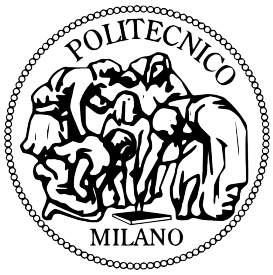
POLITECNICO DI MILANO

Scuola di Ingegneria dei Sistemi

Corso di Laurea Magistrale in Ingegneria Matematica



Tesi di laurea magistrale

GPU implementation of a shell element structural solver aimed at fluid-structure interaction problems

Relatore:
Prof. Alfio Quarteroni

Correlatori:
Dott. Massimiliano Cremonesi
Dott. Nicola Parolini

Tesi di laurea di:
Andrea Bartezzaghi
matricola 770702

Anno accademico 2012-2013

Ringraziamenti

Innanzitutto, desidero ringraziare calorosamente il Prof. Alfio Quarteroni per avermi concesso l'opportunità di svolgere la tesi sotto la sua supervisione, per il suo supporto e la sua disponibilità nonostante i numerosi impegni.
Ringrazio molto il Dott. Nicola Parolini e il Dott. Massimiliano Cremonesi per il costante aiuto che mi hanno fornito durante tutto il periodo di tesi, per l'immensa pazienza e la disponibilità, a qualsiasi ora del giorno e della notte.
Ringrazio anche il Prof. Umberto Perego e il team di Ingegneria Strutturale per il supporto fornitomi durante lo sviluppo del codice strutturale, e il Dott. Matteo Lombardi per il tempo che mi ha dedicato a Losanna e per avermi aiutato a fare chiarezza sulla parte di interazione fluido-struttura.
Un ringraziamento speciale lo dedico alla mia famiglia, in particolare ai miei genitori, che da sempre mi motivano e mi danno la forza per andare avanti, e le mie sorelle, che mi sopportano ogni giorno. Ad Alessandra, per avermi sempre incoraggiato e sostenuto anche nei momenti peggiori. A Ruggero, per le lan, le sessioni di lavoro e la potenza computazionale che mi ha messo a disposizione. A Boes, Gio e Silvia per le distruttive arrampicate. A Ele, Kons, Manu e Miri per gli svaghi, le cene e le bevute. A Jack, Kens, LG e Nick per le serate poker e Pes. A Giorgio, Ivan e Pagan per la sopravvivenza in Svizzera e le "studiate" al Rolex Center. A tutti i miei amici, per avermi aiutato a sopravvivere alla vita di studente.
Infine al Cassone, che, nonostante la veneranda età e i frequenti acciacchi, continua ad essere un fedele compagno di avventure.

Abstract

The study of thin structures is very common nowadays and useful in different fields. An important example is the analysis of sail dynamics. In this context, accurate simulations of the interaction between the sail and the wind are also required. However, this kind of fluid-structure interaction problems are very computationally expensive.

First objective of this thesis is the implementation of an highly efficient shell finite element structural solver, designed to run on GPU (Graphics Processing Unit) hardware. In order to fully exploit the GPU computational power, an explicit central difference time-advancing scheme is adopted. Domain is discretized using MITC4 shell elements in large displacements formulation, due to their adequate numerical properties and ability of avoiding shear-locking problems and simulating sail wrinkles. Techniques adopted during the development, such as algorithms, memory management and code optimizations, are described in details. Numerical tests and benchmarks are carried out and performances are compared with the commercial software Abaqus.

Second objective of this thesis is the development of a partitioned strongly coupled fluid-structure interaction solver, implemented in OpenFOAM, an open-source CFD framework. The fluid dynamics problem is solved using the PISO scheme, while the solver implemented in the first part handles the structural problem. The mesh-motion process and interpolation algorithms are analyzed and implemented in GPU in order to gain performance and reduce memory requirements. Finally, results of numerical and performance tests on the developed FSI solver are reported.

Keywords: fluid-structure interaction, finite element method, shell dynamics, mesh-motion, GPU parallelization, CUDA

Sommario

L'analisi di strutture sottili è attualmente un importante settore di ricerca, anche in relazione allo studio di fenomeni di interazione fluido-struttura. Un esempio è l'analisi della dinamica delle vele interagenti con il vento. Questo tipo di problemi risultano però essere computazionalmente molto onerosi.
In questa tesi, inizialmente si focalizza l'attenzione sul solo problema strutturale. Si propone lo sviluppo di un efficiente solutore ad elementi finiti di tipo guscio, progettato appositamente per l'esecuzione in GPU (Graphics Processing Unit). Il software viene successivamente integrato in un'applicazione più ampia mirata alla risoluzione di problemi di interazione fluido-struttura.
Per quanto riguarda il solutore strutturale, per poter usufruire appieno della potenza computazionale offerta dalla GPU, l'avanzamento in tempo viene effettuato tramite uno schema esplicito alle differenze finite centrate. Il dominio è discretizzato tramite elementi guscio di tipo MITC4, con formulazione in grandi spostamenti. Questo tipo di elementi è l'ideale per lo studio della dinamica delle vele, dato che sono immuni a problemi di shear-locking e permettono di simulare accuratamente le increspature del tessuto senza bisogno di modelli aggiuntivi. In questo documento, tutte le tecniche adottate durante lo sviluppo del codice sono discusse dettagliatamente, assieme agli algoritmi, alla gestione della memoria e alle ottimizzazioni effettuate. Utilizzando il software Abaqus come riferimento, sono riportati i risultati di test sia numerici che di prestazioni, sottolineando l'ottimo guadagno ottenuto grazie all'implementazione in GPU in termini di tempi di calcolo.
Secondo obiettivo di questa tesi è lo sviluppo di un'applicazione destinata alla risoluzione di problemi di interazione fluido-struttura, all'interno dell'ambiente open-source OpenFOAM. Il problema strutturale viene risolto tramite il solutore sviluppato nella prima parte, mentre quello fluido tramite lo schema PISO. Il due problemi sono legati da un accoppiamento forte e partizionato, con sotto-rilassamento di Aitken per velocizzare la convergenza. Viene anche analizzata l'implementazione in GPU degli algoritmi di interpolazione e mesh-motion, per incrementare ulteriormente le prestazioni e diminuire il consumo di memoria. Infine, sono riportati i risultati numerici su due casi di interazione fluido-struttura, con analisi dei benefici ottenuti grazie all'utilizzo della GPU.

Parole chiave: interazione fluido-struttura, metodo degli elementi finiti, dinamica dei gusci, movimento della griglia, parallelizzazione in GPU, CUDA

Contents

List of Figures

List of Tables

Introduction

Scientific computing is a continuously growing field. Thanks to the new technologies, there is always more computational power available. This permits the adoption of more and more complex and sophisticated mathematical models to simulate reality, which aid in most engineering fields.

The study of thin solids, with size along one dimension much smaller than along the others, is of fundamental importance in different applications. Accurate analysis of laminate sheet dynamics is necessary to optimize and prevent problems during forming or molding processes. Studies regarding package production for liquid food have been conducted in [5, 9, 14] using shell finite element formulations. Sail dynamics is another subject which requires the modeling of thin elastic structures. In particular, in order to fully study the dynamics of a sailing boat, it is necessary to analyze the interaction between the sails and the air field in which they are immersed. This kind of fluid-structure interaction problems has started to be analyzed relatively recently, since that the required computational time is very high. Analysis on sail-wind interaction has been conducted in [20, 21, 28, 33]. In this application shell elements are of fundamental importance in order to correctly capture sail wrinkling phenomena.

In recent years, the general purpose GPU (Graphics Processing Unit) programming scene is growing fast. While being initially designed to accelerate real-time 3D graphics operations, over the years programmers realized that GPUs' immense power could be used also for different applications. Its inherent vectorized architecture and massive amount of floating-point arithmetic units make it a very powerful tool for the scientific programmer at a relatively low cost. While in the past GPU programming was hard and required lots of technical notions about the underlying hardware architecture, nowadays thanks to the introduction of frameworks such as CUDA™, by NVIDIA® Corporation[1], or OpenCL™, by the Khronos™ Group[2], it has become more accessible and widespread.

The main objective behind this work is the design of a GPU based implementation of a shell element structural solver. In order to fully exploit the GPU computational power, an explicit time-stepping scheme based on

[1] http://www.nvidia.com/object/cuda_home_new.html
[2] http://www.khronos.org/opencl

central finite difference is employed [30, 37, 18]. Spatial discretization is performed adopting the MITC4 shell element [2, 1, 4], a four-node element with five degrees of freedom per node, three translational and two rotational. This element does not suffer from shear-locking problems thanks to the different interpolation of in-plane deformation convariant components from the out-of-plane deformations (the term *mixed interpolation* derives from this approach). Thanks to the explicit time-integration method and mass lumping, the resulting algorithm is well suited for a GPU implementation. This structural solver has been designed and implemented from scratch during this work. A complete redesign from the start was needed in order to develop an algorithm specifically aimed at running on GPU. Two structural test cases have been simulated and results checked against the literature and the commercial software Abaqus FEA®[3]. Performance benchmarks have also been conducted against Abaqus, in order to have a measure of the speedup gained thanks to the GPU implementation.

The second purpose of this work is the inclusion of the presented GPU structural solver in an integrated framework for the simulation of fluid-structure interaction problems. All the recent FSI models have the big limit of being very computationally and memory demanding. A good solution is thus to exploit the GPU hardware in order to speed up computations. In this work a segregated strongly coupled solver is employed [29, 12, 20]. It has been implemented inside the open-source OpenFOAM® framework[4]. The fluid problem is formulated resorting to the Arbitrary Lagrangian–Eulerian (ALE) approach [20], while the structural problem is expressed in a total Lagrangian formulation. Given the fluid domain Ω_f and the structural domain Ω_s, we name $\mathbf{F}$ the fluid problem, solved in Ω_f, $\mathbf{S}$ the structural problem, solved for the displacements (although using shell elements rotations in each node are computed also) in Ω_s, and $\mathbf{M}$ the mesh-motion problem, which calculates the motion of mesh internal points given the boundary movements, solved in the initial fluid domain Ω_f^0.

[3] `http://www.simulia.com`
[4] `http://www.openfoam.com`

This leads to the following coupled problem:

$$\begin{cases} \mathbf{F}(\mathbf{v}, p, \dot{\eta}) = 0 & \text{in } \Omega_f(t) \\ \mathbf{S}(\mathbf{u}) = 0 & \text{in } \Omega_s(t) \\ \mathbf{M}(\eta) = 0 & \text{in } \Omega_f^0 \\ \mathbf{v} = \dot{\mathbf{u}} & \text{on } \Gamma(t) \\ \sigma_f(\mathbf{v}, p)\,\mathbf{n}_f = \sigma_s(\mathbf{u})\,\mathbf{n}_s & \text{on } \Gamma(t) \\ \eta = \mathbf{u} & \text{on } \Gamma^0 \end{cases}, \tag{1}$$

where $\mathbf{v}$ and p represent the fluid velocity and pressure respectively, $\mathbf{u}$ the displacements computed solving the structural problem and η the positions of fluid mesh points. The fluid problem $\mathbf{F}$ requires also the velocity of mesh-motion $\dot{\eta}$ in its ALE formulation. The last three conditions link the problems together at the interface Γ: a kinematic condition ensures that fluid velocity is equal to structure velocity at the boundary, a dynamic condition makes sure normal stresses are consistent at the interface and the last statement imposes the boundary conditions on the mesh-motion problem to reflect the changes in geometry undergone by the structural mesh.

The incompressible Navier–Stokes equations for the fluid problem are solved using the PISO algorithm [13]. In wind-sail interaction problems, especially in downwind configurations, full RANS (*Reynolds Averaged Navier-Stokes*) simulations are needed in order to correctly model turbulence influence; in this work the $k-\omega$ *SST* model is employed (as in [20]). The structural problem is handled by the presented GPU solver. IDW (*Inverse Distance Weighting*, [36, 20]) and RBF (*Radial Basis Function*, [3]) interpolation algorithms have also been implemented in GPU in order to speed up the mesh-motion process. A GPU matrix-free approach to IDW interpolation is also presented, which reduces the memory consumption enormously and permits the usage of larger fluid meshes.

This work is divided into two parts. In the first part, mathematical models and algorithms behind the solvers are discussed. In chapter 1 the structural problem is presented, with a brief introduction to shell dynamics and details about the finite element formulation and time-advancing scheme. The fluid problem is briefly introduced in chapter 2, together with the finite

volume discretization of Navier–Stokes equations and the PISO algorithm. In chapter 3 the partitioned strongly coupled FSI solver is presented. The second part of this work is opened by a brief introduction to GPU programming, in chapter 4. It is followed by the main discussion about the GPU structural solver, in chapter 5, where the design and algorithms behind the implementation are presented and tested. The GPU implementations of interpolation procedures are explained in chapter 6. Finally, in chapter 7 the full FSI solver is presented, with accompanying results and performance analysis. At the end of this dissertation, limits of the current implementations will be discussed and possible future developments will be shown.

This work has been partially supported by Regione Lombardia and CINECA Consortium through a LISA Initiative (Laboratory for Interdisciplinary Advanced Simulation) grant.

This work is not approved or endorsed by ESI Group, the producer of the OpenFOAM® software and owner of the OpenFOAM® trade mark.

Part I
Models

In this first part, the mathematical models and formulations on which this work is based are presented, together with the underlying hypothesis assumed and some numerical schemes used in the implementation.

Chapter 1

Solid mechanics

The primary objective of this work consists in implementing a shell finite element structural solver. Shells can be used to model solids which are thinner along one dimension with respect to the others. To describe their dynamical behavior it is thus necessary to take the more general solid mechanics equations and apply approximations derived from the underlying hypothesis of the assumed shell theory.

In this work, the shell model is based on the widely accepted *Reissner–Mindlin* kinematical hypothesis [7]. It is assumed that any material line originally orthogonal to the midsurface in the undeformed configuration remains straight and unstretched during the deformations, even if it does not remain perpendicular to the midsurface. Furthermore, stresses normal to the midsurface are assumed to be zero.

In real problems objects are made by materials with different properties that often change across the whole body. For example, sails are built with fibers that have different orientations from zone to zone, such that, under load conditions, the sail tends to assume a pre-designed optimal shape. In this work, we assume a simple homogeneous isotropic linear elastic material; however, with the framework developed, the extension to more complex models should be straightforward. Moreover, in order to cope with large laminates or sail simulations, solid mechanics is analyzed in the context of small deformations and large displacements. Loads applied to this kind of structures are relatively high compared to the overall stiffness, so large cumulative displacements are expected. Locally, strains remain small and the equations over the single element can thus follow the small strain theory.

In this chapter, a basic introduction to shell dynamics is presented, with references to the Reissner–Mindlin shell theory. Firstly, static analysis is considered, with a brief discussion about the problem linearization. An approach based on finite elements is then introduced, with details about small and large displacements formulation using the adopted MITC4 element. In the final section, dynamic analysis is discussed and the chosen explicit time-advancing scheme is briefly presented.

1.1 Static analysis

In order to solve the elastic structural problem, the *Virtual Work Principle* [6] is exploited. To cope with large displacements, equilibrium has to be imposed in the deformed configuration. The VWP claims that, naming the displacement field $\mathbf{u}$ and the small strain tensor ϵ, for each valid virtual variations $\delta^t\mathbf{u}$ and $\delta^t_0\epsilon$ that satisfy the boundary conditions, we have that:

$$\int_{{}^tV} {}^t\sigma_{ij}\delta^t_0\epsilon_{ij}d^tV = \int_{{}^tV} f_i^B\delta^t u_i d^tV + \int_{{}^tS_f} f_i^{S_f}\delta^t u_i^{S_f} d^tS + \sum_m F_i^m\delta^t u_i^m, \quad (1.1)$$

where σ is the *Cauchy* stress tensor and $\mathbf{f}^B$, $\mathbf{f}^{S_f}$ and $\mathbf{F}^m$ express body, surface and concentrated loads respectively. In this notation, the subscript on the left represents the initial configuration, while the superscript, again on the left, represents the current configuration (at time t). The time indexes in this equation are fictitious, as they just indicate which configuration the variables belong to; the analysis keeps its static nature, since, at this stage, we are considering the steady problem.

Following a *total Lagrangian* approach [1], all integrals are valuated over the initial configuration. Two new tensors are then needed to express the internal work with respect to the original configuration:

$$W_{int} = \int_{{}^0V} {}^t_0\Pi_{ij}\,\delta^t_0 e_{ij}\,d^0V, \quad (1.2)$$

where $\mathbf{e}$ is the *Green–Lagrange* strain tensor and $\mathbf{\Pi}$ represents the second *Piola–Kirchhoff* tensor. In order to better represent these tensors it is necessary to introduce some other quantities.

First of all, the deformation gradient tensor:

$$ {}^{t}_{0}X_{ij} = \frac{\partial\, {}^{t}x_i}{\partial\, {}^{0}x_j}, \tag{1.3} $$

where ${}^{0}x_j$ and ${}^{t}x_i$ represent coordinates of points on the original configuration and the current configuration, namely:

$$ {}^{t}\mathbf{x} = {}^{0}\mathbf{x} + \mathbf{u}, \tag{1.4} $$

where $\mathbf{u}$ represents displacements occurred from time 0 to t.
The deformation gradient tensor expresses how fibers have stretched and rotated from time 0 to time t. It can be uniquely decomposed, using the polar decomposition theorem, into a product of an orthogonal tensor $\mathbf{R}$, representing rotations, and a symmetric positive definite tensor $\mathbf{U}$, representing stretches:

$$ {}^{t}_{0}\mathbf{X} = {}^{t}_{0}\mathbf{R}\, {}^{t}_{0}\mathbf{U}. \tag{1.5} $$

Introducing the *Cauchy–Green* right deformation tensor:

$$ {}^{t}_{0}\mathbf{C} = \left({}^{t}_{0}\mathbf{X}^{T}\, {}^{t}_{0}\mathbf{X}\right), \tag{1.6} $$

the Green–Lagrange strain tensor can then be written as:

$$ {}^{t}_{0}\mathbf{e} = \frac{1}{2}\left({}^{t}_{0}\mathbf{C} - \mathbf{I}\right) = \frac{1}{2}\left({}^{t}_{0}\mathbf{X}^{T}\, {}^{t}_{0}\mathbf{X} - \mathbf{I}\right) = \frac{1}{2}\left({}^{t}_{0}\mathbf{U}^{T}\, {}^{t}_{0}\mathbf{U} - \mathbf{I}\right). \tag{1.7} $$

From this equation it is clear how the Green–Lagrange strain tensor is invariant with respect to rigid transformations.
The second Piola–Kirchhoff stress tensor is:

$$ {}^{t}_{0}\mathbf{\Pi} = \det\left({}^{t}_{0}\mathbf{X}\right) {}^{t}_{0}\mathbf{X}^{-1}\, {}^{t}\sigma\, {}^{t}_{0}\mathbf{X}^{-T}, \tag{1.8} $$

where ${}^{t}\sigma$ is the Cauchy stress tensor. $\mathbf{\Pi}$ is symmetric and expresses the stress relative to the reference configuration. It is energy conjugate to the Green–Lagrange strain tensor and it is invariant to rigid transformations too.

It is easy to demonstrate that [6]:

$$\int_{{}^tV} \left({}^t\sigma : \delta^t\epsilon\right)\, d^tV = \int_{{}^0V} \left({}^t_0\mathbf{\Pi} : \delta^t_0\mathbf{e}\right)\, d^0V. \tag{1.9}$$

This result is of fundamental importance and allows the integration to be carried out on the reference volume.

We can now proceed with the linearization of the large displacements problem around the solution obtained at time t. The vector of displacements is decomposed into two parts, the current known solution ${}^t\mathbf{u}$ and the increment $\Delta\mathbf{u}$, such that:

$${}^{t+dt}\mathbf{u} = {}^t\mathbf{u} + \Delta\mathbf{u}. \tag{1.10}$$

The increment is the problem unknown we want to find. The same procedure is applied to the components of the Green-Lagrange strain tensor:

$$\begin{aligned} {}^{t+dt}e_{ij} &= \frac{1}{2}\left({}^{t+dt}u_{i,j} + {}^{t+dt}u_{j,i} + {}^{t+dt}u_{k,i}\,{}^{t+dt}u_{k,j}\right) \\ &= {}^te_{ij} + \frac{1}{2}\left(\Delta u_{i,j} + \Delta u_{j,i} + \Delta u_{k,i}\,{}^tu_{k,j} + {}^tu_{k,i}\Delta u_{k,j} + \Delta u_{k,i}\Delta u_{k,j}\right). \end{aligned} \tag{1.11}$$

Therefore, their virtual variations are:

$$\delta^{t+dt}e_{ij} = \delta^{t+dt}\epsilon_{ij} + \delta^{t+dt}\eta_{ij}, \tag{1.12}$$

where the linear part is:

$$\delta^{t+dt}\epsilon_{ij} = \frac{1}{2}\left(\delta\Delta u_{i,j} + \delta\Delta u_{j,i} + {}^tu_{k,i}\delta\Delta u_{k,j} + \delta\Delta u_{k,i}\,{}^tu_{k,j}\right) \tag{1.13}$$

and the nonlinear part is:

$$\delta^{t+dt}\eta_{ij} = \frac{1}{2}\left(\delta\Delta u_{k,i}\Delta u_{k,j} + \Delta u_{k,i}\delta\Delta u_{k,j}\right), \tag{1.14}$$

having deleted all other known and fixed quantities, since they have no virtual variation. The second Piola–Kirchhoff tensor is also decomposed:

$${}^{t+dt}\mathbf{\Pi} = {}^t\mathbf{\Pi} + \Delta\mathbf{\Pi}, \qquad \Delta\mathbf{\Pi} = \frac{\partial\mathbf{\Pi}}{\partial\mathbf{e}} : \Delta\mathbf{e} = \mathbf{D} : \Delta\mathbf{e}, \tag{1.15}$$

where $\Delta\mathbf{e} = {}^{t+dt}\mathbf{e} - {}^t\mathbf{e}$ and $\mathbf{D}$ is the fourth-order tensor which expresses

the material constitutive law. Under the assumption of small deformations, the law that links $\Delta\mathbf{\Pi}$ and $\Delta\mathbf{e}$ is the same that links σ and the small deformations tensor ϵ.

In order to proceed with the linearization, the nonlinear part of the Green–Lagrange strain tensor is neglected, obtaining:

$$\begin{aligned}
{}^{t+dt}\mathbf{\Pi} : \delta^{t+dt}\mathbf{e} &= \left({}^{t}\mathbf{\Pi} + \Delta\mathbf{\Pi}\right) : \delta^{t+dt}\mathbf{e} \\
&= \left({}^{t}\Pi_{ij} + D_{ijkl}\Delta e_{kl}\right) : \left(\delta^{t+dt}\epsilon_{ij} + \delta^{t+dt}\eta_{ij}\right) \\
&\cong \left({}^{t}\Pi_{ij} + D_{ijkl}\Delta\epsilon_{kl}\right) : \left(\delta^{t+dt}\epsilon_{ij} + \delta^{t+dt}\eta_{ij}\right) \\
&\cong {}^{t}\Pi_{ij}\delta^{t+dt}\epsilon_{ij} + {}^{t}\Pi_{ij}\delta^{t+dt}\eta_{ij} + D_{ij}\delta^{t+dt}\eta_{ij} + D_{ijkl}\Delta\epsilon_{kl}\delta^{t+dt}\epsilon_{ij}
\end{aligned} \tag{1.16}$$

It is now possible to rewrite the main virtual work principle equation (1.1) using the linearized expression between the second Piola–Kirchhoff tensor and the Green–Lagrange tensor. Keeping surface forces only to simplify the notation, the linearized large displacements equation of equilibrium is:

$$\int_{{}^{0}V} D_{ijkl}\Delta\epsilon_{kl}\delta^{t+dt}\epsilon_{ij}d^{0}V + \int_{{}^{0}V} {}^{t}\Pi_{ij}\delta^{t+dt}\eta_{ij}d^{0}V = \tag{1.17}$$

$$= \int_{{}^{t+dt}S_f} f_i^{S_f}\delta^{t}u_i^{S_f}d^{t+dt}S - \int_{{}^{0}V} {}^{t}\Pi_{ij}\delta^{t+dt}\epsilon_{ij}d^{0}V, \tag{1.18}$$

where all terms which do not depend on the increment $\Delta\mathbf{u}$ have been moved to the right hand side.

1.1.1 Formulation using covariant coordinates

Other than the standard orthonormal basis, in continuum mechanics other bases are commonly used, with vectors not orthogonal and not unitary in modulus. Given an Euclidean space and a fixed origin, three linearly independent vectors $\mathbf{g}_i$, with $i = 1, 2, 3$, represent a *covariant* basis for the space: a generic vector can be expressed as linear combination of the three basis vectors. It is possible to define a corresponding *contravariant* basis, with vectors $\mathbf{g}^j$, such that:

$$\mathbf{g}_i \cdot \mathbf{g}^j = \delta_{ij}, \tag{1.19}$$

where δ_{ij} is the *Kronecker delta*. A generic vector $\mathbf{u}$ can then be represented as:

$$\mathbf{u} = u_i \mathbf{g}^i \qquad \text{or} \qquad \mathbf{u} = u^i \mathbf{g}_i, \tag{1.20}$$

where u_i are the *covariant components* and u^i are the *contravariant components* or vector $\mathbf{u}$. The metric tensor can be expressed in covariant components too, defined as $g_{ij} = \mathbf{g}_i \cdot \mathbf{g}_j$, with $\mathbf{g}_i = g_{ij}\mathbf{g}^j$, or in contravariant components, defined as $g^{ij} = \mathbf{g}^i \cdot \mathbf{g}^j$, with $\mathbf{g}^i = g^{ij}\mathbf{g}_j$. These bases are particularly useful to express scalar and tensor products.

In order to simplify calculations it is often convenient to define some part of the formulation in covariant basis and other parts in contravariant basis (for details see [4]). The deformation gradient tensor can be expressed using the covariant basis in this way:

$$\overset{t}{_0}\mathbf{X} = {}^t\mathbf{g}_j \otimes {}^0\mathbf{g}^i, \tag{1.21}$$

where ${}^t\mathbf{g}_j$ and ${}^t\mathbf{g}^i$ are covariant basis and contravariant basis respectively in the reference configuration and ${}^0\mathbf{g}_j$ and ${}^0\mathbf{g}^i$ are covariant and contravariant basis in the current configuration:

$${}^0\mathbf{g}_j = \frac{\partial \mathbf{x}}{\partial \xi_j}, \qquad {}^t\mathbf{g}_j = \frac{\partial (\mathbf{x} + \mathbf{u})}{\partial \xi_j}, \tag{1.22}$$

having denoted ξ_j as the local non-Cartesian coordinates.
The Green–Lagrange strain tensor can then be written as:

$$\begin{aligned} {}^t\mathbf{e} &= \frac{1}{2}\left(\overset{t}{_0}\mathbf{X}^T \; \overset{t}{_0}\mathbf{X} - \mathbf{I}\right) \\ &= {}^t\tilde{e}_{ij}\, {}^0\mathbf{g}^i \otimes {}^0\mathbf{g}^j = \frac{1}{2}\left({}^t g_{ij} - {}^0 g_{ij}\right) {}^0\mathbf{g}^i \otimes {}^0\mathbf{g}^j. \end{aligned} \tag{1.23}$$

Therefore:

$$\begin{aligned} {}^t\tilde{e}_{ij} &= \frac{1}{2}\left({}^t\mathbf{g}_i \cdot {}^t\mathbf{g}_j - {}^0\mathbf{g}_i \cdot {}^0\mathbf{g}_j\right) = \frac{1}{2}\left(\frac{\partial (\mathbf{x} + \mathbf{u})}{\partial \xi_i} \cdot \frac{\partial (\mathbf{x} + \mathbf{u})}{\partial \xi_j} - \frac{\partial \mathbf{x}}{\partial \xi_i} \cdot \frac{\partial \mathbf{x}}{\partial \xi_j}\right) \\ &= \frac{1}{2}\left(\frac{\partial \mathbf{x}}{\partial \xi_i} \cdot \frac{\partial \mathbf{u}}{\partial \xi_j} + \frac{\partial \mathbf{u}}{\partial \xi_i} \cdot \frac{\partial \mathbf{x}}{\partial \xi_j} + \frac{\partial \mathbf{u}}{\partial \xi_i} \cdot \frac{\partial \mathbf{u}}{\partial \xi_j}\right). \end{aligned} \tag{1.24}$$

The linear and nonlinear parts can be separated:

$$\begin{aligned} {}^{t}\tilde{e}_{ij}|_{linear} &= \frac{1}{2}\left(\frac{\partial \mathbf{x}}{\partial \xi_i}\cdot\frac{\partial \mathbf{u}}{\partial \xi_j}+\frac{\partial \mathbf{u}}{\partial \xi_i}\cdot\frac{\partial \mathbf{x}}{\partial \xi_j}\right) \\ {}^{t}\tilde{e}_{ij}|_{nonlinear} &= \frac{1}{2}\left(\frac{\partial \mathbf{u}}{\partial \xi_i}\cdot\frac{\partial \mathbf{u}}{\partial \xi_j}\right). \end{aligned} \tag{1.25}$$

Under the hypothesis of small displacements, the nonlinear part of the Green–Lagrange strain tensor is neglected.

In order to speed up the calculations, it is possible to avoid continuous transformations between covariant and Cartesian coordinates, exploiting the nice properties of covariant and contravariant components when applied in double dot products. To calculate the deformation energy, contravariant components of the stress tensor will be multiplied against covariant components of the strain tensor.

In order to enforce in a correct way the plane stress state hypothesis, typical for shells, it is necessary to define a local Cartesian reference system aligned with the shell, to correctly follow the surface normal evolution:

$$\mathbf{e}_1 = \frac{\mathbf{g}_1}{\|\mathbf{g}_1\|}, \qquad \mathbf{e}_2 = \mathbf{e}_3 \wedge \mathbf{e}_1, \qquad \mathbf{e}_3 = \frac{\mathbf{g}_1 \wedge \mathbf{g}_2}{\|\mathbf{g}_1 \wedge \mathbf{g}_2\|}. \tag{1.26}$$

Covariant and controvariant components of stresses and strains respectively can be written in the following way:

$$\begin{aligned} \sigma &= \tilde{\sigma}^{\alpha\beta}\,\mathbf{g}_\alpha \otimes \mathbf{g}_\beta = \sigma_{ij}\,\mathbf{e}_i \otimes \mathbf{e}_j, \\ \epsilon &= \tilde{\epsilon}_{\alpha\beta}\,\mathbf{g}^\alpha \otimes \mathbf{g}^\beta = \epsilon_{ij}\,\mathbf{e}_i \otimes \mathbf{e}_j. \end{aligned} \tag{1.27}$$

Applying scalar products it is possible to write that:

$$\begin{aligned} \tilde{\sigma}^{\alpha\beta} &= \sigma_{ij}\Big(\mathbf{e}_i \cdot \mathbf{g}^\beta\Big)\Big(\mathbf{e}_j \cdot \mathbf{g}^\alpha\Big), \\ \epsilon_{ij} &= \tilde{\epsilon}_{\alpha\beta}\Big(\mathbf{g}^\alpha \cdot \mathbf{e}_j\Big)\Big(\mathbf{g}^\beta \cdot \mathbf{e}_i\Big). \end{aligned} \tag{1.28}$$

In order to use a more compact notation the following vectors are introduced:

$$\tilde{\epsilon} = \begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ \gamma_{12} \\ \gamma_{23} \\ \gamma_{13} \end{bmatrix}, \qquad \tilde{\sigma} = \begin{bmatrix} \sigma^{11} \\ \sigma^{22} \\ \sigma^{33} \\ \tau^{12} \\ \tau^{23} \\ \tau^{13} \end{bmatrix}. \tag{1.29}$$

We call $\mathbf{Q}_{sh}$ the transformation matrix such that:

$$\epsilon = \mathbf{Q}_{sh} \cdot \tilde{\epsilon}. \tag{1.30}$$

Equation (1.28) shows how the matrix which transforms stresses from Cartesian to contravariant components is in fact the transposed of $\mathbf{Q}_{sh}$, the matrix which transforms strains from covariant to Cartesian components.
Finally, the deformation energy can be calculated:

$$\begin{aligned} 2W &= \sigma_{ij}\epsilon_{ij} = \tilde{\sigma}^{\alpha\beta}\tilde{\epsilon}_{\alpha\beta} = \tilde{\epsilon}^T \cdot \tilde{\sigma} \\ &= \epsilon^T \cdot \sigma = \epsilon^T \cdot \mathbf{D} \cdot \epsilon = \tilde{\epsilon}^T \cdot \mathbf{Q}_{sh}^T \cdot \mathbf{D} \cdot \mathbf{Q}_{sh} \cdot \tilde{\epsilon}, \end{aligned} \tag{1.31}$$

where $\mathbf{D}$ is the elastic constitutive matrix, which relates stresses and strains:

$$\sigma = \mathbf{D} \cdot \epsilon \tag{1.32}$$

Under the assumption of plane stress state, in a Cartesian reference system aligned with the shell element, the matrix can be written as [5]:

$$\mathbf{D} = \frac{E}{1-\nu^2} \cdot \begin{bmatrix} 1 & \nu & 0 & 0 & 0 & 0 \\ \nu & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & k\frac{1-\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & k\frac{1-\nu}{2} \end{bmatrix}, \tag{1.33}$$

where E is the Young modulus, ν is the Poisson ratio and k the shear factor, which is equal to $\frac{5}{6}$.

When large displacements are expected, it is necessary to formulate the virtual work equation using the Green–Lagrange strain tensor and the sec-

ond Piola–Kirchhoff stress tensor, as described in the previous section. The latter one can be written using covariant coordinates as:

$$\begin{aligned} {}^{t}_{0}\mathbf{\Pi} &= \det\left({}^{t}_{0}\mathbf{X}\right) {}^{t}_{0}\mathbf{X}^{-1} \cdot {}^{t}\sigma \cdot {}^{t}_{0}\mathbf{X}^{-T} \\ &= \det\left({}^{t}_{0}\mathbf{X}\right) \left({}^{0}\mathbf{g}_j \otimes {}^{t}\mathbf{g}^j\right) \cdot \left(\tilde{\sigma}^{mn}\, {}^{t}\mathbf{g}_m \otimes {}^{t}\mathbf{g}_n\right) \cdot \left({}^{t}\mathbf{g}^p \otimes {}^{0}\mathbf{g}_p\right) \\ &= \det\left({}^{t}_{0}\mathbf{X}\right) \tilde{\sigma}^{mn}\, {}^{0}\mathbf{g}_m \otimes {}^{0}\mathbf{g}_n. \end{aligned} \tag{1.34}$$

The linearization procedure is the same as the previous section. The displacement vector is decomposed in a known part and the increment:

$${}^{t+dt}\mathbf{u} = {}^{t}\mathbf{u} + \Delta\mathbf{u}. \tag{1.35}$$

The Green–Lagrange tensor components are decomposed too:

$$\begin{aligned} {}^{t+dt}\tilde{e}_{ij} &= \frac{1}{2}\left({}^{0}\mathbf{g}_i \cdot \frac{\partial^{t+dt}\mathbf{u}}{\partial\xi_j} + {}^{0}\mathbf{g}_j \cdot \frac{\partial^{t+dt}\mathbf{u}}{\partial\xi_i} + \frac{\partial^{t+dt}\mathbf{u}}{\partial\xi_i} \cdot \frac{\partial^{t+dt}\mathbf{u}}{\partial\xi_j}\right) \\ &= \frac{1}{2}\Bigg({}^{0}\mathbf{g}_i \cdot \frac{\partial^{t}\mathbf{u}}{\partial\xi_j} + {}^{t}\mathbf{g}_i \cdot \frac{\partial\Delta\mathbf{u}}{\partial\xi_j} + {}^{0}\mathbf{g}_j \cdot \frac{\partial^{t}\mathbf{u}}{\partial\xi_i} + {}^{t}\mathbf{g}_j \cdot \frac{\partial\Delta\mathbf{u}}{\partial\xi_i} \\ &\qquad + \frac{\partial^{t}\mathbf{u}}{\partial\xi_i} \cdot \frac{\partial^{t}\mathbf{u}}{\partial\xi_j} + \frac{\partial\Delta\mathbf{u}}{\partial\xi_i} \cdot \frac{\partial\Delta\mathbf{u}}{\partial\xi_j}\Bigg). \end{aligned} \tag{1.36}$$

The virtual variation is then calculated for each term, obtaining:

$$\delta^{t+dt}\tilde{e}_{ij} = \delta^{t+dt}\epsilon_{ij} + \delta^{t+dt}\eta_{ij}, \tag{1.37}$$

where:

$$\begin{aligned} \delta^{t+dt}\epsilon_{ij} &= \frac{1}{2}\left({}^{t}\mathbf{g}_i \cdot \frac{\partial\delta\Delta\mathbf{u}}{\partial\xi_j} + {}^{t}\mathbf{g}_j \cdot \frac{\partial\delta\Delta\mathbf{u}}{\partial\xi_i}\right), \\ \delta^{t+dt}\eta_{ij} &= \frac{1}{2}\left(\frac{\partial\delta\Delta\mathbf{u}}{\partial\xi_i} \cdot \frac{\partial\Delta\mathbf{u}}{\partial\xi_j} + \frac{\partial\Delta\mathbf{u}}{\partial\xi_i} \cdot \frac{\partial\delta\Delta\mathbf{u}}{\partial\xi_j}\right) \end{aligned} \tag{1.38}$$

are the linear and nonlinear components respectively. In the problem linearization the latter ones are neglected. Taking also into consideration the underlying hypothesis of the shell kinematics theory it is then possible to linearize the double scalar product between the second Piola–Kirchhoff stress tensor and the linear part of the Green-Lagrange strain tensor.

