Master thesis : Performance evaluation and optimization of a GPU-enabled Discontinuous Galerkin code

Auteur : D'Antonio, Marco
Promoteur(s) : Geuzaine, Christophe
Faculté : Faculté des Sciences appliquées
Diplôme : Cours supplémentaires destinés aux étudiants d'échange (Erasmus, ...)
Année académique : 2021-2022
URI/URL : http://hdl.handle.net/2268.2/15924

University of Salerno - Department of Computer Science

University of Liège - School of Engineering and Computer Science

Master's Degree in Computer Science

Performance analysis and optimization of a GPU-enabled Discontinuous Galerkin solver

Master's thesis carried out to obtain the degree of
Master of Science in Computer Science by Marco D'Antonio

Supervisors
Prof. Filomena Ferrucci
Prof. Christophe Geuzaine
Matteo Cicuttin, Ph.D.

Candidate
Marco D'Antonio - 0522500958

Academic Year 2021-2022

Contents

Introduction

The use of GPUs for general-purpose computing is a trend that has gradually established itself over the years, gaining more and more momentum with the development of modern programming models. Over the past decade, the use of GPUs for high performance computing has enabled better performance on a variety of problem types, and it is no coincidence that seven of the top ten supercomputers in the world feature graphics accelerators. Much like parallel software development on CPUs however, development on GPUs hides pitfalls and can become complex, especially in modern supercomputers where the use of multiple GPUs opens up even greater challenges.

To build parallel applications with optimal performance, it is therefore necessary to be aware of the issues of both the platforms for development and the hardware on which the applications are developed. The developer going to implement solutions that aim for high performance must not only use the abstractions made available by the programming model, but must be aware of the impact that each instruction has with respect to performance.

This thesis focuses in particular on analyzing the performance and possible optimizations achievable on a solver based on the Discontinuous Galerkin method for Maxwell's equations, Gmsh DG[1], developed at the Applied and Computational Electromagnetics research group of the University of Liège. Applications of this type perform simulations that deal with the computation of millions of floating-point values per second, and which require appropriate optimization to be completed in a reasonable time. Thus, the main research question that we are trying to answer is:

How well can a reasonably simple DG solver exploit modern hardware?

In order to get an answer to this question, we structured the work in the following objectives

1. propose new optimizations to be implemented to improve the current limits of the application;
2. evaluate the performance of the application on real problems, perform scaling analysis and specific analysis related to the stages of computation;
3. help in the implementation of support for the use of multiple GPUs and performance analysis of the results obtained.

The solver has already been tested on real problems, for example, to simulate an electrostatic discharge on a device, and some performance results have been shown [1]. However,

[1]`https://gitlab.onelab.info/gmsh/dg`

a comprehensive analysis that can direct future developments and provide further insights had not yet been carried out. The starting point for performing these analyses is a thorough study of the existing code and the numerical concepts behind it. Then, starting with known bottlenecks, implementation of optimization strategies can be considered, followed by an analysis of these strategies to validate improvements. Finally, performance analysis must be performed both at the overall application level and at the level of individual computational kernels. Scaling analysis deals with overall performance on a real-world problem, investigating how the application adapts as computational and memory resources increase. The analysis of the kernels must be done through the use of specialized tools to derive accurate data, particularly with regard to analyzing the number of floating-point operations performed and memory traffic.

The first three chapters will be in a certain sense introductory to the work of this thesis. Chapter 1 will explain the physical and mathematical foundations related to the problem and the Discontinuous Galerkin method used to solve it, along with providing insights regarding how the method can be parallelized on multiprocessor systems. Chapter 2 will introduce the reader to GPU architecture and programming for general-purpose computing, with a special focus on the cautions needed to achieve good performance during GPU development. Chapter 3 describes the implementation of the Discontinuous Galerkin solver, starting by performing a review of the existing literature on solvers using graphics accelerators and then moving to the details of the code, along with the strategy used for parallelization.

The next three chapters are the heart of this work, showing the results obtained for each of the set goals. Chapter 4 discusses the optimizations implemented in the code and some of the optimization proposals that were considered for implementation. Chapter 5 focuses on analyzing the code from an application performance point of view through the scaling analysis on a multiprocessor system, and from a kernel perspective through the use of roofline analysis. Finally, Chapter 6 covers the multi-GPU implementation, describing how systems composed by multiple GPUs can be built and interconnected, the difficulties regarding the implementation, and showing the performance obtained on the platform at our disposal.

This thesis has been carried out under the Erasmus+ program with the collaboration between the University of Salerno and the University of Liège.

Chapter 1

Physical and mathematical context

Electricity and magnetism are two phenomena that, at least qualitatively, are known to humanity since the ancient times. Their systematic study however started only in the late 17th century, and since then countless scientist contributed to the improvement of our knowledge about these subjects.

Once thought as separate phenomena, it was only in 1820 that the first hints of a possible connection between electricity and magnetism were discovered by Hans Christian Ørsted. By observing that the needle of a compass was deflected by a wire carrying an electric current, we can confidently say that he started the journey to the unification of electricity and magnetism. The subsequent studies carried out by Faraday, Ampère and Maxwell provided new insights, that finally led to the Maxwell's equations. Such equations represent perhaps the major scientific achievement of the 19th century: electricity, magnetism and optics unified in a single theory.

Maxwell's equations are central in electrical engineering, as they allow to model and predict the behaviour of the most disparate devices and systems. Their manual solution is however impossible in almost every practical setting, therefore accurate, reliable and fast numerical methods are needed. In this chapter we introduce the basics of the Discontinuous Galerkin method, which is at the core of the code that was analyzed and optimized in this thesis.

1.1 Vector calculus

We recall here the main vector calculus notions that are used in the electromagnetic theory. For conciseness, we limit ourselves to $\mathbb{R}^3$.

A vector of $\mathbb{R}^3$ is denoted with boldface font, as in $\boldsymbol{u}$; its x, y, z components are denoted as u_x, u_y and u_z respectively. Similarly, a vector-valued function is denoted with boldface font, as in $\boldsymbol{f}(\cdot)$ and its components as $f_x(\cdot), f_y(\cdot), f_z(\cdot)$.

The nabla operator ∇ is defined as the column vector of the partial derivatives

$$\nabla = \left[\tfrac{\partial}{\partial x}\ \tfrac{\partial}{\partial y}\ \tfrac{\partial}{\partial z}\right]^T.$$

The nabla operator can be applied to sufficiently smooth scalar functions $f : \mathbb{R}^3 \to \mathbb{R}$ or to sufficiently smooth vector functions $\boldsymbol{f} : \mathbb{R}^3 \to \mathbb{R}^3$ yielding different quantities of interest:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix}, \qquad \nabla \times \boldsymbol{f} = \begin{bmatrix} \frac{\partial f_z}{\partial y} - \frac{\partial f_y}{\partial z} \\ \frac{\partial f_x}{\partial z} - \frac{\partial f_z}{\partial x} \\ \frac{\partial f_y}{\partial x} - \frac{\partial f_x}{\partial y} \end{bmatrix}, \qquad \nabla \cdot \boldsymbol{f} = \sum_{w \in \{x,y,z\}} \frac{\partial f_w}{\partial w}. \tag{1.1}$$

The three expressions appearing in (1.1) are the *gradient* of f, the *curl* of $\boldsymbol{f}$ and the *divergence* of $\boldsymbol{f}$ respectively. We recall also that the following identities hold:

$$\nabla \times (\nabla f) = \mathbf{0} \quad \forall f, \qquad \nabla \cdot (\nabla \times \boldsymbol{f}) = 0 \quad \forall \boldsymbol{f}. \tag{1.2}$$

A central ingredient for the discussion of the numerical methods employed in this thesis is the divergence theorem. For sufficiently smooth functions $\boldsymbol{f} : \mathbb{R}^3 \to \mathbb{R}^3$, it holds that

$$\int_V (\nabla \cdot \boldsymbol{f})\, dv = \int_{\partial V} \boldsymbol{f} \cdot\, d\boldsymbol{s}. \tag{1.3}$$

By applying the substitution $\boldsymbol{f} \to \boldsymbol{f} g$ to (1.3) and using the vector calculus identity $\nabla \cdot (\boldsymbol{f} g) = (\nabla \cdot \boldsymbol{f})g + \boldsymbol{f} \cdot (\nabla g)$ we obtain an important corollary that will be used in the following:

$$\int_V \boldsymbol{f} \cdot (\nabla g)\, dV + \int_V (\nabla \cdot \boldsymbol{f}) g\, dV = \int_{\partial V} (\boldsymbol{f} \cdot \hat{\boldsymbol{n}}) g\, dS. \tag{1.4}$$

1.2 Maxwell's equations

Let $\boldsymbol{x} \in \mathbb{R}^3$ denote a position vector and $t \in \mathbb{R}^+$ the time, where $\mathbb{R}^+ = \{x : x \in \mathbb{R}, x \geq 0\}$. The time-domain Maxwell's equations are written in differential form as

$$\nabla \times \boldsymbol{e}(\boldsymbol{x}, t) = -\frac{\partial \boldsymbol{b}(\boldsymbol{x}, t)}{\partial t}, \tag{1.5}$$

$$\nabla \times \boldsymbol{h}(\boldsymbol{x}, t) = \frac{\partial \boldsymbol{d}(\boldsymbol{x}, t)}{\partial t} + \boldsymbol{j}(\boldsymbol{x}, t) + \boldsymbol{j}_s(\boldsymbol{x}, t) \tag{1.6}$$

$$\nabla \cdot \boldsymbol{b}(\boldsymbol{x}, t) = 0, \tag{1.7}$$

$$\nabla \cdot \boldsymbol{d}(\boldsymbol{x}, t) = \rho(\boldsymbol{x}, t), \tag{1.8}$$

where the following quantities are involved:

- $\boldsymbol{e} : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}^3$ is the *electric field*, which has units of V/m (Volts per meter),
- $\boldsymbol{b} : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}^3$ is the *magnetic flux density*, which has units of T (Tesla),
- $\boldsymbol{h} : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}^3$ is the *magnetic field*, which has units of A/m (Ampère per meter),

- $\boldsymbol{d} : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}^3$ is the *electric displacement field*, which has units of C/m^2 (Coulomb per square meter)

- $\boldsymbol{j} : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}^3$ is the *current density* in conductive media, whereas $\boldsymbol{j_s} : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}^3$ is the *current density* due to the sources. Both fields have units of A/m^3 (Ampère per cubic meter)

- $\rho : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}$ is the *charge density*, which has units of C/m^3 (Coulomb per cubic meter)

From now on, the spatial and temporal dependence of all the fields will be considered implicit, therefore we will omit to indicate it in the equations. The four equations (1.5)-(1.8) were devised by different scientists, in particular:

- Equation (1.5) is the Faraday-Neumann law, and states that a time-varying *magnetic flux density* $\boldsymbol{b}$ induces a circulation of the *electric field* $\boldsymbol{e}$. It formalizes the concept of electromagnetic induction, discovered by Michael Faraday in 1831. The equation in this form is due to Franz Ernest Neumann, who introduced it in 1845.

- Equation (1.6) is the Ampère-Maxwell law, and describes the magnetic fields that form around electric currents. More in detail, a circulation of the *magnetic field* $\boldsymbol{h}$ is given by the sum of $\boldsymbol{j}$, which collects the current densities due to electrical conduction in the materials and $\boldsymbol{j}_s$, which collects the current density sources.

  The third term, the temporal variation of the *displacement field* $\boldsymbol{d}$, is named *displacement current* and was introduced by James Clerk Maxwell in 1865. This led to the unification of the magnetic and electric fields and to the formalization of the propagation of electromagnetic waves.

- Equation (1.7) is the magnetic Gauss' law. It states the non-existence of magnetic monopoles, as in a zero-divergence field all the field lines are closed.

- The electric Gauss' law (1.8) finally allows to state (by integrating on the considered domain) that the electric flux through a closed surface equals the charge enclosed by the same surface.

Normally, Maxwell's equations are solved in a domain $\Omega \subset \mathbb{R}^3$ and subject to specific initial and boundary conditions. The most common boundary conditions are:

- $\hat{\boldsymbol{n}} \times \boldsymbol{e} = \boldsymbol{0}$, known as *Perfect Electric Conductor* (PEC),

- $\hat{\boldsymbol{n}} \times \boldsymbol{h} = \boldsymbol{0}$, known as *Perfect Magnetic Conductor* (PMC),

- $Z\hat{\boldsymbol{n}} \times \boldsymbol{h} = \hat{\boldsymbol{n}} \times (\boldsymbol{e} \times \hat{\boldsymbol{n}})$, known as *Impedance Boundary Condition* (IBC).

Materials are taken into account by the three *constitutive relations*

$$\boldsymbol{d} = \epsilon \boldsymbol{e}, \qquad \boldsymbol{b} = \mu \boldsymbol{h}, \qquad \boldsymbol{j} = \sigma \boldsymbol{e}, \tag{1.9}$$

which are called *electric constitutive relation*, *magnetic constitutive relation* and *Ohm's law* respectively. The quantities ϵ, μ and σ are the *electric permittivity*, the *magnetic permeability* and the *conductivity*, and depend on the considered material; we will refer to these quantities with the general designation of *material parameters.* In the most general setting all the three quantities are tensors belonging to $\mathbb{R}^{3\times 3}$ and can be position- and time-dependent. In the context of this thesis however, we will only consider materials which are *linear, locally homogeneous, isotropic* and *nondispersive*, leading to piecewise constant, scalar material parameters: this setting, despite not being fully general, covers a huge class of practical engineering problems.

Electric permittivity and magnetic permeability are commonly further decomposed as $\epsilon = \epsilon_r \epsilon_0$ and $\mu = \mu_r \mu_0$ respectively, where the terms with the subscript 0 are the *free space* parameters, whereas the terms subscripted with r are the *relative* parameters. In SI units, the values of the free space parameters are $\epsilon_0 = 8.8541878128 \times 10^{-12}$ F/m and $\mu_0 = 4\pi \times 10^{-7}$ H/m.

In order to show that the electromagnetic field propagates in waves, we consider the source-free setting where $\boldsymbol{j}_s = \boldsymbol{0}, \rho = 0$ and that there is no conductive media, therefore $\sigma = 0$. By substituting the constitutive relations (1.9) in (1.5)-(1.6) and rearranging, one obtains

$$\mu \frac{\partial \boldsymbol{h}(\boldsymbol{x}, t)}{\partial t} = -\nabla \times \boldsymbol{e}(\boldsymbol{x}, t), \tag{1.10}$$

$$\epsilon \frac{\partial \boldsymbol{e}(\boldsymbol{x}, t)}{\partial t} = \nabla \times \boldsymbol{h}(\boldsymbol{x}, t). \tag{1.11}$$

By taking the divergence of (1.10) and using the second identity in (1.2), we deduce that (1.7) is automatically satisfied by (1.10). In addition, by taking the divergence of (1.10), using the same identity as before and recalling that we are in a source-free setting, we deduce that (1.8) is automatically satisfied by (1.11). Therefore, we do not need to enforce explicitly (1.7) and (1.8) during the solution process.

Now, by taking the curl of (1.10) and substituting (1.11) in the result, one obtains

$$\nabla \times \mu^{-1}(\nabla \times \boldsymbol{e}) + \epsilon \frac{\partial^2 \boldsymbol{e}}{\partial t^2} = \boldsymbol{0}, \tag{1.12}$$

confirming that the electromagnetic field propagates as waves with speed $c = 1/\sqrt{\mu\epsilon}$.

In the following, the system of equations (1.10)-(1.11) will be called *first order formulation* of the Maxwell's equations, wheareas (1.12) will be called *second order formulation* of the Maxwell's equations. In the context of this thesis, we will be interested in solving the first order formulation using appropriate numerical methods.

1.3 Numerical solution of PDEs

Partial differential equations (PDEs) like Maxwell's equations are rarely solvable analitically, therefore appropriate numerical methods must be used to approximate their solution. In addition, it must be noted that differential equations are statements involving all the points of a domain, which are clearly in infinite number and therefore intractable on a computer. It is thus necessary to appropriately discretize the computational domain *and* the differential equation at hand.

In order to gently introduce the subject of spatial discretizations for PDEs, we start by describing the Finite Volume method in 1D before moving to the more complex Discontinuous Galerkin method, also in 1D. Once all the basics are in place, we will finall move to the Discontinuous Galerkin method for the Maxwell's equations. We limit the level of detail to what is needed for this thesis, we refer the interested reader to [2, 3, 4] for a more in-depth discussion.

1.3.1 Computational domain and meshes

In this section, some general definitions about meshes will be recalled, see [5, Chapter1] for more details.

Definition 1 (Mesh). *Let Ω be a polyhedral domain. A discretization of Ω is a collection of polyhedral elements $\mathcal{T} := \{T_1, \dots, T_n\}$ such that*

$$T_i \cap T_j = \emptyset, \quad \forall i \neq j \in \{1, \dots, |\mathcal{T}|\}; \qquad \bigcup_i \overline{T_i} = \overline{\Omega}, \quad i \in \{1, \dots, |\mathcal{T}|\}$$

Definition 2 (Mesh size). *Let $\mathcal{T}$ be a mesh of Ω. The quantity h_T denotes the* diameter *of an element T. The* mesh size *is denoted as the real number*

$$h := \max_{T \in \mathcal{T}} h_T.$$

In the following, the notation $\mathcal{T}_h$ will be used to denote a mesh whose size is h.

Definition 3 (Mesh skeleton). *Let $\mathcal{T}_h$ be a mesh covering Ω. The skeleton Γ, the internal skeleton Γ_{int} and the boundary skeleton Γ_{bnd} are the sets*

$$\Gamma := \bigcup_{T \in \mathcal{T}} \partial T, \qquad \Gamma_{int} := \Gamma \setminus \partial\Omega, \qquad \Gamma_{bnd} = \Gamma \setminus \Gamma_{int}.$$

Definition 4 (Mesh faces)**.** *Let $\mathcal{T}_h$ be a mesh covering Ω. A subset $F \subset \overline{\Omega}$ is defined to be a face if one of the following two conditions hold:*

- *Given two distinct elements T_1 and T_2, the set $F = T_1 \cap T_2 \subset \Gamma_{int}$ is nonempty, and in this case F is called internal face.*
- *Given an element T, the set $F = T \cap \partial\Omega \subset \Gamma_{bnd}$ is nonempty, and in this case F is called boundary face.*

The faces of an element T are the elements $F_i \subset \Gamma$ such that $F_i \in \partial T$. We denote $\mathcal{F}_T$ the set of the faces of T.

Definition 5 (Normals on faces)**.** *Let $\mathcal{T}_h$ be a mesh covering Ω. For each T and for each $F_i \in \mathcal{F}_T$, we denote with $\hat{\boldsymbol{n}}$ the outward normal on F_i. Given two adjacent elements T^+ and T^- sharing the face F, $\hat{\boldsymbol{n}}^+$ denotes the normal on F pointing from T^+ to T^-. The normal $\hat{\boldsymbol{n}}^-$ is similarly defined to be the normal on F pointing from T^- to T^+.*

- $\Gamma := \bigcup_{T \in \mathcal{T}_h} \partial T$ (skeleton)
- $\Gamma_{int} = \Gamma \setminus \partial\Omega$
- T^+ and T^- generic elements sharing a face
- $F := T^+ \cap T^- \subset \Gamma_{int}$
- $\hat{\boldsymbol{n}}^+$ and $\hat{\boldsymbol{n}}^-$ normals of T^+ and T^- on F

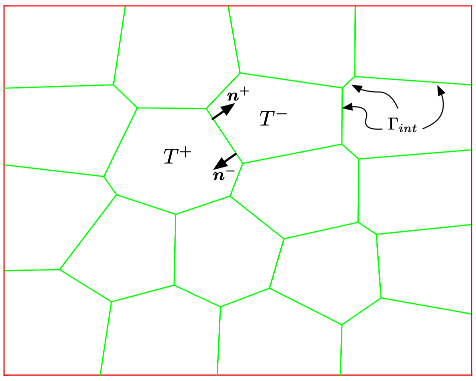


Figure 1.1: Visual summary of the symbols introduced to denote mesh elements.

1.3.2 Conservation laws in one dimension

Let $u(x,t) : \mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$, $f(u)$ a *flux function* and $g(x,t) : \mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$. We consider the conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = g, \tag{1.13}$$

posed on a certain domain Ω and subject to the appropriate boundary conditions. In addition, we consider only linear fluxes of the form $f(u) = au$. The equation (1.13) is solved by some numerical method on a discretization of Ω, obtaining a numerical solution u_h which is clearly different from the true solution u. The solutions u and u_h are sought in some spaces V and V_h respectively, which will be made more precise further ahead. Once a solution u_h is obtained, by plugging u_h in (1.13) it is possible to define the *residual*

$$\mathcal{R}(x,t) := \frac{\partial u_h}{\partial t} + \frac{\partial f_h}{\partial x} - g, \tag{1.14}$$

which clearly can not be zero, otherwise we would have obtained the exact solution u to the equation. Our goal is to obtain a numerical method that solves (1.13) yielding a discrete solution u_h such that the residual $\mathcal{R}$ has some properties. Such properties depend on the chosen numerical method.

To fix the ideas, let the computational domain be $\Omega = [0,1]$; we are thus considering a 1D setting. In order to obtain a spatial discretization of Ω we can divide the interval [0,1] in N segments. The segments so obtained are the elements $T_1, \dots, T_N$ of the mesh. Each segment T_n has a start point $x_{n-1/2}$ and an end point $x_{n+1/2}$, therefore $h_{T_n} = x_{n+1/2} - x_{n-1/2}$.

1.3.3 The Finite Volume Method

A possible strategy to approximate the solution of (1.13) in our discrete domain, is to consider its cell-by-cell average

$$\bar{u}_n = \frac{1}{h_{T_n}} \int_{T_n} u \, dV, \qquad \frac{d\bar{u}_n}{dt} = \frac{1}{h_{T_n}} \int_{T_n} \frac{\partial u}{\partial t} \, dV.$$

In this way, the discrete approximation is a piecewise constant function whose value in each element is the average of the solution in the considered element. In order to obtain a numerical method from such strategy, we proceed as follows:

- **Step 1:** once the domain is discretized and the elements T_n are identified, take the average of (1.13) *elementwise* obtaining

$$\frac{1}{h_{T_n}} \int_{T_n} \frac{\partial u}{\partial t} dV + \frac{1}{h_{T_n}} \int_{T_n} \frac{\partial f}{\partial x} dV = \frac{1}{h_{T_n}} \int_{T_n} g \; dV, \quad \forall T_n \in \mathcal{T} \tag{1.15}$$

- **Step 2:** Apply the divergence theorem (1.3) to the spatial term of (1.15)

$$\int_{T_n} \nabla \cdot \boldsymbol{f} dV = \int_{\partial T_n} \boldsymbol{f} \cdot \hat{\boldsymbol{n}} dS \quad \xrightarrow{\text{or, in 1D}} \quad \int_{T_n} \frac{\partial f}{\partial x} dl = f^{n+1/2} - f^{n-1/2},$$

  where $f^{n-1/2}$ and $f^{n+1/2}$ are the values of the flux at the boundaries of T_n. We can then obtain the element-local statement

$$\frac{d\bar{u}_n}{dt} + \frac{f^{n+1/2} - f^{n-1/2}}{h_{T_n}} = \bar{g}_n \tag{1.16}$$

With the two steps above we obtained a semi-discrete scheme that provides N equations, one for each element T_n. The scheme however is still incomplete and does not allow to approximate our original equation (1.13). Indeed, recalling that we consider only fluxes of the type $f(u) = au$ and that we chose a piecewise constant approximation for u_h, one immediately realizes that u_h at the interfaces between elements is not single-valued.

In order to complete the scheme then, one needs to provide a way to specify the values of the *numerical fluxes* $f^{n-1/2}$ and $f^{n+1/2}$. It turns out that the choice of the numerical fluxes is a major topic in the study of the discontinuous methods, therefore no attempt will be made to discuss the subject further. We refer the reader to [4, 6] for the details, here we limit to mention the two most simple and common possibilities:

- **Centered fluxes:** The centered fluxes are the simplest and most intuitive possibility. In this case the value at the interfaces is fixed by taking the average between the solution in the two adjacent cells. More formally:

$$f^{n+1/2} := \frac{a}{2}(\bar{u}_n + \bar{u}_{n+1}). \tag{1.17}$$

- **Upwind fluxes:** The upwind fluxes have a slightly more complex expression:

$$f^{n+1/2} := \frac{a}{2}(\bar{u}_n + \bar{u}_{n+1}) + \frac{a}{2}(\bar{u}_n - \bar{u}_{n+1}) \cdot \hat{\boldsymbol{n}}. \tag{1.18}$$

The precise definition of the fluxes now allows us to obtain a usable numerical scheme. Let us consider, only for illustration purposes, a forward finite difference discretization of the time derivative. Let $\Delta x = h_{T_n}$, the finite volumes discrete update equations are

$$\bar{u}_n^{i+1} = \bar{u}_n^i - \frac{\Delta t}{\Delta x}(f^{n+1/2} - f^{n-1/2}). \tag{1.19}$$

If we consider centered fluxes we have

$$f_i^{n+1/2} = \frac{a}{2}(\bar{u}_n^i + \bar{u}_{n+1}^i), \qquad f^{k-1/2} = \frac{a}{2}(\bar{u}_{n-1}^i + \bar{u}_n^i). \tag{1.20}$$

By replacing (1.20) in (1.19) we obtain

$$\bar{u}_n^{i+1} = \bar{u}_n^i - \frac{a\Delta t}{2\Delta x}(\bar{u}_{n+1}^i - \bar{u}_{n-1}^i),$$

which is directly implementable in software. In order to have stability, the scheme is subject to the Courant–Friedrichs–Lewy (CFL) condition

$$\frac{a\Delta t}{2\Delta x} \leq 1.$$

On the other hand, if we use upwind fluxes we have

$$f^{n+1/2} = \frac{a}{2}(\bar{u}_n^i + \bar{u}_{n+1}^i) + \frac{a}{2}(\bar{u}_n^i - \bar{u}_{n+1}^i), \quad f^{n-1/2} = \frac{a}{2}(\bar{u}_{n-1}^i + \bar{u}_n^i) - \frac{a}{2}(\bar{u}_{n-1}^i - \bar{u}_n^i), \tag{1.21}$$

therefore, by replacing (1.21) in (1.19) we obtain

$$\bar{u}_n^{i+1} = \bar{u}_n^i - \frac{a\Delta t}{\Delta x}(\bar{u}_n^i - \bar{u}_{n-1}^i),$$

which again is easily implementable in software. In the upwind case the CFL is

$$\frac{a\Delta t}{\Delta x} \leq 1.$$

We remark that the centered fluxes result in a method which is essentially equivalent to a central finite difference, whereas the upwind fluxes result in a backward finite difference. This has important implications, the most significant being that the upwind fluxes introduce numerical diffusion. More in detail, we recall that

$$\frac{d^2 v}{dx^2} \approx \frac{v_{n+1} - 2v_n + v_{n-1}}{\Delta x^2}.$$

Arranging appropriately the terms when replacing (1.21) in (1.19), it is possible to obtain

$$\bar{u}_n^{i+1} = \bar{u}_n^i - \frac{a\Delta t}{2\Delta x}(\bar{u}_{n+1}^i - \bar{u}_{n-1}^i) + \frac{a\Delta t\Delta x}{2}\frac{(\bar{u}_{n+1}^i - 2\bar{u}_n^i + \bar{u}_{n-1}^i)}{\Delta x^2},$$

where we remark the appearance of a diffusion term of purely numerical origin and whose influence is reduced by either reducing the mesh size or the timestep. It is therefore of considerable importance to be aware of the influence of upwind effects in the computation results.

The method just described goes under the name of *Finite Volume Method* (FVM) and is widely used in computational fluid dynamics. The FVM however is a low order method, as the resulting approximation is only first order. In order to obtain increased accuracy, it would be desirable to have high order approximation, and this is what brings us to the Discontinuous Galerkin method discussed in the next section.

1.3.4 The Discontinuous Galerkin method

The ideas behind the Discontinuous Galerkin (DG) method are not dissimilar from those behind the FVM. Indeed, DG can be seen as a FVM where instead of simply averaging the solution, we use a more sophisticated technique.

Before going any further, we must define the discrete space used in the Discontinuous Galerkin method (for a rigorous treatment see [5, 7]).

Let $L^2(\Omega)$ be the standard Lebesgue space of square integrable functions. The discrete DG solution lives in the space

$$V_h := \left\{ v \in L^2(\Omega) : v_{|T_i} \in \mathbb{P}_d^k(T_i) \right\}, \qquad i \in \{1, \ldots, |\mathcal{T}|\}. \tag{1.22}$$

Intuitively, this definition asserts that the functions of V_h are piecewise polynomial functions that can jump on the mesh interfaces.

We also recall that a d-variate polynomial p of degree k can be written as a linear combination of basis functions

$$p(\boldsymbol{x}) = \sum_{j=1}^{N_d^k} p_j \phi_j(\boldsymbol{x}) = \boldsymbol{p}^T \boldsymbol{\phi}(\boldsymbol{x}), \qquad \boldsymbol{x} \in T \tag{1.23}$$

where the linear combination weights are denoted as p_j, the basis functions as $\phi_j(\boldsymbol{x})$; in addition $\boldsymbol{p}$ is the column vector $\{p_j\}_{j\in\{1,\dots,N_d^k\}}$ and $\boldsymbol{\phi}(\boldsymbol{x})$ is the column vector $\{\phi_j(\boldsymbol{x})\}_{j\in\{1,\dots,N_d^k\}}$. Clearly, once a base is chosen and fixed, only the coefficient vector $\boldsymbol{p}$ needs to be stored to fully represent the original polynomial. A function of V_h can therefore be represented as a column vector $[\boldsymbol{p}_1 \,|\, \dots \,|\, \boldsymbol{p}_N]$ which is the juxtaposition of all the coefficients of the polynomials attached to the mesh elements.

In addition, we recall that polynomials can be integrated numerically exactly using quadrature rules. Given a polynomial $p(\boldsymbol{x})$ and a reference element T_{ref}, the integral of p in T_{ref} can be computed as a sum of evaluations of p in the points $\boldsymbol{q}_w$ weighted by the coefficients ω_w specified by a quadrature rule Q:

$$\int_{T_{\text{ref}}} p(\boldsymbol{x})\, dV = \sum_{(\boldsymbol{q}_w,\omega_w)\in Q} \omega_w\, p(\boldsymbol{q}_w).$$

In order to integrate on a generic element T different from T_{ref} using a quadrature, an appropriate change of variable has to be made.

A possible choice for the basis functions ϕ_j are the Lagrange polynomials: for example, if a 1-variate polynomial of degree k in the interval $[x_0, x_k]$ needs to be interpolated, we consider the $k+1$ points $\{x_0, \dots, x_k\}$ and define the functions

$$\ell_j(x) = \frac{(x-x_0)}{(x_j-x_0)} \cdots \frac{(x-x_{j-1})}{(x_j-x_{j-1})} \frac{(x-x_{j+1})}{(x_j-x_{j+1})} \cdots \frac{(x-x_k)}{(x_j-x_k)} = \prod_{\substack{0\le m\le k\\ m\ne j}} \frac{x-x_m}{x_j-x_m}, \tag{1.24}$$

where we remark that $\ell_j(x_m) = \delta_{jm}$, being δ the Kroneker symbol. Clearly, if x_0 and x_k are the endpoints of the element T_n, the Lagrange functions form a polynomial basis on that element; we use the notation $\ell_j^{T_n}$ in that case. The Lagrange functions can be defined also in 2D or in 3D, we refer the reader to the standard finite element literature for the details [8]. Using the Lagrange basis functions, we can define the approximation space as

$$V_h := \bigoplus_{n=1}^{N} \left\{ \ell_j^{T_n} \right\}_{j\in\{1,\dots,N_d^k\}} = \bigoplus_{n=1}^{N} \left\{ \boldsymbol{\ell}^{T_n} \right\}.$$

With all the preliminaries now in place, we can start to discuss the construction of the Discontinuous Galerkin (DG) method. Let $v \in V_h$ be a *test function.* In order to devise the

DG method, we start by requiring that the residual is orthogonal to all the test functions in V_h, or that the expression

$$\int_\Omega \left(\frac{\partial u_h}{\partial t} + \frac{\partial f(u_h)}{\partial x} - g \right) v \, dV = 0, \qquad \forall v \in V_h, \tag{1.25}$$

holds. In particular, as the approximation space is discontinuous and piecewise polynomial, (1.25) is equivalent to requiring for all $T_n \in \mathcal{T}$ that the local statement

$$\int_{T_n} \left(\frac{\partial u_h^{T_n}}{\partial t} + \frac{\partial f(u_h^{T_n})}{\partial x} - g \right) \ell_j^{T_n} \, dV = 0, \qquad \forall j \in \{1, \ldots, N_d^k\}, \tag{1.26}$$

holds. This generates N_d^k equations per element however, in an entirely similar manner to the FVM case, the procedure would not yield a solvable discrete problem: we therefore need to introduce the numerical fluxes f^* in order to fix the values at the mesh interfaces. Before, however, we need to slightly refine their definition. To this aim, we introduce the

Definition 6 (Average and Jump operators)**.** *Let $v : \Omega \to \mathbb{R}$ and let $F \in \Gamma_{int}$ be the face shared by elements T^+, T^-. We define the verage and ump operators as follows:*

$$\textit{Average: } \{v\}_F(x) := \frac{1}{2}\left[v|_{T^+}(x) + v|_{T^-}(x) \right]$$
$$\textit{Jump: } [\![v]\!]|_F(x) := v|_{T^+}(x) - v|_{T^-}(x)$$

If F belongs to the boundary of the domain (i.e. $e \subset \partial T \cap \partial\Omega$):

$$\{v\}_F(x) := v|_T(x), \qquad [\![v]\!]|_F(x) := v|_T(x).$$

If v is vector-valued, the average and jump operators act component-wise.

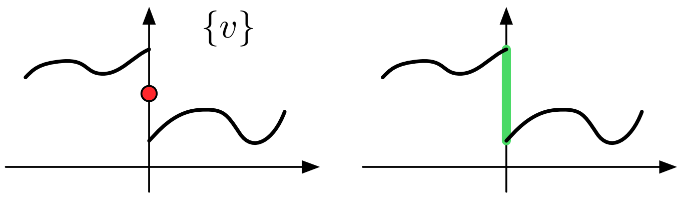


Figure 1.2: Graphical representation of the average (left) and jump (right).

If clear from the context, the subscript specifying the face on which operators act will be omitted.

