{\Delta}\hat{\mathbf{n}} + \dot{\mathbf{s}} \cdot \hat{\mathbf{n}}_\mathbf{0} + \dot{\mathbf{s}} \cdot \boldsymbol{\Delta}\hat{\mathbf{n}} \tag{2.58}$$

where V_n is the wall face velocity normal component. This is used to alter the true slip boundary condition (for which $V_n = 0$) and emulate wall movements. $\mathbf{V}_\infty$ is the asymptotic speed, $\boldsymbol{\Delta}\hat{\mathbf{n}}$ is the face normal unitary vector rotation, $\dot{\mathbf{s}}$ is the wall face speed and $\hat{\mathbf{n}}_\mathbf{0}$ is the original face unitary normal vector.

It must be noted that in this work, thanks to the GPU acceleration of the mesh deformation algorithms and thanks to DTS, simulations performed using transpiration boundary conditions and true mesh deformation have about the same computational costs. This way in this work true mesh deformation is always preferred thanks to superior results accuracy.

The implementation of the transpiration boundary conditions is discussed in 5.3.11.

2.2.8 Trim analyses

Trim analyses are aimed to provide a static aeroelastic solution. The idea is to investigate the effects of the static structural behavior of the aerodynamic component under aerodynamic loads. Trim solution are useful for two different purposes.

First of all they provide more accurate steady solutions with respect to more classical steady analyses with fixed geometry. By allowing the wall boundaries to be freely deformed under the aerodynamic loads, the purely aerodynamic behavior of the component is coupled with its structural properties. As will be discussed in the results sections of this work, this is particularly important for some kinds of aeronautical components like airplanes and wings (e.g the trim of the HiReNASD wing described in 8.1) where the differences between the purely aerodynamic steady solution and the static aeroelastic solution are easily appreciable. However, there are also cases in which trim solutions provide only negligible accuracy improvements on the final solution. This is the case with the trim simulation of the NASA's Rotor 67 investigated in 10.3 where the characteristic curves given by the static aeroelastic simulations are basically overlapped to the ones provided by the purely aerodynamic simulations. In fact, given the shape and the properties of the adopted material, the blade of the Rotor 67 is relatively stiffer with respect to the HiReNASD wing. Thus in the entirely operating regime of the rotor, aerodynamic loads are unable to deflect the blade more than a fraction of the blade thickness. In a typical plane wing instead (e.g. the HiReNASD case), wing tip displacements are usually multiple of the airfoil thickness. This means that in the case of the Rotor 67, given the small blade displacements, negligible differences are obtained locally in the flow field. Consequently this is also translated in negligible differences for what concerns the integral parameters such as the characteristic points values.

The second purpose of a static aeroelastic simulation is to provide the initial conditions for a subsequent dynamic aeroelastic simulation. In fact, unsteady aeroelastic simulations are usually performed by studying vibrations around equilibrium conditions. Since the unsteady aeroelastic simulation involves the mechanical properties of the investigated aerodynamic component, the equilibrium conditions themselves must consider also the structural behavior of the aerodynamic component. As an example, in the flutter analysis of the BSCW blade of the AePW2 (see 8.3), before starting the unsteady analysis, a trim simulation is performed in order to provide the equilibrium initial conditions. These conditions are characterized by a particular wing vertical displacement, wing rotation and the related aerodynamic fields. This way, the expensive unsteady simulation that would be required to go from initial guess conditions up to the steady aeroelastic equilibrium solution is avoided. The BSCW blade of the AePW2 presented in this work is characterized by two rigid degrees of freedom, pitch and plunge. However, trim simulations can obviously be performed also with a modal representation of the structural behavior, provided that modal shapes and stiffness are computed with a FEM solver.

From a mathematical and numerical point of view, trim analyses are basically steady analyses in which the static structural behavior is taken into account (only stiffness without inertial and damping properties). Thus, all the convergence acceleration techniques previously described in 2.1.10 can be adopted to accelerate convergence. DTS, described in 2.1.11 is not employed. Thus, given a certain mesh size, the costs of a trim analysis are in the order of a steady analysis, lower then a fully unsteady analysis. The costs of a trim analysis with respect to a steady aerodynamic analysis are bigger since mesh deformation is required and convergence is checked not only on aerodynamic residuals but also on structural residuals. Anyway, it is expected that if the converged trim simulation is characterized by small displacements with respect to

the original configuration, with the implemented strategies the computational time required by a trim simulation is of the same order of magnitude of the one required for a purely aerodynamic steady simulation.

When performing trim analyses, system 2.30 is reduced to its steady form, i.e.:

$$[K]\{q\} = \{Q\} \tag{2.59}$$

so the purpose of the simulation is to find the generalized displacements (thus the object shape) under aerodynamic loads. Obviously aerodynamics depends from the object shape itself, so an iterative algorithm must be employed. It must be noted that when performing trim simulations of rotating components like turbomachinery and open rotors blades, the effects of the rotating solid domain must be taken into account since they change the structural properties with respect to the non-rotating case [122].

Here, the strategy adopted in the solver to provide the trim solutions is presented and discussed:

1. The first step is the computation of the possible deformations of the aerodynamic component. This means rigid degrees of freedom or modal shapes degrees of freedom. As an example, for the BSCW wing of the AePW2, see 8.3, the wing vertical displacement and wing rotation are selected as the only possible degrees of freedom. In the case of the trim of the HiReNASD wing 8.1 or the Rotor 67 10.3 blade, instead, a modal reduction is performed before the trim simulation, using a FEM solver (like NASTRAN or Code_Aster). The FEM analysis is performed considering an unloaded model of the aerodynamic object. Anyway, for this step the requirements are: a set of possible object deformations (modal shapes or rigid d.o.f) and the correspondent stiffness values. As explained in 2.2.4, RBF are used to assembly the interface matrix between aerodynamic mesh wall nodes and structural mesh wall nodes. This operation is performed only once and the same interface matrix can be used also for subsequent dynamic aeroelastic analyses;

2. The aerodynamic solver is started using the undeformed mesh. The purely aerodynamic solution is obtained with the fixed geometry (see 2.1.12). This will provide the initial guess for the true aeroelastic simulation;

3. Once the purely aerodynamic solution is obtained, the solver is restarted in trim mode. The original mesh is read alongside the steady-state solution fields to be used as initial guess;

4. Now aerodynamic loads have to be computed. If the free degrees of freedom are rigid displacements, pressure and viscous stresses are integrated over the object walls and projected in the 6 directions (3 forces and 3 moments). If a modal representation of the structural behavior is employed, the situation is more complicated. In fact, this means that alongside the aerodynamic mesh, also the structural mesh is provided. Thus, recalling 2.2.4 and in particular equation 2.54, loads are projected from the aerodynamic wall nodes to the structural wall nodes. Now using equation 2.33 loads on structural wall nodes are translated to generalized forces thanks to the modal shapes matrix. This way, after this step of the trim algorithm, forces related to the available degrees of freedom are now available;

5. The next step is the solution of the system 2.59 for the generalized displacements, knowing the degrees of freedom stiffness and and the related forces. Thanks to the modal representation of the structural behavior the total number of free degrees of freedom is limited (usually in the order of 10). Thus the solution of the system requires an amount of computational time that is basically negligible with respects to other algorithms inside the trim analysis, especially with respect to the GPU-accelerated aerodynamic convergence and mesh deformation. At the end of this step generalized displacements $\{q\}$ related to free degrees of freedom are known and the aerodynamic mesh has to be updated accordingly. Relative residuals related to degrees of freedom are computed employing a normalization of the difference from the previous computed values. These residuals are used to check convergence from the structural point of view;

6. Next, it is necessary to update the shape or the position and rotation of the object accordingly to the static structural response. Thus, first of all wall displacements are computed. With rigid degrees of freedom this is straightforward. The situation is more complicated with the modal representation. In fact, from the previous step the generalized displacements are known but they need to be mapped to the structural nodes. Thus, expression 2.32 must be employed to translate generalized displacements into structural nodes wall displacements. Then, by exploiting the interface matrix 2.53 it is possible to compute aerodynamic mesh wall nodes displacements from structural mesh wall nodes displacements;

7. Now, from aerodynamic mesh wall nodes displacements it is possible to compute the aerodynamic mesh internal nodes displacements. This can be done in exactly the same way described in 2.2.6;

8. Knowing the new positions of internal nodes it is possible to update the mesh metrics: this involves cell volumes, cell centers, face centers, face normals, face areas;

9. The entire algorithm is repeated until both the aerodynamic residuals and structural displacements residuals are converged or the maximum number of iterations is reached;

Few aspects of the strategy require attention. The trim solver could be directly started from a user-prescribed guess solution, without firstly performing a steady-state simulation with the completely rigid structure. However, starting from an already converged steady-state solution with rigid structure is better, because, if small displacements are expected, the trim solution should converge relatively fast, with a computational cost in the order of a purely aerodynamic steady simulation. Otherwise, if the trim solver is started from e.g. a constant pressure, temperature and velocity fields, this could slow down the convergence, since not only aerodynamics is involved but also structural displacements. Another important aspect, especially related to GPU executions, is that the purely aerodynamic computations are repeated for a certain number of iterations between two consecutive mesh updates. This is done in order to allow a better aerodynamic convergence before the next mesh update. In fact, since an explicit solver is implemented in this work, it is unlikely that a single aerodynamic (pseudo time) iteration is sufficient to reach convergence over the new wall shape. Furthermore,

from a computational point o view, since the mesh update requires CPU-GPU data transfer, it is better to reduce the total number of times that data is exchanged, without undermining convergence properties. As will be described in 10.3 and 8.1 for a turbomachinery blade and wing cases respectively, good trade-off values are represented by $500 - 2000$ purely aerodynamic iterations between two consecutive mesh updates and about a total of $20 - 50$ mesh updates. Obviously these numbers are purely indicative and they depend from the particular case under investigation, especially from the magnitude of the expected deformations. Usually 50000 explicit aerodynamic iterations are sufficient for trim convergence by restarting the solver from a steady-state solution with fixed geometry. Considering that a steady solution with fixed geometry requires about $20000 - 50000$ iterations (depending if Euler or RANS equations are employed) and that mesh deformation algorithms are GPU-optimized, the computational time required for a trim solution is basically in the same order of magnitude of what required to obtain a steady solution with fixed geometry.

Other important aspects are related to the idea of performing the structural model reduction with the modal representation of the structural behavior. As explained, the modal reduction of the structural behavior provide both an accurate and efficient representation of the elastic solid. When the free response on the aerodynamic component have to be computed, both in a steady and unsteady simulation, if the starting model is a FEM model, theoretically all the FEM degrees of freedom have to be used to compute the structural response. This computation is done on CPU while the GPU is waiting for the displacements update. Since the focus in this work is on the reduction of the wasted GPU time, a modal reduction is preferred over a full FEM representation. This way the system that needs to be solved is smaller with the modal approach. As previously explained, this is true when the solid behavior is described with a number of modal shapes, stiffness and masses that is smaller than the one provided by the original FEM model. The choice of the total number of adopted low stiffness modes is not straightforward. However, with the aim of a trim simulation, different checks can be performed in order to asses the correct reconstruction of the elastic behavior with the modal representation. First of all modal convergence can be checked by computing trim solution with N modes and then compute again the trim solution with $N + 1$ modes. After a certain small number of adopted low frequency modes, the contributes of higher modes become negligible for what concerns the final shape of the aerodynamic object. Another useful check for modal convergence is related to the elastic energy. When the blade or wing structure is deformed under aerodynamic loads, modal degrees of freedom assume certain values that allow the structure to be in equilibrium with external loads. Thus, considering each mode stiffness and generalized displacement, it is possible to compute the elastic energy related to each mode. If an unitary mass normalization is employed, it is possible to directly compare the elastic energy of each mode. The idea is that modes with higher stiffness can be neglected if their contribution to the total elastic energy of the equilibrium configuration is negligible. All of these aspects will be used with the NASA's Rotor 67 trim test case in 10.3, where it is possible to see that the first two modes contributes for over the 95% of the total elastic energy and using 3 or 4 modes to describe the elastic behavior provides only negligible differences.

It must be noted that trim analyses, as much as steady analyses in general, could be potentially performed also through an unsteady simulation. In this case it could be

possible to perform an unsteady simulation starting from an initial guess (e.g. constant pressure, temperature and velocity fields and non-deformed geometry) or re-starting from a steady solution with non-deformed geometry. Then, the unsteady analysis is started with the introduction of the structural behavior and mesh deformation and by leaving the aerodynamic component to be free to be deformed (this procedure is described in details for true unsteady analyses in 2.2.10. Obviously this second strategy that employs unsteady simulations should provide the same results of the true steady-state aeroelastic strategy described in this section (provided that a steady-state solution in the investigated conditions effectively exists). It is easy to understand, however, that a steady trim analysis is order of magnitude computationally cheaper than an unsteady aeroelastic analysis converged to a steady-state solution.

2.2.9 Forced oscillations analyses

Here the strategy adopted to perform forced oscillations analyses is described. These are unsteady analyses where the deformation of the aerodynamic component is enforced and not computed as an aeroelastic response due to aerodynamic loads. Forced oscillation analyses can be used for different purposes. They can be used when the case effectively involves the movement of the aerodynamic component described through a time law. Forced oscillations simulations can be employed for stability analyses such as flutter analyses or aerodynamic damping analyses (for turbomachinery). Flutter analyses, in particular, can be performed using two different strategies. In fact it is possible to perform a free response analysis due to an initial perturbation, as better described in 2.2.10 and by checking if oscillations are sustained. The second strategy takes advantage of what described in 2.2.3, enforcing modal shapes deformations through an opportunely specified time law. For what concerns aerodynamic damping analyses, the concept is similar to the latter idea: a particular deformation of the blade is enforced using an oscillatory time law and then at post-processing it is possible to compute the aerodynamic damping value using energetic concepts. This is described in details later in 3.4.

Here, the algorithm adopted for forced oscillations analyses is presented:

1. The first step is the computation of the possible degrees of freedom of the aerodynamic component. This means rigid displacements or modal shapes. As an example, for the aerodynamic damping computation of the SC10 2D and 3D cases (see 10.1 and 10.2) pitch and plunge rigid displacements are considered. Instead, for what concerns the flutter analysis of the AGARD 445.6 wing (see 8.2) a modal analysis is performed before the unsteady aerolastic analysis, providing the modal frequencies and shapes that will be enforced using an opportunely specified time law. The modal analysis is performed once using a FEM solver, considering the unloaded aerodynamic component and discretizing it with an opportunely choice of FEM elements (e.g. beams for wings and shells for turbomachinery blades). With rotating cases, during modal analysis centrifugal effects must be taken into account. If modal shapes are adopted as degrees of freedom, the interface matrix has to be computed knowing aerodynamic wall nodes positions and structural wall nodes positions. This is performed only once before the aeroelastic simulations, as explained in 2.2.4;

2. The purely aerodynamic steady-state solver 2.1.12 or trim solver 2.2.8 is executed to perform the steady analysis of the aerodynamic component over its original shape. This is done in order to provide initial conditions for the next unsteady aeroelastic analysis. Depending on the particular case a steady analysis with fixed geometry could be sufficient. This is the case of the AGARD 445.5 wing since equilibrium is reached with symmetrical conditions;

3. The aeroelastic solver is restarted from the just computed steady-state solution and deformed mesh. The user has to specify the time law that describes the evolution of the prescribed displacements. For aerodynamic damping analyses this is usually an oscillatory time law (i.e. a (co)sinusoidal time law as for the SC10 cases). For the flutter computation used for aeronautical components, like for the AGARD 445.6 wing, modal shapes are enforced using an opportunely specified time law. This could be a rounded step time law allowing to excite a particular frequency range;

4. The time law is evaluated at the current physical time in order to compute the current values of displacements and velocities of the considered degrees of freedom. This is true both for rigid degrees of freedom and generalized modal displacements. This is required in order to compute aerodynamic mesh points displacements and faces velocities for the ALE formulation;

5. If the considered degree of freedom is a rigid movement then it is straightforward: a loop over the aerodynamic mesh wall points is performed to compute their displacements and velocities. In case of translations a uniform field of displacements and velocities is computed. In case of rotations, the rotation center has to be specified. If the considered degrees of freedom are represented by modal shapes, the computation of wall nodes displacements and velocities involves the use of the interface matrix between the structural mesh and the aerodynamic mesh. The modal shape matrix is also involved. First of all, using expression 2.32, structural nodal displacements and velocities are computed from generalized displacements and velocities. Then, aerodynamic mesh wall nodes displacements and velocities are computed from structural wall nodes displacements and velocities using relation 2.53;

6. The next step of the algorithm involves the update of the aerodynamic mesh internal points displacements and the update of internal faces velocities (for the ALE formulation). The former is accomplished using the IDW algorithm previously described in 2.2.6. The latter is explained in details in [127];

7. Knowing the updated internal nodes positions it is possible to update the mesh metrics: this involves the computation of new cell volumes, cell centers, face centers, face unitary unitary vectors. At the end of this step the new mesh is known;

8. Now the purely aerodynamic solution is computed over the updated mesh, using the ALE framework and the DTS strategy in order to exploit the same convergence acceleration techniques adopted for steady analyses (see 2.1.12). Using DTS formulation, purely aerodynamic explicit iterations are performed until aerodynamic

convergence. In particular, the aerodynamic solution is considered converged if about 3 orders of magnitude of aerodynamic relative residuals are lost. Eventually the solution is also considered converged if a maximum number of iterations is reached. The DTS formulation requires the solution of the previous time step to be stored;

9. In order to compute generalized loads, expressions 2.54 and 2.33 are employed;

10. The algorithm is repeated for the next physical time step;

An important aspect that must be underlined about forced oscillations analyses regards the computational point of view and GPUs executions. In order to perform unsteady aeroelastic computations, besides the adoption of the DTS strategy it would be also possible to advance the solution in physical time by deactivating LTS and other convergence acceleration techniques and use a global (very) small physical time step that satisfy CFL conditions everywhere in the computational domain. This way using a single explicit aerodynamic iteration it would be possible to advance the solution in physical time. However, this would also mean that for each aerodynamic iteration, the mesh would be updated. Since the interface matrix and the computation of the aerodynamic mesh wall displacements and velocities from structural mesh displacements and velocities are performed on the CPU, this would mean a CPU-GPU data exchange for each aerodynamic iteration. Thus, this strategy is not implemented in the solver since it would be a bottleneck for GPU executions. Furthermore, as already mentioned, when using an explicit global time step formulation very small physical time steps are required. The choice of the physical time step value is strictly related to the frequencies of interest. Despite some peculiar applications (e.g. acoustics), usually in aeronautical fields when studying wings and blades for flutter analyses, aerodynamic damping analyses and forced oscillations analyses the frequencies of interest are related to structural frequencies (and their fractions). The main advantage of implicit time step formulations over explicit formulations is the possibility to adopt higher physical time steps in order to reconstruct and analyze only the frequencies of interest, dissipating whatever is outside the frequency range. DTS is here adopted in order to keep an explicit formulation to exploit GPUs architecture and convergence acceleration techniques. At the same time this allows to use implicit-like physical time step values to save computational time that would be wasted if employing a fully explicit global time step strategy.

It must be noted that when employing this kind of aeroelastic analyses with the aim of flutter prediction through the computation of the H_{am} aerodynamic transfer function matrix, the effects of the rotating speed must be considered within the initial FEM computation. The $H_{am}(\Omega)$ matrix thus becomes a function of the angular speed Ω.

2.2.10 Free oscillations analyses

Free oscillations analyses are similar to the previously described trim and forced oscillations analyses. These are unsteady analyses in which the structure is free to respond due to its structural properties. This is similar to trim analyses since the wall displacements (and also velocities in this case) are not enforced by the solver by a user-provide time law (like what happens in forced analyses) but they are naturally obtained from the solution of the structural system 2.30, forced with aerodynamic loads. This

means that despite trim analyses, now also the structural mass matrix (and eventually the damping matrix) is involved in the solution of the structural system. The common points with forced oscillations analyses are the algorithms adopted after the computation of the rigid or generalized displacements and velocities. However, as said, this time the computation of rigid/generalized displacements and velocities is performed differently, through the solution of system 2.30. This kind of analyses is adopted for different purposes. In literature they are usually adopted for flutter analysis, although a more computationally efficient strategy is described in 2.2.3 using forced oscillations strategy and post-processing. The idea behind performing stability analyses using free oscillations is quite simple. First of all, an aeroelastic equilibrium configuration is computed. In order to asses if the aeroelastic system is asymptotically stable, simply stable (fixed amplitude oscillations, flutter) or unstable, starting from the equilibrium solution, a small perturbation is introduced in the system (e.g. enforcing small structural displacements or velocities as initial conditions for the unsteady analysis). This is the strategy adopted for the flutter analysis of the BSCW wing for the AePW2 8.3, where the aeroelastic solver is restarted from trim conditions with a small initial perturbation applied on the wing pitch speed. The wing is then left free and the evolution of pitch and plunge degrees of freedom is monitored. Multiple simulations have to be performed in order to find the exact dynamic pressure that leads to sustained pitch oscillations with constant amplitude (meaning null damping).

Here the procedure adopted to perform free oscillations analyses is described. As previously mentioned, this procedure has common aspects with trim and forced oscillations analyses.

1. First of all the degrees of freedom of the aerodynamic component are computed. This is exactly the same step performed with trim and forced oscillations analyses. The free degrees of freedom could be rigid displacements (like what happens with the BSCW wing of AePW2) or modal shapes (like the AGARD 445.6 wing). Anyway, each degree of freedom shape is coupled with its mass, stiffness and eventually damping structural values obtained from the FEM analyses and other sources for corrections (e.g. identification). If modal shapes are adopted as free degrees of freedom, the interface matrix between the structural mesh wall nodes and the aerodynamic mesh wall nodes is computed. This is computed only once for the entirely simulation;

2. As for the forced oscillations analyses 2.2.9, at this point, depending from the case under investigation, a simple steady-state purely aerodynamic analysis with fixed geometry could be performed, if (like what happens for the AGARD 445.6 flutter analysis) the aeroelastic equilibrium configuration is symmetrical. Otherwise, a static aeroelastic analysis (2.2.8) has to be performed to provide the correct initial conditions for the unsteady solver (like what happens for the flutter analysis of the BSCW wing in AePW2);

3. The aeroelastic solver is restarted from the steady/trim solution (by reading saved flow fields, original mesh and absolute displacements). Like with the trim solver, the structure is now free to deform due to aerodynamic loads and like with the forced oscillation solver, DTS and ALE face velocities are now employed. As with the trim solver, the first step is the integration of the aerodynamic loads over

the deformable walls. If rigid degrees of freedom are adopted, aerodynamic loads (pressure and stress wall fields) are integrated over the moving boundaries in order to compute forces and moments around the rotation center. If modal shapes are instead adopted as degrees of freedom, the operation is more complicated. In this case in fact, the first operation is using expression 2.54 to translate loads defined over the aerodynamic wall nodes to loads defined over structural wall nodes. This is performed thanks to the pre-computed interface matrix, using RBFs. Next, modal shape matrix is used to translate structural mesh wall nodes loads into generalized forces, see 2.33;

4. The aim of this step is the computation of the structural response under rigid or generalized forces. For this purpose, similarly to what happens inside the trim algorithm, the structural system with aerodynamic forces (2.30) has to be solved in order to compute rigid/generalized displacements and velocities. This operation is performed on the CPU and two options are available. The system can be solved using an implemented system solver or by coupling the aeroelastic solver with an external structural solver, like MBDyn, thanks to the socket-based interface. In any case, the system 2.30 is solved knowing the structural properties and the previously computed aerodynamic forces in order to obtain displacements and velocities. The degrees of freedom displacements and velocities solution at the previous physical time step is also required;

5. In this step aerodynamic mesh wall nodes displacements and velocities has to be computed. With rigid degrees of freedom this is straightforward since, with translations, displacements and velocities are uniform fields. With rotations the rotations center has to prescribed. The situation is more complicated with generalized displacements, where expression 2.32 has to be employed in order to compute structural wall nodes displacements. Then, using the interface matrix and expression 2.53, aerodynamic mesh wall nodes displacements and velocities are computed. At the end of this step aerodynamic mesh wall nodes displacements and velocities are known;

6. From aerodynamic mesh wall nodes displacements, internal nodes displacements are computed using the IDW algorithm described in 2.2.6. After this operation, internal faces ALE velocities can be computed. Furthermore, mesh metrics is updated;

7. Now the purely aerodynamic formulations 2.1.12, coupled with DTS formulation and ALE framework, are employed to obtain the solution over the updated mesh. DTS requires the aerodynamic solution of the previous time step. As mentioned in 2.2.9, thanks to DTS formulation, steady-state convergence acceleration techniques can be exploited to obtain the new aeroelastic solution. In particular, using DTS instead of a global physical time-stepping strategy, multiple purely aerodynamic iterations (with LTS) are performed before the next physical time step, thus before the next mesh update. Aerodynamic iterations are performed until residual convergence or until the user-specified maximum number of iterations is reached. Residuals are considered converged if 3 orders of magnitude are lost during explicit iterations;

8. The algorithm is repeated for the next physical time step until the user-specified physical final time is reached;

Few aspects have to be discussed regarding free oscillation unsteady simulations. First of all, as previously discussed in 2.2.9, free oscillations analyses could be performed using global physical time-stepping strategies instead of DTS. However, the problems are the same as for the forced oscillations analyses. With global physical time stepping, due to CFL requirements, the chosen physical time step is usually orders of magnitude smaller than the minimum requirements related to the frequency content of interest (excepts for some particular analyses like high speed impacts and acoustics). Thus, in order to obtain an implicit-like computationally efficient solver and allowing the aeroelastic simulation to proceed with the desired physical time step value, DTS is preferred. This is true for aeroelastic analyses, like flutter analyses, where low frequencies (in the order of structural frequencies) are involved in flutter mechanism. The second problem is related to the fact that, as previously said, with a global physical time-stepping strategy the aerodynamic mesh would require updates every explicit aerodynamic time step. Since data transfer between the CPU and the GPU would be required to perform this operation, DTS is again preferred over global physical time-stepping.

Another important aspect when performing this kind of simulation is related to the choice of the integration scheme for the solution of the system 2.30. In fact, as discussed in [104], different integration schemes possess different accuracy orders and accumulate a different amount of error regarding solution phase and amplitude. Different time schemes are implemented in the solver, like IE (Implicit Euler, very dissipative, requires very small time step to provide accurate solutions), BDF2 (Backard Difference Formula, 2nd order, more accurate than IE both on amplitude and phase) and CN (Crank Nicholson, 2nd order, no errors on amplitude and phase but could lead to numerical problems). Besides the implemented formulations to accomplish this purpose, the solver can be also coupled with external structural solvers like MBDyn thanks to the socket-based interface.

Finally, as better described in 8.3 with the example of the flutter analysis of the BSCW wing for the AePW2, a critical user choice is the value of the physical time step that is used both for the aerodynamic and structural side of the unsteady aeroelastic simulation. In fact, in this particular test case, as also noticed by other research groups, a dependency of the aeroelastic damping factor g from the chosen physical time step seems to be present. This means that, especially when performing flutter analysis with free oscillation strategy, it is fundamental to give attention to numerical parameters like the physical time step. Thus, a sensitivity analysis with respect to adopted numerical parameters should be performed to obtain reliable results. Performing flutter analyses with free oscillation strategy is often found in literature. However it is worth to remind that flutter analyses can also be performed using a forced oscillation strategy with a specifically tuned time law coupled with post-processing operations, as better described in 2.2.3.

As for the other kind of aeroelastic analyses, when performing turbomachinery or open rotors simulations, the effects of the angular velocity over the structural behavior must be taken into account [122].

CHAPTER 3

Turbomachinery and Open Rotors extensions

As said in the introduction, the main purpose of this work is to implement a GPU-accelerated general-purpose solver capable to handle complex aeronautical cases. However, when dealing with turbomachinery and open rotors, further formulations and numerical schemes are required. Turbomachinery and open rotors represent very different applications in the aeronautical world. Some formulations and schemes are required to correctly model the problem, such as the use of modified inlet boundary conditions (turbomachinery only) to enforce particular values of total pressure and temperature. Furthermore, when the solution is expected to be cyclic (both for turbomachinery and propfans), it is possible to exploit the intrinsically axial symmetry of the case to improve computational efficiency. In this case periodic boundary conditions can be adopted for a single-blade domain reduction. However, when performing unsteady analysis with IBPA values different from 0, the single-blade periodicity is broken. Nonetheless, the single-blade domain reduction can be still adopted by implementing a modified version of periodic boundary conditions, the so called "time-delayed" boundary conditions. These are capable to emulate the effects of the delayed oscillations of adjacent blades without the need to actually discretize them in the computational domain. This formulation is specifically designed for unsteady solvers working in the time domain rather than in the frequency domain (e.g. linearized or harmonic balance formulations). Another fundamental formulation is represented by the Multiple Reference of Frame (MRF) that allows rotating domain analyses, such as for turbomachinery and propfans, without the need to actually rotate the mesh.

3.1 Turbomachinery and open rotors

Here a brief introduction of the turbomachinery and open rotors fields is provided, focusing on the most important problems and current research areas. Nowadays the need of configurations that guarantee both high efficiency and performances is much important as other kind of requirements like: safety, acoustics, emissions. A good introduction to the nowadays turbomachinery problems is found at [61, 62] and here the most important concepts useful for this work are reported. Among the different challenges, that the aeronautical industry has to face, the most important are represented by:

- A design that leads to excessive high NOx emissions could not be accepted;
- A design that has high efficiency in a normal undistorted flow but produces excessive stress levels under asymmetric conditions could not be accepted;
- An aerodynamic efficient design will not be accepted if the associated noise levels are too high;

It is easy to understand that numerous aspects have to be simultaneously considered when designing a turbomachinery. This is also valid for open rotors, especially for what concerns the noise levels [122]. Blade vibrations due to resonance or flutter represent an hazard since the oscillatory behavior is strictly related to the fatigue life of blades. As an example, there have been reported cases of blades detachments in the past. Obviously this situation has to be avoided since it represents a serious safety problem. As [62] explains, during the early days of axial-flow compressors development, different configurations experienced dangerous vibrations at part speed (below the design point rotating speed) operations. After investigations, it was found that the oscillatory behavior could be attributed to flutter. Furthermore, flutter conditions could be also encountered after start-up, e.g. with high flight speed at low altitude conditions. In general, three phenomena related to blade vibrations can be considered:

- Resonance;
- Acoustic resonance;
- Flutter;

The analysis of turbomachinery resonance has many aspects in common with a typical helicopter rotor resonance investigation. Figure 3.1 shows a typical turbomachinery Campbell diagram. It is possible to recognize the similarities with a typical helicopter rotor case. From the high RPM values on the x axis it is clear that this is a turbomachinery case rather than an helicopter rotor. Anyway the concepts are the same: at the design point, resonance must be avoided. The problem here is that during the start-up procedure, the turbomachinery could encounter resonance due to the intersection between the lines representing multiples of the rotating frequency and the curves representing the modal frequencies (1F for first bending and 1T for first torsion). However, as for helicopters blades, resonance conditions are encountered for a very small time period, avoiding possible structural damages. Resonance is a phenomenon strictly related to structural properties, while "acoustic resonance" is instead strictly related to the

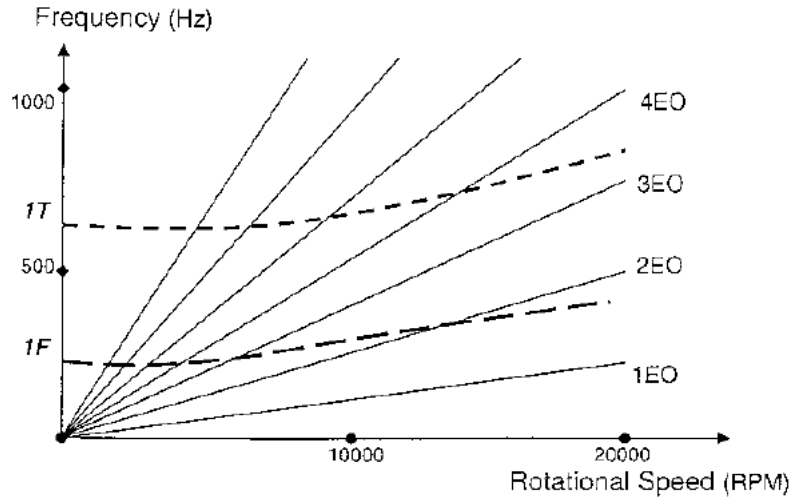


Figure 3.1: *Typical turbomachinery Campbell diagram [62].*

flow properties. Acoustic resonance could happen when conditions of subsonic axial flow are encountered and the geometric properties are such that:

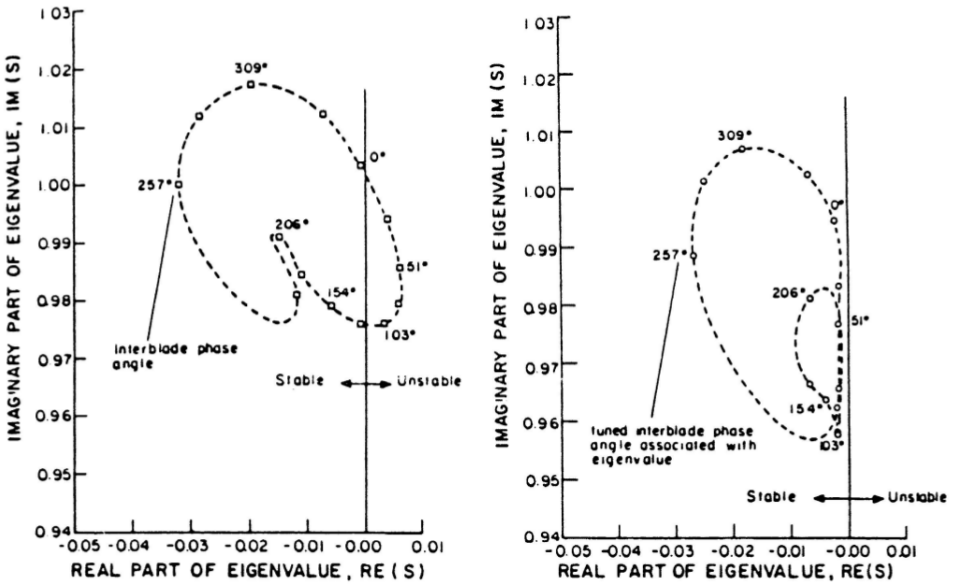
$$\begin{aligned} \frac{s}{V_p^+} &= \frac{2\pi v}{\omega} - \frac{\sigma}{\omega} \\ \frac{s}{V_p^-} &= \frac{2\pi v}{\omega} + \frac{\sigma}{\omega} \end{aligned} \tag{3.1}$$

where $V_p^{\pm}$ is the propagation speed of small perturbations in the two directions alongside the blade cascade, ω is the blade vibration frequency, σ is the Inter-Blade Phase Angle (IBPA) and v is a positive integer. The concept of IBPA will be explained in details later in 3.8 when discussing time-delayed boundary conditions. Roughly speaking it represents the phase angle related to the delay between the vibration of two adjacent blades. As for the resonance, acoustic resonance must be avoid because it could lead to a reduction of aerodynamic damping (see 3.4) and large vibratory stresses.

Finally, flutter is another source of vibratory stresses that could reduce the blades fatigue life, leading to fractures. Flutter, for definition, is the instability of the aeroelastic system, thus structural and aerodynamic subsystems have to be considered coupled when investigating such phenomenon. Though flutter could be investigated analyzing the stability of the aeroelastic system with classical methods adopted for aeronautical cases described in 2.2.3, for what concerns turbomachinery an energetic approach is usually adopted (see 3.4). If blades are disturbed aerodynamically, they tend to vibrate with their natural modes, with small amplitudes. This in turn leads to aerodynamic forces perturbations that influence the blades themselves. At certain conditions it is possible that this phenomenon is such that a net work is done on the blades and vibrations are self-sustained. The vibrations could increase from their initial small amplitude and lead to a limit cycle due to the nonlinearity of the phenomenon. In these conditions of self-sustained vibrations the blades fatigue life could be drastically reduced, leading to potential failures. The investigation of rotor stability using the concept of aerodynamic damping is discussed in 3.4 while the formulations to perform such kind of unsteady

simulations involving mesh deformation have been already discussed in 2. In this work the flutter analysis of a typical propfan configuration is performed in 10.4 using the numerical tools already presented in 2.2.3. In 10.1 and 10.2, the aerodynamic damping analysis of the 2D and 3D SC10 (Standard Configuration 10) is performed using the energetic concepts that will be described in 3.4. Besides this kind of stability investigations the literature does not present many cases of turbomachinery static aeroelastic investigation. Thus, in 10.3 the trim analysis of the NASA's Rotor 67, a well known axial rotor configuration, will be performed. Using the same tools presented in 2.2.8, this will allow to determine the possible advantages offered by taking into account blade deformability due to aerodynamic loads when computing turbomachinery performance curves (see 3.3). The possibility of extending such a technology also to turbomachinery applications is very important given the recent effort towards design and development of turbomachinery components with advanced, highly deformable materials (e.g. composite).

The importance of turbomachinery aeroelastic stability investigation is also confirmed by the recent researches regarding the concept of mistuning. Roughly speaking mistuning means breaking, at certain measures, the axial symmetry of the rotor by changing the blades geometry or structural properties. As explained in [61, 62, 109] mistuning can be exploited in order to improve the aeroelastic stability of a rotor or even stabilize a rotor that is unstable under certain conditions. As an example, figure 3.2 [62] shows the stabilization effects provided by an ad-hoc blades mistuning. It is



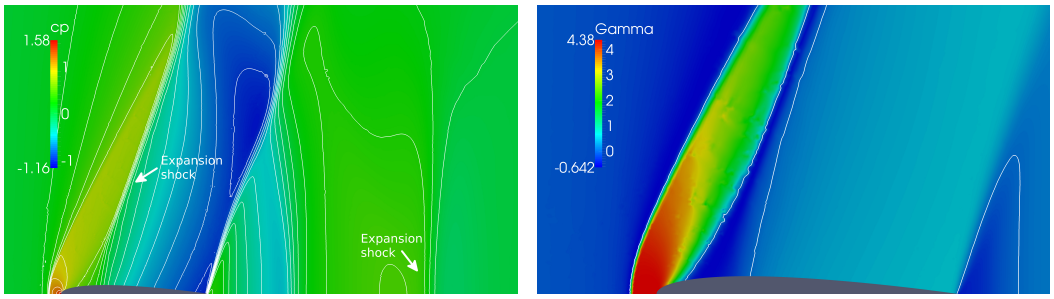
(a) Tuned configuration (b) Mistuned configuration

Figure 3.2: *Advantages that could be achieved with an opportune mistuning in turbomachinery [62]: on the left the real and imaginary part of the less stable eigenvalue at different IBPA angles with a tuned configuration, on the right with a mistuned configuration. With specific mistuning it is basically possible to stabilize the system for all IBPA angles.*

possible to see that after employing mistuning the system is stable, i.e. the less stable eigenvalue get negative real values for all IBPA angles. It must be noted, however,

that investigating this kind of asymmetries increases the computational effort of simulations. This is due to the fact that many domain reduction techniques, e.g. single blade domain reduction, could not be directly employed when non-null IBPA values are considered.

Another recent trend in turbomachinery is represented by Organic Rankine Cycle (ORC) applications [59, 76]. The idea here is the use of specific fluids and operating conditions to reach the dense gas region [66]. This is done mainly in order to obtain higher efficiency, thanks to the fact that within specific conditions the entropy jump across a shock is quite small. However, these applications represent a challenge from the point of view of numerical simulations. In fact, in dense gas conditions unusual phenomena could happen, like expansion shocks or mixed waves. Figures 3.3 [56, 66] show a simple NACA 0012 airfoil under specific dense gas flow conditions. It is clearly visible the expansion shock. As can be seen from the varying γ, for this kind of case



(a) C_P (b) γ

Figure 3.3: *BZT phenomena, expansion shocks [66] over a NACA 0012 airfoil at null angle of attack. On the left it is possible to see the C_P (pressure coefficient) jump from higher values to lower values (expansion shock). On the right it is highlighted the fact that for these particular dense gas region cases the specific heats ratio cannot be considered constant.*

the usual simple PIG model, for which $\gamma = CONST$ cannot be employed. In fact, the PIG model is specifically designed to model fluids inside the ideal gas region and is unable to represent the aforementioned phenomena. Thus, real gas models have to be adopted. A simple real gas model is represented by the Van Der Waals model that is capable to reconstruct expansion shocks and mixed waves when adopted in the dense gas region and at the same time provide results that are consistent with the PIG model when adopted in the ideal gas region. Many other real gas models have been developed and adopted in CFD solvers.

As for other aeronautical components like helicopter blades and wings, CFD has a fundamental role in blades geometry optimization. Optimization loops can be used in order to test different configurations until an optimum shape is found. Many different algorithms like genetic algorithms can be adopted for this purpose to search the values that minimize a prescribed optimization function. Basically, given a set of parameters, a particular shape can be built and a CFD simulation can be executed in order to find the value of the function to be optimized. As the optimization loop could require dozens of CFD simulations it is easy to understand that the bulk of computations is given by the numerical solution of the flow around the geometry. It is thus easy to understand that a fast and at the same time accurate solver is required in the early stages of the wing/blade

design. These two requirements are satisfied in this work through GPU acceleration and the implemented compressible (U)RANS formulations, allowing the solver to quickly and accurately reconstruct both compressible and viscous phenomena.

Alongside turbomachinery, open rotors are nowadays an interesting research field since they could represent the next generation of aircraft propulsion systems. A very good introduction of the open rotors world is represented by [122]. Roughly speaking, an open rotor is a gas turbine whose fan stage is not within the nacelle. As an example, figure 3.4 shows the experimental setup of the SR-5 propfan propeller investigated in this work. The 10-blade wind tunnel model was designed by Hamilton Standard in early 80's during the Advanced Turboprop Project [78]. Very different configurations

Figure 3.4: *SR-5 open rotor configuration [78].*

can be found such as pusher, where the propellers are mounted at the front of the engine and puller, where the propellers are mounted behind the turbine stages. Furthermore, Contra-Rotating Open Rotors (CROR) configurations have been developed, where, as the name suggests, two propeller rows rotates in different sense. The main advantage of this configuration is a reduced fuel consumption. This allows also a reduction of the rotor diameter and the possibility to obtain a certain thrust with respect to single propeller configurations. CRORs are usually adopted for pusher configurations while single propellers are generally employed in puller configurations. One of the most important problems of open rotors is represented by the produced noise. With CRORs the reduced angular velocity allows a reduction of the noise, but at the same time the aerodynamic interaction between the two propellers represents an additional source of noise. The concept of CROR is not new, as in 1975 NASA started investigating CROR capabilities. However, the research was slowed down due to noise problems and the reduction of the oil barrel price and thus the need for an high efficient solution for aircraft propulsion. Nowadays however, the open rotor concepts were resumed due to the need of fuel costs saving and the more stringent limitations on NOx emissions. The two main concepts to possibly achieve these targets are currently represented by Ultra-High Bypass Ratio engines and CRORs.

As for any aeronautical component, alongside performances investigation, aeroelastic investigations are required to guarantee safety in all operating conditions. Figure 3.5 shows the flutter boundaries of a typical open rotor configuration. As it is possible to

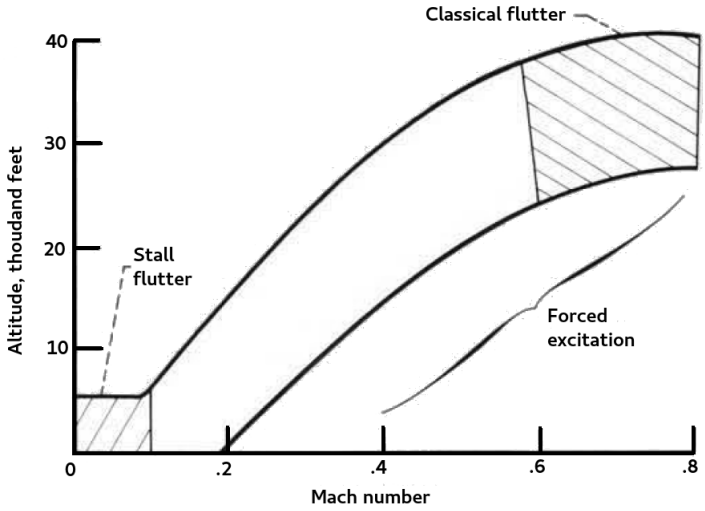


Figure 3.5: *Flutter boundaries for a typical open rotor configuration [78].*

see, at low Mach numbers stall flutter could occur due to separations. With relatively high Mach numbers, over 0.6, the classical flutter could occur. Problems related to forced excitation generally occur over the entire flight envelope due to unsymmetrical flow produced by gusts and fan-wing interactions. All of the presented aspects highlight the importance to perform numerical computation for this kind of trending configurations, both for what concerns steady-state solutions for performance assessment (see 9.3) and for what concerns flutter investigation (see 10.4).

Turbomachinery and propfans represent quite different aeronautical applications. Nonetheless, they share important basic physical mechanisms and thus many numerical formulations can be adopted for both applications. The purpose of this introduction was to briefly show some of the most recent trends in turbomachinery and open rotor fields, highlighting the fact that these kinds of configurations represent an important challenge from a numerical point of view.

Very specific applications like CRORs and ORCs are beyond the scope of this work. Here the focus is mainly posed on aeroelasticity. In particular, the target is represented by the steady-state and flutter computation of typical open rotors configurations and the steady-state, trim and aerodynamic damping analyses of turbomachinery cases.

3.2 Aerodynamics and modelling

One of the goal of this work is to implement a general purpose solver capable to handle classical aeronautical and turbomachinery/open rotors cases. In order to reach this target, from the point of view of the formulations purely related to aerodynamics, the focus is on a non-linear time-accurate solver. Here this choice is briefly explained alongside the introduction of other possible formulations with their advantages/disadvantages. As [62] suggests, alongside the non-linear time-accurate formulations adopted in this work, the other two strategies usually adopted for the turbomachinery CFD are represented by the time-linearized approach and the harmonic balance method. Both these formulations work in frequency domain, exploiting the temporal periodicity typical of

turbomachinery cases, with the aim of drastically reducing the computational effort.

3.2.1 Time linearized approach

This approach exploits the fact that usually in turbomachinery unsteadiness is relatively small when compared to the mean flow quantities. The idea is to decompose the flow in a mean flow plus an unsteady perturbation. This perturbation is supposed to be small, such that linearization is possible. Considering also the small perturbations to be periodic in time, it is possible to switch from time domain to frequency domain, exploiting Fourier series with spatially varying coefficients. The result is a set of partial differential equations in frequency domain called "time-linearized" equations.

Let us consider the 2D Euler equations for simplicity:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{f}_x}{\partial x} + \frac{\partial \mathbf{f}_y}{\partial y} = 0 \tag{3.2}$$

where $\mathbf{f}_x$ and $\mathbf{f}_y$ represent the convective fluxes. It is possible to decompose the solution in a mean ($\bar{\mathbf{U}}$) and unsteady small periodic perturbations ($\mathbf{U}'$) components:

$$\mathbf{U}(\mathbf{x}, t) = \bar{\mathbf{U}}(\mathbf{x}) + \mathbf{U}'(\mathbf{x}) e^{j\omega t} \tag{3.3}$$

Substituting 3.3 in 3.2 the first order terms give:

$$j\omega \mathbf{U}' + \frac{\partial}{\partial x}\left(\left.\frac{\partial \mathbf{f}_x}{\partial \mathbf{U}}\right|_{\mathbf{U}=\bar{\mathbf{U}}} \mathbf{U}' \right) + \frac{\partial}{\partial y}\left(\left.\frac{\partial \mathbf{f}_y}{\partial \mathbf{U}}\right|_{\mathbf{U}=\bar{\mathbf{U}}} \mathbf{U}' \right) = 0 \tag{3.4}$$

The zeroth-order non linear steady equations that provide the mean flow $\bar{\mathbf{U}}$ are solved using usual non-linear CFD methods. The time-linearized equations are basically steady equations since there is no dependence from time t and can be thus solved with the same convergence acceleration techniques presented in 2.1.10. It is thus possible to add a pseudo time derivative term [62] and advance the solution in pseudo time until convergence to a steady-state condition, where the pseudo time term is null.

The main disadvantage of the linearized approach relies in the linearization itself: if unsteadiness is high the strategy cannot be employed. However, a non-linear frequency domain formulation is possible: the harmonic balance technique. The main advantage of the time-linearized approach is nonetheless its high computational efficiency since the equations are linearized and thanks to the Fourier transformation the temporal dependency is avoided.

3.2.2 Harmonic balance

This technique allows a fully non-linear frequency domain approach. As for the time-linearized approach the flow is considered periodic in time. For turbomachinery cases the flow could be also periodic in space or a certain time delay related to the IBPA concept can be considered. This will be better explained in 3.8. The idea behind harmonic balance is to express the solution variables as a Fourier series. As an example the density can be expressed as:

$$\rho(\mathbf{x}, t) = \sum_n R_n(\mathbf{x}) e^{j\omega n t} \tag{3.5}$$

this is done also for momentum and energy. The coefficients maintain the spatial dependency and the series are truncated after N terms. The next step is to substitute the series in the equation 3.2. This way an expanded Fourier series is obtained and terms can be grouped by frequency since each frequency component must vanish. The resulting equations have the following form:

$$\frac{\partial \tilde{\mathbf{f}}_{\mathbf{x}}(\tilde{\mathbf{U}})}{\partial x} + \frac{\partial \tilde{\mathbf{f}}_{\mathbf{y}}(\tilde{\mathbf{U}})}{\partial y} + \tilde{\mathbf{S}}(\tilde{\mathbf{U}}) = 0 \tag{3.6}$$

where $\tilde{\mathbf{U}}$ is the vector of the Fourier coefficients of the conservative variables. The coefficients related to convective fluxes are nonlinear functions of $\tilde{\mathbf{U}}$. The computation of these fluxes is difficult and computationally expensive and difficulties arise when turbulence models are also considered for a viscous approach. Further details can be found in [62].

One of the reasons behind the choice of the non-linear time-accurate approach is represented by the fact that AeroX should be a general purpose solver, aimed to be used also for classical aeronautical cases like wings and aircraft. This means that it should be able to process both steady and unsteady cases with or without any temporal of spatial periodicity. Frequency-based formulations have the advantage of allowing to perform single frequency simulations with computational costs in the order of an equivalent steady-state simulation. The same simulation with a time-accurate solver would require a computationally expensive full unsteady simulation. However, as explained in 2.2.2, with a single time-domain simulation it is possible to excite a wide range of frequencies, exploiting a blended time step displacement law. In this case, when investigating flutter in classical aeronautical cases, non-linear time-domain formulations reveals their true power.

When performing turbomachinery simulations, different levels of discretization and modelization can be employed for the same case. Let us consider a multi-stage axial compressor. For the same case it is possible to perform a 2D single-row single-blade steady-state inviscid simulation or a full 360° multi-row multi-stage unsteady viscous simulation. Obviously a trade-off between results accuracy and computational costs is required in order to obtain results that are meaningful from an engineering point of view and that can be obtained in a reasonable amount of time. Different domain reduction techniques can be employed in order to avoid the discretization of the whole rotor/stator rows while retaining spatial periodicity effects. As an example, if a 1-blade spatial periodicity is supposed, then discretizing the whole blade row is potentially a waste of computational power. In fact a single blade can be discretized and the so-called periodic boundary conditions can be employed to emulate the presence of the other blades without actually discretizing them. AeroX implements multiple computational domain reduction techniques for turbomachinery analyses, such as periodic boundary conditions 3.7, time-delayed boundary conditions 3.8, mixing plane, MRF 3.6. All these techniques will be described in this chapter. The same techniques can be also exploited for open rotors.

3.3 Turbomachinery performance map

Turbine and compressors are made by rotor and stator blades. They can be axial or centrifugal and made by multiple stages/rows. The so-called characteristic curves or performance maps represent the most important indicator of the performances that they can guarantee. Here a brief introduction regarding turbomachinery performances is provided. For more details, e.g. velocity triangles, notation for angles..., the reader is referred to [61, 62]. Figure 3.6 shows a typical compressor stage performance map. Different curves appear in the figure. The x-axis is usually used for the mass flow. It

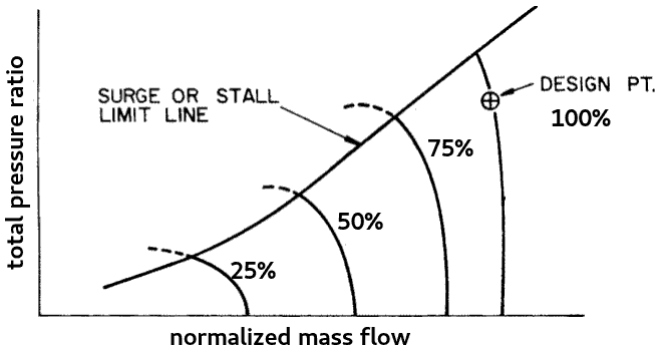


Figure 3.6: *Typical compressor performance map, total pressure ratio, [62] (here modified). The vertical curves represent operating regimes with constant angular speed (in % with respect to the design angular speed).*

can be the true mass flow or, like in the picture, a normalized mass flow. One normalization is usually represented by dividing the mass flow for the choke mass flow. This way the normalized mass flow at choke conditions is 1 and for higher total pressure ratio is < 1. The different vertical curves represent operational conditions at different constant rotation speeds of the rotor, moving to the right for increasing speeds. The y-axis in this case is represented by the total pressure ratio between compressor inlet and outlet. Let us consider a single curve with constant angular velocity. What happens is that moving from the bottom to the top of the curve, the mass flow is reduced while the incidence on the blade, the loads on the blade, the work performed by the blade on the flow are increased. This allows to obtain a certain value of the total pressure ratio. It can be seen that the design point is located on the 100% rotational speed curve and for high total pressure ratios. If, for some reasons, e.g. disturbances, the incidence on the blades is increased, this could be high enough for stall conditions to appear. The surge/stall limit line represents this limit for different angular speeds. It is important for the design point to stay at a sufficient distance from this line. Numerical simulations are usually concerned with the computation of the 100% angular velocity curve. In this work, for example, the performance curves of the NASA's Rotor 67 fan are computed, see 10.3. Alongside the total pressure ratio curve, the total temperature ratio and efficiency curves are usually computed. From the numerical point of view computing a compressor performance map means performing multiple steady-state simulations, one for each point used to discretize the curve. This is usually done by enforcing total temperature, total pressure and flow velocity direction on the inlet and changing boundary conditions on static pressure on the outlet. In AeroX inlet boundary conditions of this kind are enforced using what explained in 3.9. On the outlet, instead,

for each performance point two strategies could be employed. It is possible to start by enforcing on the outlet a static pressure equal to the inlet total pressure, to simulate choke conditions. Then the outlet static pressure is increased and when the simulation is converged a new performance curve point is found. This is done until stall conditions are reached. This is the strategy adopted in this work as it is quite stable and easy to implement. The main disadvantage is represented by the fact that it is more difficult to discretize the curve near stall, where for little outlet static pressure changes an high mass flow change is obtained. The other strategy is represented by an outlet boundary condition that allows to directly enforce a user-defined mass flow value. Differently from the first strategy, this allows an easy performance curve discretization near stall conditions. However, it comes with difficulties near choke conditions where the performance curve is basically vertical. The main disadvantage of the enforced mass flow boundary condition is represented by the fact that it is difficult to enforce this quantity on an explicit solver that adopts a ghost cell approach for boundary conditions and use conservative variables. Instead, this strategy is straightforward in other formulations, e.g. with pressure-based solvers like in [102]. It must be noted that performance map computations are somehow the equivalent of the steady $CL - \alpha$ curve of an aircraft or wing: it is computed supposing effective steady-state conditions without any relation between different points, i.e. without supposing any kind of transition between one point and the next one. When blade dynamics is considered, new problems like resonance and flutter can be investigated.

3.4 Turbomachinery aeroelasticity

Steady-state aeroelastic simulations, i.e. trim simulations, represent a very important step for a complete aeroelastic investigation with classical aeronautical cases. However, in turbomachinery literature it is very difficult to find such kind of simulations. The focus is usually on flutter analyses exploiting the so-called energetic approach. One of the purposes of this work is to perform the trim of a typical turbomachinery configuration. In particular, the NASA Rotor 67 axial fan rotor trim is investigated in order to assess if performing this kind of simulations is effectively useful to improve results accuracy. As noted in [122] for an open rotor configuration, as an example, the effects of aerodynamic loads are basically negligible with respect to centrifugal effects, in such a way that a steady-state purely aerodynamic solution is sufficient to achieve good results accuracy. A good resource regarding the analytic and numerical modelization of a typical open rotor blade can be found in [122]. Another good resource regarding the structural modelization of turbomachinery can be found in [61]. Anyway, it must be noted that the structural modelization of open rotors and turbomachiney rotors is quite similar since in both cases the in-vacuum blade deformation has to be computed and modal shapes are computed considering centrifugal effects. It must be noted that as for helicopter blades, the in-vacuum turbomachinery/open rotors mode frequencies depend from the rotating speed. This can be easily viewed in the Campbell diagram 3.1 where a parabolic trend is obtained by increasing the rotational speed from $0\,RPM$. This behavior can be modeled as follows:

$$\omega_n^2(\Omega) = \omega_{n,0}^2 + K_n \Omega^2 \tag{3.7}$$

This basically means that for a certain mode n, at a certain angular speed Ω, the square of the modal frequency is basically given by the sum of the squared non-rotating frequency $\omega_{n,0}$ plus a term that depends from the squared angular speed multiplied by a certain factor K_n. Aeroelastic investigations are usually performed at a certain angular speed. Changing the speed requires to perform again the computations of the resulting modal shapes and frequencies. An important aspect is given by the fact that when performing aeroelastic analyses with the concepts introduced in 2.2.3, the aerodynamic transfer function matrix H_{am} must be considered a function of the angular velocity Ω, $H_{am}(\Omega)$. This is valid both for for what concerns dynamic stability analyses and trim analyses since the structural behavior becomes dependent from the effects given by blades rotation.

As said, for safety reasons it is necessary to guarantee that in usual operating conditions flutter is not present. However it is possible that this kind of self-excited phenomenon is encountered when operating far from the the design point. Dowell [62] offers a very good picture of the problem. Figure 3.7 shows a typical compressor map with flutter boundaries. Although flutter conditions can be classified from a qualita-

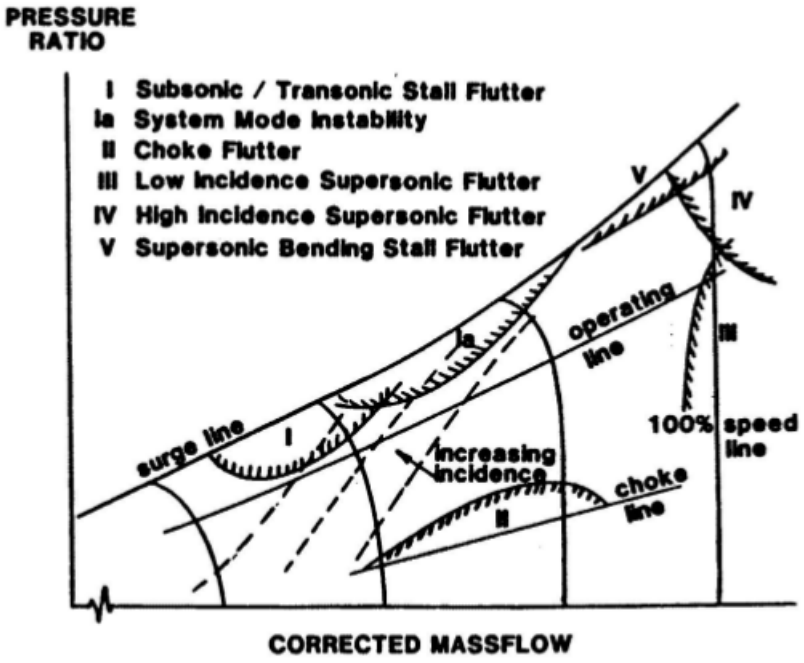


Figure 3.7: *Flutter boundaries for a typical compressor map [62].*

tive point of view based on the flow compressibility and viscous phenomena, it is still defined as the instability of the aeroelastic system. Region I is related to the subsonic/transonic stall flutter. In these conditions the mass flow is reduced while the blades incidence is high. Separations could occur in the flow. Rotating stall happens at part-speed conditions and is characterized by a circumferentially asymmetric flow in axial compressors. In these conditions, regions of reduced or reversed flow rotate in the same direction of the rotor speed but slower. These regions could coalesce into patches. The conditions at which this phenomenon occurs and the number of patches still represent a challenge from a numerical point of view. What is more important is that the rotating

stall is a strictly aerodynamic phenomenon, there is no dependency from the structural properties of the rotors. However, the unsteady flow that derives from rotating patches is translated to unsteady periodic loads on the blades, thus vibrations. This could lead to resonant conditions or self-excited oscillations (flutter) if certain structural and aerodynamic conditions are met. Stall flutter is a deeply nonlinear phenomenon and usually the related blade shape vibrations follow the shape of the first torsional mode. Region Ia is related to the so-called system mode instability. In these conditions the blade incidence and load are high. Nonetheless, separations are not the essential part of the flutter mechanism. Instead, what happens is that the Mach number over the blades could exceed the value of 1, leading to oscillatory shocks. Blade vibrations could combine to oscillatory shocks in such a way to pump energy into the system, leading so self-sustained oscillations, i.e. flutter. Region II is related to choke flutter. At choke conditions the inlet total pressure is in the range of the outlet static pressure and the blade incidence is quite low with respect to operating conditions with a given angular speed. Choking flutter usually occurs in the middle stage of multi-stage compressors at part-speed conditions. The mechanism of choke flutter is not fully understood but is related to both compressibility effects and separations. What happens is that blade vibrations and separations could change the throat location (related to the minimum flow area between adjacent blades where the flow reaches sonic conditions) in such a manner to pump energy into the system and sustain the oscillations. Region III is related to low incidence supersonic flutter. At these conditions the axial component is subsonic and the blades incidence is low. As figure 3.7 suggests these conditions could be encountered following the operating line when overspeed conditions are met, i.e. when the angular velocity exceed the design limits. Region IV is responsible for high incidence supersonic flutter. Here the blades incidence is high and the pressure ratio is over the normal operating conditions. Blade vibrations coupled with the presence of in-passage oscillatory shocks are responsible for the instability mechanism. Thus, compressibility effects are fundamental here. Finally in region V, supersonic bending flutter, is again characterized by high compression ratio. This region is near the stall line and the instability mechanism involves the stall phenomenon. Here both compressibility and viscous effects are fundamental as alongside with stall a detached bow shock is present at each blade passage entrance. From this flutter qualitative introduction it is possible to see that when performing aeroelastic investigations on turbomachinery, both compressibility and viscous effects are important. Thus, to accurately simulate this kind of phenomena a compressible RANS solver is required. Furthermore, complex phenomena like oscillating shocks and interactions between shocks and separations suggest the use of a full non-linear solver rather than a linearized formulation.

As explained in [62] turbomachinery blades are usually made of aluminum alloys, steel or stainless steel, and recently also titanium and beryllium. When considering aeroelastic phenomena like flutter and forced oscillations the fluid/structural mass ratio is defined as [98]:

$$C_{mass} = \frac{m}{\rho\pi\left(\frac{C}{2}\right)^2} \tag{3.8}$$

where m is the mass per unit span of the blade/arifoil, ρ is the air/gas density, C is the chord length. The mass ratio in turbomachinery applications can be such that the critical mode and frequency may be taken to be one or a combination of the modes cal-

culated/measured in vacuum. As explained in [109] the mass ratio of turbomachinery blades is generally higher than those of airplane airfoils. As noted in [61, 105], one of the main consequences is that since usually the instability involves a single d.o.f. per blade (torsional or bending) and with the aforementioned kind of materials aeroelastic modes are basically equal to in vacuum modes, the so-called "assumed-mode" unsteady aerodynamic analysis can be employed. The idea is basically to evaluate the net work done by aerodynamic loads by enforcing blade oscillations with certain structural frequency and amplitude. The aerodynamic damping basically represents a normalization of this net work and is positive for stable motions. Assuming a sinusoidal blade motion, it is possible to calculate the energy input per cycle due to aerodynamics and blade displacements as follows [61]:

$$W = \int_T F_g \, dx_g = \int_0^{\frac{2\pi}{\omega}} F_g \left(t\right) \, \dot{x}_g \, dt \tag{3.9}$$

where F_g is the force acting on its work-conjugate displacement x_g, T is the oscillation period and ω the correspondent pulsation. It is possible to rewrite the work by defining a non-dimensional coefficient, C_w:

$$W = \frac{1}{2}\rho V^2 c^2 \Delta r C_w \tag{3.10}$$

where ρ is the flow density, V the reference flow speed, Δr the reference span length and C_r is the non-dimensional work coefficient. Considering a plunge or pitch movement it is possible to write the aerodynamic damping as follows:

$$\begin{aligned} \xi_h &= -\frac{C_w}{\pi h^2} \\ \xi_\alpha &= -\frac{C_w}{\pi \alpha^2} \end{aligned} \tag{3.11}$$

for the plunge and pitch movements respectively. h and α represent the plunge and pitch oscillation amplitudes respectively. The aerodynamic damping is basically a normalization of the aforementioned work per cycle coefficient for the considered displacement amplitude. The aerodynamic damping may become negative in some flow conditions, forcing the structure into an unstable mode. Aerodynamic loads are usually small when compared to inertial and elastic forces. However, if the blades are such that structural damping is very low, the negative aerodynamic damping could be higher in magnitude, leading to unstable behavior. It must be noted, as showed in [109] that despite the fact that the final value of the aerodynamic damping is obtained through an integration over the entire blade surface, different blade regions could locally contribute to instability while others to stability. The aerodynamic damping investigation could be questionable in some modern applications, especially when using composite materials or high aspect ratio fan blade. As explained in [109], in fact, with small mass ratio values the aerodynamic coupling cannot be neglected in the aeroelastic behavior. As an example, considering the same blade, with titanium a value of $C_{mass} = 105.8$ is obtained, while with composite materials a value of $C_{mass} = 36.3$ is obtained, which is multiple times smaller. Wings airfoils typically have one order of magnitude smaller mass ratio values with respect to turbomachinery blades [98]. Nonetheless, the energy approach is very

simple from a computational point of view: once the purely structural blade modes are identified (accounting also for rotating effects), an unsteady simulation is performed, enforcing periodic oscillations with a specified mode. Then, at post-processing, input oscillations displacements and output loads are multiplied together and integrated following definition 3.9. This way the searched value of the aerodynamic damping (after normalization) is obtained. Obviously, the entire procedure has to be repeated for each IBPA (see 3.8) value under investigation. This approach to flutter investigation is adopted in this work to asses the stability of the SC10 2D and 3D configurations, as showed in 10.1 and 10.2 respectively. As can be seen, this strategy implies certain hypotheses that could be quite limiting. Thus, when performing the flutter investigation of the SR-5 propfan in 10.4, the classical aeronautical strategy 2.2.3 is instead adopted, considering the true aeroelastic modes, taking into account both centrifugal and aerodynamic effects.

3.5 Turbomachinery and open rotor formulations

In the following sections, formulations specifically designed for turbomachinery and open rotor cases will be presented. It must be noted that with the formulations presented in chapter 2 the solver is potentially already capable to perform such kind of simulations. The following formulations represent strategies to drastically reduce (in terms of orders of magnitude) the computational effort required by cases where axial symmetry can be exploited. For each formulation the related justification of the computational effort reduction is provided.

3.6 Moving Reference of Frame

The Moving Reference of Frame (MRF) formulation is used to allow the solver to perform steady-state or unsteady simulations with rotating domains without the necessity to actually rotate the mesh, reducing the computational effort due to the avoided mesh metrics update. Without MRF, in fact, one idea to perform computations with rotating domains would be to perform an unsteady simulation employing mesh deformation (simple rotation) and ALE formulations. Despite the fact that this strategy would require an expensive unsteady simulation with DTS even when a steady-state solution is searched, at each physical time a mesh update would be required, further increasing the computational costs. The formulation adopted in this work is based on [46] where it is possible to find the full mathematical procedure. Here, just the conclusions are presented and, in particular, the terms that must be added to the solver to fully implement the formulation. A fundamental advantage of this strategy is its easy implementation from a computational point of view. In fact, it exploits the already implemented ALE formulation and just requires an additional source term and a modification of the wall boundary conditions. Since one of the purposes of the solver is to perform dynamic aeroelastic computations, ALE is already implemented. Another advantage of this MRF formulation is that, since it exploits ALE formulation, the solution is always computed in absolute reference of frame. This way, momentum obtained from the conservative solution directly represents the complete vector, considering the flow velocity relative to the blade plus the rotation speed of the entire domain. The relative momentum/speed/Mach number can be obtained by post-processing the solution, knowing the

MRF velocity of each cell.

3.6.1 Exploiting ALE formulation

Following the guidelines of [46] the ALE framework already implemented for mesh deformation is exploited for MRF. This means that considering equation 2.16, the term $\mathbf{v}$ for mesh face i is computed as follows:

$$\mathbf{v_i} = \boldsymbol{\Omega} \times (\mathbf{C_i} - \mathbf{O_\Omega}) \tag{3.12}$$

where $\mathbf{v_i}$ is the ALE velocity of face i, $\boldsymbol{\Omega}$ is the rotor angular velocity, $\mathbf{C_i}$ is the face center position, $\mathbf{O_\Omega}$ is the origin of the rotation (due to the cross product it is important only the distance from the face center to rotation axis). This is computed for both internal and boundary faces. It is important to say that with the aim of convective fluxes computation, just the face normal component of $\mathbf{v_i}$ is important.

Alongside ALE face velocities, boundary conditions have to be modified consistently. Basically, wall velocity has to be changed in order to be consistent with the rotation:

$$\mathbf{u_i}|_{wall} = \boldsymbol{\Omega} \times (\mathbf{C_i} - \mathbf{O_\Omega}) \tag{3.13}$$

where $\mathbf{u_i}|_{wall}$ is the boundary condition value on the boundary face i, $\boldsymbol{\Omega}$ is the rotor angular velocity, $\mathbf{C_i}$ is the boundary face center, $\mathbf{O_\Omega}$ is the origin of the rotation. In this case, with respect to $\mathbf{v_i}$, also the tangential face component is important in viscous cases.

3.6.2 Source terms

To complete the implementation of MRF, a source term must be added on momentum equations. This is represented by:

$$\mathbf{S_{\Omega,k}} = -\boldsymbol{\Omega} \times \mathbf{m_k} \tag{3.14}$$

which is added to the momentum residuals after a multiplication for the k-th cell volume. $\mathbf{m_k}$ is the momentum solution inside cell k.

3.6.3 Few considerations

It must be noted that when performing unsteady simulations with mesh deformations and rotating domains, like for the propfan flutter investigation in 10.4, the contributions of MRF and mesh deformation are added together to compute the final face velocity:

$$\mathbf{v_{i,MRF+defo}} = \mathbf{v_{i,MRF}} + \mathbf{v_{i,defo}} = (\boldsymbol{\Omega} \times (\mathbf{C_i} - \mathbf{O_\Omega})) + \mathbf{v_{i,defo}} \tag{3.15}$$

where $\mathbf{v_{MRF}}$ is the ALE contribution given by the MRF strategy while $\mathbf{v_{defo}}$ is the contribution given by mesh deformation. Wall boundary conditions are also adjusted accordingly.

As mentioned, with MRF momentum is computed in an absolute reference frame. Often, in turbomachinery, the relative Mach number field is required (as for post-processing purposes in Rotor 67 test case, see 10.3). This can be easily computed:

$$\mathbf{u_{k,rel}} = \mathbf{u_k} - \mathbf{v_{k,MRF}} \tag{3.16}$$

where in this case $\mathbf{v_{k,MRF}}$ is the rotation velocity of cell k due to MRF. The cell velocity is computed using 3.12 but with the cell center location instead of the face center location. Anyway, relative components are computed by post-processing the solution, since the absolute velocity is directly computed from density ρ_k and (absolute) momentum $\mathbf{m}_k$ conservative solutions.

Furthermore, it must be considered that concerning its GPU implementation, MRF is quite efficient. Boundary conditions are just modified in their values when building the ghost cell and this is just a matter of pre-processing. The ALE formulation is in general computationally efficient since face velocities are required by the convective fluxes kernel that has an high floating point operations to global memory accesses ratio (see 6.2). Finally the source term requires few floating point operations and is computed in the same kernel adopted for turbulence models source terms computation.

3.7 Cyclic boundary conditions

Cyclic boundary conditions allow to reduce the whole $360°$ rotor/stator domain to an arbitrary number of blades sectors. In fact, if for certain flow/geometry conditions it is possible to hypothesize that the solution will be N-blade periodic, simulating dozens of blades is just a waste of computational resources, both in term of memory and floating point power. The idea is to reduce the $360°$ computational domain to an N-blades sector and enforce the same solution on the two periodic boundaries. Usually a perfect 1-blade periodicity is supposed, reducing the aerodynamic mesh to a single-blade domain. This is done especially for steady-state simulations when design conditions are investigated and the flow is supposed to be free from strong disturbances. However, there are complex phenomena, such as rotating stall, where the flow forms stalled regions that can coalesce and rotate along the blade cascade. In this case a single-blade reduction cannot be employed. The idea behind periodic boundary conditions is quite simple. Consider as an example the 2D Goldman turbine blade mesh of figure 9.1(a). The solution on the top boundary must be the same of the solution on the bottom boundary if the considered domain is representative of a single-blade periodic solution. The cyclic boundary conditions apply on those two boundaries. AeroX uses a ghost cell approach to implement boundary conditions, allowing to use convective fluxes in the same way adopted for internal faces. Thus, in AeroX, basically periodic boundary conditions are adopted to find the ghost cell values of each periodic boundary, using the solution on the other side of the domain. Since the solver implements an explicit time advancing scheme, it is possible to use the solution of the previous pseudo time iteration of the internal cells of one periodic boundary and enforce it as the ghost cell value on the other periodic side. From an analytic point of view periodic boundary conditions can be expressed as follows, for steady-state conditions:

$$\mathbf{U}(\mathbf{x})|_{\mathbf{x}\in b_1} = \mathbf{U}(\mathbf{T}(\mathbf{x}))|_{\mathbf{T}(\mathbf{x})\in b_2} \tag{3.17}$$

where $\mathbf{x}$ is a location on the first periodic boundary b_1 and $\mathbf{T}$ is the operator that applied to that location returns the correspondent location on the other periodic boundary b_2. As said, AeroX stores the solution inside cells (CC-FVM), while boundary conditions are enforced using ghost cells. Thus, equation 3.17 cannot be directly used. Basically, when assembling convective fluxes on a face $f \in b_1$ two solutions are required: the one

provided by the internal cell attached to f and the one provided by the ghost cell. The value of the ghost cell is taken from the internal cell related to the face correspondent to f but located on boundary b_2. This location is obtained with the transformation operator $\mathbf{T}$. This is easy when both periodic boundaries are discretized in the exact same way, i.e. with faces of same size and correspondent location. Problems arise when the two boundaries are discretized differently. In this case an interpolation is required in order to build the ghost cell value. Basically, the solution on the ghost cell related to face f on boundary b_1 is computed as follows:

$$\mathbf{U}^{GC}_{f \in b_1} = \sum_{i=1}^{N_{b_2}} \mathbf{U}^{CELL}_i \, W_{f,i} \tag{3.18}$$

where N_{b_2} is the total number of faces on boundary b_2, $\mathbf{U}^{CELL}_i$ is the solution of the internal cell related to face i on boundary b_2 and $W_{f,i}$ is the weight related to face i on boundary b_2 and face f on boundary b_1. This weight is computed from the overlapping area between faces f and i. Obviously, in order to speed up computations, the sum related to face f is performed only on faces i having a non-null weight $W_{f,i}$. This is implemented using addressing arrays that for each face f associate the list of faces i with non-null weights $W_{f,i}$. The computation of addressing and weights arrays is performed in pre-processing by OpenFOAM. The interpolation, addressing and weights are performed in the same way also for 3D cases. The main differences between 2D and 3D cases is given by the fact that in 2D cases the two periodic boundaries are related to a translation transformation. In 3D, instead, the transformation from boundary b_1 to b_2 is also related to a rotation since the single-blade sector has a specific periodicity angle. As an example, a single-blade sector of a 4 blades rotor has a periodicity angle of $90°$. For an 8 blades rotor the single-blade sector angle is $45°$ and so on. The problem is that when interpolating the solution from one boundary to build the ghost cell solution of the other boundary, the momentum solution of the previous pseudo time must be opportunely rotated to obtain a consistent boundary condition. This implies that for each periodic boundary a consistent rotation tensor must be built and used to correct momentum after the interpolation procedure:

$$\mathbf{m}^{GC,1}_f = \mathbf{R}_{b_1} \mathbf{m}^{GC,0}_f \tag{3.19}$$

where $\mathbf{m}^{GC,0}_f$ is the ghost cell momentum obtained with the interpolation 3.18, $\mathbf{m}^{GC,1}_f$ is the momentum corrected with the rotation, $\mathbf{R}_{b_1}$ is the rotation tensor for boundary b_1. The two periodic boundaries have different rotation tensors since the relative rotation angles are one the opposite of the other. The rotation tensor for each boundary is computed in pre-processing by OpenFOAM.