1.2 The MITC4 element

As finite element to discretize shells we consider the *MITC4* (*Mixed Interpolation of Tensorial Components*) element [1, 2, 4]. It is the four node version of the generic *MITCn* element, where n represents the number of nodes. They are *Solid-Like Shell Elements*: they were built starting from a tridimensional solid *brick* element and then imposing kinematic constraints typical of thin laminates. The element is outlined in figure 1.1.

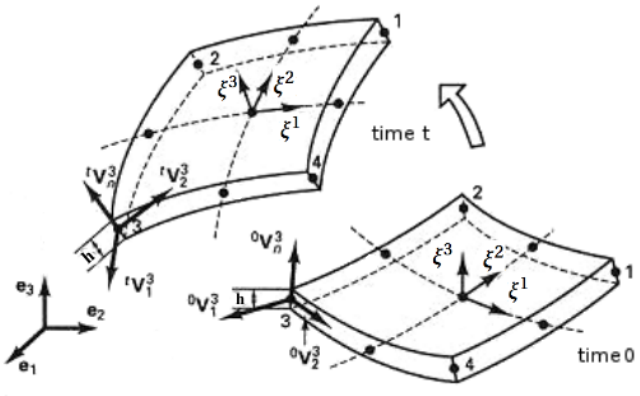


Figure 1.1: MITC4 shell element.

Considering a generic shell body, in order to describe its configurations, a fixed Cartesian coordinate system { X, Y, Z } is introduced, with versors $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, together with a local coordinate system { ξ_1, ξ_2, ξ_3 }, not necessarily orthogonal. The position of a generic point on the shell body in the reference configuration is described by the following mapping:

$$ {}^0\mathbf{X} = \mathbf{\Phi}(\xi) = \bar{\mathbf{\Phi}}\left(\xi^1, \xi^2\right) + \xi^3 \mathbf{L}\left(\xi^1, \xi^2\right), \quad -\frac{t_h}{2} \leq \xi^3 \leq \frac{t_h}{2}, \tag{1.39} $$

where t_h is the shell thickness (here for simplicity assumed to be constant over the whole body), $\bar{\mathbf{\Phi}}$ is the in-plane midsurface chart and $\mathbf{L}$ represents the out-of-plane director vector field.

The MITC4 is an isoparametric element: the same shape functions are used both to transform from global Cartesian coordinates to local coordinates and to interpolate displacements in a generic point inside the element

from its nodal values. The bilinear shape functions are:

$$h_1\left(\xi^1,\xi^2\right)=\frac{1}{4}\left(1-\xi^1\right)\left(1-\xi^2\right), \tag{1.40}$$

$$h_2\left(\xi^1,\xi^2\right)=\frac{1}{4}\left(1+\xi^1\right)\left(1-\xi^2\right), \tag{1.41}$$

$$h_3\left(\xi^1,\xi^2\right)=\frac{1}{4}\left(1+\xi^1\right)\left(1+\xi^2\right), \tag{1.42}$$

$$h_4\left(\xi^1,\xi^2\right)=\frac{1}{4}\left(1-\xi^1\right)\left(1+\xi^2\right). \tag{1.43}$$

In each node k a director vector $\mathbf{V}_n^k$ is defined such that it is orthogonal to the midsurface in the reference configuration and rotates during the deformations. In global Cartesian coordinate system the position of a generic point at time t will then be:

$${}^t\mathbf{x}\left(\xi^1,\xi^2,\xi^3\right)=\sum_{k=1}^{4}h_k\,{}^t\mathbf{x}_k+\xi^3\sum_{k=1}^{4}\left(\frac{t_h}{2}h_k\,{}^t\mathbf{V}_n^k\right), \tag{1.44}$$

where the first part refers to in-plane interpolation, while the second part is the out-of-plane interpolation.
In a total Lagrangian approach, displacements are obtained from the current positions with respect to the original positions in the reference configuration. Using the vector:

$$\mathbf{V}_n^k={}^t\mathbf{V}_n^k-{}^0\mathbf{V}_n^k \tag{1.45}$$

displacements are obtained as it follows:

$$\begin{aligned}{}^t\mathbf{u}\left(\xi^1,\xi^2,\xi^3\right)&=\sum_{k=1}^{4}h_k\left({}^t\mathbf{x}_k-{}^0\mathbf{x}_k\right)+\xi^3\sum_{k=1}^{4}\frac{t_k}{2}h_k\left({}^t\mathbf{V}_n^k-{}^0\mathbf{V}_n^k\right)\\&=\sum_{k=1}^{4}h_k\,{}^t\mathbf{u}_k+\xi^3\sum_{k=1}^{4}\frac{t_k}{2}h_k\mathbf{V}_n^k.\end{aligned} \tag{1.46}$$

These equations are used to calculate the components of the Green–Lagrange strain tensor and the second Piola–Kirchhoff tensor.

1.2.1 Small displacements formulation

Under the hypothesis of small rotations, it is possible to describe the components of $\mathbf{V}_n^k$ in each node as small rotations around a local orthonor-

mal basis, composed of:

- ${}^0\mathbf{V}_n^k$;
- ${}^0\mathbf{V}_1^k = \frac{\mathbf{i}_y \times {}^0\mathbf{V}_n^k}{\|\mathbf{i}_y \times {}^0\mathbf{V}_n^k\|_2}$, or ${}^0\mathbf{V}_1^k = \mathbf{i}_z$ if denominator is zero;
- ${}^0\mathbf{V}_2^k = {}^0\mathbf{V}_n^k \times {}^0\mathbf{V}_1^k$.

Under this assumption, changes of director versor can be described using these vectors:

$$\mathbf{V}_n^k = -{}^0\mathbf{V}_2^k \alpha_k + {}^0\mathbf{V}_1^k \beta_k, \tag{1.47}$$

where α_k represents the small rotation around ${}^0\mathbf{V}_1^k$ and β_k the small rotation around ${}^0\mathbf{V}_2^k$. The mid-plane torsion is not considered, so the component parallel to normal director version is neglected. In this way, only two rotational degrees of freedom are needed for each node, summing up to 20 degrees of freedom per element (3 translational and 2 rotational in each of the 4 nodes).
In order to solve the elastic problem, the principle of virtual work, introduced above, is used. Using two matrices $\mathbf{B}$ and $\mathbf{H}$, it is possible to express displacements and deformations with respect to nodal values, so that the VWP equation (1.1) becomes:

$$\begin{aligned} \int_{{}^tV} \delta\mathbf{U}^T \cdot \mathbf{B}^T \cdot \mathbf{D} \cdot \mathbf{B} \cdot \mathbf{U} d^tV = &\int_{{}^tV} \delta\mathbf{U}^T \cdot \mathbf{H} \cdot \mathbf{f}^B d^tV \\ &+ \int_{{}^tS_f} \delta\mathbf{U}^T \cdot \mathbf{H}_{S_f} \cdot \mathbf{f}^{S_f} d^tS \\ &+ \sum_m \delta\mathbf{U}^T \cdot \mathbf{H}_m \cdot F^m, \end{aligned} \tag{1.48}$$

having used the relations:

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = \mathbf{H} \cdot \mathbf{U}, \qquad \epsilon = \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{bmatrix} = \mathbf{B} \cdot \mathbf{U}, \qquad \sigma = \begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{bmatrix} = \mathbf{D} \cdot \epsilon. \tag{1.49}$$

From here on, for the sake of simplicity, only surface forces will be considered. Equation (1.48) is then simplified to:

$$\int_{{}^tV} \delta\mathbf{U}^T \cdot \mathbf{B}^T \cdot \mathbf{D} \cdot \mathbf{B} \cdot \mathbf{U} d^tV = \int_{{}^tS_f} \delta\mathbf{U}^T \cdot \mathbf{H}_{S_f} \cdot \mathbf{f}^{S_f} d^tS. \tag{1.50}$$

It is now necessary to perform the integration on the finite elements. Considering that U contains nodal values (thus it is constant) and the Virtual Work Principle must be satisfied for every virtual displacements field $\delta\mathbf{U}$, equation (1.50) becomes:

$$\mathbf{K} \cdot \mathbf{U} = \mathbf{F}_E, \tag{1.51}$$

where:

$$\mathbf{K} = \int_{{}^tV} \mathbf{B}^T \cdot \mathbf{D} \cdot \mathbf{B} d^tV, \qquad \mathbf{F}_E = \int_{{}^tS_f} \mathbf{H}_{S_f} \cdot \mathbf{f}^{S_f} d^tS. \tag{1.52}$$

Matrices **B** and **H** are calculated considering the kinematic relations and assumptions made for the MITC4 element in the previous paragraph. Considering the single element, vector **U** contains the generalized nodal displacements:

$$\mathbf{U}^T = \begin{bmatrix} u_{x1} & u_{y1} & u_{z1} & \alpha_1 & \beta_1 & u_{x2} & u_{y2} & u_{z2} & \alpha_2 & \beta_2 & ... \end{bmatrix}. \tag{1.53}$$

Matrix **H** (3x20 elements) derives directly from equation (1.46) and expresses displacements from nodal values in **U**. Matrix **B** (6x20 elements), used to express strains, is trickier to obtain, since it requires calculating displacement derivatives with respect to the global coordinate system. Thus it is necessary to define the Jacobian matrix of local-to-global coordinates transformation:

$$J = \begin{bmatrix} \frac{\partial x}{\partial \xi_1} & \frac{\partial y}{\partial \xi_1} & \frac{\partial z}{\partial \xi_1} \\ \frac{\partial x}{\partial \xi_2} & \frac{\partial y}{\partial \xi_2} & \frac{\partial z}{\partial \xi_2} \\ \frac{\partial x}{\partial \xi_3} & \frac{\partial y}{\partial \xi_3} & \frac{\partial z}{\partial \xi_3} \end{bmatrix}. \tag{1.54}$$

Therefore displacement derivatives are calculated with respect to local coordinates at first and then they are transformed to global coordinates. Con-

sidering, for example, the first component of displacement $\mathbf{u}$:

$$\begin{bmatrix} \frac{\partial u_x}{\partial x} \\ \frac{\partial u_x}{\partial y} \\ \frac{\partial u_x}{\partial z} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial u_x}{\partial \xi_1} \\ \frac{\partial u_x}{\partial \xi_2} \\ \frac{\partial u_x}{\partial \xi_3} \end{bmatrix}. \tag{1.55}$$

Finally, $\mathbf{K}$ and $\mathbf{F}$ are obtained performing the integration of the reference element, multiplying it by the Jacobian:

$$\begin{aligned} \mathbf{K} &= \int_{-1}^{+1}\int_{-1}^{+1}\int_{-1}^{+1} \mathbf{B}^T(\xi_1,\xi_2,\xi_3)\cdot\mathbf{D}_{sh}\cdot\mathbf{B}(\xi_1,\xi_2,\xi_3)\det(\mathbf{J})\,d\xi_1\,d\xi_2\,d\xi_3, \\ \mathbf{F}_E &= \int_{-1}^{+1}\int_{-1}^{+1} \mathbf{H}_{S_f}\left(\xi_1,\xi_2,\bar{\xi_3}\right)\cdot\mathbf{f}^{S_f}\left(\xi_1,\xi_2,\bar{\xi_3}\right)\left\|\frac{\partial\mathbf{x}}{\partial\xi_1}\wedge\frac{\partial\mathbf{x}}{\partial\xi_2}\right\| d\xi_1\,d\xi_2, \end{aligned} \tag{1.56}$$

where $\mathbf{D}_{sh}$ is the usual elastic constitutive matrix under the assumption of plane stresses, expressed in the global coordinate system using an appropriate transformation matrix $\mathbf{Q}_{sh}$:

$$\mathbf{D}_{sh} = \mathbf{Q}_{sh}^T\cdot\mathbf{D}\cdot\mathbf{Q}_{sh}. \tag{1.57}$$

Integrals are evaluated numerically using Gaussian quadrature.

1.2.2 Large displacements formulation

When it is necessary to cope with large displacements the previously defined rotations α_k and β_k around the local nodal basis cannot be used anymore. With this purpose, in this work the formulation described in [5] has been adopted.

The difficulty resides in the expression of the difference between ${}^{t+dt}\mathbf{V}_n^k$ and ${}^{t}\mathbf{V}_n^k$, since these directors can undergo large rotations. It is then necessary to express them as rigid rotations.
The general expression for a rotation matrix is:

$$\mathbf{R} = \mathbf{I} + \sin(\theta)\,\mathbf{W} + (1-\cos(\theta))\,\mathbf{W}^2, \tag{1.58}$$

where $\mathbf{I}$ is the identity matrix, θ is the angle of rotation and $\mathbf{W}$ is a skew-symmetric matrix such that:

$$\mathbf{W}\mathbf{u} = \mathbf{k}\wedge\mathbf{u}, \qquad \forall\mathbf{u}\in\mathbb{R}^3, \tag{1.59}$$

where $\mathbf{k}$ is the rotation axis. It can be written, in the global Cartesian reference system, as:

$$\mathbf{W} = \begin{bmatrix} 0 & -k_3 & k_2 \\ k_3 & 0 & -k_1 \\ -k_2 & k_1 & 0 \end{bmatrix}. \tag{1.60}$$

It is possible to express the rotation using a single vector $\boldsymbol{\phi}$, parallel to the axis $\mathbf{k}$ and with modulus equal to θ. In this way:

$$\mathbf{R} = \mathbf{I} + \frac{\sin(\theta)}{\theta}\boldsymbol{\Phi} + \frac{(1-\cos(\theta))}{\theta^2}\boldsymbol{\Phi}^2, \tag{1.61}$$

where Φ is a skew-symmetric matrix built in the same way as $\mathbf{W}$ with the components of $\boldsymbol{\phi}$.

A generic rotation in three dimensions can be seen as three different consecutive rotations around three orthogonal axes. The rotations are not commutative, so their order is important. Using the local nodal basis in the reference configuration (described in the previous section), if the rotation around the midsurface normal vector is performed first, there is no transformation of the director. Therefore, the vector ϕ holds only two independent components in every node. It can be written as:

$$\tilde{\boldsymbol{\phi}}_k = \begin{bmatrix} \alpha_k & \beta_k & 0 \end{bmatrix}, \qquad \theta = \sqrt{\alpha_k^2 + \beta_k^2}, \tag{1.62}$$

where, in contrast with the previous section, α_k and β_k are not angles, but just independent components of the vector ϕ. In global Cartesian coordinates it is equal to:

$$\boldsymbol{\phi}_k = \alpha_k \mathbf{V}_1^k + \beta_k \mathbf{V}_2^k. \tag{1.63}$$

This approach has the advantage of storing only two rotational components for each node, allowing memory saving.

Differently from the case of small displacements, when dealing with large displacements it is necessary to solve the elastic problem by steps. The solution is obtained through incremental updates, which can be written, at iteration n, as:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \Delta\mathbf{U}^n, \tag{1.64}$$

where $\Delta\mathbf{U}^n$ is the displacement increment that can be obtained solving:

$$\mathbf{K}_T\left(\mathbf{U}^n\right)\cdot\Delta\mathbf{U}^n=\mathbf{F}_E-\mathbf{F}_I\left(\mathbf{U}^n\right), \tag{1.65}$$

where $\mathbf{F}_E$ is a vector containing all external loads and $\mathbf{F}_I$ collects internal elastic forces on nodes. $\mathbf{K}_T$ is the tangent stiffness matrix, obtained linearizing the virtual work principle against increments of translational and rotational nodal values.

First of all, increment of displacements in a generic point has to be written. In order to do so, calling $\boldsymbol{\psi}_k$ the vector which expresses the rotation of director in node k from time t to $t+dt$, it is necessary to write the rotation matrix $\mathbf{R}\left(\boldsymbol{\psi}_k\right)$. An *updated Lagrangian* approach is mandatory in this case, since the series expansion of $\mathbf{R}$ is acceptable only when dealing with small rotations. Stopping the expansion at the second order, increments can be written as:

$$\Delta\mathbf{u}\left(\xi^1,\xi^2,\xi^3\right)=\sum_{k=1}^{4}h_k\Delta\mathbf{u}_k+\xi^3\sum_{k=1}^{4}\frac{t_k}{2}h_k\left(\boldsymbol{\psi}_k+\frac{1}{2}\boldsymbol{\psi}_k\cdot\boldsymbol{\Psi}_k\right){}^t\mathbf{V}_n^k, \tag{1.66}$$

where $\boldsymbol{\Psi}_k$ is the antisymmetric matrix calculated with $\boldsymbol{\psi}_k$.
Their virtual variation is then calculated and decomposed into a constant and a variable part:

$$\begin{aligned}\delta\Delta\mathbf{u}_{const}\left(\xi^1,\xi^2,\xi^3\right)&=\sum_{k=1}^{4}h_k\delta\Delta\mathbf{u}_k+\xi^3\sum_{k=1}^{4}\frac{t_k}{2}h_k\delta\boldsymbol{\Psi}_k{}^t\mathbf{V}_n^k,\\ \delta\Delta\mathbf{u}_{var}\left(\xi^1,\xi^2,\xi^3\right)&=\frac{\xi^3}{2}\sum_{k=1}^{4}\frac{t_k}{2}h_k\left(\delta\boldsymbol{\Psi}_k\cdot\boldsymbol{\Psi}_k+\boldsymbol{\Psi}_k\cdot\delta\boldsymbol{\Psi}_k\right){}^t\mathbf{V}_n^k.\end{aligned} \tag{1.67}$$

These are used to calculate Green–Lagrange strain tensor components and put in the final equation, where terms above the first order (in increments of nodal parameters) are neglected. In covariant components this leads to

the linearized equation:

$$\int_{^0V} D^{\alpha\beta\gamma\delta}\Delta\epsilon_{\gamma\delta}\delta^{t+dt}\epsilon^{const}_{\alpha\beta}d^0V + \int_{^0V} {}^t\Pi^{\alpha\beta}\delta^{t+dt}\eta_{\alpha\beta}d^0V + \tag{1.68}$$

$$+\int_{^0V} {}^t\Pi^{\alpha\beta}\delta^{t+dt}\epsilon^{var}_{\alpha\beta}d^0V - \int_{^{t+dt}S_f} f^{\alpha}_{S_f}\delta^{t+dt}u^{var}_{\alpha}d^{t+dt}S = \tag{1.69}$$

$$= \int_{^{t+dt}S_f} f^{\alpha}_{S_f}\delta^{t+dt}u^{const}_{\alpha}d^{t+dt}S - \int_{^0V} {}^t\Pi^{\alpha\beta}\delta^{t+dt}\epsilon^{const}_{\alpha\beta}d^0V. \tag{1.70}$$

Left-hand-side terms contributes to the tangent stiffness matrix calculation, while right-hand-side terms are expression of external forces vector $\mathbf{F}_E$ and nodal forces vector (equivalent to internal elastic forces) $\mathbf{F}_I$ respectively. These last terms express the imbalance between internal and external loads and thus they can be used as residual for an iterative method.

1.2.3 Mixed interpolation

The *shear-locking* problem is very common in finite element analysis, which derives from the underlying assumptions of piecewise polynomial behavior dictated by the finite element shape functions. This acts as an additional constraint on the whole element behavior. In the case of shell elements, it results in an artificious shear stiffness larger than the real system, especially when thickness is small compared to element size in the other directions.

In order to overcome the shear-locking problem, in [1] the covariant components of in-plane deformations are interpolated separately from the out-of-plane deformations. This process characterizes the MITCn elements and their name: *Mixed Interpolation of Tensorial Components*.
Considering the element on the ξ_1-ξ_2 plane sketched in figure 1.2, the sampling points A, B, C and D have local coordinates (0, -1, 0), (1, 0, 0), (0, 1, 0) and (-1, 0, 0) respectively. Covariant components of shear deformations in the sampling points are obtained through interpolation from the known nodal values:

$$\begin{aligned}\tilde{\epsilon}_{23} &= \frac{1}{2}(1+\xi_1)\,\tilde{\epsilon}^A_{23} + \frac{1}{2}(1-\xi_1)\,\tilde{\epsilon}^C_{23},\\ \tilde{\epsilon}_{13} &= \frac{1}{2}(1+\xi_2)\,\tilde{\epsilon}^D_{13} + \frac{1}{2}(1-\xi_2)\,\tilde{\epsilon}^B_{13}.\end{aligned} \tag{1.71}$$

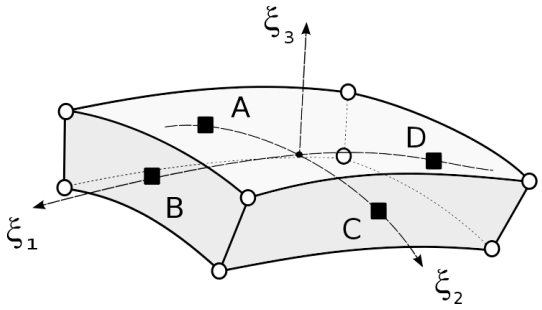


Figure 1.2: Sampling points considered for mixed interpolation.

When assembled, these changes in the covariant components induce a modification in the whole small strain tensor in Cartesian coordinates, in-plane deformations included. With this kind of interpolation, the element presents sort of membrane locking only in an highly *warped* configuration (figure 1.3): in this situation it becomes necessary to refine the mesh.

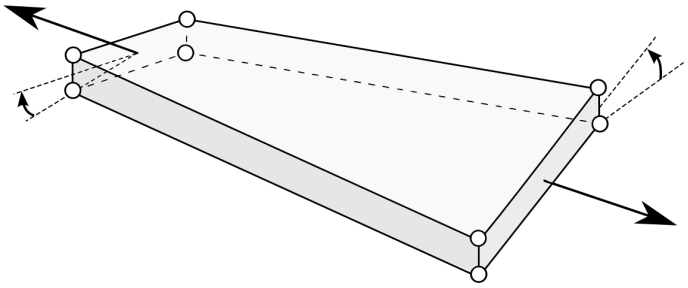


Figure 1.3: Warped shell element.

1.3 Dynamic analysis

Our goal is to simulate the dynamical behavior of shell structures. Until now we have presented a static model. It is now necessary to introduce the time variable and the inertial components in the virtual work principle equation (1.1):

$$\int_{t+\Delta t V} \rho \ddot{u}_i \delta u_i d^{t+\Delta t}V + \int_{t+\Delta t V} c\dot{u}_i \delta u_i d^{t+\Delta t}V + \int_{{}^0V} \left({}_0^{t+\Delta t}\mathbf{\Pi} : \delta_0^{t+\Delta t}\mathbf{e}\right) d^0V =$$
$$= \int_{t+\Delta t V} f_i^B \delta^{t+\Delta t} u_i d^{t+\Delta t}V + \int_{t+\Delta t S_f} f_i^{S_f} \delta^{t+\Delta t} u_i^{S_f} d^{t+\Delta t}S + \sum_m F_i^m \delta^{t+\Delta t} u_i^m.$$

Following the notations and variables introduced in last sections, the mass matrix $\mathbf{M}$ is introduced, whose components are:

$$M_{ij} = \int_{t+\Delta t V} \rho H_i H_j d^{t+\Delta t}V, \tag{1.72}$$

where ρ is the material density, and the damping matrix $\mathbf{C}$, whose components are:

$$C_{ij} = \int_{t+\Delta t V} c H_i H_j d^{t+\Delta t}V, \tag{1.73}$$

where c represents the damping coefficient. Actually, the calculation of the damping matrix $\mathbf{C}$ is not performed this way, since the coefficient c is hardly known. There are different techniques to build it. One of the most popular, implemented in this work, combines linearly the mass and stiffness matrix with appropriate coefficients.
It is now possible to write the semi-discretized motion equation:

$$\mathbf{M}\ddot{\mathbf{U}} + \mathbf{C}\dot{\mathbf{U}} + \mathbf{F}_I(\mathbf{U}) = \mathbf{F}_E, \tag{1.74}$$

where $\mathbf{U}$, $\dot{\mathbf{U}}$ and $\ddot{\mathbf{U}}$ are vectors containing the degrees of freedom values, velocity and acceleration respectively, $\mathbf{F}_I$ represents the internal elastic forces and $\mathbf{F}_E$ the external loads respectively. Equation (1.74) is a system of nonlinear second order differential equations and an appropriate time-advancing scheme is needed to integrate it. Generally, they can be classified into implicit and explicit time integration methods. The former ones impose

equilibrium of the system at the end of the time-step and are generally unconditionally stable but computationally heavy, since they require matrix inversions and sub-iterations in order to cope with the nonlinearities. Explicit methods, instead, expect equilibrium at the beginning and do not iterate inside the time-step, assuming a time-step size small enough. Therefore, they are only conditionally stable and end up requiring many small but quick time-steps. Their structure suits better the GPU hardware, thus in this work an explicit integrator is used.

1.3.1 Central difference method

Integration in time is done using an explicit half-station central finite differences approach [30, 37]. It is simple and has the largest stability limit for second-order accurate integration formulas [18], and represents a common choice for this kind of problems.
At the beginning of time-step n, the internal forces vector $\mathbf{F}_I^n$ has already been calculated in the previous step and the external forces vector $\mathbf{F}_E^n$ is assumed to be known.
Given the amplitude of time-step Δt, the velocity field is firstly approximated at half-step $n + 1/2$:

$$\dot{\mathbf{U}}^{n+1/2} = \frac{1}{\Delta t}\left(\mathbf{U}^{n+1} - \mathbf{U}^{n}\right). \tag{1.75}$$

Then the acceleration is evaluated at the n-th step:

$$\ddot{\mathbf{U}}^{n} = \frac{1}{\Delta t}\left(\dot{\mathbf{U}}^{n+1/2} - \dot{\mathbf{U}}^{n-1/2}\right) \tag{1.76}$$

and velocity is updated:

$$\dot{\mathbf{U}}^{n} = \frac{1}{2}\left(\dot{\mathbf{U}}^{n+1/2} + \dot{\mathbf{U}}^{n-1/2}\right). \tag{1.77}$$

Assuming the damping matrix proportional to the mass matrix, such that:

$$\mathbf{C} = \alpha\mathbf{M}, \tag{1.78}$$

with $\alpha \geq 0$, and putting equations (1.75), (1.76), (1.77) inside (1.74) we obtain:

$$\dot{\mathbf{U}}^{n+1/2} = \left(\frac{2 \cdot \Delta t}{2 + \alpha \Delta t}\right) \mathbf{M}^{-1} \left(\mathbf{F}_E^n - \mathbf{F}_I^n\right) + \left(\frac{2 - \alpha \Delta t}{2 + \alpha \Delta t}\right) \dot{\mathbf{U}}^{n-1/2}. \tag{1.79}$$

New displacements can then be obtained integrating in time, using the calculated half-step velocities:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \Delta t \; \dot{\mathbf{U}}^{n+1/2}. \tag{1.80}$$

Once displacements are known, it is possible to compute the strain field inside each finite element and then the stress field can be evaluated by integrating the constitutive law. The new internal forces vector $\mathbf{F}_I^{n+1}$ is then used in the next time-step to calculate the accelerations.

This explicit method, in conjunction with mass lumping, which makes the mass matrix diagonal, has a remarkable advantage: it does not require any matrix factorization or inversion, but instead transforms the problem (1.74) into an uncoupled system of equations, that can be solved individually on a per-degree-of-freedom basis (this fact will be highly exploited in the parallelized algorithm). The only complex operation that needs to be done for every time-step is the internal forces calculation. The computational burden of each time-step is very low, but the method is not unconditionally stable: it requires Δt to be lower than a critical value: $\Delta t \leq \Delta t_{crit}$. This is called the *Courant-Friedrichs-Levy* condition [8, 30]. In particular, Δt should be the smallest period of vibration of the system, considering all degrees of freedom. When structural elements are concerned, in [1] a limit value of $\Delta t_{crit} = T_n/\pi$ is proposed, where T_n is the mesh smallest eigenperiod: following this criterion, it is possible to select a lower bound to the time-step size as $\Delta t = 2/\omega_m$, where ω_m is the highest frequency over the whole mesh related to a single finite element. Generally this limit value can be very low. This represents the major disadvantage of using an explicit time advancing method.

Chapter 2

Fluid dynamics

In the fluid-structure interaction cases studied in this work a solid mechanics problem is coupled with a fluid dynamics problem. The first one has been introduced in the previous chapter. The mathematical model underlying the latter one will be presented in the following sections, together with the numerical methods employed in OpenFOAM to solve it.

2.1 *Navier–Stokes* equations

Considering a generic Newtonian fluid, it holds that:

$$\begin{cases} \dfrac{\partial(\rho\mathbf{v})}{\partial t} + \operatorname{div}(\rho\mathbf{v}\mathbf{v}) = \operatorname{div}(\sigma) + \rho\mathbf{b} \\ \dfrac{\partial\rho}{\partial t} + \operatorname{div}(\rho\mathbf{v}) = 0 \end{cases}, \tag{2.1}$$

where $\mathbf{v}$ is the fluid velocity, ρ is its density and $\mathbf{b}$ represents external volume forces (per unit mass). The first equation expresses the conservation of momentum, while the second one, the continuity equation, expresses the conservation of mass. σ is the total stress tensor and is defined, in the case of a Newtonian fluid, as:

$$\sigma = -p\mathbf{I} + 2\mu\mathbf{D} + \frac{2}{3}\mu\operatorname{div}(\mathbf{v}), \tag{2.2}$$

where p is the fluid pressure, μ is the dynamic viscosity and $\mathbf{D}$ is the strain rate tensor, equal to:

$$\mathbf{D} = \frac{1}{2}\left(\nabla\mathbf{v} + \nabla^T\mathbf{v}\right). \tag{2.3}$$

For the sake of simplicity in this work incompressible fluids are treated only. Therefore, equations (2.1) can be significantly simplified, obtaining the usual *Navier–Stokes* equations:

$$\begin{cases} \dfrac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla \mathbf{v}) = \mathbf{f} + \nu \Delta \mathbf{v} - \nabla \tilde{p} \\ \\ \operatorname{div}(\mathbf{v}) = 0 \end{cases}, \tag{2.4}$$

where ν is the kinematic viscosity, defined as $\nu = \frac{\mu}{\rho}$. $\tilde{p}$ is obtained as $\tilde{p} = \frac{p}{\rho} + gh$, including in the pressure term the gravity effect too.

In order to solve these equations, OpenFOAM follows an approach based on projection methods. In particular, it solves them in two steps: initially, a velocity field is built such that it satisfies the momentum equation but without the pressure gradient term; finally, the field is corrected in order to satisfy the continuity equation, by means of the solution of an appropriate pressure equation.

Carrying out the divergence to the momentum equation and simplifying it with the continuity equation, the following Poisson problem for the pressure is obtained:

$$\operatorname{div}(\nabla p) = -\operatorname{div}\left[\operatorname{div}(\rho \mathbf{v}\mathbf{v} - \mathbf{S}) - \rho \mathbf{b} + \frac{\partial(\rho \mathbf{v})}{\partial t}\right], \tag{2.5}$$

where $\mathbf{S}$ is the viscous part of the stress tensor: $\mathbf{S} = \sigma + p\mathbf{I}$.

In our cases, both density and viscosity are considered to be constant, so the equation can be further simplified (see [13]):

$$\Delta p = -\frac{\partial}{\partial x_i}\left[\frac{\partial(\rho v_i v_j)}{\partial x_j}\right]. \tag{2.6}$$

This is the equation to solve in order to guarantee the calculated velocity field to be solenoidal.

In the implementation of these equations it is important to cope also with the motion of the fluid mesh, which can undergo substantial deformations and thus the overall domain and control volumes can change. Therefore, in the standard fluid dynamic equations the terms have to be changed in order to take into account for the fluxes caused by border motion: it is necessary to employ a *Arbitrary Lagrangian–Eulerian* approach. For further details see [20].

2.2 Finite volume method

The vast majority of commercial CFD codes employs finite volume methods to discretize the fluid equations. OpenFOAM follows this approach too in its principal solvers.

For a brief introduction to finite volume methods, let us consider a simple scalar conservation equation:

$$\frac{\partial w}{\partial t} + \operatorname{div}\left(\mathbf{F}\left(w\right)\right) = s\left(w\right), \qquad x \in \Omega,\ t > 0, \tag{2.7}$$

with proper boundary and initial conditions. w represents the unknown scalar, $\mathbf{F}$ is the flux function and s is the source term.

The domain is divided into a finite number of polyhedral control volumes and the equations are integrated on each of them. Thanks to the divergence theorem, the second term can be turned into a surface integral, which represents the flux of the modeled quantity across cell faces:

$$\frac{\partial}{\partial t}\int_{V_P} w\ dV + \int_{\partial V_P} \mathbf{F}\cdot\mathbf{n}\ d\sigma = \int_{V_P} s\ dV. \tag{2.8}$$

The mean value w_P over the cell P is the main unknown value of finite volume methods:

$$w_P = \frac{1}{|V_P|}\int_{V_P} w\ dV. \tag{2.9}$$

Calling N_P the set of cells which share one face with V_P, it is possible to write the conservation equation in the cell P in this way:

$$|V_P|\frac{dw_P}{dt} + \sum_{s\in N_P}\int_{V_P\cap V_s} \mathbf{F}\cdot\mathbf{n}\ d\sigma = \frac{1}{|V_P|}\int_{V_P} s\ dV. \tag{2.10}$$

Integrals can then be calculated adopting appropriate quadrature rules.

Following the notation in [13], in a finite volume discretization of the Navier–Stokes equations, for each cell P the momentum conservation equation can be written in this way (assuming an implicit time-advancing scheme):

$$a_P^{v_i} v_{i,P}^{n+1} + \sum_l a_l^{v_i} v_{i,l}^{n+1} = Q_{v_i}^{n+1} - \left(\frac{\delta p^{n+1}}{\delta x_i}\right)_P, \tag{2.11}$$

where l is an index referring to neighbor nodes taking part to the momentum equation for the cell P and the matrix $\mathbf{a}$ holds all coefficients. Q

contains all terms that can be explicitly calculated using v_i^n and the forces, or other terms, that depend on v_i^{n+1}. This notation will be used again in the brief introduction to the scheme implemented in OpenFOAM for solving the velocity-pressure coupled problem.

Meshes used in the cases analyzed in this work are composed principally by hexahedral and tetrahedral cells. OpenFOAM has been developed with the precise intent of managing unstructured mesh. Therefore, it uses collocated grids, putting nodes at the center of each cell. In order to guarantee stability, values on faces are obtained employing a method that resembles the *Rhie–Chow* interpolation. For further information see [17].

2.3 Solution of the pressure problem

OpenFOAM uses iterative projection methods to solve the pressure equation. In particular, it employes the *SIMPLE* algorithm (acronym for *Semi-Implicit Method for Pressure-Linked Equations*), the *PISO* algorithm (acronym for *Pressure Implicit with Splitting of Operators*) and the *PIMPLE* algorithm (developed specifically for OpenFOAM and result of the combination of the other two schemes).
They are all quite similar. In all three approaches, velocity and pressure correction terms are introduced:

$$\begin{aligned} v_i^m &= v_i^{m^*} + v_i', \qquad i = 1, 2, 3 \\ p^m &= p^{m-1} + p', \end{aligned} \tag{2.12}$$

where m is an index of the sub-iterations inside time-step t^n.
The common procedure is then divided into the following substeps, executed iteratively at each time-step, until convergence is reached:

1. as first approximation of v_i^{n+1} and p^{n+1} the velocity and pressure values obtained in the previous time-step are used.