Using the notation just introduced, the centered fluxes are written as

$$f^* = a\{u_h\},$$

whereas the upwind fluxes take the form

$$f^* = a\{u_h\} + \frac{a}{2}[\![u_h]\!] \cdot \hat{\boldsymbol{n}}.$$

With the new notation in place we proceed locally element-by-element, integrating by parts two times the spatial term of (1.26). With the first integration by parts we obtain

$$\int_{T_n} \frac{\partial u_h^{T_n}}{\partial t} \boldsymbol{\ell}^{T_n}\, dV - \int_{T_n} f(u_h^{T_n}) \frac{\partial \boldsymbol{\ell}^{T_n}}{\partial x}\, dV = \int_{T_n} g\boldsymbol{\ell}^{T_n}\, dV - \int_{\partial T_n} (\hat{\boldsymbol{n}} \cdot f^*)\boldsymbol{\ell}^{T_n}\, dS,$$

and we remark that by replacing $\boldsymbol{\ell}^{T_n}$ with the constant function, the expression collapses back to the FVM. Subsequently, we proceed with a second integration by parts on the spatial term, obtaining

$$\int_{T_n} \frac{\partial u_h^{T_n}}{\partial t} \boldsymbol{\ell}^{T_n}\, dV + \int_{T_n} \frac{\partial f(u_h^{T_n})}{\partial x} \boldsymbol{\ell}^{T_n}\, dV = \int_{T_n} g\boldsymbol{\ell}^{T_n}\; dV + \int_{\partial T_n} \hat{\boldsymbol{n}} \cdot (f - f^*)\boldsymbol{\ell}^{T_n}\; dS.$$

Recalling that locally we approximate a polynomial and we use the Lagrange basis, we have that

$$u_h^{T_n}(x) = \sum_{j=1}^{N_d^k} \mathsf{u}_j\, \ell_j^{T_n}(x),$$

where $\{\mathsf{u}_j\}_{j\in\{1,\dots N_d^k\}} = \mathbf{u}^{T_n}$ are the unknown coefficients of the local polynomial approximating the solution. We can now introduce the element mass matrix $\mathsf{M}_{ij}^{T_n}$ and the element stiffness matrix $\mathsf{S}_{ij}^{T_n}$ as

$$\mathsf{M}_{ij}^{T_n} = \int_{T_n} \ell_j^{T_n}(x)\ell_i^{T_n}(x)dx, \qquad \mathsf{S}_{ij}^{T_n} = \int_{T_n} \frac{\partial \ell_j^{T_n}(x)}{\partial x} \ell_i^{T_n}(x)dx,$$

where the integrals are computed with appropriate quadratures, to finally obtain the semi-discrete scheme

$$\begin{aligned} \mathsf{M}^{T_n} \frac{d\mathbf{u}_h^{T_n}}{dt} + \mathsf{S}^{T_n}\mathbf{f}_h^{T_n} - \mathsf{M}^{T_n}\mathbf{g}_h^{T_n} &= \left(f_h^{T_n}(x^{k+1}) - f^*(x^{k+1})\right) \boldsymbol{\ell}^{T_n}(x^{k+1}) \\ &\quad - \left(f_h^{T_n}(x^k) - f^*(x^k)\right) \boldsymbol{\ell}^{T_n}(x^k). \end{aligned} \tag{1.27}$$

Once a time integration algorithm and the appropriate numerical fluxes are chosen, the scheme is directly implementable.

Computational properties of DG

For the aims of this work, it is of primary importance to realize which are the computational properties of (1.27). Indeed, all the volumetric terms are element-local and can be computed independently and in parallel. The only communication between adjacent elements takes place in the computation of the fluxes.

In addition to the element-level parallelism the matrix-vector products offer another level of parallelism, as they can be parallelized by row. Indeed, if we consider the matrix-vector product $\mathbf{y} = \mathsf{M}\mathbf{x}$, all its entries

$$\mathsf{y}_i = \sum_j \mathsf{M}_{ij}\mathsf{x}_j$$

can be computed in parallel. Taking into account the way in which GPUs work, this enables DG to be extremely efficient on such architectures (See Chapter 2).

Another point that must be noted about (1.27) is that M^{T_n} and S^{T_n} depend on the element T_n and they are potentially all different across the mesh. In order to reduce the computational and memory costs it is possible however to introduce a reference element and have a single mass matrix and a single stiffness matrix: we give the intuition in 2D, the technique is easily extended to 3D. Let T_n be a triangle whose vertices are $\boldsymbol{p}_1^{T_n} = (x_1, y_1)$, $\boldsymbol{p}_2^{T_n} = (x_2, y_2)$ and $\boldsymbol{p}_3^{T_n} = (x_3, y_3)$. In addition, let T_{ref} be the *reference triangle* whose vertices are $\hat{\boldsymbol{p}}_1 = (0,0)$, $\hat{\boldsymbol{p}}_2 = (1,0)$ and $\hat{\boldsymbol{p}}_3 = (0,1)$. Finally, let (x, y) be a pair of coordinates on T_n and (ξ, η) be a pair of coordinates in T_{ref}. An affine transformation T from T_{ref} to T_n is obtained as follows:

1. Define a set of linear basis functions $\hat{N}_i(\xi, \eta)$ on the reference triangle such that $\hat{N}_i(\hat{\boldsymbol{p}}_j) = \delta_{ij}$, where δ is the Kroneker symbol. Such basis functions are

$$\hat{N}_1(\xi,\eta) = 1 - \xi - \eta, \qquad \hat{N}_2(\xi,\eta) = \xi, \qquad \hat{N}_3(\xi,\eta) = \eta.$$

2. Form the linear combination

$$\mathsf{T}(\hat{\boldsymbol{p}}) = \mathsf{T}(\xi,\eta) = \sum_{i=1}^{3} \hat{N}_i(\xi,\eta)\,\boldsymbol{p}_i^{T_n},$$

which can be expanded as

$$\begin{aligned}\begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}(1-\xi-\eta) + \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}\xi + \begin{bmatrix} x_3 \\ y_3 \end{bmatrix}\eta \\ &= \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix}\begin{bmatrix} \xi \\ \eta \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \boldsymbol{T}\hat{\boldsymbol{x}} + \boldsymbol{b}.\end{aligned}$$

It is easy to see that the last expression establishes a bijection between T_{ref} and T_n.

It turns out that the functions $\hat{N}_i$ are the first order Lagrange functions on a triangle, and $N^{T_n}(\boldsymbol{x}) = N^{T_n}(\mathsf{T}(\hat{\boldsymbol{x}})) = \hat{N}_i(\hat{\boldsymbol{x}})$. Therefore, by the change of variable rule for the integrals

$$\mathsf{M}_{ij}^{T_n} = \int_{T_n} N_j^{T_n}(\boldsymbol{x})N_i^{T_n}(\boldsymbol{x})dx = |\det \boldsymbol{T}| \int_{T_{ref}} \hat{N}_j(\hat{\boldsymbol{x}})\hat{N}_i(\hat{\boldsymbol{x}})d\hat{\boldsymbol{x}} = |\det \boldsymbol{T}|\, \mathsf{M}_{ij},$$

where the integral is computed using a quadrature rule, or even manually for the lowest-order cases. It is thus possible to compute the reference mass matrix only once, store it, and obtain the physical mass matrix by a simple rescaling. Even if this procedure requires some computation, especially at high order it enables huge memory savings and data reuse, which are central for a GPU implementation of DG. The same idea can be applied to the stiffness matrix too, however in multiple dimensions the procedure is a bit more involved. In the following we will use the term *differentiation matrix* instead of stiffness matrix; the reason for this will be clear in the next section. Let

$$\nabla = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix}, \qquad \hat{\nabla} = \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{bmatrix}.$$

By the chain rule we have that

$$\boldsymbol{T}^\top \left(\nabla N_i^{T_n} \circ \mathsf{T} \right) = \hat{\nabla}(N_i^{T_n} \circ \mathsf{T}) \qquad \Longrightarrow \qquad \nabla N_i^{T_n} = \boldsymbol{T}^{-\top} \left((\hat{\nabla} \hat{N}_i) \circ \mathsf{T}^{-1} \right),$$

which allows us to write the differentiation matrix in the direction $w \in \{x, y\}$ as

$$\mathsf{D}_{ij,w}^{T_n} = \int_{T_n} \frac{\partial N_j^{T_n}(\boldsymbol{x})}{\partial w} N_i^{T_n}(\boldsymbol{x}) d\boldsymbol{x} = |\det \boldsymbol{T}| \sum_{\hat{w} \in \{\xi, \eta\}} \boldsymbol{T}_{w\hat{w}}^{-\top} \int_{T_{ref}} \frac{\partial \hat{N}_j(\hat{\boldsymbol{x}})}{\partial \hat{w}} \hat{N}_i(\hat{\boldsymbol{x}}) d\hat{\boldsymbol{x}} \tag{1.28}$$

$$= |\det \boldsymbol{T}| \sum_{\hat{w} \in \{\xi, \eta\}} \boldsymbol{T}_{w\hat{w}}^{-\top} \mathsf{D}_{ij,\hat{w}}. \tag{1.29}$$

The last formula in particular is telling us that the differentiation matrix in the physical direction w is computed as a linear combination of the differentiation matrices in the reference directions $\hat{w}$. The linear combination is weighted by the coefficients of $\boldsymbol{T}^{-\top}$, which turns out to be the inverse transpose of the Jacobian of T.

Finally, also the integral appearing in the boundary term can be pre-calculated, and gives rise to the *lifting matrix*. In the 1D case shown above, the lifting matrix reduces to a 2×2 identity matrix, however in the multi-dimensional case the structure depends on the type of element. We postpone the discussion of the lifting matrix to the next section.

1.4 The Discontinuous Galerkin method for Maxwell equations

The 1D conservation law (1.13) is generalized to multiple dimensions as

$$\mathcal{Q} \frac{\partial \boldsymbol{q}}{\partial t} + \nabla \cdot \boldsymbol{F}(\boldsymbol{q}) = \boldsymbol{g},$$

where $\boldsymbol{q} \in \mathbb{R}^n, n \in \mathbb{N}$ is the vector of the n *state variables*, or *state vector*, $\mathcal{Q} \in \mathbb{R}^{n\times n}$ is the matrix of the material parameters, $\boldsymbol{g}$ are the sources and $\boldsymbol{F}$ is the *flux function.* In particular

$$\nabla \cdot \boldsymbol{F} = \sum_w \frac{\partial F_w(\boldsymbol{q})}{\partial w}, \qquad w \in \{x, y, z\},$$

if we assume to be working in $\mathbb{R}^3$. The Maxwell's equations (1.10)-(1.11) can be written in form of a conservation law [2] by setting

$$\mathcal{Q} = \begin{bmatrix} \epsilon & 0 \\ 0 & \mu \end{bmatrix}, \quad \boldsymbol{q} = \begin{bmatrix} \boldsymbol{e} \\ \boldsymbol{h} \end{bmatrix}, \quad \boldsymbol{F} = \begin{bmatrix} -\boldsymbol{F_h} \\ \boldsymbol{F_e} \end{bmatrix}, \quad \boldsymbol{g} = \boldsymbol{0},$$

where, by letting $\hat{\boldsymbol{e}}_i$ be the i-th vector of the canonical basis of $\mathbb{R}^3$,

$$\boldsymbol{F}_A = \begin{bmatrix} (\hat{\boldsymbol{e}}_0 \times \boldsymbol{A})^T \\ (\hat{\boldsymbol{e}}_1 \times \boldsymbol{A})^T \\ (\hat{\boldsymbol{e}}_2 \times \boldsymbol{A})^T \end{bmatrix} = \begin{bmatrix} 0 & A_z & -A_y \\ -A_z & 0 & A_x \\ A_y & -A_x & 0 \end{bmatrix}.$$

Notice that $\nabla \cdot \boldsymbol{F}_A = \nabla \times \boldsymbol{A}$; in addition we consider a piecewise constant flux function. We proceed as usual by testing the differential equation with a test function $\boldsymbol{v}$ and integrating on Ω

$$\mathcal{Q}\frac{\partial}{\partial t}\int_\Omega \boldsymbol{q}\cdot\boldsymbol{v}\, dV, \; + \int_\Omega (\nabla \cdot \boldsymbol{F}) \cdot \boldsymbol{v}\, dV = \int_\Omega \boldsymbol{g}\cdot\boldsymbol{v}\, dS.$$

As in the 1D case we proceed elementwise integrating by parts twice and introducing the numerical flux $\boldsymbol{F^*}$:

$$\mathcal{Q}\frac{\partial}{\partial t}\int_T \boldsymbol{q}\cdot\boldsymbol{v}\, dV + \int_T (\nabla \cdot \boldsymbol{F}(\boldsymbol{q})) \cdot \boldsymbol{v}\, dV = \int_T \boldsymbol{g}\cdot\boldsymbol{v} dV + \int_{\partial T} (\hat{\boldsymbol{n}} \cdot (\boldsymbol{F}(\boldsymbol{q}) - \boldsymbol{F}^*) \cdot \boldsymbol{v}\, dS.$$

Also for the Maxwell's equations it is possible to define the centered and the upwind fluxes [2, 3]. In this case however, the fluxes are not as simple as in the 1D case. The expression for the centered fluxes is

$$\hat{\boldsymbol{n}} \cdot \boldsymbol{F}^* = \begin{bmatrix} -\hat{\boldsymbol{n}} \times \{\boldsymbol{h}\} \\ \hat{\boldsymbol{n}} \times \{\boldsymbol{e}\} \end{bmatrix},$$

whereas the expression for the upwind fluxes is

$$\hat{\boldsymbol{n}} \cdot \boldsymbol{F}^* = \begin{bmatrix} -\hat{\boldsymbol{n}} \times \{\boldsymbol{h}\} - \frac{Y}{2}\hat{\boldsymbol{n}} \times (\llbracket \boldsymbol{e} \rrbracket \times \hat{\boldsymbol{n}}) \\ \hat{\boldsymbol{n}} \times \{\boldsymbol{e}\} - \frac{Z}{2}\hat{\boldsymbol{n}} \times (\llbracket \boldsymbol{h} \rrbracket \times \hat{\boldsymbol{n}}) \end{bmatrix},$$

where $Z = Y^{-1} = \sqrt{\mu/\epsilon}$ [2, 3]. In addition, on the boundary it holds

$$\hat{\boldsymbol{n}} \cdot \boldsymbol{F} = \begin{bmatrix} -\hat{\boldsymbol{n}} \times \boldsymbol{h} \\ \hat{\boldsymbol{n}} \times \boldsymbol{e} \end{bmatrix},$$

therefore it is now possible to compute the expression of the whole surface term on the right-hand side in the two cases. For the centered fluxes we have

$$\hat{\boldsymbol{n}} \cdot (\boldsymbol{F} - \boldsymbol{F}^*) = \frac{1}{2} \begin{bmatrix} \hat{\boldsymbol{n}} \times [\![\boldsymbol{h}]\!] \\ -\hat{\boldsymbol{n}} \times [\![\boldsymbol{e}]\!] \end{bmatrix},$$

whereas for the upwind fluxes we have

$$\hat{\boldsymbol{n}} \cdot (\boldsymbol{F} - \boldsymbol{F}^*) = \frac{1}{2} \begin{bmatrix} \hat{\boldsymbol{n}} \times [\![\boldsymbol{h}]\!] - Y\hat{\boldsymbol{n}} \times ([\![\boldsymbol{e}]\!] \times \hat{\boldsymbol{n}}) \\ -\hat{\boldsymbol{n}} \times [\![\boldsymbol{e}]\!] - Z\hat{\boldsymbol{n}} \times ([\![\boldsymbol{h}]\!] \times \hat{\boldsymbol{n}}) \end{bmatrix}.$$

We remark that in both cases the surface terms are functions of only the jumps of the fields, fact that must be taken into account in the implementation.

The compact form of the equation can now be expanded in the three electric and the three magnetic components. In the case of the centered fluxes for example, we obtain the element-local electric equations

$$\begin{aligned}
\epsilon\frac{\partial}{\partial t}\int_T e_x v\,dT - \int_T \frac{\partial h_z}{\partial y} v\,dT + \int_T \frac{\partial h_y}{\partial z} v\,dT &= \frac{1}{2}\int_{\partial T} (\hat{n}_y[\![h_z]\!] - \hat{n}_z[\![h_y]\!]) v\,dS - \int_T j_x v\,dT,\\
\epsilon\frac{\partial}{\partial t}\int_T e_y v\,dT + \int_T \frac{\partial h_z}{\partial x} v\,dT - \int_T \frac{\partial h_x}{\partial z} v\,dT &= \frac{1}{2}\int_{\partial T} (\hat{n}_z[\![h_x]\!] - \hat{n}_x[\![h_z]\!]) v\,dS - \int_T j_y v\,dT,\\
\epsilon\frac{\partial}{\partial t}\int_T e_z v\,dT - \int_T \frac{\partial h_y}{\partial x} v\,dT + \int_T \frac{\partial h_x}{\partial y} v\,dT &= \frac{1}{2}\int_{\partial T} (\hat{n}_x[\![h_y]\!] - \hat{n}_y[\![h_x]\!]) v\,dS - \int_T j_z v\,dT,
\end{aligned} \tag{1.30}$$

where j_x, j_y and j_z are the volumetric sources, and the element-local magnetic equations

$$\begin{aligned}
\mu\frac{\partial}{\partial t}\int_T h_x v\,dT + \int_T \frac{\partial e_z}{\partial y} v\,dT - \int_T \frac{\partial e_y}{\partial z} v\,dT &= \frac{1}{2}\int_{\partial T} (\hat{n}_z[\![e_y]\!] - \hat{n}_y[\![e_z]\!]) v\,dS,\\
\mu\frac{\partial}{\partial t}\int_T h_y v\,dT - \int_T \frac{\partial e_z}{\partial x} v\,dT + \int_T \frac{\partial e_x}{\partial z} v\,dT &= \frac{1}{2}\int_{\partial T} (\hat{n}_x[\![e_z]\!] - \hat{n}_z[\![e_x]\!]) v\,dS,\\
\mu\frac{\partial}{\partial t}\int_T h_z v\,dT + \int_T \frac{\partial e_y}{\partial x} v\,dT - \int_T \frac{\partial e_x}{\partial y} v\,dT &= \frac{1}{2}\int_{\partial T} (\hat{n}_y[\![e_x]\!] - \hat{n}_x[\![e_y]\!]) v\,dS.
\end{aligned} \tag{1.31}$$

The surface sources (for example a plane wave source) do not appear in the above equations and can be implemented simply by modifying the jumps [3].

The equations so obtained resemble closely what we obtained in the 1D case. In particular, we remark that for each equation there is a mass term and two differentiation terms, in addition to the flux term that we are going to discuss.

In a setting where a nodal basis (like the Lagrange basis) is used, the jumps are computed simply by subtracting the degrees of freedom on the face shared by two adjacent triangles T_1 and T_2 (Figure 1.3 left). The process is done between each pair of adjacent

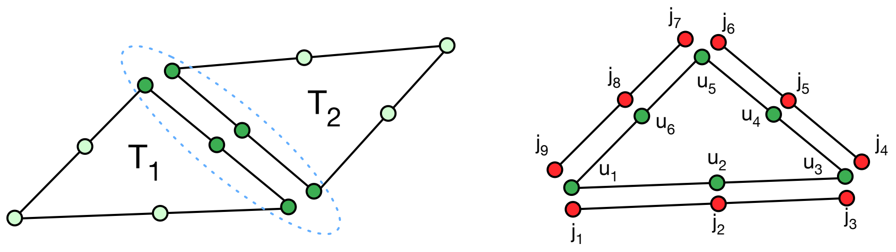


Figure 1.3: Degrees of freedom involved in the computation of the fluxes between two triangles T_1 and T_2 (left) and association between flux DoFs and element DoFs (right). Both pictures are in the case where Lagrange basis functions are used.

elements, obtaining a new vector of jumps for each face (Figure 1.3 left). The jumps so obtained are subsequently used, as we remarked before, to compute the flux in the boundary of the element with an expression comparable to

$$\int_{\partial T} \alpha [\![u]\!] v \, dS = \sum_{F \in \partial T} \int_F \alpha [\![u]\!]_{|F} v \, dS, \tag{1.32}$$

where we split the integral on the boundary in a sum of integrals on the single faces. We recognize that the integral on the face F is actually the mass matrix M^F of F whose entries we denote as m^F_{ij}. For second order elements for example, such a mass matrix has dimension 3×3. In that case, if we denote with $\mathsf{j} = \{j_i\}_{i \in \{1,\dots,9\}}$ the vector collecting all the jumps on the faces of a triangle T (Figure 1.3 left) and with $\mathsf{u} = \{u_i\}_{i \in \{1,\dots,6\}}$ the local DoFs of T, we realize that a term like (1.32) is computed as the matrix-vector product $\mathsf{u} = \mathsf{L}\mathsf{j}$ where

$$\mathsf{L} = \left[\begin{array}{ccc|ccc|ccc}
m^{F_1}_{11} & m^{F_1}_{12} & m^{F_1}_{13} & & & & m^{F_3}_{31} & m^{F_3}_{32} & m^{F_3}_{33} \\
m^{F_1}_{21} & m^{F_1}_{22} & m^{F_1}_{23} & & & & & & \\
m^{F_1}_{31} & m^{F_1}_{32} & m^{F_1}_{33} & m^{F_2}_{11} & m^{F_2}_{12} & m^{F_2}_{13} & & & \\
 & & & m^{F_2}_{21} & m^{F_2}_{22} & m^{F_2}_{23} & & & \\
 & & & m^{F_2}_{31} & m^{F_2}_{32} & m^{F_2}_{33} & m^{F_3}_{11} & m^{F_3}_{12} & m^{F_3}_{13} \\
 & & & & & & m^{F_3}_{21} & m^{F_3}_{22} & m^{F_3}_{23}
\end{array}\right].$$

From the previous discussion we can deduce that, once implemented, the resulting computational structure will be that depicted in Figure 1.4. In particular, we recognize a volumetric path and a boundary path. The volumetric path consists of the following phases:

- Differentiate fields: the spatial derivatives of the six fields are computed,
- Compute curls: using the derivatives computed in the previous step, the curls are computed by subtracting the degrees of freedom of the corresponding fields.

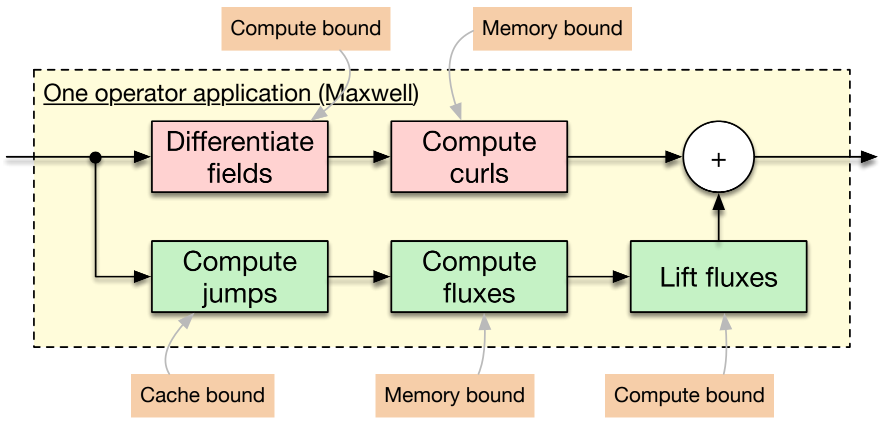


Figure 1.4: Computational steps required in order to do one application of the Discontinuous Galerkin operator.

The boundary path, on the other hand, corresponds to the following phases:

- Compute jumps: compute all the quantities $[\![\cdot]\!]$ appearing in the equations and store them,
- Compute fluxes: using the jumps computed in the previous step, compute the full expressions appearing inside the integrals on the right-hand sides of (1.30) and (1.31),
- Lift fluxes: bring the fluxes from the surfaces to the volumes using the matrix L.

Time integration

In the GMSH/DG code, the time integration of the equations is done via the Runge-Kutta 4 method or via the leapfrog method. Both methods are standard and we refer the reader to [1] and [3] for the details.

Chapter 2

Graphics Processing Units

GPUs were born as specialized electronic circuits designed to accelerate image manipulation and processing. However, their highly parallel structure can be used for general-purpose computing, especially when dealing with embarrassingly parallel problems, to reach better performance by mixing GPU and CPU execution.

GPUs must not be seen as a panacea though, instead they have to be seen as a useful resource that can be used when certain conditions are met. In fact, not all tasks can be parallelized in the expected way on GPUs: some of them are embarrassingly parallel and benefit a lot from GPU parallelization; others on the other hand may have many dependencies between data, which makes using GPUs pointless.

Another crucial difference between CPUs and GPUs is that the first can run tens of threads in parallel, are equipped with broad instruction sets and are more devoted to data caching and control flow; GPUs instead can run thousands of threads in parallel, which means that most transistors are devoted to data processing and very few are used for control flow and caching (an example of this can be seen in Figure 2.1). The computing power derived from the transistors is exploited using the SIMT (Single Instruction, Multiple Thread) execution model in which a bundle of thread (possibly) executes the same instruction on usually different data.

Finally, it has to be noted that the programming of parallel systems is not an easy task. In the early days of general-purpose computing on GPUs (GPGPU), solving a problem on a GPU required it to be reformulated to fit into the graphics pipeline, for example arrays would become textures, the processing to be done would be a shader and the computing was done during the rendering phase: this was the only interface to GPU hardware. Modern GPGPU makes developing solutions for GPUs easier, with the emergence of programming models and platforms like CUDA and OpenCL, which allow developers to ignore the underlying graphical concepts in favor of high-performance computing concepts. Some of these platforms are developed by graphics unit vendors, like CUDA for NVIDIA GPUs [9] and ROCm for AMD GPUs [10], while others are open standards for heterogeneous computing such as OpenCL [11] and SYCL [12], that also support other accelerators like FPGAs, DSPs and custom chipsets [12]. As CUDA is our platform of choice, the rest of this chapter will be heavily based on the CUDA Toolkit documentation [9].

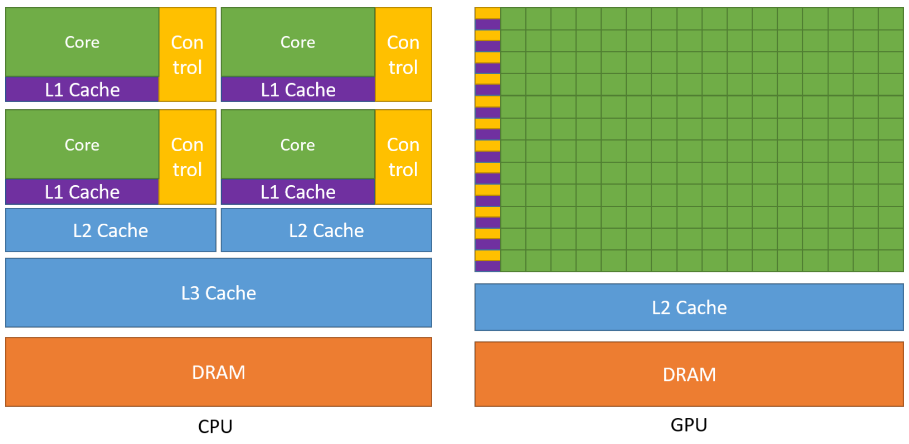


Figure 2.1: GPUs devote more transistors to data processing (in green) and have much fewer transistors for control units (in yellow) and caches (in purple) (NVIDIA Corporation 2022).

2.1 Architecture fundamentals

GPU cards are accelerators connected to the CPU via a bus (e.g. a PCIe bus) which is used for data exchange. The typical steps involved in using a GPU for GPGPU are the following:

1. Data is copied to the GPU memory
2. Computation is performed on the GPU
3. Results are copied back to the CPU

Different programming models have different terminology to refer to the components of GPUs, we will use CUDA terminology because it is the one used throughout the work of this thesis.

CUDA is a general purpose parallel computing platform and programming model introduced by NVIDIA in 2006 for their GPUs to truly expose their devices as a generic data-parallel computing device. Before CUDA, GPU could only be programmed using a graphics API, imposing a high learning curve to the novices and the overhead of an inadequate API to non-graphical applications.

The major point of CUDA was thus having a completely dedicated programming model for developing general-purpose applications in a way that is familiar to specialists in parallel computing. Therefore, from that point on a GPU could be really used as a highly-parallel device and as such, organized in a set of processors, called *Streaming Multiprocessors*, made of compute elements, called *CUDA Cores* that are hardware units that run computations. Figure 2.2 shows how a GPU device is organized. This type of organization is much more similar to a collection of CPUs, each with its own cores, rather than a "black-box" accessible through a graphics API.

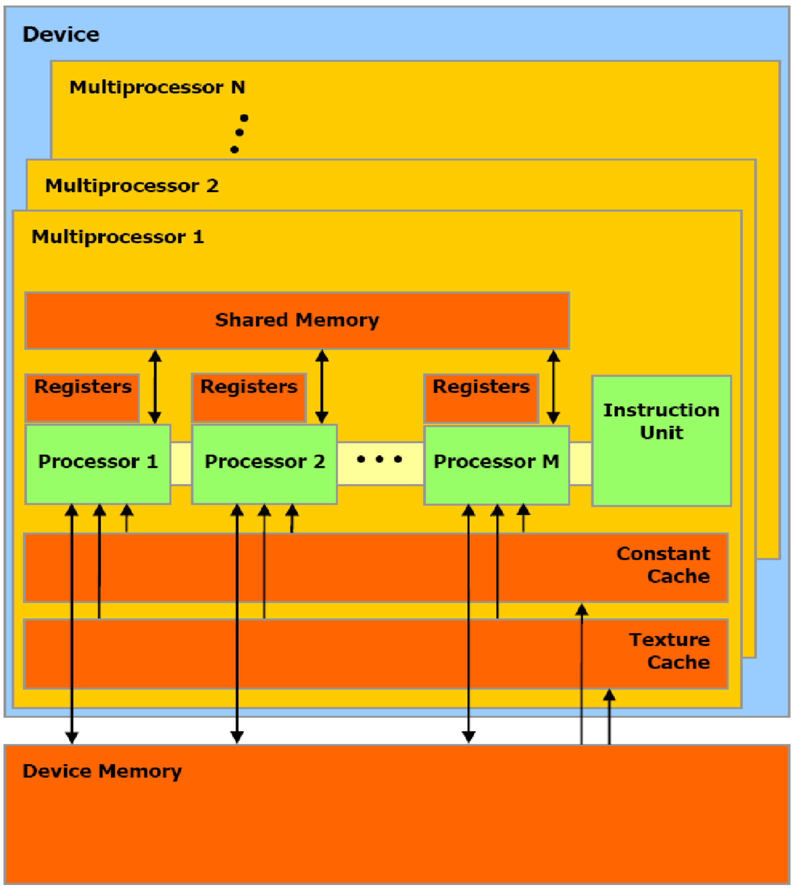


Figure 2.2: The hardware model of an NVIDIA GPU, showing the array of streaming multiprocessors with their on-chip memories, cores and instruction schedulers and the off-chip device memory (NVIDIA Corporation 2007).

2.1.1 CUDA Compute Capability

In CUDA terminology, the *Compute Capability* (CC) of a device is a version number that identifies the features supported by the GPU hardware. Knowing the hardware compute capability is crucial to fine tune the performance of applications; in fact, many strategies to improve performance are based on values that depend on the hardware version.

Compute capability version number is composed by a major revision number X and a minor revision number Y, and it is denoted by $X.Y$. Devices with the same revision number share the same *core architecture* which is identified by a codename. Some of core architectures are consumer-oriented, while others are HPC-oriented. The latter have a much higher bandwidth thanks to the use of HBM memory instead of GDDR memory and have more double-precision floating-point core units than the consumer-oriented GPUs. In this work, when analyzing single GPU performance, we worked with compute capability 7.0, based on the *Volta* core architecture.

2.1.2 Thread execution model

The programming model allows the developer to define C++ functions, called *kernels*, which are executed in parallel by N threads on N different CUDA cores. CUDA Threads are logically grouped in *thread blocks* that can be organized into a one-dimensional, two-dimensional or three-dimensional layout to provide a natural way to invoke a kernel on different data layouts. Blocks are then also organized into a one-dimensional, two-dimensional or three-dimensional *kernel grid* of thread blocks. An example of this organization can be seen in Figure 2.3, showing a bidimensional grid in which also blocks are also bidimensional. The dimensions of grids and blocks are independent of each other.

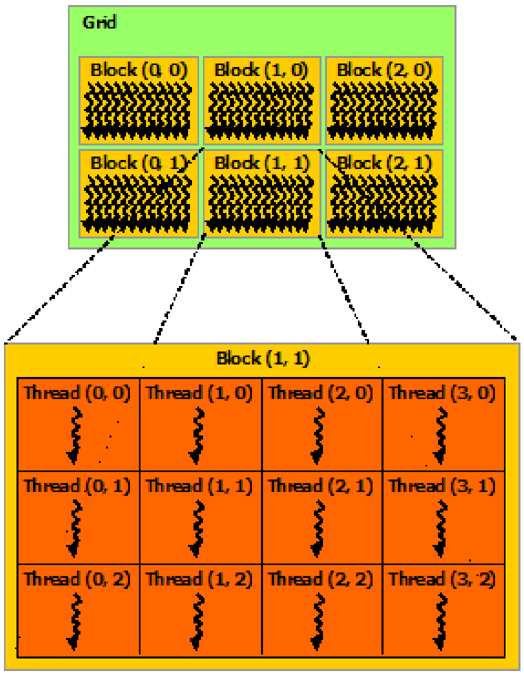


Figure 2.3: Example of a bidimensional grid with bidimensional blocks (NVIDIA Corporation 2022).

From a hardware point of view, when a kernel is launched, the thread blocks are distributed among the streaming multiprocessors of the GPU. When a multiprocessor is given a thread block, it partitions it into groups of 32 threads called *warps* and each of them is issued one instruction by the instruction scheduler, called *warp scheduler*. The way a block is partitioned into warps is always the same, in fact each thread has a thread ID with respect to the grid, and each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

Virtually, each thread of a warp may need to run a different instruction, for instance the pseudocode in Algorithm 1 shows a situation in which half of the warp executes an instruction and the other half another one. After the branch instruction has been executed only one instruction can be issued by the scheduler to the whole warp, thus half of the warp will be inactive while the other half will work.

```
if $threadId < 16$ then
    do something;
else
    do something else;
end
```

Algorithm 1: Pseudocode showing a situation in which some threads of the warp will be inactive while the others work.

So, the idea of SIMT execution model that GPUs implement is to make an efficient use of the limited warp schedulers – these are usually four per streaming multiprocessor – by having all threads performing the same instruction.

To fully exploit the capabilities of the processing elements the GPU provides, the organization of computation is usually different from a multicore computation on CPU. For instance, let us consider the vector sum between two vectors $v = (v_1, \dots, v_n)$ and $u = (u_1, \dots, u_n)$. In the usual "CPU approach" one would loop over the index of the vectors and put the result in a new vector w, and in a multicore context we would split the vector into chunks and give each chunk to a process. On GPU the approach is different, in the sense that we use a much finer granularity: we launch a grid of n threads in which each thread computes the sum $w_i = v_i + u_i$, for $i \in \{1, \dots, n\}$. Thus, computation on GPU is usually organized so that one thread is responsible for computing one output value of the given problem.

The Discontinuous Galerkin method is more complex than a vector sum: as explained in Subsection 1.3.4 the method ultimately approximates the solution by using piecewise polynomial functions attached to the mesh elements. Thus, to approximate the solution, once a base function is fixed (e.g. Lagrange polynomials) the values that the method will compute are the coefficients of the polynomials, that are called *degrees of freedom*. As before, even if the computation is divided in many phases as will be shown in Chapter 3, the code organization of each kernel is such that each thread is responsible to perform the computations related to a single degree of freedom. This means that in the heaviest computations we deal with millions of parallel threads, which may seem too much, but actually it is exactly what we want – to a certain degree – to obtain maximum occupancy and latency hiding, concepts that we will explain further in the upcoming section.

Having more threads also means more thread blocks, which are a very handy abstraction for dealing with scalability. As Figure 2.4 shows, GPUs with more streaming multiprocessors will automatically be more performant with respect to a GPU with fewer multiprocessors, because the thread blocks will be distributed among more multiprocessors allowing for a more efficient parallelization.