3.8 IBPA and time-delayed boundary conditions

Time-delayed boundary conditions [128] represent an ad-hoc modification of periodic boundary conditions in order to handle unsteady cases with non-null IBPA values (usually denoted by σ). The computational advantage provided by these boundary conditions is that they allow to exploit the single/N-blade domain reduction. First of all, let us discuss the concept of IBPA introduced by Lane [93] and explained also in [61]. IBPA

is the core of the so-called traveling wave approach. Roughly speaking the Inter-Blade Phase Angle (IBPA) is the phase angle related to delay of the periodic oscillations of two adjacent blades. If all the rotor blades are oscillating with $\sigma = 90°$ this means that two adjacent blades in the row are oscillating with the same period T, shape and amplitude but with a time delay of $T/4$. If $\sigma = 180°$ then the delay is $T/2$ and so on. Despite the fact that numerical simulations could be carried out with arbitrary IBPA values thanks to time-delayed boundary conditions, the fundamental IBPA value is:

$$\sigma = \frac{2\pi}{N} \tag{3.20}$$

where N is the total number of row blades. IBPA is the non-dimensional spatial frequency of a periodic disturbance that travels circumferentially down-rotor or up-rotor blade to blade. IBPA is probably the most important parameter in turbomachinery aeroelasticity. Let us consider, as an example, a rotor with N blades and 1 vibration d.o.f. (e.g. blade torsion) for each blade. If blades are identical and the hub and shroud are supposed to be perfectly rigid, a total number of N degrees of freedom is found in the structural system. With a modal analysis, a total number of N modes with the same frequency would be found since there is no possibility for each blade to communicate with any other from a structural point of view (while this is still possible with aerodynamics). The allowable N discrete values of IBPA are ($n = 0, 1, ...N - 1$):

$$\sigma_n = \frac{2\pi n}{N} = 0, \frac{2\pi}{N}, ..., \frac{2\pi\,(N-1)}{N} \tag{3.21}$$

This can be also re-written considering the angles from $-\pi$ to $\pi - \frac{2\pi}{N}$, associating positive values to the forward traveling waves (i.e. in the direction of rotation) and negative values to the backward traveling waves. The so-called traveling wave approach assumes that all blades are equal to each other due to cyclic symmetry, that all blades vibrate harmonically with the same frequency and, as said, a delay σ from one blade to the next one is taken into account. Knowing the delay between two blades in term of σ and the vibration frequency ω of the blades it is possible to find the value of the time lag:

$$\Delta T_{\sigma_n} = \frac{\sigma_n}{\omega} \tag{3.22}$$

this time lag is the concept at the basis of the time-delayed boundary conditions [128] implemented in AeroX. Basically, equation 3.17 is modified in:

$$\mathbf{U}(\mathbf{x}, t)|_{\mathbf{x}\in b_1} = \mathbf{U}(\mathbf{T}(\mathbf{x}), t - \Delta T_{\sigma_n})|_{\mathbf{T}(\mathbf{x})\in b_2} \tag{3.23}$$

Time-delayed boundary conditions are very similar to usual periodic boundary conditions: they share the same interpolation and faces/weights addressing concepts alongside the need of a rotation tensor to correct momentum in 3D cases. The main difference is that when building the ghost cell value on face f on boundary b_1, the solution that is taken from internal cells of boundary b_2 is searched between the stored solution of previous physical times, accordingly to delay ΔT_{σ_n}.

An important aspect of the numerical implementation of delayed boundary conditions is given by the fact that the solution is stored at discrete physical times that depend from the user-prescribed physical time step Δt used to perform the unsteady simulation. However, when using these boundary conditions the solution at a physical time

between two stored physical times could be required. This is due to the fact that in general ΔT_{σ_n} is not a multiple of Δt. In this work this is tackled using a linear interpolation of the two stored solutions right after and right before the required physical time.

From the computational point of view, the strategy adopted in [128] is here implemented to reduce the memory consumption required to store the solution at different times. Since the delay related to the chosen IBPA and the physical time step Δt adopted to advance the solution are known before the simulation, a fixed-size memory block is allocated to store the boundary cells solution at multiple times. This memory block is treated as a list that is continuously overwritten on its older values in a circularly manner at each new physical time step.

It must be noted that using time-delayed boundary conditions is not the only way to perform computations with non-null IBPAs. As an example, with a 2-blades computational domain and periodic boundary conditions it is possible to directly perform computations with $\sigma = 180°$ (by enforcing vibrations with the right delay on the two blades) and with a 4-blades domain an IBPA of $\sigma = 90°$ can be investigated (with the four blades vibrating with the right delay). The advantage of time-delayed boundary conditions relies on the fact that all IBPA values can be investigated with a single-blade computational domain, reducing the mesh size but requiring the storage of the boundary solution at previous physical time steps.

3.9 Total pressure and temperature inlet boundary conditions

Usually, in aeronautical problems the computational domain for wings and aircraft investigations relies on the presence of a farfield boundary where asymptotic flow conditions are enforced. These conditions are applied on velocity, static pressure and static temperature. Then they are converted in the correspondent conservative values of density, momentum and total energy and can be applied using subsonic/supersonic inlet/outlet or characteristics-based boundary conditions through the ghost cell approach (see 2.1.8, [136]). This is also valid when simulating open rotors. In turbomachinery, however, the situation is more complicated since one of the goals is the computation of the performance map that involves quantities like total pressure ratio and total temperature ratio between the inlet and outlet boundaries. In this kind of simulations the following quantities have to be enforced on the inlet boundary:

- Total pressure $P_0|_{inlet}$;
- Total temperature $T_0|_{inlet}$;
- Flow direction $\mathbf{w}|_{inlet} = \left(\frac{\mathbf{u}}{|\mathbf{u}|}\right)\Big|_{inlet}$;

Furthermore, a way to enforce these quantities using the ghost cell approach is required. The user provides the aforementioned quantities that are in turn used to build the ghost cell solution. This solution, alongside the internal cell solution is processed by convective fluxes algorithms, see 2.1.5. Here, the algorithm adopted to obtain static pressure, static temperature and velocity from the user-prescribed quantities is briefly showed:

1. A zero-gradient static pressure condition is considered: $P|_{inlet} = P_{internal}$;

2. Using internal cell solution for pressure, β is computed: $\beta = \frac{P_{internal}}{P_0|_{inlet}}$;
3. This is in turn used to compute static temperature: $T|_{inlet} = \beta^{\frac{\gamma-1}{\gamma}} T_0|_{inlet}$;
4. A parameter, z, is computed: $z = \sqrt{2C_P\left(T_0|_{inlet} - T|_{inlet}\right)}$;
5. The velocity vector is computed as: $\mathbf{u}|_{inlet} = z\mathbf{w}|_{inlet}$;
6. From these boundary values the ghost cell values are obtained through interpolation using internal cell values;

Isentropic gasdynamics relations are used through the β parameter to link temperature with pressure and kinetic energy with temperature.

CHAPTER 4

GPGPU

The aim of this chapter is to introduce the fundamental concepts of the General Purpose GPU (GPGPU). The chapter begins describing the history of GPGPU, starting for the first approaches, that were cumbersome but effective, until the modern languages like CUDA and OpenCL. In particular, the most important aspects of the language here adopted, OpenCL, will be presented. The most important differences between GPUs and CPUs architectures will be covered. Examples of using GPGPU to accelerate computations with respect to modern multi-core CPUs will be showed. Finally, the most important limitations and efficiency problems that must be kept in mind when programming GPUs will be explained. This chapter is not aimed to be a detailed OpenCL tutorial, but can still be useful to understand if a particular algorithm can be adapted to the GPU architecture in an easy and/or efficient way. Here the focus is on GPGPU. Some basic concepts related to parallel computing are presented in appendix A.

4.1 History of GPGPU

Reconstructing the history of GPGPU is not easy. In fact, before the current standards like OpenCL [24] and NVIDIA CUDA [26] numerous and non-standard approaches have been tried in the years before the launch of NVIDIA CUDA SDK in 2007. Earlier GPGPU approaches were born in the first years after 2000 thanks to the growing GPU performances and the possibility to program the graphical processors in more flexible ways. From the very beginning, GPU architectures were specifically designed with the aim of achieving high data parallelism, which is exactly what is needed to perform graphical computations. However, with GPU development, from both hardware and software sides it appeared more and more feasible to program GPUs for other purposes rather than graphics. The first approaches in GPGPU were somehow cumbersome since

the numerical problem that exhibit high data parallelism had to be mapped to an equivalent graphical problem in order to be processed by the GPU. This basically means that functions applied to data arrays had to be mapped to transformations applied to buffers of pixels and vertexes [77]. Thus, initial approaches to GPGPU were based on the use of graphical APIs and libraries like OpenGL and DirectX. Later, in 2007 NVIDIA launched NVIDIA CUDA, initially compatible with the 8*** GPU series. This was a breakthrough in GPGPU since an API and C-like programming language were specifically developed in order to use NVIDIA GPUs to perform general purpose numerical computations. The main advantages provided by CUDA were represented by a more direct control over GPU hardware, without the need to translate numerical operations into graphical operations. The concept of "kernel", a function that is executed in parallel by thousands of GPU threads performing operations on "buffers", i.e. data arrays, led to a more intuitive way of programming GPUs for general purpose computations. Despite this kind of easiness, however, GPGPU requires anyway more effort than the usual CPU programming. The main "disadvantage" of CUDA is represented by the fact that only GPUs of its company, NVIDIA are supported. Almost in the same period, AMD/ATI launched the Close To Metal (CTM) technology, a low-level programming interface aimed to GPGPU programming for their own hardware, especially for the X1*** GPU series. However, CTM approach was more low-level than CUDA and after its initial release, AMD focused its efforts on OpenCL. In fact in 2008, the Khronos Group [24] released the OpenCL 1.0 specifications, a CPU API and C-like GPU programming language specifically designed for heterogeneous computing, mainly aimed to ease GPGPU programming. Since the very beginning OpenCL was supported by a consortium composed by the most important hardware vendors, like Intel, AMD, IBM, and NVIDIA itself. Thanks to the runtime compilation concept, the same source code can be compiled and executed on a very wide range of devices (CPUs, GPUs, FPGAs, and other kind of accelerators like Intel Xeon Phi, at least in its first versions), independently form the underlying hardware architecture. Despite the fact that today OpenCL and CUDA represent the two most important adopted GPGPU approaches, other solutions are available. As an example, OpenACC [28] provide GPGPU capabilities at very high level, exploiting the OpenMP way of parallelizing the code through the use of # pragma keyword. At the time of writing version 8.0 of CUDA SDK and provisional specifications for OpenCL version 2.1 are available.

4.2 CPU vs GPU architectures

CPUs and GPUs have very different hardware architectures and programming tools specifically designed for their main purposes. It is highlighted that despite GPGPU could accelerate computations up to one order of magnitude with respects to a CPU of the same price range or power consumption, GPUs are not aimed to substitute CPUs. Rather, GPUs are aimed to offload CPUs for some very specific kind of numerical computations. The main differences between a typical CPU parallel architecture and a typical GPU parallel architecture are given by the fact that CPUs are optimized for MPMD/MIMD parallelism and GPUs are optimized for SPMD/SIMD/SIMT parallelism (see appendix A). Graphical computations require the same operations to be performed on large data sets. Thus due to the particular optimization of the GPU architecture,

also GPGPU concepts can be efficiently exploited only when the algorithm expose data parallelism. CPUs on the other part, are more general purpose as they can be usually programmed to satisfy all SIMD/MIMD/SPDM/MPMD paradigms. However, with the same price or power consumption, GPU SIMD performances are much higher than what provided by the CPU counterpart. Figure 4.1 shows schematically the most important elements in a typical CPU and GPU architectures. We can see that a typical

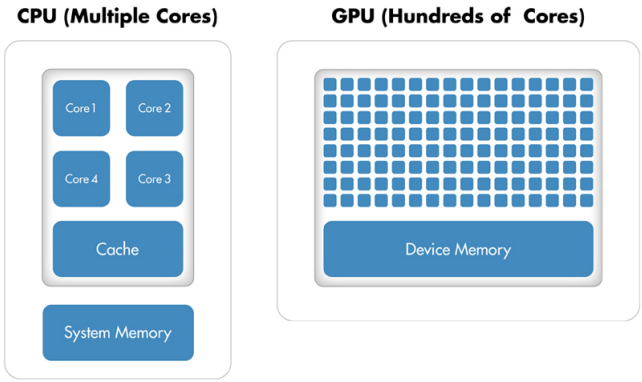


Figure 4.1: *CPU vs. GPU architecture [4];*

CPU has few cores (nowadays starting from 2 for cheap CPUs, up to more than 8 for expensive server-oriented CPUs). A typical GPU features hundreds, even thousands of cores. As an example, the AMD Fury X GPU has 4096 cores running at $1.05\,GHz$, while Intel i7 3930K CPU has 6 cores running at $3.2\,GHz$. Usually, GPU cores have lower frequencies than CPU cores. It is immediately understood that parallelizing an algorithm on GPU just to use one or few GPU cores is not convenient. In fact, if the algorithm is poorly parallelized or exhibit task parallelism instead of data parallelism, the GPU could slow down the execution even by orders of magnitude with respect to a serial CPU execution. CPU cores are optimized to execute the same or different instructions streams concurrently. The latter is exactly what required for a typical desktop computer where the user wants to execute different applications at the same time. This is also useful for servers that provide contents to multiple users at the same time. Thus, GPU cores are numerous but less general purpose than CPU cores. CPU cores have also SIMD capabilities thanks to instructions sets like SSE or AVX. However, modern GPUs usually exhibit one order of magnitude higher single precision floating point performances (GFLOPS) with respect to CPUs, especially when the same price-level is considered. As an example, Intel declares for the i7 3930K $182\,GFLOPS$ [5] while AMD declares 8600 theoretical single precision GFLOPS for Fury X [37]. It must be noted that usually these values are just theoretical values that can be reached only with opportunely tuned algorithms that exploit specific hardware functions, like FMA (Fused Multiply-Add) instruction. An important difference between CPUs and GPUs is that usually CPUs have half the computational power when performing double precision operations (due to how SSE/AVX works) with respect to single precision, while for GPUs this is not necessarily true. HPC oriented GPUs usually exhibit a fraction of the computational power in double precision (like $1/2$ or $1/4$) with respect to single precision performances. Cheaper gaming GPUs double precision performances

are more limited. Usually, double precision performances are one order of magnitude smaller than single precision performances on this kind of GPUs. Furthermore, the cost of an HPC-compliant GPU is usually one order of magnitude higher than the cost of a gaming GPU. This means that while a gaming GPU costs hundreds USD, an HPC GPU usually costs thousands USD. As an example, the AMD Fury X gaming GPU has 1/16 performances in double precision with respect to single precision. One of the goals of this work is the tuning of the solver for single precision executions in order to exploit cheap gaming GPUs. Anyway, the solver is also compatible with double precision. A test in order to search for possible results discrepancies between single and double precision executions was performed in this work. Benchmark results are showed in 6. Performing GPGPU computations in single precision also have memory occupancy advantages since float variables requires half the memory of double variables.

Another difference between CPUs and GPUs is represented by the fact that a CPU has bigger cache with respect to a typical GPU architecture. Without going into details related to the different cache levels, a modern CPU has different MB of cache, while a modern GPU usually a couple or less than $1\,MB$ of cache. GPUs in fact are optimized to provide high floating point throughput by processing the data streams read from GPU memory in a sequential manner, while CPUs cache are optimized to reduce latency. Following the previous example, Intel i7 3930K has $12\,MB$ of cache while AMD Fury X has $2\,MB$ of cache.

Alongside the architecture of the processor itself, differences between CPUs and GPUs also regard the adopted memory, in term of quantity, technology, bandwidth and latency. In fact, CPUs rely on the system RAM, while GPUs rely on their own on-board memory. For what concerns the quantity, usually system RAM is bigger than GPU memory. This is due to the fact that the CPU has to handle the operating system and hundreds of other processes alongside the application that the user want to run. On GPUs memory, instead, only data specific for few particular applications are stored. This data concerns graphical textures/vertexes/pixels and general purpose buffers for GPGPU applications. In a typical mid-range gaming desktop machine it is usually found $8\,GB$ or $16\,GB$ of system memory and $\sim 4\,GB$ of GPU memory. The situation is different for workstations where more system memory is required to handle simulations, thus $64\,GB$ of RAM are frequent. HPC GPUs instead feature more memory than gaming GPUs, thus while typical gaming GPUs have $4 - 8\,GB$ or memory, typical HPC GPUs have $12 - 24\,GB$ of memory. AeroX tackles the limited amount of memory available on GPUs, especially on gaming GPUs by supporting single precision floating point representation that halves the memory requirements of double precision computations. Furthermore the solver employs an explicit time stepping scheme that reduces memory requirements from a numerical point of view (avoiding matrix storage). For what concerns memory technologies, CPUs nowadays use DDR3 or DDR4 technology while GPUs use GDDR5 technology and recently the HBM (High Memory Bandwidth) technology. The purpose of CPUs memory is mainly to reduce latency while the purpose of GPU memory is mainly to improve bandwidth. In fact high bandwidth is required in graphical and SIMD GPU computations when millions of threads executing on thousands of cores have to be fed with data. On CPUs, low latency memory guarantees system responsiveness in a multi-task scenario. GPU memory latency could be a bottleneck for GPU executions, and as explained in 4.3, in order improve

computational efficiency, numerous threads and core private memory operations are executed between two memory accesses. This way it is possible to use the time spent waiting data coming from memory in order to perform computations. Thus, the solver developed in this work has been specifically tuned to reduce GPU memory bottlenecks as much as possible. Usually GPUs memory have one order of magnitude higher bandwidth than CPUs memory. Has an example the HBM memory of the AMD Fury X GPU theoretically provides $520\,GB/s$, while a typical DDR4 system provide a bandwidth in the order of $50\,GB/s$ (the value depends from frequency and number of channels). A peculiar aspect of CPU/GPU HPC-oriented memory is represented by the ECC (Error Correcting Code) feature. Server-oriented memory has this kind of feature in order to provide higher reliability for some peculiar applications. HPC GPUs usually features ECC memory too. The goal of this work is to use gaming GPUs where ECC memory is not employed. Obviously the solver is also compatible with GPU ECC memory since from the programmer point of view this feature is not directly manageable. In order to asses if the lack of ECC memory could lead to differences in results, in this work a test with an ECC-compliant HPC GPU was also performed (see 6.2.4).

Despite the CPU, the GPU is not running an operating system kernel. Thus, a CPU is always required in order to organize the work to be sent to the GPU, since the GPU cannot work autonomously. The GPU is used as an accelerator that helps the CPU to perform specific data parallel computations. The CPU is used to pre-process the data and compile the code that will be executed on the GPU. This means that before data and code are stored in GPU memory, they are stored in system RAM and accessible by the CPU. When the CPU offloads computations on the GPU it firstly needs to send code and data to the GPU memory, which is specifically designed with high memory bandwidth to keep the GPU cores fed with work. This data transfer is achieved through the bus that connects the GPU to the rest of the system, which is the PCI-Express bus. PCI-Express is a general purpose and flexible standard bus that in its current fastest version (version 3.0, 16x) allows a bandwidth up to about $16\,GB/s$. By considering this number, the role of GPU memory is immediately understood. In fact, PCI-Express bandwidth is one order of magnitude lower than a typical GPU memory bandwidth, which is in the order of $100\,GB/s$. Thus, a continuous data-transfer between the CPU and the GPU would lead to a bottleneck. GPU memory is aimed exactly to tackle this problem. Even if system RAM is faster than PCI-Express, data transfer between GPU and CPU is still limited to the slowest bus in the chain, i.e. PCI-Express. Thus, the idea in GPGPU is to firstly transfer input data and code from system RAM to GPU memory and then allow the GPU processor to perform computations by exchanging data with its on-board memory. CPU-GPU data transfer is anyway required since sooner or later the results have to be recovered. In the particular case of this work intermediate results have to be checked for convergence (see 5.2.2) and saved on disk. Thus, an opportunely defined trade-off between computations and results checking is fundamental.

Important differences between CPUs and GPUs regard costs and power consumption. Since GPU cores are specifically designed to provide high floating point computational performances for algorithms that are mapped to the SIMD/SPMD model, it is easy to understand that for specific kinds of numerical problems modern GPUs provide higher GFLOPS/Watt ratios with respect to modern CPUs, up to about one order of magnitude. Figure 4.3(a) shows schematically the power efficiency improvements

provided by both kind of hardware during last years. Though this may not be that relevant for a single desktop computer, it is a critical factor when hundreds or thousands of CPUs and GPUs are assembled to build a cluster or a supercomputer. In fact supercomputer power consumption can easily reach the MW order. Power consumption is somehow related to the so called TDP, Thermal Designed Power [6], which is the maximum amount of heat generated by the CPU/GPU that the cooling system is required to dissipate in typical operations. It is underlined that the TDP is related to the cooling system and not directly to power requirements of the processor. However, it can still be useful to do some considerations. Considering the two aforementioned CPU and GPU, the Intel i7 3930K has $130\,W$ TDP and provide $182\,GFLOPS$ while the AMD Fury X has $275\,W$ TDP and provide $8600\,GFLOPS$. It is evident that this high-end gaming GPU has double the TDP of the CPU. However, since the computational power is one order of magnitude higher than the one provided by the CPU, it is evident that the GFLOPS/Watt ratio of the GPU is one order of magnitude higher with respect to the CPU. Roughly speaking, if it is possible to execute a particular algorithm on the GPU in a computationally efficient way, multiple CPUs would be required to provide the same computational power, leading to higher power consumption and more dissipated heat with respect to the use of a single GPU. Furthermore, it must be noted that motherboards that support multiple CPUs are very expensive (in the order of $1000\,USD$) and connecting multiple CPU nodes requires hardware to be duplicated (RAM, CPU, motherboard, hard disk, network cards, cases...) with all the related costs, plus an interconnecting bus. Instead, nowadays, multiple GPUs can be easily installed on a single motherboard, even on cheap desktop-oriented motherboards in a price levels of $150\,USD$, thanks to AMD CrossFire and NVIDIA SLI or simply thanks to the availability of multiple PCI-Express 16x slots. Also, roughly speaking, having 4 GPUs on the same workstation requires less space than any solution that provides 4 multicore CPUs. Another aspect related to GFLOPS/Watt ratio is that, especially in cluster and supercomputer, opportunely sized air conditioning systems have to be employed in order to guarantee that the hardware will be used within a correct range of temperature. Air conditioning means again power consumption and space. Thus, if one GPU installed in a single workstation can be effectively exploited to provide the same computational power provided by, even a small, computer cluster, the advantages in term of space, heat and power consumption are evident. However, it must be underlined again that only peculiar numerical fields can take advantage of GPGPU. Thus is not the case, CPU-based architectures are anyway required.

4.2.1 When using GPGPU

GPGPU can be exploited basically where heavy data-parallel computations are required. It is quite simple to get an idea of the many possible applications of GPGPU by just visiting the BOINC project website. BOINC is an open source software for volunteer grid computing. Among the different projects, the GPU accelerated algorithms include:

- Astronomy, astrophysics and astrobiology;
- Mathematics;
- Cryptography;

- Proteins-related simulations;

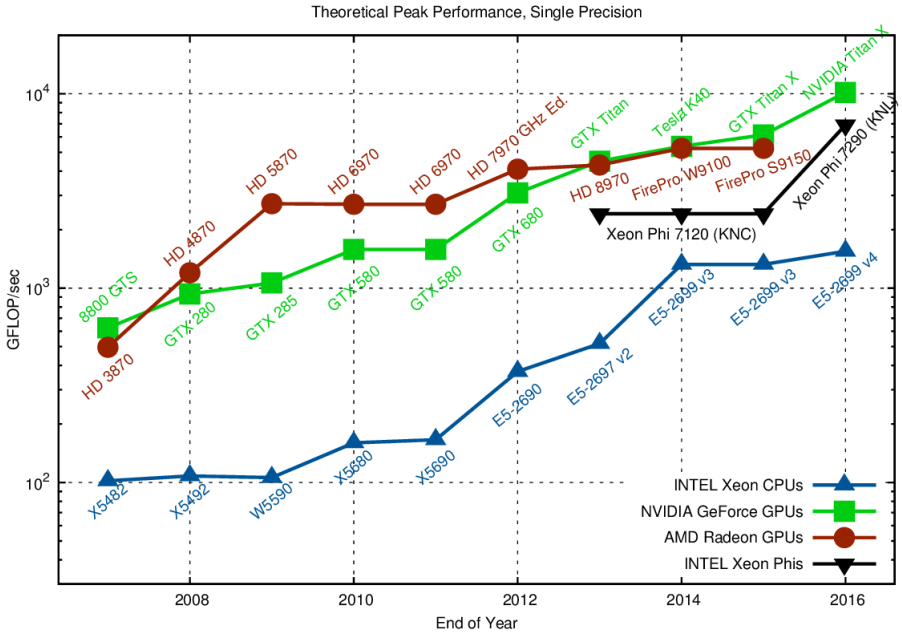
It is also worth to cite a different grid computing project, Folding@home [19], focused on protein folding. In particular, Folding@home released its first GPGPU client back in 2006. Despite the aforementioned projects, other GPGPU applications may include structural and fluid simulations, like in this work.

Due to the nature of graphical computations, the GPU architectures perform very well with algorithms that exhibit embarrassingly parallel [36] workloads, i.e. when little or no effort is needed to separate the problem intro a number of parallel tasks. An example is represented by brute-force algorithms for password cracking. Roughly speaking, the password is somehow gathered in some transformed form. This could mean hashed form or anyway in a form obtained after the application of a specific algorithm. Anyway the plain-text password is not available. The transformation algorithms are opportunely designed in such a way that should not be possible to reverse the operation and recover the plain-text form of the password. The idea behind password brute-forcing is to generate a password list, apply the known transformation algorithm to each password candidate and finally check if the transformed candidate is equal to the transformed password to be cracked. The transformation algorithm can be applied to each password candidate concurrently, without any form of collaboration between different candidates. It is thus easy to understand that GPUs can outperform CPUs in this kind of applications, thanks to the their intrinsically data-parallel architecture. As an example, using the John the Ripper [23] software, the AMD 380X GPU resulted $42\times$ faster than the AMD A10-7700K CPU in cracking WPA-PSK of a previously captured Wi-Fi 4-way handshake [17]. Both these CPU and the GPU were adopted in this work and are actually installed in the same desktop computer. The obtained speed-up could seem very high, especially if we consider that the GPU costs just $1.5\times$ the CPU. However, it is noted that in other kinds of applications, like CFD, the situation is different since somehow collaborative effort between different computational units is required. As an example, if we distribute face fluxes computations among GPU cores by assigning one GPU core to one mesh face, how can we deal with the fact that two different cores may have to update the same cell residuals concurrently? The mapping of the CFD/FSI algorithm to the massively parallel GPU architecture will be discussed later in 5.

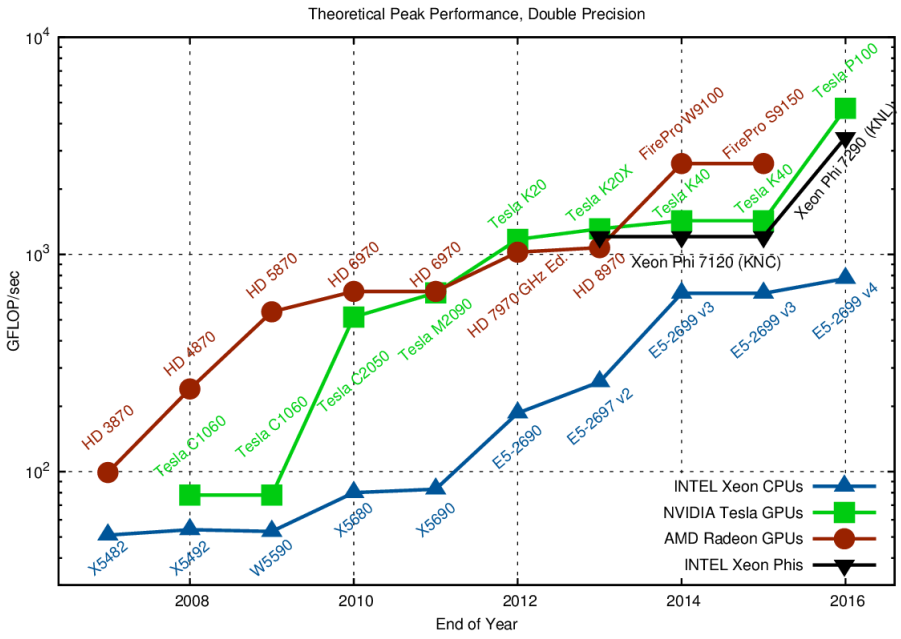
4.2.2 Gaming GPUs

One of the most important ideas of this work is to accelerate computations by exploiting cheap gaming GPUs. First of all, the choice of gaming GPUs is done thanks to their relatively low cost with respect to specifically-designed HPC GPU that are available today and that exhibit about the same single-precision performances. A mid-range price gaming GPU today is available for less than $300\,USD$ and features over $5\,TFLOPS$ in single precision [37, 38]. Usually the same single precision computational power in HPC GPUs is offered at more than $2500\,USD$. Furthermore, today basically every new and few years old pc features a GPU capable of GPGPU through OpenCL, even if it is a very cheap GPU. This is possible thanks to the fact that despite NVIDIA CUDA, OpenCL is compatible with both NVIDIA and AMD GPUs. Obviously HPC GPUs are more expensive but offer more. In particular, as previously said, HPC GPUs offer higher double precision computational performances, bigger on-board memory

and ECC memory technology. Figure 4.2(a) shows the single precision computational power of last years CPUs, gaming GPUs and HPC GPUs, while figure 4.2(b) shows the same regarding double precision performances. The comparison is repeated in



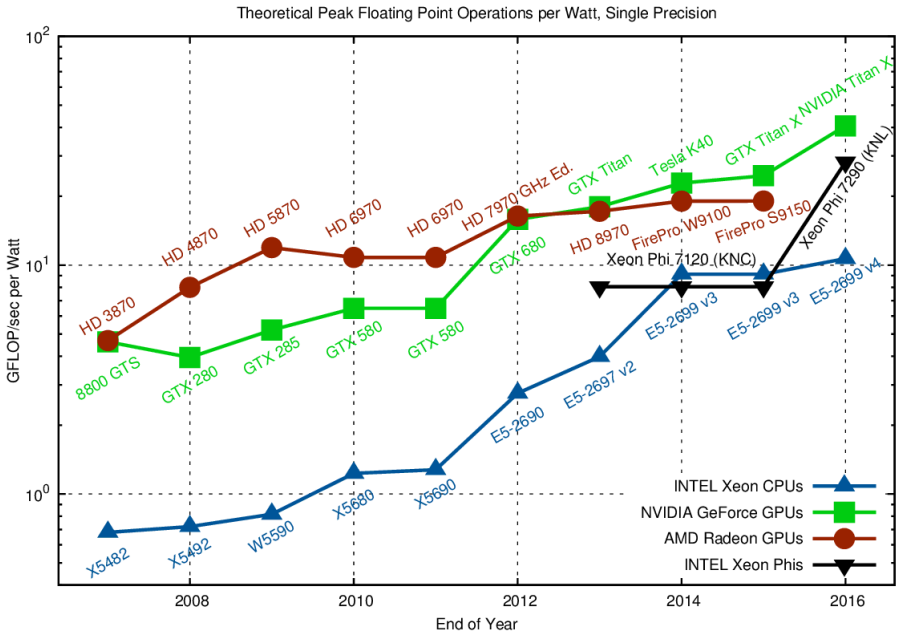
(a) Single Precision



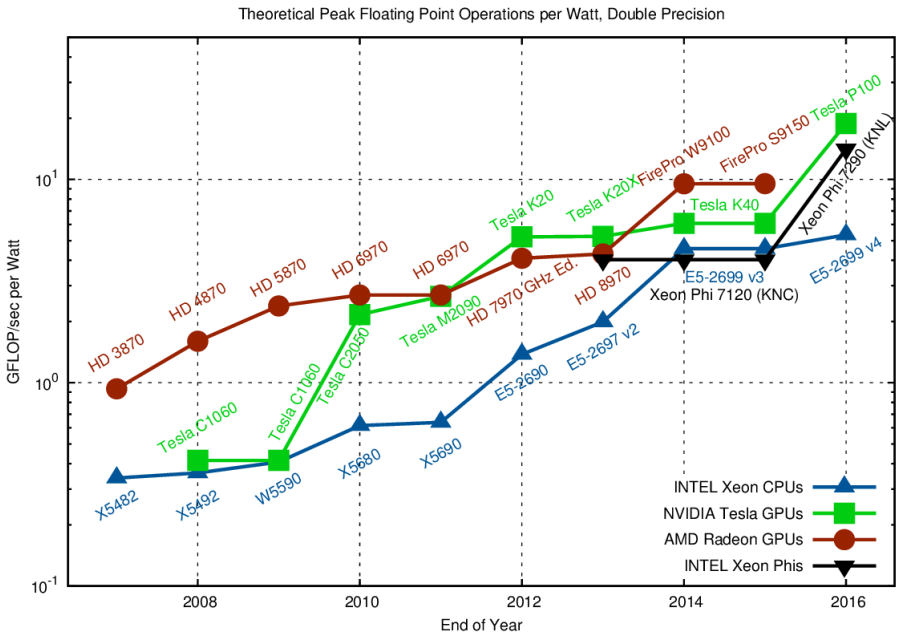
(b) Double Precision

Figure 4.2: *Modern CPUs and GPUs theoretical floating point computational power [2].*

figures 4.3(a) and 4.3(b) for what concerns the performance per Watt for single and double precision respectively. In AeroX different strategies have been implemented



(a) Single Precision



(b) Double Precision

Figure 4.3: *Modern CPUs and GPUs computational power to power requirements ratio [2].*

in order to bypass the requirement of an HPC-class GPU. The adopted strategies to exploit gaming-class GPUs are discussed in 5.2.3. Benchmarks comparing the differences between the use of single and double precision were performed and the results are showed in 6. During AeroX benchmarks it appeared that just negligible differences are obtained between the single and double precision in terms of results accuracy. For

what concerns memory limitations of gaming GPUs with respect to HPC GPUs, this problem is tackled by implementing an explicit time-stepping scheme. This way there is no system matrix to be stored, allowing cases with millions of cells to be simulated even using a mid-range gaming GPU, like one of the GPUs adopted in this work (see 6.1). Explicit time-stepping schemes are also well suited for the SIMD/SPMD/SIMT architecture of GPUs thanks to the fact that the total work can be easily split among the hundreds/thousands of cores available on modern GPUs. In fact, since no linear solver algorithms are employed, cells can be updated independently, by simply knowing the solution from the previous pseudo time step. Furthermore, GPU memory limitations are also tackled naturally with the use of single precision with respect to double precision, thanks to the fact that memory requirements are basically halved (excepts for arrays of integers). The choice of an explicit scheme is also supported by the perfect mapping of numerical schemes into the SPMD/SIMD hardware architecture of a typical GPU.

Figure 4.4, already presented in the introduction of this work, shows the advantages provided by the exploitation of GPGPU in terms of performances (GFLOPS) to costs (USD) ratio. The important aspect is not the numbers themselves since they are just theoretical values and depend from the chosen CPU/GPU couple. What is important is the fact that, when the numerical problem exhibits the possibility of GPGPU exploitation, the advantages in term of performances/costs given by GPUs are evident.

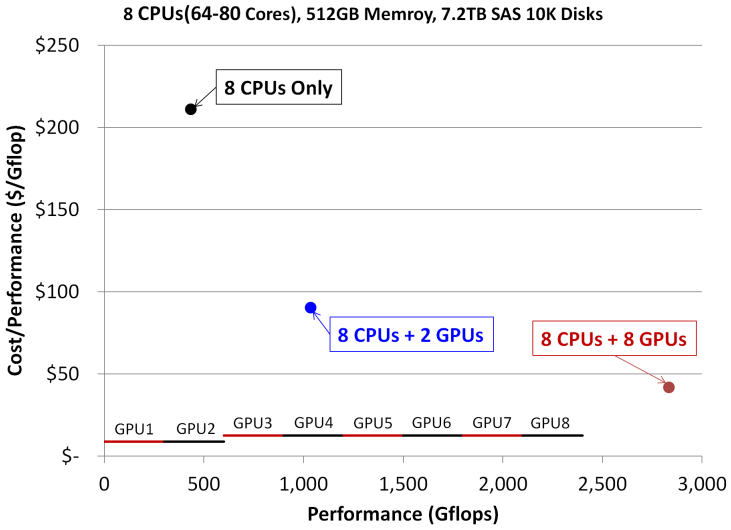


Figure 4.4: *Performances to costs ratio, GPGPU advantages [3].*

ECC memory is a typical feature for HPC GPUs due to the requested higher reliability. However, in this work no problems were faced with the use of gaming GPUs without ECC memory. Different cases were simulated with both gaming GPUs and an ECC-compliant HPC-level GPU (a relatively old, at the time of writing, NVIDIA Tesla C1060) but no differences attributable to memory problems were encountered, see 6.2.4. Obviously, thanks to OpenCL and the implemented typedef-based single/-double precision switch, the solver is natively compatible also with HPC GPUs, with double precision and with ECC memory without requiring any sort of source code modifications or tuning.

4.3 Advantages and drawbacks of GPGPU

In this section the most important aspects of GPU programming are introduced. These aspects are strictly related to the underlying hardware architecture of the GPU processor and memory but must be taken into account by the programmer in order to achieve an efficient GPU execution. In fact, these aspects have to be considered since the very beginning, when designing the algorithm, before its actual implementation. Otherwise, bottlenecks could be so relevant that the GPU execution could be slower than a CPU serial execution, even by orders of magnitude. Porting an algorithm from CPU to GPU in an efficient manner could require to re-think the entire algorithm in order to be adapted to a SIMD/SPMD/SIMT model.

4.3.1 Problem size

The GPU architecture is optimized to execute thousands, millions of threads concurrently on a processor composed by hundreds or few thousands of cores. Due to the underlying hardware architecture, high computational efficiency is reached only when enough threads are scheduled for execution on the GPU processor. When a single core of a GPU is compared to a single core of a CPU, the CPU core is more powerful in serial executions. CPUs are in general faster than GPUs also when executing few threads concurrently. This is due to the fact that the computational power of the remaining hundreds/thousands of GPU cores would be wasted, leading to an effective computational power orders of magnitude smaller than the theoretical peak performances. In fact, in order to achieve speed-ups that truly show GPGPU advantages, at least few thousands of threads have to be scheduled, meaning that the numerical case under investigation has to be big enough to fully load all the GPU cores. However, problem size is not the only aspect that has to be considered when designing GPGPU code. Other sources of problems, e.g. branch divergence and non-coalesced memory accesses could led to slow executions if not opportunely addressed. These phenomena lead to bottlenecks that basically waste computational time by letting GPU cores waiting for data from memory or waiting for other GPU cores that are following different program paths, even if thousands of threads are scheduled for execution. Problem size is not a particular problem in CFD applications, like in this work, since usually meshes of hundreds of thousands or millions of cells are employed. These numbers are more than enough to keep the GPU processor fully loaded. An investigation of the performances dependence of AeroX from the mesh size is performed in 6.2.3.

4.3.2 Branch divergence

This is an aspect intrinsically related to GPU processor architecture and its SPMD/SIMD/SIMT model. When a kernel does not contain any conditional statement, each thread executes the same instructions on different data, following SIMD approach. In short terms, branch divergence problems happen when different GPU threads instead follow different paths of the same kernel, thus follow two different instructions streams. This situation by definition is allowed in SPMD since the two code paths belong to the same program. Besides CPU SSE/AVX instruction sets that follow a tightly SIMD approach, SPMD/SIMT and GPUs in general allow multiple program path to be followed concurrently. In a typical CPU architecture, different cores can follow different instructions

streams without any particular computational efficiency issue. However this is not true for typical GPU architectures. Even though in GPGPU multiple code paths are allowed to be executed "concurrently", the underlying GPU hardware is not advanced enough to allow different cores to perform different operations "simultaneously". Roughly speaking, as showed in figure 4.5, when branch divergence happens, some cores are performing the 1st instructions set while the other, that need to perform instructions sets 2 and 3, are waiting. This means that branch divergence reduces GPU cores load-

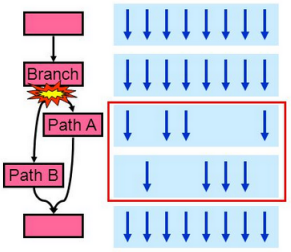


Figure 4.5: *Branch divergence [from internet]. Blue arrows represent threads, pink boxes represent groups of instructions. From the branch two code sub-paths are followed by different threads.*

ing, leading to computational efficiency reductions. When all the sub-paths related to branch divergence are completed, all the threads converge back to the same unique execution path. Thus it is fundamental to avoid or at least reduce branch divergence in GPGPU programming. Sometimes this is not entirely possible but if opportunely limited does not particularly reduce the computational efficiency of the entire algorithm. This is just a quick explanation of branch divergence related issues. The situation is more complicated. In fact branch divergence problems are also related on the thread scheduling strategies adopted by the particular GPU architecture. As an example, in NVDIA Fermi architecture when a SM (Stream Multiprocessor, basically a group of GPU cores (a Compute Unit in OpenCL notation), executes a thread block (called work-group in OpenCL notation), basically a group of threads, the block is subdivided in sub-groups of 32 threads. Each group is called "warp" and executes concurrently with other warps. A warp executes one instruction at time, so maximum efficiency is achieved when all the 32 threads in the same warp perform the same operation, following the same execution path. However if branch divergence occur among threads of the same warp, the execution is slowed down. When instead different warps follow different paths but all threads inside the same warp follow the same path, peak computational efficiency is still reached. AMD architectures have a similar concept of warp which is called "wavefront" but with 64 threads width. The same discussed concepts can still be applied to wavefronts. It is understood that branch divergence has not to be seen as a limitation of the GPU architecture. As for the sequential memory access problems (see 4.3.3), branch divergence on GPU is just a consequence of the underlying GPU processor hardware design that is tuned to reach peak efficiency when all cores are performing the same operation. In fact, this is what required for graphical computations where the same operations have to be performed on numerous graphical entities like pixels. In AeroX one source of branch divergence is the capability of the solver to handle hybrid meshes, where different cells have different numbers of neighbors. This is translated in different number of operations when each thread is assigned to each cell and a loop over

the cell faces is performed. Obviously, when a particular GPGPU algorithm is known to exhibit branch divergence it is possible to limit the performance loss by accurately tune the algorithm on the basis of the underlying warp/wavefront size, but this would bind peak performances to a particular underlying hardware architecture.

4.3.3 Memory coalescing

This is an aspect related to GPU memory. As for branch divergence, the problems with memory coalescing have not to be seen as some kind of GPU memory architecture limitations. In fact GPU memory is specifically designed to maximize performances for graphical computations, where sequential memory accesses are required. Roughly speaking the idea is that sequential graphical entities are expected to read sequential data from memory since the same operation has to be performed on a grid of entities. As previously mentioned, GPU processors have smaller cache memory than CPUs. Following the SIMD paradigm the idea is to apply the same stream of instructions on a stream of data using different cores that are working in parallel, each one on its own data. Since GPUs are aimed to graphical computations, their entire design is optimized for this kind of numerical computations. Maximum GPU computing efficiency is reached when the cores are processing private data, using private memory and registers installed on the GPU processor, reducing the GPU global memory read and write accesses to the initial input of data and the final results writing. The concepts of private memory and global memory will be explained later in 4.4.2. Since accessing the main GPU memory (global memory) requires different clock cycles, when GPU cores are accessing this memory they are wasting computational time. Private core memory is instead fast enough to fed cores with data without slowing down execution. Anyway this problem is partially masked by the fact that usually thousands/millions of threads are scheduled on hundreds/thousand of available GPU cores. However, it is very important for GPGPU to design the code such that data is firstly fetched by GPU cores from GPU global memory. Then as much floating point computations as possible are performed in a fully SIMD approach (trying also to minimize branch divergence previously described in 4.3.2) exploiting private memory. Finally results are written back to GPU global memory by each core. In order maximize bandwidth when accessing data on GPU global memory, memory accesses are coalesced, meaning that multiple data requests can be combined together in a single memory transaction [7]. Briefly, for NVIDIA GPUs what happens is that when all threads inside a warp read data in sequential order (i.e. thread 0 is reading at location 0, thread 1 is reading at location 1, and so on...) all these accesses can be combined together in a single memory read request, speeding up memory access and thus execution. For AMD GPUs the same applies for wavefronts (which is the equivalent of NVIDIA's warps). This is just a brief introduction to memory coalescing as different architectures have different hardware features that could improve memory access in the case of non perfectly sequential requests or misalignment [7]. However, if memory coalescing is poorly exploited in the code, GPU cores could waste computational time, waiting for multiple memory transactions to be performed. In this case the execution could be slowed down until a point in which GPGPU is not providing advantages over CPU executions. It must be noted, however, that sometimes it is not possible to fully exploit memory coalescing, especially when some kind of memory mapping/addressing is required, e.g. with unstructured

meshes in this work. The, although small, cache installed in GPU processors could also help. Furthermore, when some kind of collaboration between cores is required, as explained in 4.4, the so called "local memory" (OpenCL notation) or "shared memory" (NVIDIA CUDA notation), installed on GPU processor, can be exploited to speed-up data exchange between threads of the same thread block (NVIDIA CUDA notation) or work-group (OpenCL notation). However, the general idea is to optimize the code as much as possible to minimize the total number of memory transactions and maximize core private computations, minimizing also branch divergence. As said, in this work one source of problems with memory access is related to the capability of the solver to handle unstructured mesh. Since addressing arrays have to be used, memory cannot always be accessed sequentially.

4.3.4 Debugging and profiling

Here, GPGPU applications debugging and profiling is briefly discussed. This topic will be covered also later in 5.2.4 and 5.2.5 when discussing the AeroX implementation details. With CPUs debugging and profiling is quite simple. Over the years numerous different tools were developed, both command line based and GUI based. Nowadays modern IDEs (Integrated Development Environment) usually provide profiling and debugging capabilities. However, especially in Linux environments, different well know command line tools, like grprof [22] for profiling, gdb [21] and valgrind [34] for debugging are available. valgrind in particular is very useful in case of buffer overflows/segmentation faults since it can easily detect, a the cost of a slowed down execution, the line in the code were memory is not correctly accessed. On CPU, the operating system kernel handles the processes. If errors due to numerical problems (e.g. division by 0) or memory issues (e.g. trying to execute instructions from a data segment in virtual memory) happen, the user is warned about the problem. On GPU instead there is no operating system kernel (not to be confused with the OpenCL kernel concept) running on the graphical processor. Thus, when numerical problems happen or the memory is not correctly accessed, the user is not directly warned. Obviously if divisions by 0 are performed the results will likely contain NaN (Not a Number) values. However it is not easy to find the exact line of code where an array is accessed in write/read over its bounds. Usually, when memory is not correctly handled, two executions of the solver with the same configuration (mesh, solver parameters, boundary and initial conditions) provide two different results due to the fact that memory could be read over the array size from locations that contain "random" numbers. In this case debugging could be difficult. AMD, NVIDIA and Intel provide SDKs to program, debug and profile OpenCL applications alongside with runtime libraries. However, in order to track and correct memory issues a new tool has been developed, Oclgrind [27]. As the name suggests, it can be used in a similar way to valgrind. The computations are slowed down but the tool is able to tell the user the lines of GPU code where memory is not properly accessed. It can also warn for possible data-races (e.g. when two different threads are attempting to concurrently write to the same memory location). For what concerns profiling, OpenCL API directly offers functions that can be used to monitor the computational time spent by each kernel.

4.4 OpenCL

Here the most important concepts related to OpenCL API and OpenCL C programming language are introduced. This is not aimed to be a detailed presentation since details can be found in numerous books [40, 71, 90, 111, 113, 114, 134, 146] or directly inside OpenCL specifications [24]. As mentioned in the previous sections, the solver parallelization is realized through OpenCL, thanks to its ability to allow the programmer to write one source code that is than natively compatible with a wide range of devices, such as CPUs, GPUs, FPGAs, and other kind of accelerators like Intel Xeon Phi (at least the first version for now). In particular, the solver has been tested on AMD CPUs, AMD GPUs, AMD APUs, Intel CPUs and NVIDIA GPUs without encountering any problem. When an OpenCL application is executed on a GPU, the work is split in thousands/millions of threads executed concurrently by the GPU cores. When instead the application is executed on a typical CPU architecture, the work is split between few threads in order to maximize performances for the underlying shared-memory multicore CPU architecture A.4.2. Depending on the CPU implementation of the OpenCL specifications, when executed on CPU, the code could be also automatically vectorized in order to exploit SSE/AVX instruction sets A.4.1.

It must be noted that from the first release of the OpenCL specifications in 2008, with version 1.0, numerous specifications followed, i.e. versions 1.1, 1.2, 2.0, 2.1, 2.2. The main problem is represented by the fact that usually OpenCL specifications, like other specifications as C/C++, are not completely supported by their implementation when they are introduced. As an example, at the time of writing, while AMD supports OpenCL 2.0, NVIDIA supports up to OpenCL 1.2. Since NVIDIA GPUs represents a wide fraction of the GPUs installed on modern machines and since one of the aim of the use of OpenCL in this work is to provide maximum portability, the solver has been written using OpenCL version 1.2. It must be noted, however, that OpenCL 2.X-compliant devices feature backward compatibility with OpenCL 1.X. Thus, basically every OpenCL 1.X and 2.X compatible device is able to run the solver. Furthermore, the main differences between OpenCL 2.X and OpenCL 1.X are mainly focused on features like C11 atomics, CPU-GPU shared memory, and others [24], that are not directly exploited in this work.

The wide hardware compatibility provided by OpenCL is guaranteed by its clever abstractions and the runtime kernel compilation concept. When programming with OpenCL, two sets of source code have to be written. One set, written in C/C++, is for the "host". In the host source code OpenCL API functions and data types are used. The host is basically the CPU that organizes the work to be sent to the "device". The device is instead the physical processor that effectively execute numerical operations. The device could be the CPU itself, the GPU or other kind of OpenCL-compliant devices (like FPGAs or other compatible accelerators). Anyway the second set of sources is written for the device using OpenCL C language, a subset of C99 (in the case of OpenCL 1.X) with some restrictions. The device sources contain the so-called "kernels", basically functions that are compiled and executed on the device. One important thing has to be underlined. The host C/C++ code includes the OpenCL header, is pre-compiled using usual compilers such as gcc [20] and is linked with the OpenCL library. This means that an executable is created and can be launched like any other application. The

situation is different for the device code, as it is usually compiled at run-time (although it is still possible to pre-compile it for a particular device architecture). This is basically what happens with OpenGL shaders [24]. The host code has to be opportunely written in order to read the device code from disk, compile it with the device-specific runtime OpenCL compiler and then run it on the selected device. The runtime device code compilers are usually shipped with the vendors SDKs and drivers. This way if on a single system an Intel CPU is installed alongside an AMD GPU and an NVIDIA GPU, three different OpenCL runtime environments have to be installed in order to allow the execution of the device code on all the three available devices. The idea here is that the programmer or the user does not need to worry about the underlying hardware architecture but has just to choose at runtime which device use to run kernel code. This way, the OpenCL host and device code written for a CPU is exactly the same as the code for a GPU, and the code written for an AMD GPU is exactly the same as the code for an NVIDIA GPU. Since for what concerns the host code free compilers like gcc can be used alongside the OpenCL header file and library freely provided by device vendors and since the runtime device code compilation is performed by the implementations, basically programming and executing OpenCL applications can be done for free. One of the most important goals of this work is to obtain a solver that, thanks to the OpenCL concepts, is compatible with the widest range of CPUs and GPUs from the most important vendors.

Now the most important concepts and abstractions of OpenCL are briefly presented.

Opencl host

The host is basically the CPU that is used to manage the work that has to be performed by the device. On the host the code written in C/C++ that uses the OpenCL API (functions and types) is executed. The host code includes the OpenCL header file, is compiled using the usual compilers such as gcc and is linked with the OpenCL library. The host is basically used to pre-process and enqueue work on the device that is instead the hardware on which numerical computations are effectively performed.

OpenCL platform

Roughly speaking a platform is a vendor-specific OpenCL implementation. Thus, in a system with an NVIDIA GPU, an AMD GPU and an AMD CPU two platforms are available, one for the NVIDIA device and one for both AMD devices. This is one of the most important concepts on which the OpenCL abstraction is based. Thanks to it the programmer does not need to think about the underlying hardware but has just to choose at runtime the desired platform to be used to execute kernel code.

OpenCL device

Considering the example of a system composed by one NVIDIA GPU, one AMD CPU and one AMD GPU, a total of three devices are installed. At runtime when the program queries the system for the available devices, one device for the NVIDIA platform will be available and two for the AMD platform. In particular, for the latter, one device of the type CL_DEVICE_TYPE_CPU and one of type CL_DEVICE_TYPE_GPU will be obtained, with the obvious meaning of the strings. The device is basically the hardware

that executes the kernels and is where effectively the bulk of numerical computations is performed. As for the platform, the device on which kernel code is executed is chosen by the user at runtime. A device could be a CPU, a GPU, or other kind of accelerators (like Intel Xeon Phi, at least in the first versions, or FPGAs).

OpenCL kernel

Roughly speaking, kernels are C-like functions written in OpenCL C, a language similar to standard C but with some restrictions (e.g. function pointers are not allowed). Kernels are executed by the device, thus by a CPU, a GPU or an accelerator. These are the functions that actually performs numerical computations and are compiled at runtime (although off-line compilation is usually available for specific architectures) for the devices chosen for execution. Thanks to OpenCL abstractions it is possible to realize applications that exhibit both data parallelism and task parallelism. The latter is done with concepts like multiple queues, out-of-order queues and the clEnqueueTask function. However, for the purposes of this work, the focus is on using OpenCL to achieve data parallelism in order to exploit the SPMD/SIMD/SIMT architectures of GPUs. This way, the work is scheduled for execution on the device using the clEnqueueNDRange function as will be showed in 5.

OpenCL context

A context defines the environment in which kernels are defined and executed. A context consists of:

- A collections of devices on which numerical computations are performed;
- A list of kernels that will be executed on the chosen devices. It is possible to run different kernels on different devices at the same time;
- Program objects: they are obtained from the device source code and contain one or multiple kernels. Program objects are compiled using the OpenCL compiler provided by the vendor implementation;
- Memory objects: these are objects visible in the device memory and accessible within kernels. Memory objects, like buffers, are defined on the host and explicitly between the host and the devices. Since different devices and hosts may have different architectures, memory objects provide compatibility and portability to the code when exchanging data between different hardware architectures. Buffers and images are memory objects;

It must be noted that different devices from the same platform can be used in the same context while this is not possible with devices from different platforms. In the most general case, it is possible to allow data exchange between devices from different platforms but in this case data must pass through the host. In this case the host code has to be opportunely programmed to do so.

OpenCL queue

As previously said, host code is used to organize and send work to the device. This is performed using the queue abstraction. Basically, has the name suggests, the host

enqueues commands to be executed on the device. As an example, a command could be a kernel scheduling for execution on the device or a data exchange between the host and the device through buffers. Queues can be "in-order" or "out-of-order" depending if commands reordering is allowed (for the latter case). However, AeroX, for now, only uses in-order queues. At least one queue is necessary to send commands from the host to the device. Multiple queues are allowed for a device. Multiple queues could be used to potentially allow task parallelism on a single device. From the underlying hardware point of view, when a command is enqueued, data or kernels are exchanged between the CPU and the GPU. It must be noted that in order to improve performances the functions called from the host to enqueue commands on the device are non-blocking, meaning that they return almost immediately. The actual execution of the command could be instead delayed. This is the default behavior. However, when the host wants to read back results from the device, specific options must be used in order to wait the true end of data transfers before post-processing results on the host. Otherwise, meaningless data could be read by the host from buffers.

OpenCL buffer

As previously mentioned when discussing memory objects, the concept of buffer is adopted to achieve portability and compatibility when exchanging data between the host and the device or between multiple devices. A buffer is basically a memory array. A buffer can be of various types, i.e. float, double, int. Buffers are contiguous block of memory available to kernels and can be used to read and write global memory data. The idea is to fill input buffers on the host during pre-processing, transfer them to the device, read them within kernels executing on devices, write results to output buffers, transfer back output buffers to the host to be post-processed. Buffers are shared between devices of the same context. Data exchange between devices of different contexts, instead, must be explicitly handled by the host. Buffer and images are memory objects, although in this work only buffers are used. Particular attention must be given to read and write data from the host. In fact host code has to be explicitly programmed to safely access buffers when the device is not accessing them simultaneously. This is due to the aforementioned default non-blocking nature of commands enqueuing.

4.4.1 OpenCL work subdivision

Here, OpenCL concepts related to how the work is subdivided during execution, in order to achieve data parallelism, are discussed. When a kernel is executed on a device, multiple kernel instances are executed concurrently by numerous so called work-items (called threads in CUDA notation). In this work the word "thread" and "work-item" is used with the same meaning when referring to GPU executions. It must be noted that when considering the CPU as the device the equivalence is not completely true. Basically every work-item executes the same kernel code, in a true SPMD approach. As mentioned, different threads could potentially execute different paths due to branches, leading to the aforementioned problems with branch divergence. When considering GPU executions, thousands or millions of work-items are executed concurrently by the hundreds or thousands available GPU cores. Obviously in a CFD simulation there will be more work-items than cores, thus the GPU has to schedule work-items for their

execution. When the device on which the kernel is executed is instead a CPU, although the same millions of work-items have to be executed, just few operating system threads are created to hanlde work-items on the few available CPU physical or virtual (if e.g. using Intel HyperThreading) cores. This means that when the device is a CPU, a single operating system thread will be responsible for multiple work-items. Furthermore, as explained in [8] the Intel implementation has also the ability, through implicit vectorization, to also distribute work on SSE/AVX units automatically wherever is possible. It must be noted that work-items execute the same kernel but different work-items could follow different paths inside the same kernel, leading to branch divergence problems, especially for GPU architectures (see 4.3.2). Work-items can be grouped to form work-groups (called thread blocks in CUDA notation). This is especially useful when work-items collaboration is required. Considering the OpenCL abstraction and a typical GPU architecture few aspects is worth to underline. Work-items are scheduled for execution on "processing elements", following the OpenCL abstraction, that are basically mapped to GPU cores. Work-groups are scheduled for execution on "compute units", following the OpenCL abstraction, that are basically mapped to SM (Streaming Multiprocessor) in NVIDIA Fermi architecture (or similar, like SMX for NVIDIA Kepler, SMM for NVIDIA Maxwell). The same is valid for what AMD calls Compute Units (CU) for its GCN GPU architecture. Finally, while in CUDA notation a thread grid is executed for a particular kernel (i.e. the complete 1D/2D/3D work-item array), the same concept is called NDRange in OpenCL. From the programmer point of view, thinking about how to parallelize the code is a little bit different than the usual OpenMP approach for CPU multi-threading for numerical applications (see A.4.2). In fact, with OpenMP the programmer focuses the attention on the for loop that requires parallelization. Using the compiler directives it is possible to split the work by assigning fractions of the total work to few different threads, usually in the same number of the available physical/virtual CPU cores. Each thread is assigned to a range of loop index values using a private loop index variable. In OpenCL, instead, the focus is shifted on the kernel. There is no for loop: each work-item executes one instance of the kernel. The work is differentiated between different work-items using an index variable that is private and assumes an unique value for each work-item. This value is returned by some specific function like get_global_id(). Since each different work-item uses the private index value to access buffers, SPMD/SIMD/SIMT parallelism is achieved.

As said, work-items are assigned to processing element, basically GPU cores, while work-groups are assigned to compute units. Usually, what happens in GPU architectures is that the difference between two GPUs of the same architecture but different performance levels is represented by the total number of active compute units (SM for NVIDIA Fermi architecture or CU for AMD GCN architecture). In fact, usually within the same GPU architecture each SM and CU has the same features and number of cores (e.g. in NVIDIA Fermi architecture each SM is composed by 32 cores). Thus by installing on the chip more SMs or CUs it is possible to increment the total number of cores in a very modular way, with a consequent performances improvements. Figure 4.6 represents the concept.

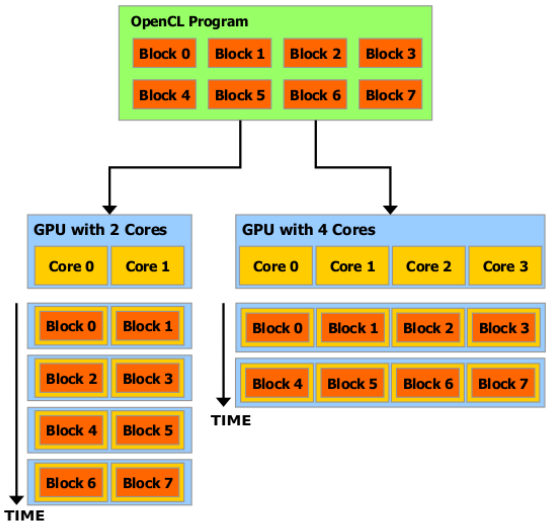


Figure 4.6: *GPU scalable architecture [114].*

4.4.2 OpenCL memory model and consistency

In the OpenCL abstraction, multiple memory levels are available for computations, each of which has particular features and accessing speeds. The memory levels available in OpenCL are basically mapped to the underlying hardware memory levels available on modern CPU and GPU architectures. The most important memory levels are represented by:

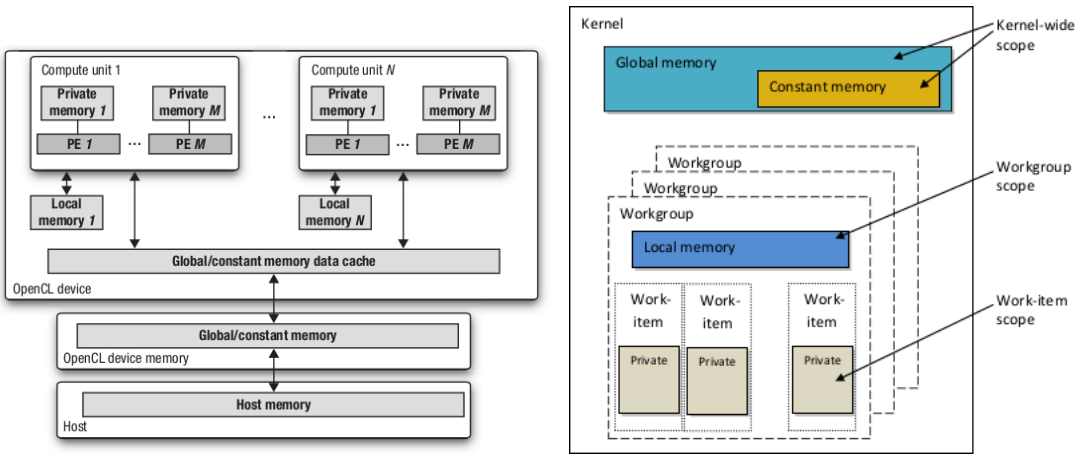
- **Private memory**: also called local memory in CUDA notation. This memory can be accessed only by the owner work-item in the same kernel execution, thus cannot be accessed by other work-items of the same work-group, by work-items of different work-groups or by the host. This is the fastest available memory and thus should be used for the bulk of computations. This memory is physically installed on the GPU chip and is limited;
- **Local memory**: also called shared memory in CUDA notation. This memory can be accessed by the work-items of the same work-group in the same kernel execution. It is slower than private memory but is faster than global memory. It is exploited when the algorithm requires some sort of collaboration between work-items of the same work-group. However, care must be taken since two different work-items cannot directly write on (or one writing and the other reading) the same local memory location concurrently. In order to avoid this kind of problems barriers have to be employed. This memory is physically installed on the GPU processor on each compute unit. The local memory values seen by work-items inside the same work-group are guaranteed to be consistent only at work-group synchronization points. Not all algorithms can exploit this memory level;
- **Global memory**: called global memory also by CUDA. This memory can be accessed by all work-items of the same kernel execution in read or write mode. This is basically the physical GPU memory that uses the GDDR5 or HBM technology and its size is in the order of GB. This is were inputs and outputs of

kernels are stored, since this is the memory used to exchange data between the host and then device. Here is where buffers are stored. Two work-items from different work-groups of the same kernel execution cannot write (or one write and the other read) at concurrently on the same global memory location. However, two work-items from the same work-group can do that if properly measures are used (e.g. barriers). The main problem is that this cannot be done for different work-items from different work-groups of the same kernel execution. In any case when the kernel execution is completed, work-items of subsequent kernel executions don't have any problem reading/writing the same global memory location, since an implicit barrier is placed at the end of any kernel. This means that for certain algorithms where some kind of collaboration (by exchanging global data) between work-items from different work-groups is required, multiple kernel executions are required. Global memory is the slowest memory level, orders of magnitude slower than local memory and private memory. This is due to the fact that global memory is off-chip. Thus, it should be only used to read input data and write results in order to maximize computational efficiency;

- **Constant memory**: called constant memory also by CUDA. This is physically implemented on the global memory from the hardware point of view but can be accessed faster than read/write global memory. This memory level can only be accessed in read mode and its size is limited;

- **Host memory**: this memory can only be accessed from the host and resides in the process virtual memory (from the hardware point of view the RAM);

Figures 4.7 shows the memory hierarchy, the relations between the different memory levels and the different OpenCL units.



(a) Relations between compute units, processing elements, host, device [111]

(b) Relations between work-items, work-groups, kernels [71]

Figure 4.7: *OpenCL memory levels.*

It must be highlighted that not all algorithms can exploit efficiently all the memory levels, especially the local memory level. It is worth to remind that when global

memory locations have to be accessed by work-items from different work-groups after a writing, multiple kernel executions have to be employed. Recently, with the new versions of OpenCL, atomic operations have been improved (using the C11 memory model), however they cannot be employed for every algorithm.

4.4.3 OpenCL code example

Here a simple example is provided in order to show different sources of problems that can slow down GPU execution, even to a level at which a single CPU core is faster. Here the aim is not to provide optimization hints, but rather show in which cases GPGPU is helpful and where instead CPU execution would be preferred. The example of the OpenCL C code to perform the sum of two vectors is firstly presented. The focus here is firstly posed on the conceptual differences between OpenMP and OpenCL for what concerns how to think about parallelism. Then the attention is posed on GPGPU performances, by showing why the simple vector sum is not that fast on GPUs. Then, the code will be modified to show in which kind of computations the GPGPU approach provides effective advantages over CPUs.

Listing 4.1 shows the kernel code that implements the sum of two vectors. This is the OpenCL C equivalent to the OpenMP-parallelized version A.1. While with OpenMP a for loop is employed and parallelized, here the focus is instead on the single work-item. Thanks to the fact that get_local_id(0) returns different values to different work-items the SIMD paradigm is realized allowing to use index i to access different data by different work-items.

Listing 4.1: *Vector sum kernel code*

```
__kernel void vectorSum
(
  __global float* a,
  __global float* b,
  __global float* c
)
{
  int i = get_global_id(0);

  c[i] = a[i] + b[i];
}
```

Alongside the kernel code, on the host buffers related to arrays a, b and c are allocated with size vectorSize. These buffers reside in global memory, while the variable i is stored in each work-item's private memory as it is declared within the kernel. The host enqueue the kernel execution using host code 4.2 where it also specifies the total number of work-items to be spawned.

Listing 4.2: *Vector sum host code for kernel execution*

```
err = clEnqueueNDRangeKernel(queue, vectorSum, 1, NULL, vecSize, NULL, 0, NULL,
    NULL);
```

The host code related to OpenCL environment initialization, finalization and buffers allocation is not showed here for simplicity. Kernel code for vector sum is composed by two global memory reads (a[i], b[i]), one floating point operation (+) and one global memory write (c[i]). These operations are performed by each of the total vectorSize

number of work-items. Let's consider now the execution of both the OpenMP version on a notebook CPU (Intel i5 2410m, $2.3\,GHz$, 2 core, 4 threads (HyperThreading), $\sim 40\,GFLOPS$ single precision) and the OpenCL version on the same notebook on the GPU (NVIDIA GT540m, $\sim 250\,GFLOPS$ single precision). The overall speed-up provided by the GPU when the sum is performed over $10k$ elements is 0.07, meaning that the CPU is orders of magnitude faster than the GPU. This poor result could be obtained due to small vector size, such that the time required to fire up the OpenCL environment is bigger than what required by the actual computations. Trying with $10M$ elements the speed-up is 0.14. Again the CPU is faster even tough theoretically the GPU has more GFLOPS. The problem here is given by the fact that, as said in 4.4.2, global memory is relatively slow with respect to GPU cores speed, requiring different clock cycles for data to be fetch, even when, like in this case, memory coalescing is possible. The problem here is that for two global memory reads, just one floating point operation is performed. This is confirmed by the fact that when using a kernel with an high floating point operations to global memory reads ratio like in 4.3 the speed-up achieved is $37.48\times$.

Listing 4.3: *Vector sum kernel code*

```
__kernel void heavyFun
(
 __global float* a,
 __global float* b,
 __global float* c
)
{
  int i = get_global_id(0);

  real ap = a[i];
  real bp = b[i];
  real cp = 0;

  for(int i = 0; i < 30; i++)
  {
    cp = 2.025*ap - 3.253*bp + 0.225*ap/(bp+1)/(cp+1);
    ap = 1.205*bp + 0.151*bp/(cp+1) - 0.107*ap/(cp+1);
    bp = 4.545*cp - 0.181*bp/(ap+1) + 0.654*cp/(bp+1);
  }
}
```

This is an important concept because in the solver high speed-ups can be achieved, especially for the convective fluxes computations (see 5.3.2) wherever numerous floating point operations are performed between the initial global memory input reads and final writes.

Finally, it is noted that since basically all the kernels implemented in this work do not make use of work-group collaborative operations, multi-dimensional NDRanges or local memory, the syntax 4.4 can be wrapped up using a macro as follows:

Listing 4.4: *Vector sum host code with wrapper*

```
err = runKernel(queue, vectorSum, vecSize);
```

CHAPTER 5

GPU implementation

In this chapter the architecture of the solver will be presented. The solver is based on the OpenFOAM framework and uses OpenCL as the API and programming language to achieve GPU acceleration. OpenCL is preferred over CUDA thanks to its wider compatibility in terms of vendors and types of processor. As discussed in chapter 4, AeroX is composed by two sets of source code files: one for the host and one for the device. The most important concepts related to the implementation of the algorithms discussed in 2 and 3.5 for the solution of aerodynamic and aeroelastic problems will be presented, with the focus on the computational efficiency reasons behind each choice. In particular, since one of the targets of AeroX is the compatibility with unstructured hybrid meshes, the efficiency problems behind this goal will be analyzed. The implementation choices are made with the aim of obtaining a fast GPU solver by distributing work chunks among the hundreds/thousands available GPU cores. However, thanks to the available CPU implementations of OpenCL, the solver is also natively compatible with multi-core CPU architectures without any source code modification. It must be noted that numerous algorithms implemented in the solver are written from scratch, while others are obtained from a specific tuning of what already available from the open source AeroFoam solver [127, 136].

5.1 Solver programming language and libraries

The aim of this section is to briefly introduce the reasons behind the use of some peculiar libraries and tools and how they are used inside the solver. These are represented by OpenCL and OpenFOAM.

5.1.1 OpenFOAM

As said, the main purpose of this work is the development of a GPU-accelerated aeroelastic compressible (U)RANS solver with turbomachinery and open rotors extensions. The bulk of the computations is related to aerodynamics and, in a smaller weight, to structural computations and mesh deformation. These computations have to be performed several times during a steady or unsteady simulation. However, there are kinds of computations that are performed only once for simulation. In particular, several preprocessing algorithms required to read the mesh and generate all the mesh metrics are performed just once at the very beginning of the solver execution. Depending on the mesh size, these algorithms are executed in few milliseconds or seconds while the aerodynamic convergence could require minutes/hours/days. Following the very basic ideas of profiling, it is worth to focus the attention on the optimization of the parts of the code where more computational effort is required. Furthermore, code reuse is always a good choice when the available code is reliable, optimized, compatible and exhibit several features. Following these ideas, the OpenFOAM framework is here employed for the pre-processing and post-processing stages of the simulations. OpenFOAM provides a very powerful C++ API (classes, functions, types,...) and is well suited for the coupling with the C-based OpenCL API. OpenFOAM sources come with a wide range of pre-implemented solvers, opportunely designed and tuned to solve ad-hoc CFD problems. The advantage provided by OpenFOAM is represented by the fact that the 3 phases of a typical solver execution (i.e. pre-processing, processing, and post-processing) are well separated. This way, it is possible to focus the attention on building the OpenFOAM-based solver, i.e. writing just the code for the processing phase that satisfy the specific requirements, without taking too much care of the pre/post-processing phases which are instead delegated to OpenFOAM. Thanks to the fact that OpenFOAM exploits C++ objected oriented concepts, the solver can be written at a very high level syntax which is very similar to equations written on paper. This is accomplished thanks to classes and operators overloading. The user does not have to take care of how equations are mapped to low level memory objects (arrays, matrices and systems to be solved). This is all done under the hood by OpenFOAM. OpenFOAM is an open-source project, so any programmer can modify the OpenFOAM code to fully achieve the prefixed goals when certain algorithms are not yet implemented. Code re-use is the idea behind the use of OpenFOAM in this work. However, due to the low-level nature of OpenCL to achieve GPU acceleration, the object-oriented mentioned way of implementing new OpenFOAM-based solver could not be directly adopted. This is due to the fact that OpenFOAM is intrinsically CPU-based, eventually with multi-process parallelization through MPI and domain decomposition. For the aims of this work, OpenFOAM is completely bypassed for what concerns the CFD/FSI computations. However, the OpenFOAM API, as mentioned, is still used for certain mesh metrics pre-processing and the final post-processing. An important advantage is given by the OpenFOAM output format compatibility with ParaView [29] that ease results visualization. In particular, while OpenCL is here adopted to provide portability and compatibility with a wide range of devices, OpenFOAM is adopted thanks to its compatibility with a wide range of mesh types and formats. Thanks to OpenFOAM mesh metrics routines, AeroX is fully compatible with structured, unstructured and hybrid meshes. Furthermore, thanks to the numerous OpenFOAM utilities, it is possible to convert the most

used mesh formats (e.g. msh) to the OpenFOAM native format, guaranteeing compatibility with many mesh generation tools. During this work, mesh generated with the most adopted tools, from academic world to industry, like gmsh, gambit, HyperMesh and Icem were successfully processed.

OpenFOAM uses the concept of "field" to store face-based and cell-based scalar, integer and vector data. Fields are basically C++ classes built around more low-level arrays. Fields provide high-level interfaces to the programmer. Thanks to C++ classes, the programmer interacts with fields rather than low-level arrays to store data, since they provide an easy and intuitive interface to access face-based and cell-based data and to perform post-processing.

The developed solver, AeroX, that actually performs CFD/FSI computations is written from scratch due to the use of OpenCL and the fact that the chosen CFD/FSI formulations are not directly found inside OpenFOAM sources or they are slightly different.