2. the intermediate velocity field is then calculated solving the linearized momentum conservation equation:

$$v_{i,P}^{m^*} = \tilde{v}_{i,P}^{m^*} - \frac{1}{a_P^{v_i}} \left(\frac{\delta p^{m-1}}{\delta x_i} \right)_P, \tag{2.13}$$

where

$$\tilde{v}_{i,P}^{m^*} = \frac{Q_{v_i}^{m-1} - \sum_l a_l^{v_i} v_{i,l}^{m^*}}{a_P^{v_i}}. \tag{2.14}$$

3. the pressure corrective equation is solved:

$$\frac{\delta}{\delta x_i}\left[\frac{\rho}{a_P^{v_i}}\left(\frac{\delta p'}{\delta x_i}\right)\right]_P = \left[\frac{\delta\left(\rho v_i^{m^*}\right)}{\delta x_i}\right]_P + \left[\frac{\delta\left(\rho \tilde{v}_i'\right)}{\delta x_i}\right]_P. \tag{2.15}$$

The term $\tilde{v}_i'$ is neglected for computational reasons. In order to cope with this approximation, in the PISO algorithm a second correction is used. In particular, after the first pressure correction, velocities are updated:

$$\begin{aligned} v_{i,P}' &= -\frac{1}{a_P^{v_i}}\left(\frac{\delta p'}{\delta x_i}\right)_P, \\ \tilde{v}_{i,P}' &= -\frac{\sum_l a_l^{v_i} v_{i,l}'}{a_P^{v_i}}. \end{aligned} \tag{2.16}$$

At this point, a second pressure problem is solved:

$$\frac{\delta}{\delta x_i}\left[\frac{\rho}{a_P^{v_i}}\left(\frac{\delta p''}{\delta x_i}\right)\right]_P = \left[\frac{\delta\left(\rho \tilde{v}_i'\right)}{\delta x_i}\right]_P. \tag{2.17}$$

This step can be executed more than once, for a number of times prescribed by the user. The term $\rho v_i^{m^*}$ is corrected every time, in order to cope with possible mesh non-orthogonalities.

4. the pressure field is then updated:

$$p^m = p^{m-1} + c_{ur} p', \tag{2.18}$$

with $0 < c_{ur} < 1$ being an under-relaxation factor, needed in the SIMPLE algorithm to reestablish the convergence speed, lost because of the approximation made neglecting the term $\tilde{v}_i'$ in the previous step. In the PISO algorithm, instead, the under-relaxation is not necessary, thanks to the second pressure corrective pass, which helps maintaining the rate of convergence.

5. the velocity field is finally updated:

$$v_{i,P}^m = v_{i,P}^{m^*} + v_P', \tag{2.19}$$

where

$$v'_{i,P} = -\frac{1}{a_P^{v_i}} \left(\frac{\delta p'}{\delta x_i} \right)_P , \tag{2.20}$$

or, as far as the PISO algorithm is concerned:

$$v^m_{i,P} = v^{m^*}_{i,P} + v''_P, \tag{2.21}$$

where

$$v''_{i,P} = \tilde{v}'_{i,P} - \frac{1}{a_P^{v_i}} \left(\frac{\delta p''}{\delta x_i} \right)_P . \tag{2.22}$$

Now the velocity field satisfies the continuity equation.

6. boundary conditions are updated, using the corrections introduced by the pressure problems.

7. finally, this procedure is iterated, using v_i^m and p^m as always better approximations of v_i^{n+1} and p^{n+1}, until corrections made are negligible and below a certain prefixed threshold.
   The process then continues with the next time-step.

Chapter 3

Fluid-structure interaction

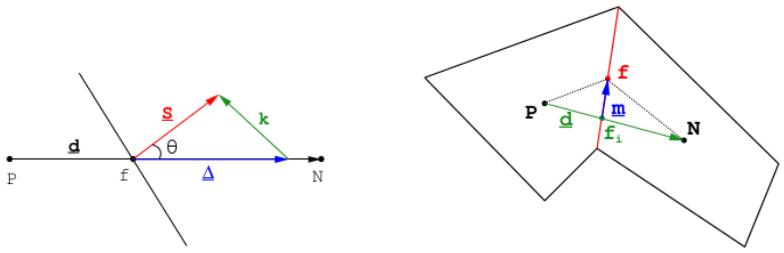
The structural and fluid dynamics problem have been introduced in the previous chapters. In this chapter, the remaining pieces necessary to build a simulation with fluid-structure interaction are presented. In section 3.1 the mesh-motion problem is described. Interpolation algorithms used to move mesh internal points or transfer quantities at the interface between the grids are discussed in section 3.2 and 3.3. Finally, in section 3.4 the coupling procedure is briefly presented.

3.1 Mesh-motion algorithms

When the structural problem is solved, a field containing displacements of all solid mesh points is obtained. These displacements actually deforms the fluid domain, imposing movements on a set of boundaries. As a consequence, the fluid mesh has to adapt: the internal points have to be repositioned in order to cope with the new boundaries. Several algorithms can perform this kind of interpolation. A good algorithm has to be quick (since that, with refined meshes, the number of internal points to move can be extremely high) and, even more important, has to preserve the quality of the mesh.

There are various indexes that permit a quantitative analysis of mesh quality. However, it is important to note that, if a control volume is different from the ideal isotropic cell, it does not mean always bad quality of the mesh. For example, the cell *aspect ratio* is the ratio between the longest edge and the shortest one; in certain situations, a high value of this indicator is

preferred, especially when cells are explicitly aligned with the flux.



(a) Non-orthogonality (b) Skewness

Figure 3.1: Sketch of the main cell quality measures.

Two indicators often used in OpenFOAM to check mesh quality are the *non-orthogonality* index and the *skewness*. The first indicator measures the angle between the segment which links two cell centers and the normal of the shared face. In figure 3.1a, P and N are the two cell centers and f is the common face, with normal S: the non-orthogonality value is the angle θ between $\overline{PN}$ and S. Best value for this indicator is zero. If θ is above 70° performances of solvers are reduced and accuracy is lowered. In extreme cases, convergence is not guaranteed.
The *skewness* measure is sketched in figure 3.1b. If f_i is the intersection between the line which links the cell centers P and N and the shared face, the distance between f_i and the face center f is the *skewness* measure. A high value of this index causes a reduction in the face integration order and, therefore, means that appropriate high order schemes for gradient calculation are required; stability is instead not undermined.

In this work two methods will be introduced: *Radial Basis Functions* interpolation and *Inverse Distance Weighting* interpolation. They are generic interpolation methods and will be used also outside the mesh-motion context, in particular to interpolate quantities between different grids (from the solid mesh to the fluid mesh and viceversa).

3.2 Radial Basis Functions interpolation

In [3], the author claims that good smoothness in internal point positioning can be achieved formulating the problem in pure algebraic terms. The method so obtained is quicker and more robust if compared to other methods based on partial differential equations, like, for example, the solution of a Laplace problem for the internal points motion [16].

Interpolation with *Radial Basis Functions* (RBF) employs a moderate number of control points to accomplish a global interpolation of all internal mesh points. For each component, the interpolation formula is the following:

$$s(x) = \sum_{j=1}^{N_b} \alpha_j \phi(|\mathbf{x} - \mathbf{x}_{b,j}|) + q(\mathbf{x}), \tag{3.1}$$

where $\mathbf{x}_b$ contains positions of the N_b control points, ϕ is the basis function, which depends on the distance between the considered point and the control points, and $q(\mathbf{x})$ is a polynomial, usually linear, which depends on the choice of basis function and on the coefficients α_j.
In order to guarantee interpolation consistency, it is required that all polynomials of order lower than q have zeros in correspondence with the control points:

$$\sum_{j=1}^{N_b} \alpha_j p(\mathbf{x}_{b,j}) = 0. \tag{3.2}$$

Choosing a linear polynomial, it holds that:

$$q(\mathbf{x}) = \beta_0 + \beta_1 x + \beta_2 y + \beta_3 z. \tag{3.3}$$

Interpolation coefficients (contained in vectors α and β) are solution of the following system:

$$\begin{bmatrix} s(\mathbf{x}_{b,j}) \\ 0 \end{bmatrix} = \begin{bmatrix} \Phi_{bb} & Q_b \\ Q_b^T & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \tag{3.4}$$

where $s(\mathbf{x}_{b,j})$ is the interpolated function value in the control points and matrix Φ_{bb} contains values produced by the basis function evaluated using the distance between pairs of control points. Each element is defined as:

$$\Phi_{(i,j)} = \phi(|\mathbf{x}_{b,i} - \mathbf{x}_{b,j}|). \tag{3.5}$$

Q_b is instead a rectangular matrix whose rows are $\begin{bmatrix} 1 & \mathbf{x}_b \end{bmatrix}$.
The obtained system to solve is dense and needs a factorization method in order to be solved. In this implementation, LU factorization is adopted, already available in OpenFOAM.

Choice about the RBF function to use is well discussed in [3] and [20]. Generally, there are locally supported functions, which involves points within a certain range r and permits to have a system simpler to solve but with a loss in accuracy, and globally supported functions, which instead needs a smoothing function in order to facilitate the calculus.

The most computationally heavy operation is the inversion of the interpolation matrix. After this step, all other internal points motion is calculated by means of formula (3.1). This approach is quite effective when control points are few but significant and the dense matrix to invert is small. It has proved to be robust and to produce high quality meshes, especially when handling rigid motion-like boundary movements. When contour motion is more complicated, however, sometimes its parameters are difficult to tune and it produces worse meshes.

3.3 Inverse Distance Weighting interpolation

In *Inverse Distance Weighting* (IDW) interpolation [36], an amount of N_b control points (on the mesh boundaries) are given, with their displacement $\mathbf{x}_{b,j}$. Other internal points position is then calculated using the following interpolation formula:

$$w(\mathbf{x}) = \frac{\sum_{j=1}^{n} \mathbf{x}_{b,j} \phi(r_j)}{\sum_{j=1}^{n} \phi(r_j)}, \tag{3.6}$$

where $r_j = |\mathbf{x} - \mathbf{x}_{b,j}|$ is the distance between $\mathbf{x}$ and the control point $\mathbf{x}_{b,j}$ and ϕ is the *weighting function*, defined as:

$$\phi(r) = r^{-c}, \tag{3.7}$$

where c is a parameter adequately chosen and tuned.
Internal points positions on the non-deformed mesh are known; therefore, the N_axN_b interpolation matrix H can be precomputed, where N_a is the overall number of internal points. It is such that, if multiplied by the control points displacement vector, it returns a vector with the displacements of all internal points. It is assembled checking, for each internal point i, which control points actually influence its position, using a vicinity check based on a prefixed cut-off range. At this point, the i-th row is assembled with the contributions given from each chosen control point to the final movement of point i, calculated using the interpolation formula.
Thanks to the cut-off check, for each internal point only a few control points contribute to its movement. Therefore, matrix H is sparse and without any recognizable pattern, since meshes are generally unstructured. Moreover, it can be really big, depending directly on the number of internal points, and this can cause memory issues if not handled appropriately.

3.4 Fluid-structure coupling

In fluid-structure interaction problems, the fluid and structural models are coupled, with the deformable structure interacting with the fluid flow. The fluid and structural problems have been separately described in last chapters; in the previous sections the mesh-motion and interpolation processes have been discussed too. All what remains is to link these pieces together by means of a coupling algorithm.

There are different techniques to solve this kind of coupled problem. With a *monolithic* approach [11], a unique system is solved, where all degrees of freedom are treated together. *Partitioned* or *segregated* FSI schemes [29, 12, 20] instead decompose the domain of computation in different parts, usually separating the fluid problem from the structural one, with appropriate handling of the interface. In this way, different existing codes, using even different kinds of discretization and optimized for one particular problem, can be coupled together. Information between them is exchanged only at domain boundaries, usually by means of interpolation schemes.

In this work a *strongly coupled partitioned approach* is used, based on a Dirichlet-Neumann interface coupling formulation (for details see [20]).

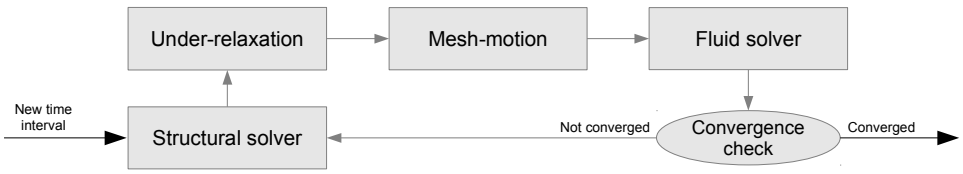


Figure 3.2: FSI coupling algorithm flow chart.

A flow chart of the coupling algorithm is sketched in figure 3.2. During each time-step the three problems are solved separately in sequence for a certain number of sub-iterations, until a convergence criteria is met. Structure displacements resulting from the structural solver are not treated directly but instead they are relaxed using the adaptive *Aitken* relaxation technique [10, 31], in order to reach convergence in fewer sub-iterations:

$$\tilde{\mathbf{u}}_{s,k+1}^{n+1} = (1 - \alpha_{k+1})\, \mathbf{u}_{s,k+1}^{n+1} + \alpha_{k+1} \tilde{\mathbf{u}}_{s,k}^{n+1}, \tag{3.8}$$

where α_k is the adaptive relaxation coefficient, set to a prefixed value in the

first sub-iteration and updated in each next sub-iteration:

$$\alpha_{k+1} = \alpha_k + (1 - \alpha_k) \frac{(\Delta \mathbf{u}_{s,k+1} - \Delta \mathbf{u}_{s,k}) \cdot \Delta \mathbf{u}_{s,k+1}}{(\Delta \mathbf{u}_{s,k+1} - \Delta \mathbf{u}_{s,k})^2}, \tag{3.9}$$

where $\Delta \mathbf{u}_{s,k} = \mathbf{u}_{s,k}^{n+1} - \tilde{\mathbf{u}}_{s,k-1}^{n+1}$ and $\Delta \mathbf{u}_{s,k-1} = \mathbf{u}_{s,k-1}^{n+1} - \tilde{\mathbf{u}}_{s,k-2}^{n+1}$.
Convergence criteria may vary and depend on the problems. Generally, the absolute or relative difference of velocity, pressure or structural displacement fields between two consecutive sub-iterations is checked against a threshold value, below whom the time-step is considered to have converged:

$$\begin{cases} \left\| \mathbf{u}_{f,k+1}^{n+1} - \mathbf{u}_{f,k}^{n+1} \right\| \leq Th_u \\ \left\| p_{f,k+1}^{n+1} - p_{f,k}^{n+1} \right\| \leq Th_p \\ \left\| \mathbf{u}_{s,k+1}^{n+1} - \mathbf{u}_{s,k}^{n+1} \right\| \leq Th_d \end{cases} \quad \text{or} \quad \begin{cases} \left\| \mathbf{u}_{f,k+1}^{n+1} - \mathbf{u}_{f,k}^{n+1} \right\| / \left\| \mathbf{u}_{f,k}^{n+1} \right\| \leq Th_u \\ \left\| p_{f,k+1}^{n+1} - p_{f,k}^{n+1} \right\| / \left\| p_{f,k}^{n+1} \right\| \leq Th_p \\ \left\| \mathbf{u}_{s,k+1}^{n+1} - \mathbf{u}_{s,k}^{n+1} \right\| / \left\| \mathbf{u}_{s,k}^{n+1} \right\| \leq Th_d \end{cases}$$

with appropriate norms. In this work, both L^2 and L^∞ norms are used, depending on the case analyzed.

Computational grids of fluid and structure may not be conforming at the interface. If they are not, or if the two problems are discretized in different ways, it is necessary to employ interpolation algorithms to transfer information across the interface. In this work, fluid equations are solved with finite volume methods, while structural equations uses a finite element discretization. Moreover, the fluid problem has its degrees of freedom in cell centers and imposes boundary conditions on face centers, while the structural problem defines displacements on the nodes. There are two different issues: transfer the structural displacements from the structural nodes to the fluid domain and transfer normal stresses from the fluid mesh to the structural domain.

Once structural displacements have been computed by the structural solver, it is necessary to:

1. obtain displacements of fluid mesh boundary points, in order to carry on the mesh-motion procedure.

2. obtain velocities in fluid mesh boundary face centers, as boundary conditions for the fluid equations.

An interpolation scheme can be employed to transfer structural displace-

ments from structural points to fluid points. However, the same interpolation scheme cannot be used to interpolate velocities from structural points to fluid face centers also, because the *geometric conservation law* would not be respected (see [19]) and the fluid problem would suffer from additional incorrect mass sources. In this work, displacements are not interpolated directly on fluid mesh nodes, but instead on a triangular subdivision of the boundary faces. The area-weighted average of those values is then computed in order to obtain velocities and continue with the mesh-motion procedure.

On the other hand, Neumann condition on normal stresses has to be imposed in order to solve the structural problem. In the weak formulation, such condition consists in a surface integral term in the right-hand-side. Therefore, the interpolation procedure must be able to transfer information from fluid face centers to structural quadrature nodes. It is also possible to interpolate values directly on structural nodes, calculating them on quadrature nodes using the structural element's shape functions, although losing some accuracy. One even more inaccurate solution could also be to use directly the fluid pressure field interpolated on the structural mesh (in element centers), thus having for each element a constant external pressure value.

One important fact to consider is that the condition of equivalence of normal stresses, imposed weakly at the boundary, is, in fact, equivalent to enforce the principle of virtual works, for any possible displacement. Therefore, the interpolation scheme employed to transfer information across the interface must be able to ensure energy conservation. This has proven to be very important in order to guarantee stability of the FSI coupling [29].

In this work, interpolation based on radial basis functions (introduced in section 3.2) has been adopted. It can be formulated so that it preserves energy conservation at the interface. For further information and details about parameter tuning see [20].

Part II
Implementation

In this second part, all what concerns the implementation of structural and fluid-structure interaction solvers is discussed, with particular attention to the developed algorithms and optimizations made. At the end of each chapter, results of numerical tests and performance benchmarks are presented.

Chapter 4

GPU parallelization

The *GPU*, acronym for *Graphics Processing Unit*, is the main processor inside graphics boards. It is responsible for all dedicated graphics calculations and operations.

The idea of using the video board and its power for purpose other than video-games and CAD design is not recent. The inherently parallel vectorized architecture of the GPU has always been exploited to perform the most various calculations, but it has always been very hard to program. In the past, it was designed to be an hardware accelerator for 2D and 3D graphics, with a fixed but very efficient pipeline structure. It posed a lot of constraints to the general purpose programmer: doing calculus that was not graphics related was difficult and a very deep understanding of the underlying hardware and its functionalities was needed.
When the first forms of programmable graphics hardware were introduced by the major video hardware companies, which embedded features like *texture combiners* or *shaders*, GPUs became much more powerful and user friendly. But it is only recently that, with the introduction of unified architectures and specialized languages such as *CUDA* or *OpenCL*, GPU general purpose programming has gained popularity and has become available to everybody who owns a decent video board.

Because of their history as gaming devices, for their internal structure GPUs are specialized in vectorized and parallel calculus. They come with a large amount of extremely quick floating point arithmetic units, a feature that makes them attractive to scientific programmers: high level of *FLOPS* (*Floating-Point Operations Per Second*) can be achieved, with an hardware

relatively cheap and not power consuming. In comparison, graphics processors dedicate the majority of transistors to perform calculations, while CPUs employ them in cache and flow control. This implies, as far as the graphics hardware is concerned, a less flexible programmability, but also an higher amount of computational power. For example, the *NVIDIA GeForce GTX 480* GPU has a theoretical peak of 1344.96 GFLOPS in single precision; the more recent *NVIDIA GeForce GTX 690* reaches even the theoretical peak of 5621 GFLOPS: a mini-cluster inside the computer tower.[1]

There are different available frameworks dedicated to GPU general purpose programming. Some of them are proprietary, directly developed by video hardware companies and obviously compatible with their GPUs only: for example *ATI Stream* and *AMD App*, developed by *AMD/ATI*, or *CUDA*, owned by *NVIDIA*.
There are also more generic development environments for which compatibility is the main goal. Examples are *OpenCL* from the *Khronos Group* and *DirectCompute* from *Microsoft*.

As far as this work is concerned, the choice of which framework to use has been made after some tests with two different kinds of environments: CUDA and OpenCL. The former one is proprietary and runs only on NVIDIA GPUs, while the latter is theoretically compatible with different kinds of graphics hardware and can run on some CPUs also. We finally have opted for CUDA mainly because of practical reasons: we believe that NVIDIA's general purpose GPU programming solutions are more mature and the rich availability of libraries, debugging and profiling tools makes it a powerful development environment. Furthermore, there have been some problems programming in OpenCL mainly related to poorly-coded and bugged device drivers developed by video board companies. It is nevertheless easy to port CUDA code to OpenCL, since they are based on very similar architectures and code structure is basically the same. Thus this choice of language is not as restrictive as it maybe seem.

[1] `http://en.wikipedia.org/wiki/GeForce_600_Series`

4.1 CUDA

CUDA, acronym for *Compute Unified Device Architecture*, is the main GPU hardware computational platform developed by NVIDIA. There are different versions of the CUDA APIs (*Application Programming Interfaces*) and GPU architectures (which can be referred using the *compute capability* index); some functions or features have been introduced from a certain architecture onward, or have a different behavior depending on compute capability. Therefore, each different architecture has its own optimization strategies to be exploited. All CUDA code developed for this thesis is optimized for and meant to be run on *Fermi* GPUs (starting from compute capability 2.0) or newer. For more details about the Fermi compute architecture see [26].

A generic CUDA application is mainly divided into two parts: host code, which runs on CPU, and device code, written in CUDA language, which runs on the GPU in a parallel fashion. The host code is responsible to initialize the GPU devices, to manage memory allocations and to transfer and launch *kernels*. A kernel is a sequence of *SPMD* operations (from *Single Program Multiple Data*) executed on an arbitrarily high number of parallel threads. A *program* is a bunch of compiled GPU code: it contains different kernels, callable directly from host code, and other functions, visible only from device code and kernels. Each thread executes exactly the same set of instructions. They are organized by the programmer in a hierarchical structure, a n-dimensional grid made of n-dimensional blocks of threads (where n can be one, two or three, example with $n = 2$ is in figure 4.1). In order to develop an efficient code, thread topology is important and must be designed accurately: it should somehow reflect the intrinsic structure of the problem to be solved.

A GPU contains a certain number of *Streaming Multiprocessors* (SM), which execute one or more thread blocks each, based on the amount of resources available and resources requested per-block. Threads are executed by the SM in groups of 32, called *warps* (a group of 16 threads is called *half-warp*), with virtually zero-overhead warp scheduling. All threads in a warp execute an instruction at a time. Thread scheduling is completely handled by the hardware, without any help from the software and transparently to the user. In order to maintain high performances, it is thus very important to

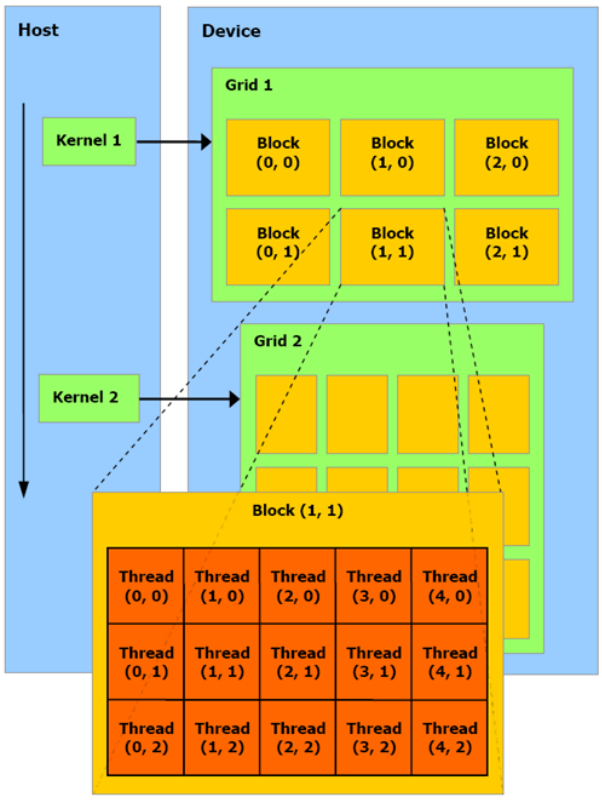


Figure 4.1: Example of two-dimensional CUDA thread hierarchy.

ensure that threads follow the same execution path, avoiding when possible conditional jumps. Execution divergence is automatically handled by the hardware, but it leads to overhead and should indeed be avoided. There are different kinds of memory which can be used by CUDA applications (a sketch of them is visible in figure 4.2):

- *Global memory*: it is the biggest off-chip memory available and its size is in the order of gigabytes. It is visible from host source code also. Bandwidth from and to host memory can be high, up to hundreds of GB/s for modern GPUs, thanks to specialized DMA transfer engines. Latency, however, is really high. Therefore, accesses to global memory should be handled with care: they can easily become the bottleneck of memory-intensive kernels. In modern GPUs (compute capability 2.x at least) there are also two levels of cache (L1 and L2), made of 128-byte cache lines. Memory accesses cached in L1 are serviced with 128-byte memory transactions, while memory accesses cached in L2

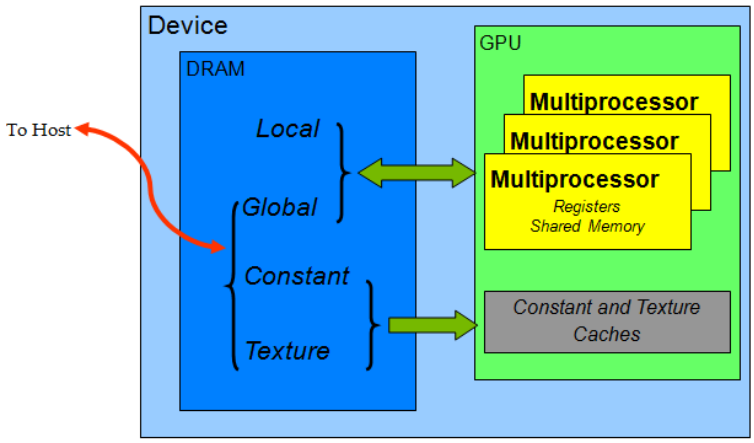


Figure 4.2: CUDA memory spaces.

only are serviced with 32-byte memory transactions. They can be enabled or disabled by the programmer and if used correctly they can result in a significant source of optimization.

- *Constant memory*: this kind of memory is currently 64 KB big in all devices and it is meant to contain constant values. It is optimized for *broadcast* memory accesses (i.e. threads of the same half-warp, or warp if compute capability 2.x or upper, access the same address), while it can drastically reduce performance when threads of the same half-warp perform transactions on different addresses at the same time. Constant memory has also an associated *constant cache*, which stores recently accessed values in order to speed up further requests to them.

- *Texture memory*: this read-only memory is useful primarily for graphics applications, since its *texture cache* is highly optimized for accesses with spatial locality (in texture coordinates) and not memory locality (as the other kinds of cache). It is also equipped with dedicated fast hardware filters (mainly linear interpolation filters) to perform texture fetches, which in certain situations can become handy and save instruction cycles.

- *Shared memory*: it is a low-latency (roughly 100x lower than global memory) on-chip memory, usually 48 KB maximum per SM. Each thread block has its own portion of shared memory, not visible from threads of other blocks. It is used primarily for inter-block commu-

nications, data reuse between threads and to organize and facilitate coalescing of reads and writes to global memory. It is not accessible outside a kernel and it is not persistent over kernel calls. It is divided into 16 (or 32 on modern GPUs with compute capability of 2.x or upper) interleaved banks of 32 bits each, which can service only one request at a time. If the same bank is accessed by more threads (but not all of them) of an half-warp (or a warp, for compute capability of 2.x or upper) at the same time it causes a *bank conflict*, which means that accesses are serialized and thus performances are reduced.

- *Registers*: they are the fastest on-chip memory areas available, with local thread scope. There is a certain amount of registers available in each SM (from 8 K up to 64 K 32-bit registers, it depends on compute capability) and they are partitioned among concurrent threads.

- *Local memory*: it is a portion of off-chip global memory with scope local to the thread. Accessing to it is as expensive as accessing to global memory. It is used when there is insufficient register space to hold local variables (*register spilling*) or if there are local arrays with indexing not resolvable at compile time. Therefore, the amount of thread variables can lead to significant performance loss if too high or not managed properly.

It is important to observe that resources of SMs are allocated for a thread block as long as that block is active (i.e. there is at least one thread which has not completed). Therefore, number of registers per thread, shared memory usage and thread block size influence the *occupancy* of SMs, the ratio between the actual number of active warps and the maximum manageable by the SM. A good level of occupancy is necessary in order to hide memory latency: when a warp is waiting for a memory request to complete, the scheduler is able to process other warps. Nevertheless, this does not mean that high occupancy is always beneficial (see [35]).

One of the most important performance consideration to follow is the coalescing of global memory accesses: read or write requests issued by treads in a warp (or half-warp if compute capability 1.x) can be coalesced into a single transfer if certain access pattern and address alignment criteria are met. They depend highly on the compute architecture (see [24, 25] for a detailed description). Global memory loads and stores are the bottleneck of

most algorithms, so it is vital to ensure coalesced accesses wherever possible in order to achieve the maximum bandwidth; shared memory is often useful for this scope, since it permits to temporarily store values randomly accessed by threads, organizing them for coalesced read and write transactions.

Memory transfers from global host memory to device memory (and vice-versa) constitute another common bottleneck in GPU algorithms. It is better to keep data in GPU memory avoiding transfers wherever possible. In recent CUDA releases it is possible to execute asynchronous memory transfers while a kernel is in execution. This is a powerful feature, but requires a good algorithm design to be exploited successfully.

An example of NVIDIA GPU with Fermi architecture is the *GeForce GTX 470*, used in most of the test cases for this work. It is equipped with 448 CUDA cores and 1280 MB of GDDR5 video RAM, with a bandwidth up to 133.9 GB/s. It contains third-generation SMs with 32 CUDA cores each and 64 KB of local low-latency memory. Each of these processors supports up to 1024 threads and 32 768 32-bit registers.

One important underlying hypothesis assumed in this work is that every structure that belongs to video memory is completely contained in it and does not need any form of swapping or streaming from host memory. This means that the entire problem must fit the available video memory and thus it is limited by its size. This should not be a problem: memory consumption of most of the algorithms described in this work when applied to problems of standard size is quite low and fits perfectly a common video board.
Extending the algorithms to deal with bigger problems should not be difficult but would require an additional splitting of the problem, with all the necessary scheduling and communications between parts, and good memory management. This would also probably break some of the optimizations currently exploited to gain performance.

Chapter 5

Structural solver

In this chapter the developed structural solver is discussed, with details about the implementation process. Two different GPU implementations are considered. The first one is based directly on the CPU algorithm, which is analyzed and divided into sub-operations, assigned to one GPU kernel each. The second implementation follows a different approach: the time advancement procedure is observed from the element point of view and implemented in a single big GPU kernel, maximizing computational efficiency while minimizing communications. Memory and code optimizations are also discussed. Results and performances of the structural solver are then checked against Abaqus in two structural test cases.

5.1 Development framework

The main objective of this work is building from scratch a finite element application to solve shell dynamics problems taking advantage of the available GPU computing power, in order to gain performance over standard CPU-based solvers. It has been developed to be modular and versatile: its aim is to be linked with other applications, for example a CFD solver, in order to easily couple the structural problem in a multi-physics framework. It will in fact be used as structural solver for fluid-structure interaction problems, as described in chapter 7.

The solver environment consists of several libraries and applications. It is a relatively small but complete package for finite element analysis: other than the solver itself, several pre-processing and post-processing utilities

have been developed, in order to facilitate its usage and result validation. It is completely written in *C++* and CUDA.

The main library where shell solver source code and CUDA kernels reside is called *Strusol* (a not original name deriving from *Shell Structural Solver*). It contains a set of classes that help the user to set up simulations.
Other functions and classes, not related exclusively to shell problem solving and finite element analysis but more general purpose, are packed in the *Utils* library. They, for instance, help logging and managing errors, parsing text files and scripts, managing file system operations in a operating system independent manner. Functions that aid in debugging are also included. Finally this library contains a set of math classes and structures that are used by the solvers.
A little mesh viewer is available in the *Viewer* library. It is extremely simple and it is not meant to be a fully featured post-processor. Nevertheless, it is an extremely useful debugging tool: it shows the actual mesh as it is being deformed in real-time, while the simulation is running. It uses *OpenGL* as rendering platform and *GLUT* for OpenGL context management.
The principal stand-alone interface to Strusol via command-line interface is called *StrusolCli*. It is mainly a command parser: using external scripts or direct commands it can load meshes, set up problems, run simulations and export results in various formats. It can be linked with the Viewer library to permit real-time rendering of results as they are computed by Strusol.
Finally, other useful source code, test cases and post-processing applications are collected in the *UtilsBin* package, completing the framework in order for it to be an easy to use and portable environment.

The structural solver SEDIS, developed at the department of Structural Engineering of Politecnico di Milano [5, 14, 9], was taken as a base for the development process of Strusol. SEDIS solves the shell dynamics problems introduced in chapter 1, using MITC4 elements. It consists on a library with an user front-end and a callable interface. It is completely written in FORTRAN and features both an explicit dynamics solver and an *Adaptive Dynamic Relaxation* solver [27] for steady cases. It has been written and debugged over the years and it is considered to be mature and stable. For this reason, in the process of development, both SEDIS and the commercial solver Abaqus have been extensively used as references for result validation. The version of SEDIS used throughout this work is able to work on multiple

threads on the same computational node thanks to an OpenMP parallelization (although an MPI implementation of an older version of the codebase was discussed in [9]).

Strusol has been built from scratch for one main reason: to be completely free from constraints of an existing design and to be organized and prepared for execution on GPU hardware from the beginning of the development. It is quite always possible to directly port a CPU application to GPU with some clever tricks; but it is only designing the algorithms from scratch, with the clear purpose of running on GPU in mind, that the best performances can be achieved.

The main Strusol library is not meant to be runnable on its own: it just exports a set of functions and classes to the user and it is up to him to program and build the final executable that will be linked against the library.

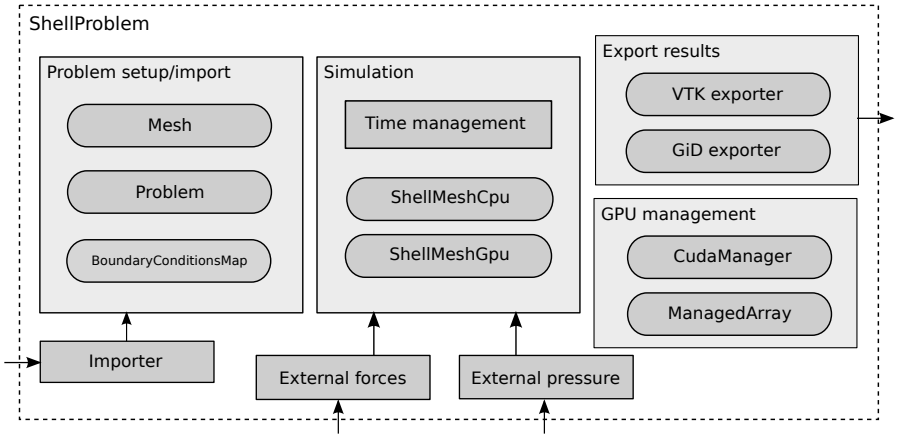


Figure 5.1: Simplified sketch of *ShellProblem* class internal structure.

There are different categories of classes exported by Strusol. The first group of classes deals with problem loading and pre-processing. A generic *Mesh* class handles the polygonal geometries with their nodes, elements and sets. Meshes and problem settings are set up manually or they can be directly imported from SEDIS and Abaqus job files: an internal parser understands a subset of Abaqus keywords and permits the user to setup problems in Abaqus, exploiting its user interface, and importing and solving them later with Strusol.

The second category of classes helps setting up the problem for its simulation. The *QuadratureRule* class calculates points and weights for standard Gaussian quadrature rules. The *Problem* class stores all materials with their mechanical properties and other global parameters, such as damping coefficients, gravity, global pressure and external loads. The *BoundaryConditionsMap* class helps setting boundary condition values on single nodes, node sets or boundaries.

A third group of classes handles the simulation of the shell problem. There are mainly two different solvers: a CPU only single-threaded version and a GPU optimized version, both with the same external interface. They take the mesh and problem settings stored in their respective classes, assemble the problem to be run and simulate it.