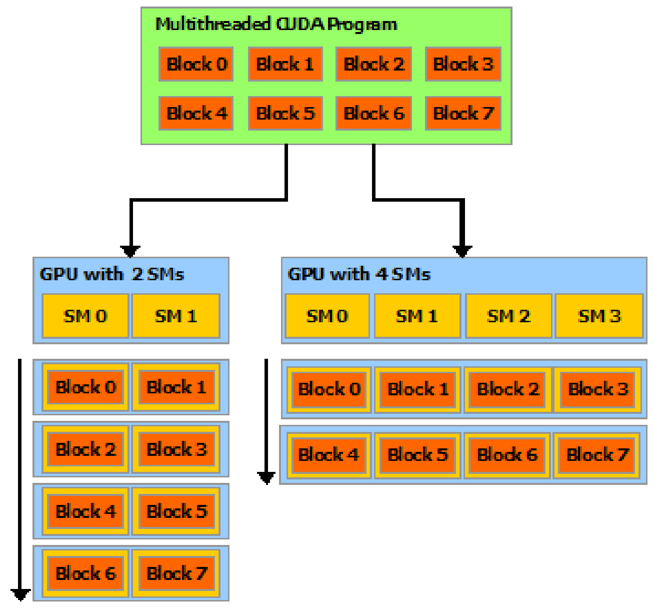


Figure 2.4: Example of how a kernel grid can automatically scale depending on the number of streaming multiprocessors on a GPU (NVIDIA Corporation 2022).

2.1.3 GPU memories

From a hardware point of view, we can find two types of memories on a GPU board: device memory (SDRAM) and on-chip caches. High-performance graphics accelerators for general-purpose computing usually use High Bandwidth Memory (HBM) as SDRAM while consumer GPUs for gaming usually employ GDDR memories. On-chip memories include a L2 cache that is shared by all SMs and other caches for each SM that depend on the device compute capability. A generic view of this organization can be seen in Figure 2.2.

Hardware memories are organized in different ways from a software point of view based on what they memorize, how they exploit caches and what they are optimized for.

Global memory Global memory resides in device memory, is the same for all threads, and it is where data from the host CPU is transferred and from which is fetched by the CPU to get the results. Global memory accesses from the GPU are supported by the L2 cache. This memory is persistent across kernel launches by the same application.

Registers Each thread has access to a *private memory*, used for local variables in a kernel. These memory locations are called registers and there are a limited number of these which depends on the device compute capability.

Local memory Another read-write memory area is the local memory, which also resides in device memory and therefore has the same performance of global memory. Local memory accesses occur only for some type of variables, such as arrays indexed with non-constant quantities, large structures or arrays that would consume too much register space, and *register spills* meaning any variable if the kernel uses more registers than available. Register spills are cached in L1 cache. Normally, local memory accesses should be avoided to achieve good performance.

Texture and surface memory Texture and surface memory are two kinds of memories that reside on device memory. Accessing texture memory is done by using the texture object API, thus creating texture objects (a piece of texture memory) at runtime that cannot be subsequently modified. A texture can be addressed as a one-, two- or three-dimensional array, while elements of the texture are called *texels* and can store integers or single floating-point values. The texture width, height, and depth refer to the size of the array in each dimension. An interesting capability of textures fetching is that during fetching the value returned by specifying the coordinates can be interpolated with the neighboring values, thus performing linear interpolation for one-dimensional textures, bilinear interpolation for two-dimensional textures, and trilinear interpolation for three-dimensional textures.

Surface memory is also accessed through its API, by creating surfaces objects, that differently from textures, can be modified through the API calls. Thus, surfaces are very similar to textures, except for being writable and not supporting interpolation during fetching.

Both texture and surface memory accesses are cached into an on-chip texture cache, that depending on the compute capability of the device could be dedicated or unified with other memories.

Shared memory Each thread has access to a shared memory visible to all threads of the block, this memory resides on-chip and because of that, it has higher bandwidth and lower latency than global memory. Threads within a block can cooperate by sharing data through this memory as a kind of communication between threads, however one has to enforce synchronization across threads in the same block to make use of data among different warps. This is due to the fact that the scheduler operates on warps, so for instance a warp that need to read from shared memory some data written by another warp could read incorrect data if synchronization had not been enforced. In order to obtain the best performance, the access to shared memory must follow some conditions that will be explained in Section 2.2.2.

Constant memory Finally, constant memory is a read-only memory which resides on device memory and that is cached in constant cache.

Constant, texture, L1 caches and shared memory organization on the streaming multiprocessor change for each compute capability, therefore to get a clearer picture we summarize caches and their uses in Table 2.1.

CC 3.x	Dedicated read-only constant cache Unified L1 cache for local memory and shared memory Dedicated read-only data cache for texture memory
CC 5.x	Dedicated read-only constant cache Unified L1/texture cache for global and texture memory Dedicated shared memory
CC 6.x	Dedicated read-only constant cache Unified L1/texture cache for global and texture memory Dedicated shared memory
CC 7.x	Dedicated read-only constant cache Unified L1/texture cache and shared memory
CC 8.x	Dedicated read-only constant cache Unified L1/texture cache and shared memory

Table 2.1: How memories organization on the streaming multiprocessor change per each compute capability.

2.2 Performance optimization

To achieve the best performance on GPU one has to have a deep knowledge of the platform on which a solution is being implemented and of the problem itself. The CUDA programming guide asserts that performance optimization revolves around some basic strategies:

- Maximize parallel execution to achieve maximum utilization;
- Optimize memory usage to achieve maximum memory throughput;
- Optimize instruction usage to achieve maximum instruction throughput;

A software using a GPU as an accelerator, may be composed by different parts that require different kinds of optimization. The usual performance limiters are the floating-point operation throughput and the memory throughput, depending on the kernel.

2.2.1 Maximize utilization

To make the most of the GPU resources, applications must be developed so that it is possible to exploit as much as possible parallelism over the components of the system. The first kind of parallelism to be exploited is the one between host (CPU) and devices (GPU): one has to design their application to have all components working at the same time.

At device level, parallelism is exploitable by executing different kernels concurrently by using *streams*. A stream is a sequence of commands (e.g. CUDA API calls or kernel launches) that execute in order; in the code a stream is represented by a stream object which can be created through the CUDA API. Different streams may execute their commands out of order with respect to one another or concurrently, thus allowing for concurrent kernel execution if enough resources are available.

Eventually, the last level is the multiprocessor one. At this level, we want to maximize resident warps, meaning that at any time, we want to have an instruction ready for each warp scheduler. In fact, at every instruction issue time, a warp scheduler gets a ready warp and issues it an instruction. The time that a warp takes to be ready, measured in number of clock cycles, is called *latency*. Therefore, full utilization is achieved when all warp schedulers always have some instruction to issue for some warp during that latency period, completely *hiding* the latency. Usually a warp is not ready because it is waiting for the instruction's operand, which means that it is waiting for a memory read. This shows how important it is to fully exploit the memory hierarchy, taking advantage of registers and on-chip memories and minimizing off-chip transactions.

2.2.2 Maximize memory throughput

To maximize memory throughput, we want to minimize data transfers with low bandwidth. This means minimizing data transfers between host and device and global memory data transfer, by exploiting shared memory and caches. Shared memory in particular is equivalent to a user-managed cache: the developer has to allocate and access it. The usual pattern when using shared memory is the following:

- Load data from global memory to shared memory
- Synchronize over the thread block
- Process data using shared memory
- Synchronize again, if necessary
- Write the results to global memory

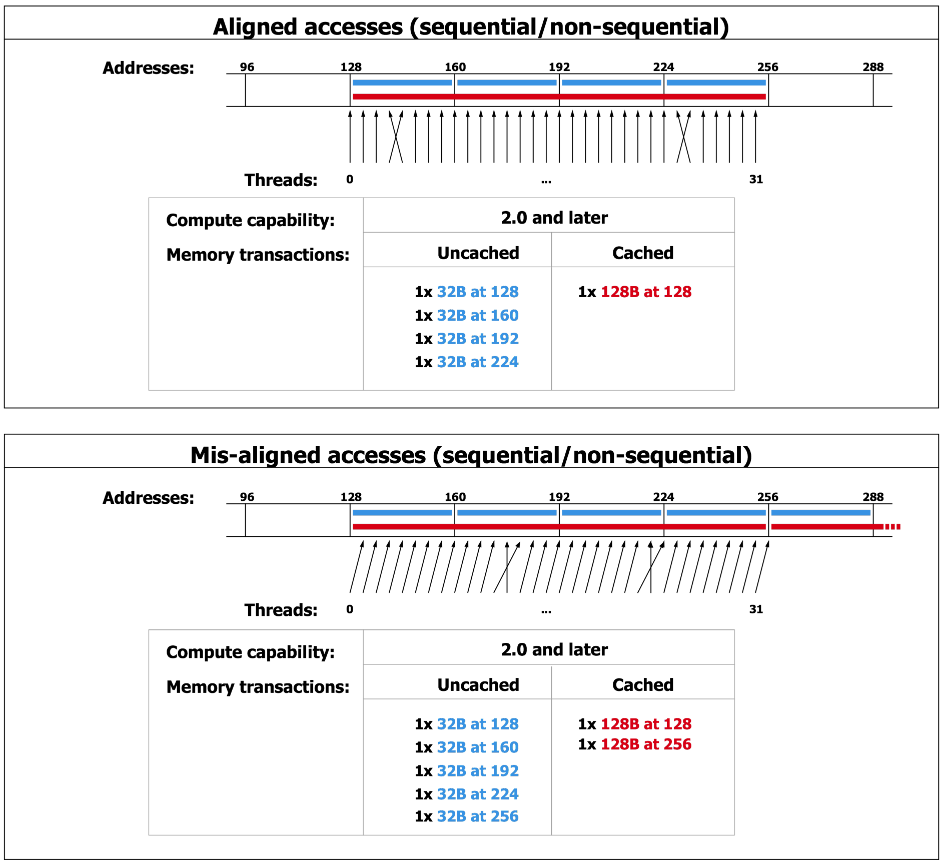
The memory throughput is also influenced by the access pattern used to access each type of memory. Optimizing memory access patterns is particularly important for memories with high latency, such as the global memory and is also important to exploit the full bandwidth when working with shared memory.

Global memory Global memory is accessed via 32-, 64-, or 128- byte memory transactions. These memory transactions must be aligned with the same byte size, meaning that only addresses multiple of 32, 64, or 128 should be read or written by memory transactions.

Thus, to maximize memory throughput we would like a global memory access to be *coalesced*, meaning that the threads in the warp access to a contiguous memory area, this allows to reduce the number of memory transactions. Along with the coalescing we also would like to access addresses aligned to the transaction size to completely minimize memory transactions. In Figure 2.5 an example of coalesced access is shown, along with the consequences of a misaligned memory access. The converse of this is that the more scattered addresses are, the more reduced the throughput will be.

Accesses to the global memory are always cached in L2 for CC 3.x and higher. For devices of CC 3.x the use of L1 cache can be customized at compile time, while for CC 5.x and higher caching in L1 cache is only performed for data that is read-only for the entire lifetime of the kernel. When using L1 cache, a load attempts to hit L1 cache, then L2 and eventually global memory, and issues a 128-byte load. Uncached mode only attempts to hit L2 cache and then global memory, here the lead granularity is 32-byte and can reduce overfetching in case of scattered memory accesses. The results of cached and uncached mode can be seen in Figure 2.5

Shared memory Shared memory is an on-chip memory, with much lower latency than global memory. It is divided into equally-sized memory modules called *banks*, which can be accessed simultaneously. So if there are some memory requests to n addresses that fall in n distinct memory banks, those can be served simultaneously, yielding an overall bandwidth which is n times as high as the bandwidth of a single module.


Aligned accesses (sequential/non-sequential)
Addresses:
96
128
160
192
224
256
288
Threads:
0
...
31
Mis-aligned accesses (sequential/non-sequential)
Addresses:
96
128
160
192
224
256
288
Threads:
0
...
31

Aligned accesses (sequential/non-sequential)

Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Mis-aligned accesses (sequential/non-sequential)

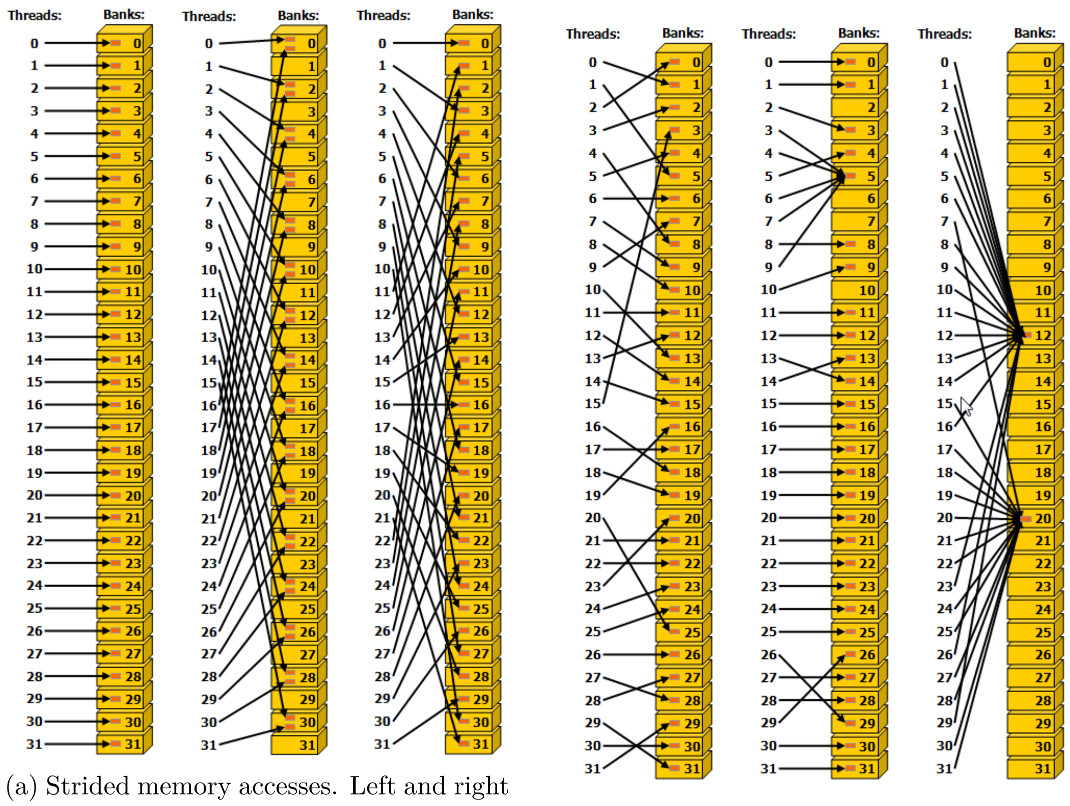
Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

Figure 2.5: Example of a coalesced global memory access with aligned and misaligned addresses, also showing how an uncached memory access may reduce overfetching (NVIDIA Corporation 2022).

In CC 5.x and higher shared memory has 32 banks, organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle, which implies that reading a 64-bit word for each thread in a warp requires *two* shared memory transactions. This is a disadvantage when an application is operating with double-precision values.

If two addresses of a memory request fall in different words of the same memory bank there is a *bank conflict* and the access has to be serialized. The hardware splits a memory request into many conflict-free requests, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts.

In CC 5.x and higher two (or more) thread accessing the same word in the same bank do not generate a bank conflict: in fact, the word is broadcasted for read accesses, while for write accesses only one of the threads will write on the bank (which one is undefined). Some examples of accesses to shared memory are shown in Figure 2.6, regarding both regular and irregular accesses to the memory banks.



(a) Strided memory accesses. Left and right have no conflict, center shows a 2-way bank conflict.

(b) Irregular memory accesses. All memory accesses are conflict-free.

Figure 2.6: Examples of shared memory accesses (NVIDIA Corporation 2022).

Texture memory Texture memory resides in device memory and is cached in *texture cache*, so a texture fetch costs one memory read from device memory only on a cache miss. The texture cache is optimized for 2D spacial locality, to achieve best performance reading addresses close in 2D. Also, it is designed for streaming fetches with a constant latency; a cache hit reduces DRAM bandwidth demand but not fetch latency.

2.2.3 Maximize instruction throughput

As a first step, achieving maximum instruction throughput is only possible if memory loads latency are completely hidden, as discussed in Subsection 2.2.1, thus always having a ready warp for each scheduler.

Now, let us focus on the instructions themselves. Arithmetic instructions have a throughput, namely the number of results per clock cycle, thus if we want to maximize instruction throughput, we want also to maximize the number of instruction with high throughput because they will give more results in the same time unit.

Other types of instructions that significantly impact performance are flow control instruction: if the threads of a warp *diverge* because of a conditional branch, then the warp has to execute each path separately, disabling the threads that are not on that path and increasing the number of instructions executed from that warp. If the control flow depends on the thread ID then the controlling condition should be written to minimize the number of divergent warps.

Finally, reducing the number of instructions can increase throughput. For example by optimizing out synchronization points whenever possible or by using the `__restrict__` keyword in pointers declarations of CUDA kernels. The use of this keyword hints to the compiler that for the lifetime of the pointer, no other pointer will be used to access the data to which it points. This has to be specified to the compiler due to the fact that in C/C++ languages two different pointers can point to the same data, this situation goes under the name of *pointer aliasing*. Thus, by using the ___restrict___ keyword, we are saying to the compiler that all restricted pointers refer to not overlapping memory locations. This in turn allows the compiler to optimize memory loads and to perform code re-ordering and common sub-expression elimination, even though this could cause an increase of register pressure, which may lead to negative code performance.

Chapter 3

Discontinuous Galerkin solver

In this chapter the focus will be put upon the implementation of the DG solver. We will not focus on the theoretical point of view of the method, but on how it can be mapped on GPU and CPU hardware. The first section of this chapter will present other implementations based on GPU hardware, so that we can compare the various approaches followed in our solver Gmsh DG.

Gmsh DG will then be introduced and its functioning will be explained in detail. The solver consists in a CPU and GPU implementation, with MPI parallelization for both CPU and GPU. The DG solver architecture will be shown, and we will illustrate the most important computational steps in the current solver for Maxwell's equations along with an intuition on their bottlenecks. The final section illustrates the parallelization strategy. Looking at the problem from different perspectives we will give the insight on how to arrange the computation so that it can adapt to multicore, single-GPU, and, by mixing the two strategies, multi-GPU systems.

3.1 Solvers on GPU: state of art

In order to gain some insight on the various approaches that can be followed to implement the DG method on GPU, we will make a historical survey of GPU implementations of numerical methods applied to different problems, in particular we will focus on how the problem has been mapped onto the GPU architecture.

One of the first usage of GPUs in computational electromagnetics can be found in the work of Takada et al. [13] dating back to 2009, targeting 2-dimension FDTD for Maxwell's equations. The approach followed by the authors consisted in dividing the domain of the FDTD simulation into small subdomains in order to store them in the shared memory during the kernel execution. The values of the electromagnetics fields were stored in global memory and during the execution of kernels were loaded into the shared memory of each streaming multiprocessor. The analysis of the results was carried out over a consumer-oriented GPU board, the NVIDIA GeForce GTX 280, featuring compute capability 1.3, and the results show a 20-fold maximum speedup comparing the GPU computation with a conventional CPU one.

Also in 2009, Klöckner et al. implemented a solver for Maxwell's equations using the

DG method [14]. The authors also carried out their computation and measurements on an NVIDIA GTX 280. DG is very different and more complex than FDTD: one moves from regular grids to unstructured meshes, and one deals with finite elements; hence the mapping onto the GPU hardware can be much more complicated.

In order to store the degrees of freedom of the elements, the authors use a layout that enforces memory alignment: they pack together a certain number of elements (i.e. tetrahedra) and add some padding at the end so that their size is multiple of the transaction size, these *microblocks* are then stored contiguously one after the other. Microblocks are mapped to thread blocks during the computation of kernels and the approach is that each thread has to compute the output value of one degree of freedom.

Because the method features frequent matrix-matrix multiplication and matrix-vector multiplication, the algorithm developed by the authors use shared memory to reduce repeated global memory accesses and to share data among threads. The use of shared memory enforces a certain organization of the computation: shared memory is a scarce resource, thus simulations with many elements require multiple rounds of fetching data from global memory because otherwise shared memory would be saturated. Having multiple fetch rounds requires synchronization of threads in order to avoid race conditions in thread blocks. This situation becomes even worse when considering double-precision values. Furthermore, code complexity arises due to the fact that the computation has to be organized so that it avoids bank conflicts. The implementation on GPU accelerated the solver performance by a factor of 50 relative to a serial computation on the CPUs of that period.

In 2010 a DG implementation for a multi-GPU system [15] was introduced by Gödel et al. The parallelization strategy of the authors for the multi-GPU is based on three levels of abstraction: at the first level the mesh is partitioned, and each partition is assigned to a different GPU; at the second level each finite element is mapped to a single thread block; finally, on the last level each degree of freedom is mapped to a CUDA thread. Thus, at the thread block level, the approach is different from the previous work: the microblocks approach of Klöckner et al., in which many elements are assigned to a single thread block, is abandoned in order to simplify the code [15].

Another way to organize computation, and data, is presented in the work of Fuhry, Giuliani, and Krivodonova [16] dated 2014, in which the DG method is used to solve nonlinear hyperbolic conservation laws. The main differences lie in the data layout for the degrees of freedom and in the approach for the parallelization on GPU. The data layout is organized so that degrees of freedom corresponding to one basis function, over different elements, are placed side by side, differently from previous works that store the degrees of freedom element-wise. Furthermore, their approach to parallelization is based on a *one-thread-per-element* approach, thus each thread will compute the value for the entire element, or edge, depending on the kernel. The implementation uses double-precision floating-point values and does not make use of any strategy to enforce memory alignment, neither of shared memory. Nevertheless it succeeds in achieving comparable performance with existing DG implementation for linear problems [16] such as the one of Klöckner et al. [14].

Finally, the work of Modave, St-Cyr, and Warburton [17] in 2016 analyzes the performance of different implementation of a DG solver for acoustic and elastic models. The authors provide three implementations: one with the *one-thread-per-element* strategy used by Fuhry, Giuliani, and Krivodonova, one with the *one-thread-per-DoF* strategy followed by Klöckner et al. and the third one based on the use of Single-precision GEneral-sized Matrix Multiply (SGEMM) routine of the NVIDIA library cuBLAS. The results show that the best GPU implementation depends on the polynomial degree. The computational results show a strong performance of the one-thread-per-DoF strategy until fifth polynomial degree, while the strategy using SGEMM performs better for higher polynomial degrees.

Overall, considering previous work regarding DG implementations, we note how the use of sophisticated techniques for alignment management is not particularly necessary to achieve good performance. Moreover, the use of shared memory, in addition to increasing code complexity, also clashes with the increasing size of the simulation. This therefore justifies the introduction of an implementation with a simple memory layout that relies primarily on global memory access caching to achieve good performance.

3.2 The Gmsh DG solver

Gmsh DG is a Discontinuous Galerkin solver for conservation laws. In this section we will present the architecture of the solver, going gradually down into the details of the implementations.

The architecture Gmsh DG is a layered architecture in which the base layer is occupied by Gmsh. A view of the architecture is shown in Figure 3.1. Gmsh [18] is the 3D finite element mesh generator used in order to discretize the work domain, once installed on a system, it provides an API in many programming languages to use its functionalities. The middle layer is an abstraction layer over the Gmsh API called `libgmshdg`. This allows decoupling from the Gmsh data structures and allows to have a coherent collection of structures for the upper layer to work on. The upper layer is the layer where solvers are implemented, they include the data structures and algorithms to make the DG method work on a certain problem. We can see the solvers as modules that use the data structures of the `libgmshdg` to do their computations. In turn, these modules are composed by some common components and some deployment specific component, for example the algorithms to run on CPU and on GPU. At the moment, the only solver module implemented is related to Maxwell's equations.

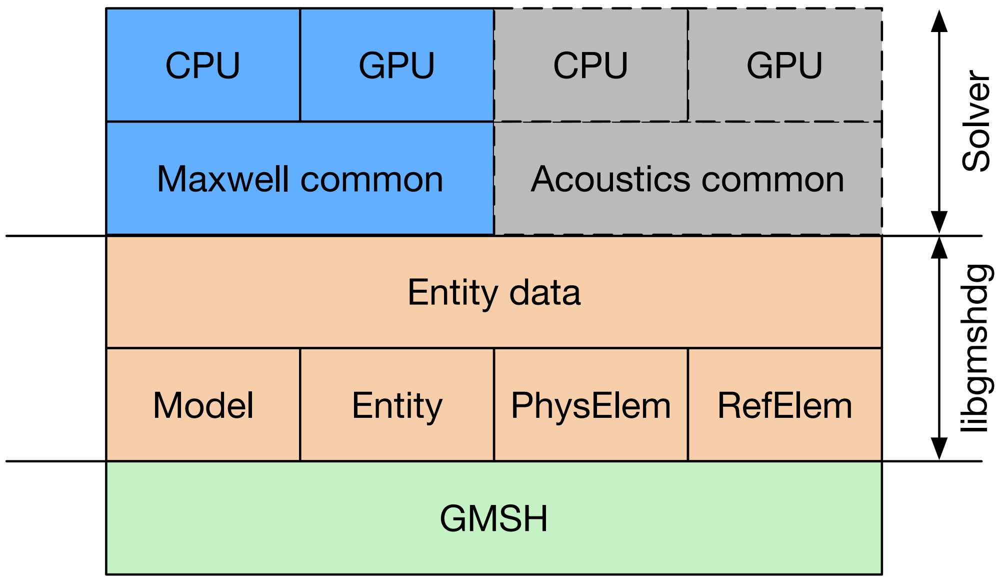


Figure 3.1: The three-layer architecture of Gmsh DG.

Before moving into the in-depth discussion of the layers of the architecture, we will talk about some general details of the implementation. Gmsh DG is implemented based on the revisiting of the work of Klöckner et al. [14], to determine to which extent the improvements of recent hardware would allow an algorithmic simplification without compromising efficiency.

The code is written in C++17, uses `cmake` as a build tool and targets distributed memory HPC systems. The deployment can be performed for CPU or GPU architectures, in particular the solver supports MPI parallelization on CPU and, carried out with this work, GPU. The MPI and/or GPU solver options must be chosen at compile time by setting the appropriate `cmake` compilation flags.

In our implementation GPU kernels are written using the CUDA C++ language extension and CUDA Runtime. The code is portable to the AMD ROCm platform by using the *Hipify* [19] tool that comes with the ROCm environment. The build environment is already set up to compile the code using Hipify by setting up the build configuration. The use of texture memory can be turned off or on at compile time for both AMD and CUDA environments. This is useful because HIP does not support textures.

Gmsh DG I/O To set up the simulation, the solver takes a *Lua* configuration script. Lua is lightweight, has very few dependencies, and it is easy to integrate into the application. The Lua configuration has a general part and a problem-specific part, the latter comprehends only the Maxwell solver module configuration for now and will be expanded as soon as other modules will be developed. The Lua configuration file also allows for the customization of the post-processing.

The output of the simulation is produced by the process with rank zero and is stored in SILO format [20], a standard regarding meshes and data upon those meshes. The Lua configuration script allows the user to choose how often in simulation timesteps a SILO output file is saved. Figure 3.2 shows Gmsh DG (Solver in the figure), its inputs and outputs.

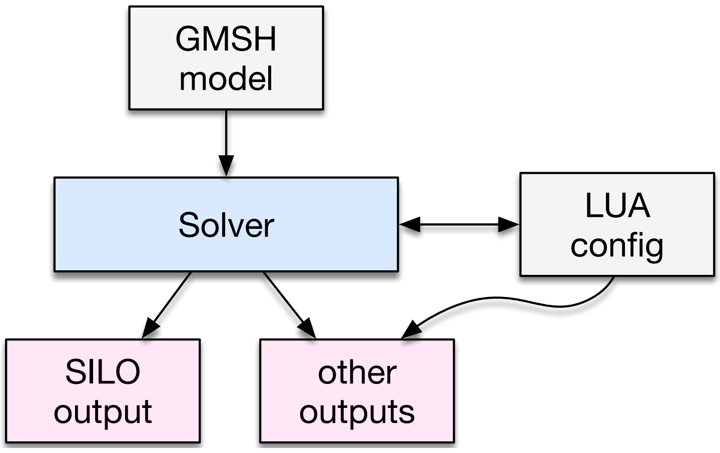


Figure 3.2: Workflow of Gmsh DG, showing inputs and outputs.

3.2.1 The `libgmshdg` layer

Since Gmsh DG is built on top of Gmsh and this application layer is built as an abstraction over it, we will illustrate some concepts regarding meshes in Gmsh.

A 3D model can be defined using its Boundary Representation: a volume is bounded by a set of surfaces, a surface is bounded by a series of curves, and a curve is bounded by two end points. [18]. Model entities are thus defined in four kinds: vertices, edges, faces and volumes. The model can be seen from a topological point of view, dealing with adjacency in the between model entities, and by a geometrical point of view, dealing with the shape (or geometry) of each model entity.

A *finite element mesh* of a model in Gmsh is a tessellation of its geometry by geometrical elements of various shapes [18]. In our solver we only work with meshes built with

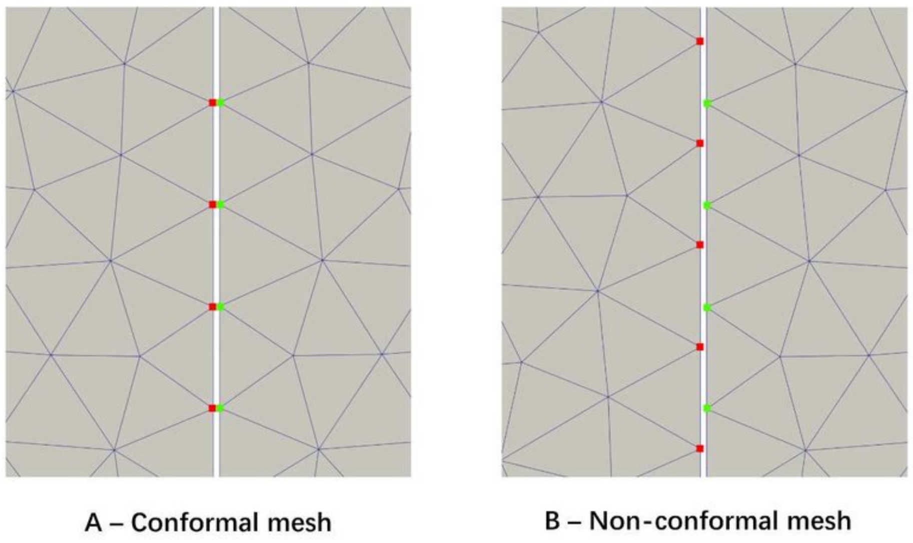


Figure 3.3: Example of 2D meshes, (A) shows a conformal mesh, while (B) shows a non-conformal mesh: in the second nodes intersect with faces and viceversa.

tetrahedral elements. Gmsh builds *conformal* meshes, meaning that elements are arranged such that if two of them intersect, they do so only along a face, edge or a node, an example of conformal and non-conformal meshes can be seen in Figure 3.3. An element is defined by an ordered list of its nodes and are stored in the model entity they discretize.

The first step of the implementation is to read the mesh. If a `.geo` script is provided, Gmsh DG will generate the mesh on the fly using the Gmsh API, otherwise if a mesh is provided, Gmsh DG will load that mesh in memory. The *model* component in Figure 3.1 contains the *entity* objects and the information pertaining the whole structure of the model, such as the interfaces and boundary surfaces descriptors. To accomplish parallelization at a process level the mesh is partitioned, and then one partition assigned to each process, thus the partition is the computational unit of an MPI process. The only exception to this is the process with rank zero, which has information about the whole Gmsh mesh for post-processing purposes.

In Gmsh DG the word entity has a different meaning than it does in Gmsh. Here entities simply represent a part of the domain. From a computational point of view, entities are collection of elements, distinguishing between the tetrahedra and their triangular faces.

In Subsection 1.3.4, when talking about the computational properties of the DG method, we mentioned that basing our computation on the actual elements of the mesh is not a feasible technique because it would require us to store a mass matrix, a stiffness matrix and a lifting matrix for each element. Thus, in order to reduce computational costs, we introduce a reference element and therefore a single mass, stiffness and lifting matrix can be used, allowing to pre-compute these matrices, and obtain the physical matrices by rescaling, as seen in Chapter 1.

These pre-computed data as well as the data needed for rescaling are stored into the `entity_data` object, which has an implementation for CPU and one for GPU, because of the different memory layouts of the deployment platforms.

3.2.2 The Maxwell solver module

As shown in Figure 3.1, the solver module contains some common utilities, and then specific code for the CPU and GPU version. The common part contains functions for the initialization and post-processing part and to evaluate boundary and interface sources.

Both the CPU and the GPU code were developed based on the flow graph of computation showed in Figure 3.4 that we recall from Chapter 1. The solver implements three methods for time integration: Euler, 4th order Runge-Kutta method, and leapfrog.

In order to carry out the simulation, the Maxwell solver module has a main data structure that is the `solver_state`. Because of the differences between CPU and GPU, two solver states are implemented, and the correct one is chosen based on the Lua configuration script. The solver state contains all the structures needed for storing the degrees of freedom during the computation, the values relative to the time integration and to material parameters.

Data layout

The organization of the degrees of freedom is particularly important because it affects how computation is performed and the possible attainable performance. We can distinguish two layouts, depending on whether we are considering volumes DoFs or surface DoFs. Regarding the volumes, the values relative to the DoFs are stored in a 1-dimensional vector and are organized by entity first, then within an entity by *orientation* (orientations are an implementation detail of H1-hierarchical basis functions; for more details we refer the reader to [21]) and within an orientation by element. The structure for the surface DoFs is very similar to the one of volumes, in fact the organization is the same until the element level, where they are further organized by face. In order to access the values within an entity or within an orientation, the `entity_data` object stores the offset relative to the entity and the number of elements that each orientation has, using this information it is possible to index any degree of freedom.

At each timestep, the computation steps shown in Figure 3.4 are performed. The computation made in each step will be presented below.

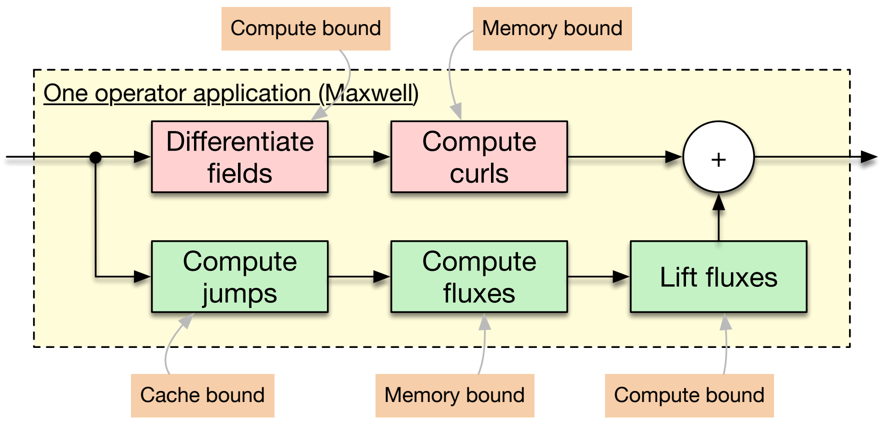


Figure 3.4: Computational steps required in order to do one application of the Discontinuous Galerkin operator.

Volumetric path

The volumetric path is depicted in red in Figure 3.4, and it is relative to the volume integration terms of (1.30) and (1.31) where we first compute the spatial partial derivatives of the six fields, and then curls are computed by subtracting the values of the derivatives.

The differentiation step consists in computing the partial derivatives of the electric and magnetic fields with respect to the coordinates. In order to find the differentiated value of the DoFs of one element, three matrix-vector multiplication between the differentiation matrices (one for each direction) and the sub-vector of DoFs relative to the element must be computed. This will result in three vectors, each of these vectors will then be weighted using the values of the Jacobian's row in order to rescale to the physical element. Finally, by summing the weighted vectors together, the differentiated degrees of freedom are computed as shown in (1.28). The computation of curls is then very simple, in fact it is a simple subtraction among the derivatives of the fields that were computed in the previous step. Here, we also apply the volumetric field sources to the electric field.