5.1.2 OpenCL

The solver host code is written in C/C++. OpenCL API functions and data types are used in the host code in order to:

- Initialize the OpenCL environment;
- Select the device aimed to execute kernels;
- Read and compile device code;
- Create and transfer buffers from/to the host to/from the device;
- Enqueue kernels execution on the device;
- Free buffers memory;
- Finalize the OpenCL environment;

Like with any other library, when programming a C/C++ application, in AeroX host code the #include directive is used to include the OpenCL header file. This is usually CL/cl.h. The host code is compiled like any other application using gcc and the opportunely specified optimization flags. Also, the OpenCL shared library (libOpenCL.so) is dynamically linked to the solver executable. From the host code point of view what described is just what required for any other application development. The computation offloading to the device is handled by the OpenCL implementation and device drivers at runtime, effectively freeing the programmer from this effort. This is one of the strengths of OpenCL.

In order to achieve the widest hardware compatibility, at the time of writing, the code uses OpenCL version 1.x. This is due to the fact that NVIDIA does not currently support OpenCL 2.x while Intel and AMD are already shipping it within their drivers/libraries/runtime implementations. One of the goals of the solver is to achieve wide GPU compatibility, in order to exploit the computational power already available in workstations, without requiring to buy new devices. Since NVIDIA is currently the most important GPU vendor alongside AMD, it is basically mandatory to achieve compatibility with its hardware.

5.1.3 Interfacing OpenCL and OpenFOAM

As said, one of the most important problems when writing the solver code is interfacing OpenFOAM and OpenCL. This is due to the fact that OpenFOAM has its own C++ classes to store arrays of data and the programmer has to use them to access mesh data structures during pre-processing and to easily save data structures for post-processing purposes. When writing an OpenFOAM-based solver using the native API, data is saved in fields that contains cell-based and face-based information. Fields are also equipped with useful functions allowed to process that data. However, the problem here is that the code that actually performs CFD/FSI computations is written in OpenCL and cannot directly interacts with the OpenFOAM framework. As explained in 4.4, when programming with OpenCL, buffers must be used to exchange data between the host and the device. Furthermore, buffers are accessed by the device during kernel execution. This basically means that when executing kernels, i.e. when solving CFD/FSI equations, OpenFOAM-based high-level objects like fields cannot be directly adopted. Buffers are basically low-level arrays of data without any particular C++ method built around them. Thus for each field of data that has to be accessed by a kernel to perform CFD/FSI computations, a buffer has to be allocated. Buffers can be read-only with respect to the device and filled during the pre-processing stage on the host with data contained in fields. This is the case of mesh addressing data since the mesh deformation strategy adopted in AeroX does not rely on re-meshing algorithms. All the other buffers are created with the read-write flag. This is true both for CFD solution data and for mesh metrics data (e.g. cells volumes, faces areas) when performing FSI simulations with mesh deformation. For certain buffers for which it is known that they are written by the device before any possible read, they are just allocated without being filled, thus saving computational time. This is the case of buffers that contain residuals. Instead, other buffers, like the ones that contain the Navier–Stokes solution (i.e. mass, momentum and total energy) have to be filled before kernels computations with initial/guess conditions. Furthermore, care is required during post-processing. In fact, in order to use the OpenFOAM post-processing API, fields have to be used. However, fields have to be filled with data computed by kernels and stored into buffers. This means that data within buffers has to be copied back from the device to the host and finally to the fields.

OpenFOAM is an open-source project used by many researchers. Currently, different versions of OpenFOAM are available. Despite the necessity to release new features and bug fixes, different versions of OpenFOAM are also available due to forks happened during last years. The adopted version at the time of writing is OpenFOAM 2.2. Newer versions, like 2.3, 2.4, 3.0 could potentially be used. However, this is not strictly necessary because, as said, OpenFOAM API is just used for pre-processing and post-processing. It is underlined that all the code that actually performs CFD and FSI computations is represented by kernels written in OpenCL C language from scratch, thus it is completely independent from the OpenFOAM API and formulations. The equations have to be solved through low-level programming, without the possibility of taking advantage of the OpenFOAM high-level, object-based syntax to represent the different equation terms.

An important aspect for GPU acceleration is the exploitation of single-precision floating point computations. The main idea is to use cheap gaming GPUs as devices,

thus single precision (SP) executions, while maintaining the native compatibility with HPC-specific GPUs that exhibit high double precision (DP) performances. OpenFOAM framework does not directly use double or float types but rather defines the "scalar" type. This is basically an alias to the actual underlying C float of double data type. In particular the floating point precision of OpenFOAM can be chosen before compilation using an environment variable, WM_PRECISION_OPTION that could be SP or DP with obvious meanings. This means that two different compilations of the OpenFOAM framework are required to allow the solver to completely switch between SP and DP. Similarly, in the solver host code that uses the OpenFOAM framework, the scalar type is adopted, requiring again two independent AeroX compilations to achieve both SP and DP capabilities. Before using any OpenFOAM-based solver or utility, the OpenFOAM environment has to be loaded. The switching between SP and DP is easily achieved by choosing between SP and DP OpenFOAM environments. The situation is instead different for what concerns the device code, i.e. kernels. Kernels are written in OpenCL C. Here a typedef using an alias called real is adopted to achieve the same concept of the OpenFOAM scalar type. However, since device code is compiled at runtime, a way to tell to the device the underlying precision of the real type is required. All kernels files are merged into a single device code file that is compiled at runtime. The idea used to switch between SP and DP on the device is to create a device code header file that contains the definition of the real type. This is placed as the first file before the device code merging, allowing subsequent kernels to know if real means float or double. This way, by selecting the correct header before the runtime compilation, it is possible to obtain the same floating point precision both on the host and the device. Obviously this is not the only way to achieve this goal but is nonetheless very simple and reliable.

Since OpenFOAM is used in this work for the pre-processing stage, it is worth to briefly discuss the most important data structures regarding the mesh since these are then handled by the solver host code and kernel code. The user has to provide the mesh in the OpenFOAM format. If this is not directly possible, different utilities, provided by the OpenFOAM framework, can be used to perform the required conversion. Initial conditions, boundary conditions and solver settings are read at runtime from specific files. constant/polyMesh/boundary contains the physical type of each boundary. system/controlDict contains the solver settings. The 0 folder contains the values of initial conditions and boundary conditions. This is better explained in [136]. At runtime the first operation performed by the solver is the read of the mesh through the OpenFOAM API. This way the mesh is stored in the native OpenFOAM format. In order to better understand the data structures accessed in kernel code showed later in this chapter it is worth to introduce some details about how OpenFOAM organizes mesh data during pre-processing. Each mesh is composed by Nv cells and Nf faces. Among the Nf faces, Nfi faces are internal faces while Nfb faces are located on boundaries. OpenFOAM handles the mesh in such a way that the IDs of boundary faces start after the last internal face ID. Furthermore, faces of the same boundary have sequential IDs. In OpenFOAM a single boundary is called patch and is composed by a set of faces sharing the same boundary type. Different types can be used, e.g. wall to tell OpenFOAM to use the patch for wall distance computations, patch to tell OpenFOAM that it is a generic patch, cyclicAMI to tell OpenFOAM to perform addressing

computations to find matching faces on periodic boundaries. It is worth to remind that faces of each patch have a global ID that has to be greater than Nfi and in the same patch the IDs are consecutive. Thus, it is also possible to compute an ID local to the patch and directly related to the work-item ID if a boundary condition kernel is executed with an NDRange composed by a number of work-items equal to the number of patch faces. Each internal face is characterized by a cell "owner" and "neighbor" that represent the two cells that share that face. The unit normal vector is directed from the owner to the neighbor. A boundary face has only the owner cell and its normal vector is exiting the fluid domain. OpenFOAM API provides also other useful information regarding the connectivity. For each cell it is possible to extract the global face IDs of surrounding cells and faces. This is useful for assembly operations concerning e.g. residuals and gradients, as will be showed in this chapter. It is reminded here that while all these structures are provided by OpenFOAM in a very intuitive object-based C++ interface, when implementing the solver using OpenCL, buffers that are basically C-like arrays of data must be adopted. Thus, multi-dimensional structures must be re-mapped into one-dimensional buffers with proper addressing. In order to understand how the addressing is performed it is worth to show an example. The example is specific for the residual assembly computation but the same concepts can be applied also for other kind of addressing. The data structure under investigation provides for each domain cell the global IDs of the surrounding faces. This is fundamental in the code since the idea is to save fluxes in buffers of Nf elements, storing each single flux, (e.g. density flux), in buffer location i. Thus, during residual assembly, the work-item k assigned to cell k need to know the global IDs of faces surrounding it. Listing 5.1 shows the host code that exploit OpenFOAM API to extract this information:

Listing 5.1: *Cell to faces adrressing host code*

```
// Allocation of 1D array containing for each cell k the the cumulated
// total number of surrounding faces.
labelList cellFacesDelimiter(Nv + 1);
// First element set to 0
cellFacesDelimiter[0] = 0;
// Lopp over cells
for( int k = 0; k < Nv; k++ )
{
    // Extract total number of faces of cell k
    label numFaces = mesh.cells()[k].size();
    // Fill the array with the number of faces for cell k (cumulated)
    cellFacesDelimiter[k + 1] = cellFacesDelimiter[k] + numFaces;
}

// Allocation of 1D array containing for each cell k the sequence of surrounding
// cells
labelList cellFaces(cellFacesDelimiter[Nv]);
// Loop over cells
for(int k = 0; k < Nv; k++ )
{
    // Extract the array of faces surrounding cell k
    labelList faces = mesh.cells()[k];
    // Find the starting location in the array to begin writing data
    label startFace_idx = cellFacesDelimiter[k];
    // Extract total number of faces for cell k
    label numFaces = cellFacesDelimiter[k+1] - cellFacesDelimiter[k];
    // Extract faces global ID and use them to fill the array
    for( int i = 0; i < cellFaces )
    {
        cellFaces[startFace_idx + i] = faces[i];
```

```
    }
}
```

The first line is used to allocate a labelList, an OpenFOAM class that defines basically an array of integers with useful public methods. The size of this array is set to Nv + 1 and the first element (element 0 in C/C++) is set to 0. This is done because data in this array will be used for two purposes inside kernels. Inside the first loop the mesh object is used to extract the total number of faces of cell k using the mesh.cells()[k].size() public method. The result is temporarily stored in the numFaces variable. This is used to store inside the cellFacesDelimiter the cumulated number of faces of each cell k. The reason behind this will be explained in a moment. After the first loop, the last element of cellFacesDelimiter is used to allocate a new array, cellFaces that will contain for each cell k the IDs of surrounding cells. Inside the loop, first the array containing the faces of cell k is obtained through the public method mesh.cells()[k]. Then, the previous filled array, cellFacesDelimiter, is accessed with the cell ID, k, to obtain the starting point, startFace_idx to access cellFaces. Thanks to the strategy of cumulated number of faces it is possible to store in a single array all the faces of all the cells, even if it is an hybrid mesh where each cell has different number of faces. In fact it is also possible to obtain with the same array the total number of faces, numFaces, surrounding cell k. The last step is a loop over the faces in order to store the faces global IDs into the cellFaces array in the correct startFace_idx position. The way these particular buffers are accessed inside kernels is explained in details when discussing residuals assembly in 5.3.6 and gradients computations in 5.3.5. It is noted that this addressing strategy is used also for other purposes, e.g. to access data of nodes surrounding faces and cells surrounding cells. Obviously, when employing addressing strategies on GPU architectures performance issues related to non-sequential memory accesses and branch divergence could happen with hybrid unstructured meshes. The presented strategy is basically performed in order to map arrays of arrays into linear memory positions.

5.2 Solver architecture details

5.2.1 Overall scheme

Here the general scheme of the solver is showed. It must be noted that due to obvious visual limitations this does not represent a detailed view of every solver sub-components and their relations. However it is anyway useful to better understand how the formulations described in 2 and GPGPU concepts are translated in the solver in multiple modules exchanging data with each other. Figure 5.1 briefly shows the scheme of the solver. From the figure it is possible to see that the mesh generated with different tools can be imported in AeroX after the processing with the right OpenFOAM mesh conversion utility. Then, the mesh is directly read by AeroX using the OpenFOAM pre-processing API. This block is also used to read boundary and initial conditions (BC/IC) using the files located in the 0 folder of the OpenFOAM case file structure. BC/IC can be also read from re-start files. The same block is also used to read AeroX settings through the system/controlDict and other files. After the OpenFOAM pre-processing, AeroX pre-processing host code is executed, using OpenCL API types and functions in order to build buffers and other useful objects. This block is also used to execute some

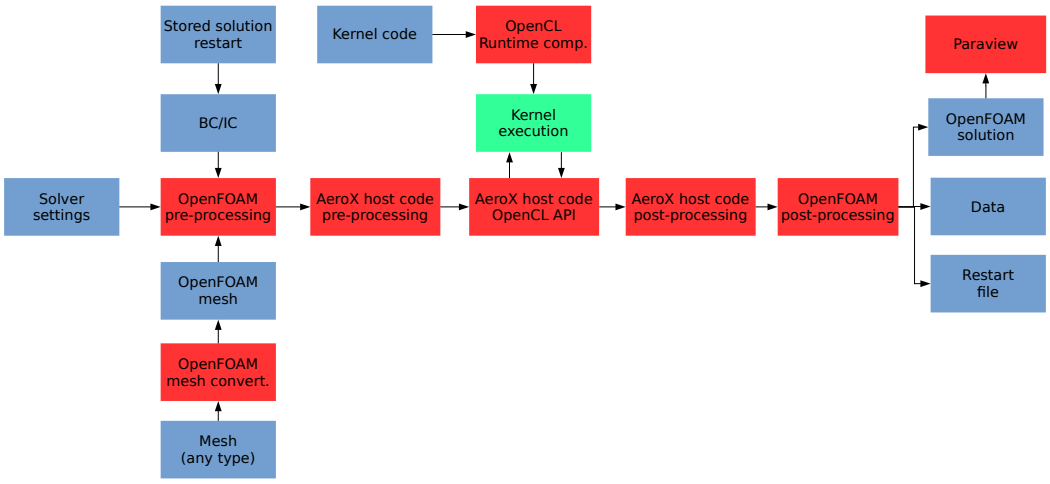


Figure 5.1: *AeroX software architecture. Blue: input/output files; Red: host execution; Green: device execution.*

AeroX specific algorithms not included in OpenFOAM, e.g. searching for extended cells for high resolutions schemes. The plain-text kernels code is compiled at runtime using ad-hoc OpenCL API and using the OpenCL implementation distributed by Intel, NVIDIA and AMD with their drivers/SDKs. When both the host and the device are ready for CFD/FSI computations, the host enqueues work on the chosen device. During computations the host and the device exchange just the essential data (convergence check, wall forces and displacements). Finally, the solution is read back from the device to the host and is firstly post-processed by AeroX specific code (e.g. to convert conservative variables into pressure, temperature and velocity). Then, OpenFOAM post-processing API is called to export the solution in order to be graphically processed with Paraview [29] or other OpenFOAM utilities.

5.2.2 Convergence check

As mentioned, within an OpenCL application, the host job is to organize and enqueue work on the device, while the device actually performs numerical computations. One important problem faced in writing a CFD/FSI solver is the check for results convergence. There are peculiar applications for which it is known a-priori the total number of computations to be performed. In this case the host just enqueues all the work to the device and waits until completion. However in CFD/FSI simulations the total number of iterations required to reach convergence is not know a-priori. The problem here is the separation of jobs between the host and the device. The host has the ability to enqueue further work to the device if the convergence is not yet reached but is the device that is actually computing the solution and thus is storing residuals in the global memory. It is therefore necessary to send data from the device back to the host to check the current convergence state. As explained in 4.2, data is exchanged between the host and the device using the PCI-Express bus, potentially leading to bottlenecks. The host needs to read data from buffers shared with the device and it has to be guaranteed that the device is not simultaneously writing those buffers to prevent race conditions. With OpenCL it is possible to overlap host-device buffer data transfers and kernels

executions without any problem if the currently executing kernel is not using the buffer that is being transferred. However, in AeroX there are certain regions where the device execution has to be stopped to allow data exchange before continuing with CFD/FSI computations since essential data like the solution and residuals are required on the host. This kind of data are in fact required in numerous kernels. Thus, it is better to temporarily block the device computations to perform data transfers. One could argue that this could lead to an excessive overhead, slowing down the entire execution. This problem is tackled by checking convergence not every pseudo time iteration but every N pseudo time iterations, usually multiple of 1000. In fact, this is an explicit solver, thus if the solution is far from convergence at the $K-th$ iteration, it is improbable that it is converged at the $(K+1)$-th iteration. However it could be at the $(K+1000)$-th iteration. A check for convergence every multiple of 1000 iterations represents a perfect trade-off between the need to allow the device to perform as much computations as possible without data-transfer overhead and the need to check convergence by the host to stop the simulation as soon as convergence is reached to avoid wasting computational time. By employing this strategy the time required for convergence check is basically negligible over the entire simulation. In fact, it must be noted that only the buffers required to check convergence are transferred. Other data is kept on the device and remains out-of-date on the host until explicitly required by the host, e.g. to perform post-processing.

Here a C-like pseudo-code of the procedure is showed to clarify the idea:

Listing 5.2: *Convergence check*

```
// Loop until convergence or max toatal iterrations reached
while( !coverged() && totalIter < maxTotalIter )
{
    // Internal loop, check convergence every maxCheckIter iterations
    for(checkIter = 1; checkIter < maxCheckIter; checkIter++)
    {
        // Perform aerdynamic computations (i.e. run kernels)
        aerodynamicIteration();
        totalIter++;
    }
    // Exchange data device -> host
    readDataFromGPU();
    // Finally check convergence and eventually exit the while loop
    checkConvergence();
}
```

where, as explained, a good trade-off value for maxIterCheck is 1000.

5.2.3 Numerical tricks for single precision

The possibility of using single precision floating point representation allows AeroX to exploit the high computational power offered by modern cheap gaming GPUs alongside the single and double precision power offered by (more expensive) modern HPC GPUs. SP provides both speed-up and memory advantages with respect to double precision due to GPU processor architectures and the fact that floats require half the memory space of doubles ($4\,bytes$ instead of $8\,bytes$). Obviously the most important problem when exploiting SP is the reduced precision with respect to DP. For certain applications DP is mandatory. This could be true also in CFD fields, especially when some kind of high order formulations is adopted. The purpose of this solver is however

to provide a fast solution using high resolution compressible (U)RANS formulations. For this purpose DP is perfectly adequate, although SP could be used with some precautions to reduce numerical errors. Both industry and academic oriented CFD solvers have usually the option to use DP or SP. The nature of the problem relies in the fact that when performing numerical simulations using computers, due to the finite memory and computational power available, it is necessary to trunk the total number of digits of a number at a certain point. Problems arise when two numbers with very different orders of magnitude are combined together in an arithmetic operation. As an example, if a very small number is added to a very big number, it could be possible that the differences between the orders of magnitude of the numbers are such that the sum is numerically equal to the biggest number. This is due to the nature of the floating point finite-precision. Of course this basically means that the smallest number is negligible with respect to the biggest number and from an engineering point of view neglecting the small number could be perfectly legit. However this is acceptable as soon as this operation is performed once. This leads to problems instead when the operation is repeated hundreds/thousands of times and an effective variation of the results is expected. Consider as an example a reduction operation where millions of small values are added up. If this operation is performed serially, using a single precision variable to store the sum and sequentially adding each element to the sum, it is possible that when the sum variable reach an high enough value, the subsequent small values are neglected. This could lead to wrong results on the final sum. Consider the following code:

Listing 5.3: *Sum reduction*

```
#include<iostream>
#include<stdlib.h>

int main(int argc, char* argv[])
{
  // Initialize the iter variable and the sum in double and single precision
  int iter = 0;
  double DPsum = 0;
  float  SPsum = 0;

  // Read user input (total number of iterations)
  iter = atoi(argv[1]);

  // Perform sum computation in DP and SP
  for( int i = 0; i < iter; i++ )
  {
    DPsum += 1.0e-6;
    SPsum += 1.0e-6;
  }

  // Finally print the sum in DP and SP
  std::cout << "DPsum: " << DPsum << std::endl;
  std::cout << "SPsum: " << SPsum << std::endl;

  return(0);
}
```

when executed for $1k$ iterations leads to the following results:

```
[andrea@arch-polimi testSum]$ ./exe 1000
iter: 1000
DPsum: 0.001
SPsum: 0.001
```

when executed for $1M$ iterations leads to the following results:

```
[andrea@arch-polimi testSum]$ ./exe 10000000
iter: 10000000
DPsum: 10
SPsum: 9.59211
```

One way to reduce the problem could be splitting the entire reduction into multiple sub-reductions and finally adding up the intermediate results. The idea is to try, as much as possible, to add up values with similar orders of magnitude. This is basically the strategy usually adopted to perform reductions on many-cores architectures like GPUs [9] although in that case the main purpose is the full exploitation all the GPU cores. In any case, this kind of problems are particularly important during the initial iterations.

Despite sum, division and multiplication could potentially suffer from numerical problems, especially when using SP. The problem here mainly represented by overflow conditions when the result of an operation is bigger than the maximum number that can be represented with SP. This can happen when for some reasons two big numbers are multiplied together or when a big number is divided by a very small number. Consider as an example the following piece of code related to the SST turbulence model:

Listing 5.4: *Possible overflow 1*

```
real Yomega = beta*trho*twTur*twTur;
```

which is basically the source term $\beta\rho\omega^2$ (see 2.1.7 [107, 108]). It is possible to see that the turbulence model variable ω is raised to the power of two. ω is usually relatively big near walls, especially when very fine wall discretization is employed (very small y^+). It must be noted that, in theory, the value of ω should be infinite on the wall. This, alongside the fact that during the first iterations the value of ω could change by orders of magnitude from the guess values, floating point overflow problems could happen. To tackle this, the maximum value of ω is limited to a maximum value. In fact the greatest number that SP can represent has exponent $+38$. A good maximum value for ω is represented (e.g.) by $1.0 \cdot 10^{15}\ Hzs$. It must be noted that usually, in solutions obtained by both when the near wall discretization allow y^+ inside the viscous sublayer and in the log region, ω does not reach these orders of magnitude. However, these orders of magnitude could be reached during the first iterations starting from the initial guess. Thus, to prevent NaN and Inf values, this this strategy is adopted. Once a NaN or an Inf is obtained it rapidly propagates to the entire computational domain, leading to simulation failure. It is noted that if the simulation fails for these reasons on GPUs, the user is warned only after the results are read back to host. After the initial iterations the value of ω usually starts to converge to reasonable values and the bounding strategy is not necessary anymore. This is just an example, as in the solver there are other places where this strategy can be successfully applied. Floating point overflows, in fact, could happen also when divisions are performed between big numbers at the numerator and small numbers at the denominator or when the denominator approaches the value of 0. This again could lead to results as NaN or Inf that rapidly invalidate the entire solution but that can be prevented with simple strategies. The idea is again the fact that this kind of problems happens during the initial iterations while near the final solution the variables should contain values that do not lead to any of this kind of numerical problems. Consider another example, again concerning SST turbulence model:

Listing 5.5: *Possible overflow 2*

```
real KW_wTurWall( real rho, real mu, real y, real wTurMax )
{
    return (real)(10*6.0)*mu/( ( (real)KW_beta1*rho*y*y ) + (real)SMALL );
}
```

this represents a simple function to compute the value of $\omega|_{wall}$ as a wall boundary condition. It is possible to see that if the near wall region is discretized such as y is very small, the denominator could reach very small values, leading to possible overflows. In order to tackle this problem, a SMALL value is added. This is basically a strategy to avoid Inf/NaN and at the same time to limit the maximum value of $\omega|_{wall}$ before its computation. Another operation that could potentially lead to problems is sqrt, when its argument, that normally should be positive, becomes small and finally negative. Again, if it is known that from its physical meaning the argument has to be positive but during the first iterations it becomes negative, the strategy is to force the argument to be at least bigger than a certain small value, bounding it through max(argument, SMALL) function.

Furthermore, it must be noted that SP computations could improve convergence with respect to DP due to purely numerical dissipative effects. Anyway, it is mandatory to check and validate the solver for both SP and DP. The idea is to simulate the same case using both DP and SP, using DP solution as reference to check for possible local and integral differences between SP and DP solution. This is showed in 6.2.4.

Besides this kind of "tricks" to bypass single precision numerical issues related to Inf/NaN problems it is also necessary to minimize numerical accuracy degradation due to reduced precision (with respect to DP). The idea is to solve the Euler/NS/(U)RANS equations in their non-dimensional form. Basically, the user inserts boundary and initial/guess conditions in their intuitive dimensional form. Then, when the solver is launched, all the input fields are transformed in their non-dimensional equivalent. The equations are solved using non-dimensional variables. Finally results are transformed back to dimensional form for easier post-processing. The idea here is to try to have all the variables with different physical meanings in similar orders of magnitude aiming to reduce numerical accuracy loss due to the floating point nature.

5.2.4 Debugging the device code

It is worth to discuss few aspects about how the device code has been debugged during AeroX development. As said, the solver is composed by the host code and the device code. Debugging the host code is done in the same way as for any other application with usual tools like gdb and valgrind, especially when encountering segmentation faults and floating point exceptions during execution.

The situation is different for what concerns the kernel code, written in OpenCL C and compiled at run-time by the selected OpenCL implementation for the execution on the selected device. Debugging kernel code is more difficult for different reasons. However, few tools and strategies are enough to solve almost any problem. Here, the debugging of the two most important source of bugs, floating point exceptions and memory accesses errors for the device code are briefly discussed. One of the reasons behind the difficulties in debugging device code is that on GPUs there is no operating system kernel in execution, thus there is no checking and signaling for memory and arithmetic errors. Thus, these checks must be performed in other ways.

For what concerns floating point problems, they usually manifest themselves in solver crashes due to floating point exceptions or NaN/Inf printed in residuals and solutions. In fact, if a NaN/Inf is obtained in kernel it is then propagated in residuals and consequently the solution. When these are read back to the host the user can easily see that a problem occurred. As said in 5.2.2, residuals and solutions are obtained from the GPU every user-defined number of iterations, and, since there is no direct check by the GPU over arithmetic errors, it is possible that the problem manifested itself during an iteration performed before the iteration immediately preceding the residuals and solution check. Thus, it is important to find the exact iteration where the problem happened. Then it is possible to find the kernel where the problem happened and finally the kernel code line. The strategy adopted to find the iteration and the specific kernel involved in problems consists in reading back to the host all the buffers written by the kernels just after their execution and search for possible NaN/Inf values between the buffers elements. The latter operation can be easily performed with a for loop and the isnan() and isinf() functions provided by the C/C++ standard library. The adopted code is wrapped using macros and called right after kernels executions. For every buffer involved in each executed kernel the macro showed in listing 5.6 is called:

Listing 5.6: *Search for NaN/Inf inside kernels*

```
# define checkNaN( gpuBufferName, cpuArrayName )                          \
clEnqueueReadBuffer( queue, gpuBufferName##_GPU, CL_TRUE, 0,              \
                     sizeof(cl_real)*gpuBufferName##_SIZE, &cpuArrayName[0],  \
                     0, NULL, NULL );                                     \
{                                                                         \
    for(label k = 0; k < cpuArrayName.size(); k++)                        \
    {                                                                     \
        if( isnan( cpuArrayName[k] ) || isinf ( cpuArrayName[k] )         \
        {                                                                 \
            std::cout << "At iteration " << N << " "                      \
                      << "after " << KERNEL_NAME << " "                   \
                     << #cpuArrayName << "[" << k << "]"                  \
                     << " is NaN/Inf" << std::endl;                       \
            found++;                                                      \
        }                                                                 \
}
```

The idea is to firstly read the buffer from the device into the host using the clEnqueueReadBuffer function provided by the OpenCL API. Then it is possible to check every element of the buffer for NaN and Inf values. Every buffer is read in blocking mode. Obviously this slows down the executions due to the necessity of read multiple buffers for different iterations before finding the exact point in the execution when the problem is raised. Once the iteration and the kernel are found it is possible to check for the exact operation involved with the arithmetic issue and possibly fix it. The programmer can be helped by the use of the printf() function. At the time of writing Intel, NVIDIA and AMD implementations support OpenCL extensions that allow the use of printf() function inside kernels. With this function it is possible to write on screen results of the computations while executing kernels. This function is supported by the aforementioned device vendors both for CPU and GPU executions. It is also a good choice to force the execution on a single core. This is also useful to perform benchmarks. It is worth to notice a problem related to debugging, especially with SP. For regular CPU code, the programmer writes a code that is then translated into the underlying machine code by the compiler. During compilation and execution, however,

instructions re-ordering could be applied for optimization. A possible drawback could be that also on GPU modifying the code to print variables could result in a different operations reordering. This in turn could influence the original code that involves the numerical issue, resulting into a more difficult debugging. This could happen especially when employing SP since operations could be quite sensible to numerical errors.

When programming with performance-oriented languages like C and C++ there is no direct check on array bounds. Array boundary checks have to be explicitly implemented by the programmer. It is possible that a programming error could result in reading or writing in wrong memory locations. This would lead to buffer overflows. When this happens, usually the involved process receive a segmentation fault signal from the kernel. When this kind of bugs afflicts GPU executions what is seen during this work is that either the solver crashes with NaN/Inf results or it crashes with an error raised by the OpenCL API. Finding this kind of bugs is quite difficult. However a relatively new tool, oclgrind [27] has been developed. As mentioned before, with this tools it is possible to check buffers read/write operations for possible overflows. It is also possible to check for possible data-races.

It must be noted that when the CPU is selected as the device, since the threads are handled by the operating system kernel, memory accesses and arithmetic problems are more easily caught. Furthermore vendors usually provides their own debug tools alongside with their OpenCL SDK packages, e.g. CodeXL for AMD.

5.2.5 Profiling the device code

It is worth to discuss few aspects on how the kernel code has been profiled during this work. Here the focus is on profiling single kernels and not entire executions. Thus the aspects here showed are not directly exploited for the results showed when benchmarking overall GPU executions against CPU executions (see 6.2.1). It is useful to benchmark single kernels, rather than overall executions, in order to check if a strategy performs better than another, especially when non-sequential memory accesses or branch divergence cannot be avoided for a particular formulation. At the same time it is reminded that one of the purposes of this work is to obtain a solver compatible with the widest possible range of devices. Thus, general optimization is preferred over architecture-specific optimization (e.g. considering warps/wave fronts sizes). Kernel profiling can be done using the tools provided by specific OpenCL implementations, e.g. CodeXL for AMD. Alongside with third-party tools, the OpenCL API itself offers the possibility to directly profiling kernel executions, providing execution times. Basically, in the host code, when asking for kernel execution, the call to clEnqueueNDRangeKernel showed in 4.4.3 can be bound to an OpenCL event that is used to measure the execution time. Furthermore, for CPU executions it is possible to enforce the use of a single thread (thus a single CPU core) on both AMD and Intel platforms. This is useful to analyze how the solver scales when using multiple CPU cores (see 6).

5.3 Algorithms and formulations implementation

Here, some of the kernel code parts related to the most important solver components are described. This is done in order to show how the numerical formulations are mapped to the GPU architecture and OpenCL concepts described in 4. Obviously it is not

possible to discuss all the details of all the implemented strategies here in this document. However, some of the concepts behind the code that will be presented here are also directly applied to other kernels that are not showed here. As an example, here the implementation of the Roe fluxes is briefly described. However, the same concepts regarding how the algorithm is mapped on the GPU architecture can be directly applied also to AUSM+ and CUSP convective fluxes. The same idea regards SA and SST turbulence models. Turbulence models consist in additional equations. Thus, what can be said for the single SA equation can be also extended to the two SST equations.

5.3.1 Local Time Stepping and computationally similar kernels

The first kernel executed is related to the pseudo time step computation. The code is quite simple and the formulation is explained in 2.1.10. In listing 5.7 the LTS code is showed.

Listing 5.7: *Local Time Stepping kernel*

```
__kernel void computeDeltaT
(
    real gamma,
    real CFL,
    __global real* dx,
    __global real* Wr,
    __global real* Wx, __global real* Wy, __global real* Wz,
    __global real* We,
    __global real* mu, __global real* muTur, __global real* magV,
    real Pr, real PrTur,
    __global real* dt
)
{
    // Extract work-item ID
    int k = get_global_id(0);

    // Extract cell's size from global memory buffer and store it as private
        variable
    real tdx = dx[k];

    // Read the solution from global memory buffers and store it into private
        memory
    real rho  = Wr[k];
    real Ux   = Wx[k]/rho;
    real Uy   = Wy[k]/rho;
    real Uz   = Wz[k]/rho;

    // Perform the required LTS computations
    real magU = sqrt( Ux*Ux + Uy*Uy + Uz*Uz );
    real p    = ( gamma - 1.0 )*( We[k] - 0.5*rho*magU*magU );
    real c    = sqrt( gamma*p/rho );

    real SRc  = max( magU, magV[k] ) + c;
    real SRv  = gamma/rho*( mu[k]/Pr + muTur[k]/PrTur )/( tdx + VSMALL );

    // Finally write pseudo-time step value back to global memory buffer
    dt[k] = CFL*tdx/( SRc + SRv );
};
```

First of all the kernel is declared alongside its arguments. Then the work-item ID (k) is returned by the get_global_id(0) function. Basically a one-dimensional array (NDRange) of work-items is generated with one work-item assigned to one domain cell (k) in order to compute the pseudo time step of that cell. The next lines are used to

read data from buffers for what concerns the solution of the previous iteration (or the initial guess if this is the first iteration). dx represents a measure of the size of the cell. In this work dx is computed as the ratio between the cell volume and the cell surface, although different strategies can be employed, e.g. cubic root of the volume. The read variables are used to compute the quantities SRc and SRv, as explained in 2.1.10. It's evident here that within this kernel there are few floating point operations with respect to the total number of global memory write and read operations. Considering that accessing global memory requires $O(100)$ clock cycles it is evident that this kernel will not achieve huge speed-ups. Nonetheless, it is easy to see that index k is used in every memory access, meaning that no addressing is necessary and that memory accessed sequentially. No branch divergence is possible in this kernel as there are no branches. From the host point of view the kernel is enqueued specifying an NDRange size equal to the total number of cells, i.e. Nv. The host code employed to enqueue the kernel execution is the following:

Listing 5.8: *Local Time Stepping Host Code*

```
runKernel( computeDeltaT, Nv, queue );
# include "debugger.C
```

It is reminded that, as explained in 4.4.3, runKernel() is not the standard OpenCL API but just a C wrapper macro adopted in this work that ease the syntax when enqueing simple 1D NDRanges. It is possible to see that, right after the kernel launch, the code to perform debugging is placed but executed only if specifically requested by the user. This is done to avoid useless overhead due to continuous data transfer between device and host to check for NaN/Inf values. The debugging strategy is explained in 5.2.4.

AeroX has different kernels that exhibit about the same memory access pattern and about the same ratio between memory access and floating point computations here described. As an example, the kernel that updates the molecular viscosity based on the temperature (Sutherland), as explained in 2.1.1, has basically the same structure: no branch divergence, no addressing of any kind but few floating point operations (with respect to global memory accesses). Furthermore data is read sequentially from global memory. The kernel that computes terms for unsteady simulations through DTS strategy has again the same structure.

5.3.2 Convective Fluxes for internal faces

Here, the focus is on convective fluxes computation. Only the most important aspects of the implementation of the Roe fluxes will be presented. From the computational point of view the same concepts can be directly applied to the implementation of AUSM+ and CUSP fluxes. AeroX is capable to handle hybrid unstructured grids. Since with unstructured grids there is no simple index-based way to access data in memory, no perfect sequential memory access (see 4.3.3) is possible for this kind of kernel. However, depending on mesh entities ordering and the underlying hardware architecture [7], it is still possible to coalesce some memory accesses. Perfect coalesced memory access is lost due to the fact that the kernel is executed in one instance for each internal face (i.e. assigning one work-item to one internal face) while accessing data of the 4 neighbor cell. Each face has 2 direct neighbor cells. However, in order to achieve second order spatial discretization through high resolution strategy, two other cells are required. This

allows to obtain an extended cell set [136]. It must be noted that when employing the 4 cells strategy a proper pre-processing algorithm is required. During pre-processing, for each face, the 2 extended cells have to be found. This algorithm is computationally expensive and not well suited for the GPU architecture. Thus, it is performed during the pre-processing stage on CPU. It must be noted however that the algorithm is performed only once since the extended cells IDs are kept constant for each face during the entire simulation, even when mesh deformation is employed. Furthermore, this process is parallelized on CPU using OpenMP since this stage can take few seconds when the mesh is composed by millions of cells.

The host code adopted to enqueue the kernel execution on the device is here showed:

Listing 5.9: *Internal faces convective fluxes kernel enqueuing*

```
// Convective fluxes kernel enqueue
runKernel( makeInternalFluxesInviscid, Nfi, queue );
# include "debugger.C"
```

the idea is basically to assign each of the Nfi internal faces to each work-item. In a typical mesh for industrial or academic applications, usually millions of faces are employed. This means that each GPU core (Processing Element) will be handling thousands of work-items. The switch between the different formulations, i.e. Roe, AUSM+, CUSP is handled through pre-processor directives inside kernel code in order to avoid the use of conditional statements. Kernels related to convective fluxes exhibit very high number of floating point operations to global memory accesses ratio. This is why, as showed in 6.2.2 by profiling these kind of kernels, it is possible to achieve the highest speed-ups. Since convective fluxes kernels contain numerous floating point operations it is important to check for possible numerical problems, especially when employing SP. Again, after the kernel execution, the user has the possibility to call the debug code to investigate possible numerical issues. In fact, during the development of AeroX, convective fluxes kernel code appeared to be one of the most susceptible of numerical problems.

Given the way OpenFOAM handle the mesh, each face features an "owner" and a "neighbor" cell, basically the cell on one side of the face and the cell on the other side. Furthermore, in this work also "extended owner" and "extended neighbor" are defined to achieve second order through high resolution. These buffers of integer type basically provide the addressing required to read from memory the cell solution of the previous explicit iteration in order to compute the fluxes. Other buffers that contains quantities that are directly associated to the internal face, like unit vectors and face area and ALE velocity, are instead directly accessed with the face index. This means that when investigating a case with an unstructured mesh, due to the aforementioned addressing, there is a performance loss due to non-sequential memory accesses. It is possible to improve performances by using the renumberMesh tool provided by OpenFOAM to reorder mesh entities indexes. Anyway, as previously mentioned, the purpose of this work is to obtain a general purpose solver capable to handle the widest possible mesh formats. This means that in order to be compatible with all kind of unstructured meshes a performance loss is inevitable. It must be noted, however, that despite the clock cycles lost due to non-coalesced memory accesses, the convective fluxes kernel exhibits an high number of heavy floating point operations, such as trascendental functions. Thus, the performance loss due to memory access is partially masked by the high number of

floating point operations and scheduled work-items. Furthermore it must be noted that modern hardware is usually capable to coalesce some memory accesses even when no perfect sequential accesses are employed [7]. Thus, mesh renumbering usually provide advantages thanks to the fact that faces with consecutive indexes are usually related to cells with near indexes. Listing 5.10 shows the first stage of the convective fluxes kernel, where data stored in global memory is read from buffers and temporarily saved in work-item private memory.

Listing 5.10: *Input stage of convective fluxes kernel*

```
// Work-item ID extraction
int i = get_global_id(0); // Interface label

// Addressing face -> cells
int id_L  = owner[i];     // "owner" cell index
int id_R  = neighbour[i]; // "neighour" cell index
int id_LL = owner[i];     // "extended owner" cell index
int id_RR = neighbour[i]; // "extended neighbour" cell index

// Face-related quantities
real Sf    = S[i]; // face area
real n[3]  = { nx[i], ny[i], nz[i] }; // face unitary normal vector
real t[3]  = { tx[i], ty[i], tz[i] }; // face unitary tangent vector
real b[3]  = { bx[i], by[i], bz[i] }; // face unitary vector to complete the tern
real Cf[3] = { CCfx[i], CCfy[i], CCfz[i] }; // face center
real Vf[3] = { ALEx[i], ALEy[i], ALEz[i] }; // ALE velocity

// Cells-related quantities
real C_LL[3] = { CCcx[id_LL], CCcy[id_LL], CCcz[id_LL] }; // ext. owner cell center
real C_L[3]  = { CCcx[id_L],  CCcy[id_L],  CCcz[id_L] };  // owner cell center
real C_R[3]  = { CCcx[id_R],  CCcy[id_R],  CCcz[id_R] };  // neighbour cell center
real C_RR[3] = { CCcx[id_RR], CCcy[id_RR], CCcz[id_RR] }; // ext. neigh. cell
    center

// Solution of the previous iteration
// ext. owner solution
real W_LL[5] = { Wr[id_LL], Wx[id_LL], Wy[id_LL], Wz[id_LL], We[id_LL] };
// owner solution
real W_L[5]  = { Wr[id_L ], Wx[id_L ], Wy[id_L ], Wz[id_L ], We[id_L ] };
// neighbour solution
real W_R[5]  = { Wr[id_R ], Wx[id_R ], Wy[id_R ], Wz[id_R ], We[id_R ] };
// ext. neighbour solution
real W_RR[5] = { Wr[id_RR], Wx[id_RR], Wy[id_RR], Wz[id_RR], We[id_RR] };
```

It is understood that here the __kernel keyword with all the arguments is not showed for simplicity. The work-item ID is returned by the usual get_global_id(0) function and stored in the private memory variable i which represents the absolute internal face index. It is reminded that internal faces indexes in OpenFOAM come before boundary faced IDs. The index is then used to access the addressing buffers in order to obtain the four cells indexes. These indexes are subsequently used to get cell data, like cells centers and obviously the solution of the previous explicit iteration. The second stage is the actual call to the function that contains the Roe algorithm: ,

Listing 5.11: *Convective fuxes algorithm selection*

```
// Fluxes initialization
real F_LR[5] = { 0.0, 0.0, 0.0, 0.0, 0.0 };

#if FLUXES == ROE
// Call Roe algorithm
RoeCenteredFlux( gamma, n, t, b, Vf, W_L, W_R, W_LL, W_RR, F_LR);
```

```
#elif FLUXES == AUSM
// Call AUSM+ algorithm
#else
// Call CUSP algorithm
#endif
```

Basically the data read from memory is passed to the user-selected fluxes function thanks to aforementioned mechanism that employs pre-processor directives and run-time compilation to avoid the use of C conditionals. Here the F_LR function argument represents the output while the other arguments represent the inputs. Finally, in listing 5.12 the final stage of the kernel is showed: here the work-item private array is finally stored into global memory buffers in order to be available to subsequent kernels, like residual assembly.

Listing 5.12: *Storing convective fluxes into global memory*

```
// Write fluxes in buffers
Fr[i] = F_LR[0]*Sf;
Fx[i] = F_LR[1]*Sf;
Fy[i] = F_LR[2]*Sf;
Fz[i] = F_LR[3]*Sf;
Fe[i] = F_LR[4]*Sf;
```

Now it's time to discuss what happens when the Roe fluxes function is called. Inside the function there are no global memory or local memory operations: all memory accesses are work-item private. This means that all the work-items can proceed with their work concurrently. This is basically the heaviest algorithm within AeroX from a floating point operation count. For these reasons and since there is nothing particularly interesting to be noted from a computational point of view (e.g. branch divergence or coalesced memory accesses), it is not convenient to show here all the lines of code. However it is easy to understand that such kind of heavy and parallel computations are well suited for the massively parallel GPU architecture. This is why, as said before, high speed-ups are obtained by this kernel, as benchmarks show in 6.2.2. It is worth to briefly introduce how the Roe algorithm is handled in 3D alongside with the high-resolution limiter and the necessary entropy fix. The strategy is discussed in details in [136]. Here the most important steps of the Roe algorithm are introduced:

1. The solution stored as conservative variables is locally adjusted in order to take into account the material velocity through ALE formulation (2.1.3);
2. The adjusted solution is then projected from the global cartesian reference frame to a face-local reference frame, where the face normal n represents the first direction. This allows to employ the 1D Roe formulation on a 3D problem;
3. From the solution defined in the local reference of frame, different quantities, useful for the Roe algorithm, are computed, such as the Roe's average state;
4. Eigenvalues and eigenvectors are computed using the information provided by the owner and the neighbor cells;
5. Eigenvalues are opportunely modified using the Harten and Hyman entropy fix in order to avoid non-physical solutions;
6. 2nd order centered fluxes are computed using the solution of the owner and neighbor cells;

7. The variable jumps are computed: between the owner and neighbor, between the extended owner and the owner and between the extended neighbor and the neighbor;
8. The jumps are processed by the Van Leer flux limiter;
9. The Roe 1st order upwind fluxes are assembled;
10. Centered and upwind fluxes components are added together;
11. Fluxes are adjusted to take into account ALE formulation;
12. Fluxes are transformed from the face-local point of view to the global reference of frame;

This algorithm eases the application of the flux limiter, thanks to the 4 cells strategy, to achieve high resolution. However, it could be also possible to reduce the total computational costs by directly implementing the final expressions of the Roe's fluxes, as explained in [48]. This would reduce the number of floating point operations required to perform the different steps as independent stages. The key difference is that using such a strategy it is not possible to directly apply the 1D concepts to achieve high resolution. However, high resolution could be implemented through a slope limiter strategy by considering opportunely modified right and left states obtained from the interpolation of the solutions of the four cells. The interpolation algorithm itself however requires non-negligible computational effort, that is instead not required using the previously described strategy. Thus, on one side the first strategy requires high computational effort due to different explicit transformations to allow the use of the 1D formulations. On the other side, with the interpolation strategy, the additional effort is related to the interpolation algorithm. From the point of view of results accuracy the two strategies are equivalent. However from tests it seems that slightly more numerical robustness is obtained with the first strategy in SP executions.

It must be noted that when performing RANS simulations with SA or SST turbulence models, the convective fluxes related to the additional turbulence models equations are computed in the same kernel, alongside Roe's fluxes. Listing 5.13 shows the bulk of kernel code for RANS convective fluxes computation. The structure of the code is identical to what previously presented.

Listing 5.13: *RANS convective fluxes*

```
#if TURBULENCE == SA
// SA convective flux initialization
real Fn = 0.0;

// SA solution is read
real nuTilda_L  = nuTilda[id_L];
real nuTilda_R  = nuTilda[id_R];

// SA convective flux computation
SA_advection( n, Vf, W_L, W_R, nuTilda_L, nuTilda_R, &Fn );

// SA convective flux is stored to global memory
FnuTilda[i] = Fn*Sf;
#elif TURBULENCE == KW
// SST convective fluxes initialization
real Fk = 0.0;
```

```
real Fw = 0.0;

// SST solution is read
real kTur_L  = kTur[id_L];
real kTur_R  = kTur[id_R];
real wTur_L  = wTur[id_L];
real wTur_R  = wTur[id_R];

// SST convective fluxes computation
KW_advection( n, Vf, W_L, W_R, kTur_L, kTur_R, wTur_L, wTur_R, &Fk, &Fw );

// SST convective fluxes are stored to global memory
FkTur[i] = Fk*Sf;
FwTur[i] = Fw*Sf;
#endif
```

It must be noted that in this work convective fluxes are computed and then saved in fluxes buffers instead of being directly employed to update residuals buffer. This is the same strategy adopted in [66]. The main reason behind this choice is given by the GPU architecture and OpenCL specifications. In fact, the more intuitive way to approach the problem would be to execute a convective fluxes kernel and directly write the internal face fluxes inside the two related cells (owner and neighbor) residuals. However this is not directly possible since for the same cell residuals stored in global memory multiple work-items from different work-groups would write in the same global memory locations concurrently. Another strategy would be to execute a kernel assigning one work-item to one domain cell. Each work-item would loop over the faces, compute the fluxes and then directly update the cell residuals. However, the same face fluxes would be computed two times, wasting computational time. Furthermore, with hybrid meshes this would lead to branch divergence, since different work-items would loop on different number of faces.

5.3.3 Wall treatment

Here the attention is focused on the most important computational aspects regarding the kernel code for automatic wall treatment described in 2.1.9. This is useful only for viscous simulations. In particular, the value of u_τ is required over wall faces in order to compute $\tau = -\rho u_\tau^2$ used to compute wall viscous fluxes. As will be presented in 5.3.4, convective and viscous fluxes over boundary faces are computed inside boundary conditions kernels. However, those kernels are executed once the input arguments are ready. The idea is to allocate a buffer, uTau, to be used inside boundary conditions and in other kernels that requires wall data. Considering the wall face k assigned to work-item k, the uTau[k] value of the previous pseudo time iteration is used as input argument to the wallFunction() function through the private variable uTauPriv. The value returned by the function is then stored in uTau[k]. It worth to discuss what happens inside wallFunction() for what concerns computational aspects. This is showed in 5.14:

Listing 5.14: *wallFunction function*

```
real wallFunction( real y, real dU, real rho, real mu, real uTauPriv )
{
    // Set the same number of iteration for each work-item and perform the
    // Newton-Rapson loop
    int maxIter = 20;
```

```
    for ( int iter = 0; iter < maxIter; iter++ )
    {
        real f      = dU/uTauPriv - uPlus( y, rho, uTauPriv, mu );
        real df_du  = dU/(uTauPriv*uTauPriv) + duPlus_dyPlus( y, rho, uTauPriv, mu
            )*y*rho/( mu + SMALL );
        real duTau  = f/( df_du + SMALL );
        uTauPriv   += duTau;
    }
    return uTauPriv;
}
```

The idea is to compute the value of u_τ using an iterative procedure based on the Newton–Raphson algorithm. As explained in 2.1.9 the idea is to implement a particular wall treatment that allows the solver to automatically switch between the viscous sublayer and the log layer formulations on the basis of geometric and flow conditions. The switch is not performed using a step function but through a smooth function that weights the two contributions. The wallFunction() function is executed by each work-item, each one for its particular wall face. The function showed in 5.14 is called from a kernel that basically reads global memory data, passes it to the wallFunction() function and finally writes results in uTau buffer at location k. However, inside the function all the memory accesses are performed with private memory variables. Due to the Newton–Raphson loop and the called functions numerous floating point operations are performed inside wallFunction(). duPlus_dyPlus() and uPlus() functions are called to evaluate and blend the contributions given by viscous sublayer and log region but are not here discussed as they are not important from the GPGPU point of view. The Newton–Raphson loop is performed 20 times equally for each work-item. This all means that an high floating point operations to global memory accesses ratio is achieved inside the kernel. The Newton–Raphson for loop is repeated 20 times without any convergence check in order to avoid possible branch divergence due to different work-items reaching convergence with different number of iterations. 20 iterations are enough for every possible situation, especially during the first iterations. It is also reminded that the value of uTau from the previous pseudo time iteration is used as a guess solution for the Newton–Raphson procedure, reducing possible convergence problems. As it is possible to see, SMALL values are adopted to avoid floating point issues, especially with SP. It is reminded also that these 20 iterations are performed for each wall face inside a single pseudo time iteration. Thus, if for some reasons 20 iterations are not enough inside a single pseudo time iteration, further pseudo time iterations (and thus Newton–Raphson iterations) are anyway performed to reach convergence. This is another advantage provided by the use of the pseudo time concept.

5.3.4 Boundary conditions

Here, the implementation of boundary conditions is briefly presented. As explained in 2.1.8, different kinds of boundary conditions are implemented in AeroX. Here the focus is just on computational aspects. The adopted strategy for boundary conditions is to write one kernel for each type of boundary condition. From the point of view of execution and memory accesses all the kernels related to boundary conditions are very similar. With OpenFOAM, boundary faces of the same patch have consecutive IDs. The idea here is to execute a kernel for each patch and assign one work-item to one boundary (patch) face. The host job is to loop over all patches and enqueue for execution the correct kernel based on the physical properties of the boundary. The physical

type of each patch is specified by the user in the constant file of the OpenFOAM case files hierarchy. Listing 5.15 shows the simplified host code that performs this operation:

Listing 5.15: *Host code for boundary conditions handling*

```
// loop over all patches
forAll( mesh.boundaryMesh(), iPatch )
{
    // extract physical type from OpenFOAM dictionary
    word physicalType = mesh.boundaryMesh().physicalTypes()[iPatch];

    // extract first face ID and total number of faces
    label startFace = mesh.boundaryMesh()[iPatch].start();
    label nFaces    = mesh.boundaryMesh()[iPatch].faceAreas().size();

    // ... if/else statements to choose the correct kernel
    else if( physicalType == "exampleBC" )
    {
        // variable kernel arguments setting
        setKernel( exampleBC, 0, startFace );

        // exampleBC kernel enqueueing
        runKernel( exampleBC, nFaces, queue );
        # include "debugger.C"
    }
    // ... if/else statements to choose the correct kernel
}
```

It is possible to see that the mesh object provided by the OpenFOAM API is used to loop over the patches specified in the boundary dictionary. This dictionary is then used in the loop to extract the physical type of the patch (e.g. wall, periodic, characteristic-based,...), the first face ID of the patch (startFace) and the total number of faces. The physical type of the patch is used to choose the correct kernel to be executed using an if/else statements list (this could be also done using a switch statement). The total number of faces (nFaces) is directly used to set the total number of work-items composing the NDRange. The startFace integer is passed as an argument to the kernel scheduled for execution using the setKernel wrapper macro that calls OpenCL API under the hood. The reason to pass startFace to the exampleBC kernel will be explained shortly.

Another strategy to handle boundary conditions would be to execute one single kernel for each of the Nfb faces and then handle the different kinds of patches using conditional statements inside the kernel. However, this would lead to different problems. In fact this would mean that conditional statements would be required inside kernel code to opportunely switch between numerical formulations based on face ID. Furthermore different boundary conditions require different data stored in global memory. This would mean that, in different kernel paths, accesses to different global memory buffers would be required, leading to branch divergence and non-coalesced memory accesses. The only drawback of the implemented strategy is that usually the number of faces that compose a patch is two or three orders of magnitude smaller that the total number of domain cells Nv and internal faces Nfi. As an example, the mesh for the DPW2 case (see 7.3) contains $2 \cdot 10^6$ cells, $4.5 \cdot 10^6$ internal faces, $1 \cdot 10^3$ faces for the symmetry plane, $5 \cdot 10^2$ far-field faces and $8 \cdot 10^4$ faces for the aircraft surface. Consider that nowadays a mid-range gaming GPU has around $2 \cdot 10^3$ cores it is easy to see that in some cases the total number of work-items is smaller that the total number of available cores. It must be noted however that kernels that execute boundary conditions are

heavy for what concerns floating point operations as they have to compute both convective and viscous fluxes. Thus, even if the total number of work-items is in the order of the total number of GPU cores, computational efficiency is not particularly affected when considering the overall execution.

Here the skeleton of a boundary condition kernel is briefly presented. The focus is again on the computational aspects rather than the numerical formulations. Listing 5.16 represents an example of a boundary condition kernel:

Listing 5.16: *Kernel code for boundary conditions*

```
__kernel void exampleBC
(
    int startFace,
    // other arguments
)
{
    // Extract the global face ID and
    int ii = get_global_id(0);
    int i = ii + startFace;

    // Load face-cell connectivity and metrics
    loadConnectivityMetrics();

    // Load the solution
    real rho_B = Wr[id_L];
    real mx_B  = Wx[id_L];
    real my_B  = Wy[id_L];
    real mz_B  = Wz[id_L];
    real Et_B  = We[id_L];

    // Build the ghost cell solution
    real rho_B, mx_B, my_B, mz_B, Et_B;
    buildGhostCell();

    // Build 4-cells solution arrays
    real W_LL[5] = { Wr[id_LL], Wx[id_LL], Wy[id_LL], Wz[id_LL], We[id_LL] };
    real W_L[5]  = { Wr[id_L ], Wx[id_L ], Wy[id_L ], Wz[id_L ], We[id_L ] };
    real W_R[5]  = { rho_B, mx_B, my_B, mz_B, Et_B };
    real W_RR[5] = { rho_B, mx_B, my_B, mz_B, Et_B };

    // Save boundary values for gradients
    updateBoundaryValues();

    // Call convective fluxes aglorithm
    #if FLUXES == ROE
    RoeCenteredFlux( gamma, n, t, b, Vf, W_L, W_R, W_LL, W_RR, F_LR);
    #elif FLUXES == CUSP
    // call CUSP fluxes
    #else
    // call AUSM fluxes
    #endif

    // Save convective fluxes
    Fr[i] = F_LR[0]*Sf;
    Fx[i] = F_LR[1]*Sf;
    Fy[i] = F_LR[2]*Sf;
    Fz[i] = F_LR[3]*Sf;
    Fe[i] = F_LR[4]*Sf;

    // Perform the same operation for viscous fluxes
    viscousFluxes();

    // Perform the same operation for turbulence models
    turbulence();
}
```

At the beginning, global memory is accessed using loadConnectivityMetrics() macro to extract geometrical and connectivity data for the current face and its owner cell (see 5.1.3). The startFace value is here used to extract the global face ID, ii, from the patch-local face ID, i, (i.e. the work-item ID). The global ID is used to access the faces owner cells array, obtain the owner cell ID (id_L) and access all the cell required data (solution and geometrical properties). This is also done for the extended cell (ID given by id_LL) for high resolution. Knowing the solution and the physical type of the patch it is possible to build the ghost cell solution. For simplicity the code related to viscous computations is not showed. However, the work flow, from a computational point of view, is similar to what seen when discussing convective fluxes. Finally, fluxes are stored in global memory using the global face ID, ii, in order to be consistent with the final residual assembly (see 5.3.6). The very last step is related to turbulence models since they require convective and viscous fluxes computation as well.

5.3.5 Viscous fluxes

It is important to introduce the fundamental aspects related to viscous fluxes computation. The computation of the viscous fluxes themselves is somehow similar to the previously discussed convective fluxes 5.3.2. In fact, the idea is again to assign one work-item to one internal face, use the addressing buffers to access the neighbor and owner cells data, compute the fluxes through a dedicate algorithm and finally store the fluxes into global memory to be used for later residual assembly 5.3.6. As with convective fluxes, this is done not only for the Navier–Stokes equations but also for the turbulence models equations related to SA and SST turbulence models. From the point of view of the total number of floating point operations, viscous fluxes require less computational effort than convective fluxes, however they require more global memory accesses. Thus the computational efficiency is lower, as showed in 6.2.2. Despite the fact that accessing extended cells solution is not required for viscous fluxes, the gradients of velocity and temperature are required. Since velocity is a vector field, its gradient is a tensor field. Thus 9 float/double have to be read for both the owner and neighbor cells for each work-item i computing viscous fluxes of face i. The same is valid for the 3 scalars related to the temperature gradient of each cell. Furthermore, when performing RANS simulations, 3 scalars are also read from global memory for what concerns $\tilde{\nu}$ gradient for SA. 6 scalars, for ω and k gradients, are instead required for SST. Alongside these variables, other variables have to be read from global memory within the viscous fluxes kernels. These are represented by solution variables of the previous iteration, geometrical faces data, molecular viscosity μ (since by using the Sutherland model each cell has its own independent value based on local temperature), turbulent viscosity (due to Bussinesq hypothesis). These quantities are read from global memory for both owner and neighbor cells related to each face (work-item).

As explained in 2.1.6 the Gauss formulation is here adopted to compute the required gradients on cells. The Gauss formulation represents a good trade-off between results accuracy and computational efficiency, especially for GPU executions. In fact, another possibility would be the use of a least squares approach. This was successfully adopted for a GPU accelerated full potential explicit solver [66]. The least squares strategy, however, would be quite expensive from a computational point of view in this work since for each variable of each domain cell the solution of a linear system is required

and, as explained before, the gradients of velocity, temperature and turbulence models variables are required. The main problem, however, is represented by GPU executions with hybrid meshes, where different cells could have different numbers of neighbors, thus different system sizes, thus different computational effort for each GPU core, thus branch divergence. Non-coalesced memory accesses could be also a problem with unstructured meshes. Furthermore, especially for SP executions, a least square approach could lead to numerical problems when solving the linear system with particular cell centers alignments. The Gauss algorithm here adopted still suffers from a slight efficiency loss when employing hybrid meshes. During gradients assembly, using expression 2.1.6 the idea is in fact to execute a kernel of Nv work-items, one for each cell. Since the assembly of the cell gradient is performed with a loop over the cell faces which can differ in number between different cells, different work-items could need to perform a different numbers of operations. Here, the bulk of the code for the Gauss algorithm is presented. In order to compute the k-th cell gradient for a variable X, the values of X over the faces of k are required. Thus, the first stage of the algorithm is the interpolation of the cell values over the faces. Boundary faces values are computed separately, inside boundary conditions kernels, where the solution have to be computed anyway. Listing 5.17 shows the kernel code for the interpolation of solution values over the internal faces:

Listing 5.17: *Internal faces value interpolation*

```
__kernel void gradientsInternalFaces
(
// arguments
)
{
    int i = get_global_id(0);

    // For each internal face, find the ID of the two cells (owner and neigh.)
    int id_L = localOwner[i];
    int id_R = localNeighbour[i];

    // Extract the interpolation wheights for the two cells of face i
    real w_L = linearInterpolate[i];
    real w_R = 1.0 - w_L;

    // Interpolate the solution first (conservative variable)
    real rho_I = w_L*Wr[id_L] + w_R*Wr[id_R];
    real mx_I  = w_L*Wx[id_L] + w_R*Wx[id_R];
    real my_I  = w_L*Wy[id_L] + w_R*Wy[id_R];
    real mz_I  = w_L*Wz[id_L] + w_R*Wz[id_R];
    real Et_I  = w_L*We[id_L] + w_R*We[id_R];

    // From the interpolated solution, find the face value of U,T and turb.
        quantities
    Uix[i] = mx_I/rho_I;
    Uiy[i] = my_I/rho_I;
    Uiz[i] = mz_I/rho_I;
    Ti[i]  = ( Et_I - 0.5*( mx_I*mx_I + my_I*my_I + mz_I*mz_I )/rho_I )/( rho_I*Cv
        );
#   if TURBULENCE == SA
    NuTildai[i] = w_L*NuTilda[id_L] + w_R*NuTilda[id_R];
#   elif TURBULENCE == KW
    KTuri[i] = w_L*KTur[id_L] + w_R*KTur[id_R];
    WTuri[i] = w_L*WTur[id_L] + w_R*WTur[id_R];
#   endif
}
```

One work-item is assigned to each internal face of the domain for a total of Nfi faces. Thus the host code for kernel enqueuing looks similar to 5.9, except for the different kernel name. It is worth to explain few aspects of this kernel. First of all the work-item ID, thus the internal face index, is obtained as usual. Then the owner and neighbor cells ID are obtained from the addressing buffers. A buffer, linearInterpolate, stored in global memory, is used to store the weights for the two cells related to the face i. Obviously the same cell features different interpolation weights depending on the considered face. In order to be consistent and to reduce memory requirements, just the owner weight is stored, the neighbor weight is clearly computed as the complementary to 1.0. It must be noted that different strategies could be adopted to compute the weights. In any case weights are computed once during pre-processing by the host code on CPU. In this work the weights are directly obtained by the OpenFOAM implementation. In any case it is possible to see that in this kernel the solution is accessed through addressing, thus with a potential performance loss due to non-coalesced accesses. Another aspect regards the interpolation. Here the strategy is to firstly perform the interpolation on conservative variables and then convert the conservative variables into the required variables for gradients, i.e. temperature and velocity. Another possibility would be to firstly convert conservative variables to velocity and temperature and then perform the interpolation. No particular differences in results were obtained with this second strategy. However, this second strategy requires more floating point operations since the computations of velocity and temperature need to be performed twice, one for the owner cell and one for the neighbor cell. The last lines of the kernel are related to the final storing of the interpolated values to buffers in global memory. Here, the results of the conversion from conservative variables to velocity and temperature are directly stored without the necessity of intermediate work-item private memory variables. Finally, for what concerns turbulence, no particular floating point operations are required, since equations just require the gradients of the solution itself. This is true both for SA and SST.

The last stage of the gradients computation is the use of the Gauss algorithm. At this point buffers Uix, Uiy, Uiz, T (and eventually NuTildai, KTuri and WTuri) contain internal and boundary faces values. Thus, the final stage kernel in gradients assembly is represented by listing 5.18:

Listing 5.18: *Gradient assembly through Gauss algorithm*

```
__kernel void assemblyGradient
(
// arguments
)
{
    int k = get_global_id(0);

    // Variables initialization
    real tTx  = 0.0;
    real tTy  = 0.0;
    real tTz  = 0.0;
    // same with velocity variables
    // same with turbulence model variables

    // extract total number of face of cell k
    int numFaces = cellFacesDelimiter[k+1] - cellFacesDelimiter[k];
    int startFace_idx = cellFacesDelimiter[k];
```

```
    for( int i = 0; i < numFaces; i++ )
    {
        // extract f-th face index of cell k
        int ii  = cellFaces[startFace_idx + i];

        // extract "signum" of face k
        int tsgn = signum[startFace_idx + i];

        // build face area vector
        real tS   = tsgn*S[ii];
        real Sx   = tS*nx[ii];
        real Sy   = tS*ny[ii];
        real Sz   = tS*nz[ii];

        // extract face value of variable
        real tTi  = Ti[ii];
        // same with velocity variables
        // same with turbulence model variables

        // integrate over cell boundaries, Gauss formula
        tTx  += ( tTi *Sx );
        tTy  += ( tTi *Sy );
        tTz  += ( tTi *Sz );
        // same with velocity variables
        // same with turbulence model variables
    }

    real wn = RELAX_GRAD; // user-defined variable
    real wo = 1.0 - wn;

    // extract cell volume to complete Gauss formula
    real tVV = VV[k];

    // update gradient values through relaxation
    gradT_x[k]  = gradT_x[k]*wo  + tTx /tVV*wn;
    gradT_y[k]  = gradT_y[k]*wo  + tTy /tVV*wn;
    gradT_z[k]  = gradT_z[k]*wo  + tTz /tVV*wn;
    // same with velocity variable
    // same with turbulence model variables
}
```

It is noted that here for simplicity only the lines of code related to the temperature gradient are showed. The computation of the velocity gradient is conceptually identical but from a mathematical point of view a tensor will be produced, which is translated in a matrix from a computational point of view. For what concerns turbulence models variables, again they are conceptually identical to temperature since they all represent scalar fields and their gradients are represented by vector fields. The algorithm and its kernel implementation are straightforward. The idea is to execute Nv instances of the kernel, one for each domain cell, assigning the k-th work-item to the k-th cell. Firstly the gradients are defined as private memory variables and initialized to 0. This is done in order to store the gradients into temporary variables and to update the global memory data only at the end, right after a relaxation procedure. The next step is to extract the total number of faces of the k-th cell. This is done using an integer buffer, cellFacesDelimiter that is filled using the strategy presented in 5.1.3. This buffer is used both to extract the total number of faces surrounding the cell and to find the starting point inside the cellFaces and signum buffers to extract k-th i-face related data. ii is the OpenFOAM global ID of the i-th face of cell k and is used to extract the area of the face (from buffer S) and the unitary vector (from buffers nx, ny, nz) in order to build the face area vector. The signum buffer just contains $+1$ or -1 and is used to correct the direction of the face area vector. This is done in order to reduce memory

consumption since otherwise it would be necessary to store the unit vector for each face twice, once for each cell sharing the face. This could also be performed by checking if the k-th cell is the owner or the neighbor of the i-th cell. However the implemented strategy is slightly more efficient since the other proposed strategy would require the use of conditional statements and thus would lead to branch divergence issues. After the extraction of the temperature value over the face, the contribution is accumulated in the private variable previously defined. This way it is possible to proceed with the numerical integration. Finally, after the loop, the result is divided by the volume to complete the Gauss algorithm and obtain the gradient reconstruction. However, the result is not directly used to store the gradient in global memory. A relaxation strategy is instead performed using a user-defined variable RELAX_GRAD. Gradients relaxation is another strategy employed to help convergence. It can be applied thanks to the fact that a single pseudo time iterations has no physical meaning. Otherwise, with an unsteady global time stepping strategy, this would not be possible since full gradients would be required. Gradients relaxation helps convergence especially during the first iterations.

Due to the aforementioned addressing and the quite low floating point computations to global memory accesses ratio, the kernels related to viscous fluxes exhibit less computational efficiency with respect to the convective fluxes kernels. In order to try to tackle this problem, all the kernels related to viscous fluxes are computed every N explicit iterations, while convective fluxes are computed within each explicit iteration. N is a user-defined parameter. The key idea here is to try to find a good balance between convergence rate and computational effort. From the tests performed in this work it seems that usually using N = 5 do not afflict convergence rate too much, while it allows to reduce the time/iteration/cell value (see 6.2.1). Obviously the advantages vary from case to case. Anyway the possibility to execute all the kernels every explicit iteration is still possible. Listing 5.19 better explains this strategy with pseudo host code. Recalling the pseudo host code previously showed to explain the convergence check strategy 5.2, here inside the aerodynamicIterations() function the following code is present:

Listing 5.19: *Viscous fluxes trade-off strategy*

```
// Convective fluxes host code and kernel launch for internal faces
internalConvectiveFluxes();

if(totalIter % N == 0)
{
    // Gradients and viscous fluxes related host code and kernels launch for
        internal faces
    computeGradients();
    internalViscousFluxes();
}

// Sum inviscid and visocus fluxes
sumInviscidViscousFluxes();
```

Since viscous fluxes have to be kept constant for N iterations while convective fluxes are computed every iteration, convective and viscous fluxes are stored in independent buffers. However, they are added up every explicit iteration, as showed in listing 5.20:

Listing 5.20: *Convective and viscous fluxes sum*

```
__kernel void sumInviscidViscous
```

```
(
// kernel arguments
)
{
    int k = get_global_id(0);

    Fr[k] += Gr[k];
    Fx[k] += Gx[k];
    Fy[k] += Gy[k];
    Fz[k] += Gz[k];
    Fe[k] += Ge[k];
#   if TURBULENCE == SA
    Fn[k] += Gn[k];
#   elif TURBULENCE == KW
    Fk[k] += Gk[k];
    Fw[k] += Gw[k];
#   endif
}
```

Gr[k] is the viscous flux related to density and face k. The other buffers are related to momentum, energy and turbulence models variables. This code is enqueued for execution inside sumInviscidViscousFluxes(). The idea here is to add to convective fluxes the contribution of viscous fluxes. The only floating point operations here are the subtractions. It can be seen that coalesced memory reads from global memory are possible as the index k is directly used to access memory without addressing.

5.3.6 Residual assembly

After the computation of convective and viscous fluxes over internal and boundary faces, they are used to assembly cells residuals. This is done in the following kernel:

Listing 5.21: *Residual assembly*

```
__kernel void assembly
(
// arguments
)
{
    int k = get_global_id(0);

    int numFaces = cellFacesDelimiter[k+1] - cellFacesDelimiter[k];
    int startFace_idx = cellFacesDelimiter[k];

    // initialization
    real tRr = 0.0;
    // same for momentum, energy and turbulence

    // Sum local contribution with convention owner < neighbour
    for( int i = 0; i < numFaces; i++ )
    {
        int face = cellFaces[startFace_idx + i];
        int tsgn = signum[startFace_idx + i];
        tRr -= tsgn * Fr[face];
        // same for momentum, energy and turbulence
    }
    Rr[k] = tRr;
    // same for momentum, energy and turbulence
}
```

Again, for simplicity, only the assembly of the density residual is showed. Momentum, energy and turbulence variables residuals are assembled the same way. The kernel is structured in the same way of the gradient assembly kernel showed in 5.3.5. In fact the

algorithm is very similar: one work-item is assigned to one cell and a loop over the cell faces is performed to add faces contributions to the cell residual. The addressing buffers adopted are the same described when discussing about gradients. The kernel is executed in Nv instances. Due to the compatibility with unstructured meshes, it is not possible to achieve perfect memory coalescing. However, partial coalescing, depending on hardware capabilities and mesh entities numbering is possible. Furthermore, with hybrid meshes branch divergence occurs due to the fact that different cells have to iterate over a different number of surrounding faces. However, as showed in 6.2.3, hybrid meshes do not particularly afflict the time/iteration/cell performance parameter.

5.3.7 Source terms

A single kernel is used to perform the computation of all the source terms. From the computational point of view this kernel is very similar to the previously described kernel for pseudo time step value computation (see 5.3.1). A total number of Nv work-items in the NDRange are assigned to domain cells. Memory related to cells solution is accessed sequentially using the work-item index k. For what concerns Euler and Navier–Stokes equations, the only source term is given by the MRF as explained in 3.6. The bulk of the computations is instead related to turbulence models. The last step performed in this kernel is the application of the point implicit strategy described (see 2.1.7 to improve numerical stability. The point implicit strategy is applied directly here for two reasons: avoid the execution of another kernel that will require again the read of residuals and the fact that this step must be performed before the residuals are processed by convergence acceleration techniques.

5.3.8 Convergence acceleration techniques

Right after the residual assembly and before the solution update, the kernels related to convergence acceleration techniques are executed. This includes Multi-Grid and Residual Smoothing. As explained in 2.1.10 this is done in order to provide and implicit-like convergence ratio to residuals damping. Due to the fact that this is the last step in residuals manipulation, after the convergence acceleration techniques algorithms residuals are opportunely multiplied by the factor $\frac{\Delta\tau}{\Delta V}$ to complete the pseudo time stepping strategy and update the solution. Without going into details for what concerns the convergence acceleration techniques implementation, it is possible to do some considerations related to computational efficiency. The algorithm behind RS here implemented is computationally similar to what previously discussed during gradients and residuals assembly. In fact during the first stage of the algorithm, one work-item is assigned to one domain cell (for a total number of Nv work-items) and a loop over neighbor cells is performed in order to accumulate their residuals. The second and final stage is again performed on each cell and is aimed to modify a fraction of the cell residual with the result of the previous stage. This is done in a consistent manner (i.e. with an opportune normalization based on the number of cell neighbors). This two stages can be repeated multiple times in a trade-off between computational effort and residual smoothing effects. Anyway, like the residual assembly kernel, branch divergence and non-coalesced memory accesses could lead to performance loss with hybrid unstructured meshes. The total number of floating point operations is relatively small (just the sum operations to perform the residuals accumulations) with respect to global memory reads (residuals

of surrounding cells). The MG algorithm is somehow similar. Two stages are required, one for residuals accumulation on each grid level and one to finally update the residuals of each cell. The first stage is composed by multiple kernel executions, one for each MG level. For each cell k of level L, a loop over the cells of the finer mesh level L-1 contained in k is performed. The first level is represented by the real mesh, while coarser levels are obtained in pre-processing using OpenFOAM API and a mesh nodes-based agglomeration strategy written from scratch [127]. MG is very effective in damping residuals. The main drawback from the computational point of view is represented by the high branch divergence effects and non-sequential memory accesses. This is due to the fact that different cells from level L could contain very different numbers of cells of the L-1 level. Since during kernel execution each cell k of level L is assigned to one work-item and it loops over the cells of the L-1 level, it is easy to see that this could be quite expensive from a computational point of view. Furthermore, on a mesh of about $1 \cdot 10^6$ cells about 8 MG levels can be obtained from GAMG algorithm. Usually, a good trade-off between computational effort and residuals damping is obtained with 6-8 levels. It must be noted that different meshes could perform differently depending on how the GAMG algorithm behaves, especially for what concerns the GAMG ability to agglomerate about the same number of cells inside a single cell of the coarser level (in order to reduce the effects of branch divergence). The other problem is represented by non-sequential memory accesses since addressing is required inside each cell to obtain residuals from cells of the finer level. The total number of global memory reads can be reduced if cells agglomerated in the same cell of the coarser level have similar IDs. Renumbering the mesh with the OpenFOAM renumberMesh utility usually slightly speed up the algorithm. Due to the fact that the solver is mainly aimed to handle unstructured meshes it is not possible to directly bypass problems with memory accesses and branch divergence.

5.3.9 Solution update

The update of the solution is the last step to be performed. When residuals are ready, after their assembly and manipulation with convergence acceleration techniques, they can be used to update the solution. Listing 5.22 shows the kernel code used to update the solution:

Listing 5.22: *Solution update*

```
__kernel void update
(
// arguments
)
{
    int k = get_global_id(0);

    // Previous solution is read and store in
    real tW0r = W0r[k];
    // Same for momentum, energy and turbulence

    Wr[k] = tW0r + limit( Rr[k], tW0r,   LIMIT );
    // Same for momentum, energy and turbulence

    // Update effective viscosity
};
```

For simplicity just the code related to density is showed. However the same code can be applied to momentum, energy and turbulence models variables. The idea here is to update the solution after the residuals are processed by the limiter strategy explained in 5.2.3 to help convergence during first iterations. It is important to notice that here, once turbulence models variables are updated, it is possible to update the effective viscosity μ_{EFF} and other quantities related to turbulence that will be required during the next explicit iteration. This kernel is executed by a 1D NDRange composed by Nv work-items.