Another group of source code files helps with the post-processing of results. There is an exporter to an easily readable custom file format as well as exporters to *VTK* (viewable in *Paraview*) and *GiD* files, for a more accurate post-processing work.

The last group of classes and functions is the "glue" of all the other pieces. There are buffer managers, CUDA support classes and lot of debugging utilities.

Finally, the library contains also the *ShellProblem* class (figure 5.1, which offers all functionalities from one single place: it contains everything needed to setup problems, manage time and history output, export results and animations, all with one single interface. Users can choose to manage the problem themselves (manually creating the necessary classes) or let this class do it for them. A child class of *ShellProblem* is used to link Strusol with OpenFOAM for fluid-structure interaction problems.

5.2 CPU solver implementation

After the problem is created or loaded from file, the assembly process starts. Internal buffers are initialized with their initial values, nodal directors are calculated on the whole geometry, local nodal basis are built, lumped mass coefficients are extrapolated using Gaussian surface quadrature rules (since thickness is constant over an element by hypothesis) and finally damping coefficients and gravity force on each node are calculated. After this initialization step the problem is ready to be solved. Time is dis-

cretized into small steps of size Δt, which should satisfy stability conditions. One implemented method to determine a critical time-step size is to directly apply the *Courant-Friedrichs-Lewy* condition:

$$\Delta t_e^{crit} = \frac{l_e}{\sqrt{\frac{E}{\rho(1-\nu^2)}}}. \tag{5.1}$$

This condition infers that Δt should be less than the time taken by an elastic wave to travel a shell element. It depends on material properties (density ρ, Young coefficient E and Poisson ratio ν) and element characteristic length l_e, which derives from geometrical properties of the element itself, such as area, lengths of sides and diagonals. Since it depends on the single element properties, it is calculated on each element and then the minimum over the entire mesh is kept. At the end, Δt is set as:

$$\Delta t = \gamma \min_e \{\Delta t_e^{crit}\}, \tag{5.2}$$

where $\gamma \in (0, 1)$ is a safety parameter. This calculated step size can be very small and leads to a large number of (quick) iterations to be performed.

Starting at the beginning of a time step, the time advancement process can be summarized in these steps (they derive directly from the finite element formulation and time-integration scheme described in chapter 1, but are reported here for clarity):

- First integration half-step of the central finite difference time-advancing scheme is performed: starting from degree-of-freedom (from now on *dof*) values, velocities and accelerations of the previous time-step (u_i^n, v_i^n and a_i^n respectively), half step velocities ($v_i^{n+\frac{1}{2}}$) are calculated and dof values are updated. This calculation is carried on independently for every dof (denoted by the i subscript):

$$v_i^{n+\frac{1}{2}} = v_i^n + \frac{1}{2}\,\Delta t\, a_i^n, \tag{5.3}$$

$$u_i^{n+1} = u_i^n + \Delta t\, v^{n+\frac{1}{2}}. \tag{5.4}$$

- Based on the new rotational dof values, new directors are calculated in each node and local basis are rebuilt. The new director of node k

can be updated as follows:

$$\phi = \left(\alpha_k^{n+1} - \alpha_k^n\right) {}^t\mathbf{V}_1^k + \left(\beta_k^{n+1} - \beta_k^n\right) {}^t\mathbf{V}_2^k, \tag{5.5}$$

$${}^{t+\Delta t}\mathbf{V}_n^k = \begin{bmatrix} 1 & -\phi_3 & \phi_2 \\ \phi_3 & 1 & -\phi_1 \\ -\phi_2 & \phi_1 & 1 \end{bmatrix} {}^t\mathbf{V}_n^k, \tag{5.6}$$

with α_k^n and β_k^n being old rotational dof values and α_k^{n+1} and β_k^{n+1} the new ones. The director is then normalized and taken as reference to update local nodal basis also.

- Internal forces are calculated, based on new nodal displacements and directors. The calculation is carried on each element and resulting forces are accumulated on the nodes, in order to have an internal force value $f_{I,i}^{n+1}$ for each dof i.

- External nodal forces and external elemental pressures are collected, obtaining $f_{E,i}^{n+1}$ for each dof.

- The second integration half-step of the time-advancing scheme is performed. Accelerations are updated using new internal and external forces and velocity integration is concluded:

$$a_i^{n+1} = \frac{f_{E,i}^{n+1} - f_{I,i}^{n+1} - c_i\, v_i^{n+\frac{1}{2}}}{m_i + \frac{1}{2}\,\Delta t\, c_i}, \tag{5.7}$$

$$v_i^{n+1} = v_i^{n+\frac{1}{2}} + \frac{1}{2}\,\Delta t\, a_i^{n+1}, \tag{5.8}$$

where c_i and m_i are respectively the damping coefficient and mass value associated with the i-th dof.

Dirichlet boundary conditions are easily applied: in the integration half-steps, if a dof is fixed it does not get updated, but keeps its original value. In this way, this is the only part of code that needs to know about these boundary conditions, avoiding other conditional statements that would break execution flow path.

5.3 GPU solver implementation

Independently from the underlying algorithm, every GPU application follows these four essential steps:

- organize permanent data and constants in GPU memory;
- transfer input values to GPU;
- run one or more kernels on this data;
- transfer output values and results from GPU to host.

For simple problems these steps could be trivial and reflect the actual CPU implementation. For more complex tasks instead a good planning is necessary in order to get good performance results. As said before, particular care must be taken to organize memory and manage transfers. Because of the limited *PCI-Express* bandwidth, host-to-device and device-to-host memory transfers can easily become the bottleneck of an algorithm. Therefore, it is better to avoid them when it is possible, keeping constant data on GPU memory and moving values from/to device only when necessary. This is possible as long as the needed data fits GPU memory, otherwise it becomes necessary to design a different memory management strategies that include streaming of data. As previously stated, in this work this is generally not a problem, since memory consumption is very low even with big mesh sizes. The GPU version of Strusol performs an initialization step where GPU buffers are allocated and filled with initial or constant values. Then, all simulation data remain on GPU. No memory transfer occurs until the application wants to have access to resulting displacements. When it does so, the buffer holding current dof values is transferred and displacements are extracted and returned to the caller.

In order to easily manage memory buffers, the specialized class *ManagedArray* has been developed. It represents a generic array of items (their type is passed as template parameter). Thanks to C++ operator overloading it acts like a standard array, but provides safety checks for out-of-bounds violations and helps keeping host and device memory synchronized. In fact, it keeps two distinct buffers, one in host memory and the other one on the GPU, and transparently handles memory transfers, performing them only when one of the two arrays is modified.

During the development of the GPU algorithm, a lot of different strategies have been explored. The solver at hand, thanks to its own nature, at first sight is quite easy to parallelize. It can be decomposed into different tasks which can be easily parallelized one by one. This approach produces good results, but, as it will be discussed more in details in the next sections, in order to obtain a real improvement in performance, the whole algorithm has to be reorganized and lots of different optimization techniques have to be employed.
Therefore, two different GPU implementations will be discussed. The first one is more standard and the ideas behind it come directly from the serial implementation. The second one is the result of a complete redesign of the algorithm, where constraints of optimized CUDA programming have driven the development.

5.3.1 First GPU implementation

Analyzing the structure of the CPU algorithm, an important fact is observed: it is divided into independent parts, all of which that can be seen as loop over a set of entities. In particular, these parts can be classified as follows:

- for each dof: perform first integration half-step;
- for each node: update director and local basis;
- for each node: extract displacements from dof values;
- for each element: calculate internal forces;
- for each element: calculate external forces due to external pressures;
- for each node: add contribution of external nodal forces;
- for each dof: perform second integration half-step.

Therefore, the first obvious approach for parallelizing an algorithm with this kind of structure consists on splitting the problem into these parts and assigning each of them to one separated CUDA kernel. Each kernel then exploits the inherent nature of the assigned sub-problem: if it consists in a loop over dof quantities, each thread will operate on a single dof; if it is over mesh nodes, each thread will operate on a single node, and so on. This

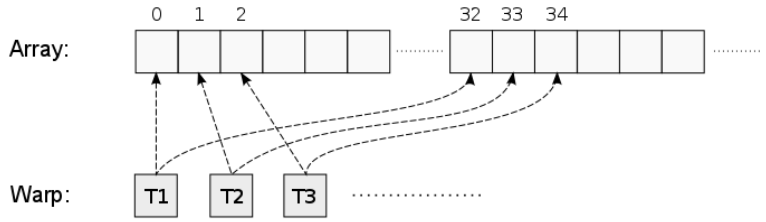


Figure 5.2: Generic thread access pattern that helps coalescing.

is a very simple fine-grain parallelization technique that can be in practice quite effective. It is obviously possible to assign to a single thread more than one entity to process. In fact, if the amount of calculations done by a single thread is too low, then latency of accesses to global memory cannot be hidden and sometimes even thread scheduling becomes a visible overhead. Therefore, in these cases often it is better to let each thread work on more entities at once (using the general pattern in figure 5.2 to pick entities, in order to ensure coalescing accesses). The actual number of entities to process can be evaluated in a pre-processing stage, trying different values until a balance is reached and maximum performances are achieved.

The main problem with this approach is that time integration is performed on dof quantities. Internal forces and external pressures are calculated element by element, but at the end they must provide a resulting value for each dof. However, dofs are shared among near elements, therefore these operations require a further *assembly* step, where resulting values are collected and forces for each dof are calculated summing contributions from all elements to which they belong.
While in CPU this is easily done, in GPU it is harder. It is basically a reduction problem: in order to collect values to be summed information about mesh connectivity and synchronization between threads (and thread blocks) are required. One approach could be employing one of the several reduction patterns that have already been studied and developed (see for example [15]). The chosen solution, implemented in Strusol, is much simpler. The main issue regarding this assembly step is that each memory location of the internal and external forces vectors is accessed by different elements on different threads, possibly concurrently. Therefore, it is necessary to avoid these conflicts and enforce that only one thread at a time can access its

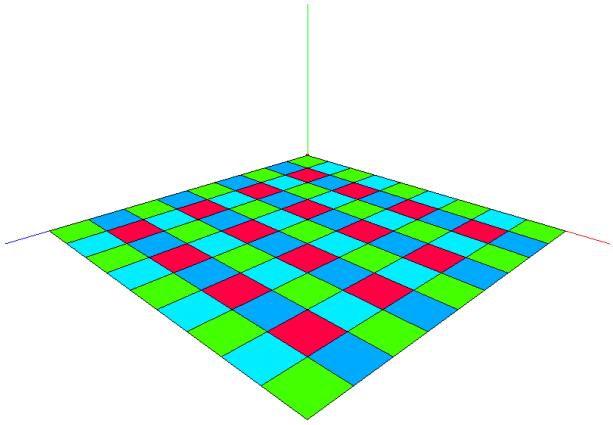


Figure 5.3: Partitioning of a quadrangular mesh.

values on these shared locations. To address this problem, a partitioning of the mesh has been introduced: elements are divided into different sets such that each element in one set does not share any nodes with other elements of the same set. An example of this partitioning can be seen in figure 5.3. It shows a quadrangular mesh composed of 81 elements with 100 nodes. It has been partitioned into 4 different element sets, shown in the figure with different colors, containing respectively 25, 20, 20 and 16 elements.
Thanks to this partitioning, assembly of internal and external forces can be performed on one element set at a time: this means sequential calls to the same CUDA kernel (with only a different parameter), which are handled efficiently by the GPU, since no kernel switch is involved. Threads are then able to write to forces vectors without any concerns of overlapping. In this way, other costly synchronization procedures are also avoided. One

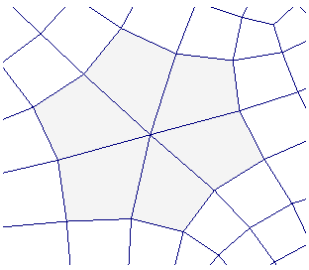


Figure 5.4: Example of node shared by many elements.

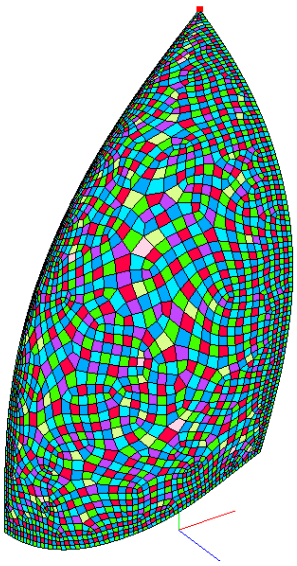


Figure 5.5: Partitioning of a sail mesh.

drawback of this approach is that mesh quality affects partitions: if lot of elements share one common node (like in figure 5.4), many element sets are created and more sequential passes have to be taken. Therefore, it is necessary to control the meshing operation and check for these situations, trying to keep the number of elements sharing the same node as low as possible.
An example of different partitioning is shown in figure 5.5 for a sail geometry, a gennaker discretized into 2694 elements, with 2846 nodes. It has been divided in element sets using a simple *greedy* approach, described in figure 5.6. It ended up split into 7 elements sets, containing respectively 667, 642, 587, 539, 183, 68 and 8 elements. Therefore, internal and external forces are computed and assembled in seven consecutive passes.

It could be possible to develop a better algorithm to partition the mesh, which would create more balanced element sets, resulting in slightly better performance. At this stage, no effort has been made to improve the greedy approach for various reasons. First of all, with some care taken while meshing the geometry, it gives satisfactory results most of the times. In any case, overall solver performance would not be improved much by better partitioning algorithms. Furthermore, the improved GPU version of the solver, which will be described below, does not need to partition the mesh, thus other solutions were not further investigated.

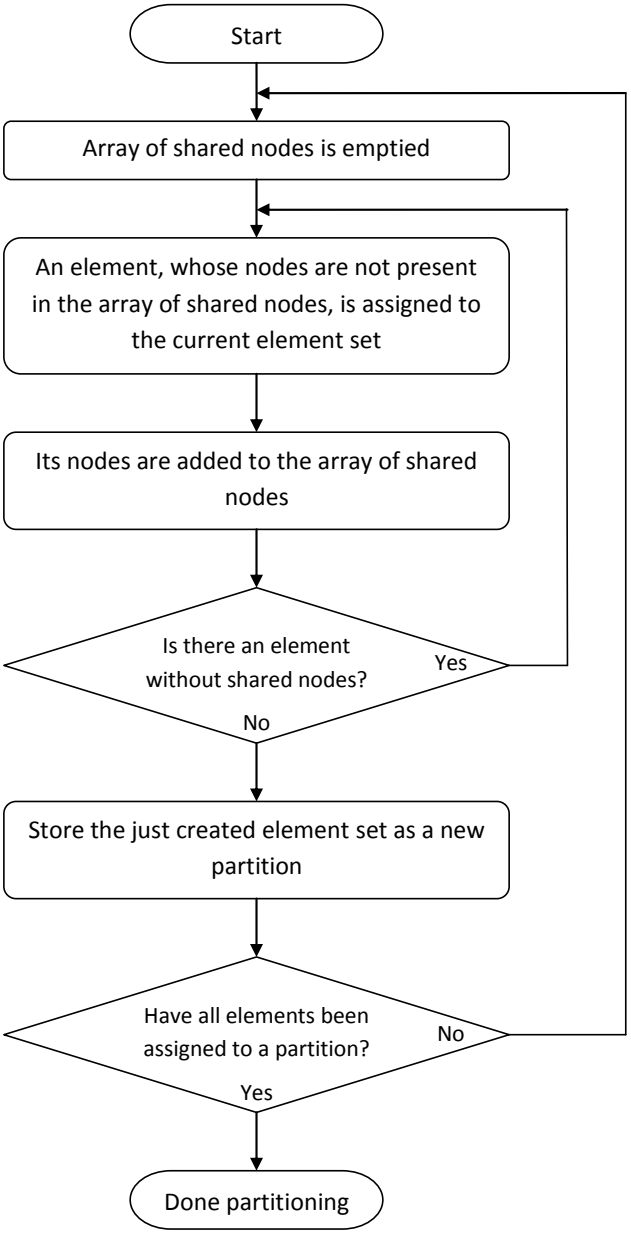


Figure 5.6: Greedy algorithm used to create mesh partitions.

5.3.2 Optimized GPU implementation

The first version of the GPU solver has been developed decomposing each time-step into sub-problems and exploiting their inherent parallelizable structure. In order to build a better implementation, a re-design of the algorithm is needed.

Solver redesign

The main issue with the previous approach is that each part of the algorithm operates on different type of entities: they are basically loops over either dofs or nodes or elements. If it was possible to express the entire time step process as a set of loops over a single type of quantity, all these loops could be aggregated into one single procedure, which could fit a single CUDA kernel and thus avoid costly kernel calls and switches.

Centering the whole algorithm around dofs as main entity is not a good idea. It would be convenient only in time integration steps. Implementation of the other passes, nodal basis update and forces calculation, would require many conditional branches and flow logic, which should be avoided in a CUDA kernel wherever possible.
Looping over nodes could be a better approach. Their one-to-five relation with dofs is convenient in order to express time integration steps (they simply would become unrolled loops over local dofs) and obviously nodal director and local basis update are trivial. Calculation of internal and external forces, the most computationally expensive step, remains hard to be performed on a per-node basis, since it requires integration over the elements. Each node would have to know which elements it belongs to, their properties and materials, and then calculate forces contributions for all those elements. This would be highly unpractical and would cause lot of wasted time to perform the same calculations more than once (considering the lack of easy synchronization between thread blocks).

All these problems are avoided choosing to express all loops over the elements. In order to do so, it is necessary to keep for each element all information about its local nodes. In this way, each element can perform time integration and nodal basis update by itself, on its own nodes, and then it can proceed to calculate forces. All steps can thus be done in one single large CUDA kernel.

In order to optimize the time integration pass internal of the time-step some rearrangements to the default central finite differences method are needed. It is necessary to maintain the forces calculation step as the last one: it is the only step that needs explicit communication between elements. To avoid global communications inside the CUDA kernel it must be placed at the end of kernel execution, so that it can rely on the implicit synchronization occurring between successive kernel calls.
As it has already been described, at time-step n the standard central finite differences method can be summarized as follows (i is the dof index):

- first integration half-step:

$$v_i^{n+\frac{1}{2}} = v_i^n + \frac{1}{2}\,\Delta t\, a_i^n, \tag{5.9}$$

$$u_i^{n+1} = u_i^n + \Delta t\, v^{n+\frac{1}{2}}; \tag{5.10}$$

- forces calculation:

$$\mathbf{F}_I^{n+1} = \text{internalForces}\left(\mathbf{U}^{n+1}\right), \tag{5.11}$$

$$\mathbf{F}_E^{n+1} = \text{externalForces}\left(\mathbf{U}^{n+1}\right); \tag{5.12}$$

- second integration half-step:

$$a_i^{n+1} = \frac{f_{E,i}^{n+1} - f_{I,i}^{n+1} - c_i\, v_i^{n+\frac{1}{2}}}{m_i + \frac{1}{2}\,\Delta t\, c_i}, \tag{5.13}$$

$$v_i^{n+1} = v_i^{n+\frac{1}{2}} + \frac{1}{2}\,\Delta t\, a_i^{n+1}. \tag{5.14}$$

In order to perform the desired optimizations it is necessary to move the forces calculation step to the end of the algorithm. In order to do so, equations (5.9) and (5.10) can be preponed soon after the second integration pass of the precedent time-step.

The procedure then becomes:

$$\mathbf{F}_I^{n+1} = \text{internalForces}\left(\mathbf{U}^{n+1}\right), \tag{5.15}$$

$$\mathbf{F}_E^{n+1} = \text{externalForces}\left(\mathbf{U}^{n+1}\right), \tag{5.16}$$

$$a_i^{n+1} = \frac{f_{E,i}^{n+1} - f_{I,i}^{n+1} - c_i\, v_i^{n+\frac{1}{2}}}{m_i + \frac{1}{2}\,\Delta t\, c_i}, \tag{5.17}$$

$$v_i^{n+1} = v_i^{n+\frac{1}{2}} + \frac{1}{2}\,\Delta t\, a_i^{n+1}, \tag{5.18}$$

$$v_i^{n+1+\frac{1}{2}} = v_i^{n+1} + \frac{1}{2}\,\Delta t\, a_i^{n+1}, \tag{5.19}$$

$$u_i^{n+2} = u_i^{n+1} + \Delta t\, v^{n+1+\frac{1}{2}}. \tag{5.20}$$

It is now possible to anticipate the force calculation pass at the end of the precedent time-step. Merging velocity updates and normalizing time-step superscripts, we obtain:

$$a_i^{n+1} = \frac{f_{E,i}^{n} - f_{I,i}^{n} - c_i\, v_i^{n-\frac{1}{2}}}{m_i + \frac{1}{2}\,\Delta t\, c_i}, \tag{5.21}$$

$$v_i^{n+\frac{1}{2}} = v_i^{n-\frac{1}{2}} + \Delta t\, a_i^{n+1}, \tag{5.22}$$

$$u_i^{n+1} = u_i^{n} + \Delta t\, v^{n+\frac{1}{2}}, \tag{5.23}$$

$$\mathbf{F}_I^{n+1} = \text{internalForces}\left(\mathbf{U}^{n+1}\right), \tag{5.24}$$

$$\mathbf{F}_E^{n+1} = \text{externalForces}\left(\mathbf{U}^{n+1}\right). \tag{5.25}$$

It is obviously necessary to perform an initialization step where initial forces, dof values and half-step velocities are calculated.

With the time advancing pass expressed in this form, there are two main advantages. First of all, the is no need to store dof acceleration and velocity values. The only buffers needed are related to dof values, half-step velocities and forces. Furthermore, this procedure does not need synchronizations inside a time-step. Having given to each element its own copy of nodal values, they are now able to perform calculations completely independently from each other. The whole time-step advancing procedure is contained in a single big CUDA kernel, where each thread is assigned to an element. A

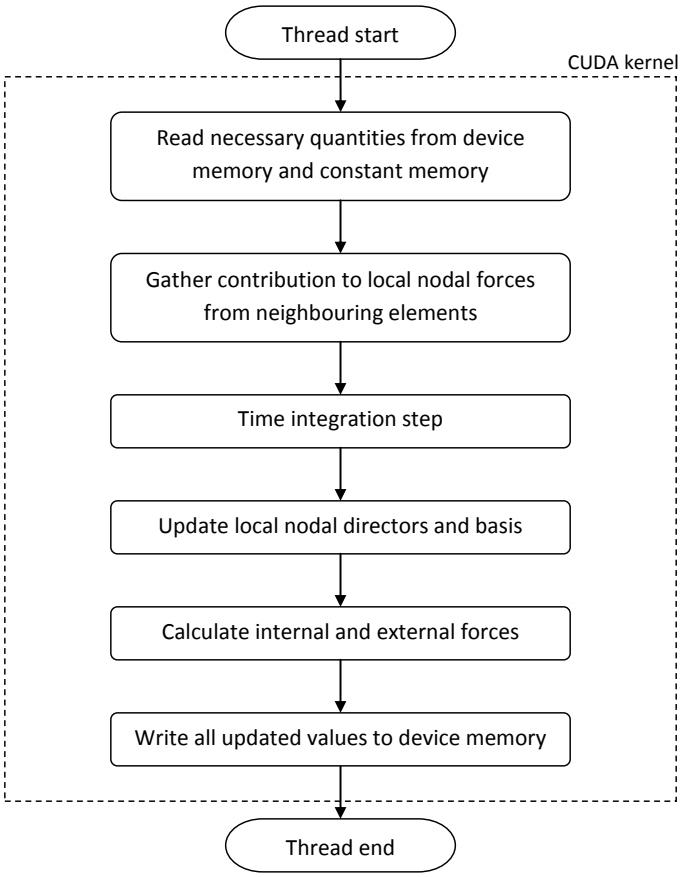


Figure 5.7: Steps performed in time integration kernel.

schematized flow chart of this procedure is shown in figure 5.7.

Performing this calculation in an element by element manner means that some calculations are done multiple times: each element has to integrate quantities and build nodal basis on its own local nodes, even if they are shared with other elements. The only communication allowed among different elements is in the force assembly step, where forces coming from near elements are gathered. This approach relies on these facts:

- as far as GPU programming is concerned, performing the same calculations more than once often is better, if that means fewer memory accesses. GPUs are much more powerful at doing calculations rather than reading/writing memory;
- in this case, forces calculation is, by far, the most computationally

expensive pass. Therefore, if there is the necessity to repeat some of the calculations (to avoid expensive memory accesses), it is better to re-calculate values related to the other lighter passes;

- memory consumption of this algorithm is not a problem, thus some memory can be wasted duplicating necessary information, if that leads to simplification in the kernel or better memory access patterns.

In order to sustain this last point, some simple benchmarks can be performed. To fully understand this point however a deeper explanation on how the buffers are managed is necessary. Memory consumption analysis is presented afterward.

Structure layout and memory optimizations

With respect to the previous GPU implementation, also GPU memory management has been redesigned and optimized. When necessary, the managed array class is now able to organize data in a structure-of-arrays (SoA) manner, instead of the more intuitive array-of-structures (AoS). These memory layouts influence how values are read and stored to device memory; therefore, since bottlenecks of most algorithms are due to memory accesses, data organization is really important in order to ensure coalescence and generally good performances of the implementation.
An example of these different memory layouts is the following. In order to store a 3D coordinates array, the common AoS layout is:

```
struct Vector3 {
    Real x;
    Real y;
    Real z;
};
struct Vector3 coordinates[N];
```

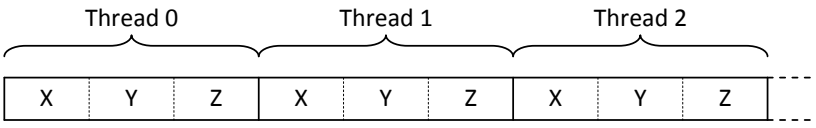


Figure 5.8: Thread access pattern with AoS layout.

When used in CUDA kernels this is often not efficient. When each thread reads a structure of values from device memory, accesses with this kind of layout often break coalescing patterns. For example, if each thread (with index i) has to read the x coordinate of the vector at position i into the array, considering all threads in a warp, the wanted values are not contiguous, therefore accesses cannot be coalesced. This causes serialized memory reads and wasted bandwidth and cycles. A common solution is to store structure values in separate arrays (structure-of-arrays approach, SoA), as follows:

```
struct Coordinates {
    Real x[N];
    Real y[N];
    Real z[N];
};
struct Coordinates coordinates;
```

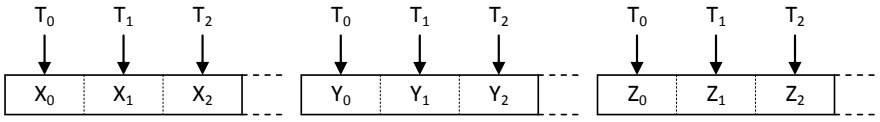


Figure 5.9: Thread access pattern with SoA layout.

With this memory layout, each thread reads a value at a time from separate arrays: the values accessed at the same time by threads inside a warp are contiguous, thus accesses can be fully coalesced and performances are dramatically increased.

This layout gives the best performance most of the times, but there are some situations where other approaches can be useful. Sometimes, having the wanted values stored close one to another is good for caching purpose and, if the structure is of the right size, accesses to it can still be fast (assuming the base address being accessed is aligned correctly): 64- and 128-bit loads and stores can be done with a single instruction by the hardware. Therefore, with vectorized types such as *float2* (two consecutive 32-bit floating point numbers) or *float4* (four consecutive 32-bit floating point numbers) an AoS layout can sometimes be better than a SoA approach, since fewer memory operations are issued and a smaller number of pointers has to be managed. It depends on the specific situation; sometimes, even

an hybrid approach could be beneficial, for example a *SoAoS* layout, with correctly aligned arrays of right-sized structures. In this cases often the best approach is *trial-and-benchmark*.

However, in all situations three-component vectorized types are never a good container: they break not only coalescence rules but also alignment. One possible solution to this problem can be clever packing of values. For example, let us assume that storage for node coordinates of an element is needed. This would mean 4 vectors (one for each local node) of 3 components (since they are vectors in a 3D space). Instead of storing these values in triples and then performing 4 memory accesses of 3 *floats* each, which is not good from alignment and coalescence point of view, it is possible to store these values in three vectors of 4 scalars, packing the fourth vector components into the last spaces of the other vectors (as in figure 5.10).

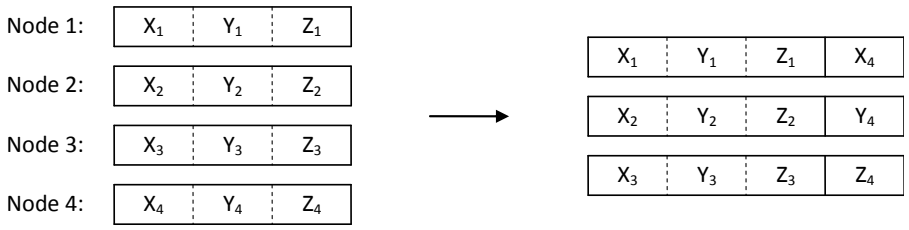


Figure 5.10: Example of vector packing.

It is then necessary to perform 3 memory accesses of 4 values each. If data type is 32-bit wide and addresses are aligned, these transfers are not slowed down. However, it is important to mention that usually double precision floating point arithmetic is wanted. But *doubles* are 64-bit wide; therefore, in the implementation described in this work best performances are achieved using generally a SoA approach for data organization (AoS is used only in a few places, with 4 x 32-bit vectorized types).

While buffers with element quantities are stored in GPU device memory, common parameters or pointers are stored in constant memory. Values such as quadrature points and weights or mass damping coefficients are shared by all elements, thus placing them in constant cache ensures good access latency.

Another optimization that was adopted regards Dirichlet boundary conditions handling. At first, to know if a dof was fixed (to zero or to some

predefined value) or free it was necessary to keep a separate memory buffers which contained one multiplier per dof: a value of 0 indicated a fixed dof, a value of 1 a free dof. This factor was then used as a multiplier for final acceleration values, leading to a non-zero acceleration and velocity only if the dof was not fixed. This solution was efficient from the calculation point of view, since it did not break execution flow path with conditional branches, but it had the draw-back of requiring other memory accesses to read multiplier values.
A better solution has then been implemented considering that a bit of information can be stored in the sign of mass values. Since mass is obviously always positive, a negative mass indicates a fixed dof. In this way, a conditional branch (over the mass sign) has been introduced in the integration pass, but the reads from device memory are avoided. This little trick improved performances quite significantly, underlining how GPU code optimizations sometimes can be unobvious.

This kernel is very computationally expensive. Calculation of internal forces requires lots of computational power, thus memory access latency is well hidden. Furthermore, almost all memory accesses are organized to be fully coalesced. The only pass that actually breaks coalescence patterns and introduces overhead is the local forces exchange pass (the second one in figure 5.7). Each element has to perform dof integration on its own local nodes. In order to do so, it needs to assemble forces on these nodes. The force calculation pass (performed at the end of the previous time-step) gives only a partial contribution to local forces: it is necessary to collect all contributions from neighboring elements also. In particular, each element needs to know, for all its local nodes, which elements contain those nodes and what their local indices are.
In figure 5.11 a simplified mesh with four elements is shown. Global node indices are written in black, local node indices in red: as an example, node 5 has local index 2 for element B, but local index 1 for element C.
Let us consider element B. At the beginning of the time-step the corresponding thread reads from device memory the contributions to local forces in its own nodes, calculated at the end of the precedent time-step. After that, it needs to access neighboring elements' local forces array in order to gather all necessary contributions to its own local forces: local node 0 needs force contribution from node 1 of element A; local node 3 needs contribution

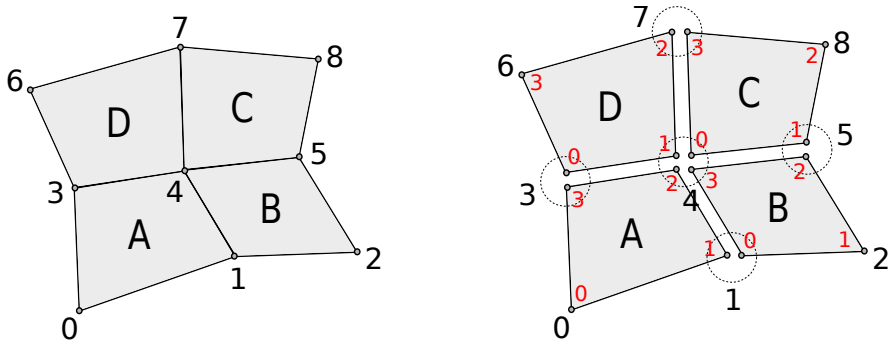


Figure 5.11: Local node numbering for neighbour arrays: node global indices are in black, local indices in red.

from node 0 in element C, node 1 in element D and node 2 in element A, and so on. These connectivity arrays are built in a pre-processing step and remain unchanged during the simulation. This is the only pass inside the time-step where un-coalesced reads to device memory are performed. However, number of accesses is small and it does not impact on overall solver performance. It could be possible to improve this pass of the algorithm, for example employing some clever ordering of elements together with block shared memory; this is left to be done in future releases of the solver.

Finally, it is possible to approximate the memory consumption of this algorithm, to prove the fact that wasting memory on duplicating necessary information can sometimes be a good idea. As far as GPU memory is concerned, the following buffers are allocated:

- read/write buffers:
  - dof values: 20 x (number of elements) scalars;
  - dof half-step velocities: 20 x (number of elements) scalars;
  - nodal directors: 4 x (number of elements) x (3 scalars);
  - nodal basis: 2 x 4 x (number of elements) x (3 scalars);
  - element local forces: 20 x (number of elements) x 2 (because of the double buffering).
- read-only buffers:
  - initial nodal directors: 4 x (number of elements) x (3 scalars);
  - dof masses: 20 x (number of elements) scalars;

- element material properties and external pressures: 4 x (number of elements) scalars;
- external nodal forces: 4 x (number of elements) x (3 scalars);
- initial node coordinates: 4 x (number of elements) x (3 scalars);
- neighbors array: 2 x 4 x (max. number of neighbors) x (number of elements) integers.

By hypothesis, let us assume that 6 is the maximum number of elements sharing the same node. Summing up all contributions, it results that each element requires 176 scalars and 48 integers. Considering double precision arithmetic, so that scalars are 8 bytes long, and 32-bit integers, each elements requires 1600 bytes of storage. A low-range gaming GPU now has often at least 1 GB of video RAM. *Tesla* computing boards are equipped usually with several GBs of memory. For each GB, theoretically 671088 elements can be stored. This is an incredible big number: usually meshes are much smaller, since the MITC4 element is considered to be quite rich, with its 5 dofs per node, and there is no need for fine refinements in most cases. In case of bigger meshes, they can always be partitioned and asynchronically streamed to GPU memory while executing; this would require specialized management of elements at sub-domain interfaces, but the overall structure of the solver would remain the same.

Code and CUDA optimizations

After this whole process of algorithm redesign, it was necessary to perform other optimizations at code level, in order to squeeze more performance out the GPU. First of all, it is important to notice that the algorithm is independent from the size of thread blocks: these have to be monodimensional, with a number of threads multiple of 32 in order to guarantee fully coalesced memory accesses, but the actual size can vary. Since it depends on far too many factors, a crude but simple way to find out the best block size to employ is to try them all: the time advancing process is run for two seconds for each block size and the number of time-steps completed in that amount of time is taken as reference in order to choose the most performant one. It is stored on a file, so subsequent simulations on the same problem are run with the chosen block size. Results of this tuning step are GPU and problem dependent. For example, in figure 5.12 average computation

time of a single time-step is shown for a problem with two different meshes (with 1596 and 3680 elements respectively) run on a laptop equipped with a NVIDIA GeForce GTX 670M (336 CUDA cores) and tower PC with a NVIDIA GeForce GTX 470 (448 CUDA cores).