Flux path

The flux path refers instead to the surface integration term in the right-hand side of (1.30) and (1.31). In this path the field data uses the surface DoFs layout, during the lifting phase, the values are reported back to the volume layout, in particular, they are lifted and added to the output of the volumetric path.

The first step for this phase is computing the inter-element jumps. During the initialization phase of the solver, a map of neighboring nodes is built, thus accessing the map allows to get the index of the neighboring DoF in order to compute the jump. Subsequently, the values obtained by the jumps are used for computing the upwind fluxes, this step we also apply interface and boundary sources, modifying the flux term. The jump and fluxes definitions can be found in Subsection 1.3.4. Finally, in the lifting phase the lifting matrix is multiplied against the sub-vector of DoFs relative to one element in order to lift from the surface to the volume layout. The result is finally added to the result of the volumetric phase.

3.2.3 Computational bounds

We now start to present the critical points and bottlenecks of the code: Figure 3.4 is used as a reference for the computational steps.

Fields differentiation During the differentiation of fields, because both the fields and their derivatives are stored as a one-dimensional array, we have an optimal locality both on CPU and on GPU. Moreover, in the GPU code, the differentiation matrix is stored in the texture memory while Jacobians are fetched directly from global memory.

The current code computes all the fields partial derivatives, but actually we do not need all of them. Because we want to compute the curl, as we recall from (1.1), we only need six partial derivatives instead of all nine. Moreover, the kernel used to compute the field derivative computes only one of the field derivatives coordinates at a time, which means that the kernel will be called six times: three for computing the partial derivatives of **E** and three for the partial derivatives of **H**. As we will see in Chapter 4, we can actually "one-shot" the derivatives' computation for each field, reducing memory accesses. We observed that the actual kernel that computes one field derivative is memory bound, but also that performance are close to the peak performance, thus the new optimization will make the kernel become compute-bound, as Chapter 5 will show.

Curls computation As discussed before, the computation of curls is a simple step in which the degrees of freedom obtained during the differentiation are subtracted among them as (1.30) and (1.31) show. The current implementation separates this step from the previous one, which is suboptimal because the values can be subtracted directly when they are computed, avoiding additional memory accesses. Because the operation has a very good locality (it is indeed just a vector subtraction) then the major bound here is the memory bandwidth.

Jumps computation As mentioned in Subsection 3.2.2, during the computation of jumps we consider two neighboring elements. Even if topologically these two elements are neighbors, it does not mean that the values accessed in memory are close. They can actually be very scattered in memory: an example can be seen in Figure 3.5. This problem led Klöckner et al. in their work to come up with a reordering algorithm in order to improve locality.

Our implementation does not implement this reordering algorithm, thus one could consider this phase a bottleneck of the application. However, we have observed that on recent hardware, both on CPU and on GPU, the improvement to memory hierarchy made this non-locality not problematic. In fact, here we reach a good bandwidth utilization, especially at higher polynomial order on GPU. Further optimization could include a more extensive use of caches during this phase, but as the analysis will point out in Chapter 5, the computation of jumps is not one of the most time-consuming tasks, thus its optimization can be delayed.

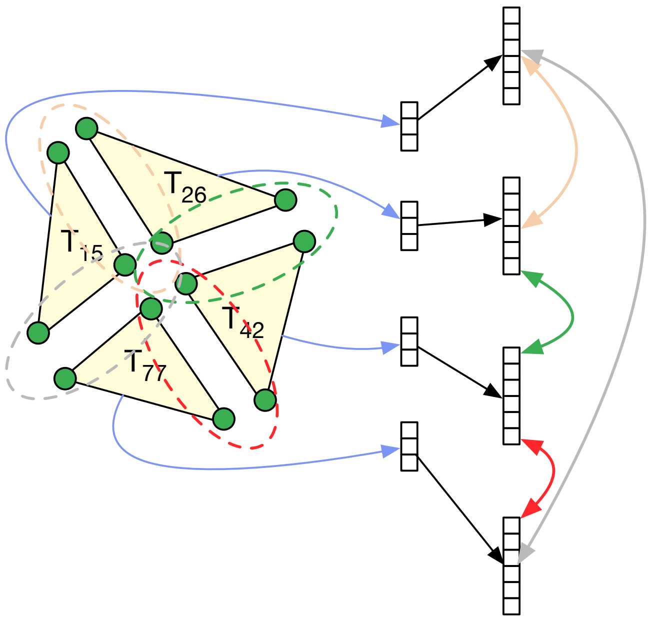


Figure 3.5: The computation of inter-element jumps can involve elements that are very scattered in memory, for example T_{15} and T_{77}.

Fluxes computation We are here computing the upwind fluxes from Section 1.4, it consists of a series of vector operations with perfect locality, so the only bound is the available memory bandwidth.

Fluxes lifting The lifting is a repeated matrix-vector multiplication between the lifting matrix and the fluxes in order to obtain the relative degrees of freedom of the element. This step is compute bound, but its performance could be improved by maximizing data reuse. In fact the matrix-vector multiplication could be transformed in a matrix multiplication, allowing the use of optimization techniques such as tiling to enhance locality. Unfortunately we will see that some difficulties arise because of the actual memory layout data structures.

On GPU, we make use of the texture memory to store the lifting matrix.

Sources As we previously told, volumetric sources are applied during the step of the computation of curls and interface and boundary sources are applied during the computation of fluxes. To evaluate the sources' contributions when running on GPU, we exploit the asynchronous CPU computation and compute the sources for time step $n+1$ on CPU while the GPU is computing time step n. This is a crucial step for the implementation because it exploits the CPU that otherwise would be waiting for the GPU computation to finish. The sources contributions are then uploaded to the GPU with an asynchronous copy.

3.3 Coarse- and fine-grained parallelization

Computing the degrees of freedom of a large mesh, based on a real test case, is not a fast task. For this reason, we need to exploit parallelization to improve the performance of the solver so that we can tackle more complex problems that would take *days* running on a single core.

Looking at the problem from afar, it all comes to the number of elements (tetrahedra) that are involved in the computation. So if we want to subdivide this work among many processes, which can be on the same machine or distributed on multiple nodes, the first idea is to assign each process a part of the elements and let it compute the output of the simulation over those elements.

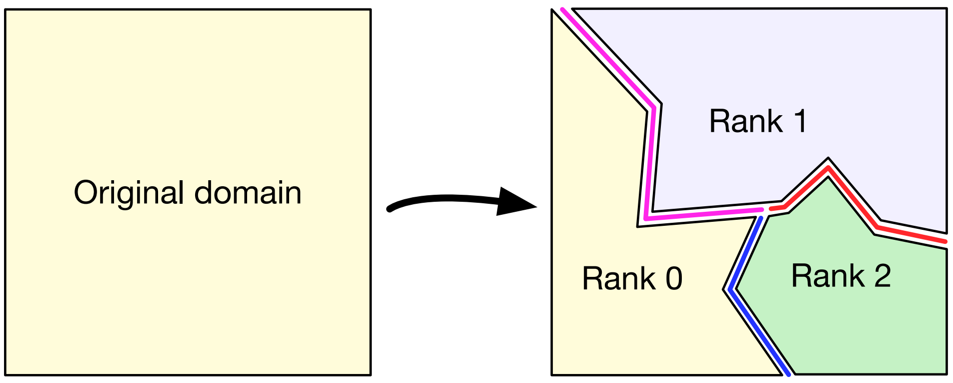


Figure 3.6: Partitioning and assignation of the mesh chunks to the processes.

Gmsh allows to partition the mesh into smaller chunks using METIS [22] [23]. The obtained partitions will have, roughly, the same number of elements so that the computation is balanced among the processes. This kind of coarse-grained parallelization is a perfect fit for the multicore model using MPI: if the mesh is big enough, every process will have a good amount of work to do, but neither too much, nor too few. Unfortunately, the implementation of the DG method is not embarrassingly parallel, which means there is the need of communication between processes. In fact, for each partition, the elements on the partition boundary need data from other elements in another partition in order to compute the jumps. The partition of the domain can be seen in Figure 3.6, which shows the *interfaces* between the subdomains where processes need to exchange data.

The GPU computation, though, does not fit well into this parallelization. The architecture of GPUs is built to have thousands of simple and independent threads working “at the same time”, this means we need a much more fine-grained parallelization than partitioning the original domain. This kind of parallelization can be found at a degree of freedom level: each computation phase has some output results, i.e. the curls, the jumps, the fluxes and the lifted fluxes. All these “output degrees of freedom” are independent of each other because each of them is computable in parallel as they resemble a matrix-vector product. Thus, the output DoFs can be used as the output value that a thread has to compute in the GPU model.

Most simulations have to compute hundreds of millions of degrees of freedom per iteration: this means that a very high number of threads will be generated for the GPU. This is not a problem, in fact that is exactly what we want: to maximize device occupancy and latency hiding, especially in the case of global memory accesses, which are the majority

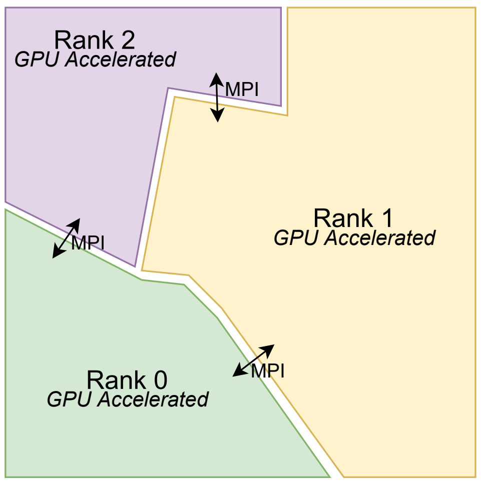


Figure 3.7: Domain partition and distribution over GPU-accelerated processes.

of memory traffic. The amount of threads also goes well with the automatic scalability of the GPU model, shown in Section 2.1.2, so that newer and more performant devices will automatically unlock better performance.

The next step is to merge these two kinds of parallelizations so that we can exploit the advantages of both. The idea is therefore to split the domain in partitions, assigning a partition to each process and then each one of these processes will solve the simulation by using a GPU accelerator, as whown in Figure 3.7. This means using multiple GPUs to compute the results and that data must be exchanged from GPU to GPU. Fortunately, thanks to recent middleware advancements, it is possible to exchange data between GPUs by simply using the usual MPI communication primitives, because the underlying implementation will deal with moving data efficiently [24][25].

Chapter 4

Proposed optimizations

This chapter will present some of the optimizations that have been implemented or proposed to improve the performance of the code. Along with these we will present some post-optimization performance analysis showing how the overall code is behaving. To obtain the results, we implemented a basic profiling infrastructure, toggleable at compilation time, that can be used to track metrics over a time period, like the number of floating-point operations or memory traffic.

4.1 Curls optimization

As mentioned before, this optimization has the main purpose of removing unnecessary computations and condensing memory accesses.

To better explain the reason behind the optimization, we remind that the curl of a vector field **F** in 3-dimensional Cartesian coordinates can be written as

$$\nabla \times \mathbf{F} = \begin{vmatrix} \hat{\boldsymbol{\imath}} & \hat{\boldsymbol{\jmath}} & \hat{\boldsymbol{k}} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ F_x & F_y & F_z \end{vmatrix},$$

which leads to the vector:

$$\nabla \times \mathbf{F} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z}\right)\hat{\boldsymbol{\imath}} + \left(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x}\right)\hat{\boldsymbol{\jmath}} + \left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y}\right)\hat{\boldsymbol{k}} = \begin{bmatrix} \frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \\ \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \\ \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \end{bmatrix}.$$

The original code would first compute all the partial derivatives of the fields **E** and **H**, even the ones not needed for the curl computation, then it would compute the subtraction to obtain the curl.

For example the kernel would take the component E_x of **E** of one mesh entity and compute $\frac{\partial E_x}{\partial x}$, $\frac{\partial E_x}{\partial y}$ and $\frac{\partial E_x}{\partial z}$, for each degree of freedom, whereas the first partial derivative is not needed. This call to the kernel then needs to be repeated for the other components of **E**, and of course for all the components of **H** for a grand total of six kernel calls.

The new kernel fuses the two old kernels, taking all the fields components as input and only computes the needed partial derivatives and performs the subtractions to obtain the curl of the field. The number of kernel calls is then just two: one for computing $\nabla \times \mathbf{E}$ and one for $\nabla \times \mathbf{H}$. It also applies boundary sources to the electric field.

We measured the execution time of the new version and compared it with the old version. To understand the reported data, it must be clear that the measurements were made only on the kernel itself. The differentiation kernel only computes the partial derivatives of *one* field coordinate, while the new kernel computes the entire curl of the field. Thus, the new kernel does roughly three times the work that the differentiation kernel does. We also reported the expected *GFlops/s*, based on the number of floating-point operations of both kernels. The "Actual Speedup" column in Table 4.1 comes from the fact that in the application we need three calls to the differentiation code to obtain the curl of the field.

Approx. Order	**Diff. Time**	**Diff. GFlops/s**	**Curl Time**	**Curl GFlops/s**	**Speedup**	**Actual Speedup** ($Speedup \times 3$)
1	6.92e-04	16.820	1.50e-03	17.171	0.46	1.383
2	3.34e-03	21.803	7.57e-03	20.860	0.44	1.322
3	5.69e-03	51.159	1.20e-02	52.294	0.47	1.422
4	1.88e-02	47.409	7.15e-02	26.8192	0.26	0.789
5	7.43e-02	30.699	1.49e-01	32.8295	0.50	1.494

Table 4.1: Elapsed time, *GFlops/s* and speedup of the old differentiation kernel and the new curls kernel on a test sample of 34632 elements.

The obtained speedup and the number of floating-point operations imply that the improvement was mainly due to the reduction of memory accesses. However, the kernel analysis in Chapter 5 will point out that also the performance in terms of *GFlops/s* increases. This discrepancy is due to the pen-and-paper computation of the number of operations, while later we use specialized profiling tools to count the operations.

To have feedback on the overall performance of the solver after this change, we also measured the speedup based on the degrees of freedom per second that we obtained on the $[0,1]^3$ resonant cavity problem model used as a validation example. Data are shown in Table 4.2.

The GPU solver was modified in the same fashion as the CPU code. The implementation has been tested on the same resonant cavity example to measure the overall performance of the solver. The measurements and the speedups are reported in Table 4.3 and show a more satisfying improvement, especially for higher approximation order.

Approx. Order	**Mesh Size**	**DoF/s Original**	**DoF/s New**	**Speedup**
1	0.05	7.91e+6	9.08e+6	1.148
2	0.05	6.58e+6	7.56e+6	1.150
3	0.05	7.27e+6	8.34e+6	1.148
1	0.04	7.21e+6	8.26e+6	1.147
2	0.04	6.12e+6	7.12e+6	1.163
3	0.04	7.04e+6	8.27e+6	1.176
1	0.03	5.56e+6	6.92e+6	1.244
2	0.03	5.17e+6	5.95e+6	1.149
3	0.03	6.55e+6	7.42e+6	1.133

Table 4.2: Degrees of freedom per second and speedups of the computation of curls on the resonant cavity validation example using RK4 time integration on CPU.

Approx. Order	**Mesh Size**	**DoF/s Original**	**DoF/s New**	**Speedup**
1	0.05	2.75e+8	2.76e+8	1.004
2	0.05	3.84e+8	4.72e+8	1.228
3	0.05	3.64e+8	4.70e+8	1.292
4	0.05	3.05e+8	4.24e+8	1.391
5	0.05	2.55e+8	3.75e+8	1.471
1	0.04	3.34e+8	3.32e+8	0.996
2	0.04	3.94e+8	4.71e+8	1.196
3	0.04	3.63e+8	4.75e+8	1.309
4	0.04	3.06e+8	4.21e+8	1.377
5	0.04	2.57e+8	3.77e+8	1.464
1	0.03	3.67e+8	3.78e+8	1.030
2	0.03	3.99e+8	4.79e+8	1.201
3	0.03	3.64e+8	4.62e+8	1.270
4	0.03	3.08e+8	4.26e+8	1.384
5	0.03	2.58e+8	3.78e+8	1.465

Table 4.3: Degrees of freedom per second and speedups of the computation of curls on the resonant cavity validation example using RK4 time integration on GPU.

4.2 Proposal: differentiation matrix layout

A change to the differentiation matrix memory layout was also considered for optimizing memory accesses. During differentiation, in order to obtain one differentiated DoF of an element we need to perform the operation explained in Subsection 3.2.2 and in (1.28).

However, in the code we do not perform the operations in the same order as explained there. The three reference differentiation matrices are stored as a single matrix in memory, as can be seen in Figure 4.1a. In order to exploit locality (and memory coalescing on GPU) we compute one differentiated DoF at a time without computing the matrix multiplications. Let us say that the differentiated DoF is the i-th of the element, then the computation is reorganized as follows:

1. Extract the vector $v = (v_0, v_1, v_2)$ from the i-th row of the differentiation matrix, where the value corresponds to the current *input* degree of freedom f. For example in Figure 4.1a if we consider the blue DoF, then we are fetching the blue values from the matrix.

2. Compute the multiplication $w = vf$ between v and the input degree of freedom f;

3. Scale back to the physical element by multiplying a Jacobian's row with w in order to get the derivative over one coordinate;

4. Accumulate the result and repeat for each input DoF of the element.

The accumulators value will be in turn used to compute the curl. In Figure 4.1a the corresponding involved values are shown.

Accessing the differentiation matrix to get the values of the vector v will of course cause some compulsory cache misses. In fact, the number of degrees of freedom in a tetrahedron is

$$N_3^k = \frac{(k+3)(k+2)(k+1)}{6},$$

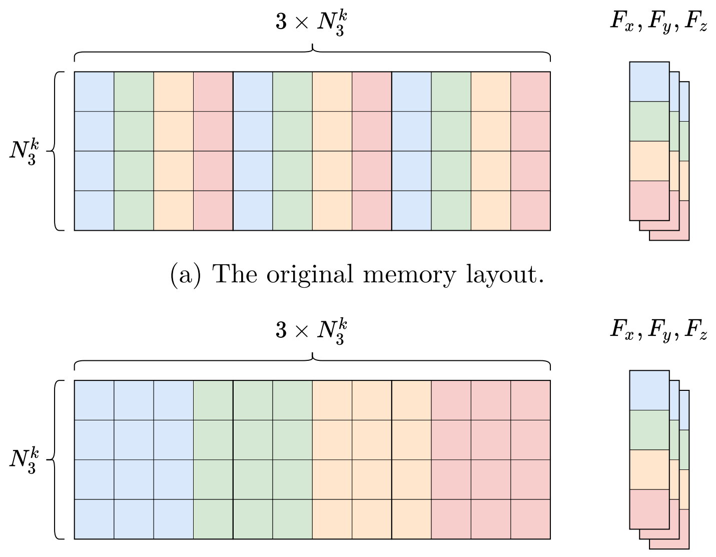
where k is the approximation order, and the values v_0, v_1, v_2 have stride equal to N_3^k. Then if we consider a 64-byte cache line, accessing the elements in this fashion would cause at least two cache miss, and a maximum of three when considering $k > 1$ (note that we work with double-precision values).

The proposed layout, shown in Figure 4.1b, would have optimized the cache misses for accessing values belonging to the same input degree of freedom, reducing them to one, compulsory, cache miss. The change in the layout was achieved by performing a permutation of the columns of the matrix.

It was observed, however, that the achieved performance were not as good as expected. That is probably due to the fact that even if the number of cache misses is almost the same, the frequency of the misses is different.

With the original layout, we have (for $k > 1$) three cache misses for every eight iterations over the input degrees of freedom.

The new layout, on the other hand, performs one cache miss every (roughly) three iterations over the input degrees of freedom, because a 64-byte cache line can contain only up to eight double-precision values. This seems to lead to worse performance than the original layout and therefore this change has been discarded.



(a) The original memory layout.

(b) The proposed new memory layout.

Figure 4.1: The memory layout of the differentiation matrices.

4.3 Proposal: fluxes lifting optimization

The current phase of fluxes lifting consists of a matrix-vector multiplication between the lifting matrix and the fluxes of the element face, as explained in Subsection 3.2.2. This has the advantage to be a very easy operation, but on the other hand it does not take advantage of data locality. The lifting matrix is in fact the same for each element having the same orientation, for this reason locality could be improved by using the cached row of the lifting matrix to compute the product with the corresponding column of the fluxes of the other elements. This will actually transform the computation from a series of matrix-vector multiplication to a matrix multiplication that can be in turn optimized by using techniques to exploit *locality*.

Doing this may raise some difficulties when trying to implement this approach in the existing solver. The main problem is due to fact that the Gmsh model has a number of orientations, and each of these has a different number of elements, therefore on CPU for each orientation we can rearrange a part of the fluxes and lifting vector as a matrix to make a matrix multiplication.

This is non-trivial on GPU because the approach is to have one thread for each entry of the matrix of lifted fluxes. Hence the usual approach of GPU matrix multiplication with tiling does not suit well with the multiple orientations. In fact, because of this, there are many matrix multiplications with different matrices size, and this forces us to launch a kernel for each orientation, leading to a higher overhead and to a very low occupancy of the GPU. A potential way to make this approach work is to use *concurrent kernel execution* so that these multiplications can be run at the same time, on different *streams*, obtaining the same occupancy of the current approach. This solution has not been implemented though because of the invasiveness of the change in the current codebase.

4.4 One-shot lifting and jumps

By following the same approach as for the curl optimization, lifting and jumps kernels also have been optimized to reduce memory accesses.

In fact, the current lifting kernel was made so that it worked only on one field coordinate, thus requiring six calls to the kernel in order to complete the lifting. The same can be said for the kernel that computes the jumps, also requiring three kernel calls for $\mathbf{E}$ and three for $\mathbf{H}$.

Kernel fusion is an optimization technique employed to make generic kernels that are called many times into a single kernel, in order for the kernels to share data and thus reduce memory accesses and possibly enable compiler optimizations. This optimization is especially useful for memory-bound kernels, as it can improve the use of caches and thus increase arithmetic intensity, in turn unlocking better performance.

This optimization has been partially implemented on CPU, where we use Eigen [26] to manage matrices and vectors, while on GPU both optimizations have been applied.

Chapter 5

Gmsh DG performance analysis

This chapter will present the performance of our solver on two levels: kernel and application level. For the kernel level analysis each of the computation steps has been analyzed in terms of performance over the different deployment platforms (except for multi-GPU, that will be treated in the next chapter). The application level analysis has been carried out in order to assess the overall application performance on real workloads. Then, we will show and discuss the results obtained by our measurements, while also having a focused look on the deployment hardware details, which can truly change how the applications scales if not taken into account. A profiling infrastructure has been implemented in the solver codebase in order to provide insights before the analyses were carried out. The infrastructure was also used to gather some data on the kernel analysis as well as the scaling analysis.

5.1 Profiling infrastructure

The first step in order to provide a thorough performance analysis is, as a preliminary phase, to get an idea of the expected results. Usually, a pen-and-paper evaluation is done in this phase. For instance when looking at the kernel, one can count memory accesses and the number of operations in the code to have an idea of the performance and arithmetic intensity of a kernel. This kind of analysis is not very accurate and specialized tools can be used in order to extract these data from executions.

However, some kind of middle ground is possible and has been implemented during this thesis. In our work we often need to know how much time one part of the computation requires. This portion of the code can be composed of many instructions and kernel calls, thus the idea is to implement an infrastructure that could record the execution time of these phases. In addition to execution time, sometimes we also need the number of operations and/or the memory bandwidth of a certain kernel; these values are the result of evolution of some metric over a given amount of time. In this phase we are not interested in the exact metric value, so we can use a pen-and-paper estimate, while the time is given by the profiler infrastructure, and we can then compute the final value in $GFlops/s$ or GB/s.

The profiling infrastructure was implemented in order to record these estimated quantities, introducing three new classes and two subclasses represented in Figure 5.1 and that will be explained below. Profiling can be turned on or off through the `ENABLE_PROFILING` flag in the `cmake` configuration.

Time counter The first class is the `timecounter` class. Its responsibility is to record the time, thus it has two member functions: `tic` that records the start time, and `toc` that records the end time and returns the difference between the two. Since Gmsh DG has both a CPU and GPU implementation, two specialized `timecounter` have been implemented, each of them using the appropriate functions to record time. Both the specialization descend from an abstract base class containing only the interface of the member functions. In particular the GPU specialization uses the CUDA Runtime *events* to capture the elapsed time; thus in order to record this time, the CPU thread will be blocked until the recording finishes. It is therefore critical to avoid profiling on GPU while testing for real performance.

Profiler The `profiler` class stores the information regarding the time of a certain part of code. Since the solver is iterative, a certain part of code will be repeated more than once. Thus the profiler will store the last time recorded and the average time. The profiler is the actual object that will be used in the code to record time, using its two member functions `tic` and `toc`, that in turn make use of a `timecounter` object. The `profiler` takes care of storing the average time of execution of the said code part. Profilers can be used through two macros that will be replaced with the actual function call when profiling is active, and will be replaced with null operations when the profiling is turned off. In order to compute time-dependent metrics such as $GFlops/s$, profilers also provide the interface to specify which quantities need to be tracked and the interfaces to access their average and last value.

Profiler logger The `profiler_logger` class is used as a container to register and retrieve the profilers, indexed by their name. The profiler logger is attached to the `solver_state` structure presented in Subsection 3.2.2. When registering a profiler, the profiler logger will take care of creating the actual object and store it. The class also provides some utility functions to print the value of all the profilers and their attached metrics.

The usual workflow to use the infrastructure is the following:

1. Create a new `profiler` through the `profiler_logger` attached to the `solver_state`
2. Optionally, add some metric to the `profiler`
3. In the code, get the `profiler` from the `profiler_logger` and use it through the macros

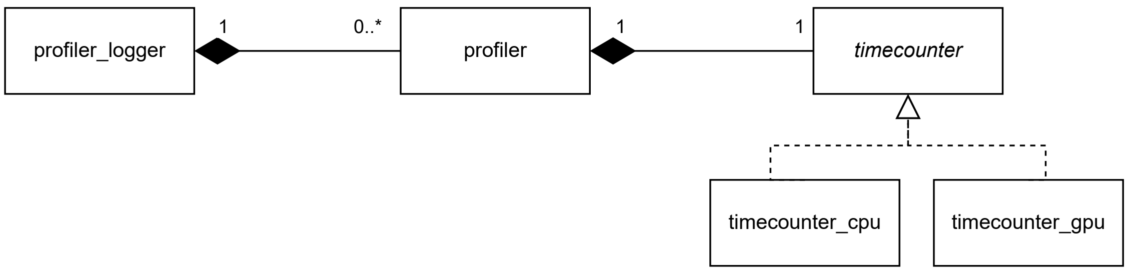


Figure 5.1: Class diagram of the class implemented for the profiling infrastructure.

5.2 Kernels analysis

Before analyzing the whole application performance, it is desirable to make some analysis at a fine granularity level to get a comprehensive idea of the performance of the solver. Moreover, this kind of analysis can give empirical data about the bottlenecks of the application and can provide useful insights on which could be the focus of next optimizations.

In order to make the analysis, we first need to gather some metrics from the kernels. These have been obtained by exploiting the profiling infrastructure when it comes to the execution times. Other specialized tools for each platform, i.e. Intel Advisor [27] for CPU and NVIDIA Nsight Compute [28] for GPU, were instead used for getting precise metrics about memory traffic and number of operations. The use of these tools is especially important when dealing with such values, as the evaluation of traffic or operations cannot be done accurately by a pen-and-paper approach. Note that even if the pen-and-paper approach can be inaccurate, it can still provide useful insights in a preliminary phase of the analysis, without the need of using tools and scripts to automate data gathering and organization.

5.2.1 Time comparison

This analysis has been carried out in order to understand which kernels take more time during the application execution and how this time evolves while increasing the approximation order. We targeted three of the possible execution modes: serial, multicore and GPU-accelerated; and both time integration methods: leapfrog and RK4.

The measurement of execution times has been made by using the solver profiling infrastructure. The results show the percentage of time that the solver takes in each of the computation and communication stages, over the time of one iteration. The mesh used for the serial and GPU measurements is the same, and is composed by 31543 elements, meaning that for each iteration 757032 degrees of freedom must be computed at approximation order one while around ten millions must be computed at approximation order five.

Serial analysis The serial solver has run on a platform with an Intel® Xeon® Gold 6126 CPU. It can be clearly seen in Figure 5.2 that with the increase in approximation order, the most time-consuming activity is the computation of curls, which is also one of the motivations that led to its optimization.

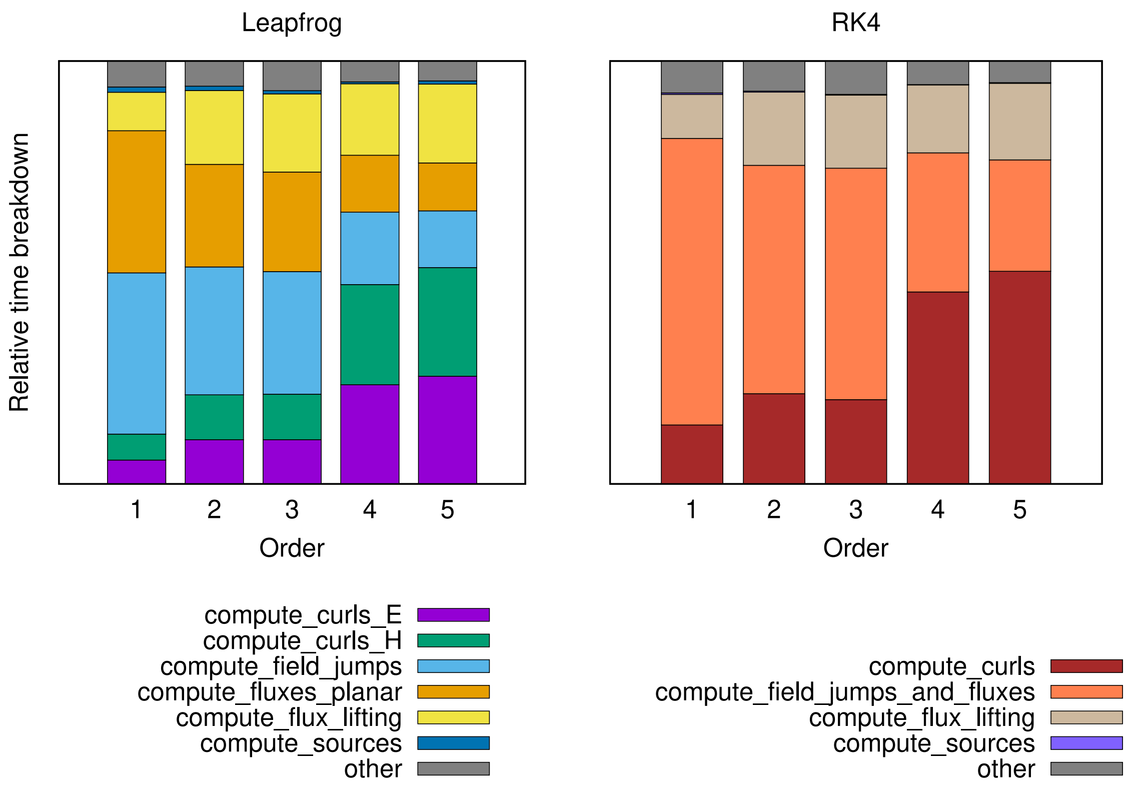


Figure 5.2: Serial solver kernels relative time comparison for leapfrog and RK4 methods over different approximation orders.

GPU analysis The single GPU results shown below have been taken on an NVIDIA TITAN V graphics processor which features the Volta microarchitecture and supports compute capability 7.0. For these results, we had to work in two stages: the first one in order to gather the execution time of the kernels, the second one in order to get the actual iteration time. In fact, because our profiling infrastructure is based on exploiting CUDA Runtime *events*, waiting for an event to complete would block the calling CPU thread making ineffective the asynchronous computation of sources, that would be serialized after the timestep computation.

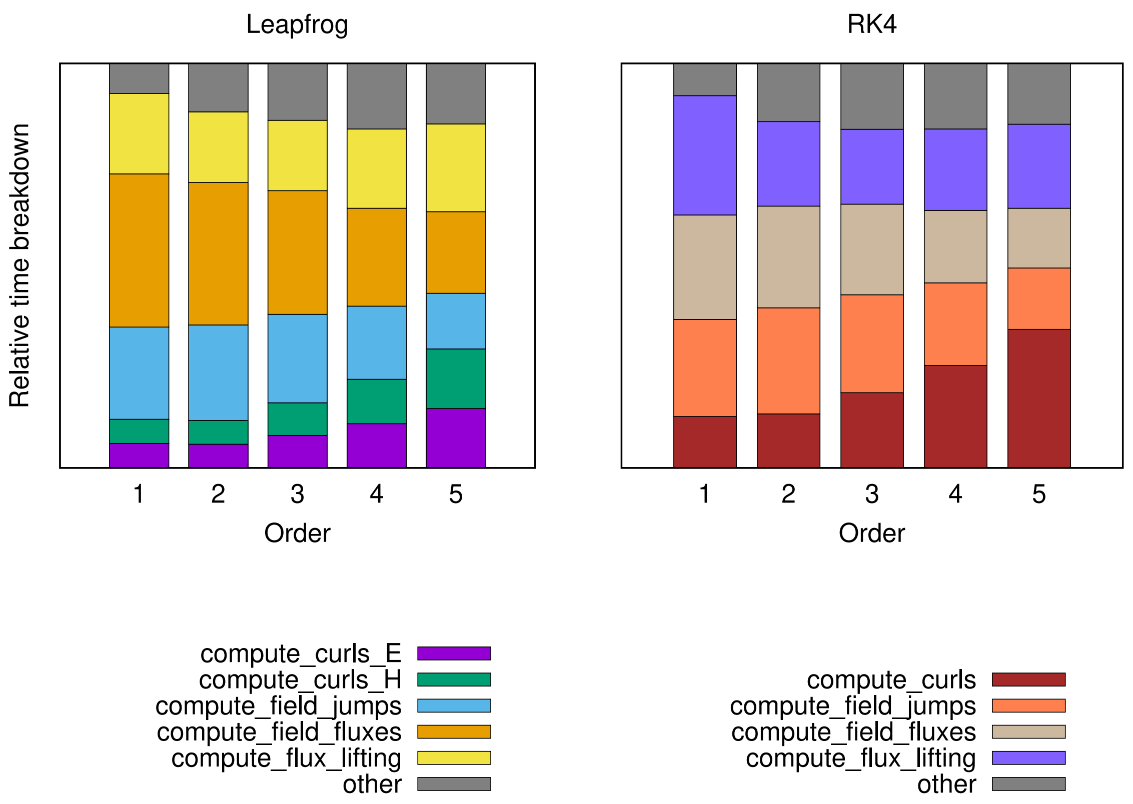


Figure 5.3: GPU solver kernels relative time comparison for leapfrog and RK4 methods over different approximation orders.

The relative time breakdown in Figure 5.3 only shows the kernels that run on GPU and the remaining time doing other tasks. The absolute time breakdown in Figure 5.4 shows both the absolute time of the kernels and the computation time of the sources that, in the GPU implementation, are computed asynchronously on CPU (cf. Subsection 3.2.3). As the figures show, on GPU the time required for computing jumps and fluxes decreases, but with a reduced rate when compared to the CPU execution.

Multicore analysis The multicore results have been obtained during the strong scaling analysis of the CPU solver, meaning that the amount of total work for the solver would remain constant during the whole analysis. These results also show the amount of time that the solver takes to exchange data between processes. The multicore measurements have been taken on eight reserved nodes on the CÉCI cluster NIC5, each node consisting of a two-socket AMD Epyc Rome 7542 CPU, with 64 total cores. The nodes are interconnected with a 100 Gbps HDR InfiniBand. More details on the testing platform and execution configuration will be given in the upcoming Section 5.3.