5.3.10 ALE and MRF

As explained in 2.1.3 and 3.6 the ALE framework is adopted to perform unsteady simulations with mesh deformation and exploited to implement MRF for turbomachinery and open rotor applications. Here the focus is on MRF. As explained, MRF is implemented using 3 components: ALE velocities, wall velocity boundary conditions and a momentum equations source terms. ALE face velocities and wall velocity boundary conditions are computed on the host during pre-processing and their values are transferred on the device through buffers that are then read when computing convective and viscous fluxes. When performing simulations with rotating domains the user provides the rotational axis direction and position, and angular velocity. Knowing the internal and boundary faces centers it is possible to compute the face velocities using the host code showed in listing 5.23:

Listing 5.23: *Host code for MRF pre-processing*

```
// Loop over all faces
for( int i = 0; i < Nf; i++  )
{
    // Extract face center
    vector Point( CCfx[i], CCfy[i], CCfz[i] );

    // Extract MRF velocity
    vector Velocity = Omega ^ ( Point - Origin );

    // Fill MRF arrays that will be transferred on device
    ALEx[i] = Velocity.x();
    ALEy[i] = Velocity.y();
    ALEz[i] = Velocity.z();
}

// Loop over all patches
forAll( mesh.boundaryMesh(), iPatch )
{
    // Select just wall boundaries
    if( mesh.boundaryMesh().types()[iPatch] == "wall" )
    {
        // Loop over all faces of the patch
        forAll( mesh.boundaryMesh()[iPatch].faceAreas(), ii )
        {
            // Addressing
            label i = ii + mesh.boundaryMesh()[iPatch].start();

            // Copy ALE velocity into velocity field boundary conditions
            U_bcx[i-Nfi] = ALEx[i];
            U_bcy[i-Nfi] = ALEy[i];
            U_bcz[i-Nfi] = ALEz[i];
        }
    }
```

```
}
```

ALE velocity is applied over all faces (internal and boundary), thus a loop over all the Nf = Nfi + Nfb faces is performed. For each face i, the user-prescribed angular velocity Omega is combined, with the operator ˆ used in OpenFOAM for vector product, with the distance between the rotation center Origin and the face center Point in order to compute the ALE Velocity. This operation can be easily performed using C++ objects and operators thanks to OpenFOAM API. The final storage is performed using 3 ALE arrays related to OpenCL buffers that will be transferred to the device. The second loop is performed over patches. Using the if statement, wall type boundaries are selected. This is done since it is necessary to consider the wall velocity due to rotation within boundary conditions computations. Index ii is local to the iPatch patch but it is necessary to extract the global index i in order to correctly access data structures. The global index is used to access ALE arrays since they are defined over all the mesh faces. Instead, i-Nfi is used to access boundary conditions values since they are defined just over the Nfb = Nf - Nfi boundary faces, in order to save device global memory. This code can be executed once during the simulation or multiple times by gradually applying MRF velocity in order to help convergence, as explained in 5.2.3. ALE and boundary conditions velocities are used in kernels whenever convective and viscous fluxes have to be computed.

Finally, as explained in 3.6, source terms on momentum equations have to be added. Basically, the the following lines have to be added to source terms computation kernel:

Listing 5.24: *MRF source term*

```
__kernel void makeSources
(
// arguments
)
{
    int k = get_global_id(0);

    // Extract solution
    real tWx = Wx[k];
    real tWy = Wy[k];
    real tWz = Wz[k];

    // Extract cell volume
    real tV  = V[k];

    // Compute MRF source term
    real Cx = -tV*( Omegay*tWz - Omegaz*tWy )*isRotor[k];
    real Cy = -tV*( Omegaz*tWx - Omegax*tWz )*isRotor[k];
    real Cz = -tV*( Omegax*tWy - Omegay*tWx )*isRotor[k];

    // Update residuals
    Rx[k] += Cx;
    Ry[k] += Cy;
    Rz[k] += Cz;

    // Computations for turbulence models
}
```

As explained in 5.3.7 the source terms kernel is executed spreading Nv work-items, one for each domain cell. First the solution is extracted from global memory and temporarily stored in private memory alongside the cell volume value. Omegax, Omegay and Omegaz are stored in global memory as single real values (thus as single 4-bytes or

8-bytes variables depending on the chosen floating point precision without the need of entire buffers) since they are constant in the rotor domain. isRotor is a buffer of Nv integer elements filled with just 1 or 0. This is used to activate the source term only in the rotor sub-domain. This way, since angular velocity is constant, a single integer buffer plus 3 real values are used. Another strategy would be to use 3 buffers with Nv elements each, to store for all the domain cells the angular velocity with an obvious increment in memory consumption (due to storage of 0s in stator domains). Furthermore it would be also possible to avoid reading the solution for the cells of the stator sub-domain with an if statement based on the isRotor[k] value. However this would lead to branch divergence issue depending on mesh numbering. Anyway here data is sequentially read using index k for all the necessary quantities. This way memory accesses coalescing is exploited. Finally, another strategy would be to execute this kernel just over rotor cells, but this would require some sort of pre-processing and addressing. Furthermore if rotor cells IDs were not consecutive then non-coalesced memory accesses could represent a problem for performances. As explained in 5.3.7, at the end of the source terms kernel, turbulence models source terms are computed and stored in global memory.

5.3.11 Mesh deformation

It is reminded, as explained in 2, that for performances reasons and thanks to the DTS strategy, aeroelastic interface building, modal interface computations and wall nodes displacements/velocities computations are performed on the host, i.e. by the CPU. Thus these stages are not discussed here. The focus here is on the most important formulations directly accelerated by the GPU regarding mesh deformation. In particular, here the implementation of the IDW algorithm (see 2.2.6) is discussed. The kernel code is quite simple and showed in listing 5.25:

Listing 5.25: *IDW kernel code*

```
__kernel void inverseDistanceWeighting
(
// arguments
)
{
    // Exctract node ID
    int k = get_global_id(0);

    // Extract original node position
    real X = Rx[k];
    real Y = Ry[k];
    real Z = Rz[k];

    // Initialize provate data for nodes displacements
    real dX = 0.0;
    real dY = 0.0;
    real dZ = 0.0;
    real NORM = 0.0;

    // Each node loops on patch nodes of deformable patches
    for ( int i = 0; i < size; i++ )
    {
        // Compute distance betwen node k and patch node i and (directly storage
            avoided)
        real dx = ( X - Cx[i] );
        real dy = ( Y - Cy[i] );
        real dz = ( Z - Cz[i] );
```

```
        // Compute weight based on a power of the inverse distance
        real dd = dx*dx + dy*dy + dz*dz;
        real IDW = 1.0/( dd + SMALL );

        // Extract patch node i displacemnets weight it and sum its contribution
        dX += IDW*Dx[i];
        dY += IDW*Dy[i];
        dZ += IDW*Dz[i];

        // Compute total weight for subsequent normalization
        NORM += IDW;
    }

    // Normalize the displacements and update node position
    Px[k] = X + dX/NORM;
    Py[k] = Y + dY/NORM;
    Pz[k] = Z + dZ/NORM;
}
```

Here Nv instances of the kernel are executed, assigning one work-item to each aerodynamic mesh node. AeroX handles absolute displacements, i.e. displacements are always defined with respect to the original undeformed mesh configuration (the mesh at time 0 for unsteady simulations or the guess mesh for trim simulations). Thus, the first step is the extraction from global memory of the original nodes positions. When work-item k reaches the loop, size iterations are performed, one for each patch node i where displacements are enforced. This includes not only wall nodes of deformable boundaries but also wall nodes of rigid patches since they contribute with null displacements but with non-null weights. Furthermore, also the contributions given by faces of other kind of patches, like periodic boundaries, are taken into account. This is due to the fact that on farfield, inlet and outlet patches and periodic patches the mesh geometry is kept fixed. Thus, to obtain a consistent internal mesh deformation the contribution with null displacements of these boundaries is taken into account through weights. Inside the loop the first operation is the computation of the distance between node k and patch node i. The reason behind the use of Cx, Cy and Cz arrays is the fact that patch displacements are actually defined on faces centers rather than mesh nodes. These ease the handling of cases where two adjacent patches are one rigid and one deformable since it is not directly possible to decide if the common mesh nodes are meant to be fixed or movable. The next step is the computation of the weights based on a power of the distance, as explained in 2.2.6. A SMALL value is added to avoid numerical problems, especially in SP executions with meshes exhibiting a very fine near-wall discretization. Then, the displacements contribution of patch node i over node k is accumulated in dX, dY and dZ. The weights are accumulated in NORM private memory variable. Finally, after the loop, the k-th node position is updated. It is reminded here, as explained in 2.2.6, that although weights are fixed due to the use of absolute displacements, they are recomputed every time instead of just being stored in global memory. This is done in order to avoid excessive device global memory usage. As will be showed in 6.2.2 the implementation of the IDW algorithm is quite efficient on GPU thanks to the high number of floating point operations due to the high number of internal nodes and the fact that all work-items require the same wall nodes data. This way GPU cache memory is exploited.

The next step is mesh metrics update. It is reminded that connectivity is preserved. Metrics computation is performed using two kernels, one for faces and one for cells.

The adopted strategies are here briefly discusses without going into details. For what concerns faces the kernel is executed in Nf instances. An addressing array is adopted to find the point composing the edges of each face. The addressing strategy is inspired on what presented for gradients and residuals assembly (see 5.1.3). However, here, instead of faces surrounding cells, the focus in on points surrounding faces. The computation of the face metrics kernel regards the update of area, normal unit vector, face center and, if required, ALE velocity. ALE velocity is updated using the formulation presented [127] considering the volume swept by the face during the physical time step and the faces normal direction. Once the face metrics is updated, another kernel is executed in Nv instances to update cell metrics: cell center and volume. The employed strategy is similar to the Green–Gauss formulation, considering face centers to compute the cell center and face areas to compute the cell volume. Thus from the same computational aspects discussed for gradients and residuals assembly are also valid for cell metrics update.

Alongside mesh deformation and the ALE velocities computation for both internal and boundary faces, the wall velocity boundary conditions must be opportunely adjusted. The host is responsible to compute both wall faces displacements and velocities. These are then transferred to the device and read inside boundary conditions kernels. Wall velocities are used to build the ghost cells solution before calling the convective fluxes algorithm. The chosen convective fluxes algorithm is then called by passing the internal cell solution (and the extended cell solution for high resolution), the ghost cell solution (and the extended ghost cell solution for high resolution) built considering wall velocity. ALE velocity is also considered for convective fluxes computations and passed as one of the arguments of the convective fluxes function. This way it is possible to obtain a consistent formulation for both inviscid and viscous simulations.

It is also worth to briefly discuss the implementation of the transpiration boundary conditions. As explained in 2.2.7, the idea is basically to opportunely alter the non-penetration boundary conditions, enforcing an opportunely crafted speed. This is done by the kernel showed in listing 5.26:

Listing 5.26: *Transpiration kernel*

```
__kernel void transpiration
(
// arguments
)
{
    int k = get_global_id(0);

    // Addressing
    int i  = wallAddressing[k];
    int ii = i - Nfi;
    int id_L = localOwner[i];

    // Air velocity
    real Ub[3]  = { Wx[id_L]/Wr[id_L], Wy[id_L]/Wr[id_L], Wz[id_L]/Wr[id_L] };

    // Moving wall velocity
    real Vb[3]  = { VelX[ii], VelY[ii], VelZ[ii] };

    // Unit vector variation
    real dnb[3] = { RotX[ii], RotY[ii], RotZ[ii] };

    // Original unit vector
```

```
    real nb[3]  = { -nx[i], -ny[i], -nz[i] };

    // Transpiration velocity, non-linear formulation
    Vn[ii] = ( Ub[0]*dnb[0] + Ub[1]*dnb[1] + Ub[2]*dnb[2] )
           - ( Vb[0]*( nb[0] + dnb[0] ) + Vb[1]*( nb[1] + dnb[1] ) + Vb[2]*( nb[2]
               + dnb[2] ) );
}
```

A different strategy is here employed to execute the transpiration kernel with respect to what presented when discussing boundary conditions (see 5.3.4). Rather than executing one kernel for each patch where transpiration is employed, a single kernel is executed involving all the wall faces. In fact, differently from the case with boundary conditions, here there is no branch divergence since boundary faces from different wall patches execute the same algorithm. However, an addressing buffer, wallAddressing is used to find the global face ID given the face ID defined among the set of wall faces (i.e. the work-item ID k). This addressing buffer contains for each wall face the correspondent face global ID, i, that can be used to access buffers like the already introduced localOwner. In order to save global memory, buffers containing data that is meaningful only for boundary faces, like the face velocity VelX, VelY and VelZ, and the face normal unit vector rotation RotX, RotY and RotZ, a new index ii is computed by subtracting Nfi (the first boundary face ID). The transpiration kernel is relatively cheap from a floating point operations content, since the bulk of computations is performed on the last line of code where the results are also directly saved into global memory. Again, index ii is used to write to global memory the transpiration speed, since it is defined just over boundary patches.

The transpiration kernel is scheduled for execution using the following host code 5.27:

Listing 5.27: *Transpiration host code*

```
runKernel( transpiration, Nwall, queue );
# include "debugger.C"
```

Nwall represents the total number of wall faces in the mesh, a subset of the boundary faces. After its computation the transpiration speed can be used inside boundary conditions kernels by opportunely modified slip boundary conditions. As usual, this is enforced indirectly by building the ghost cell solution and by passing internal and ghost cells solutions to the convective fluxes algorithm.

5.3.12 Cyclic boundary conditions

When performing turbomachinery and open rotors simulations supposing N-blade domain periodicity, periodic boundary conditions can be used to reduce the entire computational domain to N blades, as explained in 3.7. OpenFOAM provides the cyclicAMI boundary condition type that can be used to implement periodic boundary conditions. As explained in 3.7, the idea is to employ an interpolation of the solution on periodic patch A (considering the internal cells values of boundary faces) in order to allow the computation of fluxes on the other side of the domain on patch B (through ghost cell approach). The purpose of the interpolation is to find for each face of each periodic patch, the correct ghost cell solution, related to the solution on the other patch. Obviously the aim is to compute the fluxes that correctly represent periodicity. The interpolation

procedure requires addressing and weights buffers. In particular, for each face F of patch A it must be known the set of faces on patch B with which F communicates, and the weights related to them. The computation of the addressing and weights is completely performed during pre-processing by the OpenFOAM, on CPU. Thus, the solver host code job is just to build and transfer buffers with this data on the device. It is not worth to discuss the details behind the host job here. The actual computation of periodic boundary conditions is instead entirely performed by the device. Here, the implementation of periodic boundary conditions is just briefly discussed since from a GPGPU point of view the computational efficiency problems are similar to what encountered with residuals/gradients assembly. Two kernels are used: one to perform the interpolation and one to effectively apply the boundary conditions. The kernel related to the interpolation is executed for each periodic patch independently with a number of work-items equal to the number of patch faces. It must be noted that for what concerns momentum, the rotation tensor is required for 3D cases, as discussed in 3.7. This means the read of 9 scalar values for each patch. These scalars are shared between all the faces of that patch. This way cache can be exploited. The first operation performed in the interpolation kernel is, as always, the extraction of the work-item ID, i, and the related face global ID. The next step is to find for each patch face i the faces on the other patch that can be overlapped to i due to periodicity. This is done by using an addressing buffer to obtain both the total number of overlapping faces (in order to set the size of the next loop) related to face i and the starting point of the arrays that are actually used to extract the IDs and weights of the faces on the other patch. This addressing strategy is conceptually identical to what is used for residuals and gradients assembly. A for loop is then used to iterate over the overlapping faces on the other patch to proceed with the solution interpolation. As whit residuals and gradients assembly kernels branch divergence happens when different faces i have to deal with different numbers of faces on the other side of the domain. This is due to the fact that a different number of for loop iterations have to be performed. However, if faces from the two periodic patches are perfectly matching, no branch divergence happens since the loop would be performed in just one iteration, with a single weight equal to 1. However, this is rarely the case. For what concerns memory accesses, addressing is required to extract weights, global faces IDs, cells IDs and the solution. This inevitably reduce performances. In any case the time spent in this kernel is small since few thousands of faces are processed in a mesh of about one million cells. The structure of the kernel that actually applies cyclic boundary conditions is identical to what presented in 5.3.4 for other boundary condition types. The key difference here is that instead of reading boundary conditions from user-supplied data, they are directly obtained from what previously computed by the interpolation kernel. Since these values are stored sequentially, face after face, and in boundary conditions kernel are sequentially read with the face index, memory coalescing can be exploited.

5.3.13 Delayed periodic boundary conditions

As explained in 3.8, the key idea of delayed boundary conditions is the manipulation of periodic boundary conditions in order to apply on the two boundaries an opportunely chosen solution previously stored during the simulation at a particular physical time. This way it is possible to simulate the phase-shifted blades vibration with the advantage

of reducing the whole $360°$ domain to an N-blades sector (usually 1-blade sector). The procedure is implemented in 3 stages. First a pre-processing stage is performed on the host in order to compute addressing and weights thanks to OpenFOAM. This is done in the same manner as for periodic boundary conditions. However, here the host has more work to do. The solution over delayed patches (i.e. the solution of the faces owner cells, since a ghost cell approach is adopted) has to be stored for a specific number of physical times. This is due to the fact that in successive physical times the previous solutions will be required in order to perform the time interpolation. All solutions are saved on the device global memory in order to speed up the interpolation procedure without requiring a continuous data transfer between the host and the device. Obviously it is not possible neither useful to save the solution of all physical times. Knowing the delay and the physical time step size it is possible to pre-compute the total number of snapshots of the "periodic" patch solutions (solution of the internal cells owner of the patch faces) that are required for later interpolation. It is thus possible to cyclically overwrite previous solutions not useful anymore [128]. Roughly speaking this is implemented through a circularly linked list. Inside the explicit pseudo time iterations of the same physical time step the same physical time interpolated solution obtained from older physical times is read from global memory while the actual patch solution is updated every pseudo time iteration. Thus face fluxes on delayed patches faces are computed using the previous pseudo time cell solutions for the internal cells and the physical time interpolated solution of the previous physical time steps for the ghost cells. The same spatial interpolation discussed in 5.3.12 for perdiodic boundary conditions is also employed here when the faces of the two delayed patches are not perfectly matching. The kernel that actually applyies the delayed boundary condition is the same discussed in 5.3.12, except for the fact that now the ghost cell solution is computed in a different way that is not the simple weighted interpolation of the solution on the other side of the domain. In fact it is the interpolated solution on the other side of the domain at a specific previous physical time, obtained by a physical time interpolation between two physical times. It is the host (CPU) job to find, given the delays and the current physical time, the two physical times to use to perform the time interpolation. It is the device job to effectively interpolate the two solutions with GPU acceleration. This strategy requires the transfer of just few bytes of memory between the host and the device that represents the IDs of the two physical time, used as starting points of patch solution arrays. The other job of the device is the storing of the current pseudo time solution overwriting the previous solution (using the cyclic list idea [128]). The kernel that actually perform the interpolation is not showed here since, once the host has found the right time IDs, it is very similar, from the computational point of view, to a simple vector sum (see 4.4.3) since one work-item is assigned to one boundary face and the solutions of the two times are read, weighted, added and finally stored in global memory. Thus no branch divergence happens and perfect memory coalescing is reached, though the floating point to global memory reads ratio is small.

CHAPTER 6

Computational Benchmarks

The goal of this work is to obtain a solver that is both fast and accurate. The former feature is realized through the solution of the compressible (U)RANS equations inside an aeroelastic framework. The latter is realized through the GPU acceleration. Thus, it is also mandatory to investigate the speed-up advantages provided by GPU executions with respect to typical multi-core CPUs. In this chapter, before the validation of the solver through a comparison of literature and experimental results, the investigation is focused on purely computational aspects. The important concept of the time per iteration per cell will be introduced and adopted as a meter of comparison for the solver capability to accelerate simulations when executed on GPUs. The focus will be also posed on the possibility of the solver of taking advantages of cheaper gaming GPUs with respect to specifically designed HPC GPUs. The differences between results achieved using single precision and double precision will be investigated both for what concerns the speed-ups and possible CFD results accuracy problems. Finally, an investigation of the efficiency of the solver among different kinds of unstructured meshes, both hybrid and non-hybrid will be showed. This is done in order to asses the capabilities of the solver to maintain high computational efficiency with different types of meshes.

6.1 Hardware aspects

Here the most important hardware aspects related to the machines adopted to perform benchmarks are showed and briefly discussed. During this work different workstations were used for the source code development and debugging and obviously to perform useful simulations. CPUs and GPUs from different vendors were tested to guarantee maximum compatibility since this was one of the main goals of this work. CPUs from Intel and AMD, and GPUs from NVIDIA and AMD were used as OpenCL devices.

The purpose of this section is to briefly describe the main features of the devices used to perform speed-up benchmarks. As explained in 4.4.1, when a CPU is chosen as computational device, a multi-thread approach is adopted by the underlying OpenCL implementation, eventually with the use of implicit vectorization [8]. This assures that a fair comparison is performed between CPU and GPU execution times, by exploiting all the available physical (and eventually virtual with Intel HyperThreading) CPU cores. Furthermore, with environment variables, BIOS settings or other tricks it is possible to enforce a user-defined number of CPU cores to be used for kernel executions. This way it is possible to investigate the capability of the solver to scale over an increased number of CPU cores in a shared memory architecture.

Different CPUs and GPUs with different prices and performance levels were used for benchmarks. One of the goals of this section is to show that higher performance to price and higher performance to power consumption ratios can be obtained with GPU acceleration. As explained in 4.2.2, alongside the reduction of simulation times, power efficiency is a non-secondary aspect.

It must be noted that since the available devices come from different time periods, it is not straightforward to directly compare their performances with respect to their price level as, for example, a new mid-end GPU could beat an old (and maybe still) more expensive high-end GPU.

6.1.1 GPUs

Tables 6.1 and 6.2 show the GPUs employed in this work for the simulations and speed-up benchmarks. Specifications were obtained from [37, 38]. It is possible to see

Table 6.1: *GPUs used for simulations and computational benchmarks, 1/2.*

Vendor	Model	Cores	Freq.	Glob. mem.	BW	SP	DP
NVIDIA	Tesla C1060	240	$602\,MHz$	$4\,GB$ GDDR3	$102.4\,GB/s$	$\sim 930\,GFLOPS$	$\sim 80\,GFLOPS$
AMD	A10-7700K	384	$720\,MHz$	N/A DDR3	N/A	$\sim 550\,GFLOPS$	$\sim 34\,GFLOPS$
NVIDIA	GTX 660	960	$960\,MHz$	$2\,GB$ GDDR5	$192\,GB/s$	$\sim 1800\,GFLOPS$	$\sim 80\,GFLOPS$
AMD	380X	2048	$970\,MHz$	$4\,GB$ GDDR5	$182.4\,GB/s$	$\sim 4000\,GFLOPS$	$\sim 250\,GFLOPS$
AMD	290X	2816	$1040\,MHz$	$4\,GB$ GDDR5	$320\,GB/s$	$\sim 5600\,GFLOPS$	$\sim 700\,GFLOPS$
AMD	Fury X	4096	$1050\,MHz$	$4\,GB$ HBM	$512\,GB/s$	$\sim 8600\,GFLOPS$	$\sim 530\,GFLOPS$

Table 6.2: *GPUs used for simulations and computational benchmarks, 2/2.*

Vendor	Model	Price	TDP	Time	SDK
NVIDIA	Tesla C1060	$\sim 1700\,USD$	$188\,W$	2009	CUDA SDK
AMD	A10-7700K	$\sim 150\,USD$	$95\,W$	2014	AMD APP SDK
NVIDIA	GTX 660	$\sim 200\,USD$	$130\,W$	2012	CUDA SDK
AMD	380X	$\sim 230\,USD$	$190\,W$	2015	AMD APP SDK
AMD	290X	$\sim 400\,USD$	$290\,W$	2013	AMD APP SDK
AMD	Fury X	$\sim 650\,USD$	$275\,W$	2015	AMD APP SDK

that DP performances are usually a fraction of SP performances in gaming GPUs. At the time of writing the NVIDIA Tesla C1060 is a very old HPC GPU (its cost of $1700\,USD$ is referred to year 2009). Nonetheless this device was used to check for

possible advantages offered by ECC memory, see 6.2.4. It is reminded that the SP and DP performances showed in the table are just theoretical values that can be achieved only when the code is perfectly optimized for the underlying architecture, usually by exploiting FMA (Fused Multiply Add) instructions and coalesced memory accesses.

6.1.2 CPUs

Tables 6.3, 6.4 show the CPUs employed in this work for the simulations and speed-up benchmarks. It must be noted that independently from the fact that a CPU or a GPU is used as device, the CPU is always used as the host. This means that a small fraction of the computational power offered by the CPU is used to organize the work to be sent to the device. This is not particularly a problem when not all the CPU cores are used for kernels execution. In fact, thanks to multi-threading, the thread(s) that execute kernel code and the thread that execute host code can be scheduled for execution by the operating system kernel on different available cores. However, if all the available CPU cores are employed for device computations, host operations will lead to a slight overhead. The specifications showed in the tables were obtained from [5, 36] and

Table 6.3: *CPUs used for simulations and computational benchmarks, 1/2.*

Vendor	Model	Cores	Freq.	SP	DP
AMD	Phenom II X4 840	4	$3.2\,GHz$	$\sim$?	$\sim$?
AMD	A10-7700K	4	$3.4\,GHz$	$\sim 110\,GFLOPS$	$\sim 55\,GFLOPS$
Intel	i7 3930K	6	$3.2\,GHz$	$\sim 150\,GFLOPS$	$\sim 75\,GFLOPS$
Intel	i7 5930K	6	$3.5\,GHz$	$\sim 300\,GFLOPS$	$\sim 150\,GFLOPS$

Table 6.4: *CPUs used for simulations and computational benchmarks, 2/2.*

Vendor	Model	Price	TDP	Time	SDK
AMD	Phenom II X4 840	$\sim 100\,USD$	$95\,W$	2011	AMD APP SDK
AMD	A10-7700K	$\sim 150\,USD$	$95\,W$	2014	AMD APP SDK
Intel	i7 3930K	$\sim 500\,USD$	$130\,W$	2011	Intel OpenCL Runtime
Intel	i7 5930K	$\sim 600\,USD$	$140\,W$	2014	Intel OpenCL Runtime

from other different websites on the internet. Different parameters are showed in the tables. It must be noted that it is quite difficult to find the price of a device. It is basically possible to consider the vendor suggested price or what can be found on e-commerce websites. Here no details are provided for what concerns global memory size or technology, since basically for CPUs it depends on what the user decides to install as on-board RAM. At the time of writing this could be DDR3 or DDR4 technology with different frequencies, sizes and number of channels, depending on what supported by the CPU/motherboard and the user choices. Here just the CPU itself is considered. Despite GPUs architectures, with CPUs the theoretical DP performances are basically one half of SP performances due to how SSE/AVX works. It is reminded that SP and DP performances are just indicative and can be reached only when the code is fully optimized to use SIMD extensions. It is also quite difficult to find the official theoretical

GFLOPS values concerning CPUs from both AMD and Intel. The values showed in the tables may be slightly inaccurate.

6.1.3 APUs

The AMD A10-7700K is a so-called APU (Accelerated Processing Unit). It is basically a combination of a CPU and a GPU on the same chip which is installed on the motherboard [10] like an normal CPU. However, an APU has both CPU cores and GPU cores and therefore is showed in all tables 6.3, 6.4, 6.1 and 6.2. It must be noted that the GPU global memory is not showed since it can be changed by the user accessing the BIOS options as it is basically implemented using the system RAM. From the point of view of OpenCL and AeroX, the user can choose to use the GPU cores or the CPU cores but they cannot be directly combined to form a unified device. Anyway it can be seen that the SP performances provided by the 384 GPU cores are higher than what provided by the 4 CPU cores. With APUs the idea is to exploit CPU cores for serial operations while offloading heavy SIMD floating point computations to GPU cores. The PCI-Express bandwidth bottleneck is not a problem since the GPU is not installed on a discrete card. However the bandwidth provided by system memory, DDR3, is anyway lower than what provided by GDDR5 or HBM technologies of modern GPUs. As for the amount of global memory, the bandwidth is not showed in tables 6.1 and 6.2 since it strictly depends on the frequency of the RAM banks installed by the user on the motherboard slots.

6.2 Benchmark cases and results

6.2.1 Overall speed-up and multi-thread scalability

Here the results concerning the overall speed-up achieved by AeroX with GPU acceleration are illustrated [129]. A comparison between GPU executions and CPU single-thread or multi-thread executions is performed and the overall speed-up is discussed. This means the speed-up regarding entire pseudo time iterations, without profiling each single kernel (that instead will be performed in 6.2.2). The adopted performance metrics is the time per iteration per cell. Basically the solver is executed for a certain amount of pseudo time iterations, e.g. 10000, and the spent computational time is normalized by the total number of domain cells and the total number of iterations performed. This way it is possible to obtain an indicator of the computational efficiency of the chosen device that is independent from the mesh size and the convergence properties of the adopted case. However, the dependency from the type of elements and how mesh entities are indexed could trigger branch divergence and non-coalesced memory accesses performances issues. These aspects are investigated in more details in 6.2.3.

Adopting a test case specifically designed to maximize the GPU speed-up would not be fair. Thus, the idea is to benchmark the solver using a useful test case. Here the NASA's Rotor 67 fan blade test case is adopted to investigate the overall speed-up of AeroX when GPU acceleration is enabled. This case is discussed in details for what concerns the CFD results in 10.3. The mesh has about 1 million hexaedra cells. This means that it has enough cells and faces to keep all the devices listed in 6.3, 6.4, 6.4, 6.1 and 6.2 fully loaded for the entire simulation. This is not an hybrid mesh, thus there is no branch divergence during assembly kernels like residuals and gradients. However

it is an unstructured mesh. Depending from the device architecture, performances may be afflicted by a reduced memory accesses coalescing. The mesh is firstly renumbered with the renumberMesh OpenFOAM utility in order to try to reduce the overhead due to global memory accesses. Here a steady solution is searched. Thus, for now, kernels related to mesh deformation and metrics update are not considered. This is not a problem because, as will be explained in 6.2.2, even for unsteady cases with mesh deformation and trim cases, the bulk of computations is still related to steady kernels thanks to DTS approach. All convergence acceleration techniques are active and SP is exploited to maximize both GPUs and CPUs performances. Kernels related to viscous fluxes are executed at each pseudo time iteration. Tables 6.5, 6.6 and 6.7 show the time/iteration/cell obtained with the available CPUs and GPUs respectively. Tables

Table 6.5: *Time/iteration/cell (seconds) with different CPUs with SP, 1/2.*

Phenom 1C	Phenom 4C	APU CPU 1C	APU CPU 4C	i7 3930K 1C	i7 3930K 6C
$5.99e-6$	$1.74e-6$	$2.41e-6$	$9.30e-7$	$1.82e-6$	$3.67e-7$

Table 6.6: *Time/iteration/cell (seconds) with different CPUs with SP, 2/2.*

i7 5930K 1C	i7 5930K 6C
$1.28e-6$	$2.34e-7$

Table 6.7: *Time/iteration/cell (seconds) with different GPUs with SP.*

APU GPU	GTX 660	380X	290X	Fury X
$2.80e-7$	$6.37e-8$	$5.40e-8$	$2.81e-8$	$2.70e-8$

6.5 and 6.6 show for the employed CPUs both results with 1 and all the cores (4 or 6 depending on the particular model) loaded in order to investigate how well the solver scales when using multiple cores. The AMD A10-7700K is used both in CPU and GPU mode in order to check for advantages given by the APU concept when offloading numerical computations to the GPU cores. Tables 6.5, 6.6 and 6.7 show raw results that can be elaborated to show relative speed-ups between devices. The main idea is to check the overall speed-ups provided by GPUs over CPUs in multi-threading mode. In literature sometimes authors show the GPU speed-up relative to a single-core CPU execution. However this is probably not a fair comparison since the CPU is not entirely exploited. Table 6.8 shows the acceleration obtained by the available GPUs with respect to the available CPUs when all CPU cores are loaded with kernels computations. It is possible to see that except for the low-end APU in GPU mode, all GPUs are faster than CPUs (that are using all cores), and in particular faster then the high-end 6-cores Intel i7 5930K CPU. It is possible to do some considerations. The Phenom CPU is the slowest and oldest one (2011) and it is easy to recognize that it is basically the slowest device. However, it is a low-end CPU since it is in the $100\,USD$ price range. This CPU

Table 6.8: *Speed-ups with SP, GPU relative to CPU with multi-thread enabled.*

	APU GPU	GTX 660	380X	290X	Fury X
Phenom 4C	6.21×	27.3×	32.22×	61.9×	64.4×
APU CPU 4C	3.32×	14.6×	17.2×	33.1×	34.4×
i7 3930K 6C	1.31×	5.76×	6.79×	13.1×	13.6×
i7 5930K 6C	0.84×	3.67×	4.33×	8.35×	8.67×

is installed in a desktop computer alongside the GTX 660 GPU. This GPU is from around the same period, 2012, but in the $200\,USD$ price range, which is twice the cost of the CPU. However, when analyzing performances, it is possible to see that using SP in a real test case the GPU is over $27\times$ faster than the CPU using which is using all its 4 cores. Considering also that this GTX660 GPU has a TDP of $130\,W$ while the Phenom CPU has a TDP of $95\,W$ it is easy to see the advantages provided by the GPU from a power consumption point of view. This is more evident if a recent CPU, the i7 5930K, from 2014 is considered. This CPU is over $7\times$ faster than the Phenom CPU. However the i7 is an high-end CPU in the price range of $600\,USD$ and the two CPUs have 3 years difference in their architectures (e.g. i7 supports AVX that increase floating point performances). Nonetheless the GTX 660 is still almost 4 times faster than this high-end i7 CPU while costing 1/3 and being 2 years older. Furthermore, the i7 5930K TDP is slightly higher than the one of the GTX 660: $140\,W$ for the former and $130\,W$ for the latter. These aspects suggests, as an example, that if the goal is to run the solver on a workstation, instead of performing an expensive hardware upgrade by changing CPU, memory and motherboard with high-end components, it would be also possible to obtain performance improvements by just installing a relatively inexpensive GPU on the PCI-Express slot and installing the freely available OpenCL runtimes and GPU drivers form AMD/NVIDIA. These are aspects mainly related to budget or old workstations. From the table it is also possible to see that the AMD Fury X is 64 times faster than the Phenom CPU. However, this comparison is not fair as the two devices belong to two very different time periods and price ranges. This way the technology and monetary gap is too wide to directly make comparisons. However, high-end GPUs such as the AMD 290X and the AMD Fury X can be compared to the two high-end i7 CPUs since the time periods and price ranges are comparable. In particular the i7 5930K and the Fury X belong nearly to the same time period (2014 for the CPU and 2015 for the GPU) and the same price range ($600\,USD$ for the CPU and $650\,USD$ for the GPU). However, the TDP is quite different since the i7 5930K has $140\,W$ while the Fury X has $290\,W$, more than twice. When considering these two devices and their relative speed-up, it is easy to see that the GPU is almost one order of magnitude faster (8.67 times for this study-case). This means that in order to reach the same performances provided by the Fury X multiple high-end workstations would be needed and combined together using an high bandwidth inter-connection. Otherwise server-class CPUs would be required. Anyway this would mean spending at least one order of magnitude more money than what required for the single Fury X GPU. Furthermore, nowadays motherboards, even mid-range motherboards, allow the installation of multiple GPUs. As an example, both the 290X and Fury X GPUs are

installed on the PCI-Express slots of the motherboard that also hosts the i7 5930K CPU. This means that the two GPUs can be used concurrently to perform simulations on two different cases at the same time. In any case the installation of a second or third GPU just requires the bought of the GPU(s), and eventually an adequate Power Supply Unit (PSU). In fact, power consumption becomes important for this kind of multi-GPU configurations since it can be seen from tables 6.3, 6.4, 6.1 and 6.2 that the sum of the TDPs of the 3 devices is around $700\,W$. However, if the same TDP would be related to purely CPU power, performances would be lower. It is worth to remind that installing multiple CPUs on the same motherboard or in any case dealing with multiple interconnected CPU nodes requires specifically designed hardware that is in any case multiple times more expensive than having two GPUs on the same workstation or gaming PC.

Finally, it is worth to spend some words about the APU in GPU mode. As it is possible to see from the benchmarks, when the APU is used in GPU mode, the computations are performed 3.3 times faster (with respect to the 4-core CPU execution or 8.6 times with respect to the CPU single-thread execution), leading to a significant reduction of simulations times. It must be reminded that this is a low-budget APU, in the price range of $150\,USD$. Furthermore GPU cores comes already installed on the CPU, without the need to buy a dedicated PCI-Express compatible card. This way, it is worth to exploit all the APU capabilities. This APU in GPU mode proves to be just slightly slower (around 16%) than the high-end Intel i7 5930K CPU using all the 6 available cores. The i7 5930K CPU costs around $600\,USD$, which is around 4 times the price of the APU. The two devices are both from 2014 and thus directly comparable for what concerns their prices. This is thus a good example to show where OpenCL and GPUs provide advantages in terms of both money and energy savings. In fact, the power consumption (the TDP to be correct) of the APU is 47% lower than what required by the i7 5930K CPU. Finally it must be noted that this is just a comparison between GPU cores of the APU and CPU cores of the i7 5930K CPU. The CPU cores of the APU are not exploited. Exploiting the CPU cores of the APU alongside its GPU cores would lead to further advantages in terms of performances if considering the same overall TDP value. Finally a note regarding the direct comparison between APU results in CPU mode and i7 5930K results. While both CPUs come from the same time period (2014) and have around the same frequency ($3.4\,GHz$ for the APU and $3.5\,GHz$ for the i7) they have very different architectures (they are from different vendors and different consumer targets). In particular, the i7 5930K has more cache than the A10-7700K ($15\,MB$ SmartCache vs. $4\,MB$ L2 cache). Furthermore with the APU CPU cores, when using AVX instructions, two cores shares the same $256\,bit$ AVX unit, while each core of the i7 5930K has its own $256\,bit$ AVX unit. Obviously the rest of the differences are given by the two more cores of the i7 5930K, the HyperThreading support, the DDR4 memory support (while the APU supports the older DDR3 technology). All of these differences justifies the higher multi-thread performances of the i7 5930K with respect to the APU in CPU mode. This also means that while with AeroX the APU in GPU mode could be very competitive with the i7 CPU in term of performances, price and power consumption, if the aim of the workstation is instead to perform other kinds of heavy CPU-based computations the i7 would be easily be preferred. This is obviously related to the fact that, as explained in 4, GPUs are not aimed to substitute CPUs

in numerical computations but to help CPUs whenever is possible by the underlying algorithm.

Table 6.9 shows the speed-ups obtained with multi-thread CPU executions with respect to single-thread CPU execution in order to asses the capability of the solver to scale in a multi-thread CPU environment. The speed-ups obtained with each CPU are referred to the same CPU in single-thread execution. It is possible to see that the low-

Table 6.9: *Speed-ups with SP, CPU only, multi-threading enabled.*

Phenom 4C	APU 4C (CPU only)	i7 3930K 6C	i7 5930K 6C
$3.44\times$	$2.59\times$	$4.96\times$	$5.47\times$

end Phenom CPU obtains a $3.44\times$ speed-up using all the 4 cores, the A10-7700K obtain a $2.59\times$ speed-up in CPU mode with all 4 cores, the i7 3930K obtains a $4.96\times$ speed-up using all its 6 cores and the i7 5930K obtains a $5.47\times$ speed-up using all its 6 cores. It can be seen that the code scales quite well up to 4 and 6 cores using multi-threading. Obviously the slowest CPU is the older and cheaper one, the AMD Phenom, that has just 4 cores. Performances are also consistent between the two i7 CPUs, with higher performances provided by the newer CPU, since they come from different time periods (2011 and 2014) and thus have slightly different architectures. The newer CPU has slightly higher frequency ($3.5\,GHz$ vs. $3.2\,GHz$) and slightly more cache ($15\,MB$ vs $12\,MB$). They both have 6 cores and $256\,bit$ AVX units that can be exploited through implicit vectorization provided by Intel OpenCL SDK. For what concerns the APU in CPU multi-thread mode, it is possible to see that the speed-up is only $2.59\times$ when using all its 4 cores. This is probably due to the underlying "Steamroller" x86 AMD architecture of CPU cores. The cores in this CPU are grouped in modules of 2 physical cores sharing the $256\,bit$ AVX unit. What happens is that probably this allows to use the entire $256\,bit$ AVX unit when a single thread is used to perform the simulation while when 4 threads a used, only two $256\,bit$ AVX unit are available instead of 4. A speed-up greater than 2 can be probably justified by the fact that not only floating point operations are required to execute the solver. Thus, using 4 cores provides benefits for the overall execution. The Phenom CPU has only $128\,bit$ SSE units, one for each core. This is probably why in single thread executions 1 core of the A10-7700K is more the twice faster than one core of the Phenom CPU although they have similar frequencies. However, it must be noted that the Phenom CPU is 3 years older than the APU. Thus, also the architectures are quite different. For what concerns the CPU results related to the SSE/AVX utilization, the presented are just a considerations. More accurate profiling, assembly code/compiler and hardware architecture analyses are required to better justify the encountered behaviors. However, this is beyond the purposes of this work.

6.2.2 Kernels speed-up

Here, instead of analyzing the speed-up obtained when considering entire pseudo time iterations, single kernels profiling is employed. This allows a fine grained investigation of the computational efficiency of each kernel, thus each formulation and implementation strategy. As with any other application profiling, the focus here is on the most

time consuming kernels. This regards both kernels with high floating point computations and kernels with an high number of memory accesses. As explained in 5.2.5, OpenCL directly provides profiling capabilities as part of the API. Thus, it is possible to directly check the computational time required for the execution of each kernel without using third party tools. During profiling, slightly different results were obtained for the computational time required by the same kernel if profiled multiple times. These are normal fluctuations. Thus, for the same combination of kernel and device, several executions were performed and an averaging operation was employed to improve results reliability. The test case chosen for this analysis is again the NASA's Rotor 67 fan blade, described in details in 10.3. This time AeroX is executed in trim mode, thus it is possible to perform the profiling for both CFD and mesh deformation kernels. The tests are performed on the Intel i7 5930K CPU using all the 6 cores and on the AMD Fury X GPU. Single precision is employed to maximize performances of both devices. Table 6.10 shows the speed-ups obtained with the GPU over the CPU for each kernel set. The "Convective" column shows the speed-up related to the execution of the Roe

Table 6.10: *Kernels execution speed-up with GPU with SP.*

	Convective	Periodic BC	Viscous	Residuals Assembly	IDW	Mesh metrics
Speed-up	97.4×	6.67×	16.1×	5.97×	13.25×	6.98×

fluxes device code, using the extended cells strategy discussed in 5.3.2. The "Periodic BC" column is referred to the kernel that performs the solution interpolation over the periodic patches, see 5.3.12. The "Viscous" column considers to the sum of the times required for the executions of the kernels related to viscous fluxes. This means considering both gradients kernels and the kernels related to the effective viscous fluxes computation, as explained in 5.3.5. "Residuals Assembly" column is related to the assembly kernel alone, discussed in 5.3.6. "IDW" column is related to the Inverse Distance Weighted formulation kernel alone (see 5.3.11). Basically it is just the kernel adopted to update the aerodynamic mesh points positions. Finally "Mesh metrics" column represents kernels for mesh metrics computation. These are executed after mesh deformation. Both kernels for faces and cells metrics update are considered. From the table it is possible to see that different kernels exhibit different speed-ups. In particular the kernel related to convective fluxes achieve the higher speed-ups. This is due to the fact that, as explained in 5.3.2, a very high amount of floating point operations is performed between the initial global memory reads and the final fluxes writes. However there is no branch divergence that could afflicts performances. This allows very high computational efficiency on GPU architectures. For what concerns periodic BCs the speed-up is lower due to the face addressing required for interpolation that triggers both branch divergence and non-sequential memory accesses. For what concerns viscous fluxes the speed-up is governed by two aspects. On one side the kernels related to gradients assembly are basically memory bounded since just few floating point operations are required thanks to the Gauss algorithm (see 2.1.6 and 5.3.5). In case of hybrid meshes the gradients assembly kernel will lead to a performance loss also due to branch divergence (but this is not the case with the mesh here employed). On the other side the kernel that actually computes the viscous fluxes, face by face, has a bet-

ter floating point operations to memory accesses ratio. However this is not as high as with the convective fluxes kernel. This justifies the lower speed-up provided by viscous formulations kernels with respect to convective fluxes kernel. The lowest speed-up, as expected, is obtained with the residuals assembly kernel. As explained in 5.3.6 this kernel is heavily memory bounded but anyway its use allows to directly handle meshes of any kind (hybrid unstructured meshes). Few floating point operations are performed inside this kernel with respect to the total number of global memory accesses since the only required operation is the sum of the fluxes of the faces surrounding each cell. For what concerns the IDW algorithm the GPU is one order of magnitude faster than the CPU, providing a speed-up of over $13\times$. As explained in 5.3.11, this kernel is essentially composed by a for loop over wall nodes in order to extract and weight wall nodes displacements. There is no branch divergence since all the aerodynamic mesh nodes (mapped to work-items) have to loop over the same wall nodes. Cache can be exploited since the same wall nodes data is used by all work-items. Even though there is no branch divergence the total number of floating point operations is relatively small with respect to the total number of global memory reads. Thus the speed-up is smaller with respect to what obtained with convective fluxes. Finally, the speed-up provided by mesh metrics computations, both for faces and cells, is about $7\times$. Again this is due to the fact that, especially for faces metrics update, an high number of global memory reads in a non-sequential manner is required. Although this is not the case, if the mesh features faces with different number of points, branch divergence issues are also triggered since the kernel is composed by a loop over faces edges and each work-item is assigned to each face. For what concerns cell metrics update, the algorithm implemented in the kernel is computationally similar to the Green–Gauss gradient algorithm and residuals assembly computations. Again a relatively small number of floating point operations is coupled with global memory reads/writes. Anyway the high speed-up is enough to justify the use of GPU if the prices of the CPU and the GPU are considered, alongside power consumption and the other aspects already discussed.

As said, table 6.10 shows the speed-ups provided by different kernels. The speed-up values are obtained computing the ratios between the computational times required by kernels on different devices. As it is possible to see from table 6.8 considering the i7 5930K CPU and the Fury X GPU, an overall speed-up of $8.67\times$ is obtained. This is smaller then e.g. the speed-up obtained for convective fluxes. Even though different kernels are characterized by different speed-ups, they are also characterized by different execution times. This is translated in different weights on the overall speed-up value. This is also the reason why, thanks to DTS, the computational time spent for mesh deformation is basically negligible with respect to purely aerodynamics kernels, when considering the overall aeroelastic simulation. As an example, for the Rotor 67 test case, the computational time spent for each pseudo time iteration for the convective fluxes is around $1700\,\mu s$ with the Fury X and $170000\,\mu s$ for the i7 5930K (using alle the 6 cores). This leads to a speed-up of about 100 for the convective fluxes kernel alone. However the residuals assembly requires $2000\,\mu s$ for the GPU and $12000\,\mu s$ for the CPU, leading to a speed-up of about $6\times$. It is possible to see that due to the different architectures between the GPU and the CPU, the computational time spent by the GPU for the residual assembly is around the same of convective fluxes. However, with the CPU one order of magnitude gap between the two kernel execution times is obtained.

Consequently, with GPU acceleration, one order of magnitude difference in speed-up values is obtained with the same kernel This is also the reason why the overall speed-up is around $10\times$ instead of $100\times$ when considering this CPU and GPU combination. The execution times for the IDW algorithm are $1900000\,\mu s$ and $25000000\,\mu s$ for the GPU and the CPU respectively, leading to a speed-up of around $13\times$. It is possible to see that this kernel execution time is $1000\times$ higher than what required at each iteration for convective fluxes. However, while convective fluxes are computed at each pseudo time iteration, alongside with the other required kernels (e.g. viscous fluxes, assembly, convergence acceleration, boundary conditions...) IDW is performed only once for each physical time iteration or once for each trim iteration. For the particular case of the trim of the Rotor 67 here adopted for profiling, several thousands of purely aerodynamic pseudo time iterations are required between two mesh updates (see 10.3). The number of iterations is in the order of $O(10^4)$. Thus, the computational time spent by IDW and mesh update kernels becomes negligible with respect to the overall time required for pure aerodynamic iterations. As an example, a single purely aerodynamic pseudo time iteration with the Rotor 67 test case requires $112000\,\mu s$ using the Fury X. With the same GPU, $1900000\,\mu s$ are required to perform the complete mesh update (the metrics update time is negligible with respect to IDW). Considering a single mesh update every 500 iterations it is easy to see that the overall computational time spent updating the mesh is negligible with respect to the overall simulation time. In fact, with a single mesh update performed every 500 explicit iterations, only the 3% of overall execution time is occupied by mesh deformation. This is also true for unsteady simulations, where, thanks to DTS, one physical time iteration is performed every about 500-1000 pseudo time iterations (depending of convergence and user-defined parameters). In this case mesh deformation would account for just 1.6% of the computational time of each physical time iteration. This further justify the choice of the DTS formulation strategy for the GPU implementation for AeroX. In fact, using a global physical time stepping strategy for unsteady simulations with mesh deformation rather than DTS, the mesh deformation kernels have to be executed every explicit iteration, slowing down the simulation.

Figure 6.1 finally shows the speed-up provided by each GPU both in single and double precision mode, with respect to the high-end Intel i7 5930K CPU using as baseline ($1\times$ speed-up) the 6-cores multi-thread execution in single precision.

6.2.3 Mesh dependency

AeroX is a solver capable to perform simulations with unstructured hybrid meshes thanks to the compatibility with a wide range of mesh generation software provided by the OpenFOAM API and conversion tools. This means that the solver is compatible with meshes composed by different kinds of elements, such as tetrahedra, prisms, hexaedra, polyhedra. However, this important solver feature comes at the cost of a possible performance loss with GPU executions due to the need of addressing buffers, reducing coalesced memory accesses and possibly introducing branch divergence problems. It is thus worth to check how the solver behaves with meshes featuring different types of elements [129]. Furthermore, it is noted that high GPU speed-ups are obtained with meshes big enough to keep all GPU cores fully loaded. Obviously, due to branch divergence and non-perfect memory coalescing, the performances of AeroX will somehow

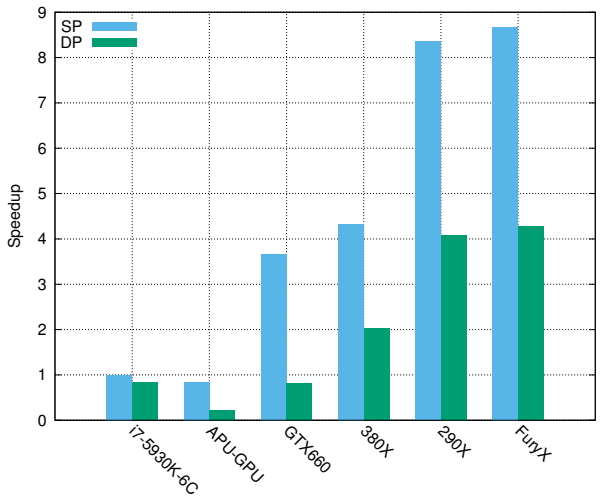


Figure 6.1: *GPU speed-ups. Baseline:* Intel i7 5930K *using 6 cores in SP mode.*

depend upon the particular mesh under investigation. Here the idea is to analyze the AeroX GPU speed-up sensitivity by changing mesh properties (size, kind) independently. This means using meshes with different sizes, and different kinds of elements. Meshes with different sizes basically change the total number of work-items instantiated by kernel executions for both face-based and cell-based kernels. Meshes with different types of elements instead change the number of operations performed inside each kernel instance when an assembly operation is involved (e.g. residuals assembly or gradients computation), possibly triggering branch divergence if the mesh is hybrid. The goal is to tune the solver in order to obtain performances as much as possible independent from the mesh properties. The adopted test case is a simple NACA 0012 wing (with chord $c = 1\,m$ and span $b = 10\,m$) in a rectangular ($40 \times 10 \times 10m$) wind tunnel. Steady inviscid simulations are performed, considering both memory-intensive (e.g. residuals assembly) and computational-intensive (e.g. convective fluxes) kernels running. The same case is simulated with the following unstructured (and some also hybrid) meshes, all with approximately $Nv \simeq 1.0 \cdot 10^6$ cells:

- a) Fully tetrahedral mesh;
- b) Fully prismatic mesh;
- c) Fully hexaedral mesh;
- d) Mixed mesh with approximately 60% tetrahedra, 20% prism, 20% hexaedra cells;
- e) Hexaedra-dominant mesh created with snappyHexMesh featuring 5% of polyhedral cells, with up to 18 faces per cell;

All meshes are firstly renumbered using the OpenFOAM renumberMesh tool based on the Cuthill–McKee algorithm in order to minimize band and optimize as much as possible the memory layout of each case. Figure 6.2 shows the time/iteration/cell obtained using all the 6 cores of the Intel i7 3930K CPU while figure 6.3 the values obtained with the NVIDIA GTX 660 GPU. These two figures show the AeroX solver

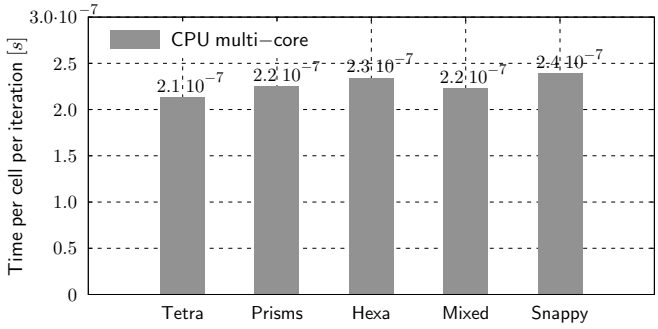


Figure 6.2: *Mesh dependency, CPU with SP.*

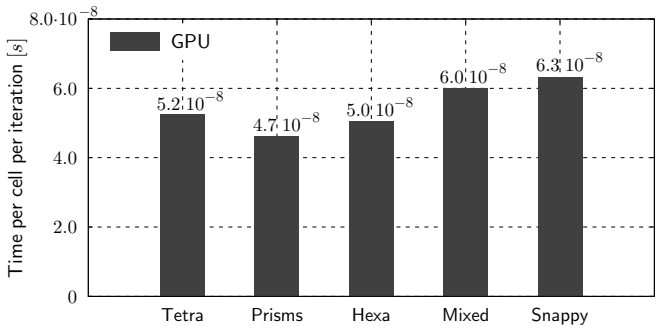


Figure 6.3: *Mesh dependency, GPU with SP.*

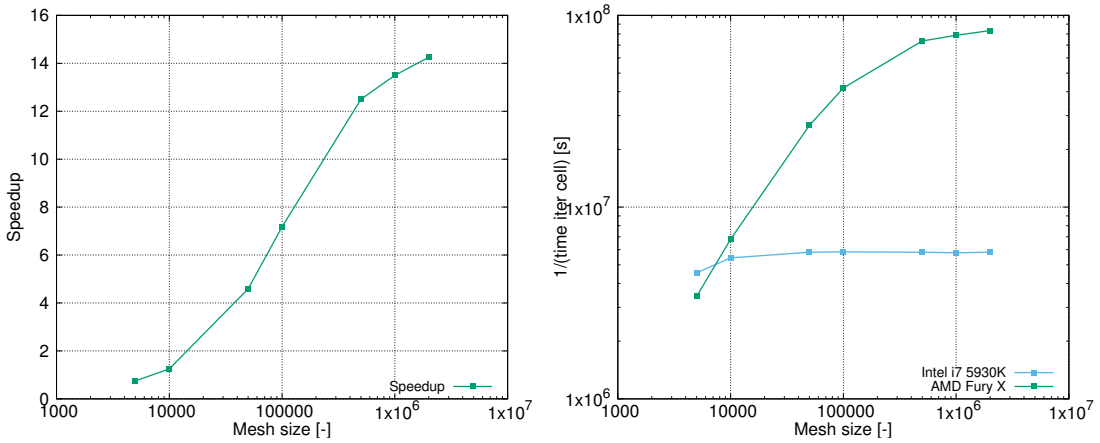
sensitivity to the mesh type of the computational cost metrics with a typical high-end CPU architecture and a typical mid-end GPU architecture. As expected, the computational cost metrics shows a negligible dependency from the mesh type in the multi-core CPU case, with a relative standard deviation below 4%. Viceversa, the computational cost metrics shows higher sensitivity to mesh type in the GPU case, with a relative standard deviation of approximately 12%. In particular the time per iteration per cell varies only slightly (due to slightly number of faces) for cases a), b) and c), while it increases significantly for cases d) and e) due to presence of cells with different number of faces, triggering branch divergence in assembly operations (e.g. residuals). Such a penalty is however limited as it does not jeopardize the observed speed-ups.

It is also worth to check the speed-up dependency from the mesh size. Since GPUs have nowadays thousands of cores and are optimized for thousands of concurrent work-items, they can reach peak performances with meshes exhibiting a sufficient number of cells. Mesh size is not particularly a problem for CPUs since they usually have few cores, thus they can be easily fully loaded just with small meshes. The dependency of the GPU speed-up from the mesh size is investigated simulating the same test case multiple times, changing every time the total number of cells and checking the time/iteration/cell performance metrics. This is done for both the i7 5930K CPU and the Fury X GPU. The adopted test case is the NACA 0012 airfoil, fully discretized with hexaedral cells. Single precision is employed. Results with different number of cells and devices are showed in table 6.11. From the table it is possible to see that, as expected, the GPU requires meshes with enough cells to achieve high speed-ups. In order to exploit as much as possible the 4096 cores of the Fury X computational power, millions

Table 6.11: *Time/iterations/cells (seconds) and speed-up dependency from mesh size, SP.*

	$5k$	$10k$	$50k$	$100k$	$500k$	$1M$	$2M$
i7 5930K	$2.20e-7$	$1.84e-7$	$1.72e-7$	$1.71e-7$	$1.72e-7$	$1.73e-7$	$1.72e-7$
Fury X	$2.92e-7$	$1.47e-7$	$3.75e-8$	$2.39e-8$	$1.36e-8$	$1.27e-8$	$1.20e-8$
Speed-up	$0.75\times$	$1.25\times$	$4.59\times$	$7.18\times$	$12.5\times$	$13.5\times$	$14.25\times$

of cells have to be employed. The CPU provides around the same time/iteration/cell for all mesh sizes. In particular, for meshes with more than $50k$ cells there is basically no difference in CPU performances. For the GPU the situation is quite different. In fact, it is possible to see that one order of magnitude difference in performances is obtained between mesh sizes of $5k$ and $2M$ cells. Furthermore, mesh dependency is quite relevant for meshes with sizes below $500k$ cells. This confirms the fact that, as in any other GPU application, the problem must be big enough to fully exploit the GPU architecture. From the table is is also possible to see that the speed-up provided by GPU acceleration increases until a plateau. Figures 6.4 show graphically, for the different mesh sizes, the speed-ups that can be achieved with the aforementioned devices and the time/iteration/cell for the two devices, highlighting that at least 1 million cells are required to achieve high efficiency for the AMD Fury X GPU. The CPU instead reaches the plateau with just few thousands of cells.



(a) Overall GPU speed-up

(b) Time/iteration/cell (seconds)

Figure 6.4: *Performance parameters trend with different mesh sizes.*

6.2.4 SP vs DP and ECC memory validations

One of the key ideas of this work, at the base of the chosen formulations and strategies is the exploitation of gaming GPUs that are one order of magnitude cheaper than HPC GPUs but provide basically the same SP computational power. The main drawbacks are: reduced global memory, reduced DP performance, lack of ECC memory capability. The purpose of this section is to prove that the solver is capable to provide accurate

results using SP instead of DP and to provide correct results without the need of ECC memory support.

As explained in 5.2.3, multiple strategies were implemented in the solver to avoid numerical problems when employing SP, such as non-dimensional equations. This is done in order to provide the same results accuracy level provided by DP execution. This problem is investigated by performing two simulations of the same test case, repeating the simulation for both floating point representations and finally checking for differences in results. This investigation is performed again on the NASA's Rotor 67 fan blade, in order to perform the checks using a real test case rather than an ad-hoc case. Different quantities can be checked to asses the reliability of SP simulations, e.g. checking residuals, integral quantities like the mass-flow at choking, and local quantities like the Mach number field at choking conditions. Here the outlet mass-flow at choke conditions for the Rotor 67 test case is adopted to perform the investigation. The simulation is performed using the AMD 290X GPU with SP and is then repeated with DP support. The difference obtained with the two floating point representations is below 0.05% on a mass-flow value of $1.562\,Kg/s$ obtained with SA turbulence model. With such a small difference it is possible to say that using SP with non-dimensional equations provides the same results accuracy obtained with DP support. No local visible differences where found inside local fields. It must be underlined that thanks to non-dimensional equations and the other strategies to avoid numerical issues with SP, alongside with results accuracy, SP executions have also the same convergence rate of DP executions. This is translated in the fact that SP and DP executions requires basically the same number of pseudo time iterations to reach convergence. This way it is directly possible to check the speed-up advantages provided by SP over DP in term of computational time. This also means that the time/iteration/cell parameter can be directly adopted to perform comparisons between SP and DP executions.

As said, the solver is specifically designed to exploit SP peak performances of cheap gaming GPUs and satisfy the limited global memory available on this kind of hardware. However, thanks to the typedef strategy (see 5.1.3) it is very easy to switch to DP mode. Obviously this feature guarantees the possibility to fully exploit HPC GPUs that exhibit bigger global memory and higher DP performances. Anyway it is worth to investigate how the solver behaves in DP executions with gaming GPUs. As showed in tables 6.3, 6.4, 6.1 and 6.2, due to how SSE/AVX works, CPUs DP performances are the half of their SP performances. For GPUs the situation is different since the architectures are very different from what found inside CPUs. With GPUs, DP performances are usually opportunely limited by GPU vendors for gaming GPUs to just a fraction of SP performances. Here the benchmarks presented in 6.2.1 are repeated with the same devices and test case with DP instead of SP. As it is possible to see from tables 6.1 and 6.2 the DP performances of gaming GPUs are just a fraction of SP performances, while for what concerns CPUs (tables 6.3, 6.4) they are just one half due to how SSE/AVX units work. As an example the AMD 380X GPU theoretical DP performances are just 1/16 of the SP counterpart. However, this is just the ratio between the two theoretical GFLOPS values when all cores are fully loaded. In real life applications, with the possible overhead due to branch divergence and memory accesses, these number cannot be easily reached. This means that in reality due to this kind of bottlenecks the gap between SP and DP performances could be different. Theoretical SP performances cannot be

easily reached in a every application. Furthermore provided the same global memory bandwidth, accessing double precision variables requires more data to be transferred than with single precision variables. This could further slow down kernel executions when employing DP. As for SP, tables 6.12 and 6.13 shows the results for CPUs and GPUs respectively. Table 6.14 shows the speed-ups provided by the GPUs over CPU

Table 6.12: *Time/iteration/cell (seconds) with different CPUs with DP.*

Phenom 1C	Phenom 4C	APU CPU 1C	APU CPU 4C	i7 5930K 1C	i7 5930K 6C
$7.96e-6$	$2.45e-6$	$3.26e-6$	$1.35e-6$	$1.54e-6$	$2.72e-7$

Table 6.13: *Time/iteration/cell (seconds) with different GPUs with DP.*

APU GPU	GTX 660	380X	290X	Fury X
$1.26e-6$	$2.86e-7$	$1.34e-7$	$6.65e-8$	$6.37e-8$

multi-thread executions. As expected, the speed-ups provided by GPUs are lower using

Table 6.14: *Speed-ups with DP, GPU relative to CPU with multi-thread enabled.*

	APU GPU	GTX 660	380X	290X	Fury X
Phenom 4C	$1.94\times$	$8.58\times$	$18.3\times$	$36.8\times$	$38.5\times$
APU CPU 4C	$1.07\times$	$4.73\times$	$10.1\times$	$20.3\times$	$21.2\times$
i7 5930K 6C	$0.22\times$	$0.95\times$	$2.02\times$	$4.09\times$	$4.27\times$

DP with respect to SP. As an example, from table 6.14 it is easy to see that the Fury X with DP is about $2.35\times$ slower than with SP though theoretically the ratio should be 16 due to hardware limitations. For what concerns the AMD 290X the obtained ratio is 2.36 with a theoretical ratio of 8. With the AMD 380X the two ratios are 3.14 and 8. With the NVIDIA GTX 660 the two ratios are 4.5 and 24. Finally for the APU the two ratios are 4.29 and 16 using GPU cores. These discrepancies can be justified with the aforementioned explanations, i.e. due to overhead not related to purely floating point operations. It is reminded that alongside the obvious advantages provided by faster executions, SP also allows the execution of bigger cases thanks to lower memory requirements.

For what concerns CPU executions in DP mode it is possible to compare tables 6.5, 6.6 and 6.12 to check for the advantages provided by SP in terms of execution times. As explained, the speed-up due to how SSE/AVX works should be theoretically two. However it is possible to see that the speed-up with SP is around 1.41 for the Phenom CPU, 1.45 for the APU in CPU mode and 1.16 for the i7 5930K. This is probably due to the fact that inside kernels, besides pure floating point operations, computational time is also spent for memory accesses and other instructions not directly related to numerical computations. It must be noted that, as with GPUs, using SP has also advantages in

term of memory consumption, allowing to simulate bigger cases with the same amount of system memory.

Finally, the solver is tested for possible differences obtained due to the lack of ECC memory in gaming GPUs. It is worth to say that during the entire development of the solver no errors were obtained that could be attributed to hardware memory issues. Anyway the check for possible problems related to the lack of ECC, and thus the possible need of ECC-complaint GPU, could be performed in different ways. As showed in [45, 151] it is possible to stress the ECC-compliant hardware first with ECC on and then with ECC off. Finally results are checked for possible differences that could be attributable to hardware memory errors, errors that could have been detected and/or corrected by ECC. It is also possible to stress the same GPU by performing the same simulation several times in a row and check for possible differences in results between the runs. In this work the NVIDIA Tesla C1060 HPC-class ECC-compliant GPU was used to check for memory problems. At the time of writing this GPU is quite old, with low FP performances, comparable to a nowadays low-end GPU. However it is still useful for this investigation in order to asses possible ECC advantages. The same test case, again the Rotor 67 fan blade, was run on the Tesla C1060 and the AMD 290X ten times but no differences in results were obtained between different runs on the same GPU. Very small differences, comparable to the numerical threshold were instead obtained between the results provided by the Tesla C1060 and the AMD 290X. However, these differences can be justified with different implementations and instructions reordering by the NVIDIA and AMD implementations since the same differences are encountered all the 10 times with exactly the same numbers. Basically the same conclusions found in [45, 151] can be applied also to this work.

CHAPTER 7

Fixed wing aerodynamic applications

After the analysis of the speed-up advantages provided by GPU executions, the solver is now validated in terms of results accuracy. In this chapter, in particular, purely aerodynamic test cases will be investigated. This is an important step required before the aeroelastic framework validation, since it is necessary to guarantee that the solver is capable to provide accurate inviscid and viscous results for a wide range of aeronautical cases before proceeding with the more complex aeroelasticity formulations validation. Alongside typical steady aeronautical cases of wings like the Onera M6 and the RAE 2822 airfoil, the 2nd Drag Prediction Workshop is adopted to asses the capability of the solver to provide accurate compressible viscous results. This is an important benchmark used to demonstrate that the solver is capable to correctly predict lift and drag coefficients.

It is worth to present different cases in order to demonstrate that AeroX is indeed a general purpose solver capable to simulate very different configurations such as airfoils, wings, aircraft and rotors.

7.1 Onera M6

In this first section the Onera M6 wing test case is analyzed. This is a 3D inviscid case used to validate the solver for what concerns convective fluxes and high resolution formulations. While details about this test case can be found in [136] it is still worth to briefly introduce the case. This is a transonic test case with an asymptotic Mach number of $M_\infty = 0.84$ and an angle of attack of $\alpha = 3.06°$. Figure 7.1 shows the detail of the wing discretization. Slip boundary conditions are enforced over the wing and the symmetry wall at the wing root, while characteristics-based boundary conditions are enforced on the farfield in order to automatically switch between inlet and outlet

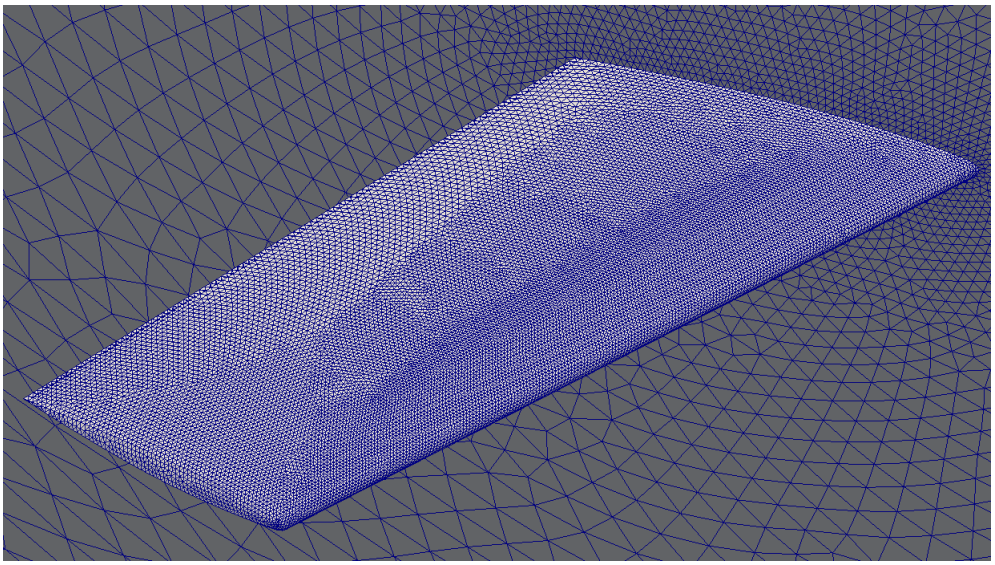
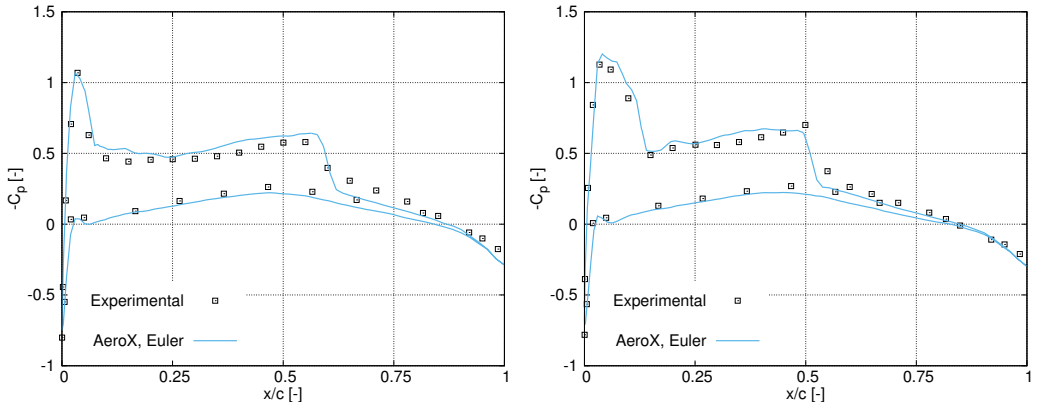


Figure 7.1: *Onera M6 mesh, wing detail.*

boundary conditions. The mesh is composed by $340k$ tetrahedral cells. This is a non-hybrid unstructured mesh, thus there is no branch divergence in the residual assembly kernel. The solver is executed searching for a steady-state solution with all convergence acceleration techniques active. EE is used to advance the solution in pseudo time. Roe fluxes with HR and extended cells strategy are adopted for the simulation. The required computational time is $239\,s$ on the AMD 380X GPU and convergence is reached in $20 \cdot 10^3$ iterations. The validation of the solution is performed comparing the solution provided by AeroX with experimental data provided in [135]. Experimental data is available for what concerns the C_P (pressure coefficient) over sections at various span locations (from 0.2 to 0.99). Figures 7.2 show the solution at different span locations, from 20% to 99%: It is possible to see that the curves provided by AeroX are in good agreement with experimental data. From figure 7.3 it is evident in the solution the presence of a shock over the wing that justify the rapid pressure change in C_P curves. In this figure a comparison is made between the solution provided by AeroX and the solution provided by AeroFoam [136]. It is clear that the solution provided by the two solvers is in good agreement.