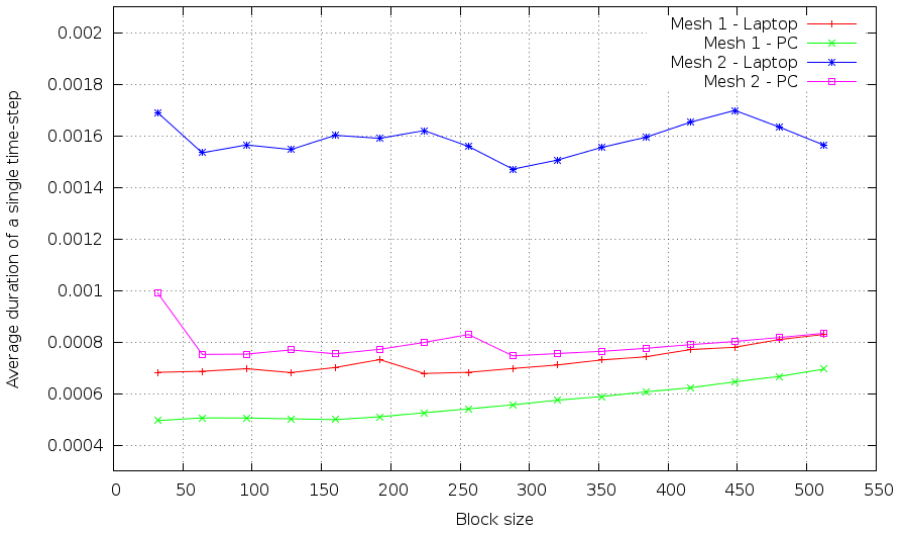


Figure 5.12: Thread block size tuning for a problem with two different meshes on two different GPU boards.

Finally, other miscellaneous optimizations to the source code were made. With the aid of profiling tools (namely NVIDIA *nvprof*, *Visual Profiler* or *Nsight*) bottlenecks of the algorithm were investigated and corrected when possible. In particular, in this kernel the biggest cause of performance drop is register spilling: there are too many local variables to fit the available registers, thus some of them are stored in local memory (which has the same latency as global memory) causing a significant performance drop. This problem was alleviated in two steps. Initially, reads and writes to global memory were performed at the beginning and at the end of the time-step respectively; in order to avoid storing values in temporary variables for the whole time-step process, reads and writes were moved before and after the actual code using them. Furthermore, in order to help the CUDA optimizer, a meticulous process of local variable scope tuning (enforced dividing manually the source code in blocks using curly brackets) and forcing variable reuse was performed, with good results in terms of performance gained (around 20% more).

This whole algorithm redesign and optimization process has proven to be very useful: with respect to the first GPU implementation, this new one is approximately 2-3 times faster, depending on the problem and the GPU board, and it could probably be improved further. In the next sections some benchmarks assessing performance of the GPU implementation will be discussed.

5.4 Structural test cases

In order to check accuracy, robustness and performance of the structural solver various test cases have been built and comparison made with results obtained using the commercial software Abaqus, a software suite for computer aided design and finite element analysis, part of *SIMULIA* by *Dassault Systèmes*. It comes with an extremely rich environment of tools that cover from 3D modeling and pre-processing to the finite element analysis itself and post-processing. It supports an extensive set of elements and integration techniques, as well as mesh generation and refinement algorithms.

In order to check results against the shell dynamics solver presented in this work, *Abaqus-Explicit* finite element analyzer has been used and tweaked to behave similarly. In particular, elements of type *S4*, without reduced integration, have been employed. Strusol comes with an Abaqus input files importer, which can read a subset of Abaqus job's keywords. Consequently, meshes are modeled in Abaqus CAD interface and directly imported into the shell solver without modifications.

Post-solution analysis is carried on using the internal Abaqus post-processor when dealing with Abaqus results. Strusol results, instead, are processed using *Paraview*, a powerful open-source scientific visualization tool.

5.4.1 Uniformly loaded circular plate

This first test is meant to check solver accuracy and to compare performance of the CPU and GPU implementation against the commercial software Abaqus. It is quite simple, but has also an essential and very rare property: an analytical solution, for the steady case, is available.

A circular plate made of an isotropic homogeneous material is analyzed in constant external pressure and simply supported edge conditions, i.e. border's vertical displacements are blocked, while rotations are allowed.

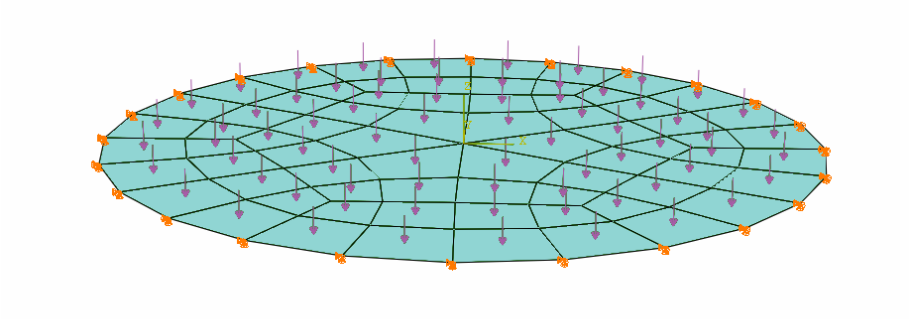


Figure 5.13: Circular plate test case.

Under small displacements assumption, an analytical solution for displacements can be found following the procedures in [32]. Using Reissner-Mindlin shell theory, vertical displacements of plate points can be calculated:

$$u_y(\rho) = \frac{p_{ext} R^4}{64D}\left(1-\rho^2\right)\left[\frac{5+\nu}{1+\nu} - \rho^2 + 16\frac{\lambda^2}{\beta}\right], \tag{5.26}$$

where p_{ext} is the constant external pressure applied on the plate, R is plate's radius and D represents its flexural rigidity:

$$D = \frac{Eh^3}{12\left(1-\nu^2\right)}, \tag{5.27}$$

where h is the thickness, E is the Young modulus, ν is the Poisson's ratio, $\rho = r/R$ is an adimensional radial coordinate, $\lambda = h/R$ is an index of how the plate is thin with respect to its radius and β is a characteristic parameter, which is, as far as homogeneous plates are concerned, equal to $5\left(1-\nu\right)$.
For all tests the following parameter values have been considered:

- Material: $E = 206\,000\,\text{MPa}$, $\nu = 0.3$;
- Geometry: $R = 10\,\text{mm}$, $h = 0.1\,\text{mm}$;
- External pressure: $p_{ext} = 0.0001\,\text{MPa}$.

These values lead to a static vertical displacement of the middle point of $0.003\,377\,\text{mm}$.

In order to compare the numerical solution against the analytical one, this problem is discretized by differently refined meshes. Since the analytical solution refers to the steady case and the solver is purely transient, a little damping is applied and the solver is left running until convergence. Numerical results are summarized in table 5.1.

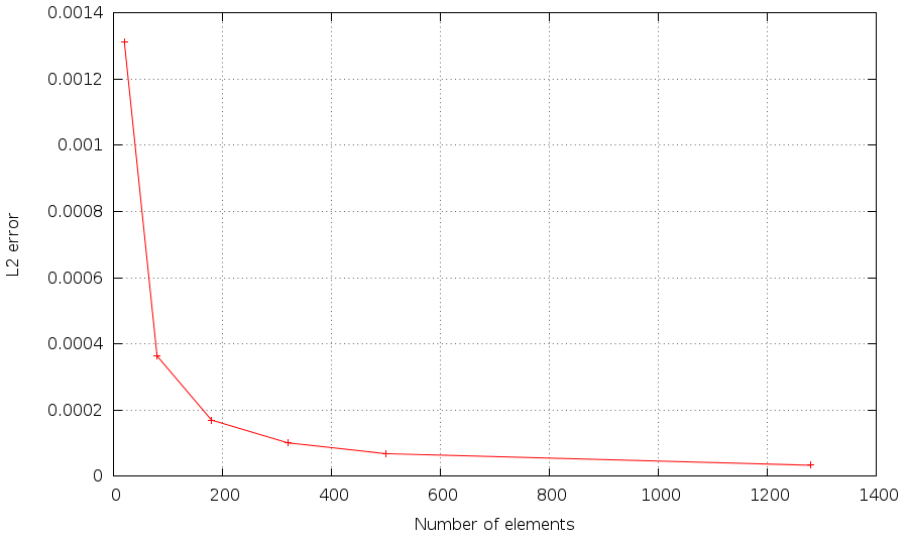
Mesh	Elements	L^2-norm error	Punct. error (%)
1	20	$1.31\ 10^{-3}$	3.772
2	80	$3.64\ 10^{-4}$	0.995
3	180	$1.70\ 10^{-4}$	0.462
4	320	$1.01\ 10^{-4}$	0.276
5	500	$6.84\ 10^{-5}$	0.189
6	1280	$3.33\ 10^{-5}$	0.095

Table 5.1: Comparison with analytical solution.

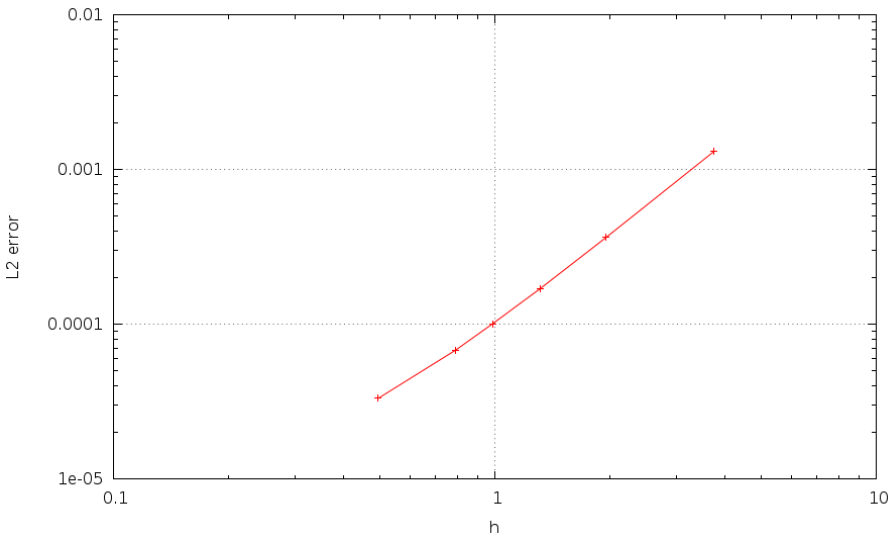
Last column of the table refers to the circular plate's central point punctual error against the analytical solution. It is just an indicative value, since punctual convergence cannot be guaranteed.

In figure 5.14, L^2-norm error is shown, firstly over the number of mesh elements, then over an approximate element size measure, in logarithmic scale. Second order convergence is achieved as expected for this kind of bilinear isoparametric elements from the finite-element analysis theory.

These results refer to a static case. The same problem setup is used to perform also a dynamics analysis. Figure 5.15 shows an example of the resulting deformation after 100 seconds of simulation. A transient analytical solution is not available, therefore results are tested against the commercial



(a) L^2-norm error over number of elements



(b) L^2-norm error over characteristic element size h, in logarithmic scale

Figure 5.14: L^2-norm error against analytical solution.

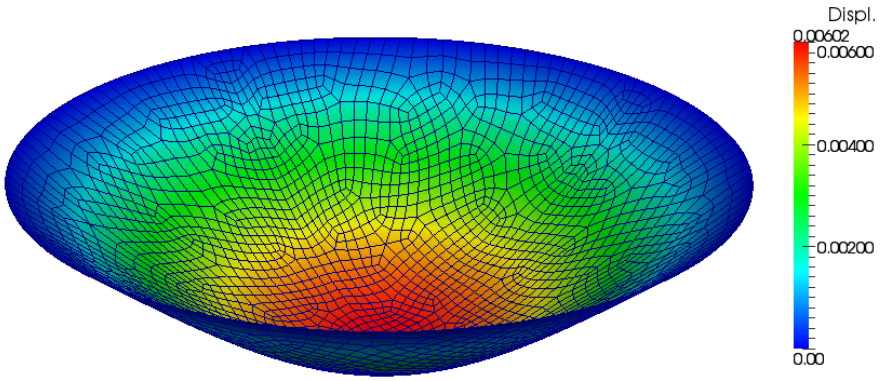


Figure 5.15: Deformed mesh after 100 seconds (3023 elements). Displacements have been amplified by a scaling factor of 1000.

software Abaqus. In figure 5.16 vertical displacements of the circular plate's central point are shown for a 120 seconds time period, calculated by both solvers. Obviously, the two solutions are practically identical, since the same discretization and element are used.

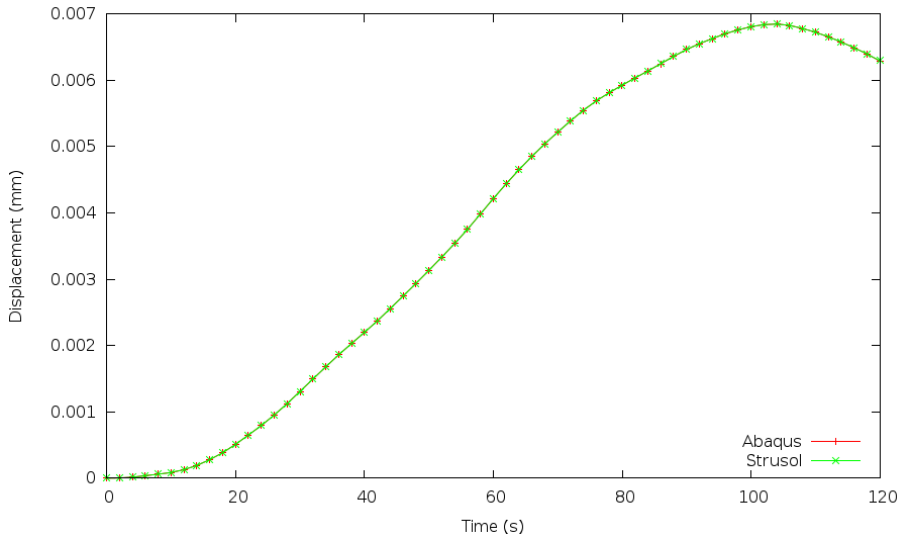


Figure 5.16: Solution comparison against Abaqus.

Having checked that Abaqus and Strusol produce the same results, it is possible to compare performance too. Different mesh refinements are tested, with more elements than the meshes used to compare results with the analytical solution, in order to make trends in time duration and speedup more visible. Their number of elements and nodes are reported in table 5.2.

Mesh	Elements	Nodes
1	500	521
2	1125	1156
3	2000	2041
4	3125	3176
5	4500	4561
6	8000	8081
7	12 500	12 601

Table 5.2: Meshes used for performance comparison.

All meshes have been created in Abaqus. In order to make the comparison as accurate as possible, the very same job files, exported from Abaqus as results from the modeling stage and input for the simulation stage, are imported in Strusol without any change; even the number of times displacements are written to file is kept the same. Abaqus mesh is made of *S4* elements, without reduced integration.

Abaqus is equipped with a good time-step size estimation algorithm that adapts it during the simulation, trying to keep it as large as possible, while preserving stability. Strusol instead calculates a (very restrictive) time-step size at the beginning and then keeps it for the whole simulation; it let the user change it when he desires, but it does not come with an included automatic adaptation algorithm yet.

As a matter of fact, the geometry of the problem at hand is very simple and Abaqus uses only two different time-step sizes trough the whole simulation period: one smaller, only for the first time interval, and the other bigger, for the rest of simulation. For the sake of comparison, in these simulations Strusol's time-step size is fixed as the smallest one used by Abaqus. Those time-step sizes are reported in table 5.3.

For each mesh, time taken to perform 120 seconds of simulation is mea-

Mesh	Elements	First Δt	Largest Δt
1	500	0.01464	0.02119
2	1125	0.00894	0.01325
3	2000	0.00638	0.00951
4	3125	0.00495	0.00736
5	4500	0.00404	0.00597
6	8000	0.00294	0.00431
7	12 500	0.00231	0.00335

Table 5.3: Time-step sizes used by Abaqus during the simulations.

sured. Measures do not include any pre-processing or post-processing time, only the actual computational time. Abaqus and Strusol CPU simulations have been run on a single core of an *AMD Phenom X4 9950 Black Edition* CPU at 2.6 GHz. Strusol GPU simulations have been run on the same machine, equipped with a *NVIDIA GeForce GTX 470* GPU, provided with 448 CUDA cores (14 SM, 32 cores/SM). All simulations have been made on Linux, without graphical user interface (to not interfere with GPU computations). Results obtained measuring computational time with double-precision and single-precision floating point arithmetic are summarized respectively in tables 5.4 and 5.5.

With the biggest mesh considered, in double precision arithmetic, Strusol runs over 33 times faster than Abaqus. Considering that Abaqus uses the largest time-step size, 1.45 times bigger than the small one, only for the first time increment, while Strusol uses it for the whole simulation, it is fair to compare the two implementations in the same situation also, turning off Abaqus' adaptive time-step and forcing it to use the same time-step size as Strusol. In this situation, the gained speedup is over 43x.

These simulations have been done using a GPU suited for video-gaming and not specialized in scientific computing. Despite the good results shown using double-precision arithmetic, these kind of video boards are more suited for single-precision arithmetic, because in 3D graphics real-time performance is required and it is important to obtain visually-appealing quick results, sometimes sacrificing physical accuracy. So, just for the sake of comparison, the same simulations have been performed with single-precision arithmetic: speedup of 46x and 61x have been obtained considering Abaqus' adaptive and fixed time-step configurations respectively. If computationally-

Mesh	Abaqus (adaptive)	Abaqus (fixed)	Strusol CPU	Strusol GPU
1	24.86	32.13	13.28	2.46
2	88.02	118.54	52.65	5.29
3	216.56	291.86	141.77	9.38
4	435.28	576.10	263.00	15.24
5	769.04	1023.15	473.38	24.46
6	1884.61	2472.49	1120.33	59.76
7	3753.52	4845.03	2308.00	111.51

Table 5.4: Computational time in seconds (double-precision arithmetic).

Mesh	Abaqus (adaptive)	Abaqus (fixed)	Strusol CPU	Strusol GPU
1	13.57	18.32	13.47	1.18
2	46.82	63.77	54.20	2.37
3	115.92	158.13	132.92	4.05
4	231.33	312.44	273.46	6.36
5	408.17	541.74	486.52	9.73
6	988.34	1318.71	1241.45	24.12
7	1973.87	2606.35	2278.53	42.43

Table 5.5: Computational time in seconds (single-precision arithmetic).

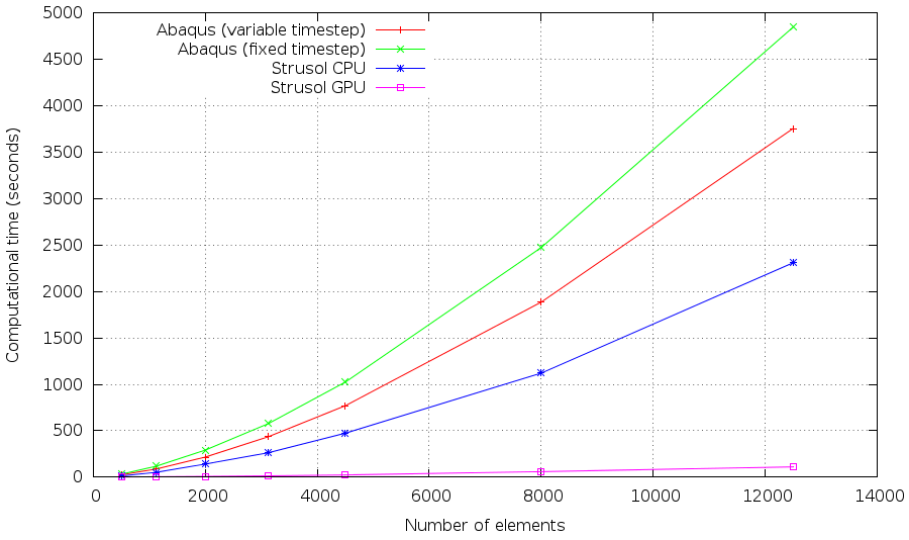


Figure 5.17: Computational time (double-precision arithmetic).

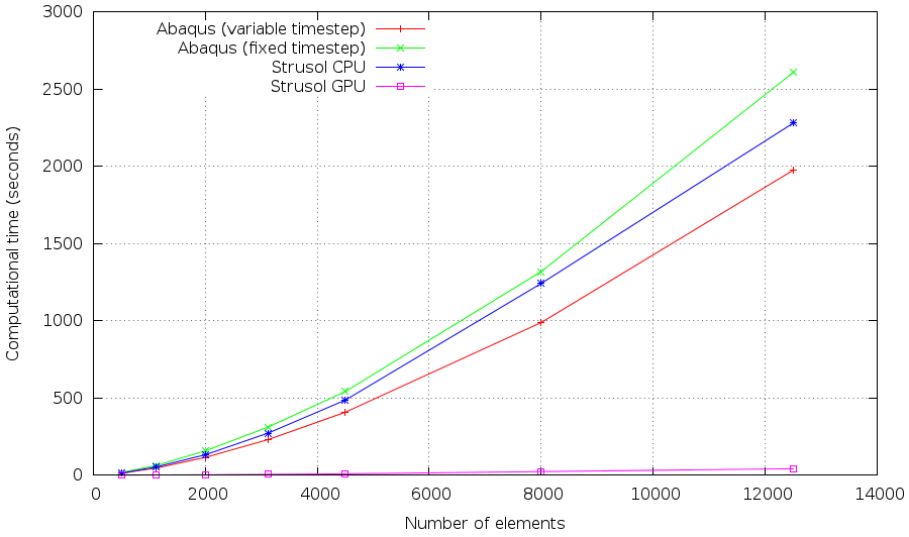


Figure 5.18: Computational time (single-precision arithmetic).

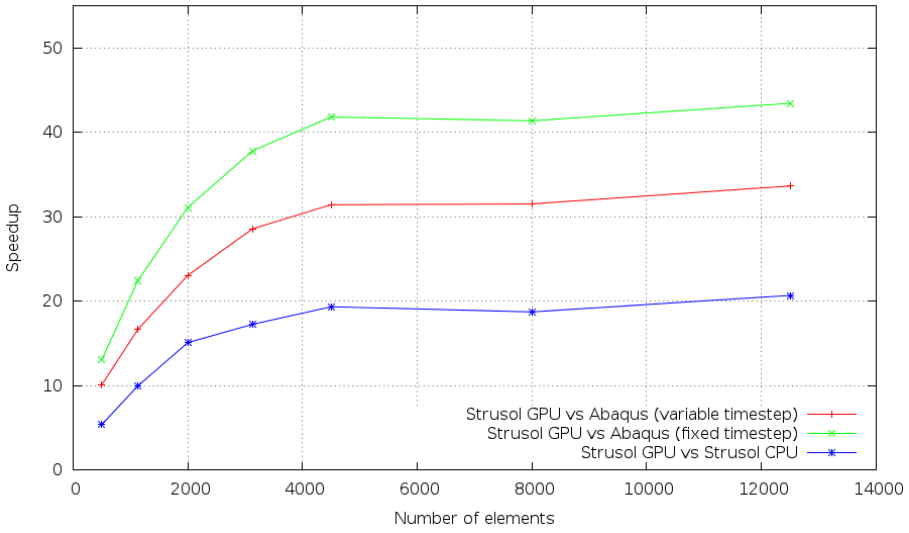


Figure 5.19: Speedup (double-precision arithmetic).

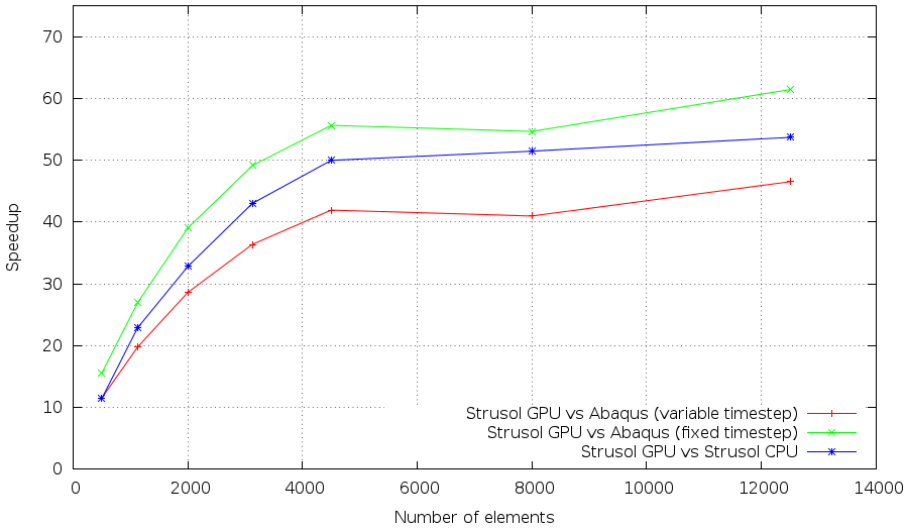


Figure 5.20: Speedup (single-precision arithmetic).

oriented GPU boards are employed, like for example *NVIDIA Tesla*, which are equipped with high-performing double-precision ALUs, even better performances are achievable and the speedup curve would saturate at an higher value. It is interesting to notice also that, even with the finest mesh considered, the GPU implementation is able to compute 120 seconds of simulation in less than 112 seconds. This means that the problem can be simulated completely in real-time, using the full MITC4 elements and without requiring reduced integration or simpler elements.

Results show how this kind of problems are well suited for GPU implementation. Explicit time integration requires a large amount of very small time-steps to be performed, but on GPU they can be executed extremely quickly and overall performances are satisfactory. It is also important to emphasize that during these simulations the CPU is practically idle. This fact can be exploited in more complex situations where different systems have to be simulated concurrently: keeping computation on GPU frees the CPU to perform other demanding tasks.

5.4.2 Clamped rectangular plate

This second structural case represents a test for the large displacement formulation and a simple benchmark for performance.
A rectangular plate is fixed at one edge. Its dynamics is analyzed under a constant tip load. Figure 5.21 shows the geometry of the problem.

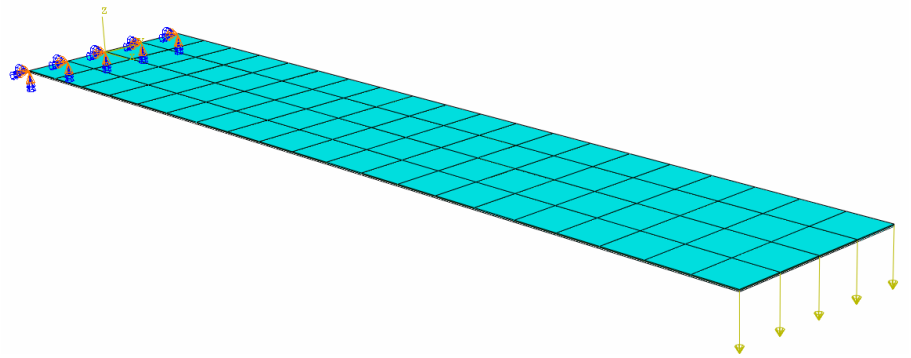
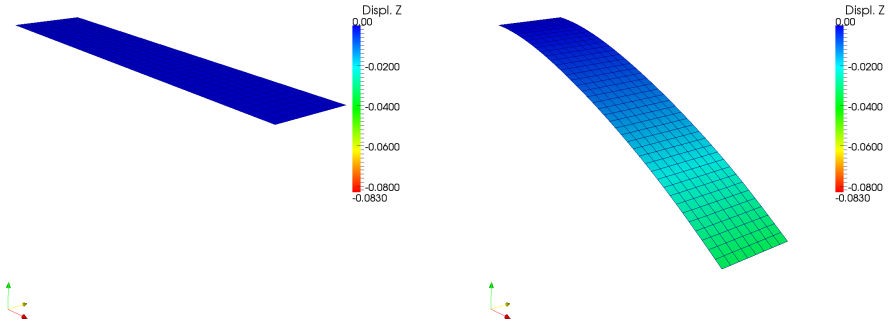


Figure 5.21: Clamped cantilever test case.

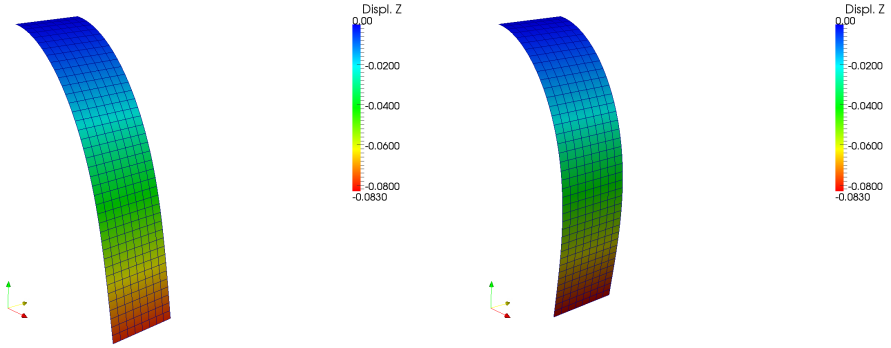
The following setup is considered:

- Material: $E = 1768\,\text{MPa}$, $\nu = 0.3$, $\rho = 3000\,\text{kg/m}^3$;
- Geometry: $100\,\text{mm}$ x $20\,\text{mm}$, $h = 0.2\,\text{mm}$ (thickness);
- External tip load: $P_{ext} = 0.001\,56\,\text{MPa}$.

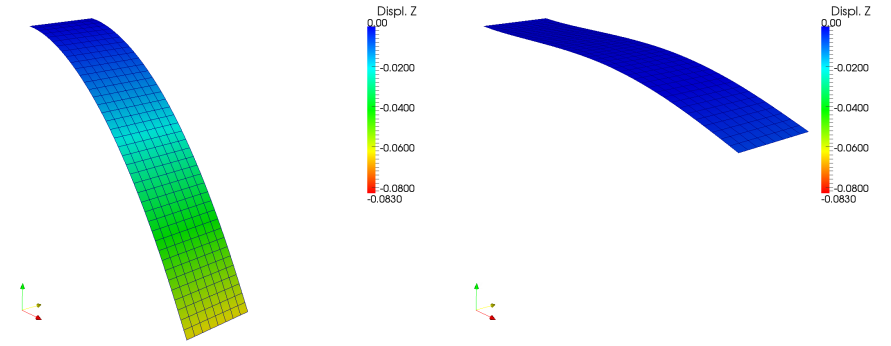
These parameters have been accurately chosen to cause large displacements of the structure, in order to test the solver in such situations. Evolution of the deformed structural mesh is visible in figure 5.22, where displacements at different times are shown. The same identical problem is again imported in Abaqus in order to check the implementation against a widely known and tested commercial software and to compare their performance. As a reference, vertical displacements of the central point on the tip of the cantilever are extracted and compared. They follow the curve shown in figure 5.23. Once again, pointwise values calculated by Abaqus and Strusol are obviously identical, even in case of large displacements. What is interesting are performance benchmarks.



(a) $t = 0\,\mathrm{s}$ (b) $t = 0.0525\,\mathrm{s}$



(c) $t = 0.105\,\mathrm{s}$ (d) $t = 0.1575\,\mathrm{s}$



(e) $t = 0.21\,\mathrm{s}$ (f) $t = 0.2975\,\mathrm{s}$

Figure 5.22: Deformations of the cantilever at different times, without displacement scaling factor.

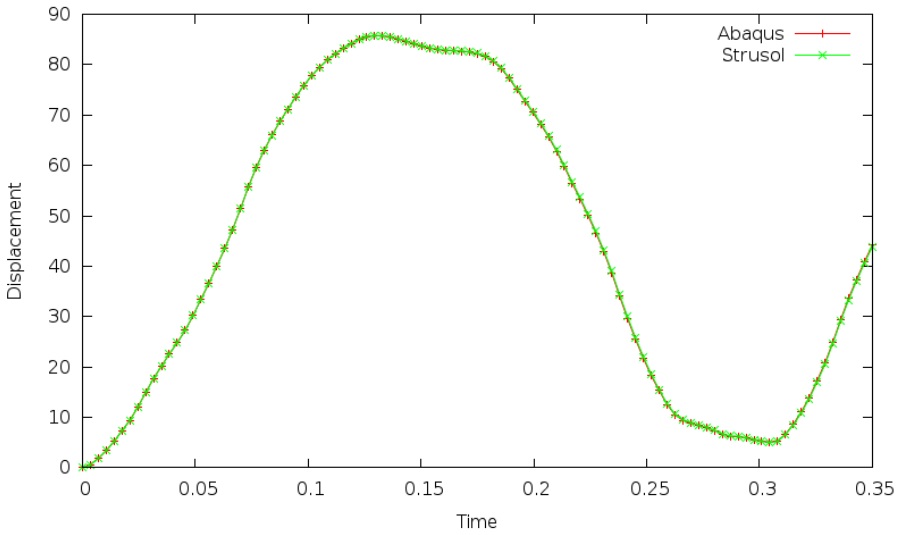


Figure 5.23: Solution comparison against Abaqus.

In order to conduct a performance comparison, different mesh refinements are tested (listed in table 5.6). This kind of mesh is well suited for benchmarks, being in fact structured and very easy to refine. Moreover, considering the GPU implementation, the regularity of the grid helps during the internal forces intercommunication process: nodes neighbor lists are of the same size, so the majority of threads executes the same operations and memory transactions, without breaking the execution path (see the implementation chapter for details).

Thanks to the time-step adaptation process, Abaqus again calculates and uses during the whole simulation only two different time-steps: a first

Mesh	Elements	Nodes
1	80	105
2	160	205
3	320	369
4	640	729
5	1280	1377
6	2560	2737
7	5120	5313
8	10 240	10 593

Table 5.6: Meshes used for performance comparison.

Mesh	Elements	First Δt	Largest Δt
1	80	$4.923 \cdot 10^{-6}$	$5.343 \cdot 10^{-6}$
2	160	$2.767 \cdot 10^{-6}$	$2.767 \cdot 10^{-6}$
3	320	$2.461 \cdot 10^{-6}$	$2.677 \cdot 10^{-6}$
4	640	$1.383 \cdot 10^{-6}$	$1.383 \cdot 10^{-6}$
5	1280	$1.231 \cdot 10^{-6}$	$1.339 \cdot 10^{-6}$
6	2560	$6.917 \cdot 10^{-7}$	$6.917 \cdot 10^{-7}$
7	5120	$6.153 \cdot 10^{-7}$	$6.696 \cdot 10^{-7}$
8	10 240	$3.458 \cdot 10^{-7}$	$3.458 \cdot 10^{-7}$

Table 5.7: Time-step sizes used by Abaqus during the simulations.

smaller one, used only for the first time increment, and a second one for the rest of the simulation. In this situation, however, the two time-step sizes are not as different as compared to the uniformly loaded circular plate examined in the previous section. This causes an unusual consequence: Abaqus computational times using a given time-step (fixed to be the smaller one of the two) is smaller than that of the adaptive case, where both time-steps are used. This means that more time is lost because of the time-step adaptation process compared to the additional time consumed using slightly smaller time increments. For the sake of comparison Strusol uses, as always, the worst time-step calculated by Abaqus. These time-steps are listed in table 5.7.

The complete simulation lasts 0.35 seconds. Total computational time taken by Abaqus (with both adaptive and fixed time-step) and the GPU implementation in Strusol is summarized in 5.8 and 5.9 in case of double and single precision arithmetic respectively. Time measures were taken in the same way as the test case in the previous section, without including any pre-processing or post-processing time, only the actual computational time. The same testing environment was used: Abaqus was run on a single core of an *AMD Phenom X4 9950 Black Edition* CPU at 2.6 GHz, Strusol in addition used the equipped *NVIDIA GeForce GTX 470* GPU board, provided with 448 CUDA cores (14 SM, 32 cores/SM). All simulations were run on Linux, without interference by background tasks or graphical user interface.

Mesh	Abaqus (adaptive)	Abaqus (fixed)	Strusol GPU
1	57.79	57.99	16.61
2	180.64	179.10	29.87
3	374.93	369.28	38.61
4	1294.34	1277.35	77.44
5	2898.43	2858.18	113.54
6	10 214.82	10 094.71	272.56
7	22 891.90	22 517.00	499.73
8	81 117.21	79 891.72	1755.99

Table 5.8: Computational time in seconds (double-precision arithmetic).

Mesh	Abaqus (adaptive)	Abaqus (fixed)	Strusol GPU
1	38.65	39.23	6.80
2	110.58	108.14	12.26
3	216.55	217.19	18.13
4	709.78	708.25	37.64
5	1541.46	1553.88	47.34
6	5541.72	5504.04	112.27
7	12 297.11	12 316.41	194.08
8	43 813.10	43 300.00	685.66

Table 5.9: Computational time in seconds (single-precision arithmetic).