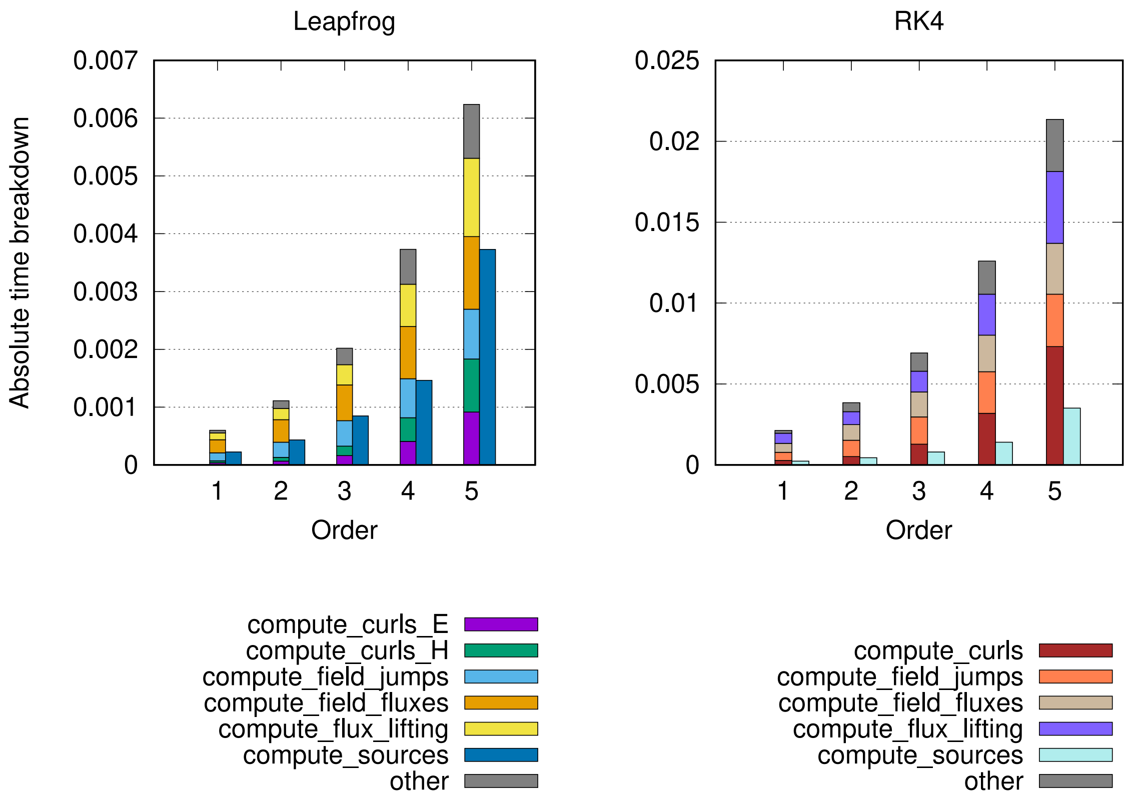


Figure 5.4: GPU solver kernels and CPU asynchronous kernels absolute time comparison for leapfrog and RK4 methods over different approximation orders.

The results in Figure 5.5 show a very high variability in communication time, especially when considering the communication among many nodes and processes. In order to explain this behavior we also tested an implementation of the solver using non-blocking communication between processes in order to reduce the receiver blocking wait time. We noticed that the time for communication was roughly the same that resulted during the previous tests, thus we would exclude blocking communication to be the cause of these anomalies. This phenomenon is still not very clear and will need some further investigation, because the pattern seems irregular and targets only some approximation orders. Furthermore we observed this issue when running other applications on the cluster, thus the problem might be due to some misconfiguration in the hardware or software facilities.

5.2.2 Roofline analysis

Roofline analysis [29] [30] is a great tool to have a visual and intuitive estimate of kernels optimization, comparing their performance to hardware limitations, providing a way to identify bottlenecks and drive code optimization.

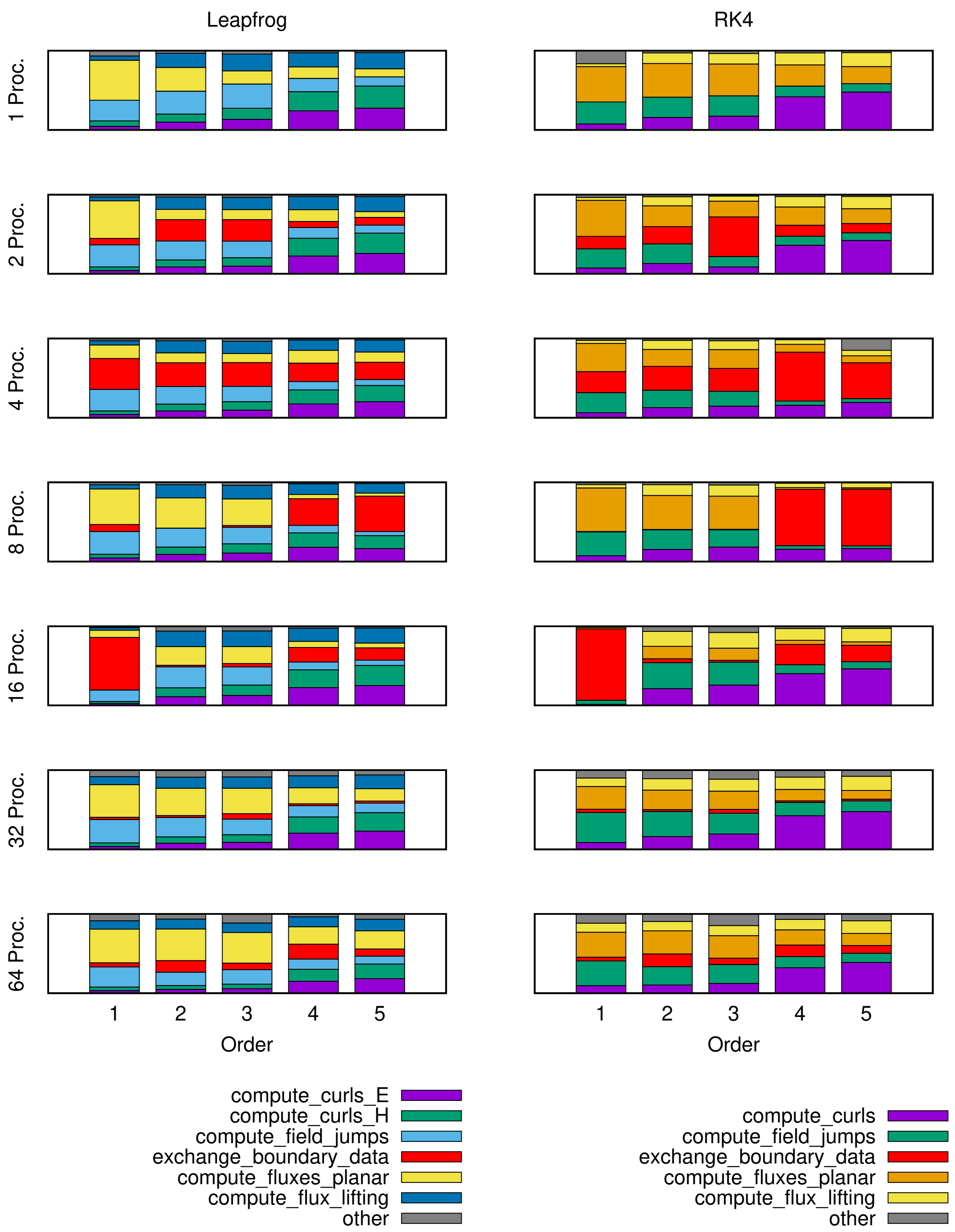


Figure 5.5: Kernels relative time comparison on multiple processes, single node, for leapfrog and RK4 methods over different approximation orders.

Each kernel has three associated metrics:

- the work W, that is the number of operations performed by the kernel; in our case it will be the number of double-precision floating-point operations;
- the number of bytes Q of memory traffic generated by executing the kernel; this quantity depends on the details of the platform, such as cache hierarchy;
- the arithmetic intensity
$$I = \frac{W}{Q},$$
that determines the number of operations per byte of memory traffic.

The roofline plot has on the x-axis the arithmetic intensity and on the y-axis the performance. A kernel is represented as a point in the plot, with a given arithmetic intensity and a performance value.

Attainable performance are bounded by hardware limitations: peak performance π, measured in *Flops/s* and peak bandwidth β, measured in *Byte/s*. This is based on the assumption that memory transfers and computation can overlap, so a kernel execution time T is limited by π or β

$$T = \max\left\{\frac{W}{\pi}, \frac{Q}{\beta}\right\} = W \cdot \max\left\{\frac{1}{\pi}, \frac{1}{\beta \cdot I}\right\}.$$

Then, maximum attainable performance P are given by

$$P = \frac{W}{T} = \frac{1}{\max\left\{\frac{1}{\pi}, \frac{1}{\beta \cdot I}\right\}} = \min\{\pi, \beta \cdot I\}.$$

This bound translates to the ceiling of the roofline plot, with a diagonal line, the function $\beta \cdot I$; and a horizontal one, π, where the intersection of the two bounds is at arithmetic intensity $I_m = \frac{\pi}{\beta}$. Kernels with $I < I_m$ are in the slanted part of the model and are memory bound, while kernels with $I \geq I_m$ are compute bound.

The original roofline model [29] relied on considering memory traffic between (theoretically optimal) caches and main memory, while the current approach is to consider the traffic between the core and the cache, allowing for the derivation of new bounds related to a given cache level, obtaining a cache-aware roofline model [31].

To build the roofline model, we need the ceilings of the models and the metrics of the kernels. To obtain the ceilings our approach involved the use of micro-benchmarks to obtain the empirical hardware limitations. The building of the roofs for the CPU model was made using the Intel Advisor tool, because the measurements were taken on an Intel CPU powered machine. The roofs for the GPU model were obtained using the Empirical Roofline

Tool (ERT) [32], which obtains the ceilings by running micro-benchmarks. Because the L1 cache roof was not obtainable using ERT, we based that roof on the results by a technical report about the NVIDIA Volta microarchitecture [33].

The metrics of the kernels were obtained using Intel Advisor for the CPU and using NVIDIA Nsight Compute for the GPU. The data were then extracted by both tools outputs and put in a plot along with the ceilings. Multiple dots on the plots are due to the fact that we made the roofline plots based on the leapfrog method, which has different kernels for the electric and magnetic field.

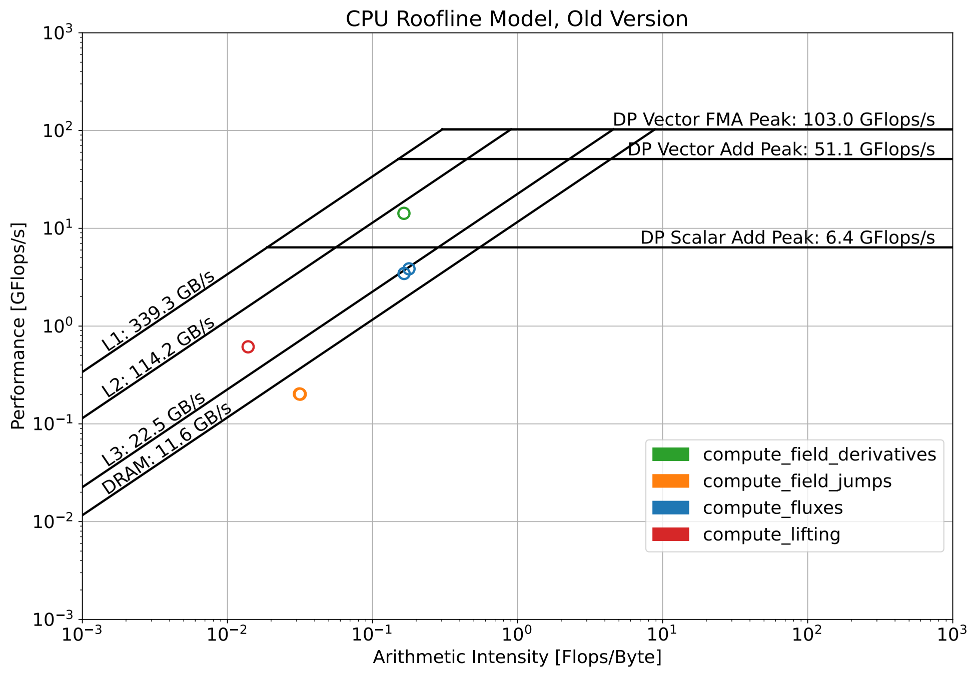


Figure 5.6: Pre-optimization roofline analysis of CPU kernels.

CPU Roofline We present the roofline results of the kernels, where we measured both the old and the new version, after the optimizations described in the previous chapter. The measurements were taken on an Intel® Xeon® Gold 6126 CPU, while the solver worked with the $[0,1]^3$ resonant cavity problem model with approximation order four.

Figure 5.6 shows kernels performance plotted on the roofline, and Figure 5.7 the performance after the optimization of the curls computation. The only kernel that changed its performance is the one relative to the field derivatives and the curls. We picked the performance relative to the most inner loop of the kernel, and as is clearly visible, the

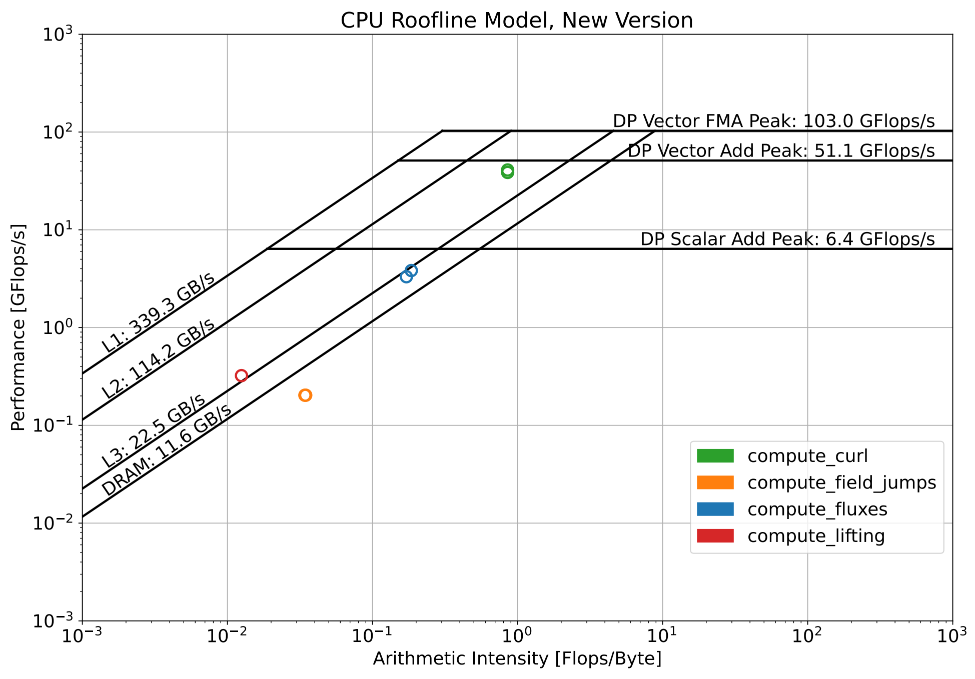


Figure 5.7: Post-optimization roofline analysis of CPU kernels.

arithmetic intensity of the kernels has changed. This is due to the fact that in the curls kernel we are computing the derivatives relative to a field all at the same time as explained in Section 4.1, thus memory traffic is increasing because we are getting the values of the degrees of freedom for each component, but the number of operations is increasing at a higher rate because we are essentially tripling the arithmetic operations while recycling data accesses to the differentiation matrix and Jacobians.

On the other hand, this analysis also proves the boundaries stated in Section 3.2.3, showing that the implementation of the kernels, despite its simplicity, makes a good use of memories, and increasing the use of caches would unlock even better performance.

GPU Roofline We show two GPU rooflines. The first, Figure 5.8, at approximation order one and the second, Figure 5.9, at approximation order five. The first thing to notice, as a difference compared to the CPU code, is that the lifting kernel exhibits better performance due to a different arithmetic intensity: the difference is probably caused by the fact that on CPU data is accessed in a non-optimal way using Eigen, since the number of operations is the same.

Moving from order one to five, the arithmetic intensity of the lifting and curls kernels

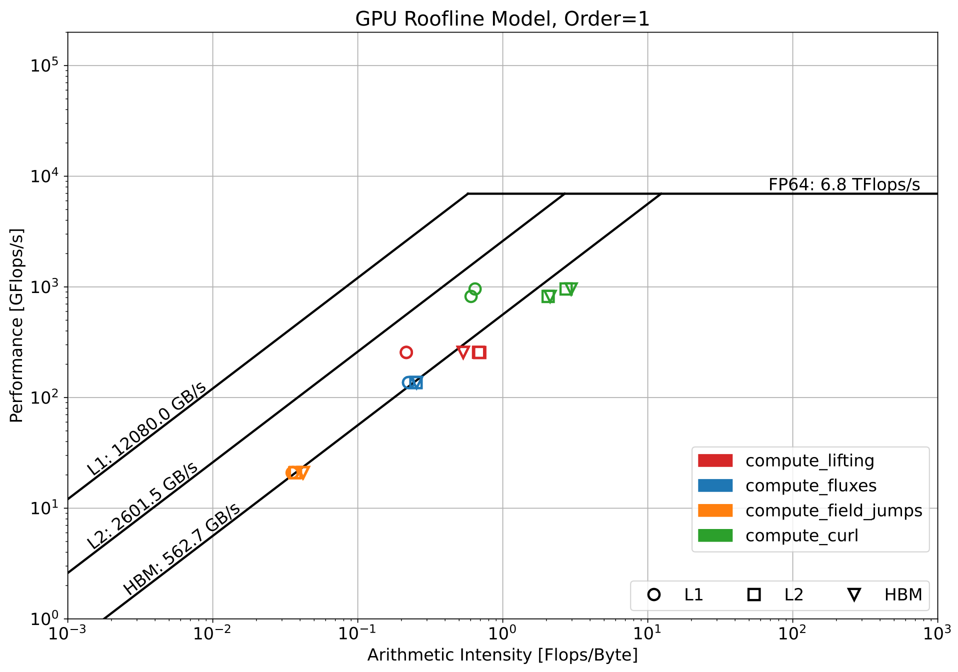


Figure 5.8: Roofline analysis of kernels on GPU, approximation order one.

increases. From Figure 5.8 we can see that curls and lifting kernels are memory bound at order one. When approximation order reaches five in Figure 5.9, the increased arithmetic intensity allows the two kernels to increase their performance, thus the curls kernel becomes compute bound while the lifting kernel is still memory bound but with increased performance.

In both CPU and GPU kernels, performance could be further improved by using caches more intensively. Some kernels like the computation of jumps are however not very well suited for caching due to the data layout. This could be resolved by using different strategies, like degrees of freedom reordering [14], but it is important to point out that even without such strategies they can reach full DRAM bandwidth. Moreover, as the time analysis highlights, the kernel that tends to take more time when increasing the approximation order is the computation of curls, which is already very close to the ceilings of the model.

5.3 Solver scaling

After showing how kernels perform and highlighting how the optimizations and execution time parameters can affect performance, it is time to show the scaling behavior of the

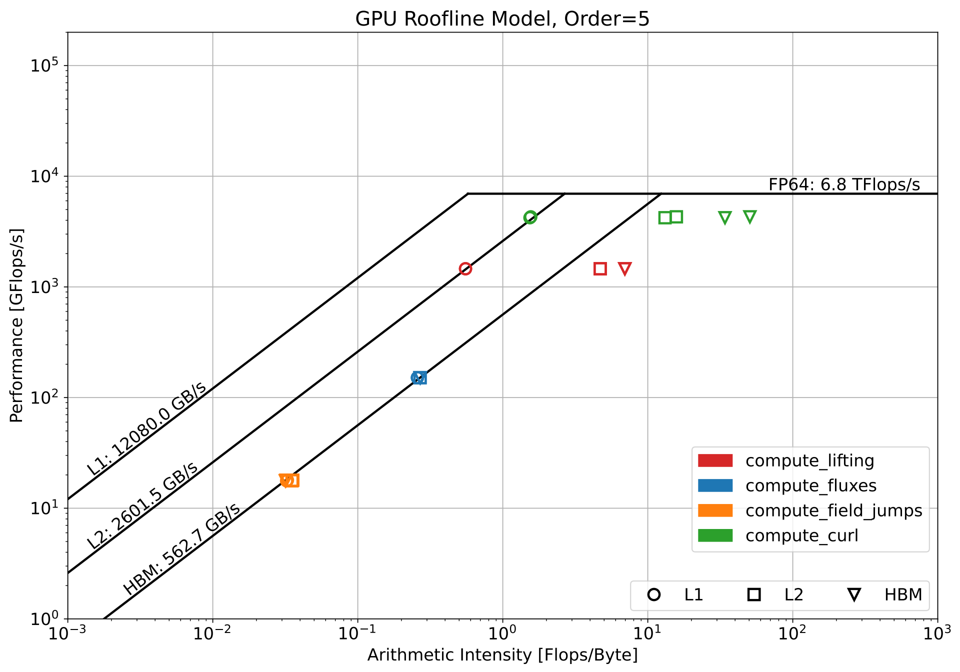


Figure 5.9: Roofline analysis of kernels on GPU, approximation order five.

solver. In this discussion we will heavily focus on the multicore scaling, comparing it with the GPU execution performance, while the next chapter will focus on the multi-GPU execution.

Scaling measures how the performance of a parallel application changes as the number of processes is increased, and whether it correctly exploits the hardware resources. There are two ways to measure scaling: *strong* and *weak* scaling. Both methods increase the number of processors N, but they differ in how they treat the problem size and are tied, respectively, to Amdahl's [34] and Gustafson's [35] laws.

For strong scaling, two metrics are used: the *speedup* and the *parallel efficiency*; both of them require the application execution time to be known. Let N be the number of processors, and $T(N)$ the execution time of the application with N processors. The speedup of the application with N processors is defined as

$$S(N) = \frac{T(1)}{T(N)},$$

and the parallel efficiency over N processors as

$$E(N) = \frac{S(N)}{N}.$$

Typically, the speedup is bounded by the number of processors, having $S(N) < N$. The parallel efficiency is therefore bounded by 1: $E(N) < 1$. It is however sometimes possible to have a *superlinear speedup* [36], meaning that the speedup will exceed N. This can happen due to a series of reasons: because the number of performed operations is smaller than the serial version, because of more caches for parallel execution or because the application runs in a heterogeneous environment.

Strong scaling considers a fixed problem size, and here Amdahl's law comes into play, which bounds the speedup of a program. In fact each program has a sequential portion s and a parallel part p, where $s + p = 1$, thus if $T(1)$ is the serial execution time, then the execution time on N processors can be written as

$$T(N) = sT(1) + p\frac{T(1)}{N},$$

and the speedup will be

$$S(N) = \frac{T(1)}{T(N)} = \frac{T(1)}{sT(1) + p\frac{T(1)}{N}} = \frac{1}{s + \frac{p}{N}}.$$

For $N \to \infty$ we have that $S(N) = 1/s$, meaning that the speedup is bounded by the sequential part of the program. The link with strong scaling is then clear: with strong scaling we fix the size of the problem and, by increasing the number of processors, we are dividing the amount of work; this in turn will make each process have less and less work, reducing the parallel part of the program and eventually leading to a degradation of performance due to the serial part of the program.

Strong scaling is therefore difficult to obtain, because in order to reach a good one, code must have been finely optimized to maximize the parallel execution of the code. Moreover, in our discussion we omitted that the execution time also involves some *overhead* due to communication or synchronization, that can harm parallel execution when the number of processors increases.

Weak scaling instead considers the problem size to increase as the number of processors increases. This is not a so bold assumption, because in practice if we have more processors we scale the problem size to make use of all the computing power. By doing this it is possible to adjust the size so that the program always runs in the same amount of time for every $N \geq 1$.

In this context, Gustafson's law operates: we assume that the run time is constant, then we can say that for every number of processors $T(N) = s + p$, where s and p are the serial and parallel part of the program. As an approximation, the law considers that only the parallel part of the program scales with the problem size. The key here is that when the problem size scales, also the number of processors scales because run time remains constant; thus we can say that "the amount of work that can be done in parallel varies linearly with the number of processors" [35]. If we consider a serial processor to run the same task, then it would require time $T_{seq} = s + pN$ and the *scaled speedup* relative to this hypothetical serial processor would be

$$SS(N) = \frac{T_{seq}}{T(N)} = \frac{s + pN}{s + p} = s + pN = N + (1 - N)s.$$

Hence, when considering Gustafson's law, we do not have any kind of upper bound on the speedup, but of course, as for the strong scaling, we are not considering overheads and the scaling of serial parts of the problems that were approximated by the analysis from Gustafson.

During weak scaling analysis, the objective is to achieve a constant run time while varying the number of processors and the size of the problem linearly. In this analysis we are interested in two metrics: the execution time and the weak scaling efficiency, defined as

$$E(N) = \frac{T(1)}{T(N)}.$$

5.3.1 Platform details

In modern clusters and supercomputers, details are important when pursuing performance measurements of an application. As systems become bigger their complexity also grows both at the hardware and software level. For this reason gathering as much information as possible about the execution environment of the target application is important.

Clusters are employed by a community of people, thus their management is done by some kind of cluster management and job scheduling software in order to:

- allocate resources to users for some duration of time;
- provide a way to start, executing and monitoring jobs;
- manage contention of resources among users by creating queues of jobs.

The clusters where we had the chance to test our application on were managed by the Slurm [37] job scheduler. Our strong and weak scaling analysis was carried out on the CÉCI cluster NIC5 hosted at the University of Liège, with the following characteristics:

- 4672 cores spread across 73 compute nodes, each with two 32 cores AMD EPYC™ Rome 7542 [38] CPUs at 2.9 GHz;
- 256 GB of RAM;

- 100 Gbps InfiniBand HDR interconnect;
- 520 TB fast BeeGFS /scratch space.

To correctly exploit a processor, it is highly recommendable to study its microarchitecture and how to fine tune the execution to obtain the expected optimal behavior from it. The following in-depth analysis of the AMD Rome processor series makes use of the HPC Tuning Guide [39] from AMD, the work of Suggs, Subramony, and Bouvier [40] on the *Zen2* microarchitecture [40] and the in-depth comparison concerning AMD Rome series [41] from which we took the finely composed Figure 5.10.

AMD EPYC 7542 is a 32 cores socket, from the AMD EPYC 7002 series which is based on the AMD Zen2 microarchitecture [40] which supports Simultaneous Multi-Threading. The silicon package is composed by up to 8 *Core Complex Die* and a central *I/O Die*. Each socket supports up to 8 memory channels and 128 lanes of PCIe Gen 4.

Core Complex Die A Core Complex Die (CCD) contains the cores and the caches of the CPU. The Core Complex Dice are connected to the I/O Die by AMD's Infinity Fabric™ [38], this allows it to access memory, I/O and each other. Each Core Complex Die contains up to two Core Complexes.

Core Complex A Core Complex (CCX) contains up to four cores with a per-core L1 and L2 cache and a shared 16 MB L3 cache. The Infinity Fabric connects each Core Complexes to the I/O Die, but not to each other [41].

The Infinity Fabric also connects the CPUs in a two socket system, which is our case. All memory and I/O connect to the I/O Die, but they can be abstracted into four logical quadrants, each with two memory channels, 32 I/O lanes and up to two Core Complex Dice.

The quadrants partition is not only logical though; in fact all the cores in a single quadrant are physically closer to the two memory channels, thus forming a Non-Uniform Memory Access (NUMA) memory architecture in which each quadrant is a NUMA domain. This can also be seen from Figure 5.10, where each Core Complex Die has its nearest Infinity Fabric switch (used to route data in the I/O Die) connected to a Unified Memory Controller (UMC). Thus, accessing memory from another NUMA domain would imply to hop on multiple switch which will introduce more latency.

These NUMA domains can be exposed to the operating system through BIOS settings, allowing to partition the processor in one, two, and four NUMA domains, interleaving memory access across eight, four and two memory channels in each NUMA domain, respectively. Settings also allow interleaving memory access across all memory channels on

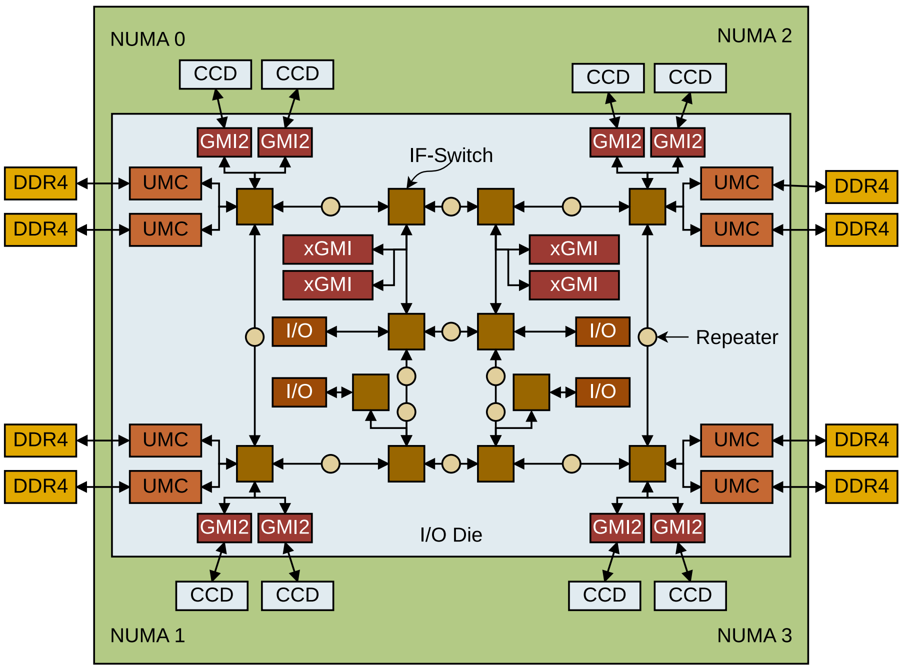


Figure 5.10: The AMD EPYC Rome processor layout (Velten et al. ACM 2022).

a 2-socket system, which is to be avoided for HPC systems, as it introduces inter-socket latency. Finally, each L3 cache can be exposed as a NUMA domain.

The AMD EPYC 7542 processor on our cluster is a 32 core socket, with 4 CCDs, one for each NUMA domain, each of which has 2 CCX. Each node has two sockets for a total of 64 cores and Simultaneous Multi-Threading is disabled. Memory is divided equally among NUMA domains, meaning that each of them will have 32 GB. Finally, the InfiniBand adapter is only connected to the PCIe adapter of the fourth NUMA domain in the first socket.

A key result is that benchmarks performed on the AMD Rome processors have shown that the RAM bandwidth of the memory channels relative to one CCD is saturated by just using three cores of a single CCX. Moreover, adding access from the second CCX on one CCD does not accomplish any increase of performance [41]. This is a behavior that has showed up during our scaling measurement and that, if not taken into account, can lead to suboptimal results.

Before moving to the scaling results, we remark how the problem is parallelized for the multicore execution: the model is partitioned using METIS and each partition is distributed to the processes that will deal with the computation. At each timestep there will be a *blocking* communication phase in order to exchange jumps data among processes.

5.3.2 Execution configuration

Our measurements were carried out on eight reserved nodes with the above-mentioned specifications. We analyzed scaling on one, two, four and eight nodes, distributing processes in a round-robin fashion over nodes, always starting with a number of processes equal to the number of nodes and subsequently doubling it.

As for strong scaling analysis, for each "*#nodes* : *#processes*" configuration we tested approximation orders from one to five having one pre-generated mesh per approximation order so that the number of degrees of freedom is roughly the same for all orders. For each iteration, the solver had to compute around 100 millions degrees of freedom.

In weak scaling we tested up to approximation order three, because we could not create a small enough mesh at higher order that would be still be comparable with the strong scaling results. Because meshes of different approximation order for the same number of processes must have the same amount of degrees of freedom, we had to generate different meshes for each "*order* : *#processes*" configuration. For the smallest mesh, the solver computes around 188 thousand degrees of freedom, while the larger is composed by 96 millions degrees of freedom. For both scalings we tested both the leapfrog and the RK4 methods.

The execution launch was made by using Slurm scripts with the `-exclusive` option, so that a job could not share nodes with other running jobs. Our application is parallelized using MPI, which allows to specify some options when launching parallel applications. Two options that can affect performance, especially with a NUMA microarchitecture, are the process *mapping* and *binding*. When talking about process assignment in the MPI context, we talk about *slot*, which is the allocation unit for a process. The number of slots on a node indicate how many processes can potentially execute on that node. Usually slots are different from hardware resources, for example because a resource manager could expose fewer slots than those available to MPI or because you can have more slots than cores. However, in our case slots and cores actually match, because we need that a process has the full computing capabilities of a core, and also for this reason we use the `-exclusive` option, so that we have the full node resources for our application.

With the process mapping policy, the process assignment to slots is made by iterating in a round-robin fashion over the specified unit. The process binding policy specifies to which unit the processes must be bound (another common term is *pinned*). For both policies, the unit can be one of, but not limited to: `core`, `L1cache`, `L2cache`, `L3cache`, `socket`, `numa`, `board`, `node`. If a mapping is specified but a binding is not, then the binding unit is the same unit of the mapping.

So, for instance, if four processes are mapped by socket and bound to core on a node with two sockets, then process with ranks 0 and 2 will be assigned to two cores on socket 0 and processes with ranks 1 and 3 will be assigned to two cores on socket 1.

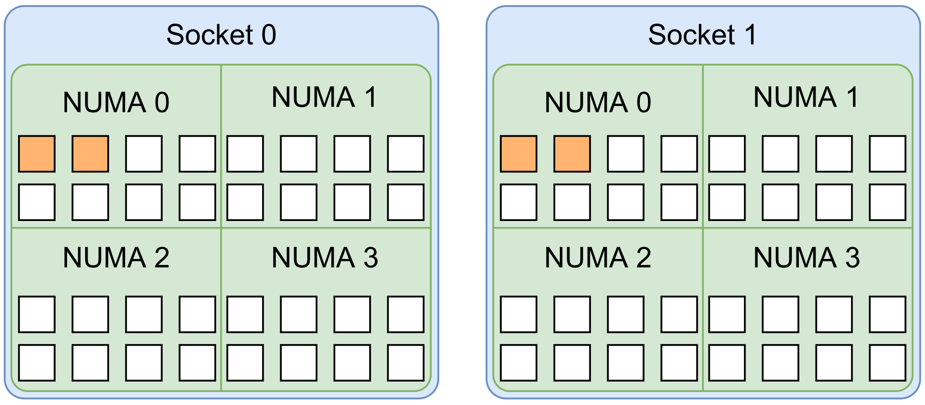
The default mapping policy is by socket, while the default binding policy depends on the number of processes: processes are bound to core when the number of processes is less or equal than two and are bound to socket otherwise.

Our first testing was made without specifying any kind of mapping or binding, thus using the default mapping and binding. This led, as will be seen, to a linear but inefficient scaling, which is due to having ignored the NUMA domain configuration on the test processor. The second testing, in which we specified the mapping, in this case by NUMA domain, led to much better performance, with heavy superlinear speedup when scaling on multiple nodes, probably due to an optimal bandwidth usage and to an increased number of available caches.