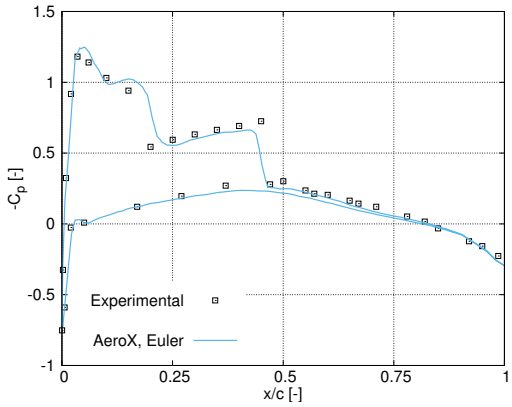
7.2 RAE

This is a well-know test case that is here adopted to show the capabilities of AeroX to simulate flows with interaction between compressible and viscous effects. Details regarding the case, reference numerical and experimental data can be found in [44, 44, 51, 60, 70]. For this test case different configurations can be investigated thanks to the available experimental data for what concerns both pressure and friction coefficients. Due to the particular airfoil shape and farfield conditions these cases are quite challenging for numerical solvers due to separations and shocks. Different solvers implementing different numerical schemes could lead to different shock positions and separation bubbles. Here three different conditions are investigated. These are showed in table 7.1. In all the three cases the Mach number is high enough to allow the for-

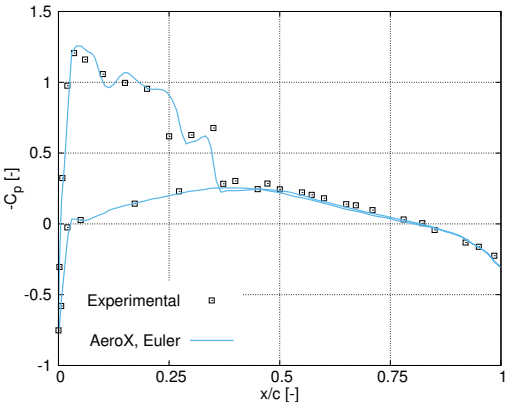


(a) C_P, 20% span

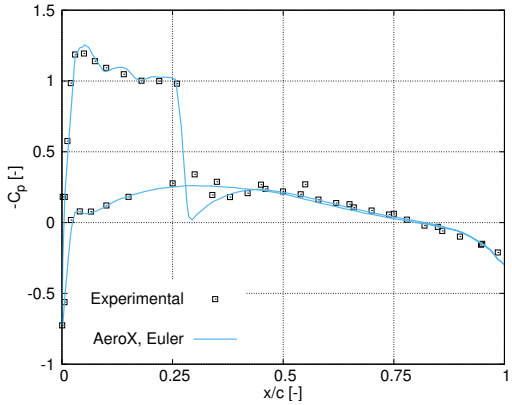
(b) C_P, 44% span



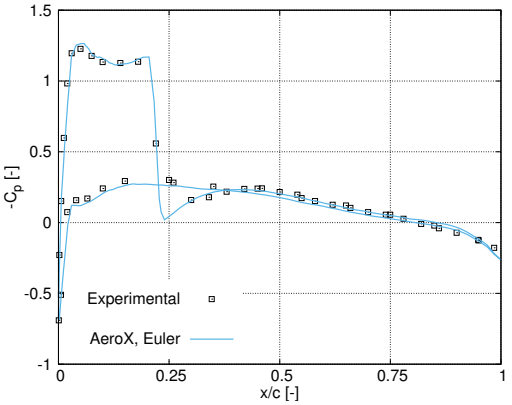
(c) C_P, 65% span



(d) C_P, 80% span



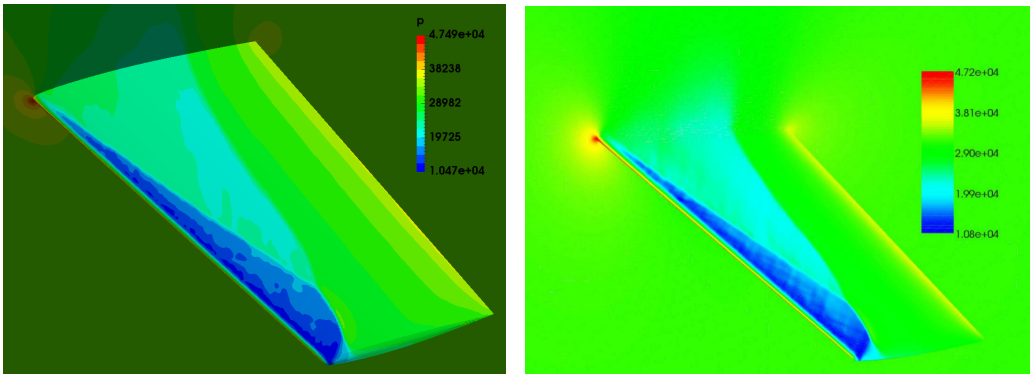
(e) C_P, 90% span



(f) C_P, 95% span

Figure 7.2: *C_P at different span locations.*

mation of a shock wave on the upper side of the airfoil. Furthermore, as mentioned before, a separation could occur on the upper side of the airfoil at the given angle of incidence. AeroX is here adopted to perform steady-state RANS simulations using SA and SST turbulence models and Roe fluxes with extended-cells high resolution strategy. The mesh adopted for these tests is composed by 38400 quadrilateral cells and the



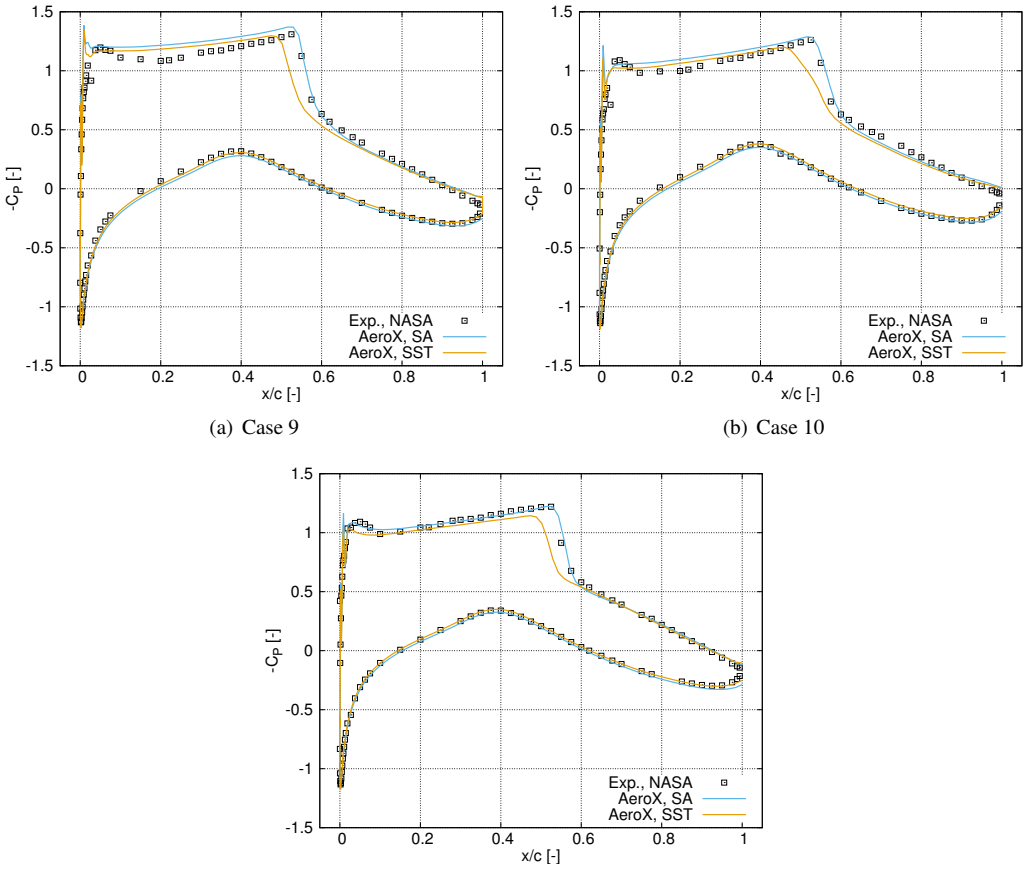
(a) AeroX (b) Reference, AeroFoam

Figure 7.3: *Pressure field.*

Table 7.1: *RAE 2822 test cases.*

Case name	M_∞	Re_{chord}	α
case 9	0.730	$6.5 \cdot 10^6$	3.19°
case 10	0.750	$6.2 \cdot 10^6$	3.19°
standard case	0.729	$6.5 \cdot 10^6$	2.31°

wall refinement allows values of y^+ in the order of 1.5, thus in the viscous sublayer, all over the airfoil. This is easily handled by the automatic wall treatment. For what concerns the total simulation time, this is about 120 seconds for each case on the NVIDIA GTX660 GPU, corresponding to $50 \cdot 10^3$ pseudo time iterations. Figures 7.4 show the results for what concerns the C_P distribution over the airfoil for the three cases under investigation. As it is possible to see the results provided by AeroX are in good agreement with experimental data for all the three cases. In particular, the shock location and pressure jump seem to be caught quite well, especially using SA turbulence model. Furthermore the C_P distributions on both the upper side and the lower side of the airfoil accurately follow the reference values. SST appears to predict shock positions slightly upstream of the experimental data. This behavior, however, is also obtained in other numerical analyses like [11] using SST. In this reference it is possible to see that SA in general produces better results for what concerns the shock position, similarly to what happens with AeroX here. It must be noted that, as noticed by other authors like [44], case 10 is quite challenging due to the difficulty to reach a steady-state solution. An oscillatory trend is obtained also with AeroX. This is just for case 10, as standard case and case 9 easily converge to a steady-state solution. Figures 7.5 show instead the C_F distribution over the airfoil for the three analyzed cases. It must be noted that for case 9 and case 10 results are taken from [44] (called "Experimental, NASA" in the legend) and reference [51] (called "Experimental, EUROVAL" in the legend), while for what concerns the standard case they are not directly available on the website [11] and thus the reference values are taken from the numerical simulation provided by another commercial software. For what concerns the standard case it can be seen that the AeroX



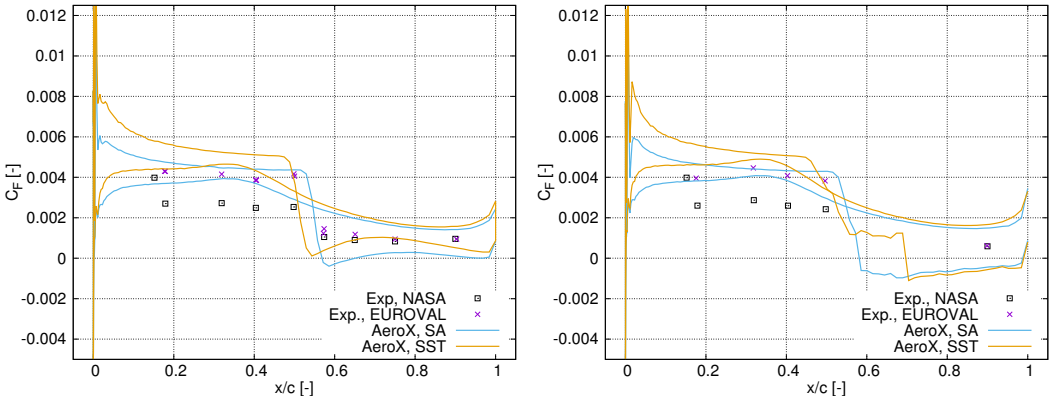
(a) Case 9

(b) Case 10

(c) Case website

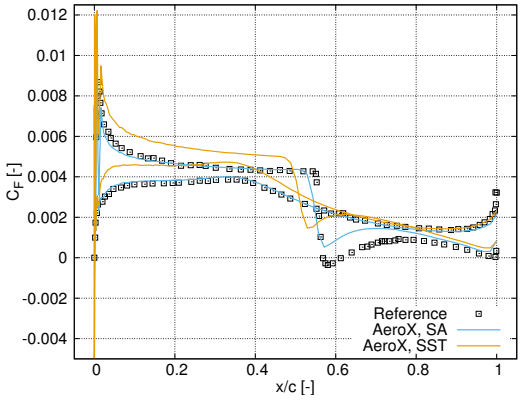
Figure 7.4: *C_P distribution over the airfoil.*

results with SA correctly match the reference data for the most part of the airfoil. However, with this turbulence model, differences are encountered on the upper side, right after the shock, where the friction coefficient seems to be overestimated with respect to the reference numerical solution. Using SST a general overestimation of the friction coefficient is obtained both on the upper and the lower side of the airfoil. Furthermore, due to the different shock locations, the friction coefficient jump is located slightly before the location provided by SA and reference data. For what concerns case 9 it is possible to see that before the shock the results provided by AeroX with SA match well experimental data of [51] while overestimate data provided by [44]. Using SST for the same case a general overestimation of the C_F is again present. However, after the shock in this case SST seems to better predict the value of the friction coefficient. A key difference between SA and SST in this case is given by the fact that while SA predicts a separation, for a very small region of the airfoil, on the upper side right after the jump, this is not obtained with SST. In this region SST better predict experimental data. Finally, for what concerns case 10 the same considerations of case 9 are valid before the shock for SA: the model is more in agreement with the experimental data provided in [51] while overestes reference data of [44]. This is also true for AeroX with SST. After the shock AeroX with both SA and SST predicts a separation over a



(a) Case 9

(b) Case 10



(c) Case website

Figure 7.5: *C_F distribution over the airfoil.*

wide region of the airfoil, basically up to almost the trailing edge. In this setup only one experimental point seems to be available from both experimental data sources after the shock on the upper side, thus it is not easy to understand if a separation should occur and how wide it has to be.

7.3 2nd Drag Prediction Workshop

Here the geometry of the 2nd AIAA CFD Drag Prediction Workshop is investigated. In particular, the geometry of the DLR-F6 configuration that represents a twin engine body aircraft of Airbus type is employed. This is a specifically designed test case [50] used to check the capabilities of numerical solvers to correctly simulate the flow around a typical aircraft geometry where both compressible and viscous effects have a determinant weight. In fact, the challenge is to accurately compute both lift and drag coefficients at different angles of attack in steady-state conditions. With these data it is then possible to compute the polar curve of the aircraft. Drag prediction is a critical challenge in the aeronautical field as it is strictly related to fuel consumption and costs. It is thus mandatory for a general purpose compressible RANS solver to be able to accurately predict the drag coefficient for the entire airplane. Numerical results provided by AeroX are

compared with literature results provided by an implicit pressure-based compressible RANS solver with SST turbulence model [102, 103] and experimental data. Both SA and SST turbulence models are well suited for this particular test case. Thus they are both employed for the simulations with AeroX to check for possible results accuracy differences, especially with high angles of attack. Even tough different angles of attack are here investigated, all the coefficients are obtained through steady-state simulations, thus there is no investigation on the effects of rapid change in the angle of attack of the aircraft.

Though this is a well-known test case, it is worth to discuss few details about the flight conditions. For the design Mach number $M_\infty = 0.75$ and Reynolds number $Re = 3'000'000$ the reference lift coefficient C_L is about 0.5. Different angles of attack, $\alpha \in [-4.8°, 1.82°]$ are investigated. Due to the model shape, positive lift values are obtained with negative angles of attack, as will be showed. Figure 7.6 shows the detailed view of the whole aircraft wall discretization. Figure 7.7 shows the detail of the boundary layer discretization. It is possible to see that the aircraft geometry is

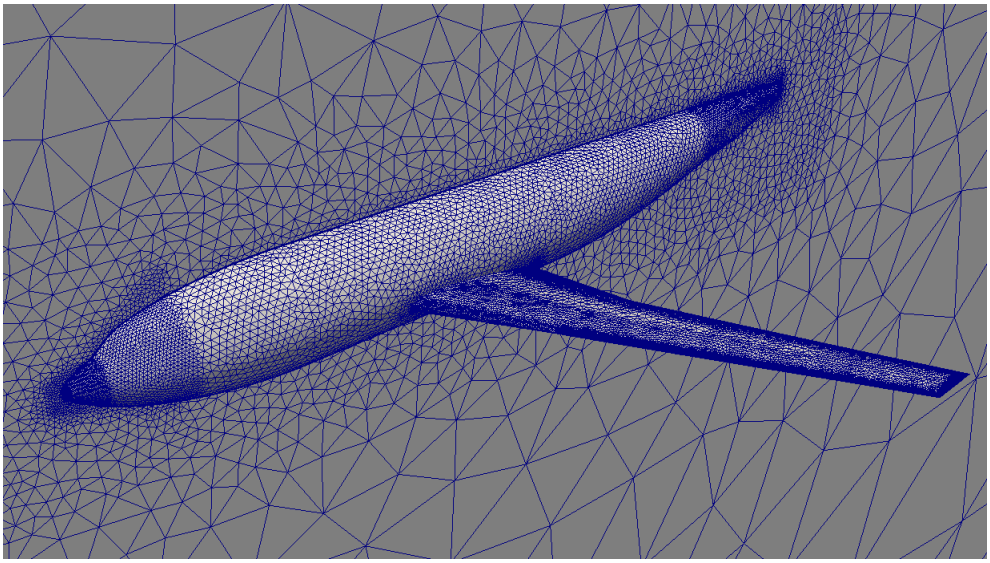


Figure 7.6: *DLR-F6 mesh discretization near the aircraft.*

surrounded by a boundary layer in order to accurately predict viscous effects and thus drag coefficients at all the angles of attack under investigation. The boundary layer discretization guarantees y^+ values raging from inside the viscous sublayer up to about 40 for all the angles under investigation. Thanks to the implemented automatic wall treatment this is perfectly fine using the blending strategy. Figure 7.8 shows the overall computational domain. Basically, a symmetry wall is adopted to reduce the computational domain since both the geometry and flight conditions are symmetrical. The computational domain is about 30 times the size of the aircraft in order for the employed boundary conditions to be representative of a true farfield situation. It is possible to see that there is enough distance between the aircraft and the farfield. Furthermore farfield cells are relatively big with respect to the aircraft size. The adopted computational mesh contains $2 \cdot 10^6$ cells. This is an hybrid mesh composed by tetrahedral cells ($\sim 61\%$) and prisms ($\sim 39\%$), thus branch divergence performance loss is expected. Nonethe-

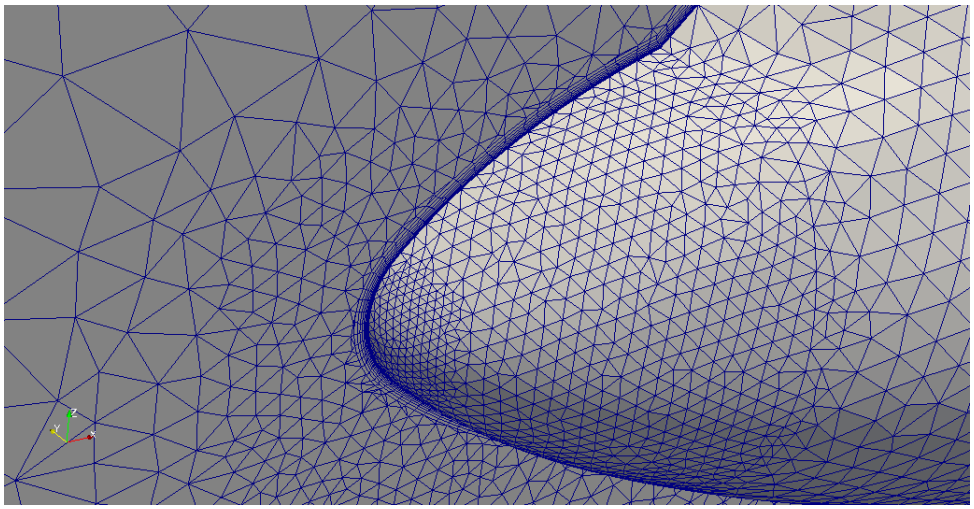


Figure 7.7: *DLR-F6 boundary layer discretization.*

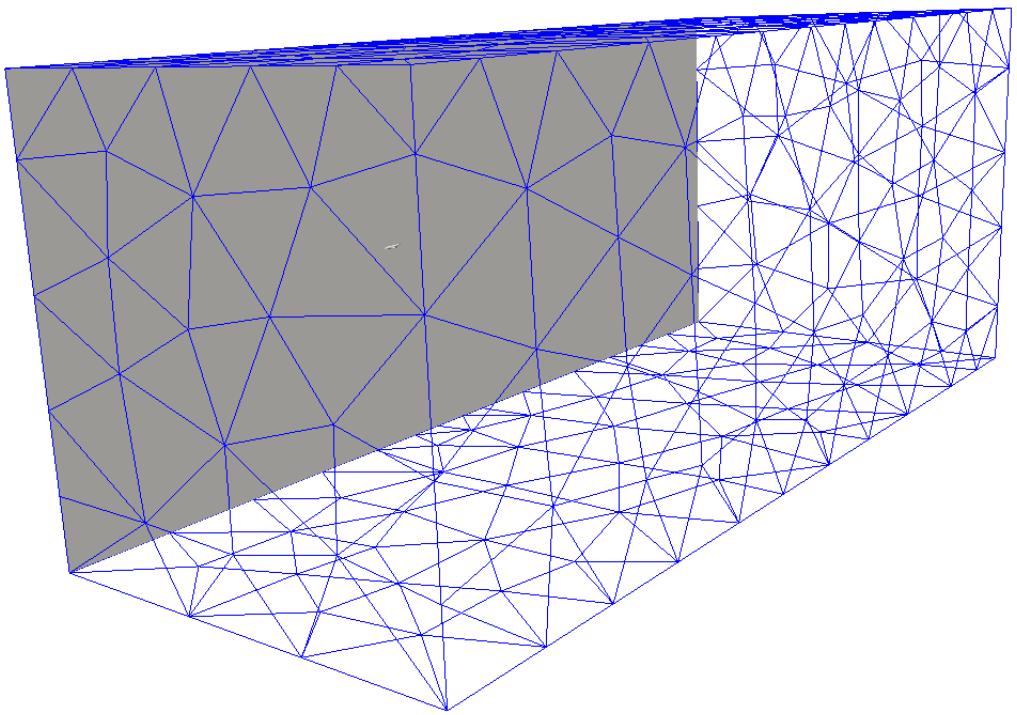
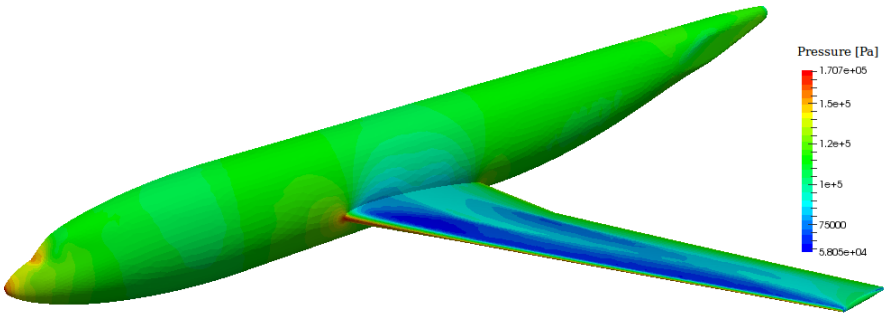


Figure 7.8: *DLR-F6 far-field discretization.*

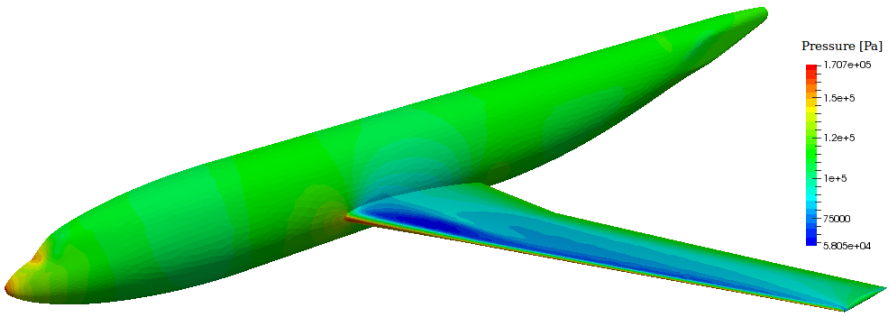
less with the AMD 290X GPU a time/iteration/cell value of $2.74 \cdot 10^{-8}\, s$ is obtained using SA turbulence model and $3.04 \cdot 10^{-8}\, s$ using SST turbulence model. These are results using SP. Thus it is possible to see that these values are in agreement with what showed in 6.2.1 for this GPU for the Rotor 67 test case, even when with an hybrid mesh branch divergence is present. Each investigated angle requires around $40 \cdot 10^3$ pseudo time iterations to reach convergence. In particular with SA the case with $\alpha = 0.02°$ took $2234\, s$ to reach convergence and $2499\, s$ for SST. Simulation times for the remaining angles are comparable with the same GPU. The same mesh is also adopted with

the implicit pressure-based compressible RANS solver with SST turbulence model to provide comparative results. Furthermore the pressure-based solver is also used with a mesh composed by $8.3 \cdot 10^6$ hexaedra cells.

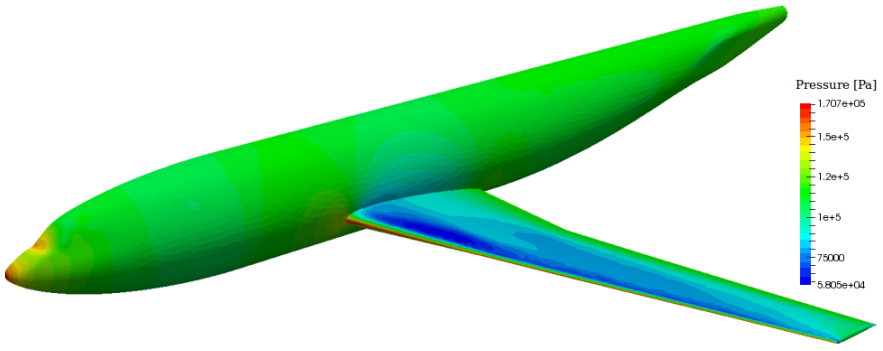
Figure 7.9 shows the results for what concerns the pressure field over the aircraft. In particular here a comparison is performed with AeroX results using inviscid simulation (Euler), AeroX with SA, and finally reference results with the pressure-based solver previously mentioned. Results with SST are basically comparable with what obtained



(a) AeroX, Euler



(b) AeroX, Spalart–Allmaras

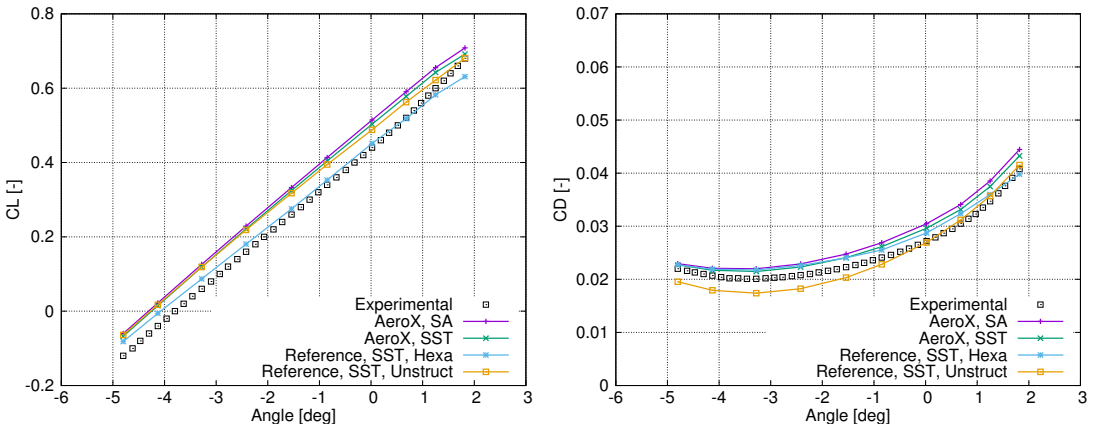


(c) Reference, SST

Figure 7.9: *Pressure distribution with AeroX and reference numerical results.*

with SA, thus they are not showed. It is possible to see that the results obtained with AeroX and SA are in good agreement with what obtained with the reference solution. The situation is different for what concerns the Euler solution since it does not take into account viscous effects. Euler solutions in this case tends to overestimate lift. As an example at null angle of attack the lift coefficient provided by inviscid simulations is

around 0.7 while viscous simulations predict about 0.5. This is a quite big difference. Thus in this case performing a viscous simulation not only allows to predict the correct drag but also improve lift prediction. It is now possible to analyze results regarding lift and drag. Figure 7.10(a) shows the value of lift coefficient with respect to the angle of attack. Figure 7.10(b) shows the value of the drag coefficient with respect to the angle of attack. Finally figure 7.10(c) shows the combination of these data in the polar curve of the aircraft. From the results it is possible to see that AeroX provides good



(a) $CL - \alpha$ (b) $CD - \alpha$

(c) Polar curve

Figure 7.10: *DLR-F6 performance coefficients.*

results using both SA and SST turbulence models. Both models provide comparable results, with slightly better results from SST for higher angles of attack. SA in fact is specifically designed for this kind of analysis of aeronautical components (especially airfoils) when no separation is present in the flow. With the employed mesh a slight constant overestimation in lift coefficient is obtained, while for what concerns drag a slight overestimation with high angles of attack is obtained. Combining these data into the polar curve it is possible to see again that AeroX provides good results with this mesh. It must be noted however that the polar curve is directly obtained combining C_L and C_D values but there is no direct reference of the angle related to those values inside the curve. Nonetheless these slight differences are probably related to the mesh itself

since it is possible to see that the pressure-based implicit solver provides differences in results just by changing to the adopted mesh.

CHAPTER 8

Fixed wing aeroelastic applications

After the validation of the purely aerodynamic formulations implemented in the solver, convective and viscous fluxes in particular, it is now time to analyze the fluid-structure interaction framework. Steady and unsteady cases will be presented. Here the focus is on classical aeronautical cases. Turbomachinery and open rotors aeroelastic cases will be presented in a next chapter. The well-known HiReNASD wing benchmark case is here adopted for the validation of the static aeroelastic solution, i.e. the trim solution. Another test case, the AGARD 445.6 wing will be adopted to investigate the flutter prediction capabilities of the solver, both in subsonic and supersonic regimes. Results for the 2nd Aeroelastic Prediction Workshop (AePW2) are here presented for what concerns the trim and flutter analyses. This in particular is a recent benchmark case specifically designed to asses computational capabilities in flutter and forced oscillations response prediction.

8.1 HiReNASD wing trim

Here the well known HiReNASD (High Reynolds Number Aero-Structural Dynamics) project wing [55] is investigated through a steady rigid and an aeroelastic (trim) analysis. In particular, the case No. 132 is considered. For this particular case, reference data provided by FUN3D numerical solver is also available from NASA [55] alongside experimental data. Furthermore, numerical results are also available in [127] and [115] for what concerns AeroFoam and S^T solvers, Euler/RANS and full potential formulation respectively. This case is important since trim analyses will be also performed subsequently for the Rotor 67 blade fan. Thus, it is better to firstly validate the solver with a classical aeronautical case and then proceed with a turbomachinery test case that is more complex due to the use of MRF and periodic boundary conditions. The

experimental setup of the HiReNASD test case is showed in figure 8.1: Basically it is a

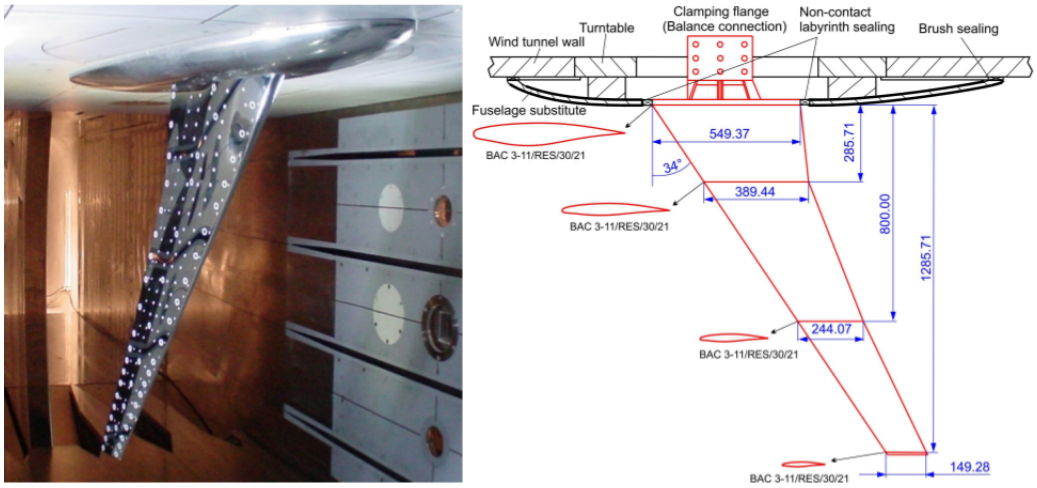


Figure 8.1: *HiReNASD wing experimental setup.*

ceiling-mounted semi-span clean-wing configuration with a leading-edge sweep angle $\Gamma_{LE} = 34°$, a span $b = 1.2857\,m$ and a mean aerodynamic chord $\bar{c} = 0.3345\,m$. From the figure it is possible to see that the wing is composed by three regions built with three different airfoils. Furthermore, a generic fuselage is included in the geometry in order to reduce the influence of the wall boundary layer during testing. Details about the geometry are available in [55, 127]. For the test case No. 132 the Reynolds number corresponding to the aeroelastic response of the wing-body couple is $Re = 7 \cdot 10^6$, while the Mach number is $M_\infty = 0.8$. The dynamic pressure is $q_\infty = 40055\,Pa$. Different angles of attack are investigated: $\alpha \in (-1.5°, 0.0°1.5°, 3.0°, 4.5°)$. Numerical and experimental data is available for comparisons regarding the C_P distribution at different span locations: $\eta \in (14.5\%, 32.3\%, 65.5\%, 95.3\%)$. Furthermore the values of the vertical displacements of the point at coordinates $(0.87303, 1.24521)$ on the wing, referring to figure 8.1, near the wing tip, are also available for the validation of the static aeroelastic response. Aeroelastic cases requires both aerodynamic and structural models. The purpose of the developed aeroelastic solver is to combine those independent models to provide the aeroelastic solution. Thus now it is time to discuss both the structural and aerodynamic models here adopted for the HiReNASD test case.

8.1.1 Structural model

The structural model here adopted is taken from [127]. The HiReNASD project provides structural models obtained through a Finite Element discretization of the wing including the body attached to symmetry wall. There, the discretization is performed using solid elements for a total of $200 \cdot 10^3$ grid points. As explained in 2.2.1 the strategy adopted in AeroX is based on a modal reduction of the structural behavior in order to accelerate as much as possible the host-based aeroelastic computations, trying to reduce the CPU overhead and keep the GPU as much as possible loaded during simulations. Here the trim validation is performed using a beam-based structural model with a limited number of d.o.f., opportunely tuned to match both the modal shapes and

frequencies obtained by the Ground Vibration Tests (GVT) measurements. In particular, the beam-based structural model here adopted is composed by 62 nodes distributed along the wing span, each of which is connected to 4 additional nodes (located at the trailing and leading edges and on the upper and lower wing surfaces) through rigid elements. This is done in order to improve the accuracy of the reconstruction of rotations by just using translations d.o.f. when assembling the aeroelastic interface between the structural and aerodynamic mesh. Table 8.1 shows the frequencies of the first 8 modes here adopted to represent the structural behavior of the HiReNASD wing. Figure 8.2

Mode ID	Frequency. (Hz)	Description
1	25.95	1^{st} bending
2	82.42	2^{nd} bending
3	117.58	1^{st} in-plane bending
4	168.42	1^{st} bending-torsion
5	258.38	3^{rd} bending
6	273.20	4^{th} bending
7	275.29	2^{nd} in-plane bending
8	275.29	2^{nd} bending-torsion

Table 8.1: *Modal frequencies and shapes of the first 8 modes of the HiReNASD wing.*

shows the shape of the first two modes and allows a better understanding on the beam-based structural discretization: It is important to say that modes are adopted in the

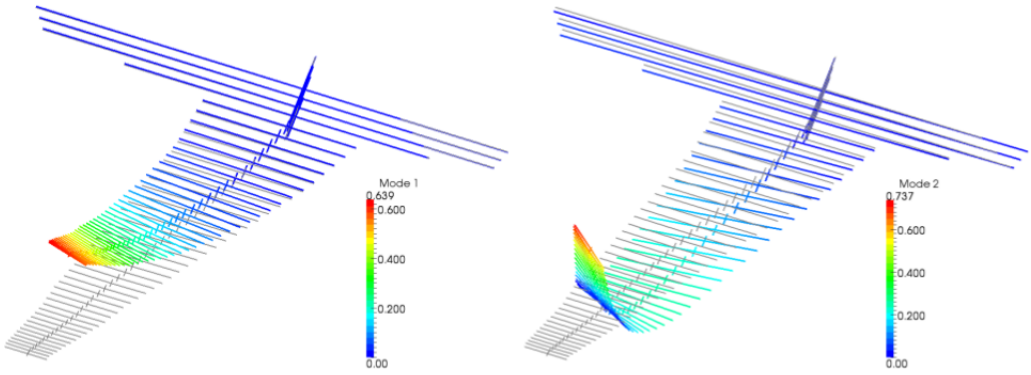


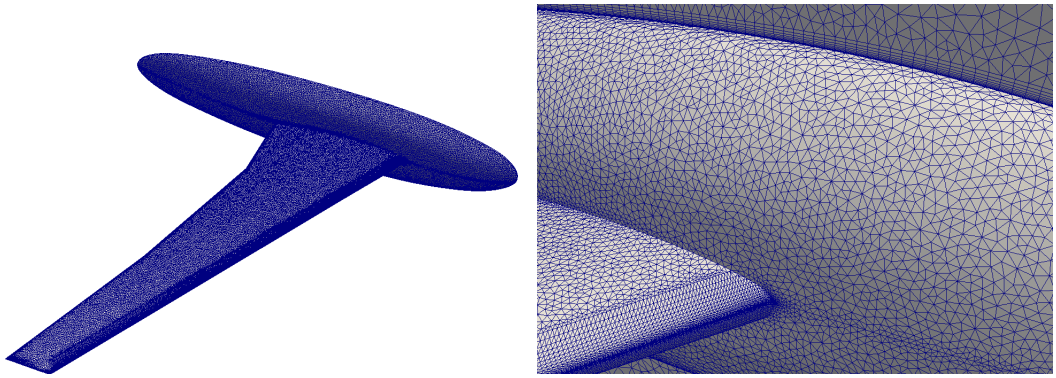
Figure 8.2: *Beam-based structural mesh discretization and shape of the first two modes (1^{st} (left) and 2^{nd} (right) bending).*

solver using a unitary mass normalization. This way it is possible to directly compare generalized displacements in order to guarantee the convergence of the modal representation of the structural behavior through the analysis of the elastic energy related to the participation of each single mode. This criterion will be also used for the trim of the Rotor 67 fan blade. However, in that case, few modes will be enough to correctly represent the structural behavior of the fan blade.

8.1.2 Aerodynamic model

After the description of the structural model it is time to discuss the aerodynamic mesh adopted for the discretization of the fluid domain and the solver settings adopted to con-

verge to a steady aeroelastic solutions. The trim validation is performed using AeroX with both Euler and RANS formulations, using SA turbulence model for viscous simulations. All the convergence acceleration techniques are employed. It is noted however that trim convergence is considered reached when both aerodynamic and modal residuals are below certain tolerances, since this investigation involves also the structural behavior of the elastic wing. It is possible to choose how many purely aerodynamic pseudo time iterations perform between two mesh updates. In this case a good trade-off is represented by 500 pseudo time iterations. This value allows good performances both for inviscid and viscous simulations with all the considered angles of attack. Pseudo time iterations are performed using EE formulation since no particular convergence problems have been encountered that would require specific smoothing capabilities provided by Runge–Kutta schemes. An hybrid mesh composed by a total number of $1.9 \cdot 10^6$ cells is adopted for the RANS simulations. This mesh is composed by 43% thetrahedra and 57% prisms. Thetrahedra are obviously used around the wing to discretize the boundary layer up to $y^+ \simeq 100$. This value is perfectly in the range supported by the automatic wall treatment. Since no complex phenomena, such as separations, should occur in the investigated conditions, this wall discretization allows to speed up computations while maintaining a good level of results accuracy. Prisms are used far from the wing up to the farfield with cells of increasing sizes in order to help smoothing residuals wherever is possible. This can be easily understood with figure 8.3 where the overall and the detailed views of the wing are showed: Since the mesh is un-



(a) Overall mesh view (b) Leading edge detailed view

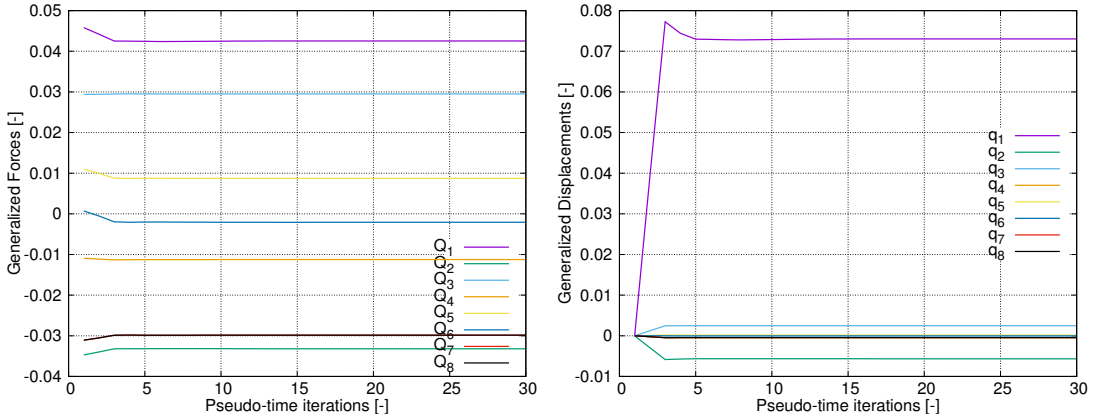
Figure 8.3: *HiReNASD mesh for RANS simulations, overall view of the body/wing part and detail view of the leading edge and boundary layer discretization.*

structured and hybrid, branch divergence occurs during assembly operations. However with the AMD 380X GPU it is still possible to achieve a time/iteration/cell of about $4.66 \cdot 10^{-6}$ seconds.

8.1.3 Trim analysis

Here the results concerning the static aeroealstic simulations are discussed. The focus is mainly on RANS results, though inviscid simulations are also performed to provide comparative data. First, it is worth to spend few words about the purely computational aspects of the problem. This case is simulated using the AMD 380X GPU. As

said 500 pseudo time iterations are used between two consecutive mesh updates. Basically, around 30 aeroelastic iterations are required to reach trim convergence for each angle of attack using RANS. This is equivalent to about $15 \cdot 10^3$ pseudo time purely aerodynamic iterations. Each trim simulation requires about 1500 seconds using the aforementioned GPU. It must be noted, however, that at each angle the trim simulation is started from a purely aerodynamic steady-state solution which is previously obtained with a purely aerodynamic steady-state simulation. Each purely aerodynamic solution requires about $20 \cdot 10^3$ iterations, that are translated to about 30 minutes of computational time. This basically means that a RANS trim solution for a single angle requires about less than one hour to be performed using the $\sim 2M$ cells mesh. Figure 8.4 shows the results for what concerns the convergence history of the first 4 generalized displacements and generalized forces with respect to pseudo time iterations for the RANS analysis at $\alpha = 1.5°$. It is possible to see that basically the solution is con-



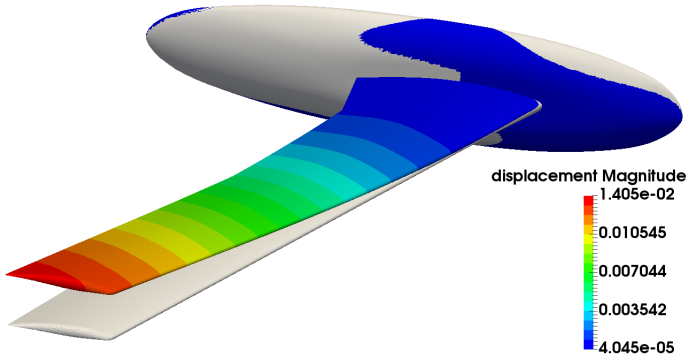
(a) Generalized forces

(b) Generalized displacements

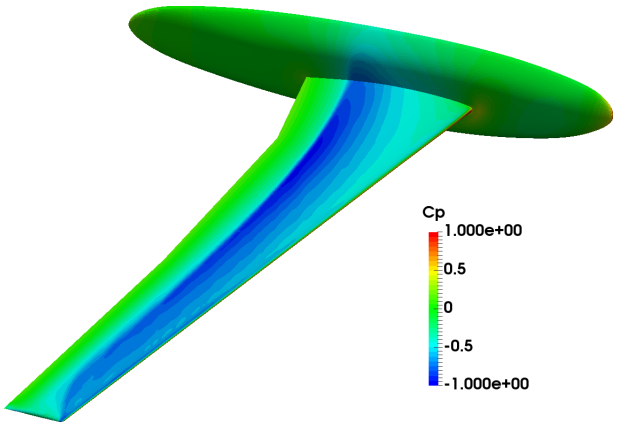
Figure 8.4: *Generalized forces and displacements convergence for HiReNASD trim at $\alpha = 1.5°$, RANS simulation.*

verged after just 15 aeroelastic iterations. However, in order to guarantee maximum results accuracy for the $\alpha = 1.5°$ case, a total number of 30 aeroelastic iterations are anyway performed. It is possible to see that the first three modes are characterized by the bigger generalized displacements. Considering that, as said, modes are scaled using a unitary mass normalization, it is possible to say that the first two modes account for almost all (over 95%) the elastic energy accumulated by the structure. Obviously the trend of generalized displacements and forces is characterized by steps. This is due to how the aeroelastic algorithm is implemented (between two successive jumps, purely aerodynamic iterations are performed). Furthermore, it is possible to see from figure 8.5(a) that the final shape of the wing basically reproduce the shape of the first mode (the one with the lowest frequency), i.e. the first bending mode. It must be noted that the displacements are opportunely magnified by a factor of $5\times$ in order to easily understand the structural deformation under aerodynamic loads. Figure 8.5(b) also shows the pressure coefficient C_P distribution over the wing. It is possible to see that a shock is present over the wing but smoothed due to viscous effects.

Figures 8.6 show at each of the aforementioned sections the C_P distribution. It is



(a) Wing shape, $5\times$ magnification

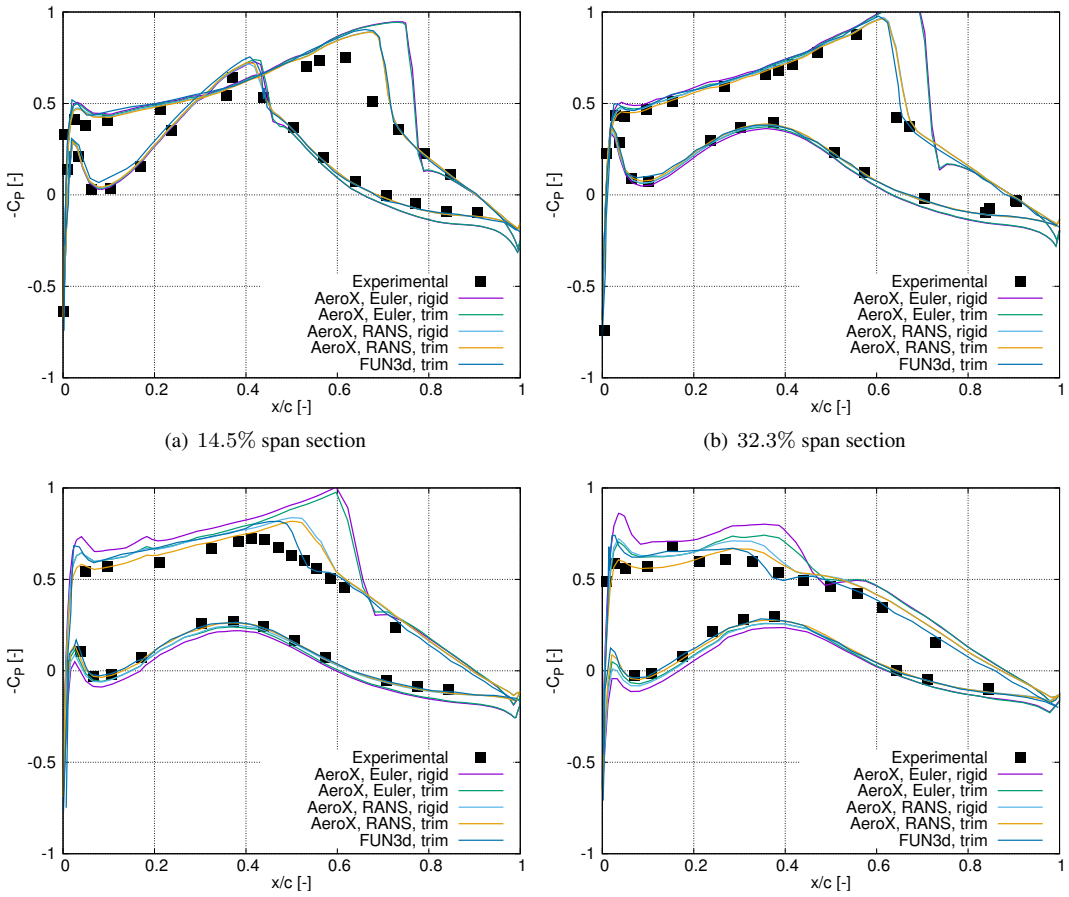


(b) C_P distribution

Figure 8.5: *HiReNASD wing trim at* $\alpha = 1.5°$*, RANS simulation.*

possible to see that the numerical solution provided by AeroX is in good agreement with both numerical and experimental data using RANS simulations. Important differences are instead obtained with inviscid simulations since it is possible to see that the shock location and the C_P jump are quite different from reference data. It is also important to notice that there are differences in C_P distributions between the purely aerodynamic and aeroelastic solutions. These are mainly encountered on the two sections with higher span values, the ones characterized by higher displacements. The results provided by AeroX are also compared with those provided by FUN3D [55]. It is possible to see that for the first two sections both solvers provide basically the same results for what concerns the viscous aeroelastic solution. Differences are found on the section at 65.5% span where FUN3D better reconstructs the shock location, though slightly overestimate the C_P near the leading edge on the upper surface. For what concerns the last section, again, results provided by AeroX seem to be slightly more in agreement with experimental data near the leading edge of the upper surface with respect to FUN3D, while for $x/c > 0.4$ the situation is reverted with FUN3D providing slightly more accurate results than AeroX.

Finally, figure 8.7 shows the vertical displacements of the considered wing point



(a) 14.5% span section

(b) 32.3% span section

(c) 65.5% span section

(d) 95.3% span section

Figure 8.6: *Pressure coefficient at different span sections for the HiReNASD test case, $\alpha = 1.5°$, RANS simulation.*

near the wing tip in a comparison between AeroX results, experimental data and numerical results from FUN3D. It is possible to see that inviscid results provided by AeroX overestimate the displacements with all the considered angles. This is due to the fact that, as it is possible to see from figures 8.6, Euler formulation predicts slightly different shocks locations, thus different pressure distributions, thus different wing loads and consequently different wing deformations. With the RANS simulations instead it is possible to see that results for the first angles are in very good agreement with both experimental data and reference numerical data. However the vertical displacements seem to be slightly underestimated with $\alpha = 4.5°$. Nonetheless it is possible to see that an underestimation in displacements is also obtained with reference numerical results though the error is slightly smaller.

8.2 AGARD 445.6 wing flutter

This is a well-know benchmark used to validate the capabilities of aeroelastic solvers to accurately predict transonic flutter conditions. The detailed description of this un-

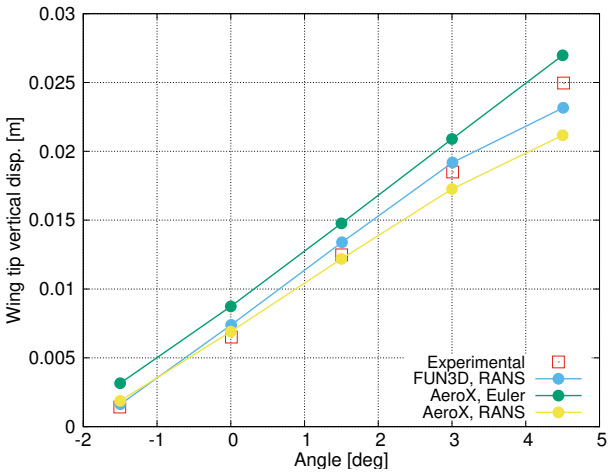


Figure 8.7: *Near-tip point vertical displacements comparison.*

steady aeroelastic test case can be found in [127]. Experimental data from the NASA Transonic Dynamics Tunnel (TDT) campaign can be found in [156]. Here the strategy adopted to compute flutter conditions is described in 2.2.3. Basically the idea is to firstly find trim conditions and then perform unsteady aeroelastic simulations by enforcing single modal shapes with a particular time law (rounded step). However, in this particular test case both the geometry and boundary conditions are symmetrical. Thus, the trim solution is basically equivalent to a simple steady-state rigid solution. From this solution the solver is restarted to perform unsteady aeroelastic simulations with enforced displacements. Flutter conditions, i.e. flutter speed V_F and flutter frequency ω_F, are computed using the root tracking strategy. More precisely, the Flutter Speed Index and Frequency Ratio are computed as follows:

$$FSI = \frac{V_F}{L_a \omega_t \sqrt{\mu}} \tag{8.1}$$

$$FR = \frac{V_F}{L_a \omega_t \sqrt{\mu}} \tag{8.2}$$

where L_a represents the reference aerodynamic length, ω_t is the first torsional frequency and μ is the mass ratio. Flutter is basically investigated as part of a post-processing procedure performed over numerical results computed from unsteady simulations. The entire procedure is repeated for different Mach numbers, i.e. $M_\infty = (0.678, 0.901, 0.960, 1.072, 1.140)$, covering the whole transonic regime. Inviscid and RANS results provided by AeroX will be compared with both experimental results obtained with an experimental campaign carried out by NASA Transonic Dynamics Tunnel in 1961 and numerical data available in literature. In particular, the reference numerical data is obtained from S^T [115], AeroFoam [127] and FUN3D [138] numerical solvers. This is an interesting test case since it shows the so-called transonic dip phenomenon, with the characteristic drop of flutter velocity in the transonic regime. Even though this is a well-know study case, under investigation for decades by numerical solvers, a general overestimation of the flutter speed is typically obtained, especially for Mach values over 1. Recently [138], a numerical study suggested also that with su-

personic Mach numbers it is important to consider more than 2 modes when performing inviscid simulations. The reason why will be showed later in results. The experimental setup for this case is depicted in figure 8.8 where it is possible to see that this is a sidewall-mounted, clean-wing configuration (weakened model No. 3) with a quarter-chord sweep angle $\Gamma_{c/4} = 45°$, a span $b = 0.762\,m$, a root chord $c_r = 0.558\,m$ and a taper ratio $\lambda = 0.66$. The wing is symmetrical and uses NACA 65A004 airfoil all over the span.

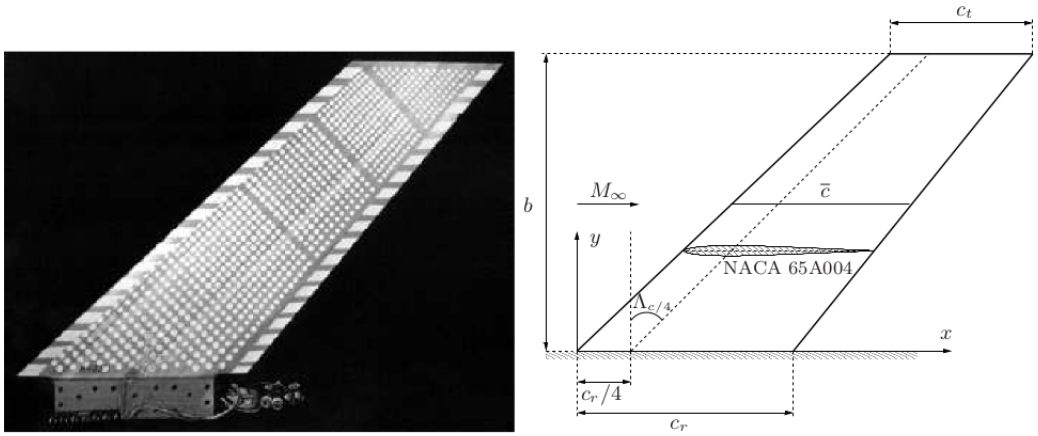


Figure 8.8: *Experimental setup for the AGARD 445.6 wing flutter investigation.*

8.2.1 Structural model

The structural model here adopted is taken from [127] where it is possible to find more details regarding the Finite Element discretization strategy and material properties. However it is worth to briefly describe few aspects here. The structural model is obtained through a Finite Element model that fits Ground Vibration Tests (GVT) literature data. The FE model is assembled in Code_Aster using 121 nodes and 200 homogeneous orthotropic triangular plate elements. The thickness distribution of the plate is opportunely tuned thanks to the use of Genetic Algorithms. This allows to obtain the modal frequencies showed in table 8.2 with a maximum relative error under 5%. The modal shapes corresponding to these frequencies are showed in figure 8.9

Mode ID	Exp. Frequency. (Hz)	FEM Frequency (Hz)	Error	Description
1	9.60	9.57	0.30%	1^{st} bending
2	38.17	39.28	2.90%	1^{st} torsion
3	48.35	50.35	4.15%	2^{nd} bending
4	91.54	93.63	2.10%	2^{nd} torsion

Table 8.2: *Modal frequencies and shapes of the first 8 modes of the HiReNASD wing.*

As for the HiReNASD wing the obtained modes are scaled through an unitary mass normalization.

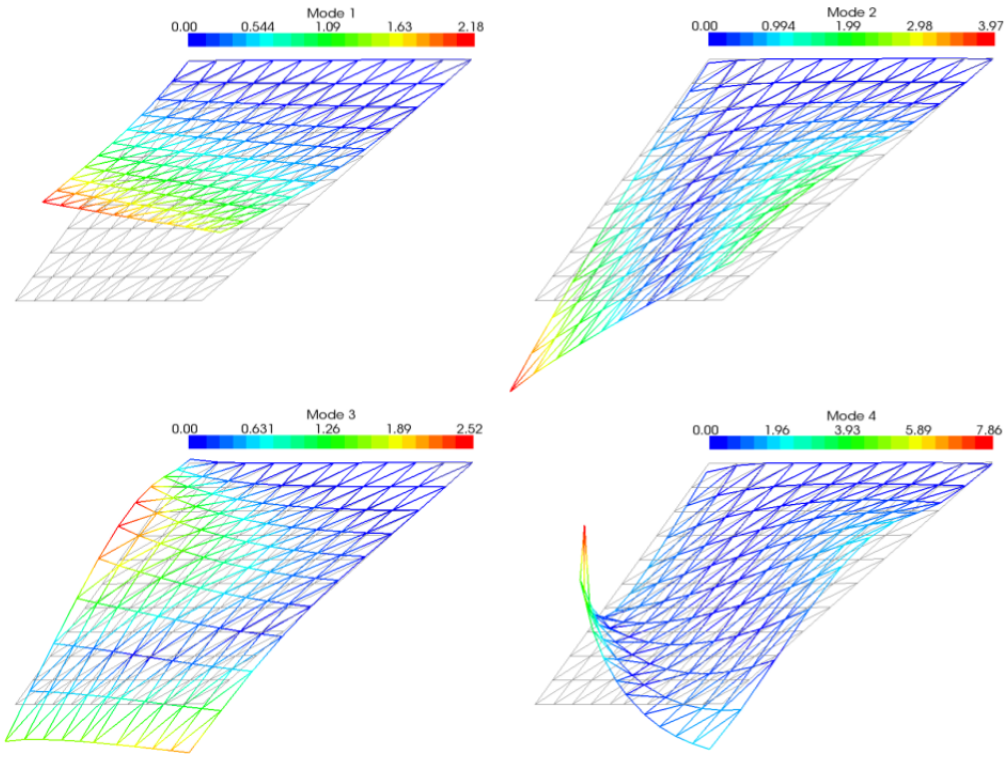
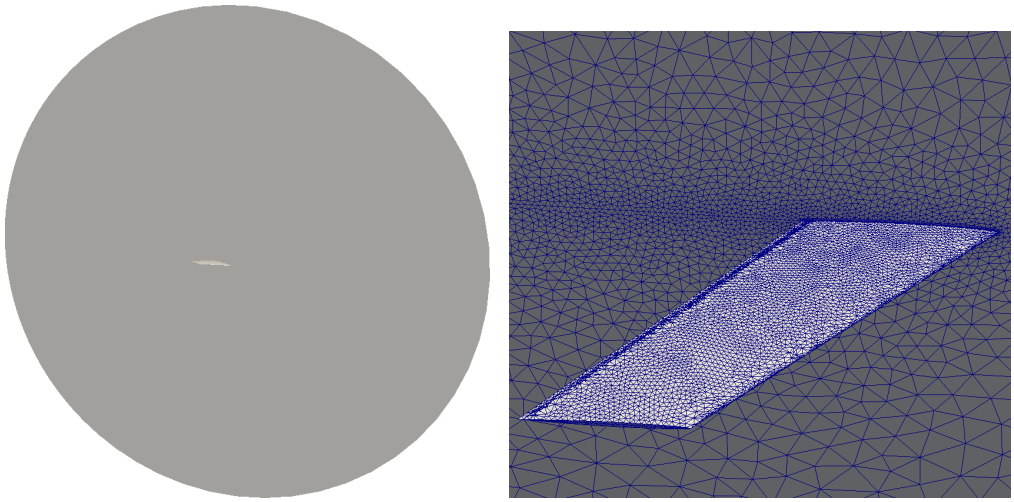


Figure 8.9: *First 4 modes for the AGARD 445.6 wing used for flutter analysis.*

8.2.2 Aerodynamic model

Here the details of the aerodynamic mesh adopted for the URANS simulations are described. However, all the simulations are also repeated using Euler formulation in order to check for possible differences due to viscous effects. Obviously the main differences between URANS and Euler meshes regard the presence of a boundary layer to refine the near wall region. This consequently lead to an higher number of cells for the URANS mesh. The unstructured hybrid mesh adopted for the URANS simulation is composed by a total number of $488 \cdot 10^3$ cells with 23% prisms, 69% tetrahedra and 8% hexahedra. The mesh is obtained using gmsh and OpenFOAM utilities in order to add the boundary layer discretization. An opportune near-wall boundary layer refinement is employed to guarantee y^+ values around 20 for the $M_\infty = 0.960$ steady-state solution. This is perfectly in the range of the automatic wall treatment through the blended log layer/viscous sublayer approach. Figures 8.10 show both the overall view of the computational domain and a detailed view of the wing discretization. It is possible to see that the computational domain is composed by an hemispherical farfield with the wing located in the central position and attached to a symmetry wall. The farfield is about 10 chords distant from the wing. Slip boundary conditions are enforced on the symmetry wall while characteristics-based automatic boundary conditions are employed over the farfiled. Finally non-slip boundary conditions are enforced over the wing for the (U)RANS simulations and slip boundary conditions for the Euler simulations. Again, in order to help smoothing residuals far from the wing, cells with increasing sizes are employed from the wing to the farfield. Wing walls are dis-



(a) Overall view of the computational domain (b) Near-wing detail view of the RANS mesh

Figure 8.10: *Overall view and near-wing detail view of the AGARD 445.6 aerodynamic mesh.*

cretized using $13 \cdot 10^3$ faces in order to guarantee the correct reconstruction of loads. The GPU used for the simulations is the AMD 380X. About $50 \cdot 10^3$ iterations are required to reach steady-state solution for the $M_\infty = 0.960$ case. This is translated to about 1200 seconds of simulation time with a time/iteration/cell of about $4.7 \cdot 10^{-8}$ seconds. All convergence accelerations techniques are active and viscous computations are performed every pseudo time iteration. RANS simulations are performed using SA turbulence model which is perfect for this kind of analysis since it is specifically designed for aeronautical cases with attached flows. Unsteady simulations are carried out with the described DTS formulation. An aeroelastic interface is assembled between the aerodynamic and structural meshes using RBF. Modal shapes are gradually enforced with a blended step time law thanks to the modal framework. IDW is used for mesh deformation and thanks to the optimized implementation, the computational time spent at each physical time step for the mesh update is basically negligible with respect to the effort required for purely aerodynamic convergence. One unsteady RANS simulation, corresponding to the introduction of a single modal shape requires about 4300 seconds of simulation time on the AMD 380X GPU. It must be noted that since 4 modes are required for each Mach number and 5 Mach numbers are investigated, a total computational time of about 24 hours is required to perform a complete compressible RANS flutter investigation with this single cheap gaming GPU. About 13 hours are instead required for the entire Euler-based flutter investigation.

8.2.3 Trim analysis

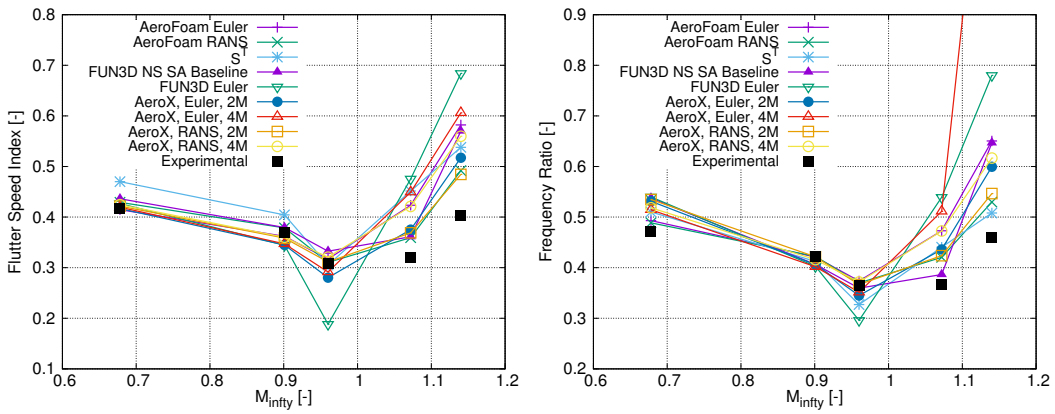
As mentioned, since the wing geometry (NACA 65A004 sections) and asymptotic conditions ($\alpha = 0°$) are symmetrical, there is no need to perform a true aeroelastic steady-state simulation. The reference equilibrium solution does not involve the deformation of the wing structure. The purely aerodynamic steady-state solution is adopted as initial conditions for the subsequent unsteady simulations.

8.2.4 Flutter Analysis

As explained in [127] it is worth to perform a dynamic linearity check when choosing the blended step amplitude. The displacements in fact have to be opportunely small in order to avoid triggering non-linear behaviors that would corrupt the small perturbations hypothesis but at the same time they have to be big enough to be clearly higher than the numerical errors threshold, especially with single precision executions. The blended step time laws are here chosen following what explained in 2.2.2 and following the suggestions provided in [127]. The idea is basically to choose a modal shape (e.g the first bending mode), perform a simulation with a given input amplitude, store the resulting generalized forces and repeat the simulation doubling the input amplitude. If the ratio between the generalized force amplitudes is around two then the linear behavior is respected with the tried amplitudes. Using a threshold value of $\varepsilon = \tan 1^\circ$ appears to be a correct choice to satisfy the aforementioned requirements.

It is now time to discuss the aeroelastic unsteady results provided by AeroX solver in order to proceed with the validation of the flutter prediction capabilities.

Figures 8.11 shows the results for the previously defined Flutter Speed Index and Frequency Ratio parameters within a comparison between AeroX executed in viscous mode with SA turbulence model, AeroX inviscid, S^T full potential solver, AeroFoam solver in viscous and inviscid mode, FUN3D inviscid and viscous with SA turbulence model and experimental results. For what concerns AeroX both results obtained considering just the first 2 modes (first bending and first torsion) and all the 4 modes are showed. In general it is possible to see that, as expected, viscous simulations with



(a) Flutter Speed Index for AGARD 445.6 wing

(b) Frequency Ratio for AGARD 445.6 wing

Figure 8.11: *Flutter indexes for AGARD 445.6 wing at different Mach number.*

SA turbulence model provide better results with respect to inviscid simulations. This is true for both AeroX and FUN3D solvers. For Mach numbers smaller than 1 basically all solvers provide very good results for the Flutter Speed Index. The situation slightly changes for Mach numbers over 1 where a general overestimation of the same index seems to be common between all the numerical solutions. Though all numerical solutions predicts the aforementioned transonic dip phenomenon, within supersonic conditions viscous solutions are clearly more accurate than the inviscid counterparts. For what concerns the Frequency Ratio, a general slight overestimation of this index

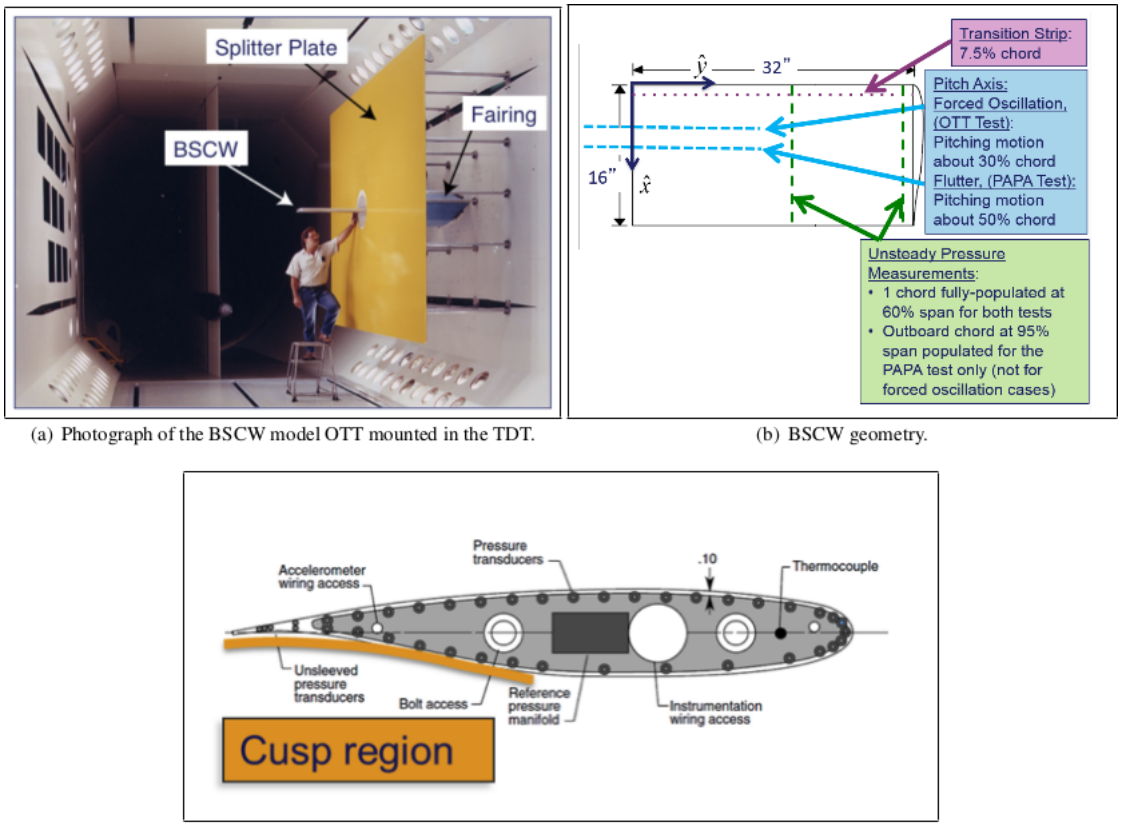
is encountered for the solution with the smallest Mach number. This is true for both viscous and inviscid solutions provided by AeroX. Very good results are obtained with $M_\infty = 0.901$ and $M_\infty = 0.960$ both with Euler and SA by AeroX. Similarly to what happens with the Frequency Speed Index, for Mach numbers over 1 a general overestimation of the Frequency Ratio is encountered. Nonetheless it is possible to see that both Euler and SA solutions provided by AeroX are in agreement with experimental data and are of comparable or better accuracy with respect to the reference numerical solutions. In general it is worth to notice the good results provided by the full potential solver S^T, which are in general comparable to RANS solutions.

As it is possible to see from the results provided by AeroX and other solvers, this case still represents a challenge for numerical aeroelastic solvers. Differences are obtained considering just 2 or all the 4 modes using the described flutter prediction procedure that involves the assembly of aerodynamic transfer function matrices. A recent paper [138] opened a new question for what concerns the inviscid case with $M_\infty = 1.140$. The paper explains that if more than the first two modes are investigated it appears that for these particular conditions the $3rd$ mode is the responsible for flutter. However, in the literature many inviscid simulations are carried out with just the first two modes, providing misleading results. With AeroX it appears that considering URANS and Euler simulations with 2 modes all Mach numbers suggest the flutter instability of the first aeroelastic mode. This is also true when considering 4 modes with RANS. However when considering 4 modes with Euler it appears that the case with $M_\infty = 1.140$ is characterized by an instability of the third mode which is exactly what suggested by the reference. In particular, both the first and the third modes become unstable, but the third mode becomes unstable at a smaller dynamic pressure. This is the reason for the differences in the Frequency Ratio between the Euler solution with 2 modes and the Euler solution with 4 modes, since the third aeroelastic mode features bigger frequencies than the first mode. In order to ease the view of the Frequency Ratio trend the last point of this simulation is not depicted. However its value is 1.43 which is a reasonable value considering that the third structural mode (at null speed) frequency is about $5\times$ higher than the first mode.

8.3 2nd Aeroelastic Prediction Workshop wing flutter

The 2nd Aeroelastic Prediction Workshop represents the most recent effort to asses the state-of-the-art of modern computational methods in predicting unsteady flow fields and aeroelastic response [81]. This workshop proposes different cases to be investigated, ranging from unsteady unforced, forced oscillations to flutter prediction analyses. The geometry under investigation is represented by the Benchmark Supercritical Wing (BSCW). The experimental setup is showed in figure 8.12. Details regarding the experimental setup are available in the official AePW2 paper [81].

As the paper suggests, the aeroelastic unsteady response of the BSCW wing is investigated in conditions such that the influence of the separated flow is considered to be minimal, yet a shock is still present. Thus, even if turbulence models are not excessively stressed due to complex phenomena like separations are recirculation bubbles, an accurate reconstruction of the interaction between viscous and compressible effects is required. Consequently this is a perfect benchmark to validate the flutter prediction



(a) Photograph of the BSCW model OTT mounted in the TDT.

(b) BSCW geometry.

(c) Cross-sectional view of the SC(2)-0414 airfoil, with BSCW instrumentation.

Figure 8.12: *Experimental setup for the BSCW wing of the 2nd Aeroelastic Prediction Workshop.*

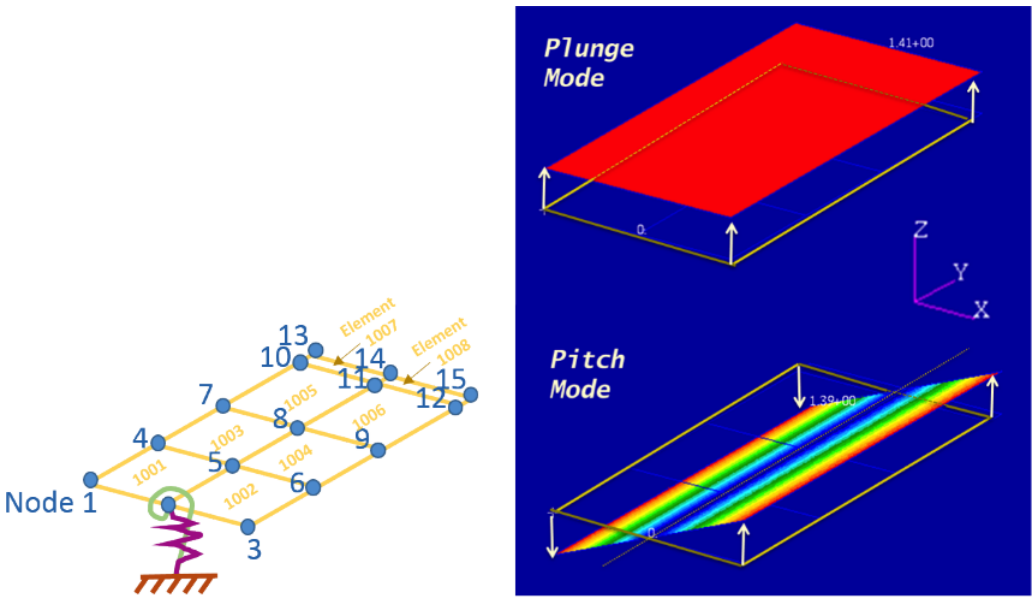
capabilities of aeroelastic compressible URANS solvers such as AeroX. As mentioned in the AePW2 paper, different kind of investigations are available. Here the choice is focused on flutter prediction. The strategy here adopted to find the flutter dynamic pressure is different from what showed for the AGARD 445.6 wing. Here the strategy explained in 2.2.10 is instead adopted. The idea is basically to perform an unsteady simulation starting from steady-state trimmed conditions, introducing a small disturbance and then letting the wing freely move under aerodynamic loads. The time history of the pitch and plunge d.o.f. is stored and post-processed in order to check for unstable behaviors, i.e. flutter. In particular, as explained in the paper, the idea is to find the dynamic pressure that leads to the unstable behavior of the wing pitch oscillations. This can be easily done by performing multiple simulations with different asymptotic dynamic pressure values q_∞, checking the damping g value of the pitch d.o.f. and through a bisection iterative algorithm change the dynamic pressure up to the value related to $g = 0$, i.e. flutter conditions. At the time of writing, preliminary reference numerical results, provided by other teams participating to the workshop, can be found at [82].

The experiments on the BSCW wing were conducted using two different mount systems: PAPA and OTT. The Oscillating TurnTable (OTT) was used to provide forced pitch oscillation data, while the flexible Pitch And Plunge Apparatus (PAPA) mount system was used to provide aeroelastic results. The BSCW wing is mounted on a splitter plate with an opportune offset from the wind tunnel wall in order to minimize

the effect of the wind tunnel wall boundary layer. Here the focus is on flutter prediction, thus the focus is on results related to the PAPA configuration. The flexible PAPA mount system is used to provide the BSCW wing pitch and plunge d.o.f. with specific stiffness values. Experimental data is available at experimental flutter conditions. From figure 8.12 it is possible to see that the BSCW wing has a span $b = 0.8128\,m$, a chord $c = 0.4064\,m$, null sweep angle and a rounded tip. The reference area is $A = 0.3303\,m^2$. The wing is composed by NASA SC(2)-0414 airfoil all over the span. For the PAPA configuration the pitch axis is located at 50% of the chord and pressure sensors are located at 60% and at 95% of the span for unsteady pressure measurements. The focus is on the Case 2 of the workshop, which is characterized by an asymptotic Mach number $M_\infty = 0.74$, a Reynolds number based on the chord of $Re = 4.450 \cdot 10^6$ and an angle of attack $\alpha = 0.0°$. The gas adopted for experiments is not air but the R-12 gas ($\gamma = 1.136$, $R = 68.765\,\frac{J}{Kg\,K}$). As the AePW2 paper suggests, these conditions should not lead to complex shock-induced separations. Other details regarding the asymptotic conditions of the tests are available in 8.3 and here are not reported for brevity.