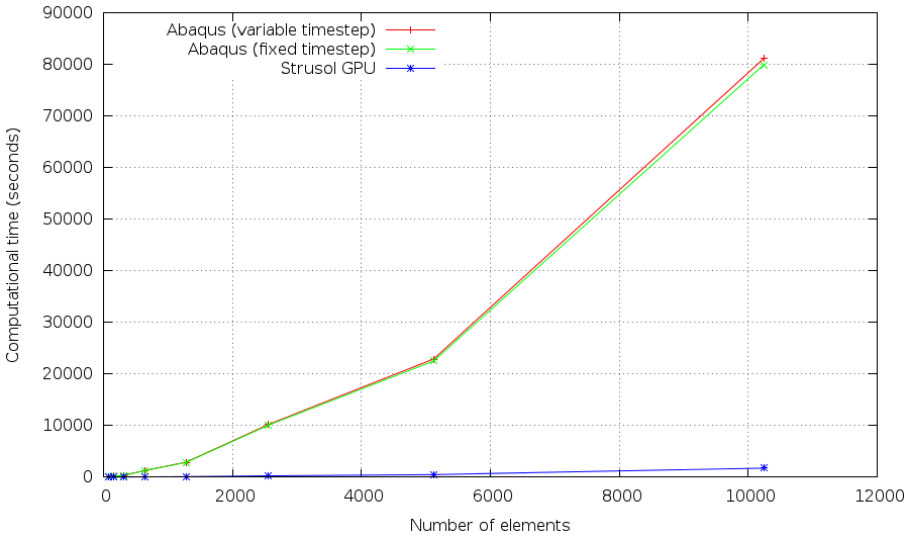


Figure 5.24: Computational time comparison (double-precision arithmetic).

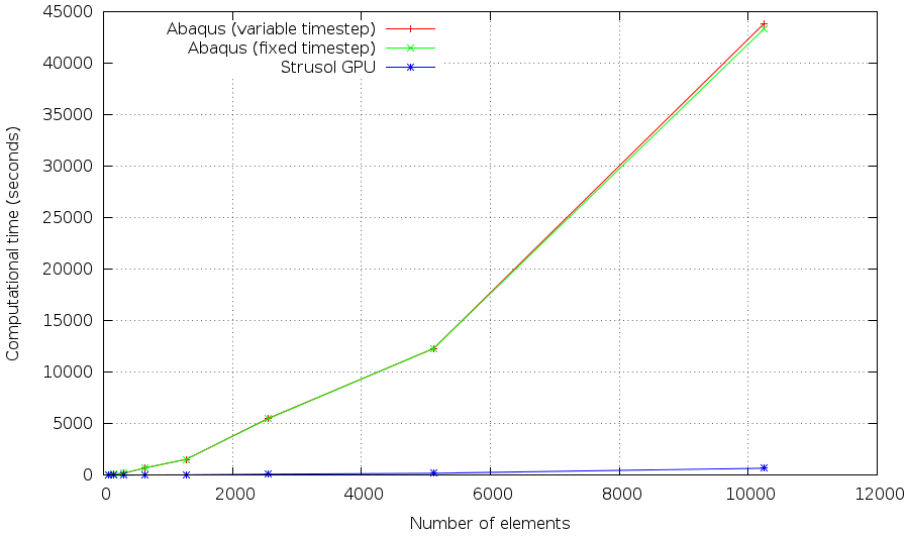


Figure 5.25: Computational time (single-precision arithmetic).

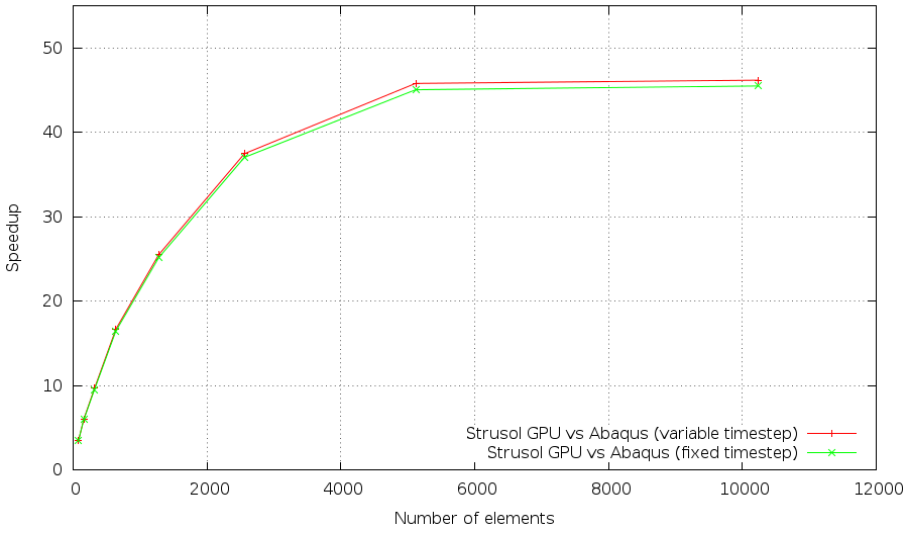


Figure 5.26: Speedup (double-precision arithmetic).

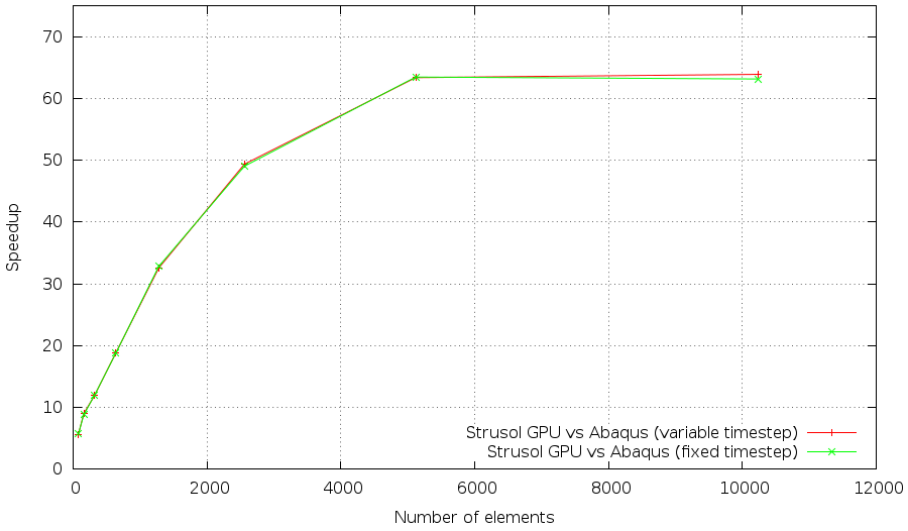


Figure 5.27: Speedup (single-precision arithmetic).

In this simulation, performances achieved by the GPU implementation are even higher if compared to the uniformly loaded circular plate test case: speedup is over 46x if double-precision arithmetic is considered and increases to 63x with single-precision arithmetic. This is probably due to the fact that the easy and regular geometry permits to employ the whole potential of the GPU board and manages to take full advantage of the parallelized algorithm. This is clear also by looking at the speedup graphs in figure 5.26 and 5.27: the software seems to have reached a saturation point, where capabilities of the GPU are used at their maximum and the speedup cannot increase any further. Obviously, these saturation curves depend heavily on the underlying GPU hardware: with computationally-oriented GPU boards they would reach higher peak values.

Chapter 6

Interpolation and mesh-motion libraries

One important ingredient for the final fluid-structure interaction solver is the implementation of interpolation methods. They are used both to transfer data between the fluid mesh and the structural mesh across the interface and to perform mesh-motion. In both these tasks, interpolation can become a serious performance eating process. When domains are refined, interpolation matrices grow, becoming an issue both from the computational time point of view and the overall memory consumption. Therefore it is necessary to study clever implementations of them in order to make the whole FSI solution process lighter and quicker.

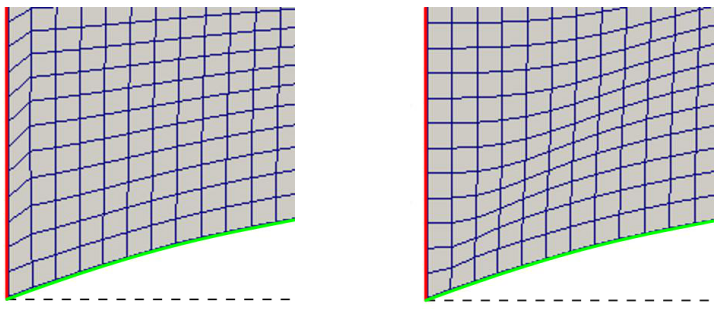
6.1 Mesh-motion solvers

Before going into the description of interpolation algorithm in detail, it is important to briefly summarize the mesh-motion process in OpenFOAM. Its internal design is extremely object-oriented. Therefore, mesh handling is implemented in an extensive hierarchy of C++ classes. The *motion solver* is the class responsible for moving mesh points when necessary. All is needed is to create a child of this class, implement the motion solving algorithm and then load it at run-time as an external shared object: it will be recognized by OpenFOAM as one of the available motion solver classes and chosen to operate, if the appropriate *dictionary* says so. It is also important to notice that, while OpenFOAM includes all the necessary machinery to apply

topological changes to meshes, in this work modifications of domain topology (i.e. addition or deletion of faces and cells, change of node orders, etc.) are not supported. If boundaries move, the mesh can only be adapted by changing the positions of its internal points.

Independently from the underlying interpolation algorithm employed, all implemented motion solvers have the same structure. There is always a set of points that controls the interpolation and a set of points where the interpolation is needed to be computed. Motion is imposed on control points; the interpolator is then called in order to reposition the other points, based on the control points influence.

Control points can be chosen inside the available boundary patches. They obviously should contain the majority (if not all) of the moving points on mesh contours. They can include also some of the fixed points of other patches: this is necessary to prevent motion of points that should not be moved, such as, for example, fixed walls. It also permits a better repositioning of points near those walls. In fact, if the fixed points are included, they contribute to the calculation of the final position of the points near them, lowering the influence of the other control points. An example of this situation is show in figure 6.1, where the red boundary is fixed and the green one is deformable. If points of the red boundary are not included in the interpolation process, internal points are influenced by the green boundary movements only, resulting in loss of cell quality near the red border, as it can be observed in figure 6.1a. If fixed points are included (figure 6.1b), they influence the interpolation too. Therefore, contributes of the green



(a) Without fixed points (b) With fixed points

Figure 6.1: Different interpolation results obtained by the inclusion or non inclusion of fixed points into the control points set.

points are balanced by the red points, resulting in a smoother interpolation (especially considering the points near the fixed boundary).

It is also worth mentioning that a large number of control points does not always lead to better results. Moreover, the size of the interpolation matrix directly depends on their amount, so does the interpolation computational time. The common motion solver class allows the use of a *coarsening factor* on given patches: doing so, not all points of those patches are included into the control points set, but, for each point chosen, a certain number of points is skipped. In this way, the patches influence the motion, while the interpolation matrix is kept small.

When solvers are run in parallel on different processors, the problem is decomposed in smaller domains. Therefore, each node performs mesh motion on its part of nodes, but all nodes require knowledge about the whole set of control points and their movement. It is then necessary, at initialization time, for the master node to broadcast all control points positions to slave nodes and to keep them updated during the simulation.

One last important feature of the motion solver is the ability to reset the interpolators when deformations have become too large: interpolation weights are recalculated and the matrix is rebuilt. In this way, quality of mesh motion is reestablished and the whole system gains in stability and robustness.

In the following paragraphs, implementations of the interpolation algorithms introduced in section 3.1 will be briefly discussed.

6.2 Implementation of RBF interpolation

RBF interpolation has been implemented in two different versions. The first one runs on CPU and is mainly based on the implementation written by the author of [3], included within the OpenFOAM repository. It has been reorganized and slightly optimized. The other implementation has been written from scratch and runs on GPU hardware.

They share a similar structure. At initialization time, the interpolation matrix is assembled based on positions of control points, then it is inverted (using LU factorization). When the interpolator is called, it calculates coefficients α and β and then proceeds with the interpolation.
If used for the sake of mesh-motion, during the initialization stage each potentially movable point is analyzed by means of a *cut-off* function, to check if it lies in the zone where the mesh-motion should apply or not. There is also a *focal point* around which the motion is centered: displacements of internal points are weighted also based on this distance. In this way, it is possible to aggregate more than one interpolator summing afterward their results, making it simpler to cope with boundary movements on different parts of the mesh.

In the GPU implementation matrix assembly is kept on CPU not parallelized, while interpolation is performed in two different CUDA kernels: the first one calculates α and β coefficients and the second one performs the interpolation. The scheme of the two kernels is similar and follows the common skeleton of dense matrix-vector multiplication: each row is treated by a block of 32 threads (size of a warp). Each thread performs the computation on elements with stride equal to 32. After that, a reduction process sum up the partial results and the first thread of the block writes the final result to the output buffer. This algorithm is adapted to perform the RBF interpolation process, including weights calculation and blending. It allows the whole computation to access device memory with coalescence, resulting in a very fast implementation.

The actual RBF interpolator used to transfer quantities across the interface is slightly different and optimized for the FSI solver, but its base remains the same as the motion interpolator implementation.

6.3 Implementation of IDW interpolation

IDW interpolation code is based on the original implementation in [20]. The *boost numeric bindings* (set of bindings provided by the *boost* libraries to interface with BLAS routines) are used to perform matrix operations, in particular the matrix-vector product function "*axpy*".
First of all, the interpolator analyzes initial positions of internal points with respect to the control points, calcolating weights and building the interpolation matrix. Afterward, when it is necessary to carry on the interpolation, it multiplies the matrix by the vector of control points displacements (one component at a time) to obtain the final internal points movement field, relative to their initial positions. As it has already been said, not all control points influence a given internal point, thus the interpolation matrix is extremely sparse, without any recognizable pattern.

As far as the GPU version is concerned, implementation of matrices with sparse storage was needed. We decided not to use an existing library in order to avoid further dependencies to other code and to have full control on the whole process, in order to simplify optimization and debugging. The only needed features are management of an appropriate sparse matrix format and matrix-vector multiplication.

Construction of the matrix is done serially in CPU and it is temporarily stored in *COO* (*coordinate*) format. It is then sorted by rows (using the *quick-sort* algorithm), compressed following the *CSR* (*Compressed Sparse Row*) format and finally arrays with compressed row indexes, column indexes and non-zero values are transferred to GPU device memory for later use.
The interpolation itself consists in just a sparse matrix-vector multiplication and the implementation follows the common scheme outlined in the previous paragraph. Unlike the CPU implementation, the three components of each point are treated at the same time for speed. A structure-of-array approach guarantees coalescing memory accesses, in order not to have performance drops.

6.4 Performance comparison

In order to compare the interpolation algorithms a simple test case has been used. They are compared in a mesh-motion context, as internal interpolation algorithms of the motion-solver.

This is a default case and it is included in the standard OpenFOAM package under the name of *movingBlock*. In the center of a $25\,\mathrm{m} \times 25\,\mathrm{m}$ quadrilateral domain there is a $5\,\mathrm{m} \times 1\,\mathrm{m}$ rectangular hole which moves following a prefixed path. This movement is obtained by an algebraic function and includes translations, rotations and scaling, resulting in an oscillatory movement.

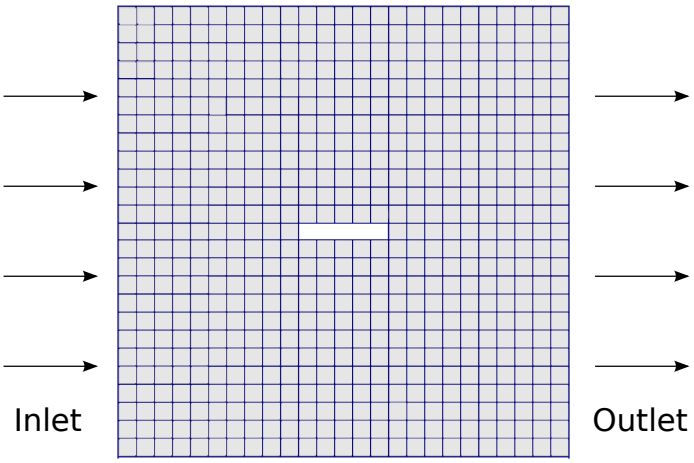
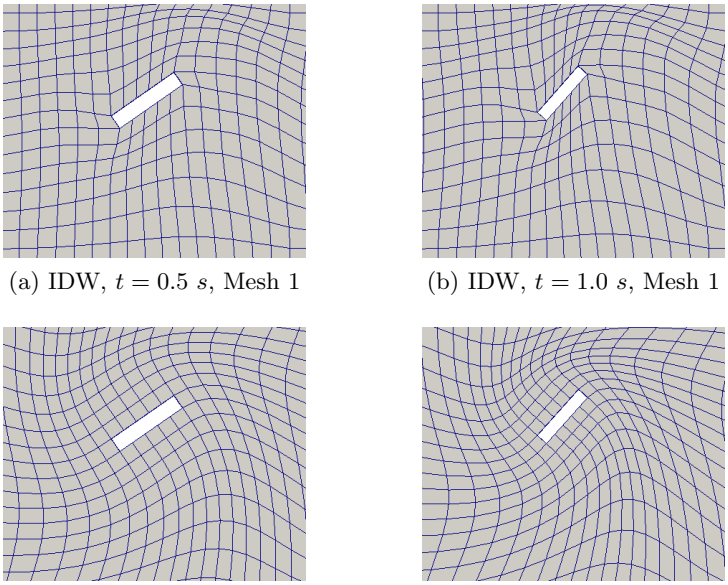


Figure 6.2: *movingBlock* domain.

It is a two-dimensional problem, but it is handled in OpenFOAM as a three-dimensional case, with cells of fixed thickness and appropriate boundary conditions on front and back faces. It is a genuine fluid dynamical case: an incompressible Newtonian fluid enters the domain from the left inlet border and leaves it through the right outlet border. As far as the velocity is concerned, on the inlet a Dirichlet condition is imposed (of $1\,\mathrm{m/s}$), while on the outlet, based on flux direction, a Dirichlet or an homogeneous Neumann condition is used. On top and bottom walls it is fixed to be zero, while on the moving internal walls the special *movingWallVelocity* condition ensures that there is no flux through the faces. About pressure, an homogeneous Neumann condition is imposed everywhere, but on the outlet, where pres-



(a) IDW, $t = 0.5\ s$, Mesh 1

(b) IDW, $t = 1.0\ s$, Mesh 1

(c) RBF, $t = 0.5\ s$, Mesh 1

(d) RBF, $t = 1.0\ s$, Mesh 1

Figure 6.3: Deformed meshes at different times.

sure is fixed to zero. At the beginning of the simulation, all cavity is filled with fluid with constant velocity (magnitude equal to 1 m/s, to the right), while pressure is zero.

Type and amplitude of boundary movements permit to avoid resetting the motion-solver during the simulations (i.e. recalculation of interpolation matrices). In figure 6.3 there is an example of deformed meshes obtained during the simulation using the two different interpolation algorithms. In order to conduct a benchmark on performance different mesh refinements have been used. Information about number of cells and points are listed in table 6.1. Performance has been measured on the same machine used to benchmark

Mesh	Cells	Points
1	620	1352
2	2480	5184
3	5580	11496
4	9920	20288
5	15500	31560
6	22320	45312

Table 6.1: Meshes used in the interpolation benchmark.

Mesh	IDW (CPU)	IDW (GPU)	RBF (CPU)	RBF (GPU)
1	0.00008	0.00011	0.00123	0.00019
2	0.00070	0.00024	0.00755	0.00039
3	0.00228	0.00046	0.02299	0.00078
4	0.00546	0.00077	0.04672	0.00130
5	0.01043	0.00118	0.08668	0.00227
6	0.01807	0.00175	0.14299	0.00337

Table 6.2: Average interpolation times.

the structural solver: a computer equipped with a quad-core *AMD Phenom X4 9950*, 4 GB of RAM memory and a *NVIDIA GeForce GTX 470* video board, with 448 CUDA-cores and 1.28 GB of video memory.

Average times (in seconds) taken to perform the interpolation step only (without the rest of the mesh-motion process) are summarized in table 6.2. Initialization passes are not included in the timings, while memory transfers (from host to device memory and viceversa) are. A graph with speedups obtained thanks to the GPU implementation over the CPU implementation of both algorithms is shown in figure 6.4.

Difference of performance between GPU and CPU implementations increases again with the mesh refinement. The achieved RBF interpolation speedup is higher because the interpolation matrix is dense and so it manages to better exploit the computational power of GPU's floating point arithmetic units. IDW interpolation, on the other hand, needs to deal with sparse ma-

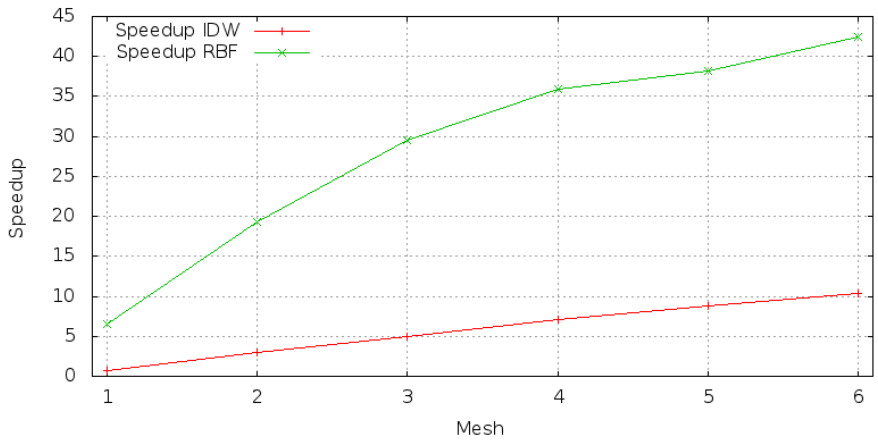


Figure 6.4: Speedups gained by the GPU implementation of interpolation algorithms over the CPU one.

trices, which require indexing arrays and thus memory accesses become a bottleneck. It is important to notice also that BLAS routines employed in the CPU implementation of the IDW algorithm are far better optimized if compared to RBF serial code included in OpenFOAM, so performance difference between GPU and CPU implementations of IDW interpolation is more contained.

In addition, matrix inversion, necessary for RBF interpolation matrix initialization, can be parallelized on GPU hardware too, obtaining significant speedups. In this work however it has not been implemented, since it matters only on the initialization step of the simulation. In cases when the interpolator needs to be reset often, a GPU implementation of the LU factorization and matrix inversion could be another good optimization for the whole final simulation.

6.5 Matrix-free IDW interpolation

After having calculated the interpolation matrix, IDW interpolation consists basically in a matrix-vector multiplication. The matrix has a number of rows equal to the total number of points affected by mesh-motion and a number of columns equal to the number of control points. Therefore, it can be really huge, even if a cut-off function is used in order to reduce its size: for each moving point, only a subset of control points is selected to influence its movement, thus making the matrix sparse. With very big meshes, however, even if the matrix is sparse and the problem is divided among different computational nodes, it can require a large amount of memory (up to several gigabytes).

One solution to overcome this high memory requirement is to avoid the initial matrix construction and re-calculate its coefficient on the fly: for each mesh point potentially affected by mesh motion, contributes of control points to its movement are evaluated at every interpolation and then summed up, to give the final point position. It is thus not necessary to store the whole matrix and memory storage is reduced drastically, at the cost of computational time to reevaluate its coefficients at every interpolation.

Each mesh point movement is independent from the other points. This fact makes this algorithm suitable for GPU implementation. GPUs are much faster doing floating-point operations than accessing memory, so this kind of

storage-less approach is ideal to be run in device kernels. One or more points are assigned to each thread, making it possible to exploit shared memory to efficiently store control points, which are common to all threads and read with fully coalesced accesses, while hiding memory latency with the good amount of calculations involved.

As optimization, the denominator in IDW interpolation formula is pre-computed for each mesh point. This needs a negligible amount of extra memory, but increases overall performance.
Therefore, the whole computation is divided in two passes: an initialization step, when the denominators are calculated, executed only once, and then the interpolation step itself. Each pass is performed by a separate CUDA kernel. Denominators are transferred only once and kept in GPU memory.

The original IDW algorithm requires an initial step too, in order to build the interpolation matrix. This step is quite expensive since it needs to allocate and fill the sparse matrix, which, as it is stated above, can be very big.

6.5.1 Performance comparison

In order to compare the different IDW implementations a simple test case has been built: a cubical mesh undergoes random displacements of some of its border faces and internal points must adapt. Different refinements of the mesh has been considered; they are listed in table 6.3, together with the number of moving points (equal to the total number of mesh points) and the number of control points used. This test has been conducted on a IBM PLX-GPU hybrid supercomputer, owned by CINECA[1]. Computational nodes are equipped with 2 six-cores Intel Westmere CPUs, 2.40 GHz, 48 GB of RAM and 2 NVIDIA Tesla M2070 GPU boards, with 448 CUDA cores and 5 GB of memory, each. They are linked together by an InfiniBand[2] network, which assures very low inter-node latency. Results of benchmark are reported in table 6.4. It is worth noting that the supercomputer is shared between different users, thus performance analysis cannot be very accurate: since nodes are shared, if a user suddenly starts his own job on the same nodes used to measure timings, it could cause a performance drop, which would be reflected on benchmark results (especially when GPUs are used by

[1] `http://www.cineca.it`
[2] `http://www.infinibandta.org`

Mesh	Cells	Moving points	Control points
1	1000	1331	484
2	4096	4913	1156
3	8000	9261	1764
4	17 576	19 683	2916
5	27 000	29 791	3844
6	46 656	50 653	5476
7	64 000	68 921	6724
8	97 336	103 823	8836
9	125 000	132 651	10 404

Table 6.3: Meshes considered for the benchmark.

multiple users performances change drastically).

Moreover, timings have been measured considering the interpolation process only and not the whole mesh-motion machinery. They don't include the initialization step, performed before the first interpolation. The column *ATLAS* refers to the standard IDW implementation in CPU, introduced in the previous section: it calculates and stores in memory the whole interpolation matrix using *boost* compressed sparse matrix class (which stores values in CSR format) and uses the linear algebra library ATLAS to perform quick sparse matrix-vector multiplication. Columns "*No-memory-CPU*" and "*No-memory-GPU*" refer to matrix-less IDW implementation, executing in CPU and GPU respectively. Timings show clearly how the matrix-free approach in CPU is significantly slower than the standard one and it is worth using

Mesh	ATLAS	No-memory-CPU	No-memory-GPU
1	0.0034	0.0192	0.0012
2	0.0681	0.1696	0.0047
3	0.1979	0.5038	0.0120
4	0.5136	1.7044	0.0392
5	1.0554	3.3998	0.0771
6	3.0783	8.3047	0.1841
7	5.0542	13.8712	0.1857
8	5.5371	27.2803	0.3640
9	8.2519	41.0152	0.5167

Table 6.4: Timings in seconds.

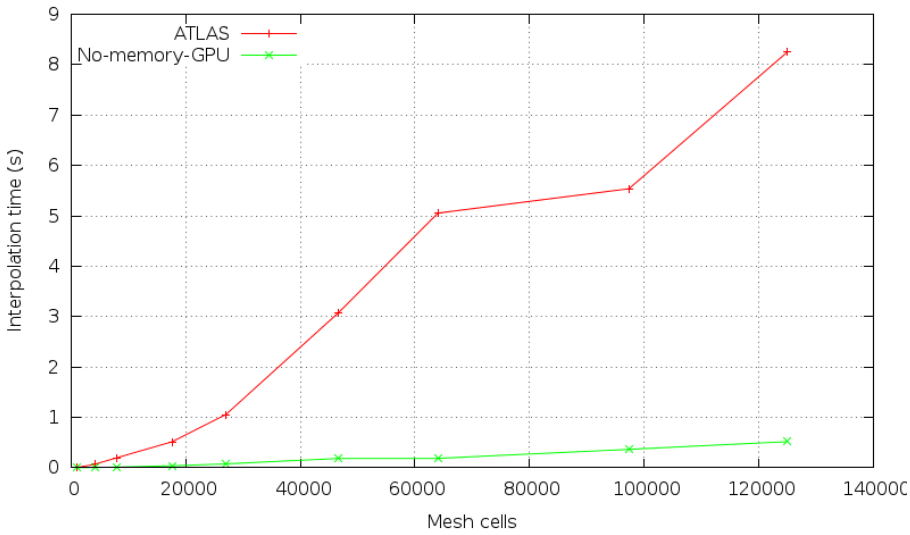


Figure 6.5: Comparison of interpolation times.

it only if memory consumption is a serious bottleneck. On the other hand, GPU implementation shows a significant speedup against the standard one; this contributes to prove the fact that often redundancy in calculations is better than storing pre-computed values in memory, as far as GPU programming is concerned.

A comparison of interpolation times of the standard CPU implementa-

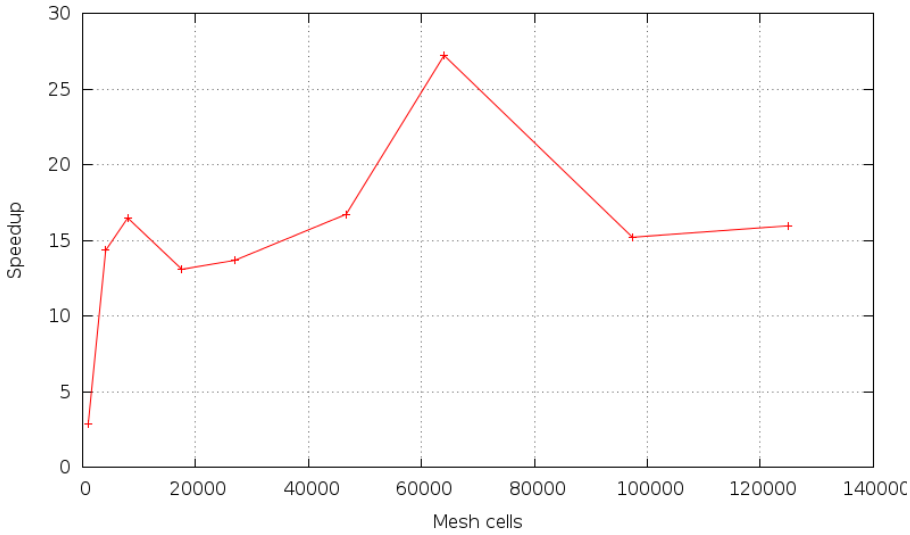


Figure 6.6: Speedup achieved by the GPU matrix-free implementation.

tion using ATLAS versus the matrix-free GPU implementation is shown in figure 6.5, while figure 6.6 shows the gained speedup.

The standard implementation needs to build the sparse matrix in its initialization step. This includes the allocation and compression of the big sparse matrix and can take a lot of time. The matrix-free approach instead ideally does not require any initialization step. The GPU implementation, however, requires CUDA to be initialized, GPU buffers to be allocated and filled with moving and control points coordinates and distances to be pre-calculated. In table 6.5 initialization step timings are summarized.

The best advantage of using the matrix-free approach instead of the standard one is the much lower memory requirement. A very brief analysis can be performed considering that the standard implementation substantially needs to store the sparse interpolation matrix. It has not a precise pattern, thus used memory depends strictly on the number of non-zeros which need to be stored. The matrix-free implementation instead does not require any memory as far as the CPU version is concerned. The GPU version needs some buffers to store moving points, control points, control field (movement of the control points), resulting fields and pre-computed distances in GPU memory (and some of them with the respective counterparts in CPU memory). Table 6.6 shows a rough estimation of memory usage. Double-precision floating-points have been considered. It is important to note that memory consumption of the standard implementation reported in the table considers the storage for the values only. It is a sparse matrix, so storage for item coordinates also takes a lot of memory, but it depends on the actual pattern

Mesh	ATLAS	No-memory-GPU
1	0.1071	0.24944
2	1.0178	0.23751
3	2.9488	0.20404
4	10.5715	0.24077
5	21.5050	0.24971
6	54.7335	0.32547
7	89.9693	0.33543
8	174.0089	0.49501
9	260.9821	0.51708

Table 6.5: Initialization timings in seconds.

Mesh	Non-zeros	Standard impl.	GPU matrix-free impl.
1	845 816	6.45 MB	95.48 KB
2	7 984 400	60.92 MB	322.87 KB
3	23 498 624	179.28 MB	589.15 KB
4	84 391 528	643.86 MB	1.21 MB
5	169 965 144	1.30 GB	1.81 MB
6	415 866 976	3.17 GB	3.03 MB
7	698 579 360	5.33 GB	4.08 MB
8	1 391 673 832	10.62 GB	6.09 MB
9	2 100 236 680	16.02 GB	7.74 MB

Table 6.6: Theoretical memory consumption.

of non-zero values and the kind of compression used. Therefore, values in the table must be considered as (very) lower bounds of the actual required memory.

In table 6.7 actual measures of memory consumption are reported. Values include all memory allocations performed from the class constructor to the end of an interpolation call. Memory consumption of the CPU standard implementation is much bigger if compared to theoretical values, since this time they take into account also the element indices inside the sparse matrix (even if the matrix is compressed in CSR format) and every other memory overhead needed by the interpolation class to operate. Values regarding the GPU matrix-free implementation follow more or less the theoretical predictions, but with a constant overhead of about 26 MB, probably due to CUDA internal allocations.

Mesh	Standard impl.	GPU matrix-free impl.
1	13.28 MB	26.07 MB
2	124.97 MB	26.17 MB
3	367.54 MB	27.33 MB
4	1.32 GB	27.63 MB
5	2.66 GB	28.97 MB
6	6.50 GB	30.63 MB
7	10.92 GB	34.28 MB
8	21.75 GB	35.29 MB
9	32.82 GB	36.28 MB

Table 6.7: Actual memory consumption.

These results show the importance of this kind of approach. When running on a cluster, shared with other users, memory can become a serious problem. MPI processes can usually access only a certain amount of memory each; if more memory is needed, it needs to be requested explicitly when the job is launched, which means higher cost per hour. The approach presented in this section shows that, if GPUs are available, they can be used not only to decrease the computational time but also to lower the amount of memory needed. In this way, overall resource consumption is reduced.

Chapter 7

Fluid-structure interaction solver

In this chapter the developed solver for fluid-structure interaction problems is introduced. In section 7.1 a simplified version of the source code is presented and commented in its parts. Afterward, the solver is employed in two FSI cases: a cube cavity with flexible bottom, in section 7.2, and a sail simulation, in section 7.3, where performances are compared to the single-sail transient solver presented in [20].

7.1 Implementation

The full fluid-structure interaction solver combines all modules described in the previous chapters in order to couple the structural and fluid problems together. In this section the whole process will be described. The solver's *main* function is explained in details with the aid of a C++-like pseudo-code, with some OpenFOAM and MPI constructs. Although it is not the real source code, it shares the same structure.

```
1 #include "setRootCase.H"
2 #include "createTime.H"
3 #include "createDynamicFvMesh.H"
4 #include "readCouplingProperties.H"
5 #include "createFields.H"
6 // ... other initialization code ...
```

First of all, OpenFOAM fluid dynamic case is initialized: parameters are read from dictionaries, the main classes which manage the dynamic finite volume mesh and time integration are created, the main fluid fields are built.

```
7  // initialize the structural problem
8  if (isMasterNode() == true) {
9      shellProblem = new ShellProblem()
10     shellProblem->createStructuralMesh()
11     // ... structural problem initialization ...
12 }
13 broadcast(shellProblem->getMeshNodes())
```

The structural problem is initialized on the master computational node. The structural mesh can be built in different ways:

- loading an external file or importing mesh and settings from an Abaqus job; this will require future interpolations between grids.
- treating a fluid mesh boundary as an elastic wall, which gets directly deformed by the structural solver.
- using a couple of zero-thickness internal patches as double-sided walls, either discretized using elements corresponding to the (quadrilateral) fluid faces or with a different set of elements created ad-hoc.