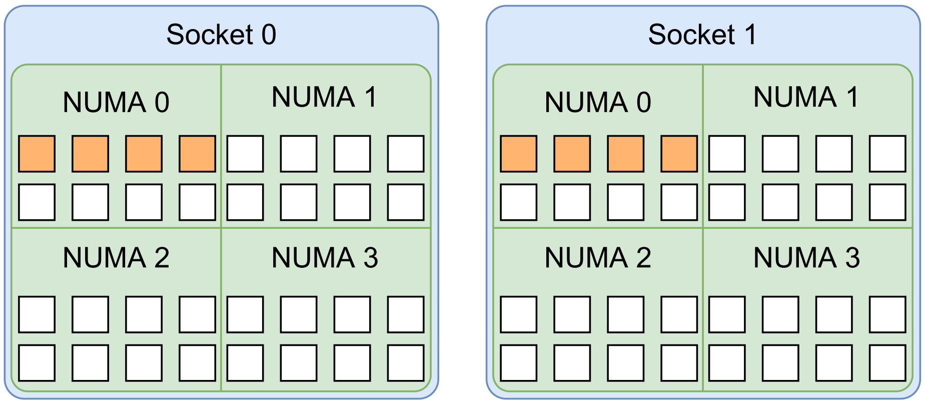
5.3.3 Strong scaling

The following pages contain the results of the strong scaling analysis. The first thing that catches the eye in our opinion is the high variability of the communication times. As we also mentioned before in Subsection 5.2.1, this could be due to the configuration of the facilities used for the measurements. However, from the results we can also see some regularities, thus we are still investigating this behavior. In fact, the strong scaling results show a regularity in its irregularity: there is always an abnormal peak in communication time at approximation order one, with 16 processes no matter the mapping policy or the number of nodes. Communication time also starts high with high approximation orders, and then it descends with a minimum at 32 processes. This is not always the case with approximation order three, which shows a minimum at 32 processes when using leapfrog time integration and a minimum at 8 processes when using RK4. With lower approximation order, this minimum seems to be always at eight processes. This number of processes for which the communication time is minimum is probably the best balance between exchanged data size and communication overhead.

The second thing that stands out is the shape of the strong scaling curve, and especially the difference between the one using the mapping by socket and the mapping by NUMA domain. The curve of the former starts linear, with a high enough efficiency, then speedup decreases and thus efficiency drops and becomes almost constant, but the scaling continues to be linear. This is a very bizarre curve coming out from a scaling analysis. The curve obtained with the mapping by NUMA domain is a more “traditional” one: it starts linear, with very high efficiency, the phenomenon of superlinear scaling comes in, and finally it flattens out at the end, usually due to a bandwidth saturation effect. The bandwidth



(a) Four processes are assigned with the socket mapping policy on a node.

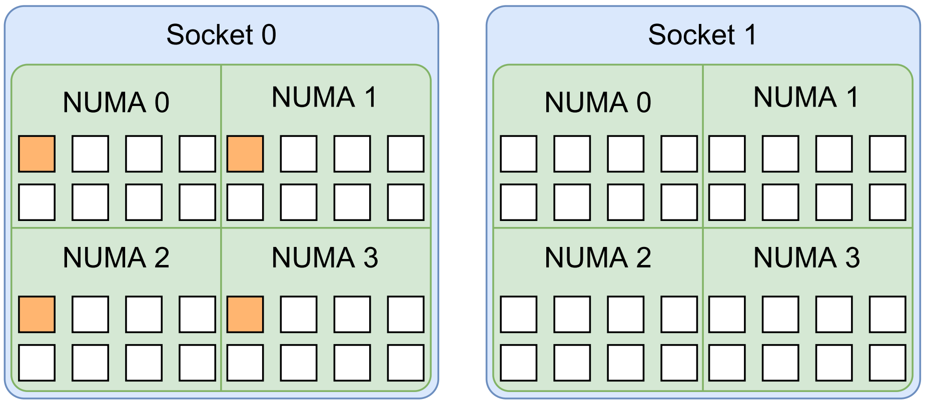


(b) Eight processes are assigned with the socket mapping policy on a node.

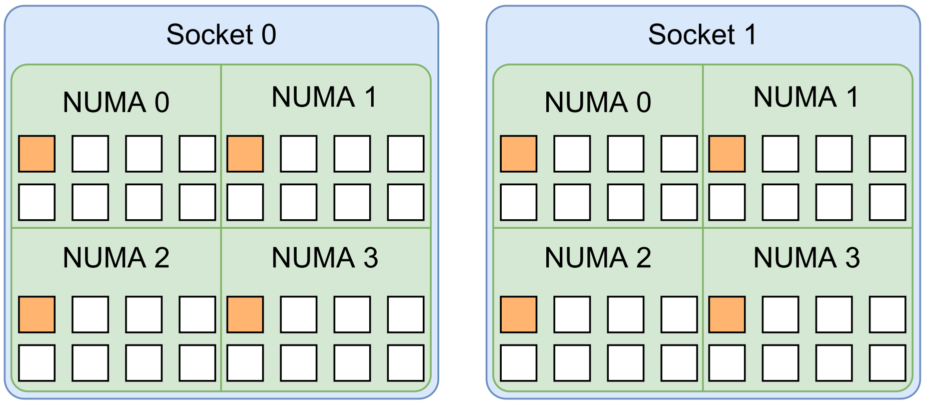
Figure 5.11: Process assignments to slots with the socket mapping policy with four and eight processes over a single node. Bandwidth for NUMA domains is not saturated in (a), while it is in (b).

is indeed the key here: both the drop of performance with the socket mapping and the flattening of the curve in the NUMA domain mapping are due to bandwidth saturation; the difference is when, where and why the saturation happens.

We anticipated it in the previous sections, but here we will explain it in detail. The saturation of bandwidth is actually at NUMA domain level: one NUMA domain corresponds to one CCD on the AMD Epyc processor, thus as mentioned in Subsection 5.3.1 and as extensively investigated in the work of Velten et al. [41], and the bandwidth for one NUMA domain is saturated by using just three of the cores that resides on it [41]. This explains the behavior of both curves: if n is the number of nodes, then for the socket mapping the drop in performance always happens between $4n$ and $8n$ processes, exactly when the number of processes per NUMA domain goes from two to four. We show this situation in Figure 5.11 showing how the processes are mapped using the socket mapping



(a) Four processes are assigned with NUMA domain mapping policy on a node.



(b) Eight processes are assigned with NUMA domain mapping policy on a node.

Figure 5.12: Process assignments to slots with the NUMA domain mapping policy with four and eight processes over a single node. Bandwidth for NUMA domains is not saturated in both (a) and (b).

(binding is not shown in the pictures) on one node. We do not have a flattening of the curve but a linear behavior because when the number of processes is doubled, the bandwidth of another NUMA domain is also exploited.

If the mapping policy is by NUMA domain, we have that the bandwidth of the NUMA domain is saturated much later, because processes are first distributed among those. The drop in performance happens between $16n$ and $32n$ processes, as before, when the number of processes per NUMA domain goes from two to four. Figure 5.12 shows the assignment to slots of four and eight processes using the NUMA domain mapping, thus, the NUMA domains are saturated at the same time when there are 32 processes on each node (four per NUMA domain) and subsequently the scaling curve flattens because there is no bandwidth left to exploit. By looking at the plots of speedup with the same number of processes and nodes we can see that the end point of the curve for both socket and NUMA domain mapping is almost the same for each approximation order. This is a natural consequence because regardless of the mapping policy, the final assignation of processes to slots will be the same.

5.3.4 Weak scaling

After the strong scaling plots, the results pertaining the weak scaling analysis are shown. In weak scaling we want time to remain constant while increasing the processors and problem size linearly. The results that we get from the socket mapping are in line with those obtained in the strong scaling: the number of degrees of freedom per second is almost the same, but the efficiency is a bit lower, probably due to the fact that in the weak scaling we start with a mesh having very few elements. With this processes mapping, communication time also show variability with a certain degree of growth that increases with the number of nodes.

The results with the NUMA mapping also show a similarity with the strong scaling. We do not get any superlinear behavior because the size of the problem grows with the number of processed, but a perfectly linearity of the scaling can be observed by looking at the number of degrees of freedom computed, as well as the time for each iteration, which remains constant until the saturation of the bandwidth comes in.

Differently from the socket mapping policy, with the second one the communication time is also kept constant most of the time and grows only with a higher number of nodes and processes, especially with the RK4 method.

Wherever possible, we show the performance results obtained by the GPU execution on the same meshes used for the weak scaling analysis. In fact, not all meshes could be tested because there was not enough memory on the GPU. The performance in terms of degrees of freedom is stable from the mesh used for eight processes on. Low performance before that point are essentially caused by the size of the mesh that is too small. As expected, the iteration time increases with the mesh size because we are working with only one device.

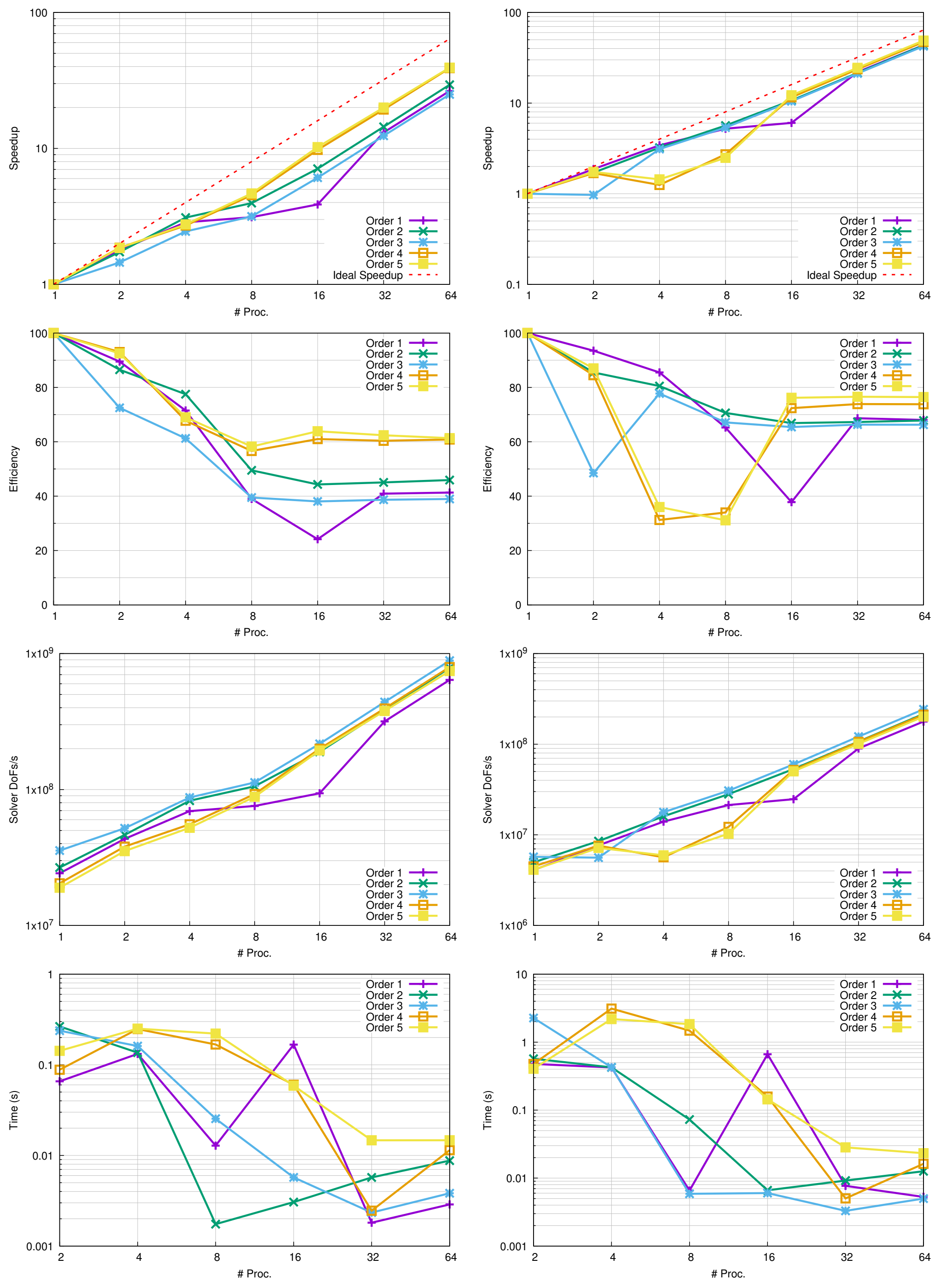


Figure 5.13: Strong scaling analysis of the solver on one node using the `socket` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: speedup, efficiency, degrees of freedom per second and communication time.

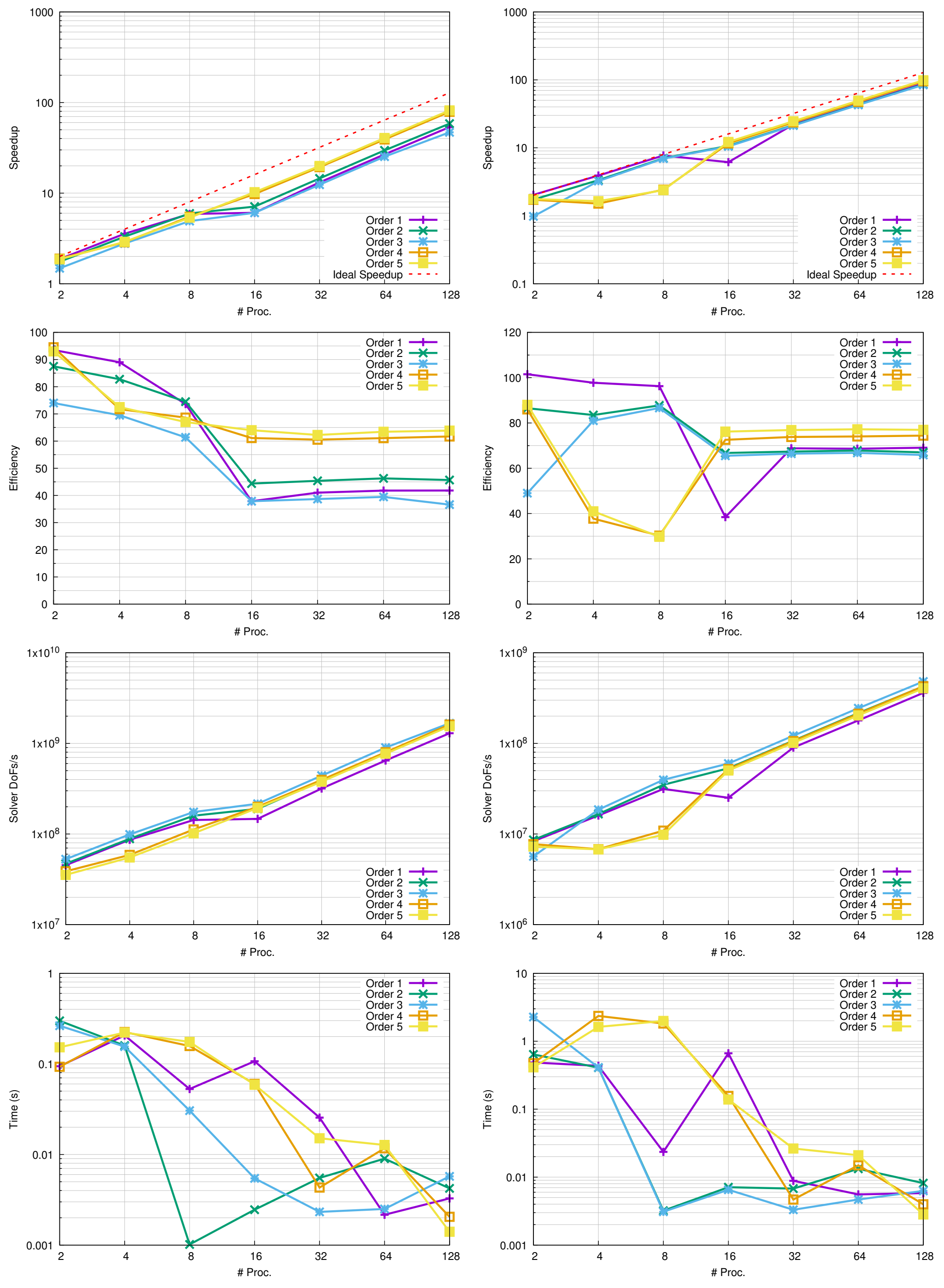


Figure 5.14: Strong scaling analysis of the solver on two nodes using the `socket` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: speedup, efficiency, degrees of freedom per second and communication time.

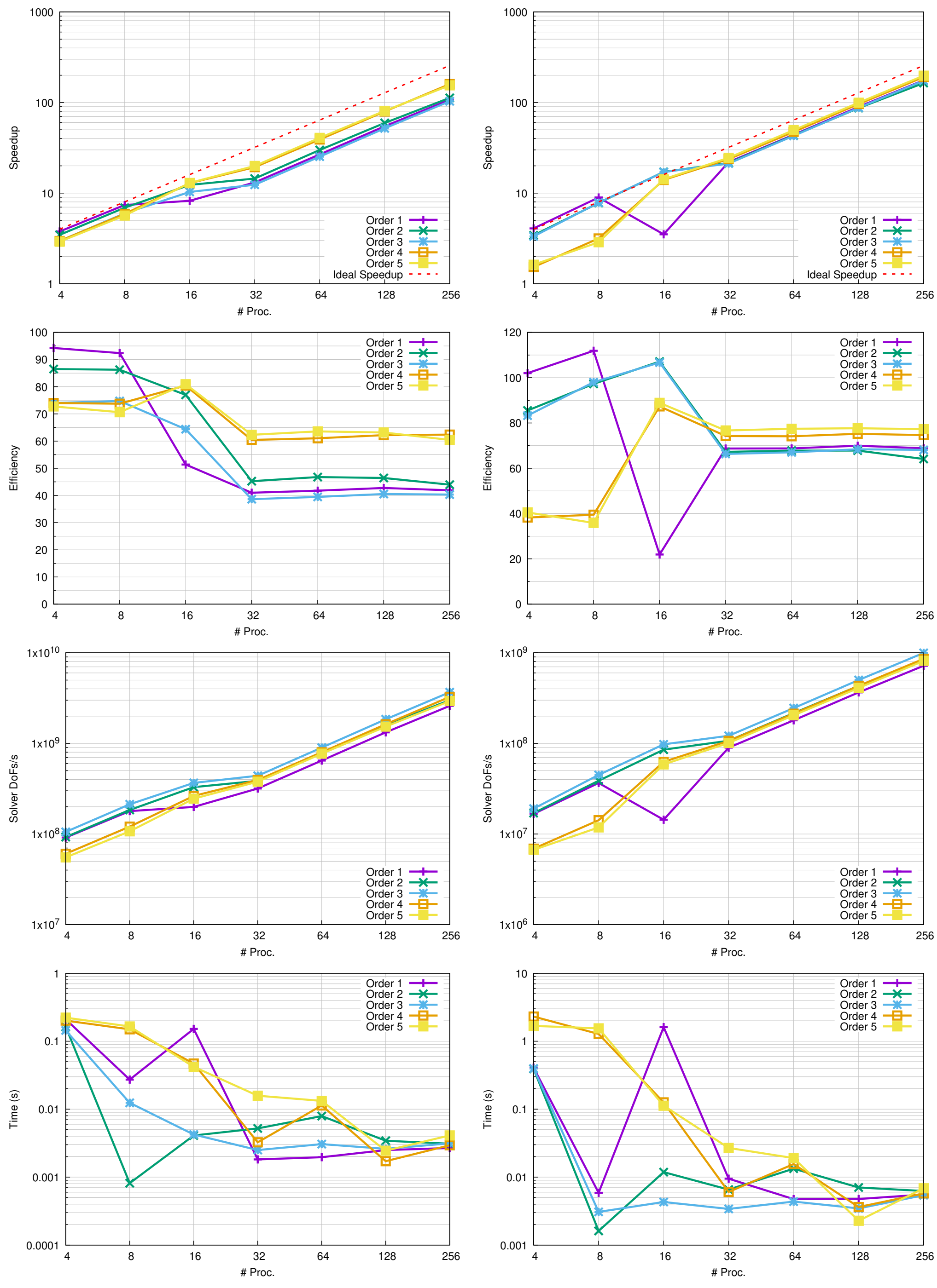


Figure 5.15: Strong scaling analysis of the solver on four nodes using the `socket` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: speedup, efficiency, degrees of freedom per second and communication time.

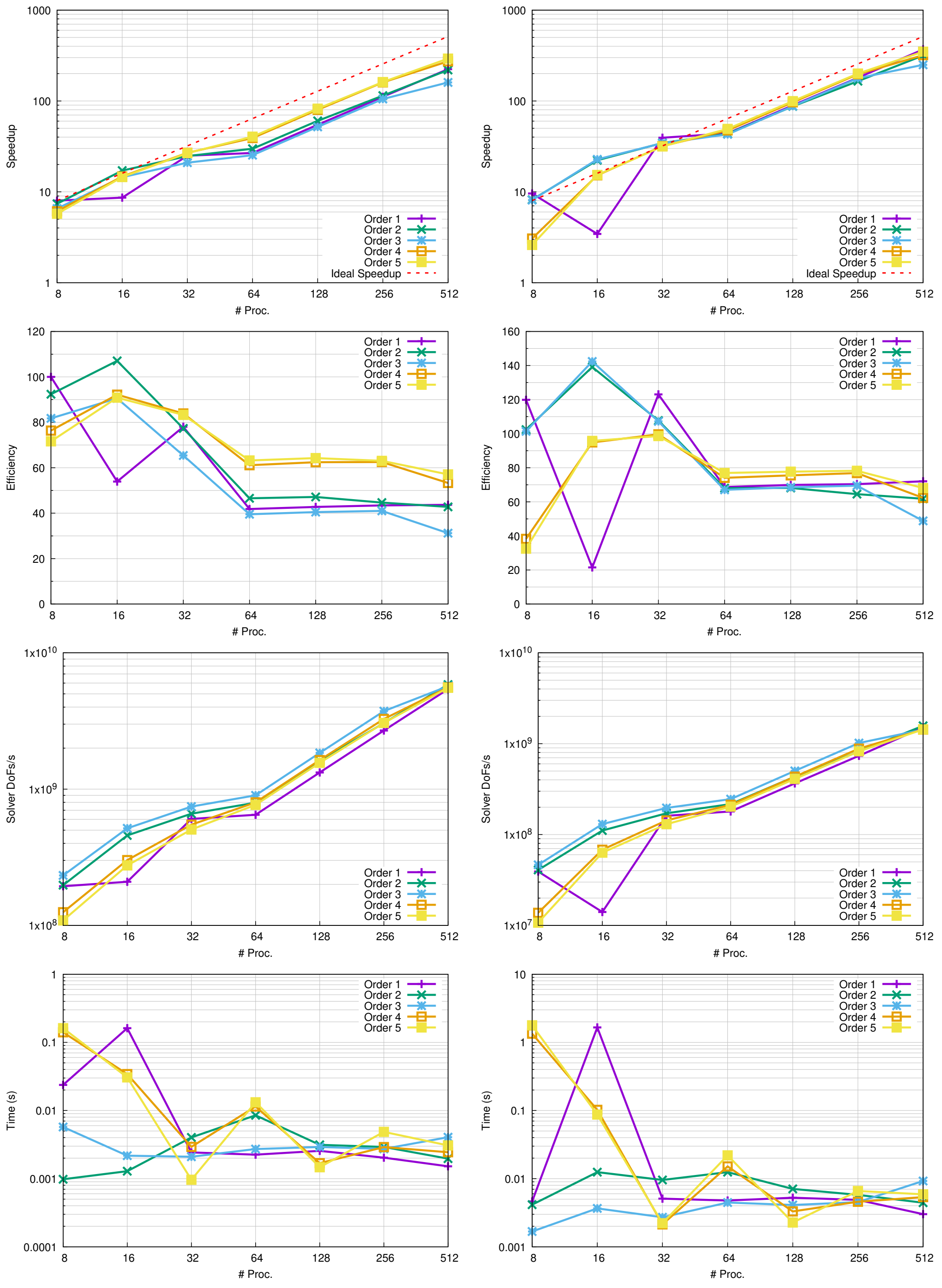


Figure 5.16: Strong scaling analysis of the solver on eight nodes using the `socket` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: speedup, efficiency, degrees of freedom per second and communication time.

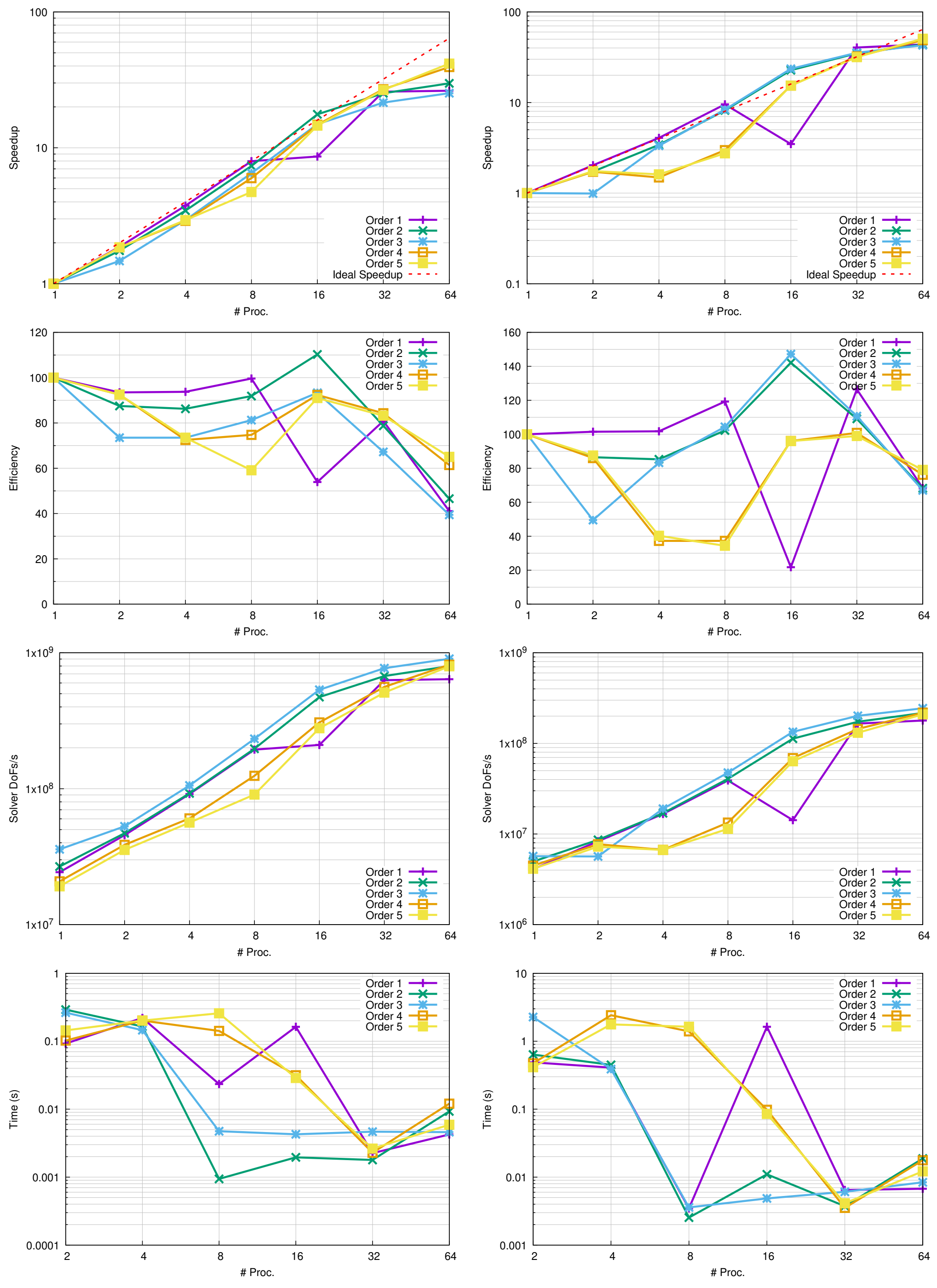


Figure 5.17: Strong scaling analysis of the solver on one node using the `numa` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: speedup, efficiency, degrees of freedom per second and communication time.

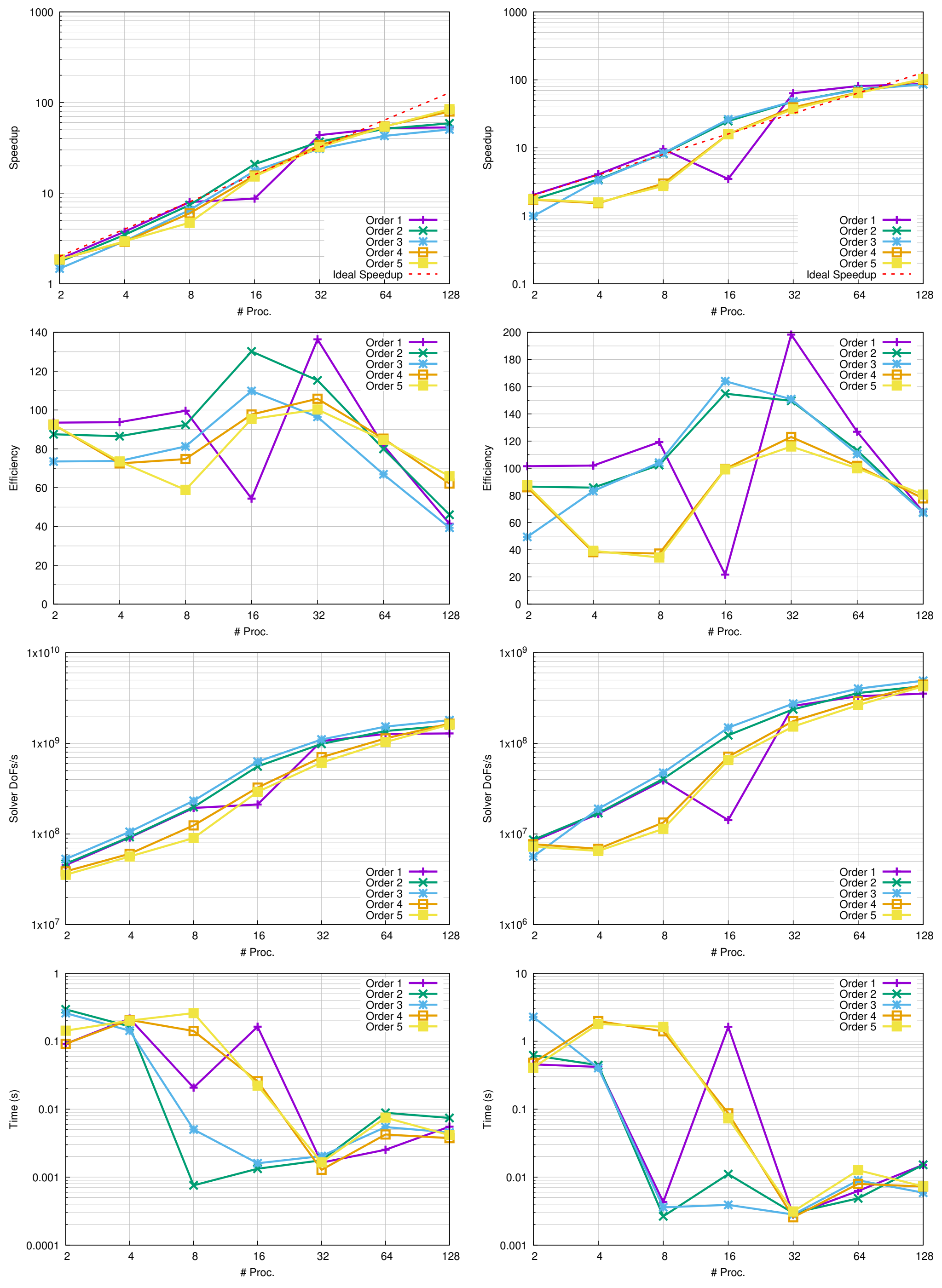


Figure 5.18: Strong scaling analysis of the solver on two nodes using the `numa` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: speedup, efficiency, degrees of freedom per second and communication time.

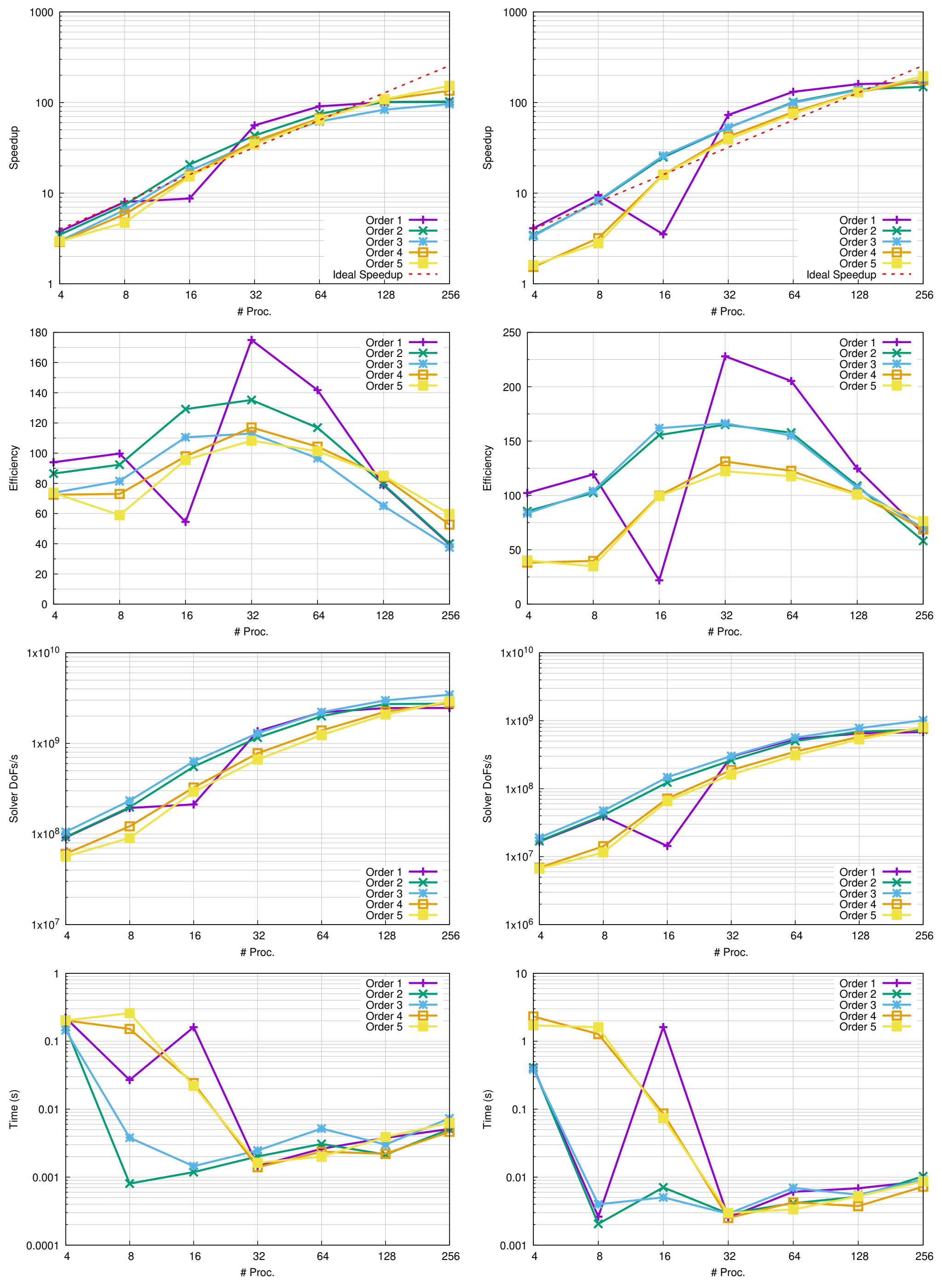


Figure 5.19: Strong scaling analysis of the solver on four nodes using the `numa` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: speedup, efficiency, degrees of freedom per second and communication time.