8.3.1 Structural model

As the paper suggests the wing was designed with the goal of being (nearly) rigid. The first modes of the BSCW wing are the following: a spanwise first bending mode with a frequency of $24.1\,Hz$, an in-plane first bending mode with a frequency of $27.0\,Hz$ and the first torsion mode with a frequency of $79.9\,Hz$. Despite these wing modes, in numerical simulations the wing itself is supposed to be perfectly rigid, leaving all the deformability to the flexible mount system. This basically means that two rigid d.o.f., wing pitch and wing plunge, with a certain stiffness and inertial properties, are adopted to represent the structural behavior of the aeroelastic system. The FE model adopted by the authors of the AePW2 is showed in figure 8.13. Basically, the FEM



(a) Finite element model: elements and grid points.

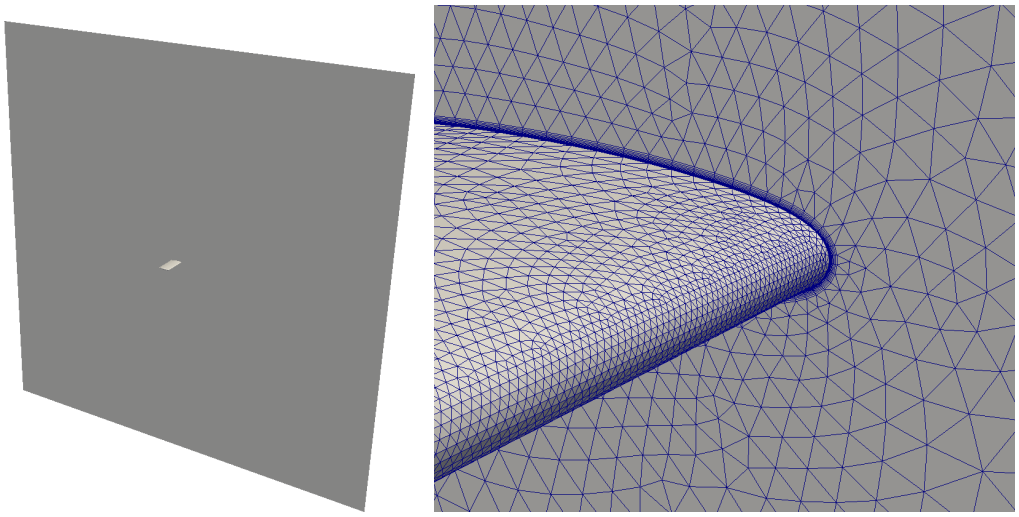
(b) Structural dynamic model mode shapes.

Figure 8.13: *Structural model adopted for the flutter prediction of the BSCW wing.*

model is characterized by a nearly rigid flat plate with the same size of the BSCW wing connected to a fixed point with two simulated springs at the pitch rotation axis. As previously mentioned for the PAPA configuration the pitch axis is located at 50% of the chord. Point 2 is located at wing root at the axis of rotation and allows only pitch and plunge displacements. This basically means that with AeroX it is possible to represent the structural behavior with two rigid d.o.f. characterized by specific inertial and stiffness values. In particular, the pitch movement is characterized by an inertia of $2.777\,slug - ft^2$ ($3.765\,kg\,m^2$) and a stiffness of $2964\,lbf - ft/rad$ ($4018.44\,N\,m/rad$). For the plunge d.o.f. the mass is $6.0237\,slugs$ ($87.91\,kg$) while the stiffness is $2637\,lbf/ft$ ($38484.12\,N/m$). This leads to a plunge mode frequency of $3.33\,Hz$ and a pitch mode frequency of $5.20\,Hz$. Since the d.o.f. are considered rigid, there is no need of an aeroelastic interface for this investigation as aerodynamic forces and moments, computed through an integration of normal and tangential loads over the wing surface, can be directly used to compute the structural response of the pitch and plunge d.o.f. at each physical time step.

8.3.2 Aerodynamic model

Different meshes are available on the AePW2 NASA website [12] with different discretization levels and total number of cells. All meshes are characterized by different millions of cells. However, for this work two meshes are built from scratch using the BSCW geometry with two different discretization levels and total number of cells. In this and subsequent subsections these two meshes will be referred as "coarse" and "fine" mesh. Figures 8.14 show the overall view of the computational domain alongside with a detail view of the mesh discretization near the leading edge regarding the fine mesh. It is possible to see the presence of a boundary layer discretization in order to cor-

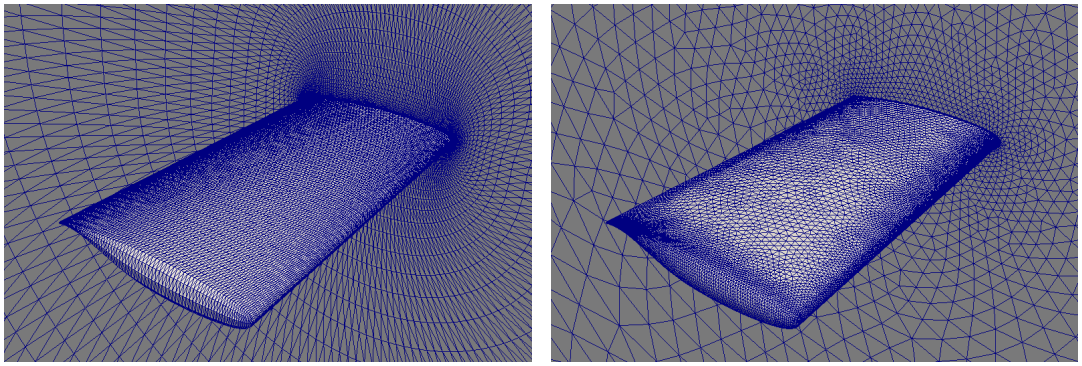


(a) Overall view of the computational domain (b) Mesh discretization detail near the leading edge

Figure 8.14: *Computational domain for the BSCW wing flutter analysis.*

rectly refine the near-wall region for (U)RANS computations. From the overall view it is possible to see that the wing is attached to a symmetry wall. While over the wing

non-slip boundary conditions are employed with the automatic wall treatment, on the symmetry wall slip boundary conditions are instead adopted. It must be noted that, as for the meshes available from NASA, there is no specific discretization of the wind tunnel wall and the splitter plate. Finally, a farfield, opportunely distant from the wing, is subjected to characteristics-based boundary conditions. The fine mesh, generated with HyperMesh, features a total number of $1.1 \cdot 10^6$ cells. It is an hybrid mesh with 32% prisms and 68% tetrahedra, the latter mainly used to discretize the boundary layer. It is thus an hybrid unstructured mesh that allows a time/iteration/cell of $4.07 \cdot 10^{-8}$ seconds on the AMD 380X GPU adopted for the simulations. The near wall wing discretization allows y^+ values around $5 - 30$, perfectly in the range of the blended approach. As already mentioned there should be no complex phenomena like separations, thus the near-wall discretization adopted is sufficient to guarantee both results accuracy and computational efficiency. The coarse mesh is instead composed by $0.5 \cdot 10^6$ cells and with the AMD 380X a time/iteration/cell value of $3.9 \cdot 10^-8$ seconds is obtained. The coarse mesh is generated with Pointwise and the near-wall refinement allows y^+ values around 100. All convergence acceleration techniques are active and, again, the domain is discretized with cell sizes opportunely chosen to exploit LTS. Finally figures 8.15 shows the different wing discretization levels provided by the coarse and fine mesh. In particular it is possible to see that the rounded tip is better reconstructed with the fine mesh. For this investigation AeroX is executed using the usual Roe scheme with high



(a) Coarse mesh (b) Fine mesh

Figure 8.15: *Wing discretization comparison between coarse and fine meshes.*

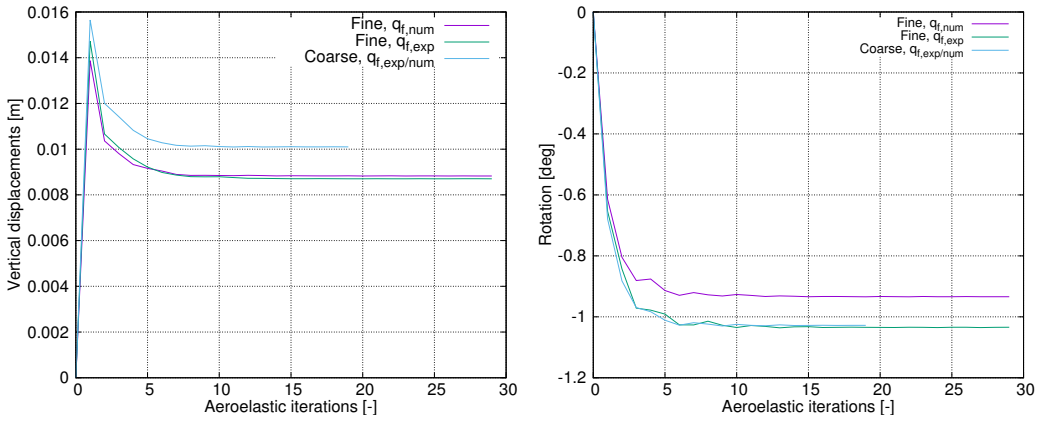
resolution and SA turbulence model.

8.3.3 Trim results

The first step for the analysis is the computation of the steady-state aeroelastic solution. This will be used as the initial condition for the subsequent unsteady simulation. In particular, the purpose of the trim simulation is to provide the initial pitch and plunge displacements from which start the flutter investigation. Another strategy would be to start the unsteady simulation directly from initial guess conditions, let the wing converge to a steady aeroelastic configuration and then start the flutter investigation. However this strategy would lead to an unnecessary computational effort since the trim solution can be directly found using the steady-state aeroelastic procedure already adopted for the

HiReNASD investigation. This strategy is also supported by the fact that, as explained in the AePW2 paper, the asymptotic conditions for the flutter investigation should not lead to complex and intrinsically unsteady phenomena such as shock induced separations.

Results for what concerns trim are related to the steady-state values of the wing pitch angle and vertical displacement. The convergence history of both is represented in figures 8.16 where it is possible to see that no particular convergence problems, such as oscillating behaviors, are obtained. Again this is what can be expected since the asymptotic conditions should not trigger complex phenomena like shock induced separations. From the figures it is possible to see that when the equilibrium is reached between



(a) Wing vertical displacement convergence history

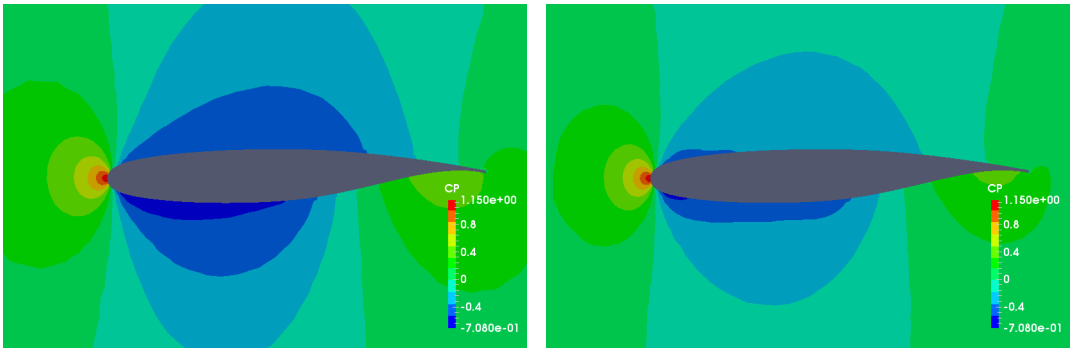
(b) Wing pitch rotation convergence history

Figure 8.16: *Wing trim displacements convergence history.*

aerodynamic loads and spring forces the wing is characterized by a positive vertical displacement and a negative wing rotation (leading edge down, trailing edge up). It must be noted that two results are showed for what concerns the fine mesh, one for the speed related to numerical flutter conditions obtained with this mesh ($U_\infty = 112.0\,m/s$) and one for the experimental flutter conditions ($U_\infty = 114.51\,m/s$). Since with the coarse mesh both numerical and experimental flutter conditions share the same asymptotic dynamic pressure (the differences are negligible), just one result is showed. It is possible to see that at experimental flutter conditions both meshes are in agreement for the trim pitch angle. However with the same experimental flutter conditions slightly different results are obtained with the two mesh for what concerns the wing vertical displacement. On the AePW2 paper preliminary numerical results at experimental flutter conditions suggest a steady-state aeroelastic solution with a pitch angle of about -1° which is in agreement with AeroX results with both meshes.

Figures 8.18 show the C_P field around the wing at the 60% and 95% span sections, the same sections that will be used to post-process unsteady data. It is possible to see that no particular complex phenomena occurs in the flow field.

Table 8.3 shows the lift, drag and moment coefficients obtained with the fine and coarse meshes with numerical and experimental flutter asymptotic conditions. In the table both coefficients obtained with a simple steady-state (purely aerodynamic) solution and trim (aeroelastic) solution are showed. From the table it is possible to see



(a) 60% span (b) 95% span

Figure 8.17: *Pressure coefficient field at different spans, fine mesh, experimental flutter conditions.*

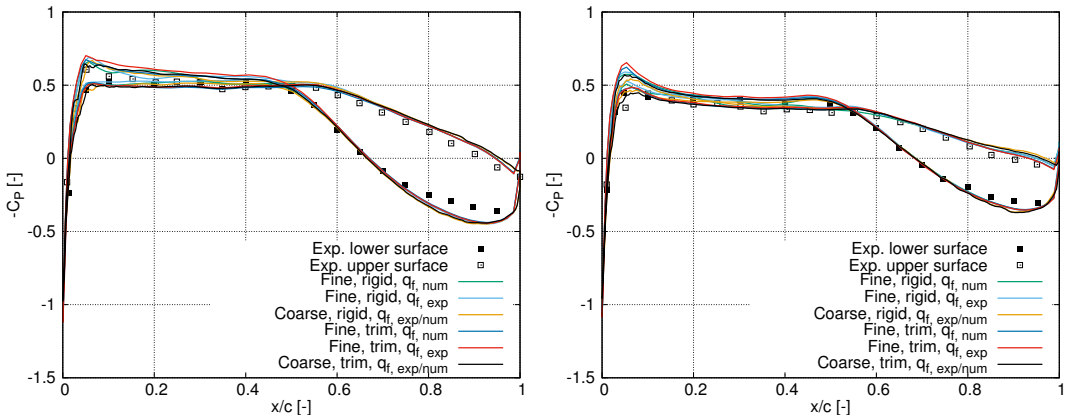
Mesh	Speed (m/s)	Trim	C_L	C_D	$C_{M,c/2}$
Fine	112.00	YES	0.1330	0.01549	−0.06308
Fine	112.00	NO	0.2101	0.01736	−0.04148
Fine	114.51	YES	0.1254	0.01534	−0.06691
Fine	114.51	NO	0.2118	0.01788	−0.04315
Coarse	114.51	YES	0.1456	0.01553	−0.06644
Coarse	114.51	NO	0.2256	0.01809	−0.04376

Table 8.3: *Trim coefficients with different meshes and speeds.*

that, as expected, the trim solution provide different coefficients since the pitch angle is about $-1°$, thus different from the completely rigid steady-state case (which is considered null).

Finally it is worth to say that trim convergence is reached in about 20 aeroelastic iterations for the coarse mesh and about 30 aeroelastic iterations for the fine mesh. Considering also the steady-state solution from which the trim procedure is started, this is translated in a total computational time of 638 seconds for the coarse mesh and 1701 seconds for the fine mesh using the AMD 380X GPU. Between each aeroelastic iteration a total number of 500 pseudo time iterations are performed.

Finally a comparison can be performed between AeroX results and experimental results for what concerns the pressure coefficient distribution over the wing at the two investigated sections. Figures 8.18 show the aforementioned quantities at the 60% and 90% span sections respectively. It is possible to see that numerical results are in good agreement with experimental data. All the solutions with different meshes and asymptotic conditions are quite similar, both when considering the steady rigid solution and the steady aeroelastic solution. The main differences between numerical and experimental results are obtained near the trailing edge over the lower surface for both sections. This behavior is encountered also with the preliminary results provided by other research groups [82].



(a) 60% span (b) 95% span

Figure 8.18: *Pressure coefficient field at different span locations, steady rigid and aeroelastic solutions, fine and coarse meshes, numerical and experimental flutter dynamic pressures.*

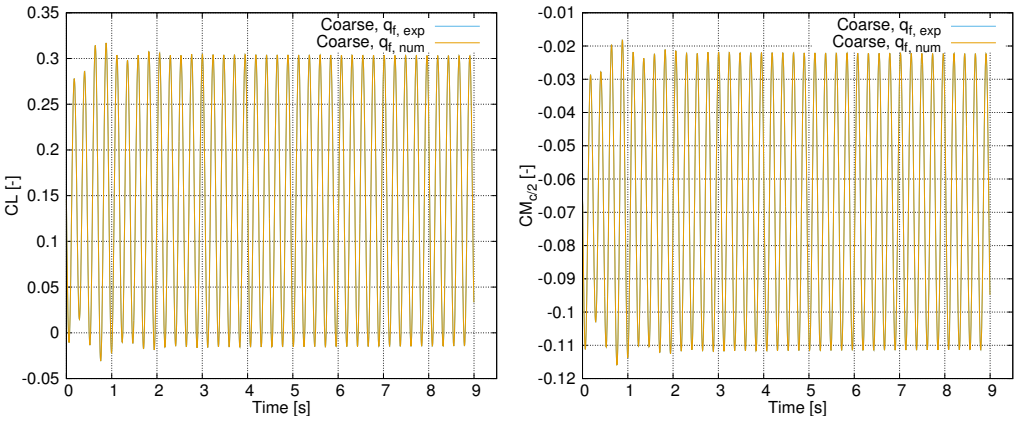
8.3.4 Flutter results

The AePW2 paper suggests to investigate what happens at both the experimental flutter dynamic pressure $q_{f,exp} = 168.8\,psf = 8082.19\,Pa$ and the predicted numerical flutter dynamic pressure $q_{f,num}$ (to be found). The more these two value are similar, the more accurate flutter prediction is provided by the numerical solver. AeroX is restarted from the just computed trim solution with the DTS formulation activated. In order to trigger oscillations, the solver is restarted from trim conditions using the computed pitch and plunge displacements but introducing an opportune perturbation on the initial pitch velocity of $-1 rad/s$. This provides energy in the system and leads to pitch and plunge oscillations. The focus here is to check if pitch oscillations are subsequently damped ($g > 0$, $q_{num} < q_{f,num}$, stable), sustained with constant amplitude ($g = 0$, $q_{num} = q_{f,num}$, neutrally stable, i.e. flutter), or divergent ($g < 0$, $q_{num} > q_{f,num}$, unstable). As the paper suggests, the focus of the investigation is on the behavior of pitch oscillations only.

An important point for this investigation is represented by the choice of the physical time step Δt. In fact, as noticed by other research groups it seems that there is a dependency of the aeroelastic damping from the physical time step value. This basically means that changing the physical time step leads to a change in the numerical predicted value of flutter dynamic pressure $q_{f,num}$. In particular, it seems that with higher physical time steps, higher damping values are obtained, pushing the numerical flutter conditions to higher dynamic pressure values. A good value for Δt, suggested by one of the groups, is represented by $\Delta t = 2.4 \cdot 10^{-4}\,s$. This value is here adopted with both the coarse and fine meshes for the unsteady simulations. The unsteady simulation is performed for about 9.0 seconds of physical time, which represents a good trade-off between computational effort and information content. In fact it must be noted that for about 3.0 seconds of physical time the solution is influenced by the initial transient due to the perturbation. After that, it is possible to collect the useful data to be used for post-processing purposes.

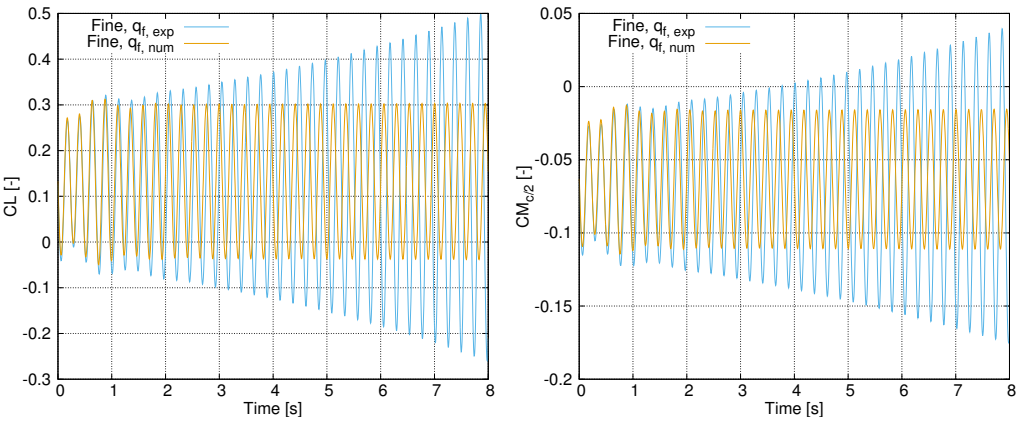
As already mentioned when discussing the trim results, the speed related to the

dynamic pressure at experimental flutter conditions is $U_{f,exp} = 114.51\,m/s$. Using AeroX it was possible to estimate a numerical flutter speed on $U_{f,num} = 112.0\,m/s$ with the fine mesh. For what concerns the coarse mesh instead, using experimental flutter asymptotic conditions provides a damping value on the pitch oscillations that is basically negligible. Thus experimental flutter conditions are also considered numerical flutter conditions for the coarse mesh. Obviously it would be expected better accuracy with the fine mesh. As will be shown in the FRF (Frequency Response Function) this is more likely to be a coincidence due to balancing errors. It is worth to briefly show the results for what concerns the unsteady behavior of the wing displacements and force coefficients. Figures 8.20 and 8.19 show the values of C_L and C_M obtained with the fine and coarse meshes with the different asymptotic dynamic pressures. It is possible to see that the coefficients oscillates around a mean value represented by the trim value.



(a) C_L (b) $C_{M,c/2}$

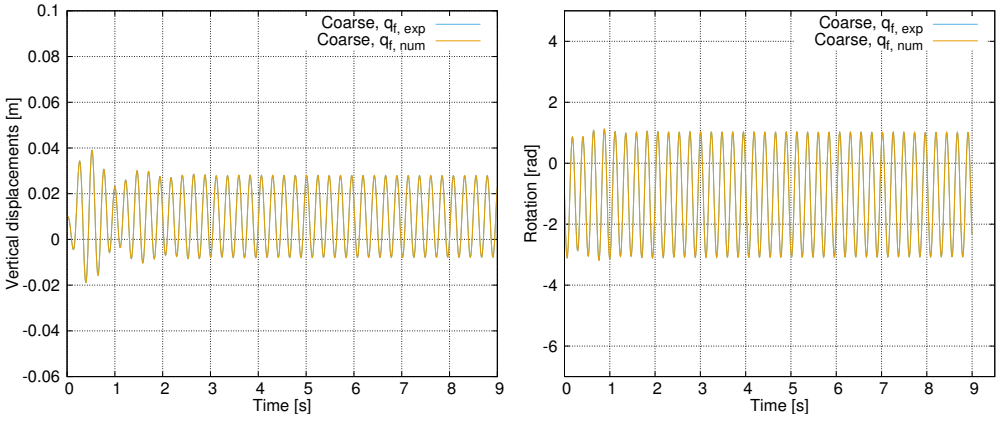
Figure 8.19: *Evolution of lift and pitch moment coefficients through time, coarse mesh.*



(a) C_L (b) $C_{M,c/2}$

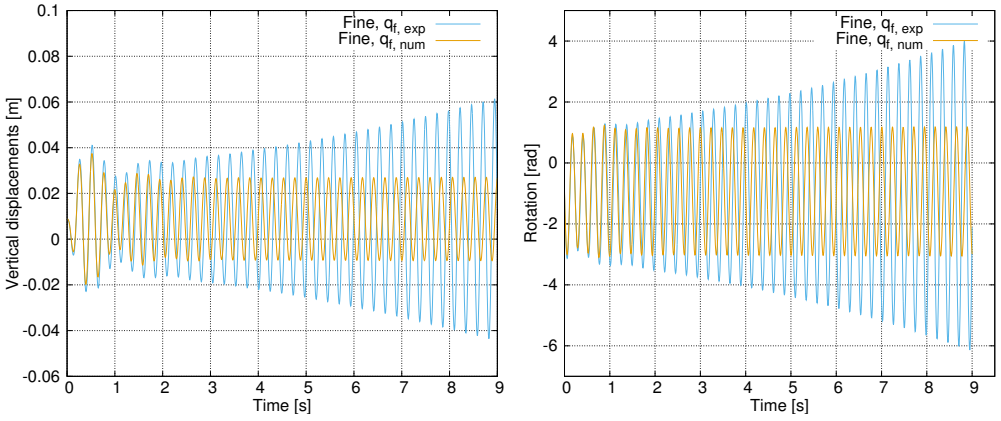
Figure 8.20: *Evolution of lift and pitch moment coefficients through time, fine mesh.*

Figures 8.22 and 8.21 show instead the unsteady values of the wing pitch rotation alongside the wing vertical displacement. Again it is possilbe to see that the wing basically oscillates around trim values. The measured experimental flutter frequency



(a) Vertical displacement (b) Pitch rotation

Figure 8.21: *Evolution of wing pitch and plunge degrees of freedom through time, coarse mesh.*



(a) Vertical displacement (b) Pitch rotation

Figure 8.22: *Evolution of wing pitch and plunge degrees of freedom through time, fine mesh.*

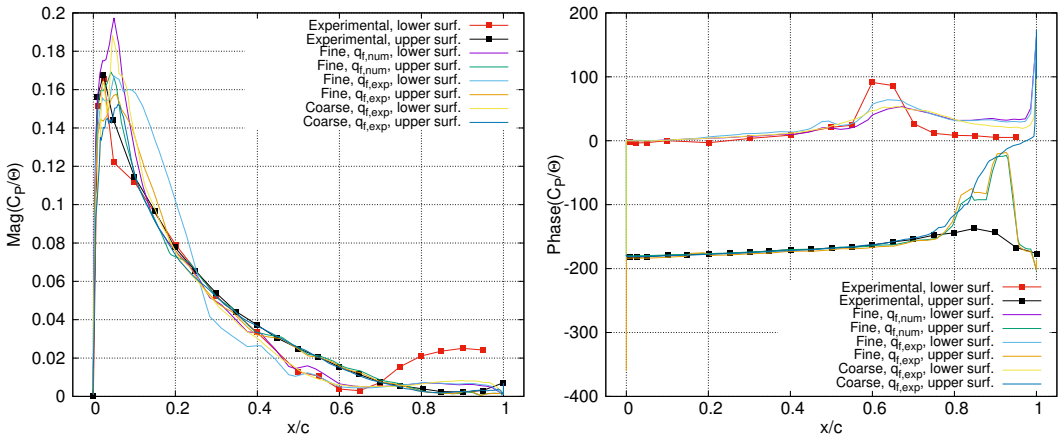
is $4.3\,Hz$. With AeroX the flutter frequency obtained at numerical flutter conditions are $4.286\,Hz$ for the fine mesh and $4.288\,Hz$ for the coarse mesh respectively. Thus the flutter frequency obtained with both meshes is in agreement with experimental data.

As said the oscillations are triggered by introducing a perturbation on the initial pitch velocity of $-1.0\,rad/s$. This is done since it seems a good value to maintain the small oscillations hypothesis and at the same time to introduce enough energy in the aeroelastic system. This is also a value that seems to be in accordance to what used in the original AePW2 overview paper 8.3 from the available figures. Obviously as the amplitude of the oscillations are related to the initial pitch velocity and this value can be arbitrary chosen, in order to obtain a meaningful comparison with other research groups, the idea is a normalization of the output oscillations based on the pitch angle.

In particular the comparison between experimental data and numerical data provided by other research groups is performed using the measure suggested in the overview paper, defined by 8.3.

$$\left| \frac{C_P}{\Theta} (f^*) \right| vs. \frac{x}{c} \tag{8.3}$$

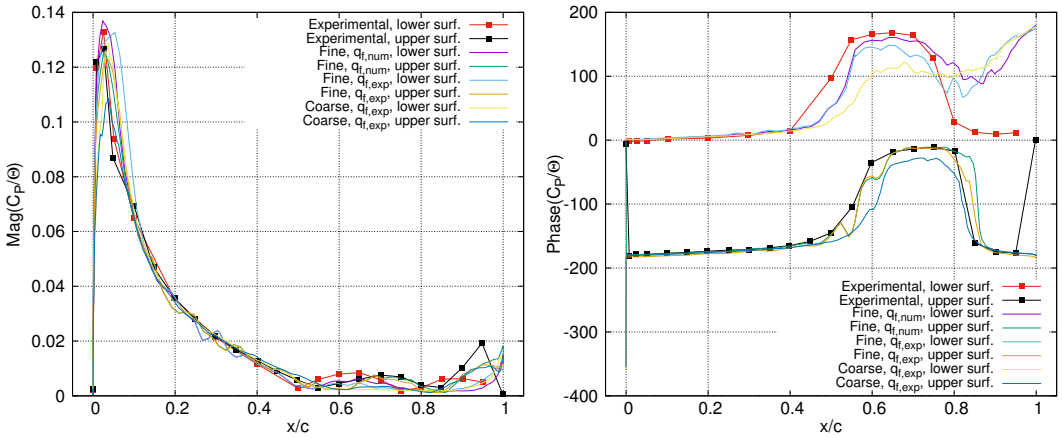
The idea is basically to save at each physical time step the C_P value over the two span sections (60% and 95%) and normalize these with the current pitch angle value. The obtained value is then processed using the FFT algorithm in order to provide for each x/c point an imaginary and real values that will be converted in magnitude and phase. Magnitude and phases will be then directly compared with experimental and numerical reference data from other research groups. Figures 8.23 show the comparison, at 60% span section, between experimental data and what provided by AeroX with the two different meshes and dynamic pressures (experimental and numerical flutter conditions). In figure 8.24 instead, the same is repeated for the 95% span location. In



(a) Magnitude (b) Phase

Figure 8.23: *Magnitude and phase of the indicator,* 60% *span.*

general it is possible to see that a good agreement is obtained with AeroX with both meshes and experimental data. In particular, the best results are obtained with the fine mesh at the computational flutter dynamic pressure. For what concerns the magnitude of the indicator 8.3, numerical results are in agreement with experimental data, except for a slight underestimation near the trailing edge for what concerns the lower surface at 60% span. For what concerns the phase, it is possible to see that at 60% span numerical results are in good agreement over almost all the chord length, while an overestimation of the phase angle is obtained on the upper surface near the trailing edge. At span 95% instead it is possible to see that a good agreement between numerical and experimental data is obtained basically on the entire x/c range. It is noted that the FRF of the indicator is performed after 3.0 seconds of physical time in order to discard the possible influence given by the initial transient. Furthermore, while the simulation is carried out with a physical time step of $\Delta t = 2.4 \cdot 10^{-4}\, s$, the solution is re-sampled such that the FFT is performed considering a time step of $\Delta t = 4.8 \cdot 10^{-3}\, s$ without any aliasing-like or information loss problems. Results provided by AeroX are in agreement also with

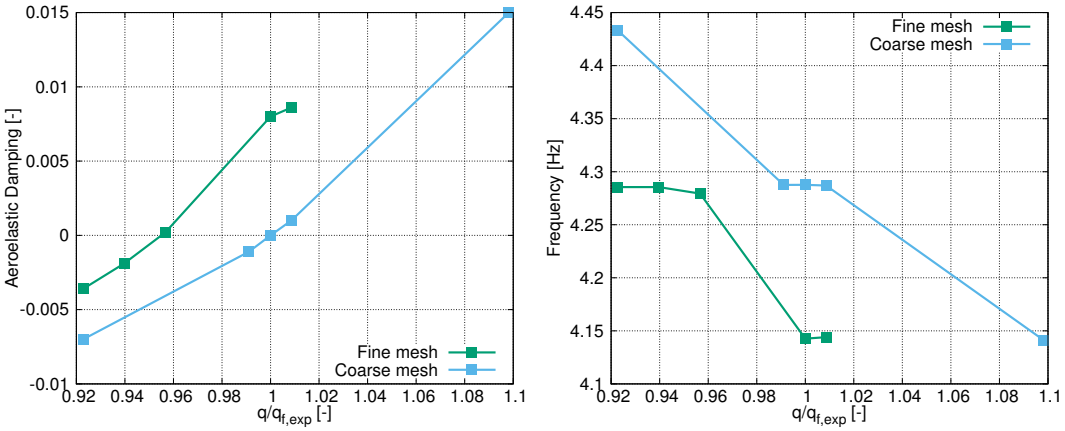


(a) Magnitude (b) Phase

Figure 8.24: *Magnitude and phase of the indicator,* 95% *span.*

preliminary numerical results provided by other research groups [82] participating to the AePW2. However at the time of writing there is still no official paper that collects all data but just preliminary plots. Thus the comparison with numerical results from other research groups is not presented here.

Figures 8.25 show the different dynamic pressure values investigated during the convergence procedure until the computational flutter dynamic pressure was found. In particular, figure 8.25(a) shows the damping values, while figure 8.25(b) shows the frequencies. It is possible to see that with the coarse mesh smaller aeroelastic damping but higher frequencies are found.



(a) Damping g (b) Frequency

Figure 8.25: *AePW2 BSCW wing flutter investigation with different dynamic pressure values.*

Finally a note concerning the computational effort of performing such kind of simulation. Using a physical time step of $\Delta t = 2.4 \cdot 10^{-4}\, s$ for a total physical simulation time of $9.0\, s$ requires the simulation of 37500 physical time steps. Using the AMD 380X GPU this is translated in a total required computational time of 133 hours for the

fine mesh and 27 hours with the coarse mesh.

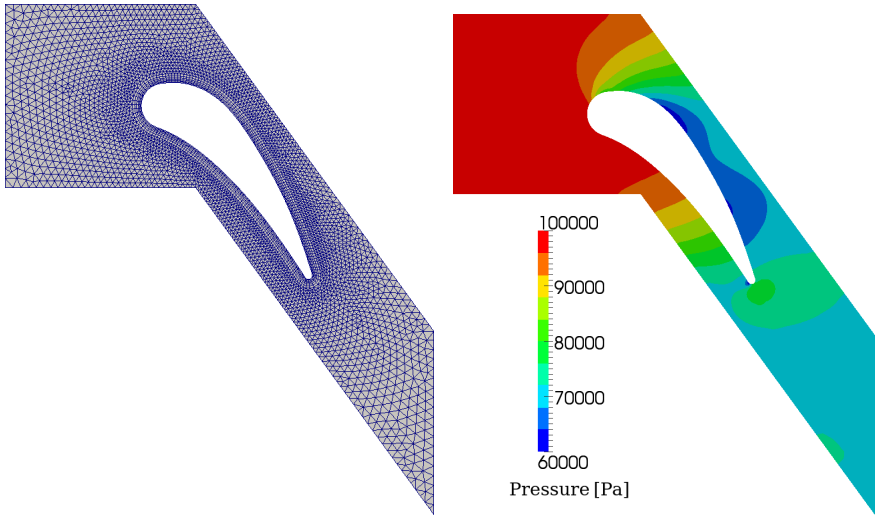
CHAPTER 9

Turbomachinery and open rotor blades aerodynamic applications

This chapter is aimed to validate the turbomachinery and open rotor extensions implemented in AeroX alongside the general purpose CFD schemes for classical aeronautical schemes. This step is obviously performed after the validation of the solver with classical aeronautical cases, in order to separately validate the general-purpose aerodynamic formulations and later the turbomachinery/open rotors extensions. Cases with different levels of complexity will be presented, from a simple 2D turbomachinery case to a 3D case with multiple MRF zones and blades communicating through mixing plane interfaces. In particular, for what concerns the 2D case, the Goldman test case will be presented and results will be compared with experimental data and other available numerical results. The well-known NASA's Rotor 67 axial compressor rotor blade is useful to demonstrate the ability of the solver to perform 3D compressible RANS simulations in a typical turbomachinery configuration. However, instead of presenting this case here, it will be discussed in the next chapter when performing turbomachinery aeroelastic simulations, in a comparison between steady rigid and steady aeroelastic results. The Aachen turbine test case is presented here to show the capability of AeroX to simulate multi-row cases. In this case two stator rows and one rotor row are modeled. Again, numerical results are compared with both experimental and numerical data available in literature. Finally the validation of the solver is also performed with a typical open rotor configuration. This latter case represents an hybrid between a classical aeronautical case (e.g. wing) and a turbomachinery case. Nonetheless this is presented now since it shares the same formulations required to simulate turbomachinery rotors. Open rotors and propfans represent a current trend, thus it is worth to asses the AeroX capability to handle such cases.

9.1 Goldman turbine blade

Here the focus is on the work of Goldman at al. [73, 129]. This is a two-dimensional turbulent simulation of a stator blade at mid-span section. Since this case involves a 2D stator domain, MRF formulation is not required. However, there are important differences between this case and a typical aeronautical airfoil case. Figure 9.1(a) shows the discretization of the computational domain. Since this is a 2D simulation, the front



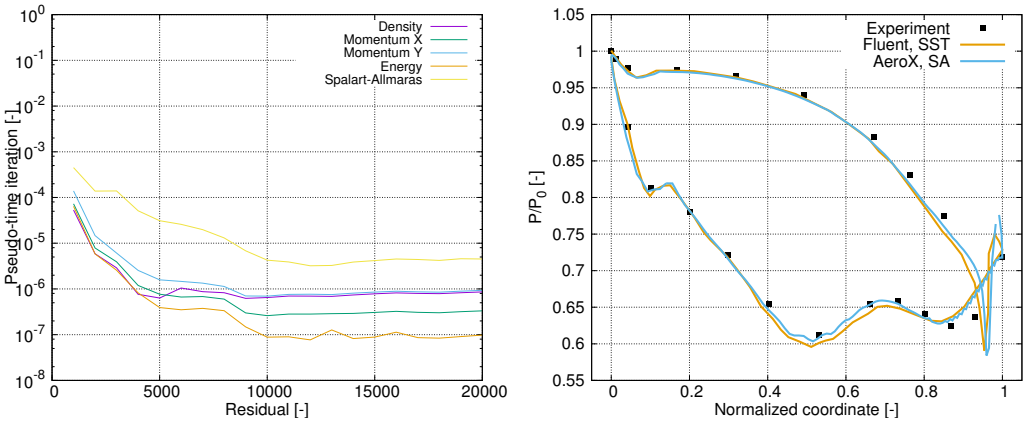
(a) Domain discretization for the Goldman 2D stator blade case

(b) Pressure field

Figure 9.1: *Mesh and pressure field results for the Goldman blade case.*

and back boundaries are represented by OpenFOAM empty patches, directly avoiding fluxes computation, saving computational time. On the top and bottom patches periodic boundary conditions are applied in order to reduce the computational domain to a single blade passage. As explained in 3.7, the use of periodic boundary conditions is subordinated to the hypothesis of a perfect symmetry of the flow over the blade row. This is the case and as will be presented with results, the use of this computational domain reduction technique allows to obtain accurate results while reducing the total computational effort. On the left, at the inlet boundary, total quantities boundary conditions are enforced, with user-defined values of the total temperature $T_0 = 287.91\,K$ and total pressure $P_0 = 101325\,Pa$. Thanks to this inlet boundary condition the flow is allowed to automatically adjust itself to the correct values of static temperature, static pressure and velocity vector. Finally, subsonic outlet boundary conditions are enforced on the outlet boundary. There is no need to use characteristics-based automatic boundary conditions since the flow is subsonic and no complex phenomena, like recirculations, occurs. Basically at the outflow boundary the ghost cell is constructed such as to obtain a zero-gradient boundary condition on velocity and temperature, while the pressure is computed interpolating the user-defined value of $71583\,Pa$ and the internal cells values. Finally, non-penetration non-slip boundary conditions are enforced over the blade through the the blended automatic wall treatment. The mesh adopted for the simulation is characterized by $7.2 \cdot 10^3$ cells, composed by 17% hexahedra for the near-wall dis-

cretization and by 83% prisms for the rest of the computational domain. This is thus an hybrid unstructured mesh. It must be noted that, as explained when performing computational benchmarks, with such a small number of domain cells, GPUs basically do not provide useful speed-ups with respect to CPUs. The case is nonetheless simulated in under $10\,s$ by all the GPUs presented in tables 6.1 and 6.2 considering that just $10 \cdot 10^3$ pseudo time iterations are enough for convergence as it is possible to see from figures 9.2(a), where the residuals history is showed. The case is here presented mainly to



(a) Residuals convergence rate with SA

(b) Mass flow convergence rate with SA

Figure 9.2: *Residual and mass flow convergence rate with SA for Goldman case.*

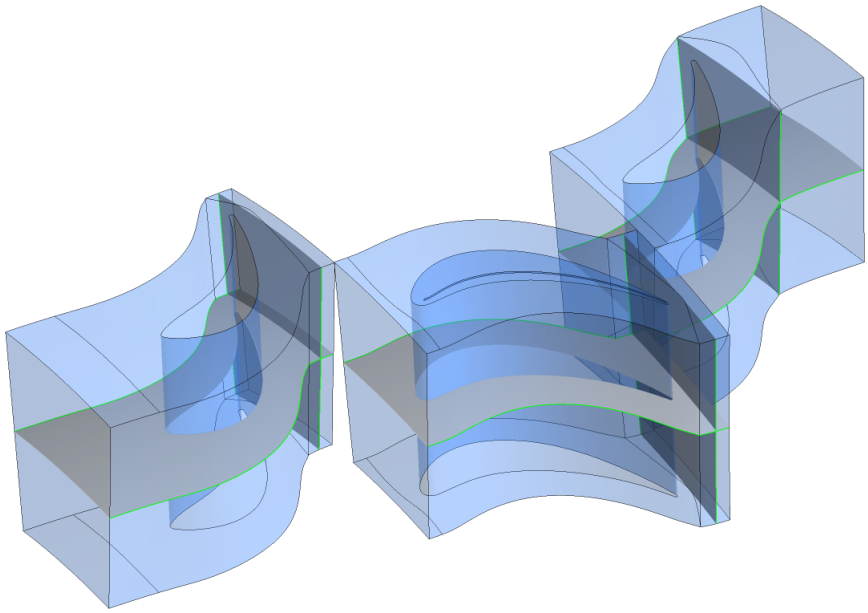
perform the validation of turbomachinery formulations like periodic and total boundary conditions. Based on the free-stream velocity and the chord length, the Reynolds number is $Re = 5 \cdot 10^5$ and the Mach number is $M_\infty = 0.2$. As it is possible to see both Reynolds and Mach number are relatively low with respect to a typical aeronautical case. In figure 9.1(a) it is possible to see the boundary layer discretization with hexaedra cells (it is reminded that OpenFOAM only handles 3D meshes, 2D cases are computed with the empty patches "trick"). With this kind of boundary layer discretization, y^+ values in the order of 200 are obtained, values that are correctly handled by the automatic wall treatment. The adopted turbulence model is SA that behaves well on this kind of simulations without complex viscous phenomena like separations and recirculations, as it is possible to see in figure 9.1(b) where the pressure field inside the domain is represented.

To complete the validation of the solver for this simple 2D stator case, figure 9.2(b) shows the comparison between AeroX, experimental data and numerical data provided by FLUENT with SST turbulence model. It is possible to see that the numerical results provided by AeroX are in good agreement with both experimental and reference numerical data.

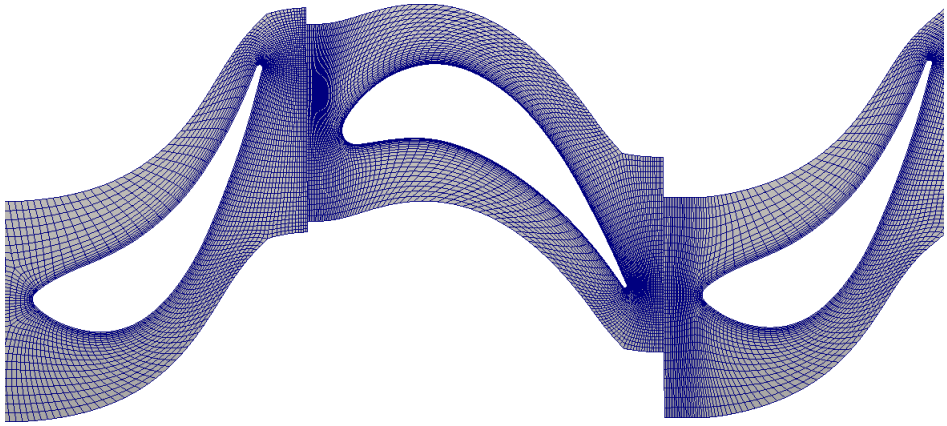
9.2 Aachen 1.5 stages axial turbine

This well-known test case is a 1.5 stages cold air axial turbine built at the Institute of Jet Propulsion and Turbomachinery at Aachen Technical University (IST RWTH Aachen, DE). This is an important test case that has been investigated by numerous authors.

Details regarding this test case can be found in [89, 119, 126, 129]. It is a multi-row 3D axial turbine case and it's important for the validation of AeroX turbomachinery features since it requires the mixing plane formulation in order to handle the interface between the 3 sub-domains, allowing at the same time the reduction of each blade row to a single blade sector, exploiting periodic boundary conditions. Figure 9.3(a) shows the 3 computational subdomains. The turbine is characterized by an hub radius



(a) Computational subdomains



(b) Hubs mesh discretization detail

Figure 9.3: *Overall and detail of the mesh discretization for the Aachen 1.5 stages turbine.*

of $145\,mm$ and a shroud radius of $300\,mm$. The blades of the first and second subdomains share the same geometry. Basically, the first and third subdomains represent single-blade sectors of stator rows composed by 36 blades, while the central subdomain represents a single-blade sector on a rotor row composed by 41 blades. Thus, MRF is applied only on the second subdomain. The angular speed for this row is $3500\,RPM$. Between the first and the second subdomains and between the second and the third

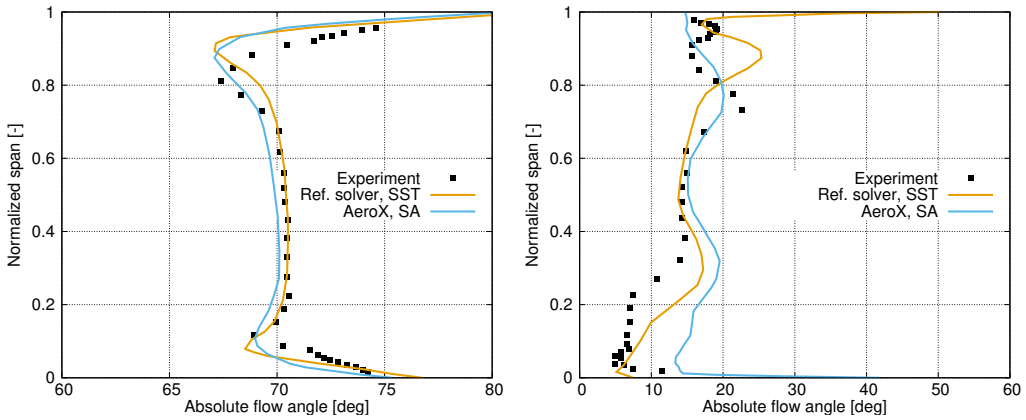
subdomains mixing plane interfaces are employed. While periodic boundary conditions allow the single-blade computational domain reduction when a single blade row is investigated, mixing planes allows the same kind of computational domain reduction when multiple communicating blade rows, with different periodicity values, have to be investigated. In fact, the single blade periodicity angle of the first and third subdomain is $360°/36 = 10°$ while for the second is $360°/41 = 8.78°$. The computational advantages provided by the mixing plane strategy come at the price of an accuracy reduction since the different periodicity is bypassed using a circumferential averaging of the solution. Nonetheless, as will be showed when comparing numerical results provided by AeroX and experimental data, the solver is capable to provide accurate results for this test case when employing the mixing plane formulation. The mesh adopted for this investigation is composed by $7.5 \cdot 10^5$ hexaedra cells (non-hybrid unstructured mesh). For this test case, modeling the entire $360°$ rows would require around $40\times$ the computational effort needed with the here employed domain reduction strategy. It must be noted, however, that the single-blade domain reduction is subordinate to the hypothesis of a perfect axial symmetry of the solution. Thus with this strategy it is not possible to directly study complex phenomena like rotating stall. Nonetheless this kind of effects are not expected in this test case for the investigated conditions. Thus, the use of mixing planes combined with periodic boundary conditions is fine. As it is possible to see in figure 9.3(b), a near-wall boundary layer discretization is employed, allowing y^+ values in the order of 50, which is good for the automatic wall treatment (log region). For this case, non-slip non-penetration boundary conditions are enforced all over the hub, shroud and blade walls. At the inlet of the first sub-domain total pressure ($P_0 = 152100\,Pa$) and total temperature ($T_0 = 305.65\,K$) are enforced while at the outflow of the third subdomain subsonic outlet boundary conditions are enforced, with a static pressure of $P = 109863\,Pa$. The investigation is performed using SA turbulence model.

Convergence is reached in about $15 \cdot 10^3$ iterations, corresponding to a total computational time of 13 minutes on the AMD 380X GPU that for this case reaches an iteration/time/cell value of $6.35 \cdot 10^{-8}\,s$ seconds. Differently from the Goldman case, the adopted GPU is here fully exploited with the given mesh.

For what concerns the validation, figure 9.3(a), already presented, shows the cutting planes adopted to extract the data used for the comparison with experimental and numerical reference results. The comparison is performed between AeroX results using SA turbulence model, experimental data and numerical data provided by a commercial code using SST turbulence model. More in particular, what is compared is the absolute flow angle at different normalized span locations. The absolute flow angle used for the comparison is defined as follows:

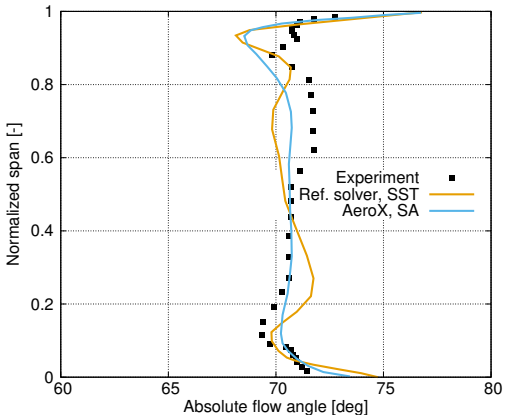
$$\gamma = \arctan(U_\theta / U_x) \tag{9.1}$$

where U_θ is the absolute circumferential flow speed and U_x is the absolute axial flow speed. The results are showed in figures 9.4. Figure 9.4(a) shows the absolute flow angle $8.8\,mm$ behind the trailing edge of the first vane (first subdomain). Figure 9.4(b) shows the absolute flow angle $8.8\,mm$ behind the trailing edge of the rotor (second subdomain). Finally figure 9.4(c) shows the absolute flow angle $8.8\,mm$ behind the trailing edge of the second vane (third subdomain). It is possible to see that a general good agreement between AeroX results and both numerical reference results and



(a) Absolute flow angle, $8.8\,mm$ behind the trailing edge of the first blade (stator)

(b) Absolute flow angle, $8.8\,mm$ behind the trailing edge of the second blade (rotor)



(c) Absolute flow angle, $8.8\,mm$ behind the trailing edge of the third blade (stator)

Figure 9.4: *Absolute flow angle at the three considered locations.*

experimental data is obtained. On both the stator sections AeroX tends to slightly underestimate the absolute flow angle with relative errors of approximately $2-3\%$, while on the rotor section AeroX tends to slightly overestimate the absolute flow angle with significant relative errors, especially in proximity of the hub. This is probably due to a spurious interaction of the mixing plane with the turbulent quantities. In fact, this effect is not significant in $360°$ unsteady simulations, as shown in [155].

Figure 9.5 shows the Mach number field inside the domain at the mid-span section of the Aachen turbine.

Finally, AeroX predicts a mass flow of $6.984\,kg/s$. This value is in agreement with what can be found in literature both for what concerns experimental data and numerical results. In particular, in [155] a value of $7\,kg/s$ is considered which is in agreement with what obtained with AeroX.

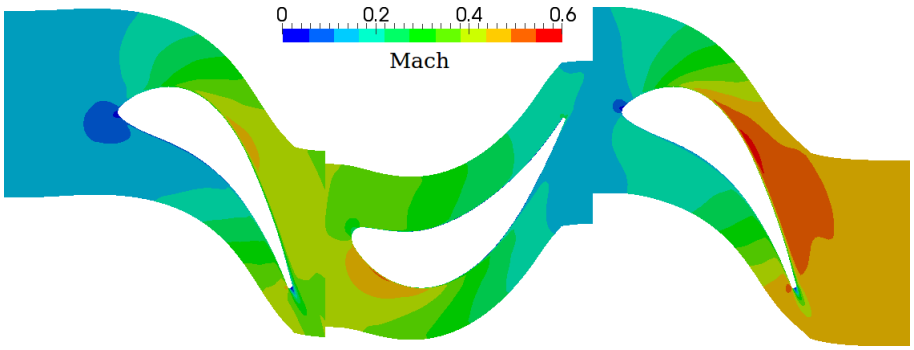


Figure 9.5: *Aachen turbine Mach number field.*

9.3 Open Rotor

The last case presented in this chapter is represented by a typical open rotor configuration. This is described in details in [144] and [122] which are also the sources of data here used for comparisons. In particular, in [122] the solution is provided by an implicit non-lienar full-potential aeroelastic solver, S^T [115]. The DLR TAU code [72] solver, using Chimera formulation [99], is instead adopted to provide RANS results [144] with Spalart–Allmaras turbulence model. Here, AeroX solutions are provided with both an inviscid analysis and a RANS analysis with SA turbulence model in order to provide a comparison with both full potential and RANS reference results. The RANS analysis is also provided as a complementary solution in order to check for possible results accuracy improvements due to considering viscous effects with respect to the plain Euler solution. Different pitch angles are investigated for a single flight condition. Without going into details for what concerns the blade geometry, figure 9.6 shows the near-blade detail of the mesh here adopted for the steady-state inviscid simulation for the particular pitch angle of $56°$. The pitch angle here adopted, $\beta_{75\%}$, is referred to the plane of rotation of the rotor disk and the 75% blade span section. This is an 8-blades open rotor. In order to reduce the computational effort a single blade passage in computed and periodic boundary conditions are employed. Furthermore MRF formulation is active in order to allow the simulation of the rotating geometry without actually employing a true unsteady simulation with mesh rotation. From figure 9.6 it is possible to see: the blade (green), the hub (purple), and the two periodic boundaries (red and light grey). The blade is attached to an hub that is rotating at $895\,RPM$. Within a single flight condition different pitch angles are taken into account for the analysis: $56°$, $57°$, $60°$ and $62°$. The meshes here adopted are composed by around $975k$ tetrahedra and are generated with GAMBIT. These are thus unstructured non-hybrid meshes for which no branch divergence is expected. The rotor diameter is $D = 4.2672\ m$ with a hub-to-tip ratio of $d/D = 0.355$, while the blade features a transition from NACA 65-series to NACA 16-series airfoil from the root to the tip. The investigation is performed with a Mach number $M_\infty = 0.75$, an altitude of $h = 10668\,m$ which correspond to a static pressure of $P_\infty = 26500\,Pa$ and a density of $\rho_\infty = 0.4135, Kg/m^3$. This, alongside the blade rotational speed is translated into a blade tip Mach number of about 1, leading to weak shocks over the blade surface.

Figures 9.7 show the obtained results for what concerns the thrust coefficient C_T,

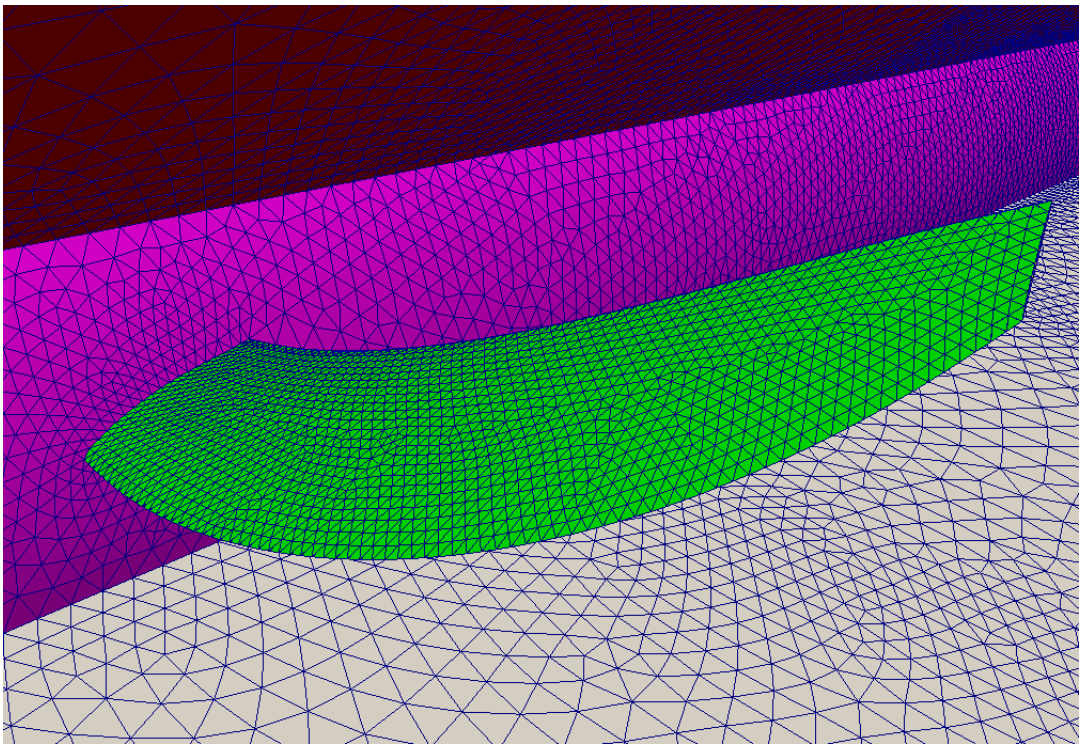
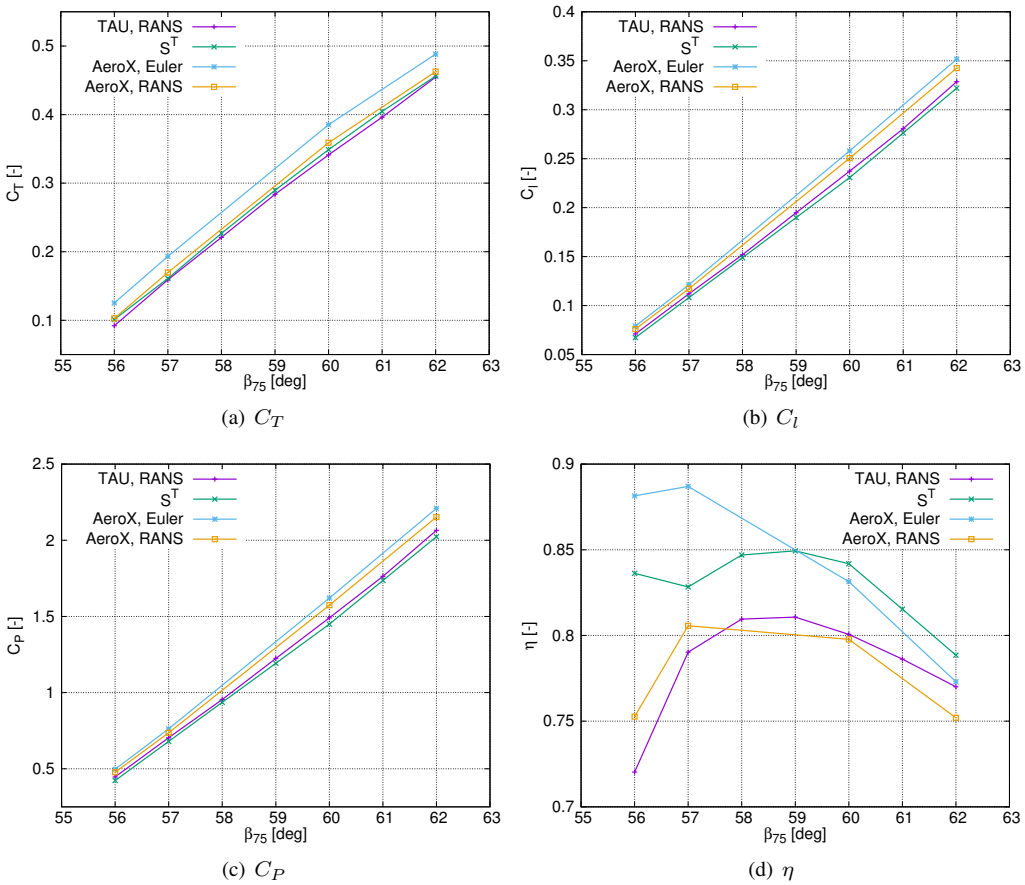


Figure 9.6: *Open rotor mesh for pitch angle of* $56°$*, blade and hub.*

the torque coefficient C_l, the power coefficient C_P (not to be confused with the pressure coefficient) and the propeller efficiency η here defined:

$$\begin{aligned} C_T &= \frac{T}{\rho n^2 D^4} \\ C_l &= \frac{C}{\rho n^2 D^5} \\ C_P &= 2\pi C_l \\ \eta &= \frac{C_T}{C_l}\frac{J}{2\pi} \end{aligned} \tag{9.2}$$

where $n = \Omega/2\pi$ is the propeller rotational speed, $J = V_\infty/(nD)$ is the advance ratio, V_∞ is the asymptotic speed, $C = \rho n^2 D^5 C_l$ is the torque. From the results it is possible to see that AeroX with Euler formulation seems to slightly overestimate the values of C_T, C_l, C_P and η, the latter especially for what concerns low pitch angles with respect to the two sets of numerical reference data. However, AeroX using RANS provides better results, in good agreement with literature. It must be noted that, as explained in [122], the fact that the full potential solution seems to perform better than AeroX with inviscid formulation is probably due to numerical dissipation related to the methods implemented in S^T. Finally figures 9.8 and 9.9 show the comparison between S^T and AeroX Euler results for what concerns the blade Mach and C_P distributions for the aforementioned flight conditions and the pitch angle of 60%. It is possible to see that despite small local differences, the overall solution is basically in agreements between both solvers.

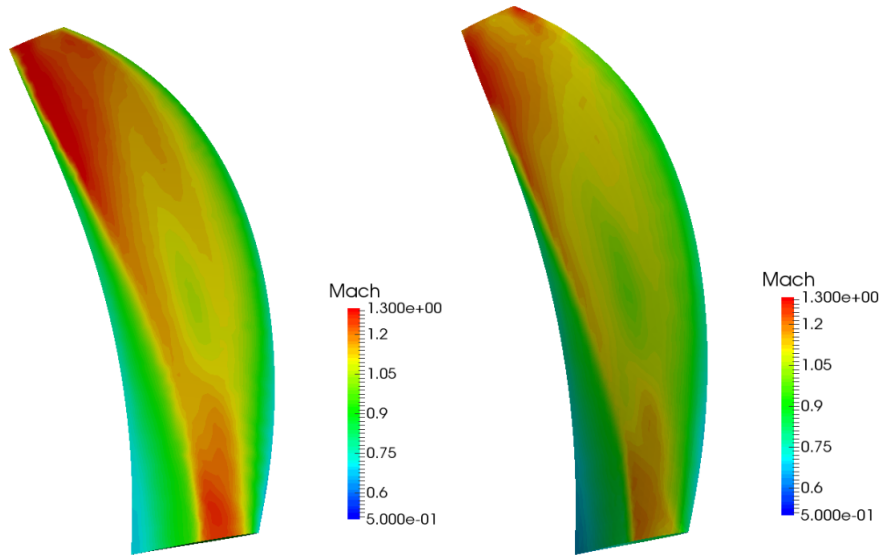


(a) C_T (b) C_l

(c) C_P (d) η

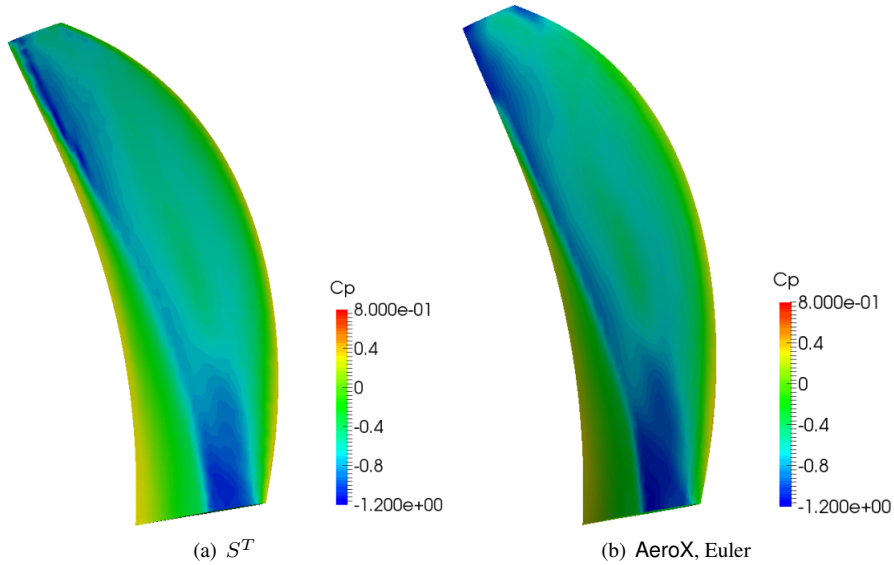
Figure 9.7: *Thrust, torque, power coefficient and efficiency of the open rotor at different pitch angles.*

For what concerns computational aspects, using the AMD 380X GPU, for each computed angle about $500\,s$ are required to obtain the RANS solution with a time/iteration/cell of $4.51 \cdot 10^{-8}\,s$, while for the inviscid solution it boils down to about $300\,s$ and a time/iteration/cell of $2.67 \cdot 10^{-8}\,s$.



(a) S^T (b) AeroX, Euler

Figure 9.8: *Mach distribution comparison, pitch angle of* $60°$.



(a) S^T (b) AeroX, Euler

Figure 9.9: *Pressure coefficient distribution comparison, pitch angle of* $60°$.

CHAPTER 10

Turbomachinery and open rotor blades aeroelastic applications

The next step after the validation of the aeroelastic framework with classic aeronautical cases is the validation with typical turbomachinery and open rotors test cases. In this chapter static and dynamic aeroelasticity analyses will be performed on three of the most important benchmark cases available in literature. In particular, the KTH's Standard Configuration 10 (SC10) and the NASA's Rotor 67 (R67) blades are chosen for the validation of the turbomachinery formulations. 2-dimensional and a 3-dimensional meshes are available for the SC10 case, allowing to asses the effects on the solution given by the 2D assumption. The SC10 represents a turbine blade, thus no MRF is required for the analysis. For the SC10 cases the aerodynamic damping analysis for a wide range of IBPAs is performed and results are compared together (2D and 3D versions) and with other numerical results available in literature. The R67 test case is instead adopted to asses the effects of considering the blade deformation due to aerodynamic loads, i.e. static aeroelasticity. As it will possible to see from the results, despite other typical aeronautical cases (e.g. the HiReNASD wing trim previously presented), due to the high blade stiffness, aerodynamic and steady aeroelastic analyses provide nearly the same results. R67 results are compared with experimental data available in literature. Finally, the flutter of a typical propfan configuration, the SR-5 blade, is analyzed in order to asses the capability of AeroX to simulate such kind of configurations that currently represents an interesting research field.

10.1 SC10 2D aerodynamic damping

The KTH's Standard Configuration 10 (SC10) is the first case analyzed here and is extracted among different well-known benchmark cases [69]. This is a single-blade

computational domain extracted from a two-dimensional compressor cascade. The airfoils are basically represented by modified NACA 0006 profiles and they are operating at subsonic inlet and outlet conditions. The investigation here is focused on the aerodynamic damping prediction. This is a simple 2D inviscid test case which is here adopted to validate the time-delayed boundary conditions implemented in the solver. As explained in 3.8 this particular boundary condition represents a smart modification of the usual periodic boundary conditions in order to allow to perform simulations with non-null IBPA values, even if a single blade domain reduction is employed. Thus, the key concept here is the IBPA. Mesh deformation alongside DTS formulation are here simultaneously employed to perform multiple time-accurate simulations, one for each investigated IBPA value. The idea is to combine results from all simulations in order to build the IBPA vs. aerodynamic damping diagram. The aim is to check if, for the given flow conditions, it is possible for this configuration to experience instability. As explained in 3.4 this can be viewed from an energetic point of view. If the aerodynamic damping is positive the flow is pumping energy into the system. It must be noted however, that differently from the analysis performed for the AGARD 445.6 wing flutter and the 2nd AePW BSCW wing flutter benchmarks, the investigation performed here is not a fully aeroelastic stability (flutter) investigation. In fact, here we are not searching for the true aeroelastic modes with certain frequencies and shapes and trying to find the conditions that lead to the aeroelastic instability. Here the investigation is limited to figure out if for the given geometry and flow conditions the aerodynamic loads are supporting or not the enforced movements.

Figure 10.1, taken from [69] describes the SC10 configuration. Without going into details, the adopted airfoil is built by superimposing the thickness distribution of a modified NACA 0006 airfoil on a circular-arc camber line. The stagger angle (angle between the chord and the axial direction) is $\gamma = 45^{\circ}$ while the chord/gap ratio is $\tau = 1.0$. From this data it is possible to build the computational domain, that for this investigation is depicted in figure 10.2. It is possible to see that a single blade is discretized. Since this is an inviscid investigation there is no need to employ a boundary layer discretization. As mentioned, periodic/time-delayed boundary conditions are adopted to reduce the total computational domain to a single blade, while retaining the effects of the presence of other blades. In figure 10.2 it is possible to see that the solver has to deal with 5 boundaries. On the top and the bottom of the domain periodic boundary conditions or time-delayed boundary conditions are employed for steady and unsteady aerodynamic damping simulations with non-null IBPA values respectively. It is noted that an unsteady simulation with a null value of IBPA can be performed directly using simple periodic boundary conditions rather than more expensive time-delayed BCs. On the left of the computational domain it is possible to see the inlet boundary while on the right the outlet boundary. As mentioned this is a subsonic case, thus it is possible to use subsonic outlet boundary conditions on the right boundary. At the inlet total pressure and total temperature are enforced. Different numerical analyses have been performed on this test case. Reference data from RPMTURBO solver and results from Verdon and from Hall can be found in literature. Sources of these data are represented by [13, 14]. These data are here used to perform the comparison both for what concerns steady and unsteady results. This is an insviscid test case, thus it is analyzed with AeroX using Euler formulation. This is useful because it is possible to

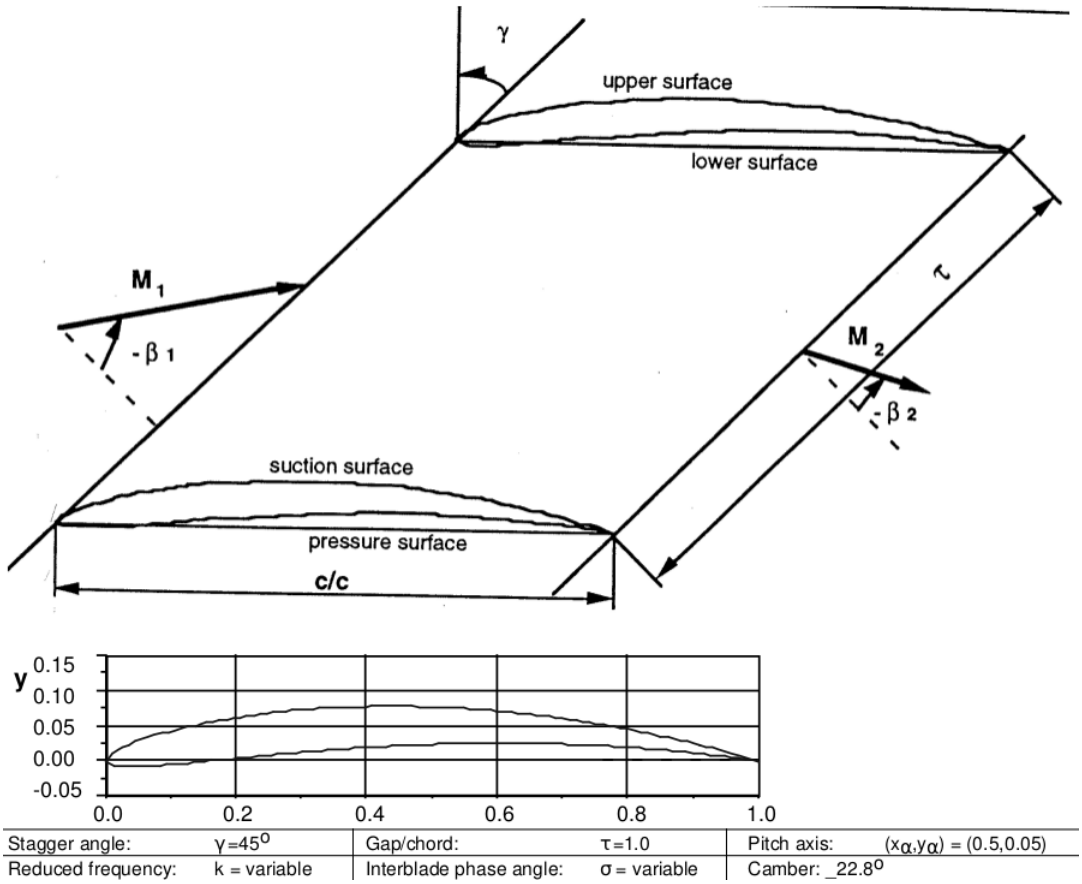


Figure 10.1: *Standard Configuration 10 cascade composed by modified NACA 0006 airfoils.*

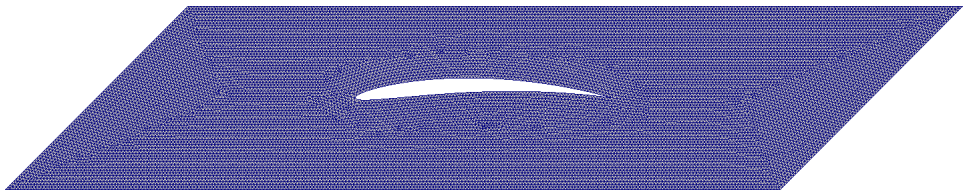


Figure 10.2: *Standard Configuration 10 cascade mesh.*

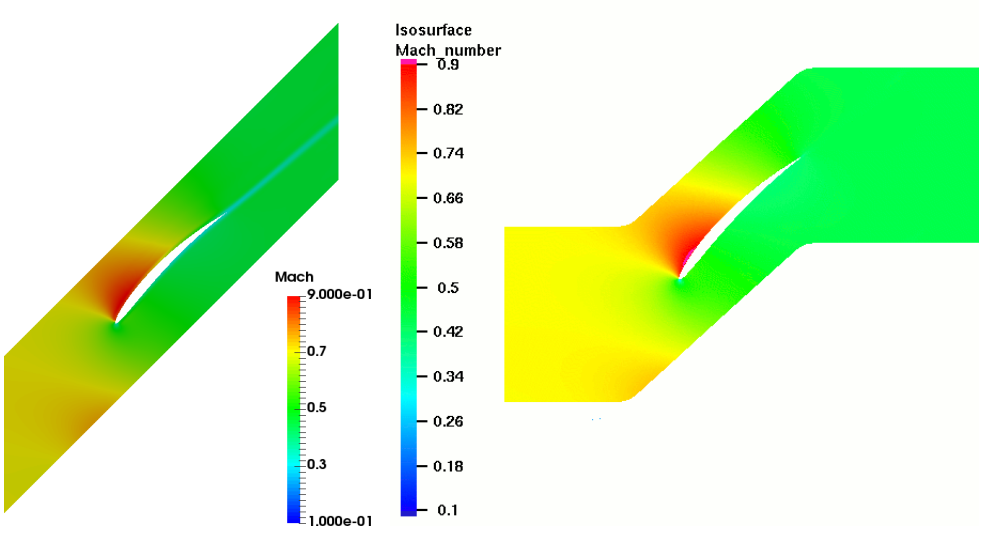
compare unsteady results provided by mesh deformation and by transpiration boundary conditions. The solver implements an optimized mesh deformation strategy. Due to the small differences in computational effort between transpiration boundary conditions and true mesh deformation, it is usually better to directly perform unsteady investigations with the latter strategy. However, this test case is here exploited to demonstrate the possibility of using transpiration boundary conditions also for turbomachinery test cases. Since inviscid simulations are performed, slip boundary conditions are enforced on the airfoil. The adopted mesh is composed by $31 \cdot 10^3$ cells, not enough to fully exploit the AMD 380X GPU but anyway enough to obtain accurate results with the Euler formulation.