All computing nodes need to have the structural mesh points. Therefore, they are broadcasted from the master node after having built the mesh.

```
14 #include "createStressFields.H"
15 #include "readTimeControls.H"
16 #include "initParametersFSI.H"
17 #include "initContinuityErrs.H"
18 #include "createInterpolators.H"
```

Initialization follows with the creation of all vectors necessary for the structural problem to manage displacement interpolation and relaxation. Other parameters controlling time advancement and FSI convergence tolerances are read. Then the variable which will hold the cumulative continuity error measure is initialized. Finally, displacement and stress interpolators are created; this process is shown more in details in the next box of code.

createInterpolators.H

```
// ...
RBFInterpolation interpolator (
    // dictionary with parameters
    couplingProperties.subDict("RBFInterpolator"),
    // structural points, acting as control points for the interpolation
    structuralPoints,
    // fluid mesh face centres at the interface
    mesh.boundaryMesh()[fluidPatch].faceCentres(),
    // initial mesh-motion point positions
    pointDisplacements.boundaryMesh()[fluidPatch].localPoints(),
    // ... other parameters ...
)
// ...
```

RBF interpolator is inizialized. Structural points are used as interpolation control points. Considering the interpolation of stresses, destination points are located in fluid mesh face centres. When interpolation of displacements is considered instead, destination points correspond to the initial mesh-motion point positions (in the *pointDisplacements* field).
Returning to the *main* function, the initialization process continues:

```
if (isMasterNode() == true) {
    // finalize structural problem initialization
    shellProblem->finalizeSetup(structUseGpu, ... )
    // calculate deltaT
    scalar structDeltaTCFL =
            shellProblem->calculateCFLDeltaT() * structCFLFactor
    shellProblem->setDeltaT(structDeltaT)
    // set time
    shellProblem->setCurTime(runTime.value())
    // restarting from a previously calculated solution?
    if (restart == true) {
        shellProblem->readRestart(runTime.timeName(), ... )
    }
    // save initial state
    shellProblem->saveState()
}
```

Structural problem initialization is finalized: all necessary GPU memory buffers are created and populated with data and the time-step to use is obtained from an appropriate CFL condition and multiplied by a conserva-

tive safety factor. If necessary, a previously computed solution is loaded, to continue from a simulation already started. The full initial shell problem state (with all buffers) is saved. States will be saved/restored during the FSI iterative process.

```
// setup Aitken relaxation
AitkenRelax aitkenRelax(defaultRelaxFactor, ... )
// save structural displacements
vectorField structDisplacementsInitial(structDisplacements)
Info << "Problem initialized" << endl
```

Finally, the whole FSI problem initialization step is finalized: Aitken relaxation class is initialized with a default relaxation factor and initial structural displacements are saved. This last buffer is necessary to perform future mesh-motion re-initializations, necessary to cope with large structural deformations.

Now the simulation main loop can be summarized:

```
// main runtime loop
while (runTime.run()) {
    // OpenFOAM time step initialization
    #include "readControls.H"
    #include "CourantNo.H"
    #include "setDeltaT.H"
    // advance time
    runTime++;

    // does the structure have to remain fixed?
    if (runTime.value() > structFixTime) {
        // save initial fluid fields for FSI iterative process
        saveFluidFields()
        // start FSI sub-iterations loop
        int subcycle = 0
        do {
            // restore previous fields
            if (subcycle > 0) {
                restoreFluidFields()
                if (isMasterNode() == true)
                    shellProblem->restoreState()
            }
            // interpolate stresses
            #include "transferStresses.H"
```

```
42              // run structural problem on master node
43              shellProblem->run(runTime.deltaT().value())
44              // interpolate displacements
45              #include "transferDisplacements.H"
46              // mesh-motion
47              #include "setMotion.H"
48              // solve fluid problem
49              #include "solveFluid.H"
50              // calculate residuals
51              #include "checkFSIConvergence.H"
52          }
53          while (isConverged() == false);
54          // update structural solver internal state
55          if (isMasterNode() == true) {
56              shellProblem->endOfSubcycles()
57              shellProblem->saveState()
58          }
59      } else {
60          // structure is fixed, solve the fluid problem only
61          #include "solveFluid.H"
62      }
63      // end of global time-step
64      runTime.write()
65      // ... output other results and restart files ...
66  }
67  Info << "End" << endl
```

The structure of the main loop is straightforward. After time-step initialization, current time is checked: if it is below a certain preset value, then the structural problem is not solved. This helps to stabilize the fluid solution before starting the real FSI problem. If time is above *structFixTime*, the FSI sub-iterations loop is entered. Sequence of operations follows the outline given in chapter 3: stresses are transferred from the fluid mesh to the structural mesh at the interface, then the structural problem is solved on the master node, displacements are transferred and interpolated, the fluid mesh is deformed by the interpolated relaxed displacement field and finally the fluid problem is solved. Checks on solution increments between sub-cycles are performed in order to determine when the FSI coupling has converged. Finally, solution fields are dumped to file and the simulation can continue with the next time-step.

Details about the implementation of the single internal steps are now described.

transferStresses.H

```
// evaluate fluid stresses
interfaceFluidStresses =
    rhoFluid.value() * p.boundaryField()[fluidPatch]
    * p.boundaryField()[fluidPatch].patch().magSf()
    * p.boundaryField()[fluidPatch].patch().nf()
// perform RBF inverse interpolation
structPatchStresses = interpolator.reverseInterpolate(interfaceFluidStresses)
// sum up contributes from each computational node
reduce(structPatchStresses, sumOp<vectorField>())
// set external force field in structural problem
if (isMasterNode() == true) {
    shellProblem->setExternalForces(structPatchStresses)
}
```

In *transferStresses.H* fluid stresses at interface are calculated first, using the fluid pressure field, then they are interpolated on the structural mesh. Each node has computed the stresses on its own domain: the *reduce* operation on the master node is necessary to gather all contributions from the other nodes and to sum them. Finally, the master node sets the resulting stresses as external loads for the structural problem.

transferDisplacements.H

```
/// get displacements
if (isMasterNode() == true) {
    structDisplacementsNotRelaxed = shellProblem->getPointDisplacements();
}
// broadcast them to the other computational nodes
broadcast(structDisplacementsNotRelaxed)
// Aitken relaxation
aitkenRelax.update(subcycle, structDisplacementsNotRelaxed)
structDisplacements = aitkenRelax.getRelaxedField()
```

transferDisplacement.H is included after the structural problem is solved. Resulting displacements are firstly copied from GPU memory to a CPU buffer, then they are broadcasted to all computational nodes. The Aitken relaxation class manages the internal double-buffering and residual fields

necessary to perform the adaptive under-relaxation of displacements at the interface.

setMotion.H

```
// check if a reset of the motion−solver is needed
if (max(magnitude(structDisplacements − structDisplacementsInitial)) > threshold) {
    // reset motion−solver
    motionSolver.reset()
    // save new initial displacements
    structDisplacementsInitial = structDisplacements
}
// interpolate displacements
fluidDisplacements = interpolator.interpolate(structDisplacements)
// set motion field
pointDisplacements.boundaryField()[fluidPatch] == (
    fluidDisplacements
)
// update mesh
mesh.update()
// update continuity error measure
#include "volContinuity.H"
```

In *setMotion.H* all the code concerning the mesh-motion process is included. Initially, the maximal structure deformation is checked to see if a too large deformation occurred and it is necessary to reset the motion-solver: in such case, interpolation matrices are recalculated using the new positions of structural mesh points as control points, so that future interpolations are smoother. Displacements are then interpolated on the fluid mesh at the interface and set as control motion field for the mesh-motion process. The fluid mesh is then updated on all computational nodes.

The fluid solver implemented in *solveFluid.H* is very similar to the implementation in *pimpleDymFoam*, shipped by default with OpenFOAM, thus a detailed description is not necessary in this context.

7.2 Cavity with flexible bottom test case

In order to check the FSI coupled solver, a classical benchmark test case has been used: a three-dimensional cubic cavity with flexible bottom is filled by a pulsating flow field, imposed by the boundary condition on the top part of the cavity. This problem has already been proposed in [22, 23] in two dimensions, and in [34, 20] in three dimensions.

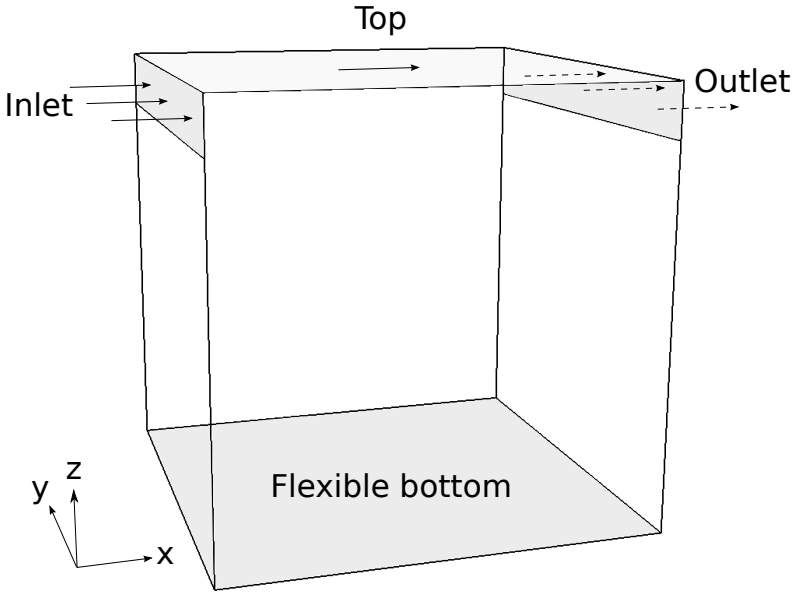
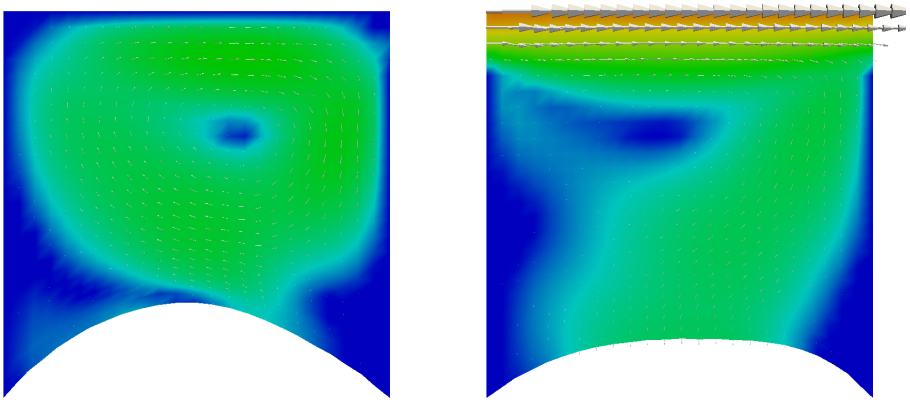


Figure 7.1: 3D cavity problem setup.

A $1\,\mathrm{m} \times 1\,\mathrm{m} \times 1\,\mathrm{m}$ cube is filled with a fluid with density of $1\,\mathrm{kg/m^3}$ and viscosity equal to $0.01\,\mathrm{m^2/s}$. A small inlet area ($1\,\mathrm{m} \times 0.125\,\mathrm{m}$) lets the fluid enter the cavity, aligned on the x direction, with the following time-dependent linear velocity profile:

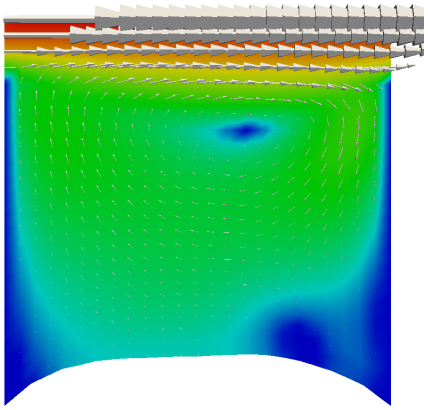
$$v\,(z) = \bar{v} \cdot \frac{(z - 0.875)}{0.125} \qquad \bar{v} = 1 - \cos\frac{(2\pi t)}{5}, \qquad 0.875 \leq z \leq 1. \tag{7.1}$$

A small outlet area of the same size lets the flow exit the domain. The top boundary imposes velocities equal to $\bar{v}$. On all the other lateral walls velocity is imposed to be zero, while on the flexible bottom the *movingWallVelocity* condition guarantees that the fluid velocity is equal to the boundary velocity,

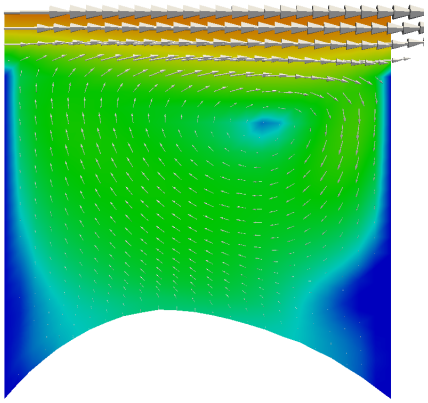


(a) $t = 10\,\mathrm{s}$

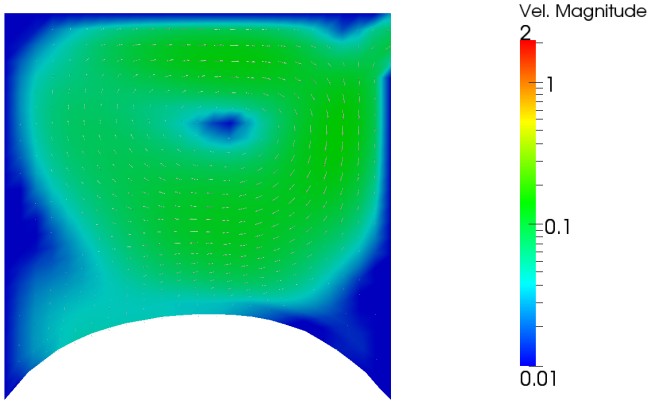
(b) $t = 11.25\,\mathrm{s}$



(c) $t = 12.5\,\mathrm{s}$

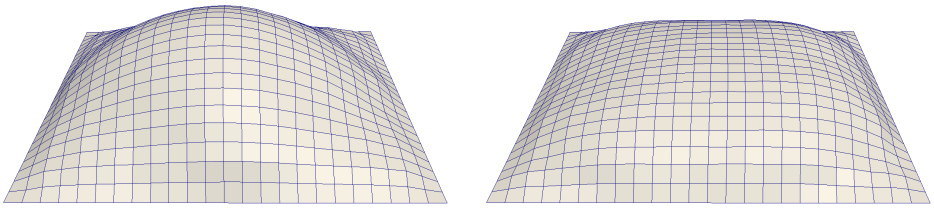


(d) $t = 13.75\,\mathrm{s}$

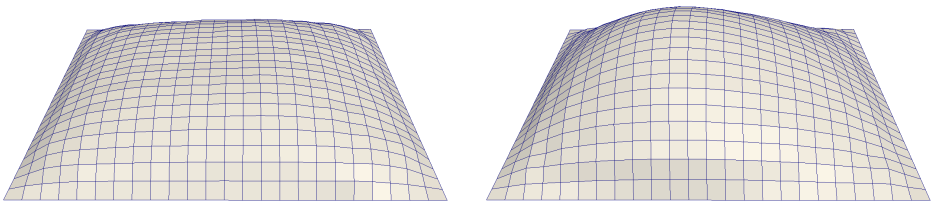


(e) $t = 15\,\mathrm{s}$

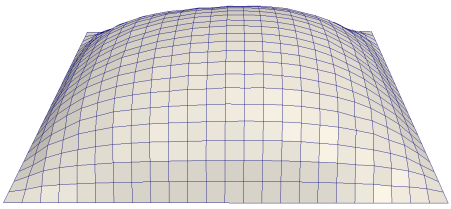
Figure 7.2: Velocity magnitude field on the X-Z plane at different times. Values are shown in logarithmic scale.



(a) $t = 10\,\mathrm{s}$ (b) $t = 11.25\,\mathrm{s}$



(c) $t = 12.5\,\mathrm{s}$ (d) $t = 13.75\,\mathrm{s}$



(e) $t = 15\,\mathrm{s}$

Figure 7.3: Bottom wall deformations at different times.

Index	Elements per side	Total elements	Points
1	8	512	729
2	16	4096	4913
3	24	13 824	15 625
4	32	32 768	35 937
5	40	64 000	68 921

Table 7.1: Different meshes used.

and flux across the patch is zero. On the outlet patch, zero pressure is imposed. Different meshes have been considered (details in table 7.1). In each case, the solver extracts the bottom patch from the fluid mesh and setups the solid shell problem directly using it. In this way, the structural grid and the fluid one are conforming and no interpolation between them is needed. Mesh-motion is carried out employing the GPU implementation of RBF interpolation. In this case it produces the best quality mesh with overall low computational effort.

In figure 7.2 velocity magnitude field is shown at different times, referring to the simulation with 24x24x24 cells and FSI time-step of 0.1 s. In figure 7.3 evolution of bottom wall deformation is shown. When compared with values presented in [34], a very similar trend is observed in the two solutions (figure

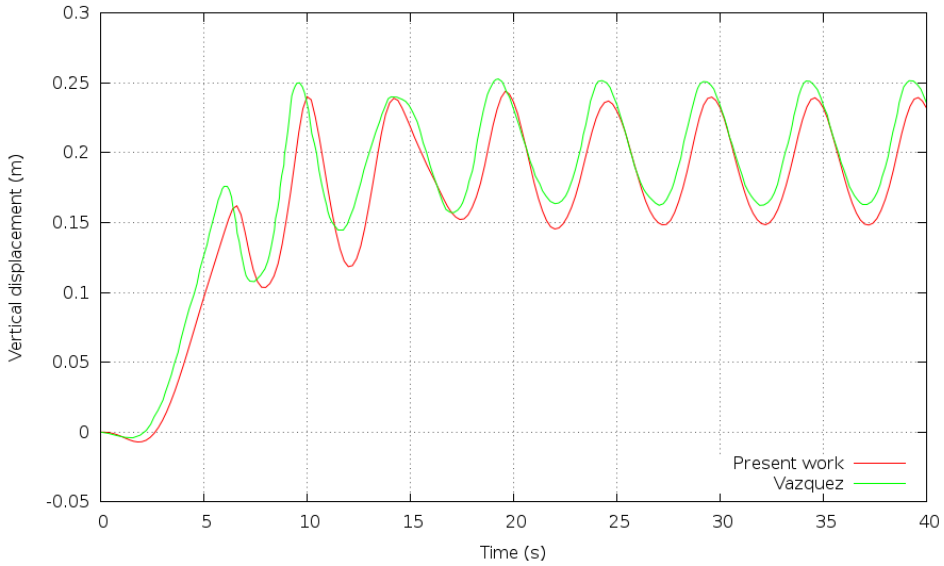


Figure 7.4: Comparison with numerical results reported in [34].

7.4). There is a small difference in the mean value, probably due to non matching boundary conditions or the different structural elements employed; it is also worth noting that even values regarding the bidimensional case presented in [34] have a slightly higher mean value than values in [22], so it may be just a tendency in his FSI setup.

Tests have been made with different FSI time-steps also. Time-steps considered are 0.2 s, 0.1 s, 0.05 s and 0.025 s. With $dt = 0.5$ s, the simulation stops after a few seconds because from there on FSI sub-iterations do not converge anymore. Vertical displacements of the central point of the bottom wall are extracted from the solutions and used as reference: results of simulations with different mesh refinements and different time-steps are shown in figure 7.5 and 7.6 respectively. The resulting curves converge to a mesh and time-step independent solution.

In order to have the FSI inner loop to converge more quickly, Aitken under-relaxation has been used on the deformations of the bottom wall. It helped a lot in lowering the amount of sub-iterations needed to reach convergence between the fluid and solid problem at each time-step. Results show that there is an (obvious) correlation between the time-step size and the number of sub-iterations performed. In figure 7.7 the amount of sub-iterations performed in each time-step is shown, for four different time-steps and two mesh refinements. When time-step size is lowered, the amount of sub-iterations needed to reach convergence decreases. This is expected, since with smaller time-steps there are also smaller differences between solutions of successive time-steps, thus the coupling is easier to be enforced. A similar relation is not observed considering instead the level of mesh refinement employed; especially when the time-step is small, the different meshes behave similarly, requiring a similar amount of sub-iterations to converge.

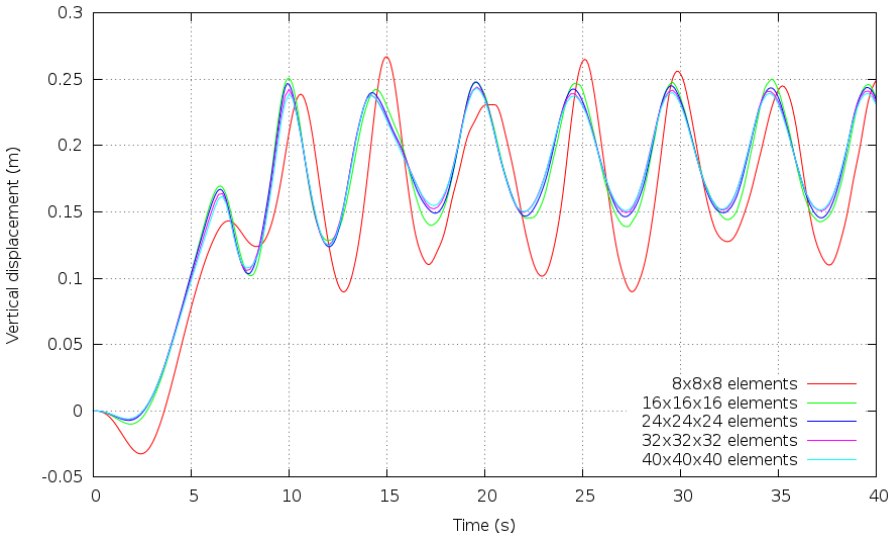


Figure 7.5: Solutions with different mesh refinements ($dt = 0.1$).

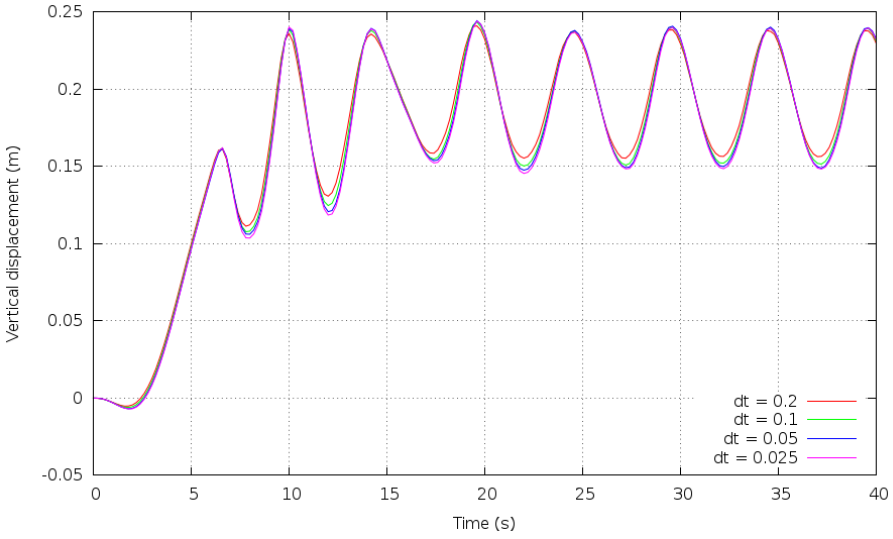
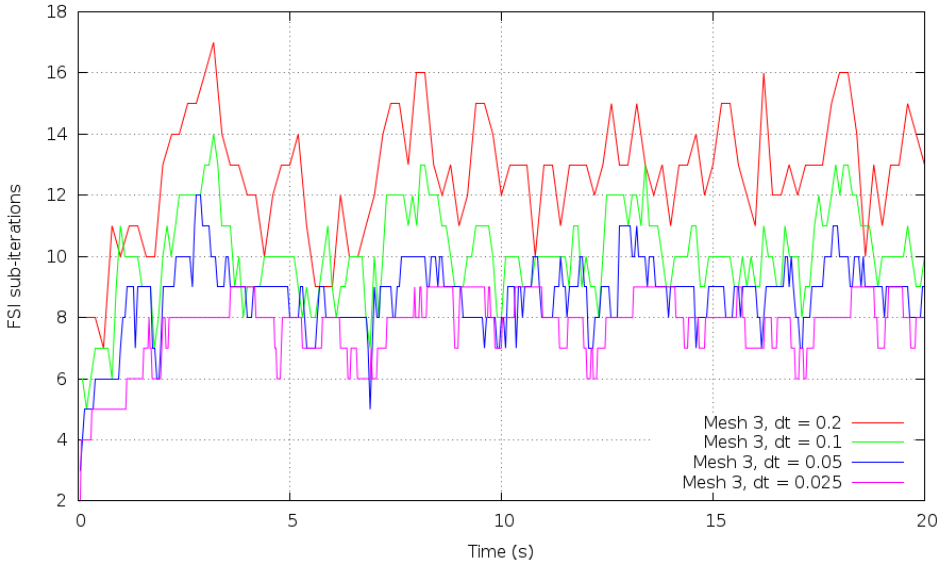
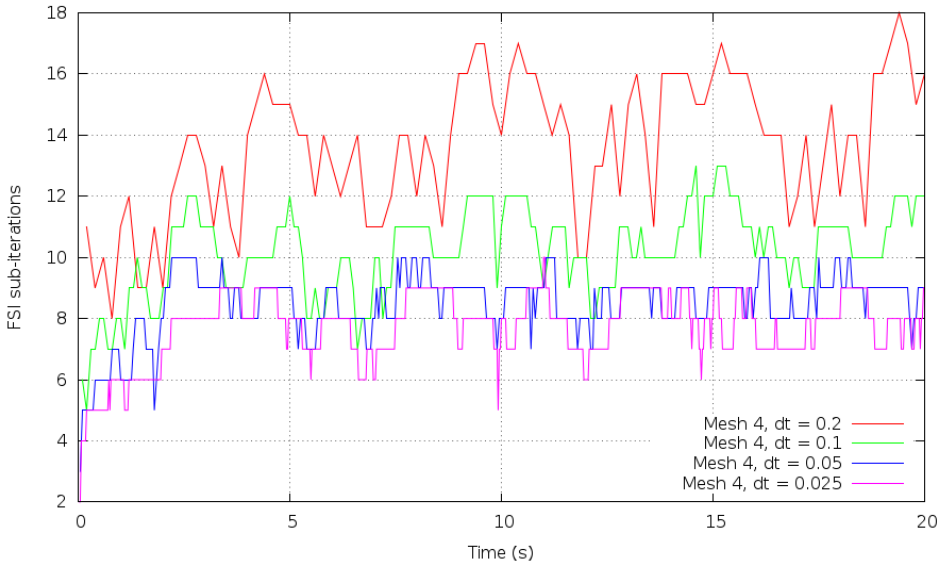


Figure 7.6: Solutions with different time-step sizes (on finest mesh).



(a) Third mesh (13 824 elements)



(b) Fourth mesh (32 768 elements)

Figure 7.7: Dependence of the amount of FSI sub-iterations on FSI time-step size.

7.3 Gennaker sail simulation

In this section, numerical results obtained by solving a more complex FSI case are reported. Performances of the solver at hand have been compared against the FSI solver used in [20], with details about the speedup obtained by the presented solver in the FSI sub-iterations.

A gennaker sail is immersed in a rectangular prism domain, filled with air. A view of the mesh is shown in figure 7.8. The bottom boundary acts as the water surface. Air flows into the domain from the lateral boundaries. The sail is made of an isotropic elastic material with density of $100\,\mathrm{kg/m^3}$, Young modulus equal to $3.76 \times 10^8\,\mathrm{N/m^2}$ and Poisson ratio equal to 0.3. Thickness of the sail is $0.001\,\mathrm{m}$. Air density is set to $1\,\mathrm{kg/m^3}$ and viscosity to $1.5 \times 10^{-5}\,\mathrm{m^2/s}$. A $k - \omega$ model is employed in order to account for turbulence. Fluid velocity is fixed by a Dirichlet condition on the bottom boundary; lateral boundaries act like inlets (fixed Dirichlet condition) if the

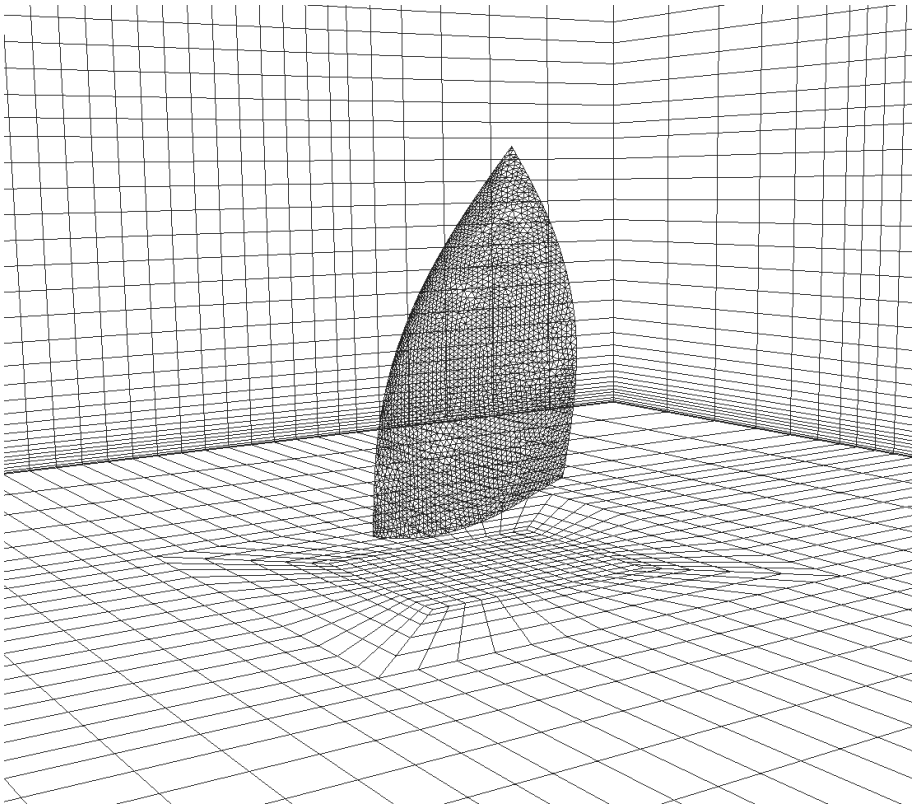


Figure 7.8: Particular of the fluid mesh.

flow is inward or outlets (homogeneous Neumann condition) if the flow is outward. A slip condition is imposed on the sky and the *movingWallVelocity* condition on the gennaker (flux across the sail is set to zero). Pressure value is fixed to zero on outlets as reference; on the other patches its gradient normal to the boundary is zeroed. Finally, appropriate wall conditions are applied to k and ω on all patches.

The velocity profile at inlets is the composition of wind velocity, which reaches zero at sea level when the sea is at rest, and the inverse of boat velocity: this results in a twisted profile, shown in figure 7.9.

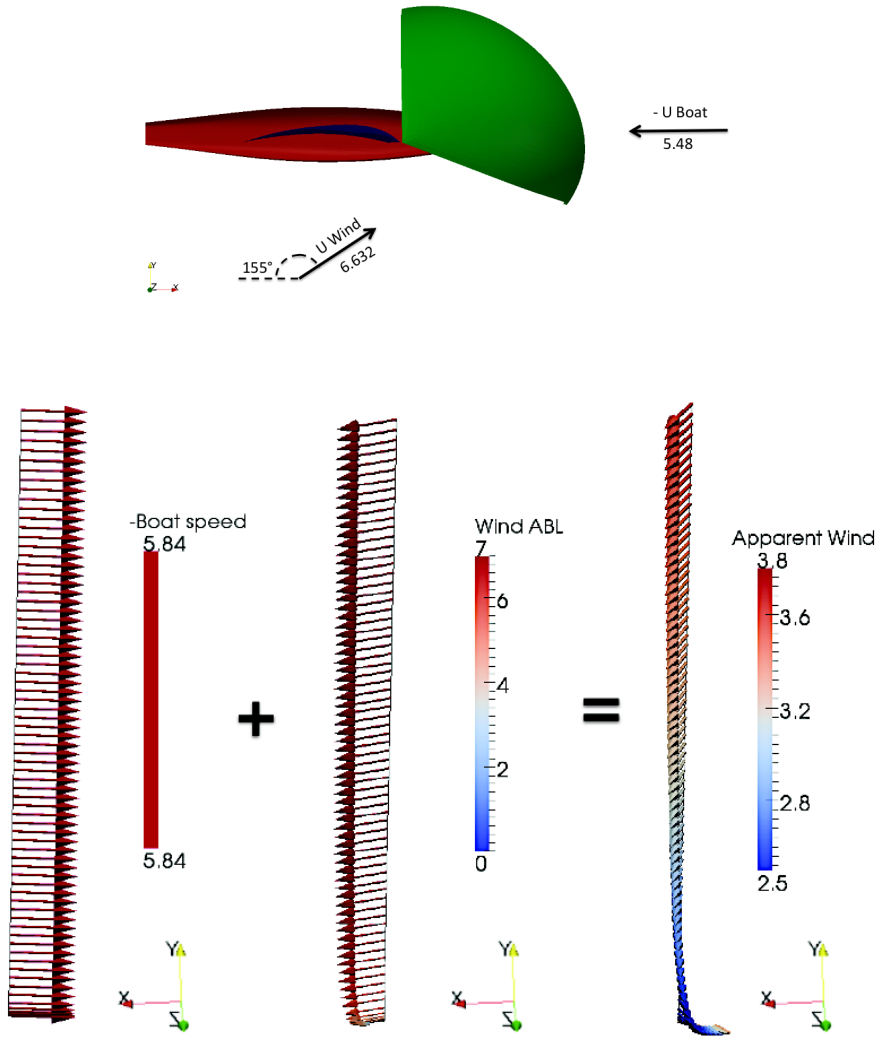
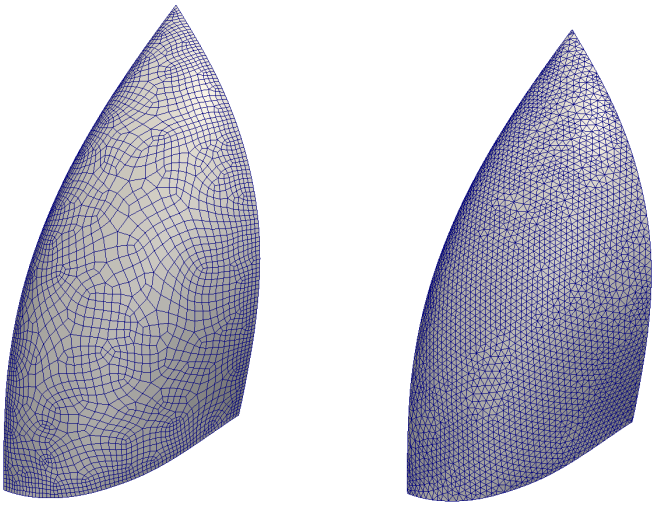


Figure 7.9: Velocity profile imposed at inflow boundaries.

Unlike the cube cavity case shown in the previous section, the sail structural problem is solved on a separate mesh. The two problems are coupled with the aid of RBF interpolation in order to exchange values between the two grids across the interface. A comparison of the structural quad mesh and the corresponding fluid mesh interface is visible in figure 7.10. The fluid mesh is composed of 216 571 cells, with 81 037 points, while the structural one is made of 2694 elements, with 2846 nodes.

The sail is fixed at its three corners: their translational dofs are set to zero, while the other rotational dofs are not fixed. A more realistic simulation would employ a trimming sheet attached to one of the bottom sail corners, in order to simulate sail opening. In this work, performance of the solver is the main concern, therefore the adopted boundary conditions are sufficient; implementation of better conditions would have no impact on performance and remains to be done in future releases.

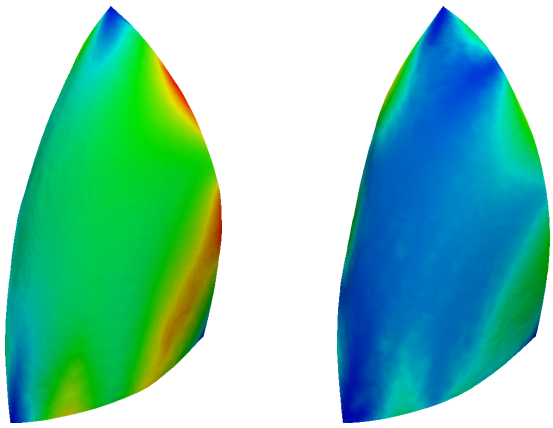


(a) Sail structural mesh (b) Sail fluid mesh

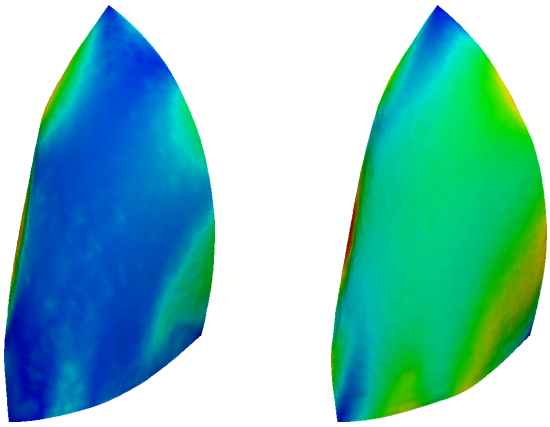
Figure 7.10: Structural sail mesh and fluid mesh.