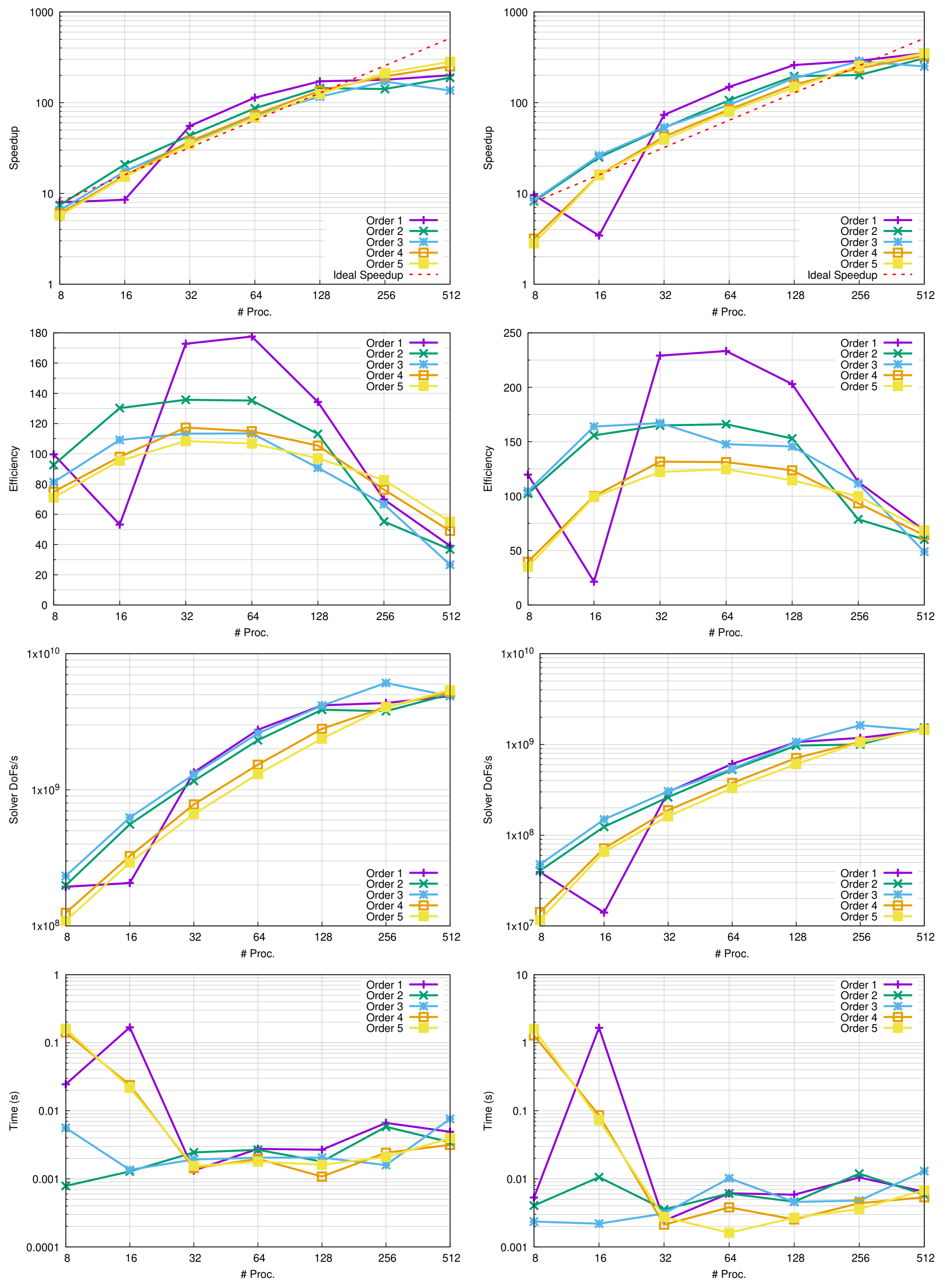


Figure 5.20: Strong scaling analysis of the solver on eight nodes using the `numa` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: speedup, efficiency, degrees of freedom per second and communication time.

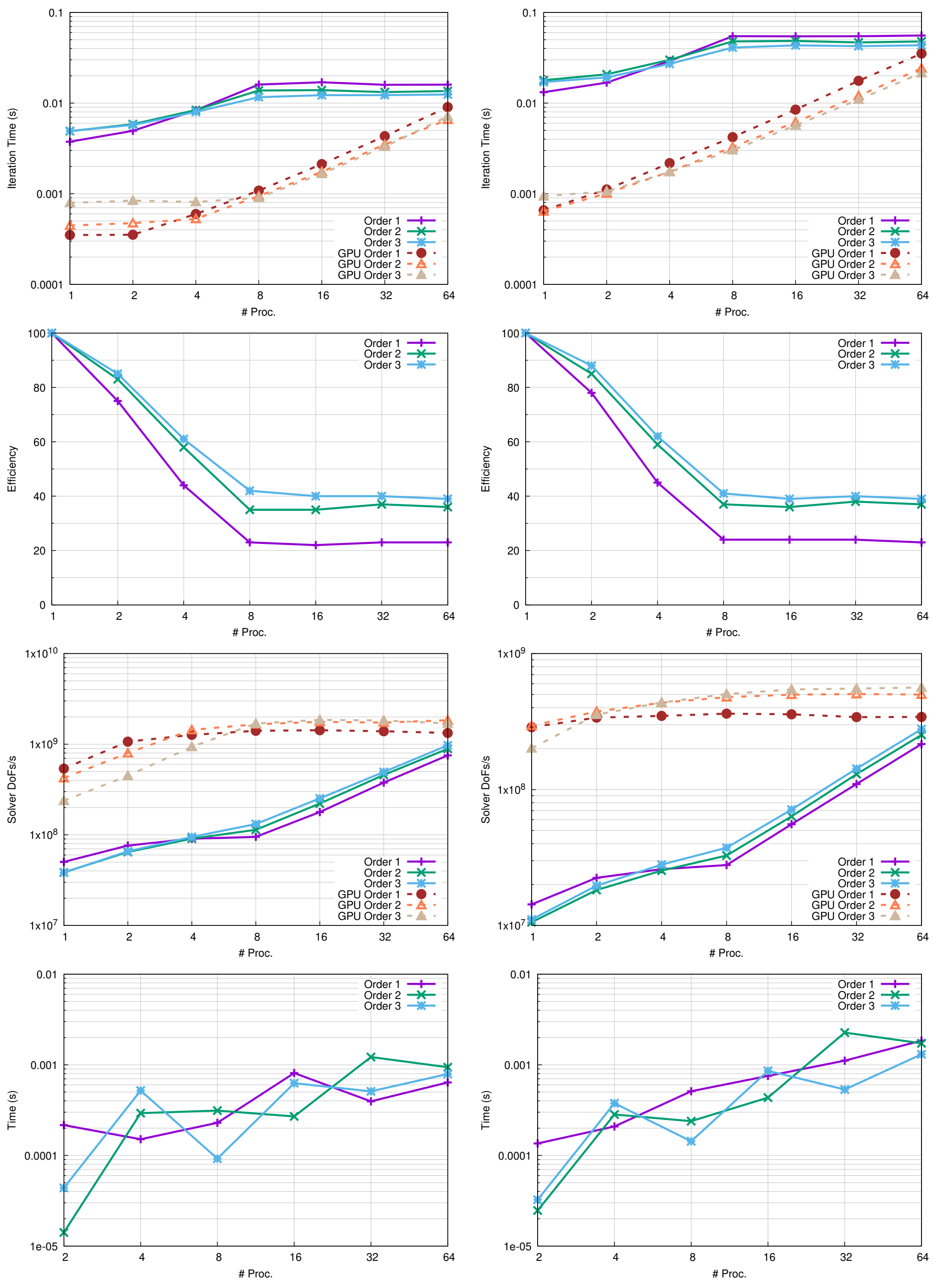


Figure 5.21: Weak scaling analysis of the solver on one node using the `socket` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: iteration time, efficiency, degrees of freedom per second and communication time.

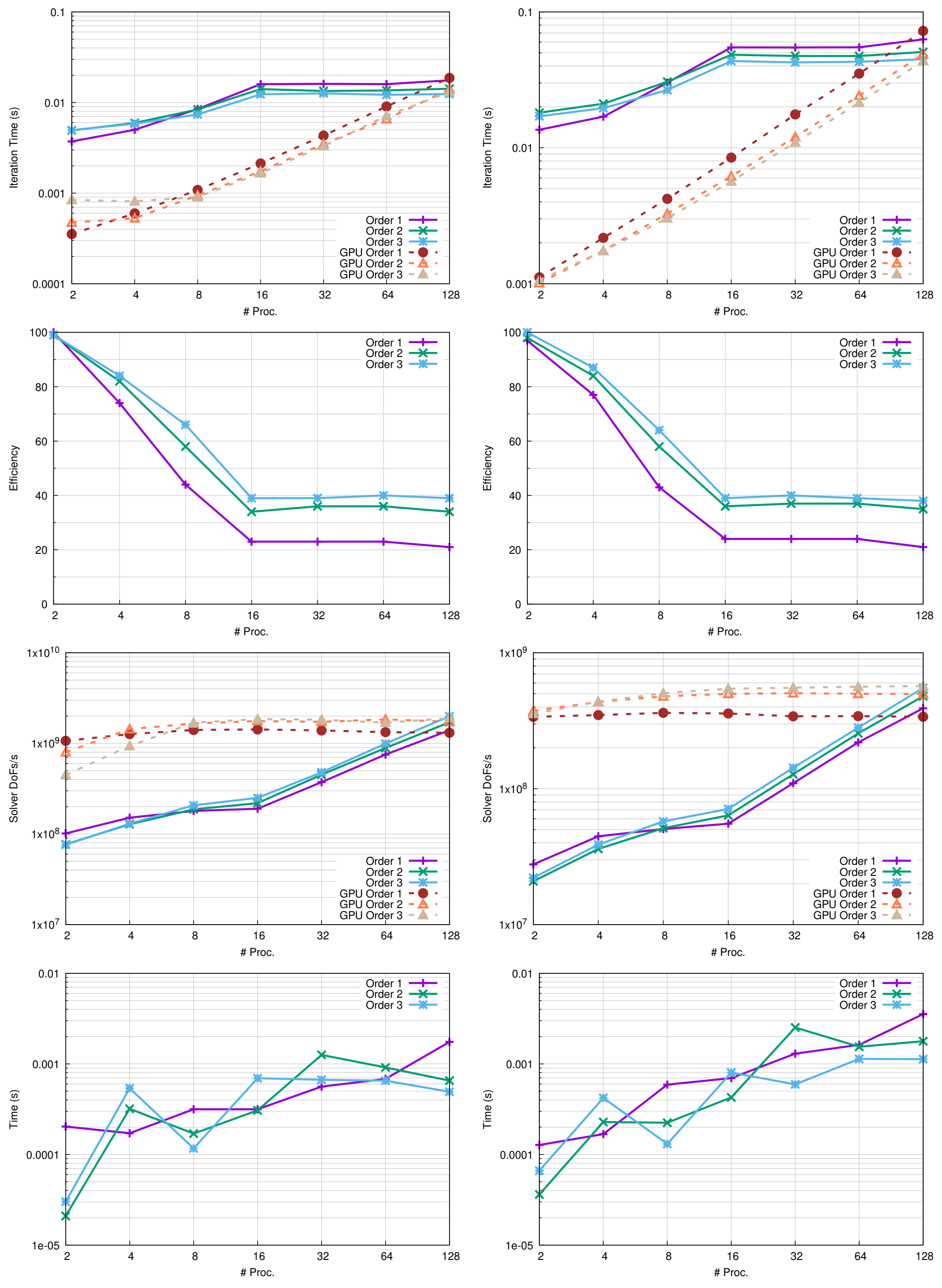


Figure 5.22: Weak scaling analysis of the solver on two nodes using the `socket` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: iteration time, efficiency, degrees of freedom per second and communication time.

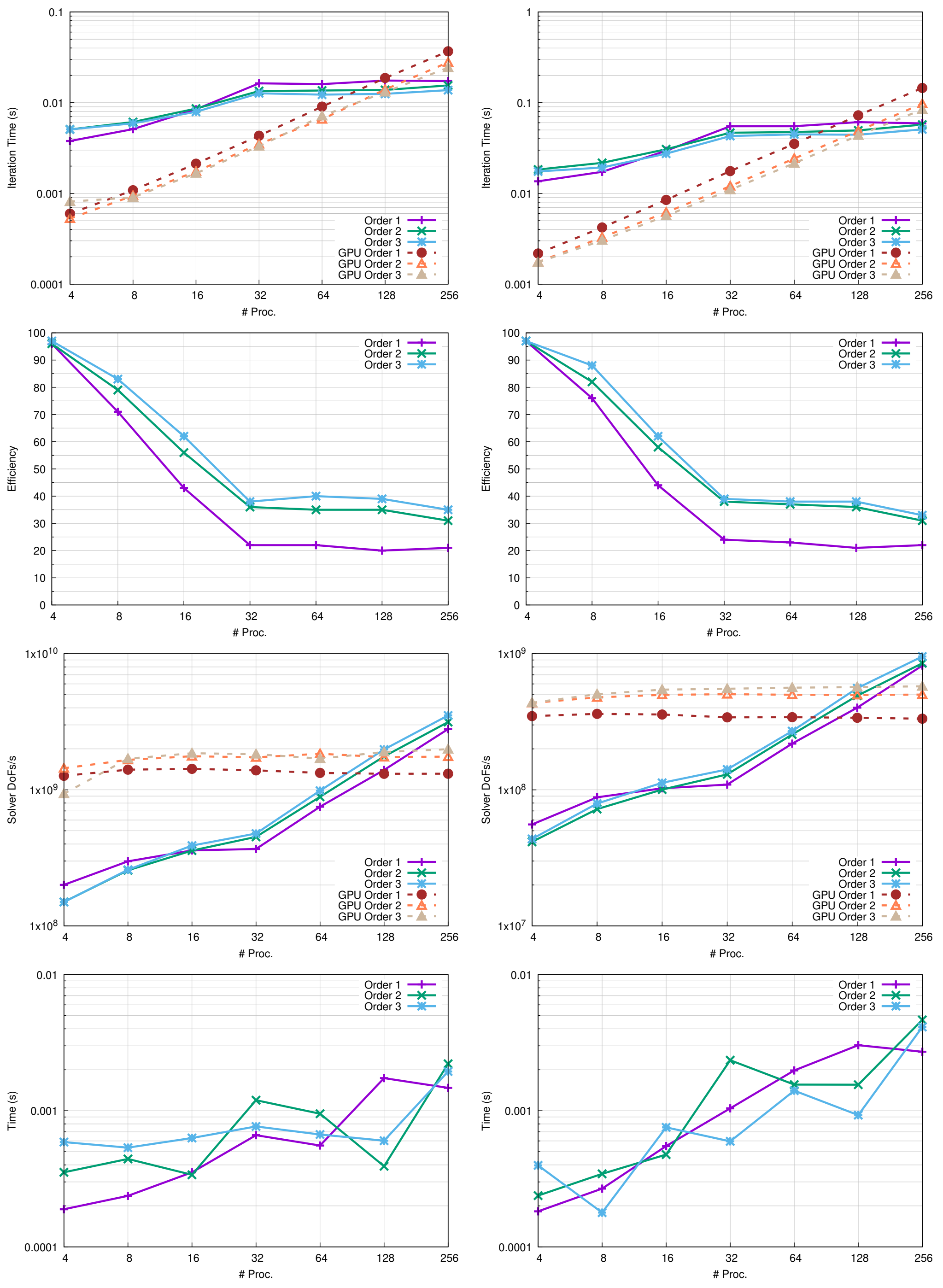


Figure 5.23: Weak scaling analysis of the solver on four nodes using the `socket` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: iteration time, efficiency, degrees of freedom per second and communication time.

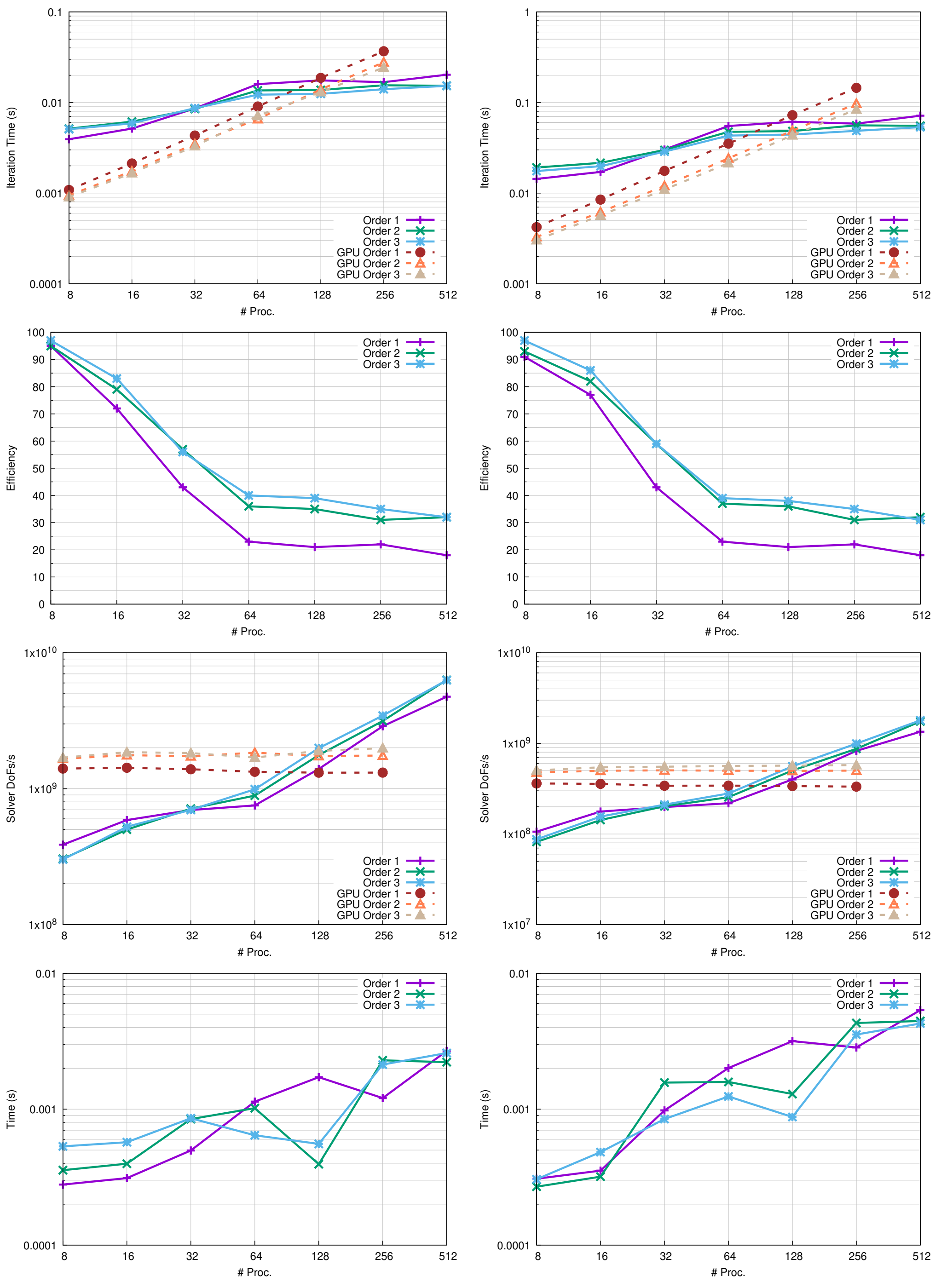


Figure 5.24: Weak scaling analysis of the solver on eight nodes using the `socket` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: iteration time, efficiency, degrees of freedom per second and communication time.

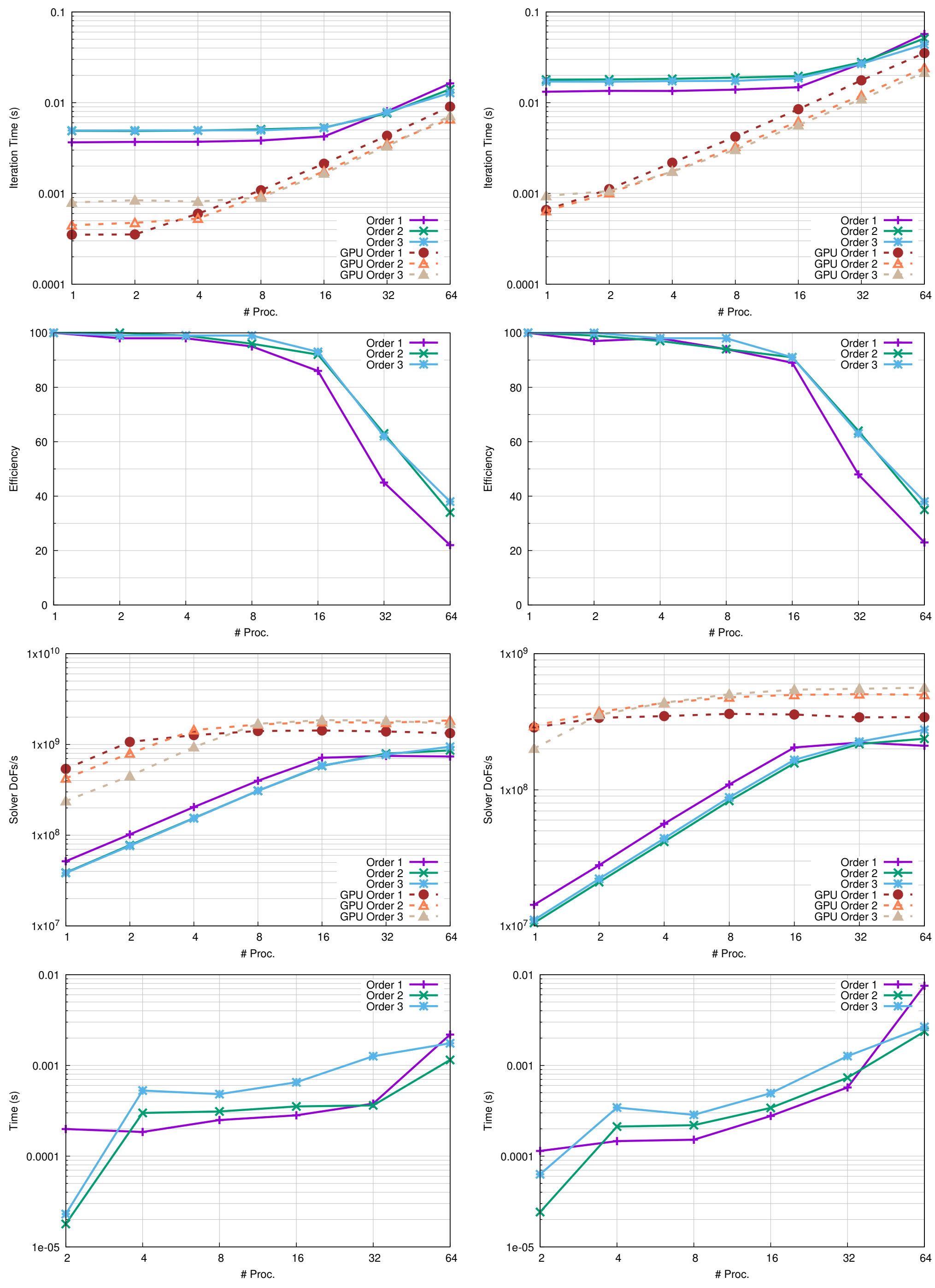


Figure 5.25: Weak scaling analysis of the solver on one node using the `numa` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: iteration time, efficiency, degrees of freedom per second and communication time.

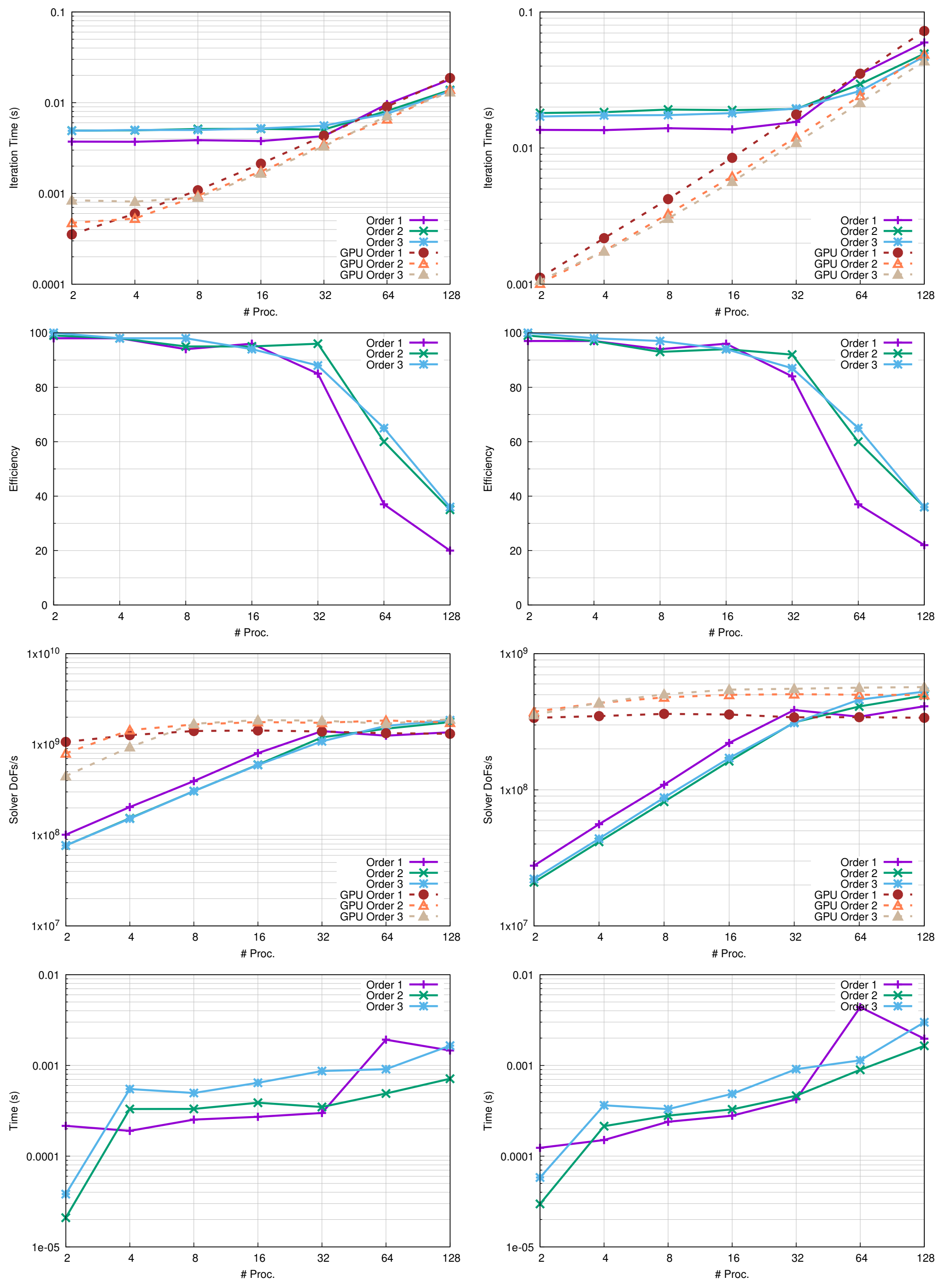


Figure 5.26: Weak scaling analysis of the solver on two nodes using the `numa` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: iteration time, efficiency, degrees of freedom per second and communication time.

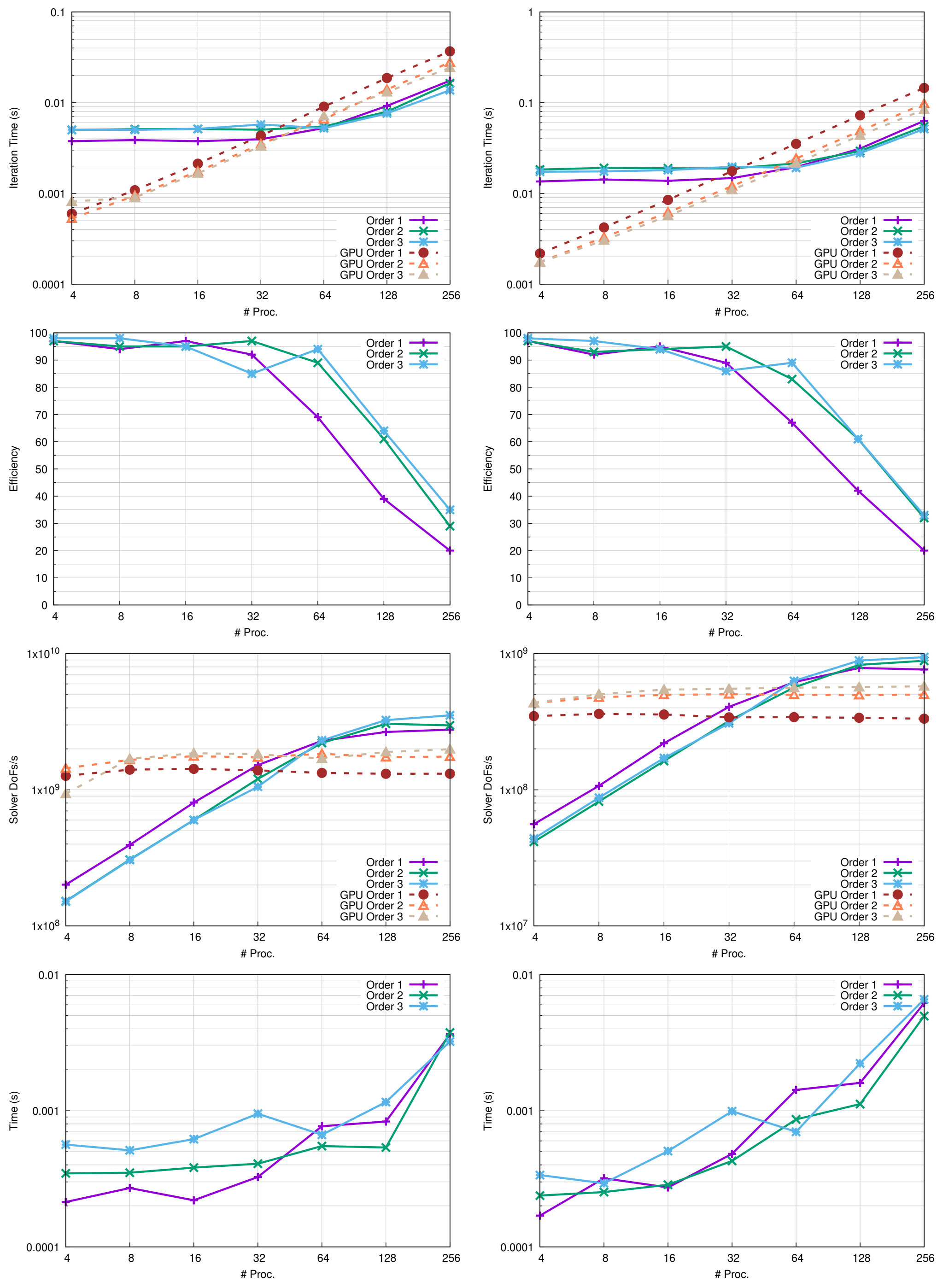


Figure 5.27: Weak scaling analysis of the solver on four nodes using the `numa` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: iteration time, efficiency, degrees of freedom per second and communication time.

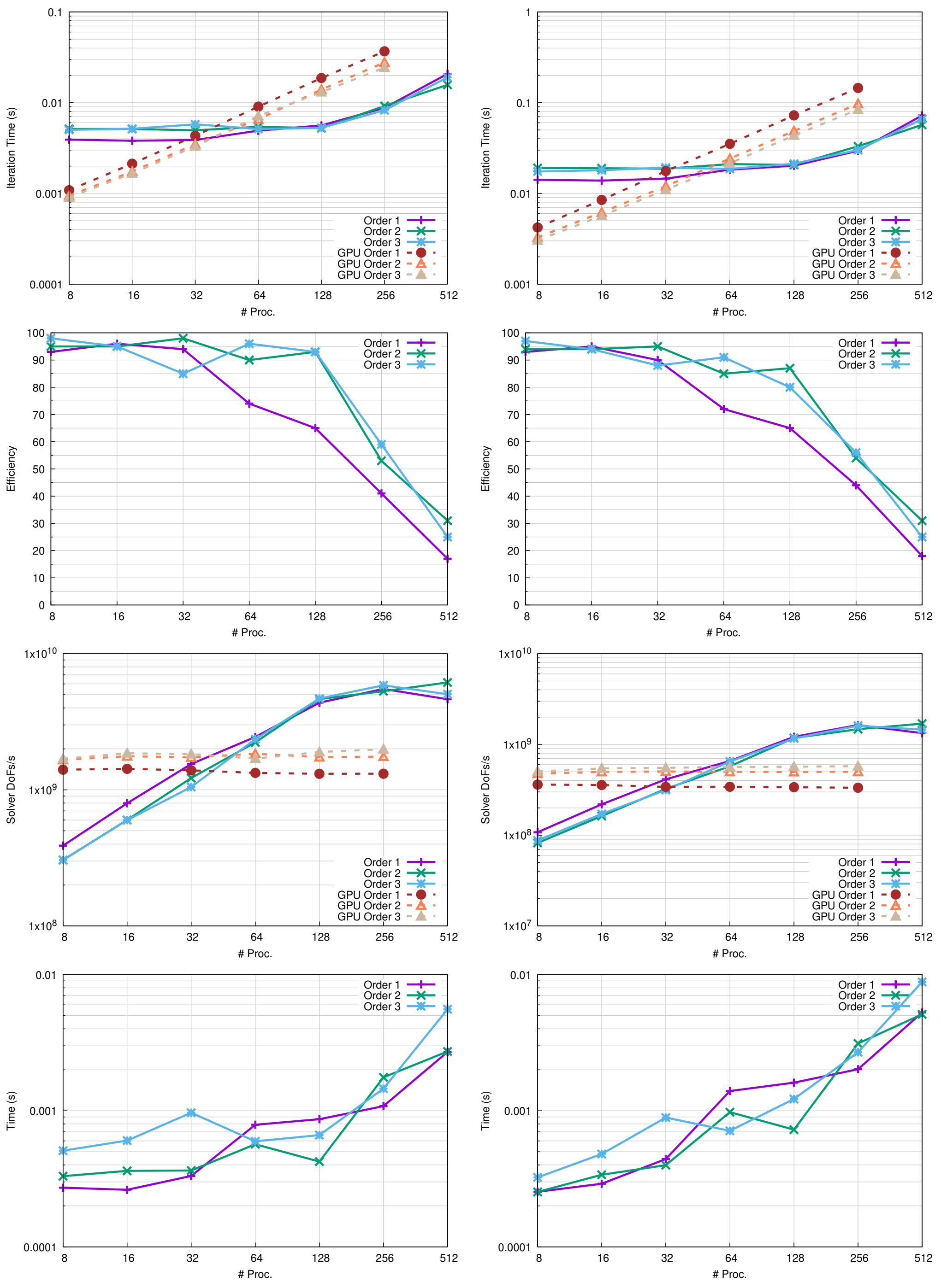


Figure 5.28: Weak scaling analysis of the solver on eight nodes using the `numa` mapping, the left column shows the results using the leapfrog method and the right column shows the results using the RK4 method. The plots show respectively: iteration time, efficiency, degrees of freedom per second and communication time.

5.4 Validity Evaluation

Due to the large experimental part of this thesis, it is important to point out the validity of our results and conclusions. Thus, in this section we will state why the research findings are adequate and whether they can be generalized. The evaluation presented in this section can be applied also the experiments that will be shown in Chapter 6, since the methodology is the same.