For this case comparison data is available for two different sets of flow conditions.

Here the subsonic set is investigated. This is characterized by the following inlet conditions: $M_1 = 0.7$ and $\beta_1 = -55°$. At these conditions there is no shock over the airfoil. Observing figure 10.1 it is possible to see that due to the stagger angle value, the airfoil is subjected to an angle of attack of $10°$. For this test case just the average inlet Mach number and angle are specified. Thus, inlet and outlet boundary conditions are opportunely tuned in order to reach the specified inlet conditions. In this investigation this can be accomplished using inlet boundary conditions of $T_0 = 300\,K$ and $P_0 = 101300\,Pa$ (and the flow velocity direction) and outlet static pressure of $P = 88000\,Pa$.

10.1.1 Steady results

The steady-state investigation is required in order to find the correct initial conditions that will be later used to re-start the solver for time-accurate simulations. As said, periodic boundary conditions are employed. With the aforementioned inlet and outlet conditions AeroX provides the Mach distribution showed in figure 10.3 in a comparison with results available from [13]. It is possible to see that for these conditions, as



(a) AeroX (b) Reference numerical results

Figure 10.3: *Mach number comparison.*

expected, there are no complex compressible phenomena like shocks. The computational domain is opportunely rotated in order to ease the comparison with reference data. The results provided by AeroX are in good agreement with reference data. It is possible to see that the computational domain of the reference data is slightly different from what used with AeroX. This is not a problem since anyway periodic boundary conditions are adopted on the top and bottom boundaries. The steady-state solution is computed in 30 seconds by the AMD 380X GPU.

10.1.2 Aerodynamic damping results

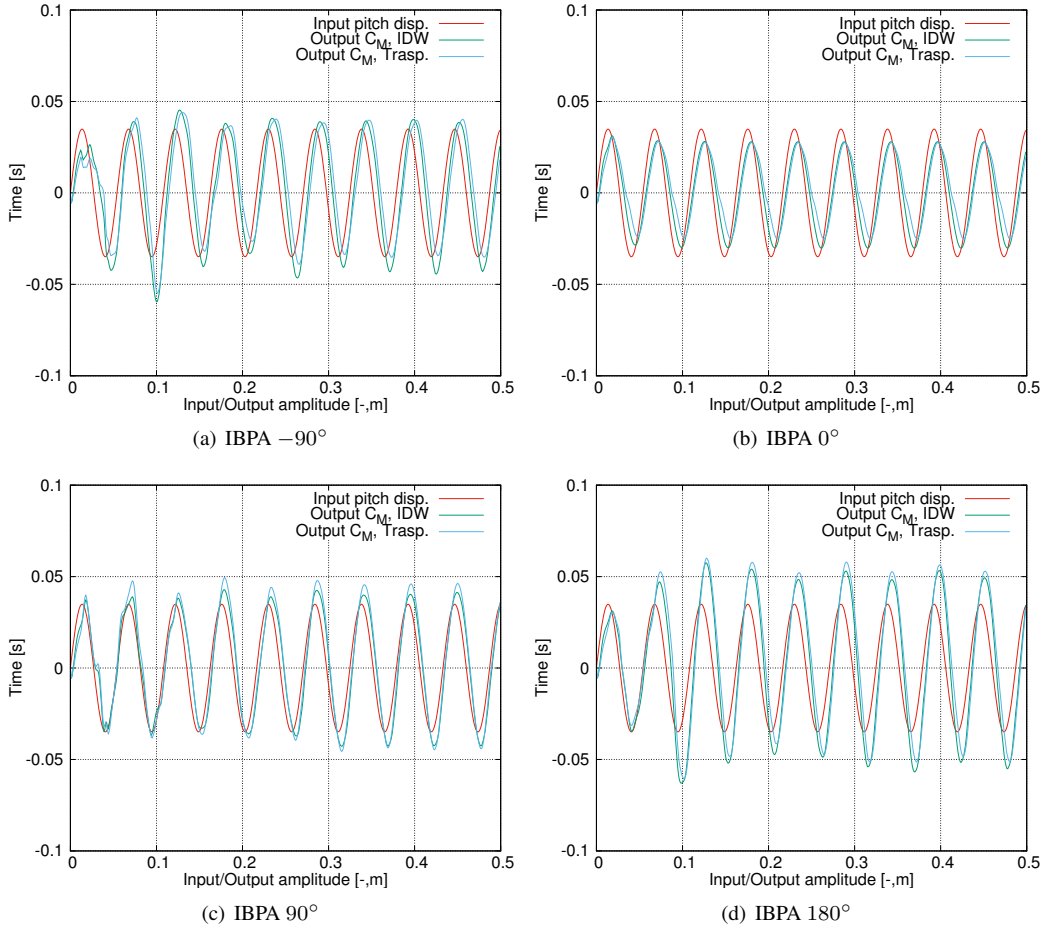
With this two-dimensional configuration the two investigated degrees of freedom are represented by the airfoil plunge and pitch around the point located at $(0.5, 0.05)$ which

can be easily located thanks to figure 10.1. Furthermore, for the aforementioned subsonic boundary conditions, different unsteady investigations with different reduced frequency k values can be performed. Here the focus is on both pitch and plunge oscillations considering $k = 0.5$, based on the blade chord. Different numerical results are available in literature from RPMTURBO, Verdon and Hall [13]. These data will be used for the comparison. As for the flutter analysis of wings, given the modal shape (in this case rigid pitch and plunge displacements) and the frequencies to be excited, another important parameter is represented by the amplitude of the displacements. As previously said this must be small enough to not jeopardize the small perturbations hypothesis but at the same time must be large enough to be clearly distinguishable from numerical errors. Following the guidelines presented in [128], a pitch angle of $\Delta\alpha = 2^\circ$ is directly employed for the pitch d.o.f.. For what concerns the plunge movement an opportune amplitude is chosen considering both the flow speed and the blade maximum oscillation speed such as to obtain about the same value of $\Delta\alpha = 2^\circ$. The adopted physical time step is $\Delta t = 1.0 \cdot 10^{-3}\, s$ that is small enough to correctly discretize the frequencies of interest and large enough to save computational time. The simulations are carried out using DTS for $0.5\, s$ of physical time. The simulations are repeated for the different IBPA values with steps of 10°.

Let's first analyze the pitch oscillations case. Figures 10.4 show the blade response in term of C_Y alongside the time history of the enforced pitch oscillations for four different IBPA angles. It is clearly possible to see that the delay between the displacement input and the aerodynamic load output is different for the different angles, leading to different values of the aerodynamic damping. In particular, the delay obtained with transpiration BCs is slightly higher, leading to slightly higher values of aerodynamic damping ξ. The unsteady simulations with all the IBPA values are re-started from the same null IBPA steady-state solution. The unsteady solutions experience an initial transient before reaching a periodic behavior. Therefore, the data used for the post-processing and computation of the aerodynamic damping is taken after $0.15\, s$. Figure 10.6(a) finally shows the IBPA vs. aerodynamic damping curves. It is possible to see that results provided by AeroX are in agreement with reference numerical solutions.

Now the same investigation is performed for what concerns the plunge d.o.f.. In this case the aerodynamic damping is computed considering the rotation of the blade around the previously defined point and the aerodynamic loads given by the moment coefficient. Figures 10.5 show the response in term of moment coefficient in a comparison between the aerodynamic output and the displacement input. Again, it is possible to see that with different IBPA values different delays between the input and the output are obtained, leading to different aerodynamic damping values. The response provided with transpiration BCs is characterized by slightly higher delays, leading to slightly higher aerodynamic damping values ξ. As for the pitch investigation, the post-processing is performed after $0.15\, s$ in order to cancel out the effects of the initial transient from steady-state conditions. Figure 10.6(b) shows the IBPA vs. aerodynamic damping curves. It is possible to see that also in this case AeroX provides results that are in agreement with other numerical reference data.

It is important to notice that reference data is obtained by means of linearized inviscid solvers while AeroX provides fully non-linear time-accurate solutions. In general with both plunge and pitch degrees of freedom the aerodynamic damping values pro-

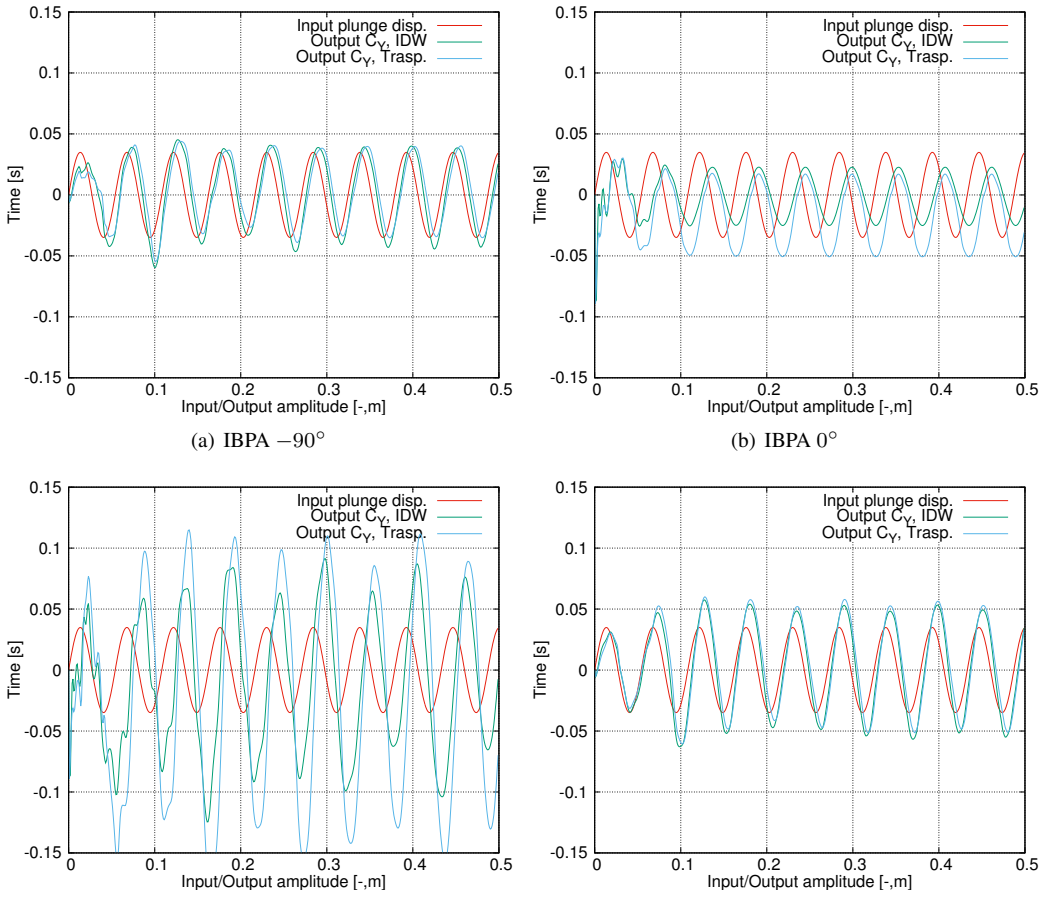


(a) IBPA $-90°$ (b) IBPA $0°$

(c) IBPA $90°$ (d) IBPA $180°$

Figure 10.4: *Comparison at different IBPAs between input (pitch rotation) and output (load coefficient C_M) for the SC10 2D case.*

vided by non-linear solutions obtained with AeroX appear to be higher than those obtained with reference linearized solvers. As in [128] results obtained in this work agree more with reference results obtained with linearized solvers with high IBPA values where the flow-field is dominated by the time-delayed/phase-lagged mutual interaction between neighbouring blades. This is because in these conditions non-linearities associated to average flow are not fully triggered yet. The general overestimation of ξ relative to the pitch case at null IBPA with respect to linearized solutions is observed also in [112] where again a non-linear approach is employed. Finally for what concerns the results obtained with transpiration boundary conditions it can be seen that they are in agreement with those obtained with a true mesh deformation, though a general overestimation of the aerodynamic damping seems to be obtained. As said, with AeroX it is better to directly use the mesh deformation algorithm since the computational cost difference between the two formulations is very small while the better results accuracy provided by true mesh deformation can be easily appreciated. In any case all the presented results at all IBPA angles are characterized by $\xi > 0$ meaning that for this geometry and flow conditions no instability is found.



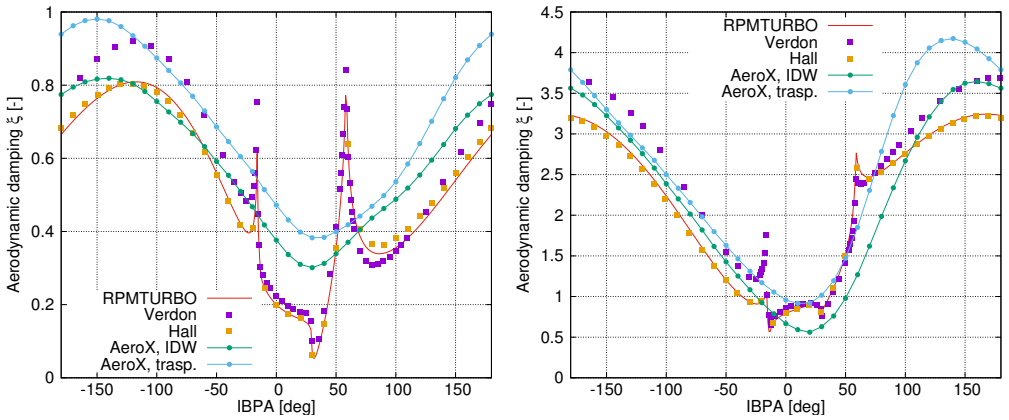
(a) IBPA $-90°$ (b) IBPA $0°$

(c) IBPA $90°$ (d) IBPA $180°$

Figure 10.5: *Comparison at different IBPAs between input (plunge displacement) and output (load coefficient C_Y) for the SC10 2D case.*

10.2 SC10 3D aerodynamic damping

Here the same investigation performed for the Standard Configuration 10 two-dimensional test case is repeated using a three-dimensional domain. This test case was proposed by Matthew Montgomery and Joe Verdon in [110]. Data can be also found in [15]. The hub radius is 3.395 chord lengths while the shroud is located at 4.244 chord lengths. The blade is composed by the same modified NACA 0006 airfoil adopted for the 2D investigation. There is no tip clearance between the shroud and the blade tip. As for the 2D configuration, different numerical reference results are available in literature. Here the comparison is performed between results computed by AeroX and numerical results available in [15, 118] for what concerns the RPMTURBO solver. Again, the investigation is carried out with Euler unsteady simulations through a re-start from steady-state null-IBPA initial conditions. The strategy for the computation of ξ is the same adopted for the 2D configuration (10.1): the aerodynamic damping is computed as a post-processing operation carried out over different unsteady simulations, one for each IBPA value. Figures 10.7 show the computational domain adopted for the simu-



(a) Pitch d.o.f.

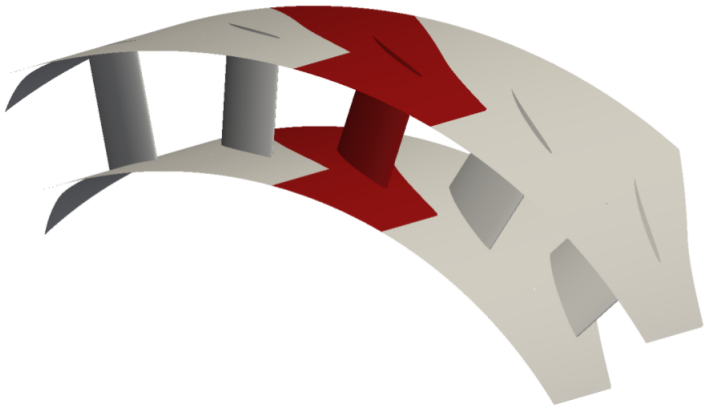
(b) Plunge d.o.f.

Figure 10.6: *Aerodynamic damping ξ for different IBPAs for the SC10 2D configuration, plunge and pitch d.o.f..*

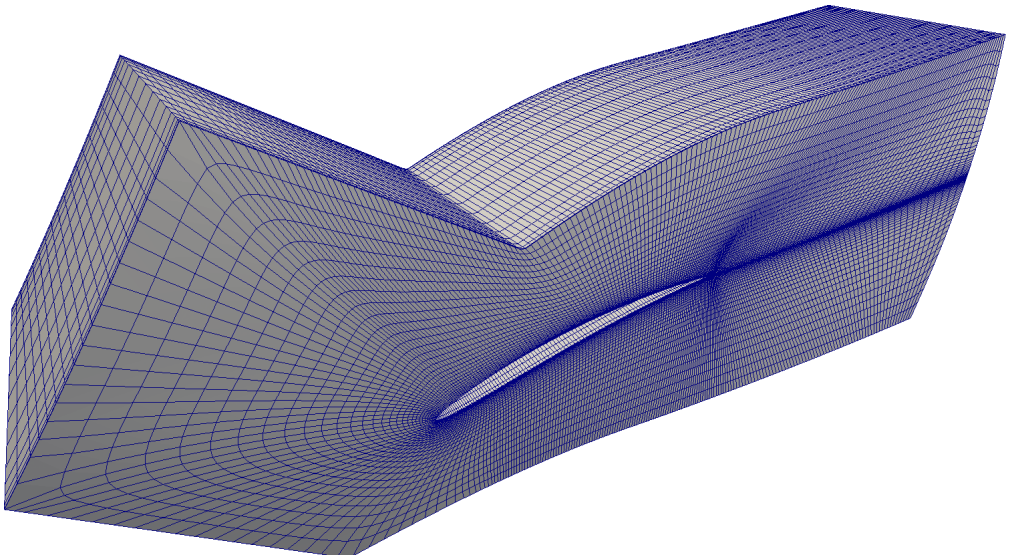
lations alongside the detail of the mesh discretization. This is basically a single blade sector extracted from a 24-blades stator row. With respect to the 2D case where just few cells were employed, here the mesh is composed by $190 \cdot 10^3$ hexaedra. With this kind of discretization the AMD 380X GPU adopted for the simulation is almost fully loaded. At the same time the adopted discretization level is enough for an inviscid simulation. Again, the steady-state solution is obtained at null IBPA with simple periodic boundary conditions on the top and the bottom of the domain while the same patches are treated with time-delayed boundary conditions for unsteady computations with non-null IBPA values. Slip boundary conditions are enforced on the hub and the shroud of the domain. Total temperature, total pressure and flow direction are enforced on the inlet boundary. In particular, for this analysis the inlet conditions are $M_1 = 0.7$ and $\beta_1 = -55°$, i.e. the same inflow conditions adopted for the 2D analysis. These conditions are obtained by enforcing on the inlet $T_0 = 300\,K$ and $P_0 = 101300\,Pa$, as suggested in [15]. On the outlet boundary, subsonic outlet boundary conditions are employed, enforcing a static pressure of $P = 87600\,Pa$, a value which is very similar to the one adopted for the 2D investigation.

10.2.1 Steady results

The steady-state solution with the given flow conditions is obtained in $148\,s$ with a time/iteration/cell of $3.150 \cdot 10^{-8}$ seconds and $20 \cdot 10^3$ pseudo time iterations using the AMD 380X GPU. Figures 10.8 show the comparison between AeroX results and numerical results from [128] for what concerns the pressure field. It is possible to see that the numerical results obtained in this investigation are in agreement with reference numerical results. As for the 2D configuration, with these conditions there are no strong compressible phenomena like shocks. The validation is completed comparing at



(a) Single blade sector computational domain (red)



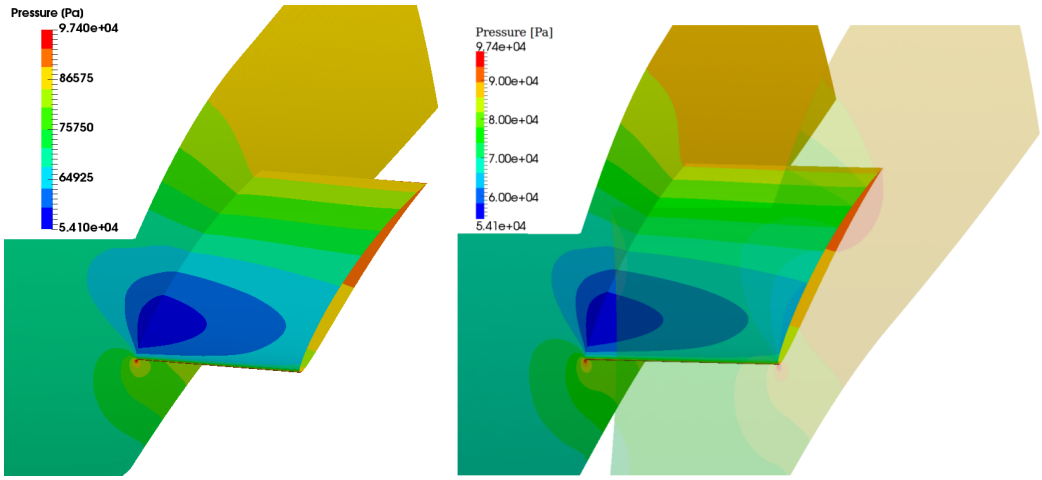
(b) Mesh discretization detail

Figure 10.7: *3D computational domain for the aerodynamic damping analysis of SC10.*

different span sections the isentropic Mach distribution defined as:

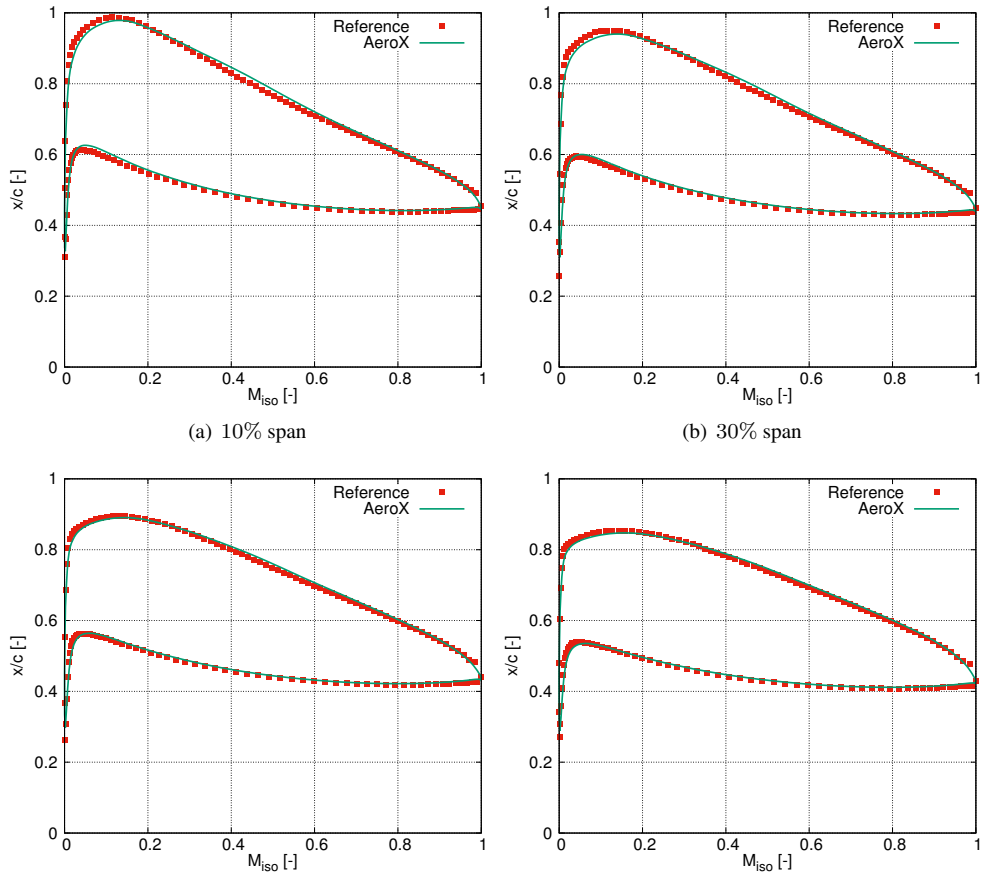
$$M_{iso} = \sqrt{\left(\left(\frac{P_0}{P}\right)^{\frac{\gamma-1}{\gamma}} - 1\right)\frac{2}{\gamma-1}} \tag{10.1}$$

The comparison is performed with numerical results available in [15] for the 3D inviscid set. Figures 10.9 show the comparison for four different span sections. It is possible to see that results provided by AeroX are in good agreement with reference numerical results.



(a) AeroX solution (b) Reference numerical solution

Figure 10.8: *Pressure field comparison for steady-state solution of SC10 3D configuration.*



(a) 10% span (b) 30% span

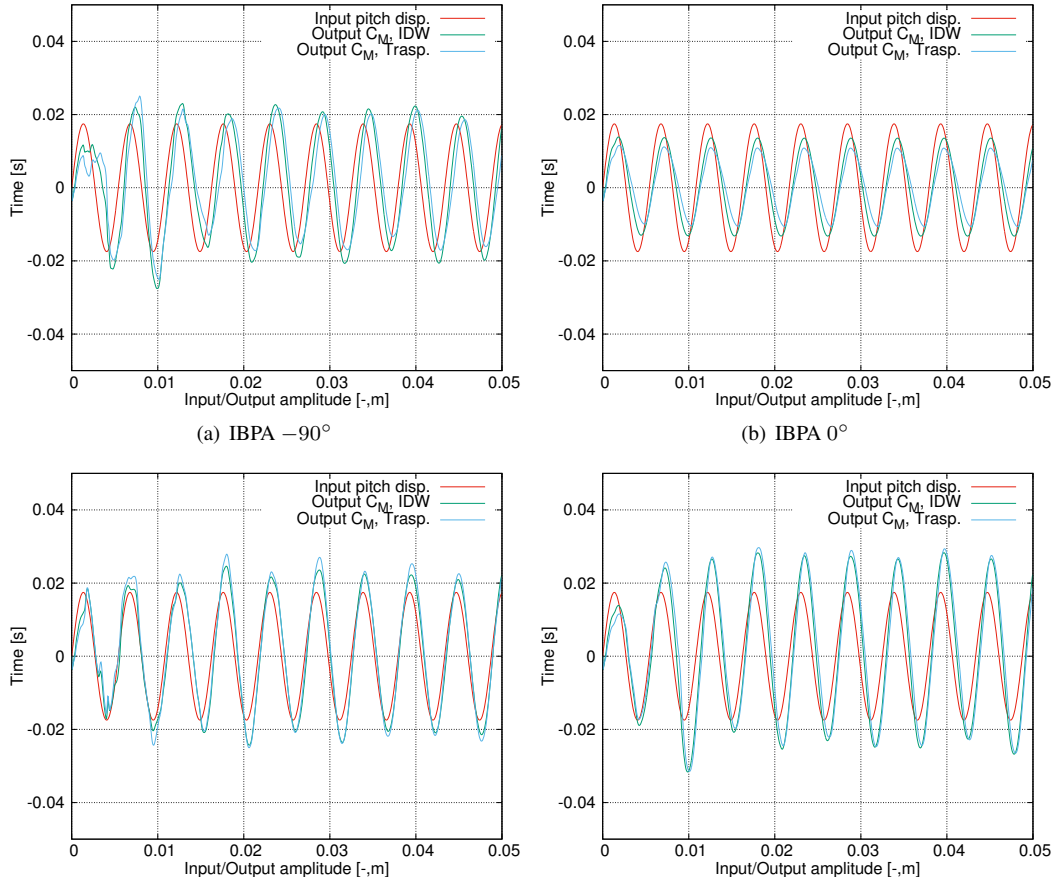
(c) 60% span (d) 90% span

Figure 10.9: *Isentropic Mach distribution at different span sections for the SC10 3D case blade.*

10.2.2 Aerodynamic damping results

From the steady-state solution the solver is re-started multiple times with different IBPA values. As for the 2D case, steps of 10° are employed to obtain a smooth aerodynamic damping curve. Each unsteady simulation is performed for 0.05 seconds of physical time. A physical time step of $\Delta t = 5.0 \cdot 10^{-5}\, s$ is used in a trade-off between computational effort and the accurate reconstruction of the interesting frequencies. As for the 2D case the reduced frequency is $k = 0.5$ but here just the pitching movement d.o.f. is considered. The adopted pitching oscillation amplitude is $\Delta\alpha = 1^\circ$ and as for the 2D case oscillations are performed around the point at $(0.5, 0.05)$ with respect to the blade airfoil coordinates.

Figures 10.10 show the blade response for four different IBPA values, comparing the input signal, i.e. pitching rotation, and the output signal, i.e. moment coefficient C_M. As for the 2D cases it is possible to see that the delay between input and output signals are different and those differences will be also found in the final aerodynamic damping curves. The simulations are performed both with transpiration boundary conditions and true mesh deformation. As for the 2D case transpiration provides higher delays and thus higher aerodynamic damping values.



(a) IBPA -90°

(b) IBPA 0°

(c) IBPA 90°

(d) IBPA 180°

Figure 10.10: *Comparison at different IBPAs between input (pitch rotation) and output (load coefficient C_M) for the SC10 3D case.*

Figure 10.11 finally shows the IBPA vs. aerodynamic damping curve. It is possible to see that results provided by the non-linear time accurate solver AeroX are in agreement with numerical reference data, both for what concerns true mesh deformation and transpiration. As expected, transpiration BCs provide slightly higher values of aerodynamic damping with respect to mesh deformation. The same considerations presented for the 2D case are valid also here for what concerns the comparison between a full non-linear solver and the numerical reference results provided by a linearized solver. Finally, for what concerns computational costs, a single unsteady simulation for this 3D case requires around 2600 seconds on the AMD 380X GPU using full mesh deformation.

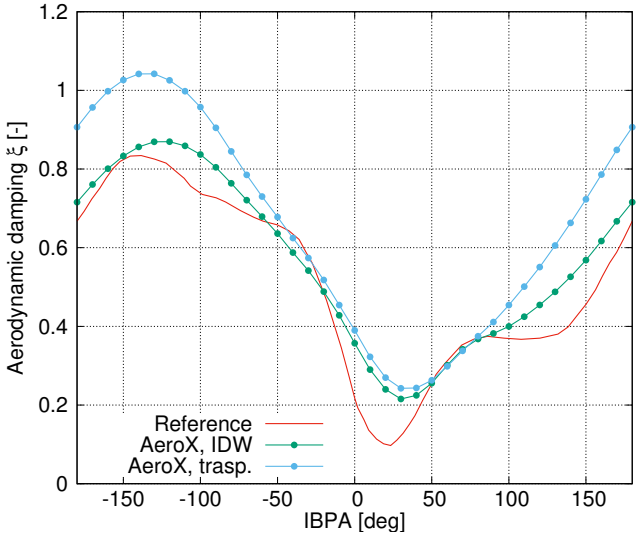


Figure 10.11: *Aerodynamic damping ξ for different IBPAs for the SC10 3D configuration, pitch d.o.f..*

10.3 NASA Rotor 67 trim

To complete the aeroelastic investigation of turbomachinery configurations, in this section the trim of the well known NASA's Rotor 67 (R67) fan blade is performed [100, 103, 129]. This is an important test case for the validation of AeroX since it involves the use of numerous different formulations simultaneously. These are represented by MRF, the modal representation of the structural behavior, IDW, aeroelastic interface, cyclic and total boundary conditions. Here the R67 is modeled as an isolated rotor. For this configuration two experimental data sets are available: one from 1989 [143] and one from 2004. This data will be used to compare results provided by compressible RANS simulations obtained with AeroX and thus perform the solver validation for this kind of configurations. Furthermore, for this transonic axial fan test case different CFD performance predictions have been published [42, 74]. The purpose of this investigation is to find the characteristic curves (performance curves, see 3.3) of the rotor, i.e. the mass flow vs. total pressure ratio and the mass flow vs. efficiency curves. The investigation will be firstly performed using a simple steady-state simulation considering the geometry to be perfectly rigid. Then the same investigation will be repeated as a static aeroelastic simulation, i.e. a trim simulation, and a comparison

between the two sets of curves will be performed in order to asses the possible advantages provided by taking into account the blade deformation under aerodynamic loads. Again, experimental data will be used to check for possible advantages provided by blade trim results. As for the previous cases analyzed, a single blade domain reduction is here adopted to reduce the computational domain under the hypothesis that the solution around a single blade is the same for the other 22 blades of the rotor. Thus, periodic boundary conditions are adopted.

Figure 10.12 shows the computational domain and its division in multiple boundaries. Differently from the previously analyzed SC10 3D configuration, here a tip clearance of $1\,mm$ is employed, separating the blade tip from the shroud wall.

MRF is employed since the rotor angular speed is $16043\,rpm$. As usual, total pressure and total temperature are enforced at the inlet boundary. In particular these values are $P_0 = 101325\,Pa$ and $T_0 = 288.15\,K$. On the outlet, instead, subsonic boundary conditions are enforced, starting with $P = 101325\,Pa$, to catch choke conditions. In order to build the entire characteristic curves, what is done is starting from computing the choke point and then gradually increase the outlet static pressure in order to find other operating points. This procedure is repeated until stall conditions. This can be done with multiple simulations starting from initial guess conditions. Here, in order to save computational time, the solver is each time re-started from the just computed performance point for which the flow conditions should be near the current point under investigation. More in detail, the solver is not re-started at all: when the solution is converged over a user-specified outlet static pressure, the solution is saved, and the outlet boundary conditions are adjusted to satisfy the new performance point conditions. This allows to save the pre-processing computational time required to read the mesh and build mesh metrics that would be instead required by truly re-starting the solver for each performance point.

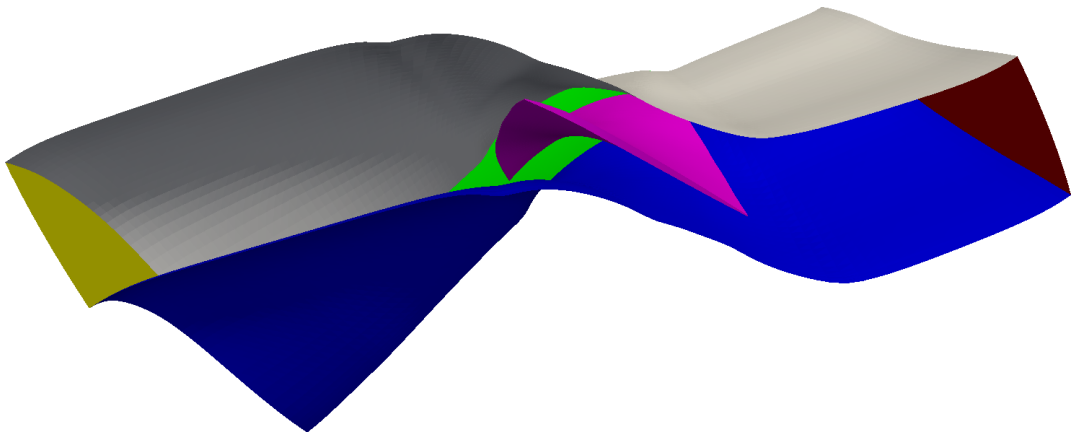


Figure 10.12: *Rotor 67 single blade computational domain. Boundaries: inlet (yellow), periodic boundaries (grey and blue), blade (purple), hub (green), outlet (red). The shroud is not shown.*

10.3.1 Structural model

As for the HiReNASD wing let's begin with the structural model. The structural mesh is composed by about $80 \cdot 10^3$ tetrahedrons (solid elements). Only the blade is modeled

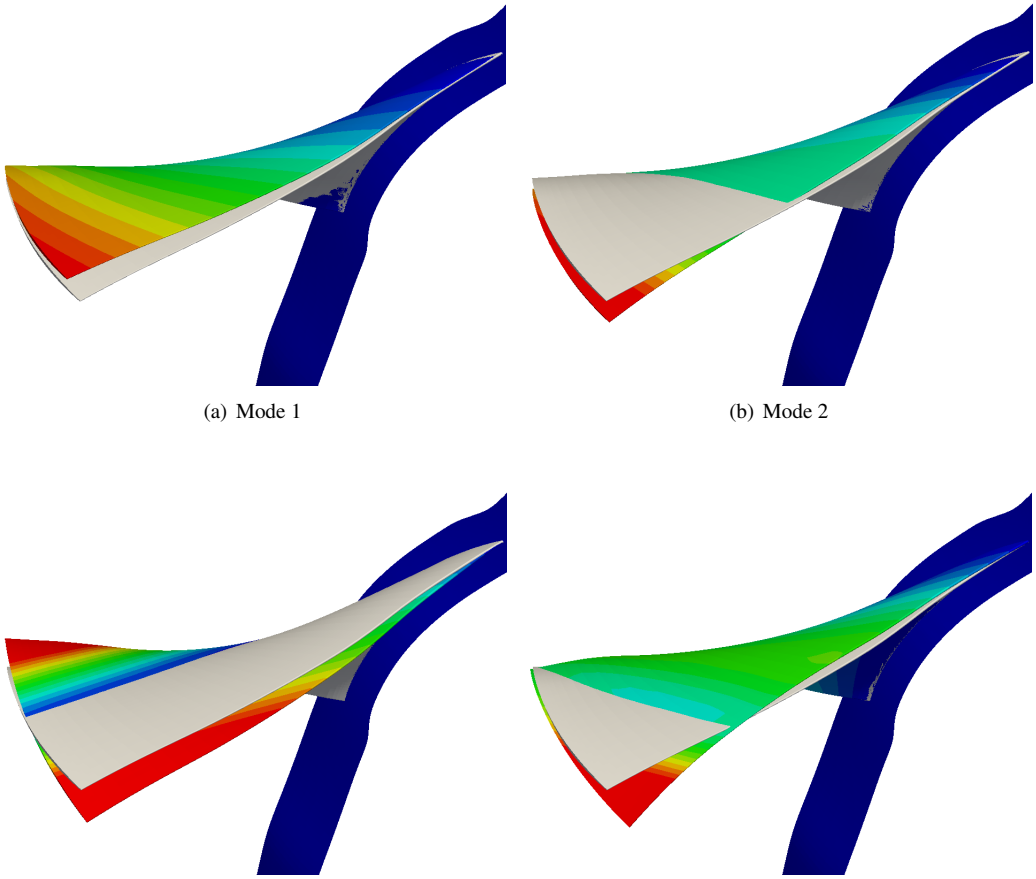
with FEM elements, while the hub and the shroud are supposed to be perfectly rigid. Thus, it is possible to directly perform the FEA on the clamped blade since there is no way for the other blades to communicate from a structural point of view. The structural analysis is performed using Code_Aster in order to extract modal shapes and frequencies. As will be presented, just the first 3 modes are potentially sufficient to perform the trim investigation in an accurate and computational efficient manner. However, 4 modes are extracted in order to further improve the results accuracy of the static aeroelastic solution. Using 5 or more modes leads to just negligible differences in numerical results. Table 10.1 shows the frequencies and the description of the first 4 modes, while figures 10.13 show the shape of those mode. Basically the first two modes are flexural while the third shape is torsional. The fourth mode is mixed. From the figures it is possible to see that the hub is considered perfectly rigid. The modal analysis is performed accounting for rotational effects.

Mode	Frequency (Hz)	Description
1	760.3	1st bending
2	2174	2nd bending
3	3146	1st torsional
4	4920	Mixed

Table 10.1: *First* 4 *modes of the Rotor 67 blade.*

10.3.2 Aerodynamic model

The aerodynamic mesh is composed by $1.1 \cdot 10^6$ hexahedra cells with a boundary layer discretization that allows $y^+ \simeq 0.5$. This kind of discretization allows the near-wall solution to be is inside the viscous sublayer. This is perfectly compatible with the blended wall treatment. Spalart–Allmaras turbulence model is adopted. Figure 10.14 shows the detail of the discretization of the R67 blade and the hub. It is noted that in the figure the actual computational domain is replicated 5 times just for visual purposes. It can be seen that the blade is well discretized both in span and chord directions. The same is valid also for the hub. Alongside the near-wall refinement, this is done in order to accurately reconstruct both compressible and viscous phenomena around the blade. In fact, the characteristic curves are computed from choke conditions up to the stall region, where separations could occur. Anyway when separations occur in the flow it is very difficult to obtain accurate solutions. Concerning the computational aspects of this investigation, the mesh is unstructured but not hybrid since the mesh is composed by 100% hexaedra cells. Thus, there is no branch divergence afflicting assembly kernels. The simulations were performed on the AMD 290X GPU for which a time/iteration/cell of $2.81 \cdot 10^{-8}$ seconds was obtained, as already discussed in 6.2.1. Considering $50 \cdot 10^3$ total pseudo time iterations required to reach the steady-state rigid solution at choking conditions, a total simulation time of $25\,min$ is required. It is noted that this is just needed to reach choke conditions starting from the initial guess. From this point to the next performance point (and so on with the other performance points requested) less iterations are required.



(a) Mode 1 (b) Mode 2

(c) Mode 3 (d) Mode 4

Figure 10.13: *Rotor 67 modes shapes, displacements magnitude from blue to red.*

10.3.3 Trim results

Here the steady aeroelastic results are discussed in a comparison with purely aerodynamic results and experimental reference data. First of all, few notes regarding the computational aspects of the simulations. The trim investigation at different user-prescribed performance points (enforcing gradually different outlet static pressures) is started from the steady-state purely aerodynamic solution at choke conditions. As said, this is obtained in 25 minutes with the AMD 290X GPU. From this solution the solver requires another 10 minutes to reach the static aeroelastic (trim) solution, again at choke conditions. From there, the trim solutions for the next performance points are computed using as initial guess the already computed trimmed performance point (thus without restarting from the purely aerodynamic steady-state solution of the correspondent outlet static pressure). This allows to speed up the computations since there is no need to completely re-start the solver, saving the time required to read the mesh and perform pre-processing operations. Furthermore the displacements of the previous trimmed performance point are also used as guess conditions. 500 purely aerodynamic explicit iterations are performed between each mesh update. It must be noted that it

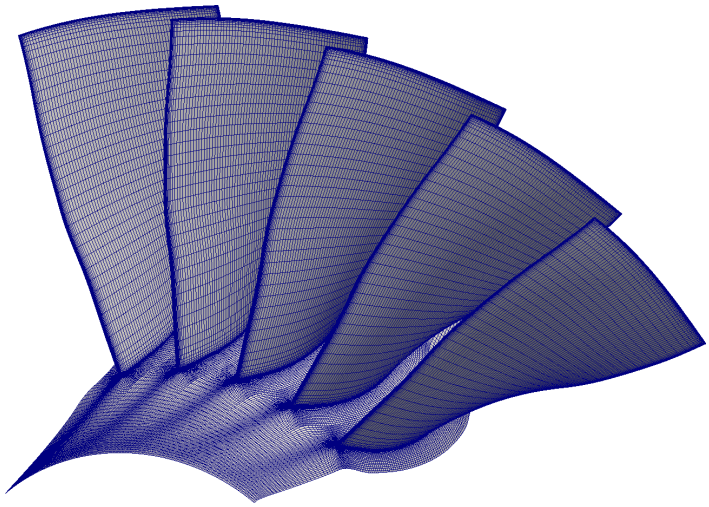
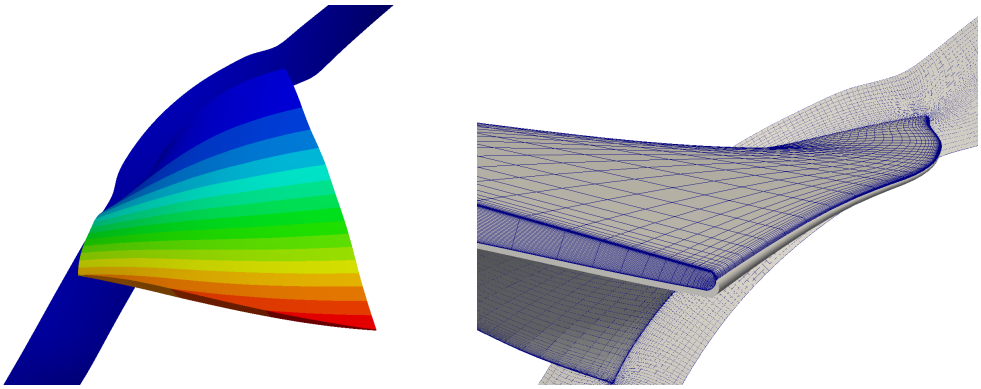


Figure 10.14: *Hub and blade mesh detail. The real computational domain is here replicated* 5 *times for visual purposes only.*

is not simple to define the total computational time required to simulate the entire performance curve of the rotor since the curve can be discretized arbitrarily with different steps of outlet static pressures, leading to different number of computed performance points from choke to stall conditions. For what concerns this investigation, in order to obtain smooth characteristic curves, both for the aerodynamic and the aeroelastic solutions, 30 performance points are used from choke to stall conditions. This is translated in a total computational time of about 6 hours for the trim solution curve. As already said thanks to the efficient GPU-accelerated mesh deformation algorithms the difference between the two computational times (purely aerodynamic and aeroelastic) is quite small. This is also due to the fact that, as will be showed, the blade displacements are relatively small with respect to what can be found for classical aeronautical cases like the HiReNASD wing presented in 8.1. Thus from the aeroelastic point of view numerical convergence is dominated by aerodynamics rather than the structural behavior or the coupling between them. This behavior was encountered also for the trim of a typical centrifugal compressor configuration [101], again using AeroX.

Now the results regarding purely CFD aspects will be presented. At choke conditions AeroX predicts a mass flow value of $1.570\,kg/s$ which is in good agreement with the experimental value of $1.589\,kg/s$, with an error of just 1.76%. Figures 10.15 show with colors and displacements the blade deformation at choke conditions. Figure 10.15(a) shows the overall displacements of the blade at the aforementioned conditions. It is possible to see that the shape is basically the same of the first mode. In fact this is the mode that provides the higher values of generalized displacements. It must be noted that the modes presented in table 10.1 are unitary-mass normalized. Thus they are directly comparable for what concerns the elastic energy content. Figure 10.15(b) shows instead the detail of the deformation near the blade tip and the blade trailing edge. It is possible to see that at these conditions the tip displacement is very small, just a fraction of the blade tip thickness. This is very different with respect to what happens with a typical aeronautical wing (e.g. HiReNASD), where the trim solution is

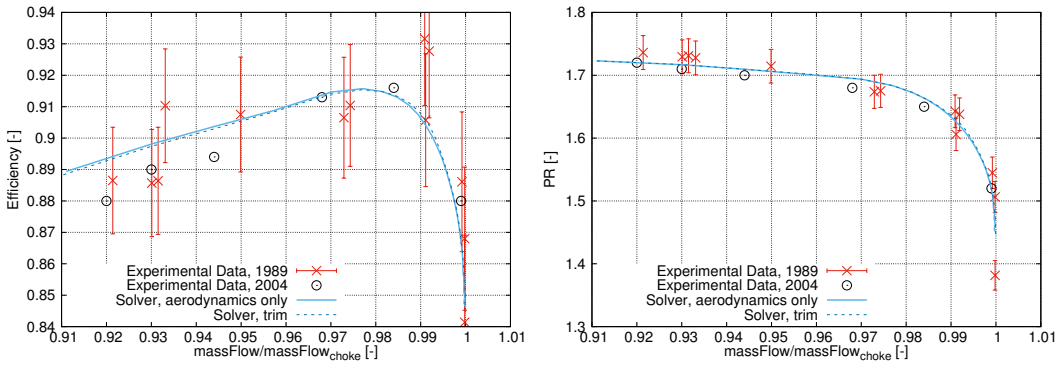


(a) Colors from blue to red

(b) True displacements of the blade, tip-trailing edge detail

Figure 10.15: *Rotor 67 blade deformation at choke conditions.*

in general characterized by tip displacements that are usually in the order or multiple of the tip airfoil thickness. The particular behavior of the R67 blade is justified by the fact that considering the material and the blade shape, the blade stiffness is higher than what can be found in a typical aeronautical wing. The blade frequencies, as an example, are one order of magnitude higher than what obtained with the HiReNASD wing. Figures 10.16 show the characteristic curves of the Rotor 67 concerning the efficiency and total pressure ratio, comparing the purely aerodynamic simulation, the aeroelastic simulation and the two sets of experimental data. The showed mass flow is normalized



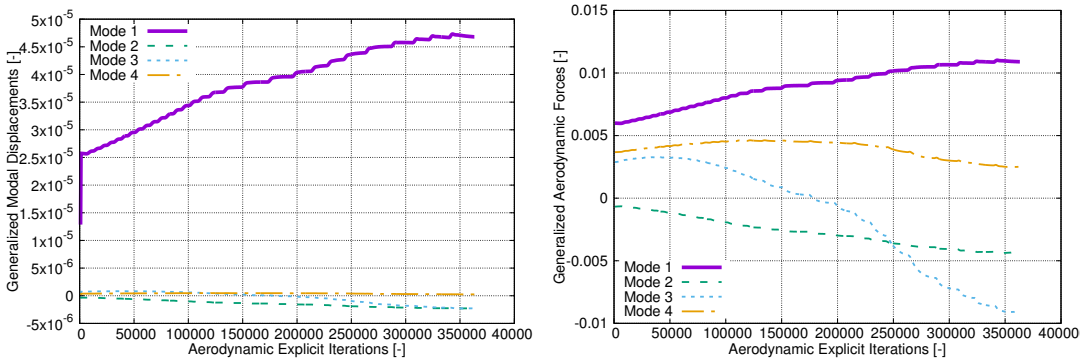
(a) Efficiency

(b) Total pressure ratio

Figure 10.16: *Characteristic curves for the Rotor 67 test case.*

by the choke value. It must be noted that the data set from 1989 presents high uncertainties. Anyway the two numerical curves are in good agreement with both sets of experimental data. In particular, the two curves are very similar to 2004 data set for what concerns the peak efficiency point while are significantly lower than 1989 measurements. The trim solution is here computed using the 4 modes previously presented. Anyway, it can be seen that the differences between the rigid and the aeroelastic curves are basically negligible. The trim solution, however, produces a slightly higher curve for the pressure ratio, especially near the peak efficiency point. In general, the small

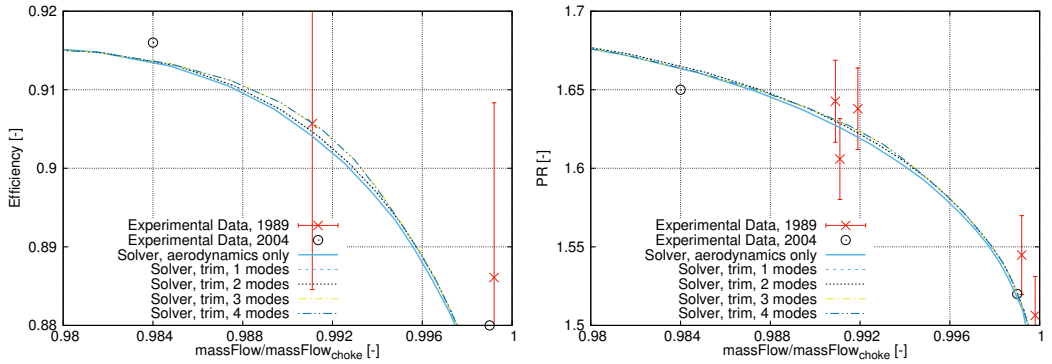
differences between the rigid and trim solution could also be inferred considering the very small displacements encountered at choke conditions. This is again due to the high blade stiffness that limits the blade deformation under aerodynamic loads. Figure 10.17(a) shows the generalized displacements ($\{q\}$) of the four considered modes during the convergence over different performance points, starting from choke conditions, up to the stall region. The curves are composed by multiple steps representing the convergence over different performance points. It is reminded that the considered modes are opportunely processed with a unitary mass normalization. Thus, knowing the generalized displacements it is possible to compute the elastic energy of each considered mode on each considered trimmed performance point. Basically almost the entire elastic energy is related to the first mode, as could be expected since in figure 10.15(b) the displacements are mainly represented by the first modal shape. The second, third and fourth modes contributions on the elastic energy are basically two orders of magnitude smaller. This justify the fact that just 3-4 modes are enough to correctly represent the structural behavior of the blade. Using just 3 modes basically the same results are obtained. From the figure it is possible to see that the generalized displacements of the fourth mode are basically negligible over the entire characteristic curve. Thus, using 5 or more modes with higher stiffness only leads to further negligible contributions in results. To provide a complete view of the trim investigation through modal representation of the structural behavior, figure 10.17(b) shows the generalized aerodynamic forces $\{Q\}$ of the four considered modes. It is possible to see that, as for the displacements, the curves are made by multiple steps representing the convergence to successive performance points. Figures 10.18(a) and 10.18(b) show the detail of the characteristic curves near the peak efficiency point for what concerns efficiency and total pressure ratio respectively. In particular, four different curves are showed, representing the employed modal representation of the structural behavior using an incremental number of low frequency modes. From the results it is possible to see that using 3 or 4 modes, as said, leads to negligible differences since the two curves are basically overlapped. Small differences are instead obtained when just 1 or 2 modes are employed. Considering these results, the modal representation can be considered converged using 3 modes. Finally figures 10.19(a) and 10.19(b) show the comparison at two span sections, 10%



(a) Generalized displacements

(b) Generalized forces

Figure 10.17: *Modal response for the trim of the Rotor 67 test case over the entire performance curves.*



(a) Efficiency

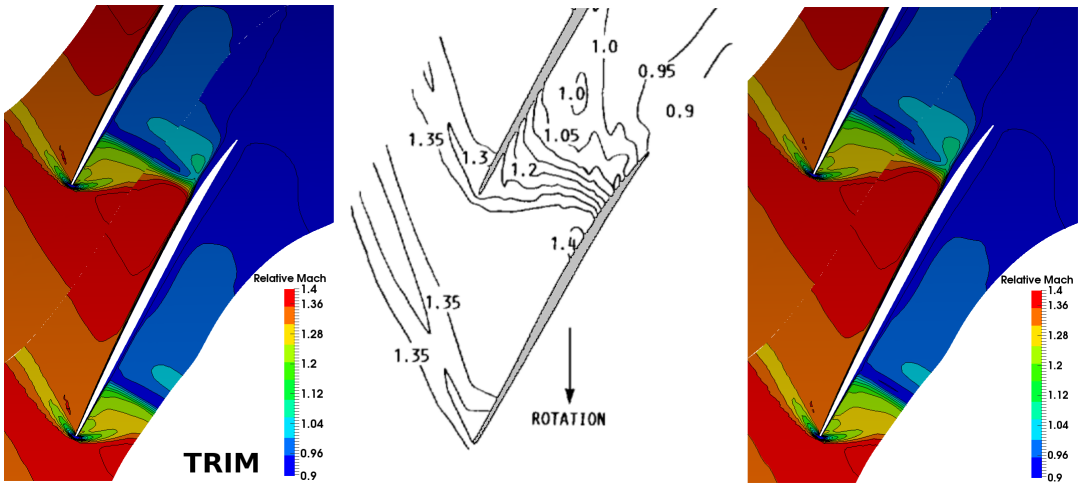
(b) Total pressure ratio

Figure 10.18: *Peak efficiency point detail of the performance curves for the Rotor 67 test case.*

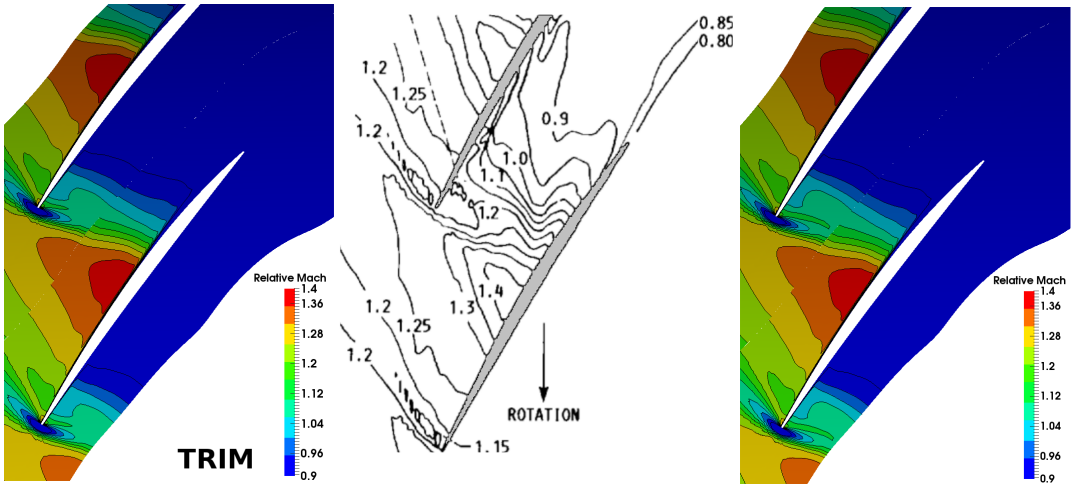
and 30% respectively, of the relative Mach number. The comparison is performed between numerical results provided by AeroX and experimental data. It is possible to see that numerical results are in good agreement with experimental data. Furthermore, negligible differences are obtained between the purely aerodynamic solution and the aeroelastic solution. This highlights the fact that not only integral quantities like performance curves show small differences between the two kind of simulations for this particular test case. Also local fields, like the relative Mach number, are characterized by small differences. Thus, it is basically possible to say that for this kind of configurations, trim solutions provide just negligible differences over classical purely aerodynamic solutions. To complete the investigation, the analysis is also performed considering an elastic modulus equal to $1/10$ of the original value. The blade shape obtained at choke conditions with the modified elastic modulus is represented in figure 10.20. It is possible to see that in this case the blade deformation is more appreciable since now it is greater than the blade tip thickness. However, it must be noted that to obtain these behavior a non-existent material is adopted in the structural modelization. Furthermore, with this magnitude of displacements, deforming the mesh is very difficult since the maximum displacements are in fact located at the tip clearance where the aerodynamic mesh cells are opportunely small in order to better represent the possible interactions between the blade (rotating) and the shroud (fixed). Thus, with this kind of blade deformations the aerodynamic mesh cells are very stretched. Nonetheless, this kind of analysis could be more and more relevant in the future due to the adoption of new technologies and applications with composite materials and 3D printing.

10.4 Open rotor blade flutter

In order to complete the validation of AeroX for aeroelastic cases with rotating components, here the flutter analysis of a typical open rotor configuration is performed. The adopted case of the SR-5 propfan is described in details in [64, 65, 78, 122]. Here the case is briefly introduced and numerical results are discussed. The investigation is performed using Euler formulation since the prime target of the investigation was a comparison between the results provided by an Euler solver and the full potential solver



(a) 10% span



(b) 30% span

Figure 10.19: *Relative Mach number comparison for the Rotor 67 test case.*

S^T with extensions for open rotors simulations. The results provided by AeroX are compared with those provided by S^T [122] and data available in the aforementioned literature. This case is quite challenging since different formulations are simultaneously involved, such as MRF, ALE, periodic BCs, IDW, RBF, modal shapes, DTS. In particular, here, the ALE formulation is used both for MRF and for mesh deformation. The geometry under investigation is a 10-blade propfan designed by Hamilton Standard in early '80s. The blade has a tip sweep of $60°$, a medium chord of $7.67\,cm$ and a span of about $30\,cm$. The propeller encountered flutter above $M_\infty = 0.7$ during the performance testing in wind tunnel and could not achieve its design point. The goal here is to predict the same flutter conditions with AeroX. The strategy adopted is the same used for the AGARD 445.6 wing, thus after the computation of the trim solution, one unsteady simulation for each considered mode is performed, enforcing the specific shape through a blended step time law. The actual flutter prediction is performed by

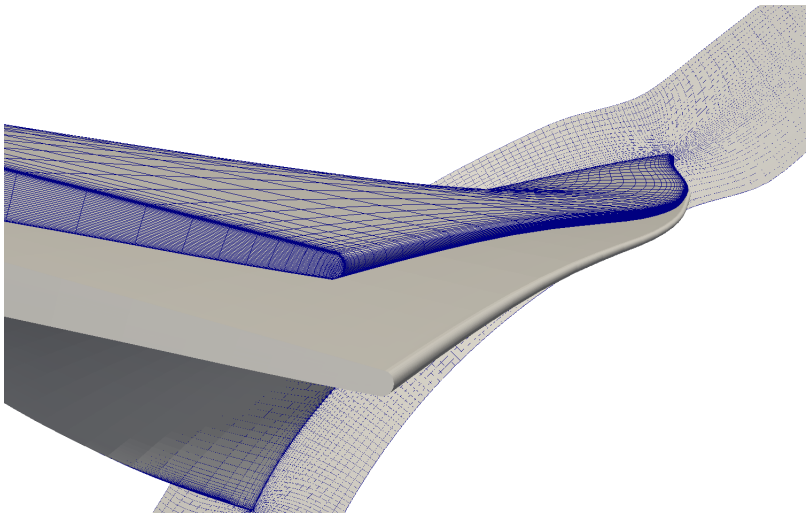


Figure 10.20: *Rotor 67 blade displacements at choke conditions with reduced elastic modulus.*

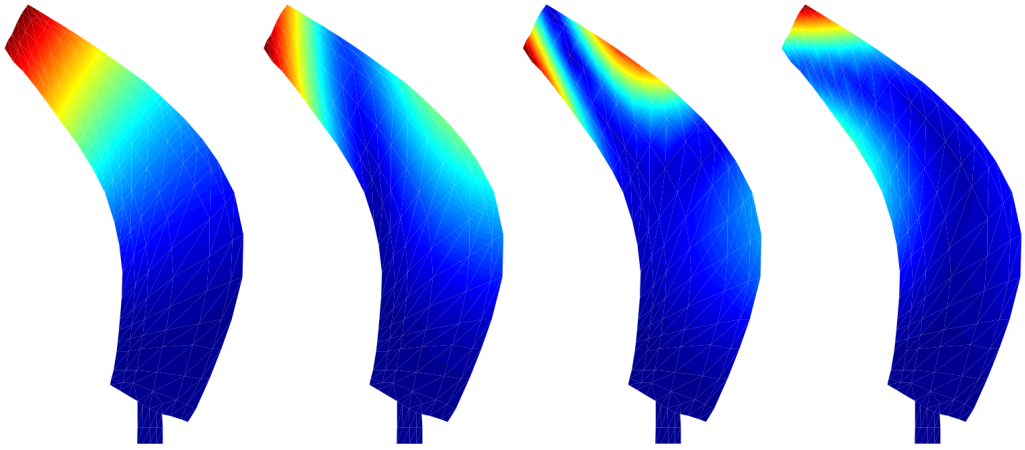
post-processing the responses of the propeller with the computation of the aerodynamic transfer matrix and the algorithm previously described.

10.4.1 Structural model

The structural model here adopted is taken from [122], thus here is just briefly presented. The blade is discretized with CQUAD4 shell elements using NASTRAN. The blade is clamped at its base. The hub is supposed to be perfectly rigid, thus the modal analysis of a single blade alone is sufficient. In fact, without the hub deformation there is no way for different blades to communicate from a structural point of view. Thus there is no possible mode that could involve different blades experiencing different deformations. This is basically the same structural modelization strategy adopted for the Rotor 67 fan blade 10.3. Table 10.2 shows the first 6 modal frequencies obtained with the blade discretization, considering the blade rotating at $6800\,RPM$. It is important to notice that here the influence of centrifugal loads is non-negligible, thus the modal analysis is carried out considering the contributions provided by the blade rotation. Figures 10.21 show the shape of the first 4 rotating modes. Although it is more difficult to describe with words the modes shapes with respect to what happens usually with aeronautical wings, it is possible to say that the first two modes are mainly bending modes while the third is basically a torsional mode.

Mode	Frequency (Hz)
1	160.9
2	287.0
3	595.9
4	670.7
5	863.2
6	1013.4

Table 10.2: *Propfan modal frequencies at* $6800\,RPM$.



(a) Mode 1 (b) Mode 2 (c) Mode 3 (d) Mode 4

Figure 10.21: *First 4 modes shape at* $6800\ RPM$.

10.4.2 Aerodynamic model

Figure 10.22 shows the near-blade mesh discretization. As said, this investigation is performed using Euler formulation, thus there is no need of a boundary layer discretization. The mesh is generated with GAMBIT and showed in figure 10.22. It is possible to

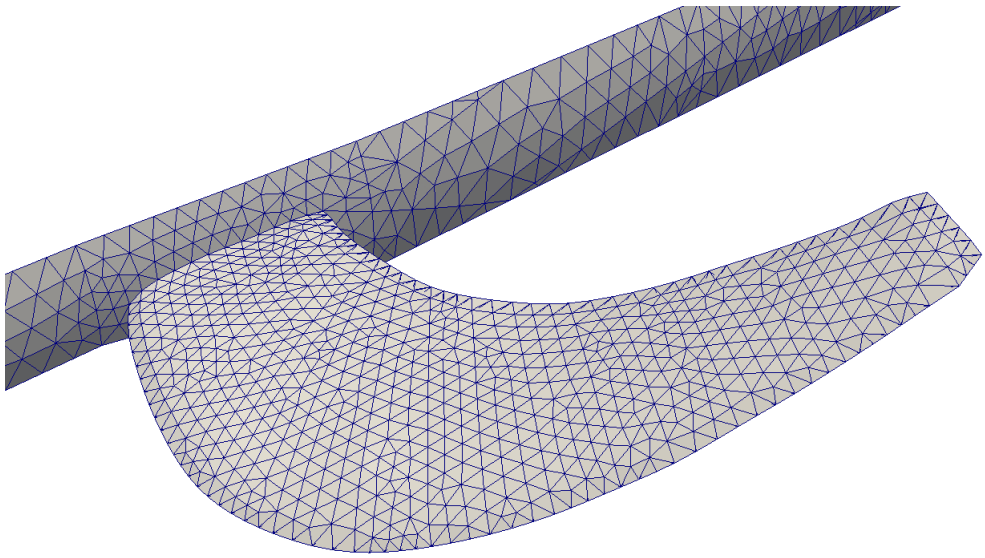


Figure 10.22: *Near-blade mesh detail.*

see the blade attached to the hub. As for the open rotor configuration investigated in 9.3, periodic boundary conditions are adopted in order to reduce the computational domain to a single blade sector. The investigation is carried out supposing that all the blades

respond the same way, or in turbomachinery terms, with a null Interblade Phase Angle (IBPA). This way the configuration is supposed to be perfectly axial-symmetrical, not only from a purely structural point of view but also for what concerns aerodynamics and aeroelastic response. The mesh is basically a sector of a cylinder and at the two bases inlet and outlet boundary conditions are enforced. On the external surface of the cylinder characteristics-based boundary conditions are employed, while on the blade and the hub slip boundary conditions are enforced. The mesh is composed by $168 \cdot 10^3$ tetrahedra while the blade is discretized with about 2700 wall faces. This is enough to perform an inviscid simulation and allows to drastically reduce the computational time with respect to a full compressible (U)RANS analysis.

Two different parameter sets are adopted for the analysis and are showed in table 10.3. These two cases are referred as case 2 and case 3. For these conditions reference data is available to perform the validation of the solver.

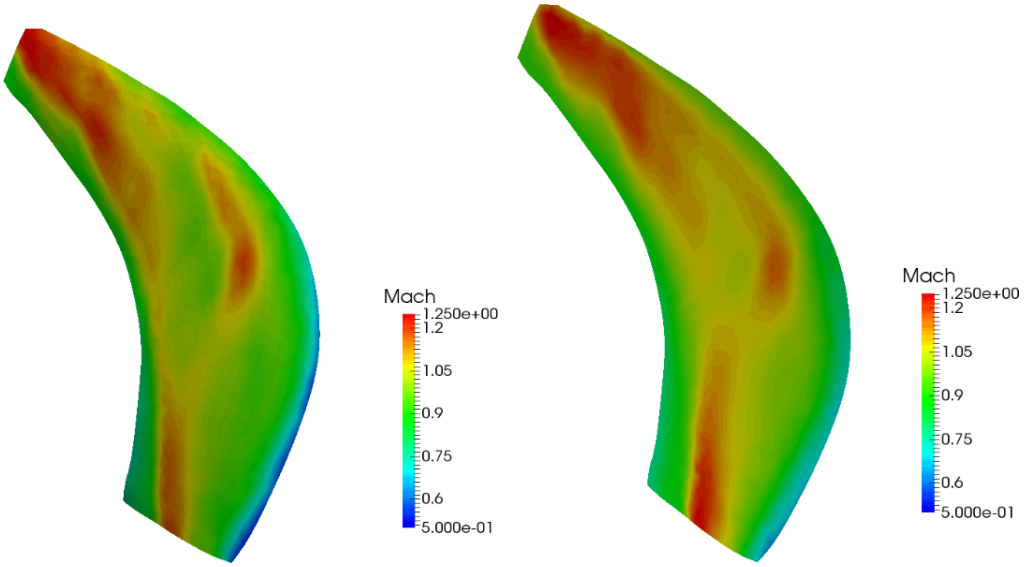
Case ID	RPM	No. of blades	$\beta_{75\%}$	V_∞ (m/s)	Density (Kg/m^3)	M_∞
2	6000	10	69.0°	263.86	1.0067	0.80
3	6800	10	69.0°	237.13	1.0433	0.70

Table 10.3: *Operating conditions for the different cases.*

10.4.3 Trim results

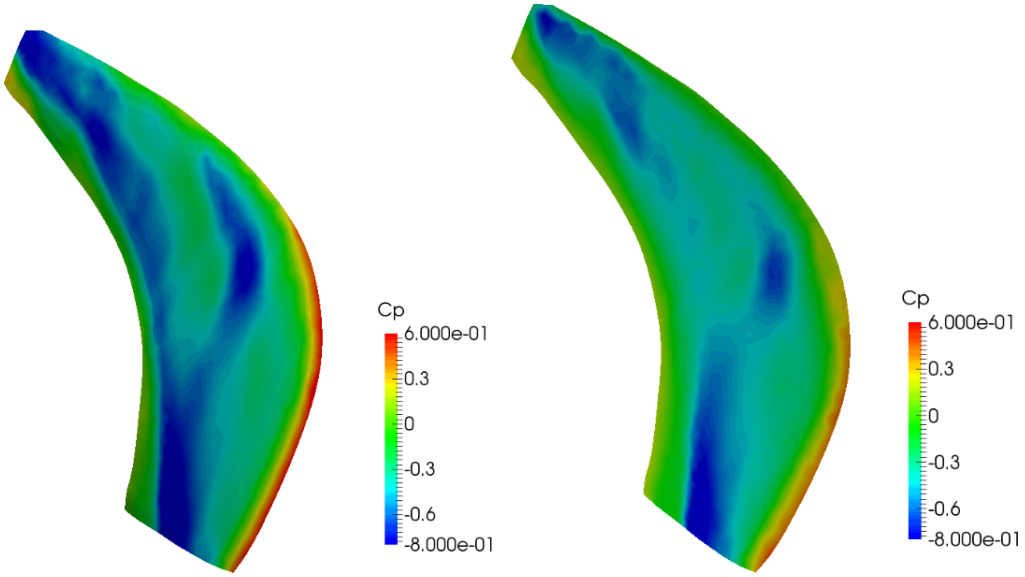
Before performing the unsteady computations for the flutter analysis, the steady-state solution has to be computed, in order to have a solution from which starting to perform the small perturbation simulations. In theory the solution from which the unsteady simulations are started should be a true aeroelastic steady solution, characterized by the deformations provided by centrifugal and aerodynamic loads effects. However, as explained in [122] for this geometry aerodynamic loads contributions are negligible. Thus, the geometry is deformed only by centrifugal loads and the modal shapes and frequencies adopted for the unsteady simulations are directly taken from the structural modal analysis, considering centrifugal effects only. Roughly speaking, the trim solution here is obtained by executing AeroX to find a simple purely aerodynamic steady-state solution, without any kind of mesh deformation. The adopted geometry for the aerodynamic mesh is however the one affected by centrifugal loads. For the conditions of table 10.3 the blade tip reaches transonic conditions. This can be easily seen in figures 10.23 where a comparison between AeroX and S^T is performed for the conditions of case 3. From the figure it is possible to see that the results provided by the two solvers are in agreement and transonic conditions are reached in some regions of the blade, especially near the tip, where the effects of the blade rotation are maximized. Due to the different rotating speeds and the asymptotic flow speed, the angle of attack at 75% of the span is respectively $8°$ and $14°$ for the case 2 and case 3. Figure 10.24 shows the pressure coefficient distribution over the blade for case 3. Again the results provided by the two solvers are in agreement with each other.

Using the AMD 380X GPU the trim solution is obtained in 88 seconds with a time/iteration/cell of $2.975 \cdot 10^{-8}$ seconds. It must be noted that for this GPU, the adopted mesh size is not enough to fully exploit the available computational power. Nonetheless, the mesh discretization is good enough for an inviscid simulation and to perform



(a) AeroX, Euler formulation (b) S^T, full potential formulation

Figure 10.23: *Mach number comparison for case 3.*



(a) AeroX, Euler formulation (b) S^T, full potential formulation

Figure 10.24: *Pressure coefficient comparison for case 3.*

a direct solution accuracy comparison with the CPU-based S^T solver using the same mesh.

10.4.4 Flutter results

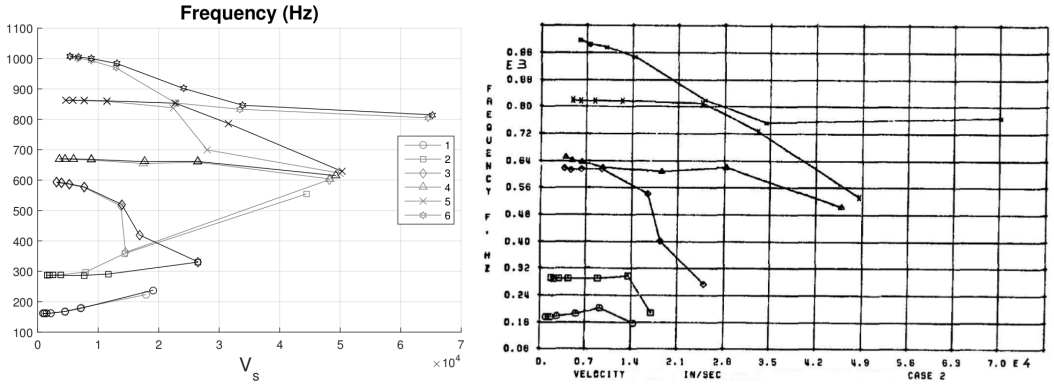
The last step is to re-start the solver from trim conditions and perform unsteady simulations by enforcing each blade modal shape through a blended step time law. Results details (e.g. aerodynamic modal response and aerodynamic transfer functions) for AeroX and S^T are available in [122]. Here just the overall results for the flutter prediction are presented. Table 10.4 shows the reference chord and mean sweep of the blade that are used for the $V-g$ and $V-\omega$ flutter diagrams. $V_S = V_\infty \cdot \cos(\Lambda)$ instead of V_∞ is

Case ID	Chord c (m)	Sweep angle Λ (deg)
2	0.07366	50.109
3	0.07367	50.138

Table 10.4: *Geometric parameters for the different cases.*

adopted for the flutter investigation. As for the AGARD 445.6 the blended step time law parameters are computed with what explained in 2.2.2 and flutter is computed as a post-processing procedure after performing the unsteady analyses, triggering for each independent simulation a different modal shape.