As initialization step, for 10 seconds of simulation, only the fluid problem is solved, while the sail is kept fixed. A slice view of the resulting velocity field after 3 seconds of full FSI simulation is shown in figure 7.12.

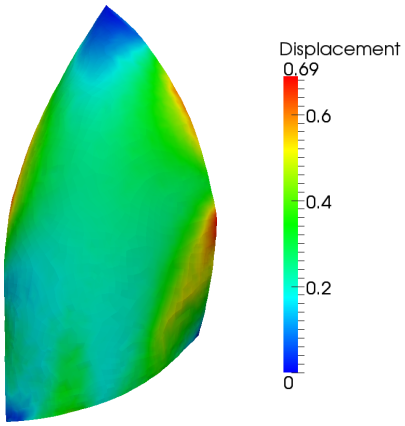
The FSI solver implemented in this work has been compared in terms of performance against the FSI solver used in [20] to perform single-sail sim-



(a) $t = 0.7\,\mathrm{s}$ (b) $t = 1.58\,\mathrm{s}$



(c) $t = 2.85\,\mathrm{s}$ (d) $t = 3.32\,\mathrm{s}$



(e) $t = 3.475\,\mathrm{s}$

Figure 7.11: Sail displacements at different times.

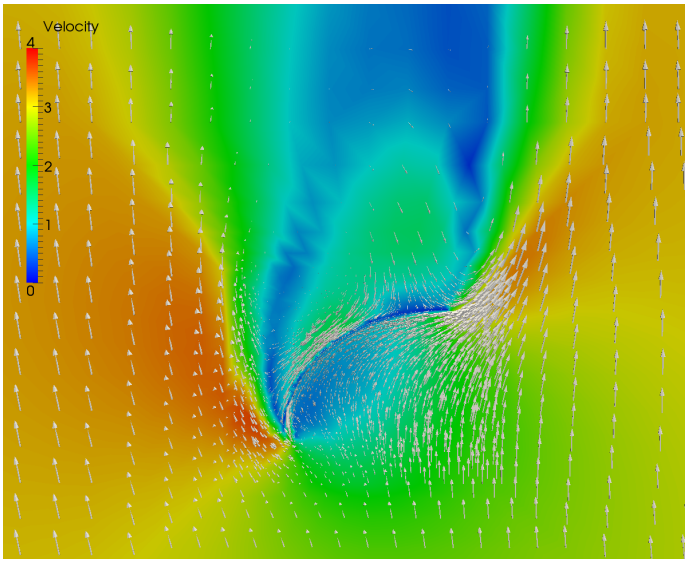


Figure 7.12: Slice view of the velocity field after 3 s.

ulations. Its underlying structure and coupling algorithm is basically the same, but the single modules which carry out the calculations are different. It uses SEDIS as structural solver, which runs only on the master computational node and uses OpenMP to run on multiple threads. IDW interpolation based mesh-motion is implemented with the aid of ATLAS library for matrix operations. The FSI solver developed for this work is equipped instead with Strusol as structural solver, which runs on master node's GPU, and the matrix-free IDW GPU implementation as mesh-motion interpolator, which uses the GPU of on each computational node. In both solvers the fluid problem is handled using OpenFOAM constructs and classes. Using basically the same structural elements, interpolation algorithms, relaxation technique and fluid solver, the two FSI solvers produce obviously comparable results. In order to compare them from the performance point of view, the FSI sub-iteration has been profiled. The test machine is equipped with a Intel Core i7 Q720 CPU, which runs at 1.60 GHz, 10 GB of RAM and a NVIDIA GeForce GTX 670M video board, with 336 CUDA cores and 1.5 GB of GPU memory. The CPU is quad-core, with a total of 8 logic cores. The simulations have been performed with 1, 2, 4 and 8 MPI processes on the same machine, so latency of communication is low (bandwidth is not a problem, since the amount of data exchanged between nodes is quite small). SEDIS runs always with 8 OpenMP threads. In table 7.2, time

Implem.	Nodes	Fluid	Struct.	Mesh	Other	Total
Presented	1	7.745	0.924	1.284	0.127	10.080
Presented	2	5.622	0.928	1.240	0.124	7.914
Presented	4	5.037	0.933	1.191	0.112	7.274
Presented	8	4.427	0.952	1.149	0.197	6.725
Solver [20]	1	7.918	20.450	1.403	0.228	29.999
Solver [20]	2	5.209	20.259	1.129	0.249	26.846
Solver [20]	4	4.786	20.342	1.000	0.315	26.443
Solver [20]	8	4.830	20.413	0.988	0.503	26.734

Table 7.2: FSI sub-iteration timings (in seconds).

taken to perform a single FSI sub-iteration by the two implementations is summarized. All timings in the table are values averaged over several FSI sub-iterations. They tend to remain similar during the simulation; they increase significantly when there is a problem in the setup of the case and solvers converge with difficulty. The "*Fluid*" and "*Struct.*" columns refer to time taken solving the fluid dynamic problem and the structural problem respectively. The "*Mesh*" column indicates how long it takes to perform mesh-motion. This time includes not only the interpolation process but also the actual mesh modifications, which is done in both solvers by OpenFOAM

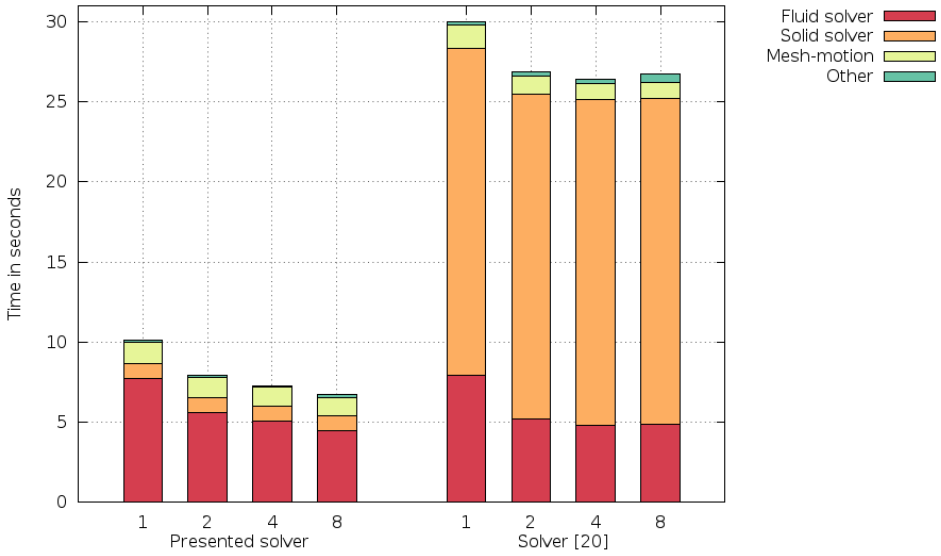


Figure 7.13: Time taken to perform a FSI sub-iteration.
Simulation run on 1, 2, 4 and 8 computational nodes.

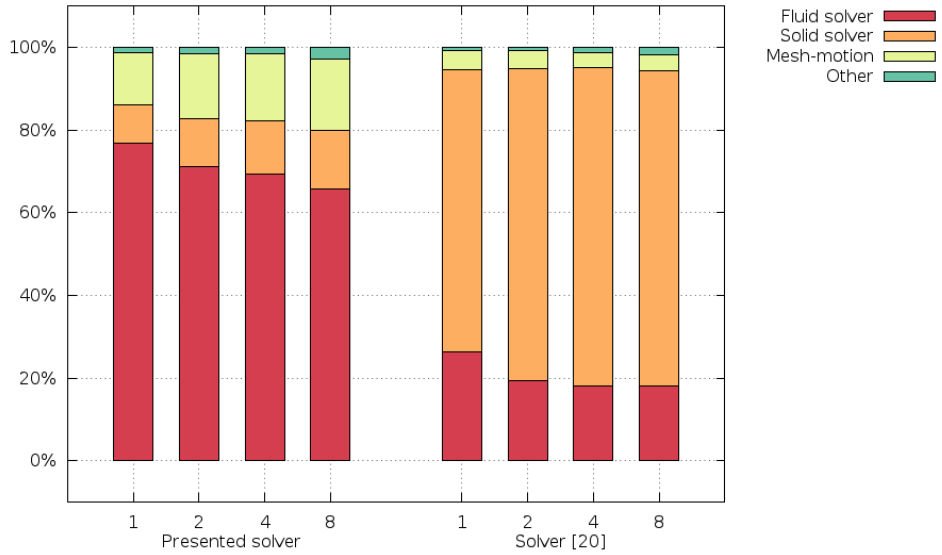


Figure 7.14: Distribution of time inside a FSI sub-iteration. Simulation run on 1, 2, 4 and 8 computational nodes.

in CPU (thus achievable speedup is limited). The "*Other*" column lists time taken to perform other calculations necessary to glue the main operations together.

The best optimization comes from the structural solver as expected, with a speedup of about 22x, thanks to the GPU implementation. Mesh-motion process performs better only with 1 or 2 nodes, it does not scale well when the amount of computational nodes increases. This happens mainly because there is a single GPU board which is shared among the processes; CUDA is capable of handling multiple processes transparently, but performances are affected. Figure 7.13 helps comparing times taken to perform a single FSI sub-iteration by the two FSI solver. It shows clearly how the structural problem solution is the bottleneck of the FSI solver [20]. Thanks to the GPU structural solver, the fluid solver instead constitutes the bottleneck of the presented implementation. Figure 7.14 shows the distribution of computational time among FSI sub-iteration process parts. It shows better how the fluid solver is the main concern now, although its performance scales with the number of processors adopted. Mesh-motion takes still an important part of the computational time. Employing the GPU implementation of the full IDW interpolation algorithm would have given better performance results, but we chose to stick with the matrix-free version in order to use

Nodes	Presented solver	Solver [20]
1	1.060 GB	3.152 GB
2	1.263 GB	3.431 GB
4	1.689 GB	3.884 GB
8	2.441 GB	4.653 GB

Table 7.3: Total amount of memory allocated.

less memory and be able to operate on bigger meshes. In order to support this fact, measure of memory consumption has been performed. Results are summarized in table 7.3 in terms of total amount of memory allocated after the initialization process. A measure of the memory used by each single computational node during a simulation is showed in figure 7.15. Domain has been decomposed using METIS[1], which tries to minimize communication between nodes. The total amount generally grows with the number of nodes since there are some buffers necessary to all of them; the amount on the single node decreases instead, obviously because the domain is partitioned and thus nodes have smaller meshes to handle. From the histogram, the presented FSI solver clearly uses much less memory then the solver [20].

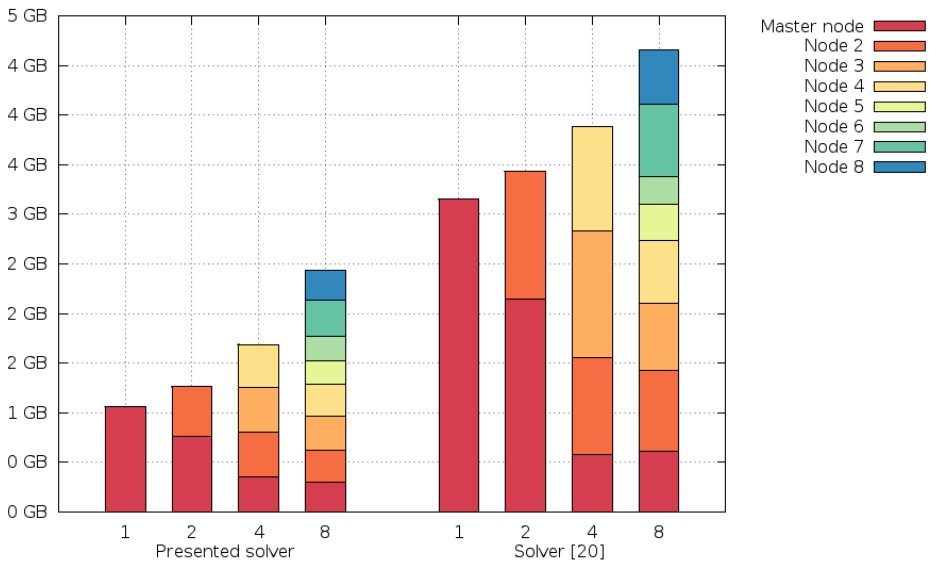


Figure 7.15: Memory allocated by each node.
Solvers run on 1, 2, 4 and 8 computational nodes.

[1] http://glaros.dtc.umn.edu/gkhome/views/metis

Fluid solver implementations are similar and both structural solvers use only a small amount of memory: therefore the real difference is caused by the motion-solvers. The main purpose of IDW matrix-free implementation is in fact to use the GPU computational power to be able to avoid storing huge interpolation matrices; this case is an example of how overall memory consumption is reduced thanks to this approach. It is also important to notice that, when the full IDW interpolation implementation is used, the amount of memory allocated on the single computational node depends heavily on how domains have been decomposed: the interpolation matrix on the computational node is sparse, with an unpredictable number of non-zero entries, which depends on the points inside the corresponding sub-domain.

Note that with 4 nodes the full simulation with the FSI solver [20] runs better than with 8 nodes; this means that the test machine is saturated with 4 processes (which is not strange, since the CPU is a quad-core) and communication between processes becomes a problem if more are used. Structural solvers are not affected since in both these implementations they run only on the master node. In a future release, designed to solve larger cases, a multi-node version of the structural solver will be implemented; with the current structural meshes employed it is sufficiently quick as it is, even though it runs on a single node, thanks to the GPU computational power.

Implem.	Nodes	Fluid	Struct.	Mesh	Other	Total
Presented	1	6.020	0.709	1.316	0.203	8.249
Presented	2	3.562	0.693	0.817	0.176	5.248
Presented	4	1.937	0.692	0.541	0.128	3.299
Presented	8	0.892	0.671	0.312	0.118	1.993
Solver [20]	1	5.879	13.180	2.182	0.256	21.497
Solver [20]	2	3.273	13.523	1.274	0.271	18.341
Solver [20]	4	1.663	13.750	0.662	0.278	16.352
Solver [20]	8	0.751	13.970	0.452	0.332	15.504

Table 7.4: FSI sub-iteration timings (in seconds) on the PLX.

These presented benchmarks were performed on commodity hardware. The same tests have been conducted on the more powerful IBM PLX-GPU (the same hybrid supercomputer described in chapter 6 and used to benchmark the matrix-free IDW interpolation implementation in GPU). Results of benchmarks are reported in table 7.4 and shown graphically in figure 7.16. It is important to highlight again that the supercomputer is shared between users, thus performance analysis could not be very accurate (although attention has been paid in order to give the best benchmark results possible).

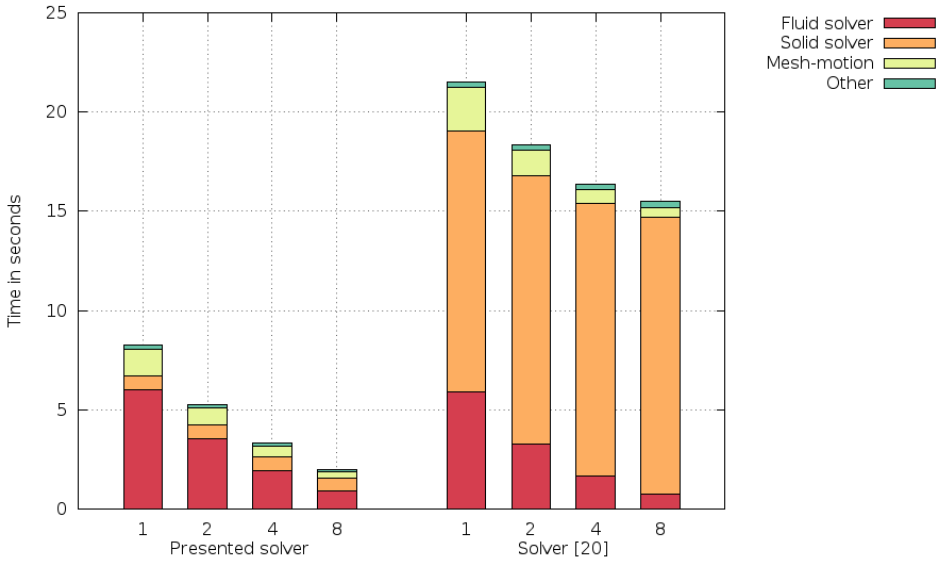


Figure 7.16: Time taken to perform a FSI sub-iteration on the PLX. Simulation run on 1, 2, 4 and 8 computational nodes.

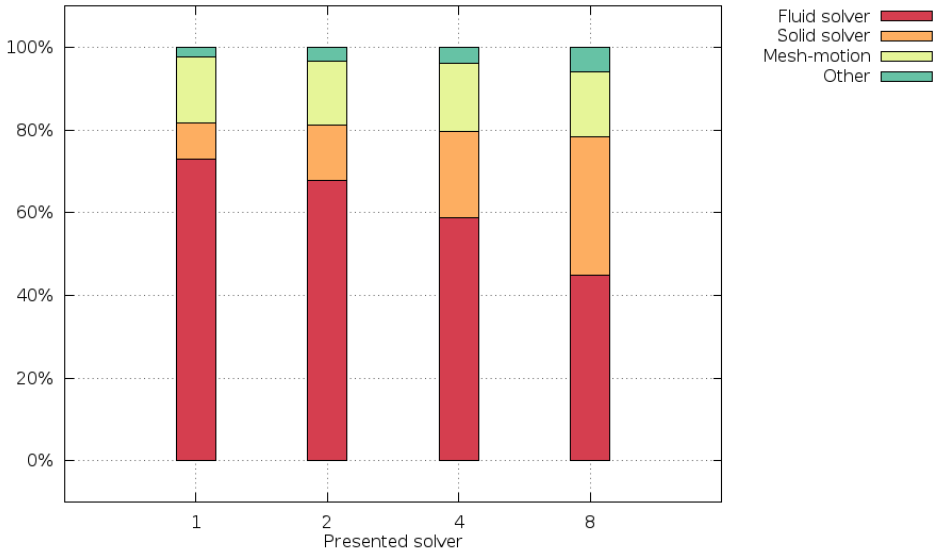


Figure 7.17: Distribution of time inside a FSI sub-iteration on the PLX. Simulation run on 1, 2, 4 and 8 computational nodes.

As expected, On the PLX the fluid solver scales much better, thanks to the powerful CPUs and the fast InfiniBand network. Unlike on the single computer, the mesh-motion GPU implementation is able to scale too. In fact, on the PLX each node has its own GPU board (if MPI processes are assigned to different nodes), while on the single computer it had to be shared. In this way, there is no performance drop caused by running multiple kernels in parallel (obviously if other users are not using the GPU at the same time). Analyzing figure 7.17, where percentages of time taken to perform the single sub-operations are reported, it is clear how, as far as the solver [20] is concerned, the structural problem starts taking the majority of time when the number of computational nodes increases. This tendency is visible on Strusol's timings too, but it is not a major issue since more nodes are needed in order for this to become a problem (considering also that the fluid solver would arrive to a saturation point) and absolute time taken to perform a full FSI sub-iteration is in any case small.

The case tested so far is a coarse version of a more complex setup. While the structural mesh employed gives satisfactory results as it is, if a better simulation is wanted the fluid mesh needs refinements, especially at sea level, in order to fully capture the effects of the twisted wind velocity profile, and in sail's boundary layer. The complete fluid mesh is composed by

Implem.	Nodes	Fluid	Struct.	Mesh	Other	Total
Presented	1	498.508	0.677	52.701	1.408	553.294
Presented	2	240.132	0.678	22.086	0.804	263.700
Presented	4	117.085	0.671	9.994	0.489	128.239
Presented	8	42.510	0.669	5.272	0.287	48.738

Table 7.5: FSI sub-iteration timings (in seconds) with the fine mesh on the PLX.

2 936 932 cells, with 1 019 731 points. A benchmark of the presented FSI solver has been performed on this refined mesh, using 1, 2, 4 and 8 nodes. Results are reported in table 7.5. Absolute time taken to perform the various sub-operations and percentages over the total FSI sub-iteration duration is plotted as histograms in figure 7.18. Total memory consumption is summarized in table 7.6, while memory usage for each node is plotted in figure 7.19. With this mesh, the fluid solver takes obviously much more time if

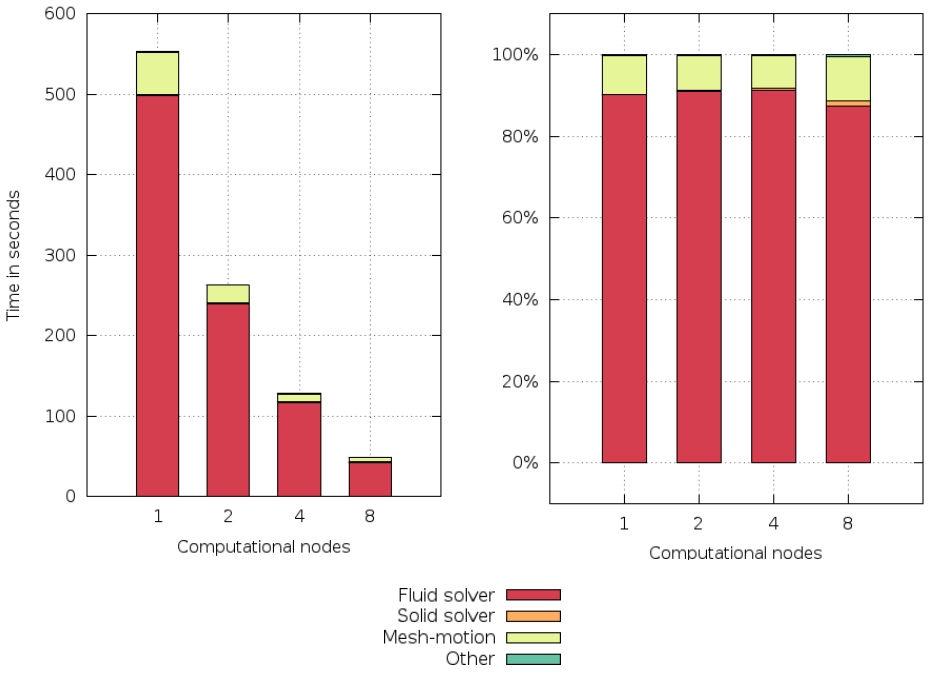


Figure 7.18: Absolute time and percentage of the total time taken by the various operations inside a FSI sub-iteration with the fine mesh on the PLX. Simulation run on 1, 2, 4 and 8 computational nodes.

Nodes	Allocated memory
1	7.804 GB
2	9.287 GB
4	9.655 GB
8	10.535 GB

Table 7.6: Total amount of memory allocated.

compared to the structural solver. Nevertheless, it scales well with the number of computational nodes involved and so does the mesh-motion process. The FSI solver in [20] was unable to run on these amounts of nodes because of the memory needed: even with 8 nodes it was not possible to satisfy all requests of memory on the nodes. It needs a decomposition in more domains in order to lower the required resources per node. During this work it was not possible to request more computational resources and perform a real comparison between the two solvers with the finer mesh; these benchmarks will be completed in the near future.

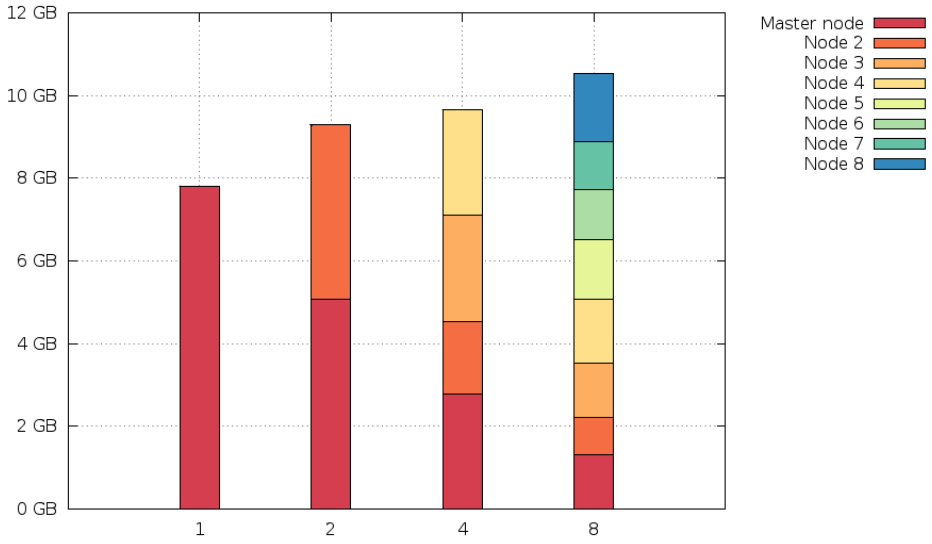


Figure 7.19: Memory allocated by each node.
Simulation run on 1, 2, 4 and 8 computational nodes.

Conclusions

In this work we have shown how GPUs can be effectively used to build implementations which outperform CPU-only algorithms.

A shell finite element structural solver has been developed from scratch with the clear purpose of exploiting the GPU hardware as much as possible. The MITC4 shell element was chosen for its ability to avoid shear-locking and model wrinkling effects, necessary feature in order to correctly simulate sails. The central finite difference explicit time-advancing scheme, together with mass lumping, led to an highly parallelizable time-stepping procedure. In order to fully exploit the GPU hardware potential, the algorithm was designed in order to perform all operations in a per-element basis, with as few communications between elements as possible. Thanks to a revised integration procedure, smart memory management and code optimizations the resulting implementation fitted a single fast CUDA kernel. Two structural problems were employed to test the implementation and measure performances of the new solver: a uniformly loaded circular plate with simply supported edges, for which there was also an analytical solution for the steady case, and a clamped rectangular plate with a constant tip load, which tested the solver in case of large displacements. Solutions obtained with Abaqus were used as reference. After having checked that the two solvers provided the same results, performances were measured. In both cases, thanks to the GPU, the presented solver proved to be several times faster than Abaqus' serial implementation, both with adaptive and fixed time-step. Moreover, all benchmarks were performed on a video-gaming GPU, which means that even better performances are achievable if a GPU suited for double-precision scientific computing is employed.

The presented solver has then been used to solve structural problems in fluid-structure interaction cases. A partitioned strongly coupled FSI solver has been developed to solve problems presenting interactions of shell structures with incompressible Newtonian fluids. The OpenFOAM open-source CFD library was chosen as framework for the implementation. The fluid equations expressed in an ALE formulation were solved using the PISO scheme, implemented with OpenFOAM classes and constructs. The Aitken under-relaxation technique helped lowering the number of sub-iterations nec-

essary to reach convergence in the strong coupling inner loop. In order to enhance overall FSI solver performances, a study on interpolation and mesh-motion algorithms has been conducted, leading to GPU optimized implementations of Radial Basis Functions and Inverse Distance Weighting interpolation methods. Compared to the corresponding CPU implementations, interpolation times decreased substantially, especially concerning the RBF interpolation. However, when highly refined fluid meshes were considered, memory consumption became a serious problem to handle. Therefore, a matrix-less version of IDW interpolation has been developed: exploiting the GPU's powerful floating-point arithmetic units, interpolation coefficients could be recalculated at every interpolation process instead of being stored in a sparse matrix, reducing memory consumption from several gigabytes to some megabytes and saving computational time too.

The solver was tested in two FSI cases. Firstly, a standard test problem of a cubic cavity with flexible bottom filled with an incompressible fluid was considered. A pulsating flow was imposed on the inlet and upper boundary, causing oscillating deformations of the flexible bottom wall. Convergence to a mesh and time-step independent solution was observed and values were successfully compared against results in the literature. Finally, a more complex FSI case was analyzed: a gennaker sail moving under the action of the wind. Performances of the presented FSI solver were directly compared to the solver proposed in [20], which had identical structure and coupling, but used the OpenMP multi-threaded SEDIS structural solver and CPU-only implementations of interpolation and mesh-motion algorithms. A good reduction in computational time was observed, causing the fluid problem solution to be the new bottleneck of the FSI solver. Memory consumption was also lowered, thanks to the matrix-less IDW interpolator, resulting in an overall reduction of resource usage per simulation.

One of the most important fact emerged during this work is that adapting an existing algorithm to run as it is on GPU devices often does not give the wanted results in terms of performance gain. This happens simply because existing algorithms are thought and designed to be run on classical CPU architectures. If a real improvement in performance is wanted then a complete redesign of the algorithm is needed. If a numerical model is meant to be run on GPU devices or hybrid architectures, it is necessary to design it from scratch with these constraints in mind, instead of sticking with old

concepts or adapting existing source codes. Ideas that do not work well in CPU sometimes are suited for the GPU hardware. Examples shown in this work can be the redundancy in keeping copies of local nodal values for each element, as far as the structural solver is concerned, or the recalculation of known values in order to avoid memory accesses in the matrix-free interpolation implementation.

It is important to note that the solvers presented in this work are prototypes and there are plenty of possible enhancements that can be implemented. As far as the structural solver is concerned, its modularity could be exploited to introduce different kinds of elements, without making changes to the underlying algorithmic structure. It would be interesting to add different kinds of boundary conditions also, for example to simulate a trimming sheet attached to one corner of the sail, in order to perform more realistic simulations. Another interesting feature to add would be the support for contact detection and response; it could be tricky to implement and optimize for GPU, but would make the solver suitable for solving a lot of other useful structural problems. The current source code could also be further optimized: it could benefit by handling better the neighboring elements, increasing instruction level parallelism, reducing register spilling even more, using better CUDA intrinsics, etcetera. The problem is that the more optimizations are made, the more it is difficult to maintain a modular and extendible code, therefore a balance should be kept.

The most direct improvement to the FSI solver to be thought of, would be the employment of a CPU/GPU hybrid version of the fluid solver too. Currently there are several studies being conducted on GPU implementation of linear algebra solvers for OpenFOAM. Using them would benefit on overall performances. A multi-GPU implementation of all sub-modules of the FSI solver would also guarantee even lower computational times.

In general, thanks to all these optimizations, the FSI solver is much quicker. Therefore, other better and more complex models can be implemented. In particular, it would be interesting to combine the sail FSI model with the rigid motion of the hull: it would represent a far more realistic setup for studying the dynamics of boats.

Bibliography

[1] Bathe K. J., "*Finite element procedures*", Prentice Hall, 1996.

[2] Bathe K. J., Dvorkin E., "*A four node plate bending element based on Mindlin/Reissner plate theory and a mixed interpolation*", International Journal for Numerical Methods in Engineering, 21(2): pp. 367-383, 1985.

[3] Bos F., "*Numerical simulations of flapping foil and wind aerodynamics: mesh deformation using radial basis functions*", PhD Thesis, Technical University Delft, 2009.

[4] Chapelle D., Bathe K. J., "*The finite element analysis of shells - Fundamentals*", Springer, 2003.

[5] Colombi M., "*Sviluppo di un modello ad elementi finiti per la simulazione di contatto e frattura di contenitori alimentari in laminato sottile*", Master Thesis, Politecnico di Milano, 2004.

[6] Corradi dell'Acqua L., "*Meccanica delle strutture - Volume 1: Il comportamento dei mezzi continui*", McGraw–Hill, 1992.

[7] Corradi dell'Acqua L., "*Meccanica delle strutture - Volume 2: Le teorie strutturali e il metodo degli elementi finiti*", McGraw–Hill, 1992.

[8] Courant R., Friedrichs K., Lewy H., "*Über die partiellen Differenzengleichungen der mathematischen Physik*", Mathematische Annalen, 100: pp. 32-74, 1928.

[9] Cremonesi M., "*Implementazione di tecniche di parallelizzazione e di un metodo lagrangiano a particelle di fluido finalizzati allo sviluppo di un codice ad elementi finiti per problemi di interazione fluido-struttura*", Master Thesis, Politecnico di Milano, 2005.

[10] Deparis S., "*Numerical Analysis of Axisymmetric Flows and Methods for Fluid-Structure Interaction Arising in Blood Flow Simulation*", PhD thesis, EPFL, 2004.

[11] Deparis S., Discacciati M., Fourestey G., Quarteroni A., "*Fluid-structure algorithms based on Steklov–Poincaré operators*", Computer Methods in Applied Mechanics and Engineering, pp. 5797-5812, 2006.

[12] Felippa C. A., Park K. C., Farhat C., "*Partitioned analysis of coupled mechanical systems*", Computer Methods in Applied Mechanics and Engineering, 190(24-25): pp. 3247-3270, 2001.

[13] Ferziger J. H., Perić M., "*Computational methods for fluid dynamics*", Springer, 1996.

[14] Giampieri A. N., "*An interface element to model the mechanical response of crease lines for carton-based packaging*", PhD Thesis, Politecnico di Milano, 2009.

[15] Harris M., "*Optimizing parallel reduction in CUDA*", NVIDIA.

[16] Jasak H., Tuković Ž., "*Automatic mesh motion for unstructured finite volume method*", Transactions of FAMENA, vol. 30, 2007.

[17] Karrholm F. P., "*Rhie–Chow interpolation in OpenFOAM*", Chalmers University of Technology, 2006.

[18] Krieg R., "*Unconditional Stability in Numerical Time Integration Methods*", ASME, Journal of Applied Mechanics, 1973.

[19] Lesoinne M., Farhat C., "*Geometric conservation laws for flow problems with moving boundaries and deformable meshes, and their impact on aeroelastic computations*", Computer Methods in Applied Mechanics and Engineering, 134: pp. 71-90, 1996.

[20] Lombardi M., "*Numerical simulation of a sailing boat: free surface, fluid structure interaction and shape optimization*", PhD Thesis, EPFL, 2012.

[21] Lombardi M., Parolini N., Quarteroni A., Rozza G., "*Numerical simulation of sailing boats: dynamics, FSI and shape optimization*", Variational Analysis and Aerospace Engineering: Mathematical Challenges

for Aerospace Design, Springer Optimization and Its Applications 2012, pp. 339-377, 2012.

[22] Mok D. P., "*Partitionierte Losungsansatze in der Strukturdynamik und der Fluid-Struktur-Interaktion*", PhD Thesis, Institut für Baustatik, 2001.

[23] Mok D. P., Wall W. A., "*Partitioned analysis schemes for the transient interaction of incompressible flows and nonlinear flexible structures*", in Wall, W. A., Bletzinger, K. U., and Schweizerhof, K., editors, *Trends in Computational Structural Mechanics*, pp. 689-698, 2001.

[24] "*CUDA C Programming Guide*", NVIDIA.

[25] "*CUDA C Best Practices Guide*", NVIDIA.

[26] "*NVIDIA next generation CUDA compute architecture: Fermi*", whitepaper, NVIDIA, 2009.

[27] Oakley D. R., Knight N. F., "*Adaptive Dynamic Relaxation Algorithm for Non-Linear Hyperelastic Structures, Part 1 - Formulation*", Computer Methods in Applied Mechanics and Engineering, 1995.

[28] Parolini N., Lombardi M., "*Unsteady FSI simulation of downwind sails*", V International Conference on Computational Methods in Marine Engineering, 2013.

[29] Piperno S., Farhat C., and Larrouturou B., "*Partitioned procedures for the transient solution of coupled aroelastic problems part 1: Model problem, theory and two-dimensional application*", Computer Methods in Applied Mechanical Engineering, 124: pp. 79-112, 1995.

[30] Quarteroni A., "*Modellistica numerica per problemi differenziali*", 4th edition, Springer, 2008.

[31] Quarteroni A., Sacco R., Saleri F., "*Matematica numerica*", 3rd edition, Springer, 2008.

[32] Timoshenko S., Woinowsky–Krieger S., "*Theory of plates and shells*", McGraw–Hill, 1959.

[33] Trimarchi D., Turnock S. R., Taunton D. J., Chapelle D., "*The use of shell elements to capture sail wrinkles, and their influence on aerodynamic loads*", in the Second International Conference on Innovation in High Performance Sailing Yachts, Lorient, France, 2010.

[34] Vázquez J. G. V., "*Nonlinear Analysis of Orthotropic Membrane and Shell Structures Including Fluid-Structure Interaction*", PhD Thesis, Universitat Politècnica de Catalunya, 2007.

[35] Volkov V., "*Better performance at lower occupancy*", GPU Technology Conference, 2010.

[36] Witteveen J., Bijl H., "*Explicit mesh deformation using inverse distance weighting interpolation*", in Proceedings of 47th AIAA Aerospace Sciences Meeting, 2009.

[37] Zienkiewicz O. C., Taylor R. L., "*The finite element method. Volume one: the basis*", 5th edition, Oxford, Butterworth–Heinemann, 2000.