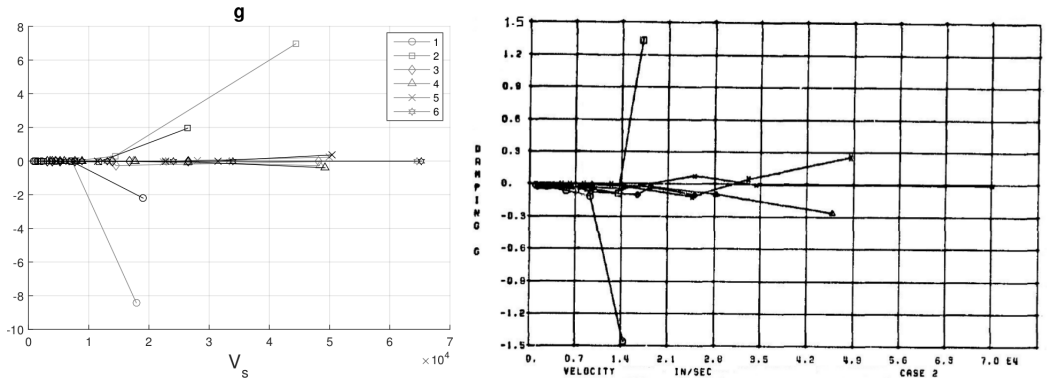
As said, the first 6 rotational modes are here used for the flutter investigation. The correspondent aeroelastic modes frequencies and damping values for case 2 are showed in figures 10.25 and 10.26 respectively. For what concerns conditions of case 3, the $V_S-\omega$ and V_S-g diagrams for the 6 aeroelastic modes are instead showed in figures 10.27 and 10.28 respectively. As explained at the beginning of this investigation, the blades encountered flutter in the experimental setup for these conditions and geometry. This is correctly predicted by the numerical results provided by both solvers. Flutter is encountered with the instability of the second aeroelastic mode, which is a bending mode. The aeroelastic instability manifests itself basically in a bending-torsional shape since the frequencies of the second (bending) and third (torsional) aeroelastic modes tend to get closer. In general, from the figures it is possible to see that numerical results provided by inviscid AeroX simulations are in agreement with those provided by S^T solver and with literature reference data.



(a) AeroX: grey, S^T: black

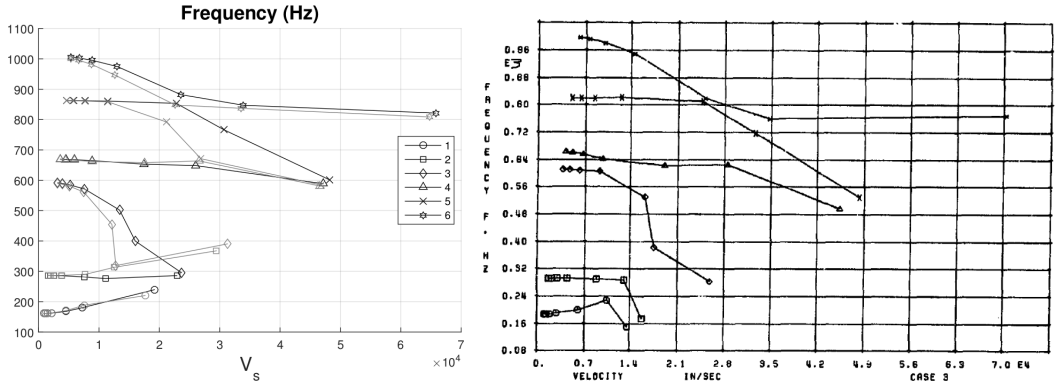
(b) Reference data

Figure 10.25: *Case 2, frequencies comparison for the 6 aeroelastic modes.*



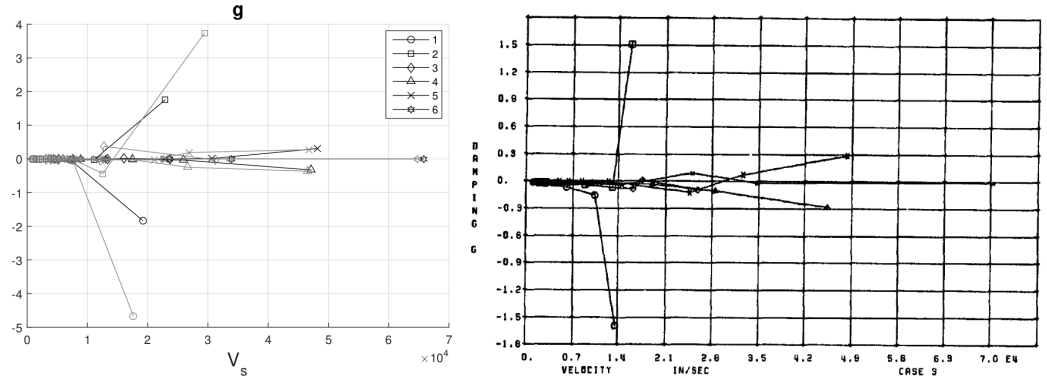
(a) AeroX: grey, S^T: black (b) Reference data

Figure 10.26: *Case 2, damping values comparison for the 6 aeroelastic modes.*



(a) AeroX: grey, S^T: black (b) Reference data

Figure 10.27: *Case 3, frequencies comparison for the 6 aeroelastic modes.*



(a) AeroX: grey, S^T: black (b) Reference data

Figure 10.28: *Case 3, damping values comparison for the 6 aeroelastic modes.*

CHAPTER 11

Concluding Remarks

In the present work we illustrated the design and implementation of a GPU-accelerated aeroelastic compressible RANS solver with turbomachinery and open rotors extensions. From the purely computational point of view different goals were achieved:

- the possibility to exploit the single precision performances of cheap gaming GPUs. This is done in order to avoid the necessity of using expensive HPC GPUs that nonetheless exhibit higher double precision performances, bigger global memory and ECC support. The implemented algorithms are opportunely designed to save memory (thanks to explicit time stepping with convergence acceleration) and guarantee accurate results with single precision support (thanks to non-dimensional equations). However, thanks to the AeroX flexibility, the solver is natively compatible with HPC GPUs and their aforementioned features;
- the backward compatibility with CPUs. Thanks to OpenCL abstractions for parallelism and the freely available implementations from different vendors, the solver is natively compatible with both CPUs and GPUs. This allows an easier debugging stage and to asses the advantages provided by the GPU acceleration through benchmarks;
- the native compatibility with a wide range of devices, from CPUs to GPUs and APUs. Devices from the nowadays most important vendors (Intel, AMD and NVIDIA) were successfully employed to perform computational benchmarks and to obtain CFD/FSI results;
- the interface between the solver and OpenFOAM, allowing an easy pre-processing and post-processing phases, guaranteeing the compatibility with hybrid unstructured meshes generated with the most important modern mesh generation tools;

In this work the effort was balanced between computational aspects and CFD/FSI formulations. The idea was to choose and adapt CFD/FSI formulations to the GPU architecture. For this purpose different formulations and strategies developed in [127, 136] were opportunely tuned and translated into OpenCL kernels in order to exploit GPU acceleration. Around these low-level schemes, turbomachinery and open rotors extensions were built in order to expand the application fields of AeroX. Thus, from the purely numerical formulations point of view, the following formulations were successfully implemented, and explained in their GPU-oriented tuning:

- Hybrid unstructured meshes handling;
- Roe convective fluxes with high resolution;
- Spalart–Allmaras and SST turbulence models with automatic wall treatment;
- Mesh deformation for steady and unsteady aeroelastic analyses;
- Turbomachinery and open rotors extensions;

Concerning the purely aerodynamic and aeroelastic point of view, the solver was validated using well-known and recent-trend cases. The 2nd Drag Prediction Workshop was adopted to asses the capability of the solver to perform steady-state simulations of aircraft shapes and to correctly catch both compressible and viscous effects in transonic and high Reynolds number conditions. A typical open rotor geometry, investigated by Stuermer, was adopted to validate the solver for steady-state solutions. Results were also compared with the recently developed full potential implicit solver S^T [115, 122] capable to provide accurate and fast solutions for this kind of geometry. Concerning turbomachinery, the well-known Goldman blade and the 1.5-stages Aachen turbine were analyzed to complete the validation of AeroX for steady-state analyses. Then, the focus was posed on aeroelasticity, starting with the solver validation with the well-known HiReNASD and AGARD 445.6 wings for trim and flutter respectively. Then, the most recent challenge in flutter prediction, i.e. the BSCW wing for the 2nd Aeroelastic Prediction Workshop, was accepted. The solver demonstrated the capability to provide accurate and fast results by using just a single cheap desktop computer with a mid-range gaming GPU. Another challenge was represented by the trim of the Rotor 67 fan blade due the apparent lack of literature results for such kind of investigation. The aeroelastic analyses were finally concluded by the flutter prediction of the SR-5 propfan blade, in a comparison with the S^T solver.

This work can be viewed as a step forward after the work of Serioli and Romanelli [136] and Romanelli [127] with their effort in the development of AeroFoam aeroelastic solver. Despite the specific test cases under investigation, the idea that is still pursued is to try to provide even cheaper, fast but accurate solutions at the very beginning of the design phases of the typical aeronautical component, raging from wings/aircraft to turbomachinery and open rotor blades. However, in order keep pursuing this goal, some effort is still required for the following challenges, regarding both the both computational and aeroelastic aspects:

- Extension to a fully hybrid shared/distributed memory architecture, using MPI and the OpenFOAM support for domain decomposition. This is useful for different reasons as will be explained in next points;

- Exploitation of the most recent OpenCL 2.x features like shared memory (particularly for APUs) and C11 atomics extensions;

- Exploitation of the capabilities offered by other accelerators like Intel Xeon Phi, through some code porting (OpenMP + AVX-512). A porting of AeroX has been completed. Preliminary tests showed that using a developer workstation equipped with an Intel Xeon Phi 7210 ($1.30\,GHz$, 64 cores, 256 threads, $16\,GB$ MCDRAM + $96\,GB$ DDR4) an average time/iteration/cell of $5.0 \cdot 10^{-8}\,s$ can be achieved using SP. This device has around $5 - 6\,TFLOPS$ SP computational power. DP computational power is halved. The porting has been done exploiting OpenMP for multi-threading and compiler auto-vectorization for AVX-512 instructions.

- Exploitation of the OpenCL heterogeneous computing capabilities. The idea is to allow the solver to automatically split work in a balanced way among the available devices, using different CPUs, GPUs and accelerators simultaneously. Nowadays, in fact, it is possible to find clusters that exhibit multi-core CPUs installed alongside GPUs on the same node. Eventually it would be possible to exploit the different device architectures to assign each job to the most appropriate kind of device, taking advantage of multiple OpenCL queues and concurrent kernel execution or just the OpenFOAM MPI-based framework, assigning one process to each device. OpenFOAM allows to control domain decomposition assigning different number of cells to different subdomains. This can be exploited to balance work distribution when the available hardware is composed by devices with different computational power and memory. Another advantage of this approach would also be to combine devices from different vendors (e.g. an NVIDIA GPU with an AMD GPU). Preliminary tests show that combining the computational power of the GPU cores of the AMD 7700K APU with the GPU cores of the AMD 380X GPU it is possible to obtain a time/iteration/cell of $5.4 \cdot 10^{-8}\,s$ using a $1 : 5$ subdomains cells ratio on a test case. For the same case the AMD 380X ($4000\,GFLOPS$ SP) alone achieves a time/iteration/cell of $6.4 \cdot 10^{-8}\,s$ and the GPU cores of the AMD 7700K APU ($550\,GFLOPS$) alone achieve a time/iteration/cell of $4.2 \cdot 10^{-7}\,s$. It is clear that even if the two devices are quite different from a performance point of view, it is possible to reduce the overall time/iteration/cell by combining them. Combining an NVIDIA GTX 560Ti GPU ($1300\,GFLOPS$ SP) with an AMD 380X GPU ($4000\,GFLOPS$ SP) a time/iteration/cell of $3.4 \cdot 10^{-8}\,s$ was achieved on the DPW2 test case, while for the same case the AMD 380X GPU alone achieves a time/iteration/cell of $4.5 \cdot 10^{-8}\,s$.

- New addressing strategies to improve memory coalescing and reduce branch divergence, in order to reach even higher GPU efficiency;

- A more complete investigation of the recent trends in open rotors and turbomachinery, especially for what concerns open rotors and CROR and ORCs;

- Extension to a wider range of multi-physics applications (e.g. thermoaeroelastic analyses, investigation of indirect effects of potential drag reduction technologies, geometry automatic optimization);

- Investigation of the state-of-the-art techniques related to parametric computing and reduced order methods. The idea is to further speed-up simulations by combining hardware/software-based strategies (like in this work) with the aforementioned methods. This would allow not only to save computational time but also to increase the complexity of the simulations.

APPENDIX $\mathcal{A}$

Introduction to parallel computing

A.1 Introduction to parallel computing

Internet nowadays is a great and reliable source of informations for what concerns programming and computer science in general. On the internet it is possible to find free tutorials and examples, free ebooks and free tools to learn and use parallel programming in order efficiently solve different kinds of problems. Even on YouTube it is now possible to find good tutorials about programming. Thus, it is important to understand that in computer science, internet is a great source, like any other paper and article. This chapter has been written mainly using the information from [25] and by mixing information from websites like wikipedia.org [36], stackoverflow.com [31] and other sources like [66].

Nowadays parallel computing is basically everywhere, from servers providing web services, supercomputer executing simulations and small smartphones running multiple apps at the same time. The focus of this work is the exploitation of the modern GPUs computational power to accelerate CFD/FSI simulations. In numerical simulations many algorithms and schemes naturally exhibit parallelism. This is true for a wide selection of research fields like physics, engineering, biology, economy, medicine, cryptography. The growing interest in parallel computing in recent decades, both on hardware and software sides, has been particularly affected by the reaching of technological limits of serial architectures. After the race to the GHz of the CPUs in the 2000s, the technological and economical focus has been moved to the multi-core processor approach both in server and desktop applications. The technological limitations associated with serial processors are mainly concerned with the fact that the information in the CPU propagates at "just" a speed in the order of light speed. Is is therefore necessary to reduce the distance between CPU elements to increase performances. However, there are physical and technological limits that have been reached for the

miniaturization of CPU elements: the most recent CPU architectures uses nanometers scales. These limits are such that it is economically and technologically more convenient to focus on parallel architectures. There are several advantages in using parallel computing:

- time saving: throwing more resources at a task will shorten its time to completion, with potential costs savings. This is very important in numerical fields. As an example weather prediction simulations have strictly solution time requirements in order to be useful;
- problem size: many problems are too large to be handled by a single computational unit, both for what concerns pure computational power and memory requirements. For this reasons supercomputers and clusters are employed;
- concurrency capabilities: with a parallel architecture it is possible to perform true multitasking. This is very important in server applications, e.g. to provide web services to thousands of users simultaneously;
- distributed resources: in parallel architectures it is possible to distribute tasks to resources that are not physically close. As an example, in BOINC [18] projects thousands of volunteers spread over the planet are contributing to solve problems by offering their computational power;

A.2 The GPGPU way

Here, first an introduction to GPGPU (General-Purpose computing on Graphical Processing Units) is provided as it is the main approach adopted in this work to accelerate numerical computations. More details are available in chapter 4. The main idea behind GPGPU, as the name suggests, is very simple: using the graphical processor to perform general purpose numerical computations instead of graphical computations. The main reason of the success of GPGPU is given by the total computational power that modern GPUs exhibit. In fact, when talking about theoretical floating point performances, GPUs are up to one or two orders of magnitude more powerful than CPUs. This is due to the fact that GPUs have hundreds, thousands of cores and their architectures are specifically designed to perform graphical computations, where the same operation is performed on large data sets. This does not generally mean that one or two orders of magnitude speed-ups can be easily achieved since numerous aspects have to be considered when talking about GPU parallelization. In order to obtain high speed-ups the implemented algorithms must be opportunely designed and optimized to exploit the underlying GPU hardware architecture, minimizing the possible bottlenecks. This is due to the fact that from the hardware point of view GPUs architectures are very different from CPUs architectures since they are designed for different purposes. In fact, GPU architectures are tuned for SIMD/SPMD/SIMT algorithms, while CPUs are more general purpose. Programming on GPU is more difficult than programming on CPU since from the very beginning of the code developing, the programmer has to think about what is going to happen on the GPU processor and memory at each line of the code. This is due to the fact that it is very easy to degrade the overall speed-up of a GPU-accelerated application with just few lines fo code due to possible bottlenecks. This

could also potentially mean that if the algorithm is not opportunely tuned for the GPU architectures, the GPU execution would results in computational times that are orders of magnitude bigger than what can obtained with the correspondent CPU execution. Another different kind of problems in GPGPU is related to debugging and profiling. Debugging on GPU is more difficult with respect to what can be done on CPU. Thinking about a typical linux programming machine, tools like gdb, valgrind and gprof cannot be directly employed with GPU code. The situation is problematic with GPU since every vendor have its own debugging/profiling framework and in general debugging a C-like GPU code requires more time than debugging a C-like CPU code. Since there is no operating system kernel running on the GPU processor, it is very difficult to catch buffer overflows or floating point errors. GPGPU is still a relatively new field with respect to CPU programming and from both hardware and software point of view the situation changes basically every year with the presentation of new hardware architectures increasing performances and introducing new features. Despite the general effort that a programmer has to spend when programming a GPU, it must be noted that GPGPU provides advantages for peculiar numerical fields, like CFD, finance, FEA, cryptography. Not all algorithms can be parallelized on GPU, especially algorithms that for their nature are intrinsically serial, like most of the operations performed by an operating system. Thus, GPGPU is not aimed to replace classic CPU programming. GPGPU is aimed to assists CPU executions for some peculiar numerical computations. This is why it is often possible to read something like "offloading the work to the GPU" or "accelerating with GPUs". In fact, as mentioned, since there is no kernel running on GPUs, a CPU is always needed in a GPGPU application, at least to organize and enqueue work to be sent on the GPU. Furthermore the most powerful supercomputers available today in the world (see Top 500 [32]) exhibit hybrid/heterogeneous architectures, where multiple nodes of multi-core CPUs are combined to GPUs or other kind of accelerators (e.g. Xeon Phi) to accelerate algorithms wherever is possible or convenient. It is again reminded that multi-core CPUs assembled in multi-node clusters are still required since some algorithms cannot be efficiently parallelized on GPUs and in any case GPUs requires CPUs to feed them with instructions and data streams. Talking about cluster and supercomputers, GPUs are employed for other very important reasons: costs and power efficiency. This may not seem a really big problem for a single computer desktop but when the power consumption is in the order of MW, it could be a determinant factor. In fact, if an algorithm could be easily and efficiently parallelized on GPU this could potentially mean that a single GPU can substitute numerous muli-core CPUs. Considering that the GFLOPS to power consumption ratio of modern GPUs is usually one order of magnitude (see 4.3(a) [2]) higher than modern CPUs the advantages are clear. This could potentially mean that a single desktop workstation with one or multiple high-end GPUs could potentially substitute a small cluster if the code is properly tuned, with evident advantages in term of electricity bills related to the power consumption of computational hardware itself and the eventual air conditioning system.

A.3 Flynn Taxonomy

The Flynn taxonomy [67] is here introduced. Flynn taxonomy is basically a way to classify serial and parallel architectures with respect to their capabilities to handle a single or multiple streams of instructions and data. Considering the two streams of data and instructions and their possible states, single and multiple, a total of four combinations are possible: SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instructions Single Data), MIMD (Multiple Instructions Multiple Data).

SISD

This is the most simple architecture. A single stream of instructions is applied to a single stream of data. This is basically what happens in a serial CPU execution, when no kind of parallelism is employed in the code. In this case the execution is deterministic, i.e. executing multiple times the SISD compiled/parsed code the results are exactly the same provided that the same data is used for each execution. It must be noted, however, that there is still no guarantee that the instructions are executed by the CPU with the same order provided by the original source code, since compiler and CPU instructions reordering and further optimization could be employed to increase performances. Furthermore, instruction level parallelism is usually employed to improve single-core CPU performances by overlapping CPU operations in order to exploit multiple CPU components at the same time. Figure A.1 shows the SISD approach: data A and B are loaded from memory, C is then computed and finally stored back to memory.

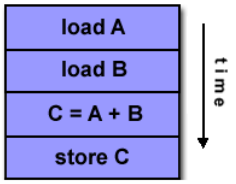


Figure A.1: *SISD architecture [25]*

MISD

With this combination of data and instruction streams, multiple operations can be performed on the same data. This is the first example of parallel architecture. However, this strategy is rarely adopted since there are very few problems on which this kind of parallelism results useful. Basically, what happens is that multiple computational units are executing independent instruction streams on the same shared data stream at the same time. This is depicted in figure A.2 where it is possible to see that n instruction streams are performing different operations on the same data (A(1)) read from memory.

SIMD

SIMD is the first useful example of parallel architecture. With SIMD the same instruction stream is performed by different computational units on different data streams.

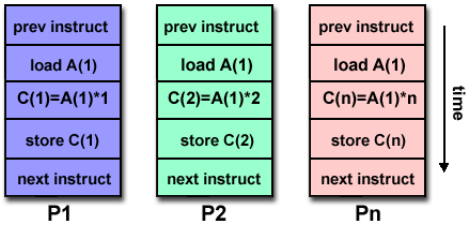


Figure A.2: *MISD architecture [25]*

This is usually what needed in numerical applications where the same operation have to be performed on large data sets. As an example, this kind of parallelism can easily speed up the the sum of two vectors, since the same operation (sum) has to be performed on each couple of numbers. Figure A.3 shows the typical instruction and data streams pattern of SIMD parallelism: the same stream of instructions is performed by each of the n computational units. However the different units are handling different data, A(1) for the first unit, A(2) for the second and so on. Today architectures exhibit SIMD capabilities in different ways. As an example, in a multi-core CPU with 4 cores it is possible to program each core to perform the same set of instructions applied to different data streams. However, with modern CPUs, SIMD parallelism can be exploited in another way, by using specialized instructions and registers available on the underlying hardware architecture. As an example, SSE (Streaming SIMD Extensions, in all versions) and AVX (Advanced Vector eXtensions) instruction sets are used in x86 and x86-64 architectures to accelerate SIMD operations. It must be noted that with SSE/AVX each single core of the CPU is capable to perform SIMD operations (e.g. operate on 4 floats at the same time with SSE). Thus it is also possible to further speed up SIMD-like algorithms by exploiting SSE/AVX and multiple cores at the same time. GPU architectures, exhibit SIMD-like parallelism, thus they are well suited for data-type of parallelism. However, GPU cores offers more the just pure SIMD parallelism.

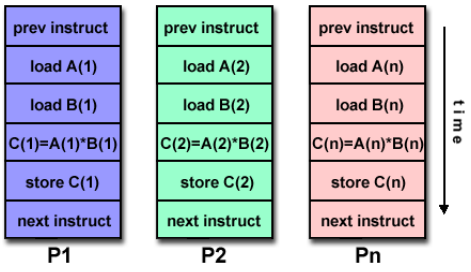


Figure A.3: *SIMD architecture [25]*

MIMD

This is the most general purpose kind of parallelism in the Flynn taxonomy. Basically MIMD means that each computational unit can perform different operations on different data at the same time. More specifically, different instruction streams are processed by different computational units and applied to independent data streams. Modern multi-core CPUs exhibit this kind of parallelism. This is thus the most popular

kind of parallelism nowadays, thanks to its advantages. As showed by figure A.4 the n computational units are performing completely different operations on independent data sets. This kind of parallelism is helpful for any server, desktop or mobile devices, where the operating system has to perform multiple different operations at the same time (e.g. browsing the internet while listening music). This kind of parallelism is well suited for task parallelism where different tasks have to be performed at the same time on different data. This is useful e.g. when a server needs to process different requests from multiple users concurrently. Numerical applications can also take advantage from MIMD parallelism: as an example certain computational units could be used to assembly a matrix and the other could be used to assembly residuals. An important aspect is that with MIMD determinism could be lost, meaning that starting from the same initial data it could be possible to obtain slightly different results. This usually happens with parallel reduction operations due to finite-precision nature of floating point arithmetic and the fact that with different executions the order of floating point operations could be slightly different, leading to the aforementioned little differences in results.

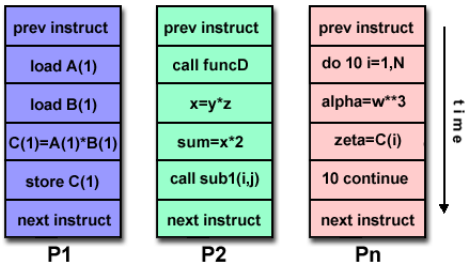


Figure A.4: *MIMD architecture [25]*

SPMD

Single Program Multiple Data is a definition outside the original Flynn taxonomy. SPMD basically means that the same program is executed in parallel on different data by different computational units. Here the key difference between SPMD and SIMD/MIMD is the fact that the same program and not the same instruction stream is shared between the computational units. Basically different computational units could be in different points of the same shared program at the same time. This usually happens when the code has branches. Thus, this is different from true SIMD since more freedom is allowed for what concerns instruction streams on different computational units. However SPMD is also different from MIMD since this freedom is limited to the single shared program. SSE/AVX instruction sets of CPUs does not allow this kind of parallelism but only true SIMD. GPUs instead satisfy the SPMD model allowing each GPU core to execute the same program on different data at the same time. However, as will be showed in A.5, branches, although are allowed, reduce GPU execution efficiency. This kind of parallelism is often called in other ways like SIMT [114] (Single Instruction Multiple Threads) by NVIDIA, meaning that the same program is instantiated with multiple threads spread among the GPU cores, allowing different threads to execute different program paths while managing different data. OpenCL and NVIDIA CUDA GPGPU programming languages uses SPMD concepts to ease GPGPU programming thanks to the so-called kernels.

MPMD

Multiple Program Multiple Data is a more general purpose kind of parallelism with respect to SPMD, since different computational units are allowed to perform different instructions streams from different programs at the same time. This is similar to MIMD, however, as said, with MPMD different instruction streams from different programs can be executed in parallel on the available computational units. Again this is useful for both applications like browsers, games and for numerical applications.

Data parallelism

As Wikipedia suggests: "Data parallelism is a form of parallelization of computing across multiple processors in parallel computing environments. Data parallelism focuses on distributing the data across different parallel computing nodes.". Roughly speaking it is possible to talk about data parallelism when the same operation has to be performed on different data. Thus, the idea is to use multiple computational units programmed to perform the same operation feeding them with different data. As an example, the sum of two vectors could be easily mapped to an algorithm that exhibit data type of parallelism, by simply spreading the total work among all the available computational units and assigning different data to different units. This way each computational unit is responsible to perform the sum of different vectors elements. This kind of parallelism is very useful in numerical simulations, and represents basically the core idea of this work. The idea is to perform the same operation, e.g. fluxes computation, of different data of the same kind, e.g. cells solutions.

Task parallelism

Again, from Wikipedia: "Task parallelism (also known as function parallelism and control parallelism) is a form of parallelization of computer code across multiple processors in parallel computing environments. Task parallelism focuses on distributing tasks, concretely performed by processes or threads, across different processors.". Roughly speaking task parallelism is allowed when different tasks can be performed (eventually on different data) at the same time. Thus, total work is split in different tasks that are spread among the available computational units. This kind of parallelism is usually achieved on CPUs using operating system concepts like threads and processes.

A.4 Parallelization strategies overview

Here a brief overview concerning the parallelization strategies, especially for CPUs, that are nowadays usually adopted is provided. The discussion is focused mainly on software and conceptual aspects, but anyway these are strictly related to the underlying hardware architectures. Depending on the underlying hardware and software architectures, usually parallelization involves concepts like shared memory, distributed memory, SIMD extensions, (N)UMA platforms. These will be briefly explained since the solver can use a combinations of these during executions, especially on CPU.

A.4.1 SIMD Extensions

Nowadays, CPUs, even the cheaper ones, have multi-cores architectures. As previously explained in A.1 they exhibit MIMD/MPMD parallelism, meaning that different instruction streams and even different programs are allowed to run on different cores concurrently, working on different data streams. Obviously, since this is the most general purpose kind of architecture, it is also possible run a multi-core CPU in SIMD/SPMD mode, by executing the same instruction stream or program on different cores and feed them with different data streams. As an example, the sum of two vectors on a dual-core CPU could be achieved by coding a program that is executed with one process and two threads. Each thread runs on each physical core and computes one half of the results, potentially allowing to reduce by a factor of 2 the total computational time required. However, this is not the only way a modern CPU can perform SIMD operations, i.e. it could be also done through SIMD extensions. As an example, x86-64 CPU architectures usually feature multiple SIMD extensions like MMX, 3DNow!, SSE (in all of its versions like SSE, SSE2, SSE3, SSE4) and AVX (in all its versions like AVX, AVX2, AVX-512). ARM architectures have something similar called NEON. In any case, the idea is to allow a single CPU core to perform the same operation on multiple data. This is different from the previous described multi-thread approach since this time just one core is capable of SIMD operations. As an example, SSE registers have 128 bit width, meaning that each CPU core can perform the same operation on 4 floats (32 bits each) or 2 doubles (64 bit each). AVX has 256 bit registers width and AVX-512 have 512 bit width. Figure A.5 shows a single-core SIMD approach applied to the aforementioned vector sum example, using 4 floats in 128 bit width registers. It is easy to understand that combining multi-core (through multi-thread) and SIMD extensions it is possible to potentially achieve high speed-ups in numerical applications, just by efficiently exploiting the available hardware. However, it must be noted that while multi-thread programming is nowadays quite simple, e.g. by using OpenMP introduced later, usually SIMD programming with SSE/AVX requires a more advanced programming knowledge. This is due to the fact that often, in order to achieve high computational efficiency, some assmbly-level programming is required. However, modern compilers like GCC have the so called auto-vecotrization capability, usually activated with flags like -ftree-vectorize, meaning that they try to vectorize code (especially for loops) and produce SIMD instructions automatically, in order to optimize the code without requiring the user to explicitly write SIMD assembly code. This is also done by the Intel implementation of OpenCL on CPUs where, thanks to implicit vectorization [8], AVX instructions are used whenever is possible. This means that in this work, when the solver is executed on CPUs, SIMD capabilities are automatically exploited. However it must be noted that usually efficiency peaks are obtained by the programmer writing low-level assembly SIMD code.

A.4.2 Shared memory system and multi-threading

Shared memory parallel computer processors, with respect to distributed memory parallel computer processors, have the capability to access all memory as a global address space. This architecture allows different processors to perform the same or different tasks by anyway sharing the same memory resources. Roughly speaking if a processor

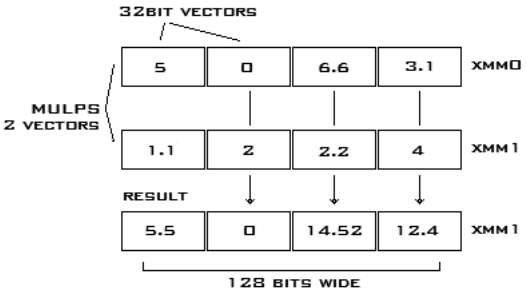


Figure A.5: *SSE example [16]*

changes a variable in memory, the change is visible by all the other processors sharing the same memory. However, the reality is not exactly this simple. There are two important implementations of shared memory concept: UMA (Uniform Memory Access) and NUMA (Non Uniform Memory Access).

UMA [25] shared memory systems are usually called SMP (Symmetric Multi Processor). In this architecture processors have equal access priority and access time to all the shared memory. However, all processors have their own private cache memory, thus memory coherence problems arises. What happens is that when a processor update a variable that is cached in its own private memory, this change is obviously immediately visible to that processors but the variable is not immediately updated in the shared memory. The same variable could be also cached in other processors. A way to obtain a uniform view of the same variable is thus required. This problem is solved using CC-UMA (Cache Coherent UMA) that uses hardware mechanisms to guarantee that all processors will see the correct updated variable when required. Example of this kind of architecture are represented by earlier dual-processors computers. Figure A.6 shows schematically the UMA architecture concept. In this work different multi-core desktop CPUs were used for benchmarks and debugging. These represents examples of SMP systems where each CPU core has the same rights of access the whole system RAM.

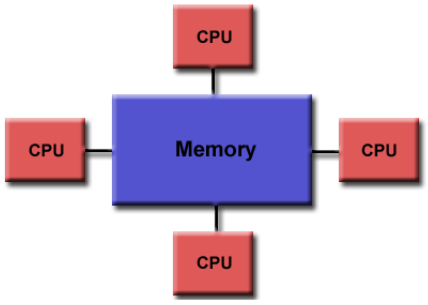


Figure A.6: *UMA architecture [25]*

NUMA [25] shared memory systems are made by physically linking more SMPs. Each SMP has its own privileged part of the global shared memory that can be accessed quickly. However, since NUMA is anyway a shared memory system, all processors shares the same global address space. What happens is that each SMP can access the

privileged memory of another SMP through an interconnect bus but with longer access times with respects to its own privileged part of the global shared memory. The meaning of "Non Uniform" is exactly related to the concepts of having different access times for different memory regions. This is done in order to improve performances since what happens is that most of the time each SMP just need data stored in its own privileged memory. However, if data stored in another memory region is requested, its recovery is always allowed by any SMP. Example of NUMA systems are modern multi-core multi-processor architectures, where on the same motherboard two or more processors are installed alongside with multiple RAM banks. Some RAM banks are flagged as privileged to a particular processor, allowing it to have the access priority. However, all RAM banks can be always accessed by other processors, although with higher access times, through the use of buses like Intel QPI or HyperTransport. Figure A.7 shows schematically the NUMA architecture concept.

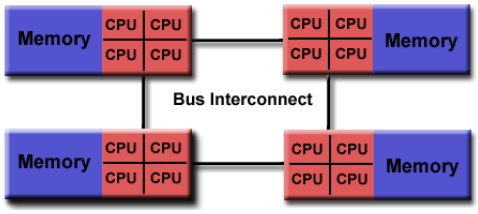


Figure A.7: *NUMA architecture [25]*

Shared memory systems have advantages over distributed memory systems (that will be explained in a moment) thanks to the fast data sharing between different tasks and thanks to the user-friendly perspective of the memory provided by the global shared address space. However, the main drawback of this architecture is given by its lack of scalability when the total number of processors increases, due to the excessive overhead of keeping the shared memory concept working. Another disadvantage is given by the fact that the programmer is responsible to correctly access a shared variable from different tasks.

Multi-thread From the software and programming point of view, shared memory systems can be exploited with multi-threading programming model. A thread is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. A thread is basically a light-weight process: a single heavy-weight process can be split into multiple threads executed concurrently to achieve task or data parallelism. As an example, in multi-core CPUs different threads from the same process (or different processes) can be executed by different cores simultaneously. Different operating systems manage threads in slightly different ways. In windows the distinction between processes and threads is marked. In linux instead, process and threads are basically the same things, "tasks", with the particularity that threads are tasks that shares memory pages. In any case threads of the same process have their own private memory and a shared memory that complies with the shared memory model and that can be used to achieve threads inter-communication. Threads of different processes don't share memory. In order to allow communication between different processes, other kinds of mechanism must be employed, like message passing, usually used in distributed memory systems. Despite communications, another advantage offered by the multi-thread approach over the multi-process approach is given by

its light-weight nature: spawning multiple threads is cheaper that forking multiple processes, requiring less CPU time and less memory. This is due to the fact that by their own nature threads share memory, but during process fork memory has to be duplicated. Different tools are nowadays available to programmers in order to easily create multi-threaded applications. Here, a brief description of OpenMP is provided to the reader, thanks to its simplicity. This is not the only choice to achieve this goal since other ways like POSIX Threads can be used. However, as will be showed, OpenMP is probably one of the most simple and flexible.

OpenMP Roughly speaking, OpenMP is an API that comprises three components: compiler directives, runtime library routines, environment variables. The simplicity behind the use of OpenMP is based on the use of compiler directives. These allows the programmer to initially write its own serial version of the code and then modify it with few lines of OpenMP complier directives in order to parallelize the execution through multi-threading. Despite other approaches like pthreads, with OpenMP the user just have to tell the compiler which part of code parallelize, while it is actually the compiler itself that generates the code required for multi-thread executions. OpenMP, like other standards, is just a set of specifications, while the compilers are required to implement its features. The most important compilers, like GCC, support OpenMP natively. Besides the relatively simplicity provided by OpenMP, the programmer still have to explicitly tell the compiler what to do with shared variables, i.e. shared variables accesses must be opportunely protected in order to avoid problems when multiple threads try to write (or one read and one write) on the same shared memory location. Thus, the user still have to take care of some important shared memory concepts like synchronization. In example A.1 the usual two vectors sum source code is showed in a C serial implementation, alongside the OpenMP multi-thread parallelization. It is possible to see the OpenMP simplicity.

Listing A.1: *Vector sum with OpenMP*

```
#pragma omp parallel
for( int i = 0; i < vec_size; i++)
{
  c[i] = a[i] + b[i];
}
```

What happens during execution is that the program is started as a process with a single thread. When the execution reaches a parallel region, multiple threads are created in order to split the work into multiple chunks, managed by multiple threads, that are spread among the available cores. Considering the example A.1, what happens is that when the single-thread execution reaches the parallelized for loop, the process is subdivided into multiple threads and different chunks of arrays a, b, c are handled by different threads. Work is split through the use of the loop iteration index i: different threads are basically looping over different values of i. Thus, i represents a thread-private variable while the three arrays are stored in shared memory and thus accessible by all threads. In this particular case each thread is accessing the shared memory without any overlapping. Thus, there is no need to protect arrays accesses. However, an implicit barrier is located at the end of the loop in order to guarantee that after the parallel region every thread can access every memory location of c without problems. In any case, when the user wants to protect the access to a shared variable from the possibility of two threads writing on

the same memory location or the possibility of one threads writing and another reading the same memory location, a so called "critical" region can be employed through the use of # pragma omp critical directive. This ensure the correctness of operations on shared variables but could slow down the execution.

As previously mentioned, the multi-thread approach can be coupled with SIMD extensions in order to exploit the whole floating point power of modern CPUs. This could be done by coupling OpenMP with auto-vectorization, or by using the new OpenMP directives (e.g. # pragma omp simd) specifically designed to tell the compiler to generate SIMD code.

OpenMP is a very powerful and easy-to-use tool. At the time of writing, the latest version of the standard introduced the possibility of heterogeneous computing, in a similar approach adopted by OpenACC, through the use of compiler directives. However, for now it is basically used for CPU parallelization through multi-threading. OpenCL rather than OpenMP is preferred in this work because, thanks its low-level nature, allows a more clear view of how the source code is translated in the underlying hardware operations and allows more control on it. Furthermore, OpenCL is specifically designed for heterogeneous computing and allows from its very first versions to just write a single source code and at runtime decide on which computing device (CPUs, GPUs or other accelerators) perform the actual computations. At the time of writing it is possible to download and use for free OpenCL libraries and tools from the most important hardware vendors like Intel, NVIDIA, AMD. OpenACC instead is basically fully supported by commercial compilers.

In this work, OpenMP is adopted in some pre-processing stages of the solver, before the OpenCL execution over GPU. This is done for specific algorithms like extended-cell search that due to their nature cannot be efficiently parallelized on GPU architectures. Another important aspect related to multi-threading in this work is the fact that when an OpenCL-compliant program uses a CPU as a device to run the so called "kernels" (the functions that requires hardware acceleration (see 4.4)), parallelism is achieved through multi-threading, in a shared-memory manner. In this work different multi-core Intel and AMD CPUs were used to perform benchmarks and debugging. During CPU solver executions the total work is split and assigned to different threads running on different cores of the same CPU. The important advantage provided here by the use of OpenCL is that all the multi-thread management is handled by the underlying implementation provided by Intel/AMD OpenCL libraries at runtime. The programmer just have to write a single source code that will be also used for GPU executions. Furthermore, OpenCL allows the coupling of multi-thread and SIMD capabilities [8].

The usual approach in numerical applications to exploit multi-threading is to try to parallelize some specific stages during the simulation. This means that a single process is started with a single thread but multiple threads are used wherever the simulation approaches a point of the code with an algorithm that can be easily mapped to a multi-thread execution. This could mean for example that when convective fluxes have to be computed, different faces are assigned to different threads, possibly scheduled for execution on different cores. However, cell conservative values are kept shared among threads in order to avoid useless memory duplication and to speed up data exchange between different workers (threads). This is also what is usually done with GPGPU and in particular in this work both for CPU and GPU executions.

A.4.3 Distributed memory systems

In distributed memory systems there is no global shared memory and each processor can directly access only its local memory. This is immediately translated in the fact that every update operation of a memory variable inside a processor private memory is not visible to other processors. In this architecture the concept of cache coherence is thus meaningless. However communications between different processors are still allowed thanks to an interconnect network. It must be noted that the network latency and bandwidth are usually orders of magnitude respectively higher and lower with respect to processors accessing their own private memory. This means that network communications have to be minimized as much as possible to achieve high computational efficiency. Usually, in distributed memory systems, clusters composed by multiple nodes are connected through network protocols like Infiniband, optical fiber or even just Ethernet. In any case, from the programmer point of view communications between different nodes have to be explicitly programmed in source code in order to tell each node when and with which other node communicate. Communications between different nodes are usually achieved in distributed memory systems through the concept of message passing. The de-facto standard is represented by MPI. The most important advantage of distributed memory systems is the fact that it is easily possible to increase the total number of processors and memory by just connecting new nodes to the network. Another advantage is represented by the fact that when the parallelized software is opportunely tuned and network communications are reduced, each node basically uses only its own local memory, allowing to reach high efficiency without any overhead due to cache-coherence related operations with high number of computational units. However, the most important disadvantage of the entire architecture is exactly the fact that if the application is poorly optimized and network communications are frequent, these data-exchange leads to bottlenecks for the entire execution. Figure A.8 shows the scheme of a typical distributed memory architecture. As said, MPI is the de-facto

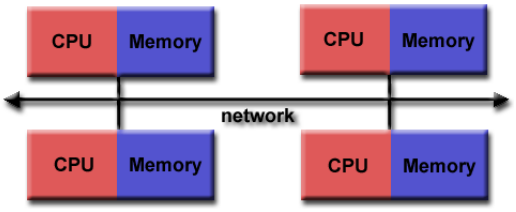


Figure A.8: *Distributed memory architecture [25]*

standard in message passing. Like OpenMP and OpenCL, MPI is just specifications. Different implementations are available, such as OpenMPI and MPICH. In any case, MPI implementations provide the programmer an API composed by a set of functions and types. These are used to initialize and terminate the MPI environment and explicitly perform inter-process communications at particular execution points. The idea behind MPI parallelization is in fact to run different processes on different nodes and allow communications between nodes through the network. In any case, the user has only to worry about when and how exchange data between nodes, while the actual communication operations are handled by the underlying implementation. The implementation handles also possible incompatibilities when exchanging data between nodes with dif-

ferent hardware architectures, guaranteeing portability. Obviously MPI could be also used to parallelize an application with multi-process on a single multi-core CPU though in this case a multi-thread parallelization is more efficient.

The usual approach when using MPI to parallelize work in numerical applications is given by the concept of domain decomposition. This means that the solver is executed in many different instances by different nodes managing different pieces of the computational domain. Inter-process communications only occur when data has to be exchanged between different sub-domains. This way network communications bottlenecks are minimized.

A.4.4 Hybrid and heterogeneous systems

Finally, hybrid and heterogeneous architectures are obtained by mixing concepts from distributed memory systems, shared memory systems, SIMD extensions and GPUs. The idea is to try to exploit the advantages provided by the aforementioned strategies while trying to minimize their drawbacks. This can be easily understood with figure A.9. Multiple shared memory systems are connected together through distributed

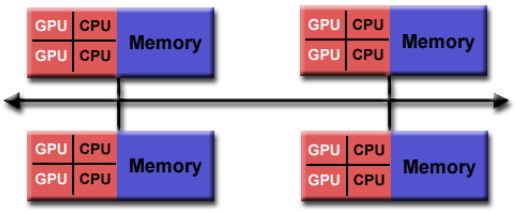


Figure A.9: *Hybrid architecture [25]*

memory system concepts. The total work is initially split using domain decomposition techniques and an instance of the program, i.e. a single process running the program, is executed on each node. This way each node is responsible for each sub-domain of the whole problem. Inside each node parallelism is handled through multi-threading, exploiting shared memory features and splitting the work at finer level, i.e. assigning sub-domain cells to each thread (thus each CPU core). Inside each CPU core, thus inside each CPU thread, parallelism is exploited through SIMD extensions in order to parallelize also simple operations like sum and multiplication. Only when different subdomains require interaction the message passing mechanism is employed to handle inter-nodes communications. Furthermore, communications are limited to only the couple of subdomains that share a boundary. In all this view, inside each node one or more GPUs can be installed to accelerate some peculiar operations for which high SIMD/SPMD performances are required, by offloading the CPUs or by working together with them. This way, by combining different hardware architectures sharing the work, a true hybrid/heteroegenous architecture is achieved, allowing to exploit all the CPU and GPU available features. It must be noted that this represents an ideal view, since not every numerical problem can be mapped to such kind of architecture.

A.5 Performance aspects

Parallel programs are more complex than serial programs since the programmer has to think about multiple data and/or instruction streams and their interaction. Two performance indicators are mainly adopted to asses the advantages provided by parallelization: speed-up and scalability capabilities. Both of them are strictly related to the software-hardware combination and not only to the program itself. Another important aspect is represented by code portability. Here all these aspects are briefly discussed.

Speed-up

Speed-up is probably the most important parameter used to check the advantages given by the parallelization of the code. One of the goal of parallelization is the reduction of the total computational time required to execute a specific job. It is easy to think that if a numerical algorithm can be parallelized, the total computational time can be reduced by allocating more computational resources. However, the situation is more complicated than this. In fact, it is nearly impossible to efficiently parallelize every single part of a program. There will be always some parts of the code that must be executed in a serial way or in any case that leads to bottlenecks with parallel executions. This basically means that parallelization can reduce the total time required to perform some specific parts of the program that can be parallelized while cannot affects all the intrinsically serial parts. This concept is strictly related to the Amdahl's law [41]. It states that, given the parallel fraction of the code P, given the serial fraction of the code $S = 1 - P$, and given the number of processors N, the potential program speed-up of the parallel version with respect to the serial version is:

$$SU_{AL}(P, N) = \frac{1}{\frac{P}{N} + S} = \frac{1}{\frac{P}{N} + (1 - P)} \tag{A.1}$$

This means that, for example, if only 50% of the code can be parallelized ($P = 0.5$, $S = 0.5$) the maximum theoretical speed-up that can be achieved, even with an infinite number of processors ($N \to \infty$) is 2. Figure A.10 shows the behavior of Amdahl's law $SU_{AL}(N, P)$ as a function of the available processors N and parametrized with respect to the parallelizable fraction P. It is possible to see that with different values of P the maximum allowable speed-up changes, and in any case the maximum speed-up reaches a plateau by increasing the number of processors N.

It must be noted that the Amdahl's law is just a theoretical law that gives the maximum speed-up that can be reached considering the two parameters P and N. In the reality it is basically impossible to reach such speed-ups. First of all it is very difficult to find the exact value of P for a particular program, even by profiling each part of the code and thinking about which one can be parallelized. Furthermore the Amdahl's law supposes that using N processors it is possible to exactly reduce by N times the time required to perform the parallelizzable fraction of the code. In real executions there are various overheads due to the necessity to split the total work that can be parallelized into different chunks and sending them to different processors, usually through the use of buses or other kind of interconnection. Furthermore during execution, except in the case of the so called "embarrassingly-parallel" algorithms, different processors has to

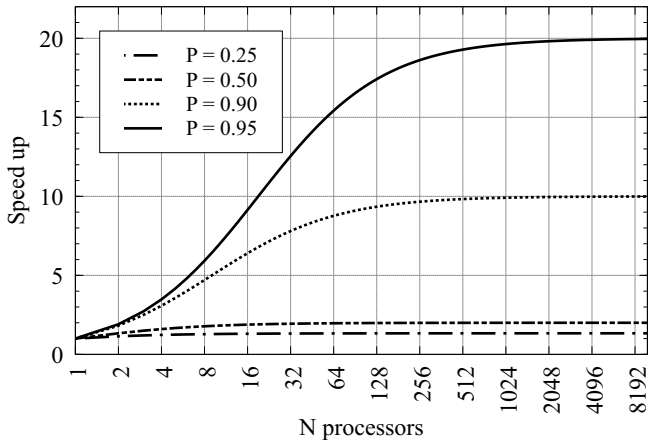


Figure A.10: *Amdahl's law [66].*

exchange data, further reducing the real speed-up. Anyway the Amdahl's law is useful to obtain the order of magnitude of the maximum speed-up that could be achieved thanks to the parallelization of the program.

Scalability

"Scalability is the ability of a problem and its solution algorithm to efficiently handle a growing amount of work" [36]. Basically there are two types of scaling: strong scaling and weak scaling, depending on how the work is split among processors. In strong scaling the total amount of work is fixed as more processors are added: this way it is possible to reduce the total solution time by increasing the total number of available processors. In weak scaling, instead, the idea is to add more processors in order to perform more computations: this way the problem size per processor is fixed while the total amount of work increases as more processors are added. This means that the total solution time is kept fixed as more processors are added but in the same amount of time more cases can be solved or a bigger problem can be solved. The type of scaling strictly depends on both software and hardware employed during the simulation.

Portability

Portability is not directly related to parallelization performances but is anyway a fundamental concept in numerical simulations. In order to increase productivity, a numerical solver should be as more compatible as possible with different underlying software and hardware architectures. This is very important, especially nowadays with concepts like heterogeneous computing where more than one kind of processors are used to perform a particular task, exploiting the different advantages provided by different kind of architectures. In parallel computing field portability is important since sometimes the same solver has to be executed on different hardware architectures (e.g. x86, PowerPC, GPUs,...) and software architectures (e.g. Linux, Windows, OSX,...). This way, in order to reduce maintenance costs and time, it is better to have software that is natively compatible with different architectures rather than have different versions of the source code for each different case. Nowadays the most important parallelization API for CPUs help the programmer to asses this problem. Different API and specifications that

are available to handle shared and distributed computing on CPU, like OpenMP, MPI are also focused on portability. Nowadays portability issues in parallel programming are not as serious as in the past years, however for the same specifications different implementations could behave differently. Thus it is still possible that in order to guarantee portability small changes inside the code are required.

Portability is one of the fundamental goals of this work. The idea is to have an aeroelastic solver natively compatible with both GPUs and CPUs with different architectures and from different vendors. Programming on GPUs is usually more difficult then programming on CPUs since the programmer always has to think about how each line of source code is translated in graphical processor and memory operations in order to achieve the best efficiency. This sometimes means re-think an entire algorithm when porting a code from CPU to GPU. Despite the conceptual problem of adapting an algorithm to the thousands cores of a GPU, from a purely software point of view source code porting is not straightforward due to the fact that usually GPUs are programmed at a lower level than CPUs. In this work the problem of portability is tackled using OpenCL API and OpenCL C language. In fact, OpenCL is specifically aimed for heterogeneous computing allowing to potentially write source code that can be compiled and executed on every device that is compatible with it. This means CPUs, GPUs, FPGAs and other kind of accelerators (like Intel Xeon Phi, at least in its first version). Portability with OpenCL is cleverly handled through concepts like runtime compilation and thanks to the library implementation and support of the OpenCL API an C language by a consortium composed by different vendors like Intel, NVIDIA, AMD, IBM, thus by a consortium composed by the most important CPU and GPU vendors. As an example, another kind of GPGPU API/language, NVIDIA CUDA, is nowadays widely adopted among researchers. However, NVIDIA CUDA with respect to OpenCL it allows the same source code to be compatible just with NVIDIA GPUs, although it usually provides better performances on NVIDIA hardware than OpenCL.

Bibliography

[1] http://turbmodels.larc.nasa.gov/sst.html.

[2] https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/.

[3] http://www.fmslib.com/mkt/gpus.html.

[4] http://ch.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html.

[5] http://www.intel.it/content/www/it/it/support/processors/000005986.html?_ga=1.40993186.1443102262.1473090482.

[6] https://it.wikipedia.org/wiki/Thermal_Design_Power.

[7] https://cvw.cac.cornell.edu/gpu/coalesced?AspxAutoDetectCookieSupport=1.

[8] https://software.intel.com/en-us/articles/auto-vectorization-of-opencl-code-with-the-intel-sdk-for-opencl-applications.

[9] http://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-simple-reductions/.

[10] http://www.amd.com/en-us/products/processors/desktop/a-series-apu#.

[11] https://www.grc.nasa.gov/WWW/wind/valid/raetaf/raetaf.html.

[12] http://nescacademy.nasa.gov/workshops/AePW2/public/.

[13] https://www.rpmturbo.com/testcases/sc10/index.html.

[14] http://www.energy.kth.se/proj/projects/Markus%20Joecker/STCF/default.htm.

[15] https://www.rpmturbo.com/testcases/sc10_3D/index.html.

[16] http://neilkemp.us/src/sse_tutorial/sse_tutorial.html.

[17] Aircrack-ng. https://www.aircrack-ng.org/.

[18] Boinc. boinc.berkeley.edu/.

[19] Folding@home. http://folding.stanford.edu/.

[20] Gcc. https://gcc.gnu.org/.

[21] GDB. https://www.gnu.org/software/gdb/.

[22] Gnu gprof. https://sourceware.org/binutils/docs/gprof/.

[23] John the ripper. http://www.openwall.com/john/.

[24] Khronos group. https://www.khronos.org/.

[25] Llnl. http://computing.llnl.gov/.

[26] Nvidia website. http://www.nvidia.it/page/home.html.

[27] Oclgrind. https://github.com/jrprice/Oclgrind.

[28] Openacc. http://www.openacc.org/.

[29] Paraview. http://www.paraview.org/.

[30] Raspberry Pi. https://www.raspberrypi.org/.

[31] Stack overflow. http://stackoverflow.com/.

[32] Top 500. https://www.top500.org/.

[33] Turbulence modeling resource (nasa). https://turbmodels.larc.nasa.gov/.

[34] Valgrind. http://valgrind.org/.

[35] Viennacl. http://viennacl.sourceforge.net/.

[36] Wikipedia. https://en.wikipedia.org/.

[37] Wikipedia ati/amd gpus list. https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units.

[38] Wikipedia nvidia gpus list. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units.

[39] Michela Alfano. *A probability-based methodology for buckling investigation of sandwich composite cylindrical shells*. PhD thesis, Italy, 2016.

[40] AMD, Inc. *AMD Accelerated Parallel Processing OpenCL*, 2012.

[41] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[42] Andrea Arnone. Viscous analysis of three-dimensional rotor flow using a multigrid method. *Journal of turbomachinery*, 116(3):435–445, 1994.

[43] Barrett Stone Baldwin and Harvard Lomax. *Thin layer approximation and algebraic model for separated turbulent flows*, volume 257. American Institute of Aeronautics and Astronautics, 1978.

[44] Robert E Bartels. Flow and turbulence modeling and computation of shock buffet onset for conventional and supercritical airfoils. 1998. NASA TP-1998-206908.

[45] Robin M Betz, Nathan A DeBardeleben, and Ross C Walker. An investigation of the effects of hard and soft errors on graphics processing unit-accelerated molecular dynamics simulations. *Concurrency and Computation: Practice and Experience*, 26(13):2134–2140, 2014.

[46] Robert T Biedron and James L Thomas. Recent enhancements to the FUN3D flow solver for moving-mesh applications. *AIAA Paper*, 1360:2009, 2009.

[47] RL Bisplinghoff, H Ashley, and RL Halfman. *Aeroelasticity, 1996*. Dover Publication Inc., Mineola, New York.

[48] J. Blazek. *Computational Fluid Dynamics*. Elsevier, 2001.

[49] Tobias Brandvik and Graham Pullan. An accelerated 3d navier–stokes solver for flows in turbomachines. *Journal of Turbomachinery*, 133(2), 2011.

[50] O. Brodersen and A. Sturmer. Drag prediction of engine–airframe interference effects using unstructured navier–stokes calculations. In *9th AIAA Applied Aerodynamics Conference*, Wahington, DC, 2001. AIAA.

[51] Francesco Capizzano. Turbulent wall model for immersed boundary methods. *AIAA journal*, 49(11):2367–2381, 2011.

[52] Luca Cavagna, Giuseppe Quaranta, and Paolo Mantegazza. Application of navier–stokes simulations for aeroelastic stability assessment in transonic regime. *Computers & Structures*, 85(11):818–832, 2007.

[53] Kuei-Yuan Chien. Predictions of channel and boundary-layer flows with a low-reynolds-number turbulence model. *AIAA journal*, 20(1):33–38, 1982.

[54] R. V. Chima, P. W. Giel, and R. J. Boyle. An algebraic turbulence model for three-dimensional viscous flows. *31-st Aerospace Sciences Meeting and Exhibit*, 1993.

[55] Pawel Chwalowski, Jennifer P Florance, Jennifer Heeg, Carol D Wieseman, and Boyd P Perry. Preliminary computational analysis of the (hirenasd) configuration in preparation for the aeroelastic prediction workshop. *International Forum on Aeroelasticity and Structural Dynamics*, 2011.

[56] P. Cinnella and P. M. Congedo. Aerodynamics Performance of Transonic Bethe-Zel'dovich-Thompson Flows past an Airfoil. *AIAA Journal*, 43(2):370–378, February 2005.

[57] J. D. Cole and E. M. Murman. Calculation of plane steady transonic flows. *AIAA Journal*, 9(1):114–121, 1971.

[58] Y Colin, H Deniau, and J-F Boussuge. A robust low speed preconditioning formulation for viscous flow computations. *Computers & Fluids*, 47(1):1–15, 2011.

[59] P. Colonna, J. Harinck, S. Rebay, and A. Guardone. Real-gas effects in organic rankine cycle turbine nozzles. *Journal of Propulsion and Power*, 24(2):282–294, 2008.

[60] PH Cook, MCP Firmin, and MA McDonald. *Aerofoil RAE 2822: pressure distributions, and boundary layer and wake measurements*. RAE, 1977.

[61] N. Donini. Aeroelasticity of Tubomachines Linearized Flutter Analysis, 2012. MSc Thesis, Politecnico di Milano.

[62] E. H. Dowell, R. Clark, D. Cox, H. C. Jr Curtiss, J. W. Edwards, K. C. Hall, D. A. Peters, R. H. Scanlan, E. Siniu, F. Sisto, and T. W. Strgana. *A Modern Course in Aeroelasticity*. Kluwer Publishers, 2004.

[63] Earl H Dowell, Robert Clark, David Cox, et al. *A modern course in aeroelasticity*, volume 3. Springer, 2004.

[64] V Elchuri, A Michael Gallo, and SC Skalski. Nastran documentation for flutter analysis of advanced turbopropellers. 1982. www.ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19840007085.pdf.

[65] V Elchuri and GCC Smith. Nastran flutter analysis of advanced turbopropellers. 1982. www.ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19840006080.pdf.

[66] M. Favale and A. Gadda. A gpu parallelized two fields full potential formulation for real gases, 2013. MSc Thesis, Politecnico di Milano.

[67] M. J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[68] Federico Fonte. Active gust alleviation for a regional aircraft through static output feedback, 2013. MSc Thesis, Politecnico di Milano.

[69] Torsten Fransson and JM Verdon. Standard configurations on unsteady flow through vibrating turbo machine cascades. 1992. Laboratoire Thermique Appliquée et des Turbomachines, EPFL.

[70] Andrey Garbaruk, Mikhail Shur, Mikhail Strelets, and Philippe R Spalart. Numerical study of wind-tunnel walls effects on transonic airfoil flow. *AIAA journal*, 41(6):1046–1054, 2003.

[71] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous computing with OpenCL*. Morgan Kaufmann, 2011.

[72] Thomas Gerhold. Overview of the hybrid rans code tau. In *MEGAFLOW-Numerical Flow Simulation for Aircraft Design*, pages 81–92. Springer, 2005.

[73] L. G. Goldman and K. L. McLallin. Cold-air annular cascade investigation of aerodynamic performance of core-engine-cooled turbine vanes: I solid vane performance and facility description. 1977. NASA TM.

[74] A. D. Grosvenor. Rans prediction of transonic compressive rotor performance near stall. *ASME Paper*, (GT2007-27691), 2007.

[75] A. Guardone and Quartapelle L. *High-resolution unstructured finite-volume methods for conservation laws*.

[76] A. Guardone, A. Spinelli, and V. Dossena. Influence of molecular complexity on nozzle design for an organic vapor wind tunnel. *Journal of engineering for gas turbines and power*, 135(4), 2013.

[77] T. R. Hagen, J. M. Hjelmervik, K.-A. Lie, J. R. Natvig, and M. O. Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, 13(8):716–726, 2005.

[78] Roy D Hager and Deborah Vrabel. Advanced turboprop project. 1988. Tech. rep. National Aeronautics and Space Administration, Cleveland, OH (USA). Lewis Research Center.

[79] J. Harinck, A. Guardone, and P. Colonna. The influence of molecular complexity on expanding flows of ideal and dense gases. *Physics of Fluids*, 21, 2009.

[80] Jennifer Heeg, Joseph Ballmann, Kumar Bhatia, Eric Blades, Alexander Boucke, Pawel Chwalowski, Guido Dietz, Earl Dowell, Jennifer Florance, Thorsten Hansen, et al. Plans for an aeroelastic prediction workshop. 2011. IFASD-2011-110.

[81] Jennifer Heeg, Pawel Chwalowski, David Schuster, Daniella Raveh, Mats Dalenbring, and Adam Jirasek. Plans and example results for the 2nd aiaa aeroelastic prediction workshop. 2015. 56th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference Kissimmee, Florida.

[82] Jennifer Heeg, Carol D Wieseman, and Pawel Chwalowski. Data comparisons and summary of the second aeroelastic prediction workshop. 34th AIAA Applied Aerodynamics Conference, AIAA Aviation 2016; 13-17 Jun. 2016; Washington, DC; United States.

[83] A. Jameson. Transonic potential flow calculations using conservation form. In *Proc. of AIAA 2nd Computational Fluid Dynamic Conference, Hartford, Conn*, pages 148–155, 1975.

[84] A. Jameson and D. A. Caughley. Transonic potential flow calculations using conservative form. In *In Proceedings AIAA Third Computational Fluid Dynamics Conference. Albuquerque*, 1977.

[85] Antony Jameson. Analysis and design of numerical schemes for gas dynamics, 1: artificial diffusion, upwind biasing, limiters and their effect on accuracy and multigrid convergence. *International Journal of Computational Fluid Dynamics*, 4(3-4):171–218, 1995.

[86] Antony Jameson. Analysis and design of numerical schemes for gas dynamics, 2: Artificial diffusion and discrete shock structure. *International Journal of Computational Fluid Dynamics*, 5(1-2):1–38, 1995.

[87] Hrvoje Jasak and Henrik Rusche. Dynamic mesh handling in openfoam. In *Proceeding of the 47th Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, Orlando, FL*, 2009.

[88] Georgi Kalitzin, Gorazd Medic, Gianluca Iaccarino, and Paul Durbin. Near-wall behavior of RANS turbulence models and implications for wall functions. *Journal of Computational Physics*, 204(1):265–291, 2005.

[89] M. Karczewski and J. Blaszczak. Performance of three turbulence models in 3d flow investigation for a 1.5-stage turbine. *Task Quarterly*, 12(3):185–195.

[90] The Khronos Group Inc. *The OpenCL Specification*, 2012.

[91] Christian Klostermeier. *Investigation into the capability of large eddy simulation for turbomachinery design*. PhD thesis, University of Cambridge, 2008.

[92] Anwar Koshakji, Alfio Quarteroni, and Gianluigi Rozza. Free form deformation techniques applied to 3d shape optimization problems. *Communications in Applied and Industrial Mathematics*, 4(EPFL-ARTICLE-197094), 2013.

[93] Frank Lane. System mode shapes in the flutter of compressor blade rows. *Journal of the Aeronautical Sciences*, 23(1):54–66, 1956.

[94] Randall J LeVeque. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[95] Randall J LeVeque and Randall J Leveque. *Numerical methods for conservation laws*, volume 132. Springer, 1992.

[96] Xue-song Li and Chun-wei Gu. An all-speed roe-type scheme and its asymptotic analysis of low mach number behaviour. *Journal of Computational Physics*, 227(10):5144–5159, 2008.

[97] Meng-Sing Liou. A sequel to ausm: Ausm+. *Journal of computational Physics*, 129(2):364–382, 1996.

[98] Earl Logan Jr. *Handbook of turbomachinery*. CRC Press, 2003.

[99] A Madrane, A Raichle, and A Stuermer. Parallel implementation of a dynamic unstructured chimera method in the dlr finite volume tau-code. 2004. Proceedings of Twelfth annual conference of the CFD Society of Canada.

[100] L Mangani, E Casartelli, Giulio Romanelli, Magnus Fischer, A Gadda, and P Mantegazza. A gpu-accelerated compressible rans solver for fluid-structure interaction simulations in turbomachinery. In *ISROMAC 2016*, 2016.

[101] L Mangani, E Casartelli, Giulio Romanelli, Magnus Fischer, A Gadda, and P Mantegazza. A gpu-accelerated compressible rans solver for fluid-structure interaction simulations in turbomachinery. In *ASME Turbo Expo 2016: Turbomachinery Technical Conference and Exposition*, pages V07BT34A022–V07BT34A022. American Society of Mechanical Engineers, 2016.

[102] Luca Mangani, Marwan Darwish, and Fadl Moukalled. Development of a pressure-based coupled cfd solver for turbulent and compressible flows in turbomachinery applications. In *ASME Turbo Expo 2014: Turbine Technical Conference and Exposition*, pages V02BT39A019–V02BT39A019. American Society of Mechanical Engineers, 2014.

[103] Luca Mangani, Giulio Romanelli, Andrea Gadda, and Ernesto Casartelli. Comparison of acceleration techniques on cfd open-source software for aerospace applications. In *22nd AIAA Computational Fluid Dynamics Conference*, page 3059, 2015.

[104] P. Mantegazza. *"BIGINO" DI DINAMICA E CONTROLLO DI STRUTTURE AEROSPAZIALI*. 2015.

[105] Markus May, Yann Mauffrey, and Frédéric Sicot. Numerical flutter analysis of turbomachinery bladings based on time-linearized, time-spectral and time-accurate simulations. *Proceedings" IFASD 2011"*, 2011.

[106] N Duane Melson, Harold L Atkins, and Mark D Sanetrik. Time-accurate navier-stokes calculations with multigrid acceleration. 1993. The Sixth Copper Mountain Conference on Multigrid Methods.

[107] Florian R Menter. Two-equation eddy-viscosity turbulence models for engineering applications. *AIAA journal*, 32(8):1598–1605, 1994.

[108] FR Menter, M Kuntz, and R Langtry. Ten years of industrial experience with the sst turbulence model. *Turbulence, heat and mass transfer*, 4(1):625–632, 2003.

[109] Tomokazu Miyakozawa. *Flutter and forced response of turbomachinery with frequency mistuning and aerodynamic asymmetry*. ProQuest, 2008.

[110] Matthew D Montgomery and Joseph M Verdon. A three-dimensional linearized unsteady euler analysis for turbomachinery blade rows. 1997. NASA Contractor Report 4770.

[111] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.

[112] Jens Nipkau. *Analysis of mistuned blisk vibrations using a surrogate lumped mass model with aerodynamic influences*. Shaker, 2011.

[113] NVIDIA Corporation. *NVIDIA OpenCL JumpStart Guide*, 2011.

[114] NVIDIA Corporation. *OpenCL Programming Guide for the CUDA Architecture*, 2012.

[115] A. Parrinello. *Independent Fields Full Potential Formulation for Aeroelastic Analyses*. PhD thesis, Politecnico di Milano, March 2012.

[116] A. Parrinello and P. Mantegazza. Independent Two-Fields Solution for Full-Potential Unsteady Transonic Flows. *AIAA Journal, Vol. 48, No. 7*, 2010.

[117] M Pelanti, L Quartapelle, and L Vigevano. A review of entropy fixes as applied to roe's linearization. *Teaching material of the Aerospace and Aeronautics Department of Politecnico di Milano*, 2001.

[118] Paul J Petrie-Repar, Andrew McGhee, and Peter A Jacobs. Three-dimensional viscous flutter analysis of standard configuration 10. In *ASME Turbo Expo 2007: Power for Land, Sea, and Air*, pages 665–674. American Society of Mechanical Engineers, 2007.

[119] T. Poehler, J. Gier, and P. Jeschke. Numerical and experimental analysis of the effects of non-axisymmetric contoured stator endwalls in an axial turbine. *ASME Paper*, (GT2010-23350), 2010.

[120] S. B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[121] M Popovac and K Hanjalic. Compound wall treatment for RANS computation of complex turbulent flows/heat transfer. *Flow, turbulence, combustion*, 78(2), 2007.

[122] D. Prederi. Flutter analysis of open rotors, 2015. MSc Thesis, Politecnico di Milano.

[123] Giuseppe Quaranta, Pierangelo Masarati, and Paolo Mantegazza. A conservative mesh-free approach for fluid-structure interface problems. In *International Conference for Coupled Problems in Science and Engineering, Greece*, 2005.

[124] L. P. Quartapelle and F. Auteri. *Fluidodinamica Comprimibile*. Casa Editrice Ambrosiana, 1st edition, 2013.

[125] L. P. Quartapelle and F. Auteri. *Fluidodinamica Incomprimibile*. Casa Editrice Ambrosiana, 1st edition, 2013.

[126] U. Reinmöller, B. Stephan, S. Schmidt, and R. Niehuis. Clocking effects in a 1.5 stage axial turbine: Steady and unsteady experimental investigations supported by numerical simulations. *Journal of Turbomachinery*, 124(1):52–60, 2002.

[127] G. Romanelli. *Computational aeroservoelasticity of free-flying deformable aircraft*. PhD thesis, Politecnico di Milano, 2012.

[128] G Romanelli, L Mangani, and E Casartelli. Implementation of a cfd-based aeroelastic analysis toolbox for turbomachinery applications. In *ASME Turbo Expo 2014: Turbine Technical Conference and Exposition*, pages V02BT45A010–V02BT45A010. American Society of Mechanical Engineers, 2014.

[129] G Romanelli, L Mangani, E Casartelli, A Gadda, and M Favale. Implementation of explicit density-based unstructured cfd solver for turbomachinery applications on graphical processing units. *ASME Paper*, 2015.

[130] G. Romanelli, E. Serioli, and P. Mantegazza. A New Accurate Compressible Solver for Aerodynamic Applications. *3-rd OpenFOAM Workshop*, 2008.

[131] G. Romanelli, E. Serioli, and P. Mantegazza. A Free Approach to Modern Computational Aeroelasticity. *48-th AIAA Aerospace Sciences Meeting*, 2009.

[132] Giulio Romanelli, Michele Castellani, Paolo Mantegazza, and Sergio Ricci. Coupled csd/cfd non-linear aeroelastic trim of free-flying flexible aircraft. In *_ 53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference _*, 2012.

[133] Filippo Salmoiraghi, Francesco Ballarin, Giovanni Corsi, Andrea Mola, Marco Tezzele, and Gianluigi Rozza. Advances in geometrical parametrization and reduced order models and methods for computational fluid dynamics problems in applied sciences and engineering: overview and perspectives. ECCOMAS, 2016.

[134] M. Scarpino. *OpenCL in action*. Manning Publications, 2011.

[135] V. Schmitt and F. Charpin. Experimental data base for computer program assessment: Pressure distribution on the onera m6 wing at transonic mach numbers. *AGARD Advisory Report 138*, 1979.

[136] E. Serioli and G. Romanelli. Un approccio libero alla moderna aeroelasticità computazionale, 2008. MSc Thesis, Politecnico di Milano.

[137] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, pages 517–524. ACM, 1968.

[138] Walter A Silva, Pawel Chwalowski, and Boyd Perry III. Evaluation of linear, inviscid, viscous, and reduced-order modelling aeroelastic solutions of the agard 445.6 wing using root locus analysis. *International Journal of Computational Fluid Dynamics*, 28(3-4):122–139, 2014.

[139] Joseph Smagorinsky. General circulation experiments with the primitive equations: I. the basic experiment*. *Monthly weather review*, 91(3):99–164, 1963.

[140] P. R Spalart and S. R. Allmaras. A one-equation turbulence model for aerodynamic flows. *AIAA*, (92-0439), 1992.

[141] Philippe R Spalart, Shur Deck, ML Shur, KD Squires, M Kh Strelets, and A Travin. A new version of detached-eddy simulation, resistant to ambiguous grid densities. *Theoretical and computational fluid dynamics*, 20(3):181–195, 2006.

[142] PR Spalart, WH Jou, M Strelets, SR Allmaras, et al. Comments on the feasibility of les for wings, and on a hybrid rans/les approach. *Advances in DNS/LES*, 1:4–8, 1997.

[143] A. J Strazisar, J.R. Wood, and K. L. Suder. Laser anemometer measurements in a transonic axial-flow fan rotor. (2879), 1989. NASA-TP-2879, NASA, Lewis Research Center, Cleveland Ohio.

[144] Arne Stuermer. Unsteady cfd simulations of contra-rotating propeller propulsion systems. *Paper No. AIAA-2008-5218*, 2008.

[145] Shigefumi Tatsumi, Luigi Martinelli, and Antony Jameson. Flux-limited schemes for the compressible navier-stokes equations. *AIAA journal*, 33(2):252–261, 1995.

[146] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki. *The OpenCL Programming Book*. Fixstars Corporation, 2010.

[147] Matteo Tugnoli. Modelli di simulazione ai grandi vortici in openfoam: una analisi comparativa, 2013. MSc Thesis, Politecnico di Milano.

[148] E. R. van Driest. On turbulent flow near a wall. *Journal of the Aeronautical Sciences*, 23(11):1007–1011, 1956.

[149] Srinivas Vasista, Srinivas Vasista, Alessandro De Gaspari, Alessandro De Gaspari, Sergio Ricci, Sergio Ricci, Johannes Riemenschneider, Johannes Riemenschneider, Hans Peter Monner, Hans Peter Monner, et al. Compliant structures-based wing and wingtip morphing devices. *Aircraft Engineering and Aerospace Technology: An International Journal*, 88(2):311–330, 2016.

[150] Cécile Viozat. *Implicit upwind schemes for low Mach number compressible flows*. PhD thesis, Inria, 1997.

[151] Ross C Walker and Robin M Betz. An investigation of the effects of error correcting code on gpu-accelerated molecular dynamics simulations. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, page 8. ACM, 2013.

[152] David C Wilcox. Formulation of the kw turbulence model revisited. *AIAA journal*, 46(11):2823–2838, 2008.

[153] J. Witteveen. Explicit Mesh Deformation Using Inverse Distance Weighting Interpolation. *47-th AIAA Aerospace Sciences Meeting*, 2009.

[154] Jeroen AS Witteveen and Hester Bijl. Explicit mesh deformation using inverse distance weighting interpolation. In *19th AIAA Computational Fluid Dynamics Conference, San Antonio, Texas, AIAA-2009-3996*, 2009.

[155] J. Yao, R. L. Davis, J. J. Alonso, and Jameson A. Unsteady flow investigations in an axial turbine using the massively parallel solver TFLO. *39-st Aerospace Sciences Meeting and Exhibit*, 2001.

[156] E Carson Yates Jr. Agard standard aeroelastic configurations for dynamic response. candidate configuration i.-wing 445.6. 1987. NASA technical memorandum 100492.

Publications

1. L Mangani, E Casartelli, Giulio Romanelli, Magnus Fischer, A Gadda, and P Mantegazza. A gpu-accelerated compressible rans solver for fluid-structure interaction simulations in turbomachinery. In *ASME Turbo Expo 2016: Turbomachinery Technical Conference and Exposition*, pages V07BT34A022–V07BT34A022. American Society of Mechanical Engineers, 2016
2. L Mangani, E Casartelli, Giulio Romanelli, Magnus Fischer, A Gadda, and P Mantegazza. A gpu-accelerated compressible rans solver for fluid-structure interaction simulations in turbomachinery. In *ISROMAC 2016*, 2016
3. Luca Mangani, Giulio Romanelli, Andrea Gadda, and Ernesto Casartelli. Comparison of acceleration techniques on cfd open-source software for aerospace applications. In *22nd AIAA Computational Fluid Dynamics Conference*, page 3059, 2015
4. G Romanelli, L Mangani, E Casartelli, A Gadda, and M Favale. Implementation of explicit density-based unstructured cfd solver for turbomachinery applications on graphical processing units. *ASME Paper*, 2015