5.4.1 Threat classification

The need for validation of results is dictated by the fact that during the experimental phase, some fallacies could be introduced because of a series reasons. We call these fallacies *threats*. A classification scheme of threats was given by Cook, Campbell, and Day in 1979, targeting a broad variety of experiments [42]. In this context an experiment is usually needed because a research want to prove some kind of causal relationship. Thus, in order to prove a research hypothesis one has to test it experimentally. These principles are shown in Figure 5.29: on top there is the hypothesis area. The conclusion is based upon the observation part shown in the bottom area.

Threats to validity can be found in the steps involved in the conducting of an experiment. In particular, threats can be found when we move from the theory to the observation, or from the cause to the effect, both in the theoretical and the experimental area. Thus, in general, there are several types of validity to be assessed:

1. *Conclusion validity* concerns the statistical relationship between treatment and outcome, making sure that there is a relationship with a given significance.

2. *Internal validity* is about making sure that the relationship observed between treatment and output is a causal relationship, and that all the factors that lead to that outcome have been considered.

3. *Construct validity* is concerned with the relation between theory and observation. Essentially the validity of the conclusions could be affected by the fact that the treatment and/or the outcome do not reflect well, respectively, the cause and the effect.

4. *External validity* deals with generalization, thus if there exist a relationship between cause and effect, we must assess whether it can be generalized out of the scope of the study.

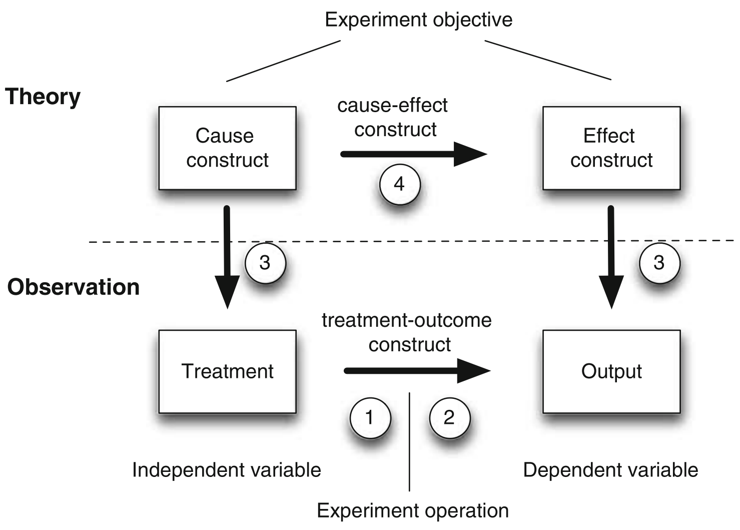


Figure 5.29: Experiment principles (Wohlin et al. Springer 2012).

The experimental principles shown in Figure 5.29 are very broad and can apply to several kinds of experimentation in many fields, so we now explain how our methodology maps to the theoretical framework. In our case the treatment in Figure 5.29 is the experiment input, thus the Lua configuration script, the mesh and the execution configuration. The output is the data relative to the memory traffic, number of operations and performance for the analysis of kernels, and are the average iteration time, the number of DoFs/s and communication time for the scaling analysis.

The observation part maps well, but the same cannot be said for the theoretical part, because the methodology of our analysis is slightly different from the usual experiments. In our context we cannot make theoretical assumptions on the performance of the code: we have to run it in order to get the data. Then, based on the data we can draw our conclusions, thus the process is reversed: from observation we move to theory. This requires to pay a lot of attention to external validity, because the conclusions drawn based on the experimentation on one environment could not be the same on another.

5.4.2 Threats mitigation

Once the basis of validity evaluation are given, and we have explained how our methodology interfaces with it, we can move to explain more in detail which threats could affect the work and how they were mitigated. The threats considered are taken from Wohlin et al. [43], and since our methodology is different, construct validity threats do not apply to our analysis.

Conclusion validity

Threats to this validity comes from issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment.

Low statistical power The power of a statistical test is the ability of the test to reveal a true pattern in the data. This threat has been mitigated at two levels: (i) when measuring performance for both kernel and scaling analysis we launched the solver with a high number of iteration in order to get meaningful results; (ii) necessarily, for the scaling analysis, we had to perform a high number of experiments and, albeit with different configurations, the pattern of data was always reconfirmed.

Reliability of measures Concerning kernel analysis, our measures were performed with specialized tools that allow to take precise measurements of memory traffic, number of operations and performance. The data obtained for the scaling analysis, thus primarily time measures, were taken using standard functions in C++ for the CPU implementation and using CUDA events for GPU, both of which are reliable.

Reliability of treatment implementation The implementation of the treatment means the application of treatments to subjects. In the context of this work the subjects are the machine on which we run the solver. During the scaling analysis the solver was launched on the cluster through the SLURM scheduler, we prepared scripts in order to submit the job, thus the correct configuration was always enforced.

Random irrelevancies in experimental setting This threat refers to elements outside the experimental setting that may disturb the results. In our case we had such a threat with the variability of communication time during the scaling analysis. However, this “noise” did not affect the analysis considerably, as it did not occur consistently, and also due to the large number of experiments, the results are still consistent.

Internal validity

Threats to internal validity are influences that can affect the experiment, without any awareness from the researcher. Thus, they threaten the conclusion about a possible causal relationship between treatment and outcome.

History When experiments are applied different times to a subject, there could happen that the circumstances are not the same. As before, the subject are the machine on which we run the application. This mainly concerns the scaling analysis, where many repeated experiments were executed. The condition in which we made the measurements were always the same, especially because the solver was launched on reserved nodes on the cluster, thus without the threat of resource contention.

Instrumentation If the artifacts for collecting data are badly designed the experiment is affected negatively. In our context this is very related to the reliability of measures: since we get our data using standard methods and profiling tools, this risk is totally mitigated.

External validity

Threats to external validity are conditions that do not allow us to generalize the results and the conclusion of our experiments.

Interaction of selection and treatment This is an effect of having a subject population, not representative of the population we want to generalize. This threat affects the scaling analysis, in fact we did not measure the multicore execution on every kind of system, for example we did not target Intel processors, and we only tested it on one cluster. Anyway, this does not jeopardize our conclusions: the execution on other processors can be tuned by changing the configuration execution in order to adapt to a given system, an example of this is the process mapping discussed in Subsection 5.3.2.

Interaction of setting and treatment This is the effect of not having the experimental setting or material representative of, for example, industrial practice. This is also a threat for us, and it is also the cause of the previous threat being present in our analysis. As we mentioned earlier, we can extend our conclusions on multicore execution to other multicore systems, however we cannot with regard to the multi-GPU execution that will be discussed in Chapter 6. The problem with generalization in this case is that we did not have the platform on which to test, thus we only assess the performance on the platform at our disposal and leave this generalization to future works.

Chapter 6

Multi-GPU implementation

Multi-GPU is probably the best approach for solving massively-parallel and large-scale problems using general-purpose devices. Thus, we need two things: a problem that can be parallelized, and a large amount of data; we have a strategy to parallelize the DG method, and we also have large amount of data to be computed because of more complex and precise simulation to be performed. The multi-GPU approach also permits tackling larger problems than a single GPU approach with a better efficiency than a multicore approach. In fact when we increase the number of GPUs we also increase the available memory, that as mentioned in the previous chapter is a limited resource when working on a single GPU. Finally, the need for a multi-GPU approach is also justified by the fact that in order to obtain the same performance of a single GPU accelerator, we need much more CPUs, which means a greater consumption of money and energy.

In this chapter we will give an overview about multi-GPU architectures, how they can be built and which are some of the problematics that arise in the development on GPU clusters. Subsequently, we will address our implementation and the reference architecture for our implementation, along with some observations on the expected performance based on the problem size. Finally, we will present some early results of the implementation on a small GPU-powered cluster.

6.1 Multi-GPU architectures

Differently from multicore systems, that have been profusely studied, multi-GPU systems and execution are far less documented. This is due to different factors: the lack of a unified interface for programming the GPU systems, as evidenced by the presence of vendor-specific platforms; different technologies for interconnecting the GPUs and ultimately the lack of a technology that can create the right amount of abstraction over the components of the system. There exist some technologies that are tackling this challenge, but they have not yet reached maturity and support in every environment. The programming model that is trying to do so is SYCL [12], but it is very far from being a de facto standard as the MPI parallel programming model.

6.1.1 GPU interconnects

As a first step, we will highlight a variety of GPU interconnects technologies. This section is based on the work of Li et al. [44], in which they present and evaluate a number of modern GPU interconnects and show their performance in terms of latency and bandwidth exploitation in Peer-to-Peer and collective communication patterns [44].

PCIe Peripheral Component Interconnect Express is a high-speed serial expansion bus standard and is the most traditional method to connect a GPU to the CPUs. PCIe is much slower than the connection between CPUs and DRAM, thus could become one of the bottlenecks in a multi-GPU architecture. Other than PCIe, systems that support multiple socket could employ also a CPU-CPU interconnect functioning as a bridge, such as AMD's Infinity Fabric [38] or Intel's QuickPath Interconnect (QPI) [45].

NVLink NVLink is a proprietary technology from NVIDIA designed as a high-bandwidth interface that supports GPU-GPU and GPU-CPU communications [46]. The first generation of NVLink was available with the Pascal core architecture, while the fourth generation NVLink is expected to be out with the Hopper architecture. It allows direct read and write on host-memory of the remote CPU and/or the device-memory of the peer GPU. NVLink is a bidirectional interface, in first generation data is sent at up to 20 GB/s, yielding a peak bidirectional bandwidth of 40 GB/s for single NVLink [47]. Furthermore, there can be multiple links per GPUs, four for the first generation, thus increasing the maximum reachable bandwidth to around 160 GB/s. The declared peak bandwidth for fourth generation is 900 GB/s. Based on how the NVLink slots of the GPUs are connected to each other, the topology and bandwidth of the GPU network changes. For instance, we can increase bandwidth between two GPUs by connecting them with four NVLinks, or we can connect each GPU with other four GPUs, of course, the more the technology advances, the more links can be made.

NVSwitch NVSwitch is an NVLink switch chip with 18 ports of NVLink per switch [48], in which any port can communicate with any other port at full NVLink speed. This technology is proposed to address all-to-all communication. In fact with only NVLink, point-to-point bandwidth is increased, but some pairs of GPUs could have worse performance due to the GPU network topology that could imply an intermediary GPU hop.

The above-mentioned are some of the hardware technologies that can be involved in the design of multi-GPU systems. There exist some specialized devices using multiple GPUs that exploit those technologies, such as NVIDIA DGX-1 and DGX-2 systems, but also supercomputers like the *Summit* [49] and *SummitDev* supercomputers at Oak Ridge National Laboratory. Finally, there exist also AMD analogous technologies: the new AMD Instinct™ GPUs exploit AMD Infinity Fabric to add GPU-GPU and GPU-CPU communication increased bandwidth, like NVIDIA NVLink does.

6.1.2 Communication middleware

Software support for multi-GPU system is also crucial and in this context the support for MPI is fundamental. Thus, specialized technologies to efficiently communicate between GPUs have been developed, which are completely transparent to the application developer. To have an idea of what this means, let us think about a node with two GPUs that need to exchange data that reside on the device memory. There are thus two processes, each one with a different GPU assigned: in order to exchange data, the first step is to copy the data from GPU to CPU memory, then send the data to the other process and finally copy the data to the receiving GPU. This process is very inefficient because of the many data copies and transfer we make.

A more intelligent approach is therefore developing a technology that can make this process efficient and make the MPI middleware aware of it. For this reason, nowadays most MPI implementation are *GPU-aware*, it is in fact possible to use the MPI operations on buffers that resides on GPU memory transparently. The implementation will in turn decide how to make the communication happen, based on the supported technologies. However, in order to transparently handle host and device addresses, there is the need for some other feature that tells MPI where an address resides without needing a change in the MPI API. This feature was introduced in CUDA 4.0 for NVIDIA devices, and it is called *Unified Virtual Addressing*, with this the host memory and the memory of all GPUs in a node are combined in one virtual address space.

From the NVIDIA side, the above-mentioned acceleration technologies are gathered under the umbrella name of *GPUDirect*. Two in particular are very interesting and can affect performance: the first is GPUDirect P2P which is used in intranode communication, it allows buffers to be directly copied between the memories of two GPUs; the second, GPUDirect Remote Direct Memory Access (RDMA) is for internode communication, with this technology GPU buffers can be directly sent to the InfiniBand network adapter, without staging through host memory.

6.1.3 Challenges

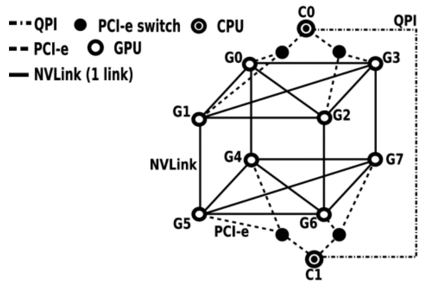
Some challenges arise with the increased complexity of a multi-GPU architecture and are related to the communication, thus strictly coupled with the GPU-GPU, GPU-CPU interconnects, with the internode connection and with the technologies that are used in order to make these connections efficient. The main problem with communication is of course *latency*: the time elapsed between the time an MPI message is generated at its source process and the time the message is delivered at its destination process. Another important factor is that the application have to exploit the available bandwidth of the communication channel, if not, it will have suboptimal performance.

The overall performance of an application can thus be impacted from the communication latency and bandwidth, therefore it is crucial to have a clear idea on how the network topology impacts on them and in which ways these undesirable effects can be alleviated.

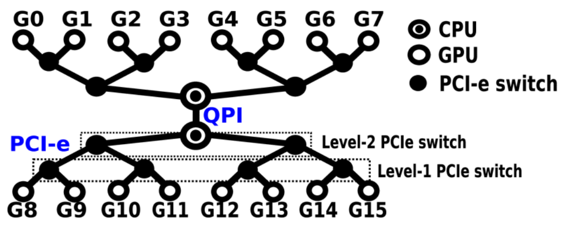
Let us focus on the latter first. The way to make communication "disappear" is to leverage *non-blocking* communication, so that communication can overlap with computation, this is not an easy task on many points. Non-blocking communication makes the whole code more complex: from the sender point of view, the developer must ensure that the data sent is not modified in any way until it is received; from the receiver point of view, it must be checked that the process has actually received the data before moving to a phase of the program when that data can be used. The main difficulty though is to adjust the execution flow of an application to allow the overlapping of communication and computation, this is usually a very intrusive change that requires an analysis of the dependencies between the data that needs to be exchanged, and the data used for the current computation phase, an analysis and optimization of the communication patterns (point-to-point or collective) and an analysis of the phases of computation in order to find the suitable candidates that will hide the latencies.

We will now present some results regarding latency and bandwidth from the work of Li et al. [44], it provides interesting insights on how the interconnects impact on performance of real-world multi-GPU applications. Their work was based on measuring latency and bandwidth of intranode and internode communication, the first by measuring them on the NVIDIA DGX-1 and DGX-2 systems, while the latter on the Summit and SummitDev supercomputers. For both they measured point-to-point and collective communication latency and bandwidth. We will focus on summarizing only the point-to-point communication – which is the main pattern in our solver – on PCIe interconnect, the only one available for our measurements. In order to have a clear idea of the internal topology of the PCIe interconnect on the DGX systems, Figure 6.1a and Figure 6.1b show their organization, which is basically a balanced tree structure, with one level for DGX-1 and two levels for DGX-2.

- For intranode P2P communication, the PCIe latencies for accessing different pairs of GPUs are similar, meaning that the latencies that we have going through a PCIe



(a) Topology of the DGX-1 system, comprehending PCIe and NVLink interconnects.



(b) PCIe two-level tree topology of the DGX-2 system.

Figure 6.1: Different PCIe topologies in DGX-1 and DGX-2 (Li et al. IEEE 2020).

switch (e.g. G0 and G1), across a local CPU (e.g. G0 and G2) and across the QPI bridge (e.g. G0 and G4) as Figure 6.1a shows, are roughly the same. Regarding bandwidth on PCIe, the work shows an interesting behavior labeled as *anti-locality* in which two GPUs sharing the same PCIe switch exhibit lower bandwidth in measurements than the bandwidth measured between GPUs going across the CPU socket or the QPI bridge.

- Regarding internode P2P communication, the measurements show that on Summit-Dev GPUDirect RDMA was not optimal, but this behavior can be explained by the limitations in the chipsets used on the system [44, 50]. On Summit though, GPUDirect RDMA show the best performance among all the configuration used for the measurements, showing the lowest latency and the highest bandwidth.

6.2 Multi-GPU on Gmsh DG

The approach for implementing the multi-GPU in the existing codebase is intuitive: porting the same behavior that we have in the multicore code to the GPU code. We recall that the decomposition of the problem for parallelization is based on partitioning the mesh, and assign each partition to one process. The only phase of computation that is not element-local is the computations of jumps, this involves two adjacent elements, thus when two elements are in different partitions, their respective process need to communicate. Since communication happens in both ways, an inter-process buffer to store the values exchanged is needed, and since we are operating on GPU memory, there are kernels to map data from the jumps buffer to the inter-process buffer and vice-versa. The communication is accomplished using the standard MPI primitives for point-to-point communication, `MPI_Send` and `MPI_Recv`.

Reference architecture Since our implementation needs to run on any kind of system, we kept a simple approach here too. For our application to run, we need a system in which each process has a different GPU device, thus if the application is launched with four processes per node, then four GPUs are needed for each node.

At the moment, our code do not provide any strategy based on locality of processes, thus for example it does not optimize the distribution of the meshes based on the proximity of processes within the topology. This kind of optimizations could be implemented in future releases of the solver, along with or preceded by the use of non-blocking communication mentioned in Section 6.1.3.

6.2.1 Solver scaling

Our scaling analysis has been carried out on a machine from the unit of applied mathematics of the *École Nationale Supérieure des Techniques Avancées*. The machine features two NVIDIA K80 GPUs which have compute capability 3.7. Because of the limited number of GPUs we could only test intranode strong and weak scaling with one and two processes. The next step could be verifying the performance on systems with a more complex topology, with more GPUs (and more recent ones) and obtain the scaling results over different nodes.

```
if (world_rank == 0) {
    MPI_Send(message_device, BUFFER_SIZE, MPI_DOUBLE, 1, tag,
        MPI_COMM_WORLD);
    MPI_Recv(message_device, BUFFER_SIZE, MPI_DOUBLE, 1, tag,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
    MPI_Recv(message_device, BUFFER_SIZE, MPI_DOUBLE, 0, tag,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(message_device, BUFFER_SIZE, MPI_DOUBLE, 0, tag,
        MPI_COMM_WORLD);
}
```

Listing 6.1: This simple communication between GPUs, `message_device` is in fact a GPU memory buffer, fails to execute on LUMI roughly half the time.

Our original plan was actually to test the multi-GPU code on the LUMI supercomputer, in particular targeting the LUMI-G Early Access Platform [51], a partition of the system composed of four nodes, each one with four MI250x GPUs. However, we found out that the only MPI implementation on LUMI that could be used to compile MPI+GPU code was not stable, causing our application to crash. Even a simple “ping-pong” code, sending some data from one GPU to the other, such as the one in Listing 6.1 (of which we only report the core), failed to execute consistently. For this reason our measurements were carried out on the ENSTA machine.

The measurements have been realized by using the same meshes from the weak scaling and testing on both single and two GPUs, thus going from 188 thousand degrees of freedom to 96 millions, we tested approximation order one, two and three on the leapfrog time integration method. The first accomplishment is that the biggest mesh, the 96 millions one can now be given as input to the solver because we have of course the double of available memory. Our analysis measured the average iteration time, the performance in terms of degrees of freedom per second and the communication time.

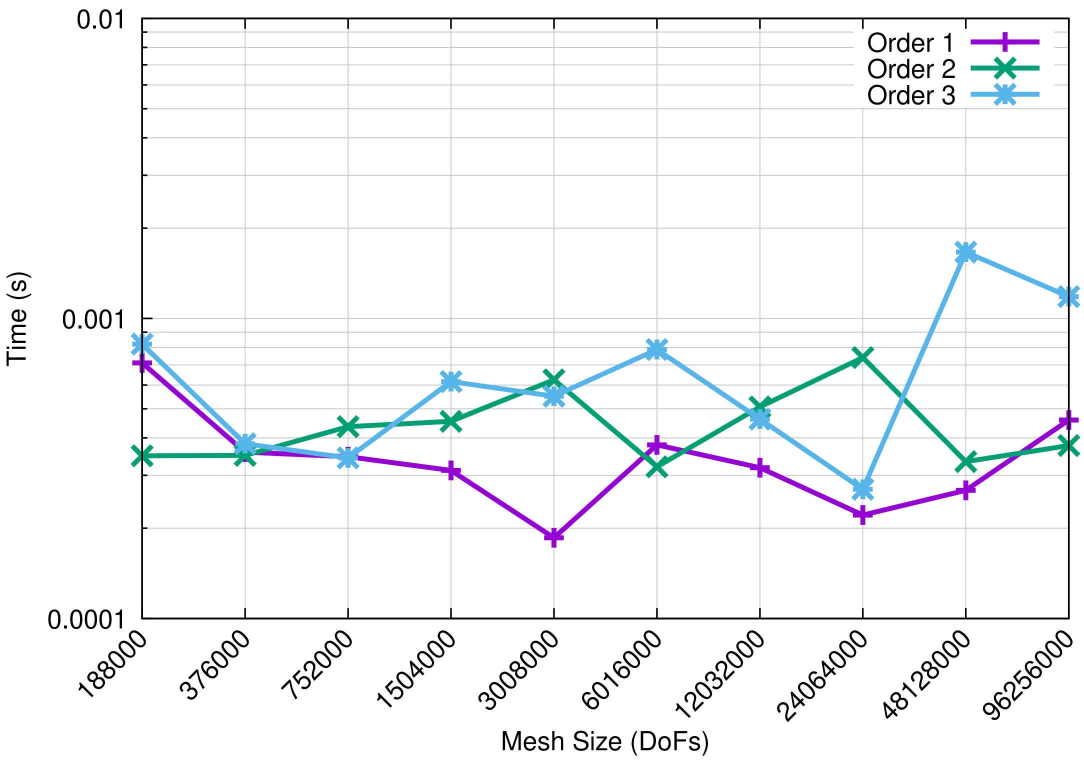


Figure 6.2: Communication times over two GPUs for approximation order one, two and three.

The time plots in Figure 6.3 show the results for both strong and weak scaling. In order to read the strong scaling results, one has to fix the problem size and look at the values of the curves vertically in that point. The ideal iteration time for two GPUs is reported in order to provide a quick way to have an idea of the speedup.

The weak scaling results, on the other hand, can be read by fixing a couple of adjacent mesh sizes: m_1 and m_2. Because we always doubled the size of the mesh, then by reading the results of the single GPU and of the two GPUs execution, respectively for m_1 and for m_2, one can have look at the weak scaling behavior. We recall that for the weak scaling the execution time should be constant when increasing the number of processors, thus one can draw an imaginary horizontal line passing by the iteration time on a single GPU for m_1 and see if the iteration time on two GPUs for m_2 is constant or not and evaluate the scaling.

Performance metrics in terms of degrees of freedom per second can be seen in Figure 6.3, on the right side of the figure. The scaling results in Figure 6.3 show that scaling matches the ideal scaling from the 12 million DoFs mesh onward for all approximation orders, even though a performance improvement is already visible from the mesh with 752 hundreds DoFs onward for order one. This is not true for all approximation orders. In fact, looking at the curves for different approximation orders, we can see that with the increasing order, the implementation needs more degrees of freedom to reach the ideal speedup. This behavior is due to the communication time, that with a few number of degrees of freedom becomes a bottleneck of the application, reducing performance. Figure 6.2 shows communication times, we can see that the behavior changes from one mesh to another, for each approximation order.

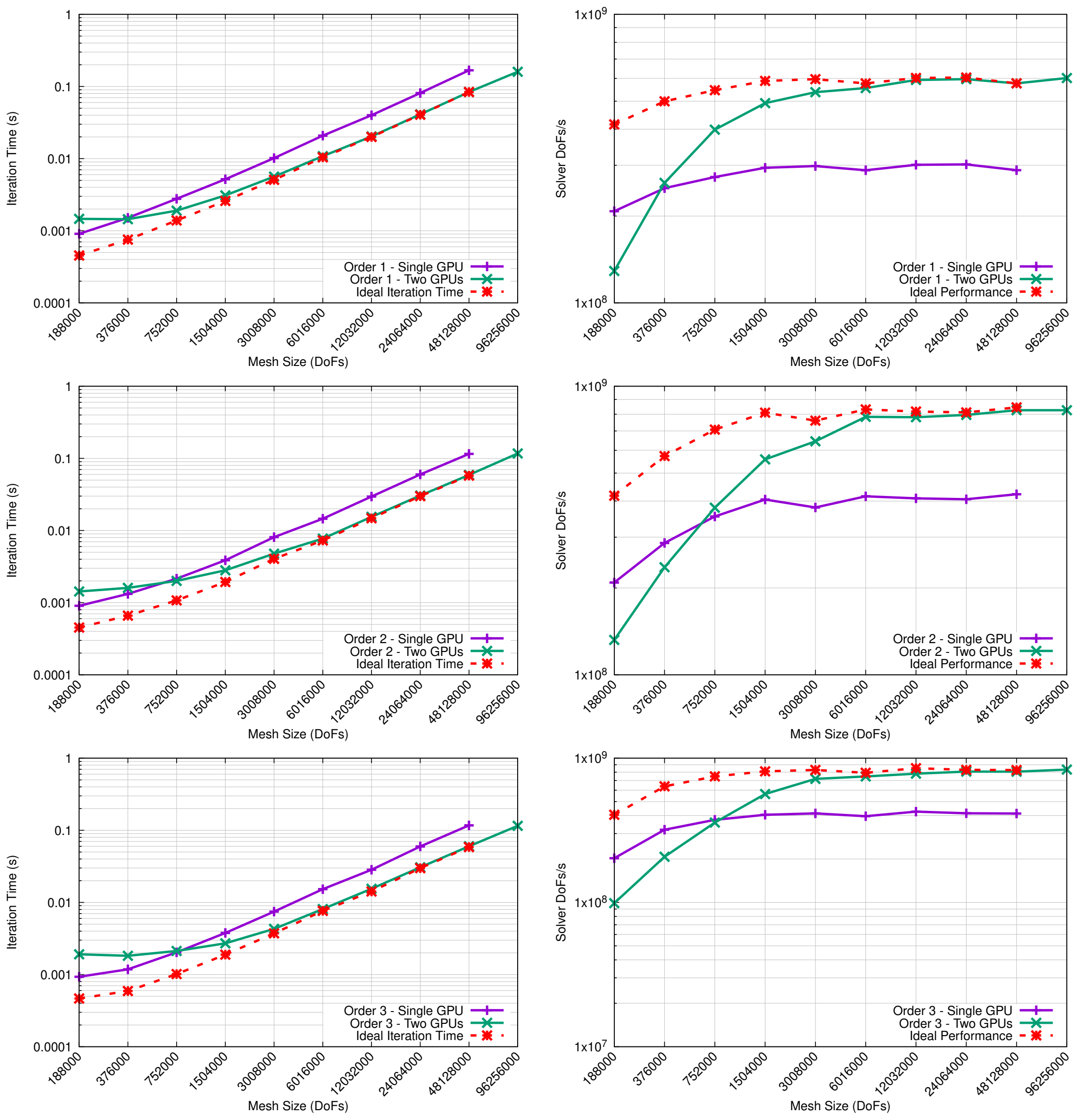


Figure 6.3: Scaling analysis of the solver on one node at approximation order one, two and three using the leapfrog method. The plots show: iteration time (left), degrees of freedom per second (right).

Conclusions

The research question that this thesis wanted to answer was the following "*How well can a reasonably simple DG solver exploit modern hardware?*", and a concise answer to it can be given by Figure 5.20 for multiprocessor systems and by Figure 5.9 for GPUs, while an answer for the multi-GPU systems can be explored in future works. In order to answer to this question we operated on three objectives.

The first objective of the thesis was to optimize the Gmsh DG solver based on its known bottlenecks. Four optimizations were proposed and two were implemented: the optimization of the computation of curls and the fusion of the kernels related to the computation of jumps and to lifting. As the analysis of the kernels showed, the new computation of curls resulted in much improved performance, this is due both to a reduction in memory accesses and increased locality, and to the possibility for the compiler to optimize operations present in the kernel. The proposal regarding the lifting of fluxes could be explored further as a possible future development, particularly on GPUs, where the use of concurrent kernels and the new computation layout could improve performance.

The second objective concerned a thorough performance analysis of the solver. The performance analysis regarding computational kernels provided accurate results through the use of the Intel and NVIDIA profiling tools and allowed to highlight how all kernels, both on CPU and GPU, exploit to the maximum the bandwidth available to the machine and especially for high orders of approximation some kernels show performance very close to the maximum peak achievable by the hardware. From this point of view, in future attention should be paid to those kernels that exhibit low arithmetic intensity, concentrating efforts on trying to increase cache utilization.

Scaling analysis on multiprocessor systems showed excellent results, especially by making the right allocation of processes, scaling perfectly up to the bandwidth saturation of the NUMA domains of the AMD processors used for testing. Weak scaling confirmed the results obtained with strong scaling and allowed us to compare the performance obtained with that relative to single GPU execution, showing that for multicore execution to outperform single GPU execution, about 64 dedicated cores are required. The results obtained also show abnormal execution time behavior in some cases that needs to be investigated further. The codebase could benefit from the use of non-blocking communication, which in some cases might reduce delays and, with some reorganization of the computation stages, might be used to hide communication times by overlapping computation with communication. Finally, it would be interesting to test the performance on Intel processors, especially to understand which could be the optimal execution configuration to achieve optimal performance.

Finally, the code for multi-GPU support has been implemented without much difficulty based on the abstractions provided by the GPU-aware MPI implementation, and the solver has been tested on a machine with two GPUs achieving good performance that becomes optimal as the problem size increases. A next development, once the platform has achieved stability, is to test the multi-GPU implementation on the LUMI supercomputer, so that performance can be verified with newer GPUs and performance can be analyzed by scaling to multiple nodes.

List of Figures

List of Tables

Bibliography

[1] Matteo Cicuttin et al. “Electrostatic discharge simulation using a GPU-accelerated DGTD solver targeting modern graphics processors”. In: *IEEE Transactions on Magnetics* (2022), pp. 1–1. DOI: `10.1109/TMAG.2022.3179309`.

[2] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer, 2007.

[3] Luis Angulo et al. “Discontinuous Galerkin Time Domain Methods in Computational Electrodynamics: State of the Art”. In: *Forum for Electromagnetic Research Methods and Application Technologies (FERMAT)* 10 (Aug. 2015).

[4] Randall J. LeVeque. *Numerical Methods for Conservation Laws*. 2nd ed. Birkhäuser Basel, 1992.

[5] D. A. Di Pietro and A. Ern. *Mathematical Aspects of Discontinuous Galerkin Methods*. Vol. 69. Mathématiques & Applications. Berlin: Springer-Verlag, 2012.

[6] Eleuterio F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. 3rd ed. Springer-Verlag Berlin Heidelberg, 2009.

[7] A. Ern and J.-L. Guermond. *Finite elements III - First-Order and Time-Dependent PDEs*. 1st ed. Springer Cham, 2021.

[8] A. Ern and J.-L. Guermond. *Finite elements I - Approximation and Interpolation*. 1st ed. Springer Cham, 2021.

[9] NVIDIA. *CUDA Toolkit*. Online. URL: `https://docs.nvidia.com/cuda`.

[10] AMD. *AMD ROCm Platform*. Online. URL: `https://rocmdocs.amd.com/en/latest/`.

[11] Khronos Group. *OpenCL*. Online. URL: `https://www.khronos.org/opencl/`.

[12] Khronos Group. *SYCL*. Online. URL: `https://www.khronos.org/sycl/`.

[13] N. Takada et al. “High-speed FDTD simulation algorithm for GPU with compute unified device architecture”. In: *2009 IEEE Antennas and Propagation Society International Symposium*. 2009, pp. 1–4. DOI: `10.1109/APS.2009.takada_fdtd`.

[14] A. Klöckner et al. “Nodal discontinuous Galerkin methods on graphics processors”. In: *Journal of Computational Physics* 228.21 (2009), pp. 7863–7882. ISSN: 0021-9991. DOI: `https://doi.org/10.1016/j.jcp.2009.06.041`. URL: `https://www.sciencedirect.com/science/article/pii/S0021999109003647`.

[15] Nico Gödel et al. "Scalability of Higher-Order Discontinuous Galerkin FEM Computations for Solving Electromagnetic Wave Propagation Problems on GPU Clusters". In: *IEEE Transactions on Magnetics* 46.8 (2010), pp. 3469–3472. DOI: `10.1109/TMAG.2010.2046022`.

[16] Martin Fuhry, Andrew Giuliani, and Lilia Krivodonova. "Discontinuous Galerkin methods on graphics processing units for nonlinear hyperbolic conservation laws". In: *International Journal for Numerical Methods in Fluids* 76.12 (2014), pp. 982–1003. DOI: `https://doi.org/10.1002/fld.3963`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.3963`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.3963`.

[17] A. Modave, A. St-Cyr, and T. Warburton. "GPU performance analysis of a nodal discontinuous Galerkin method for acoustic and elastic models". In: *Computers & Geosciences* 91 (2016), pp. 64–76. ISSN: 0098-3004. DOI: `https://doi.org/10.1016/j.cageo.2016.03.008`. URL: `https://www.sciencedirect.com/science/article/pii/S0098300416300668`.

[18] Christophe Geuzaine and Jean-François Remacle. "A three-dimensional finite element mesh generator with built-in pre-and post-processing facilities". In: *International Journal for Numerical Methods in Engineering* 11 (2020), p. 79.

[19] AMD. *Hipify*. Online. URL: `https://github.com/ROCm-Developer-Tools/HIPIFY`.

[20] T. Mcabee et al. *Silo*. Jan. 2010. DOI: `10.11578/dc.20200930.3`. URL: `https://www.osti.gov//servlets/purl/1231290`.

[21] Pavel Solin, Karel Segeth, and Ivo Dolezel. *Higher-order finite element methods*. Chapman and Hall/CRC, 2003.

[22] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. DOI: `10.1137/S1064827595287997`. URL: `https://doi.org/10.1137/S1064827595287997`.

[23] George Karypis and Vipin Kumar. "METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices". In: (1997).

[24] Open MPI. *Running CUDA-aware Open MPI*. 2022. URL: `https://www.open-mpi.org/faq/?category=runcuda#mpi-cuda-support`.

[25] NVIDIA. *An introduction to cuda-aware MPI*. 2013. URL: `https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/`.

[26] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[27] Diogo Marques et al. "Performance Analysis with Cache-Aware Roofline Model in Intel Advisor". In: 2017, pp. 898–907. DOI: `10.1109/HPCS.2017.150`.

[28] *Nsight Compute Roofline Analysis*. URL: `https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline`.

[29] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (2009), pp. 65–76. ISSN: 0001-0782. DOI: `10.1145/1498765.1498785`. URL: `https://doi.org/10.1145/1498765.1498785`.

[30] Georg Ofenbeck et al. "Applying the roofline model". In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 76–85. DOI: `10.1109/ISPASS.2014.6844463`.

[31] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. "Cache-aware Roofline model: Upgrading the loft". In: *IEEE Computer Architecture Letters* 13.1 (2014), pp. 21–24. DOI: `10.1109/L-CA.2013.6`.

[32] Brian Van Straalen et al. *Empirical Roofline Tool (ERT) v1.1.0, Version v1.1.0*. Feb. 2019. DOI: `10.11578/dc.20210423.2`. URL: `https://www.osti.gov//servlets/purl/1779081`.

[33] Zhe Jia et al. "Dissecting the NVIDIA volta GPU architecture via microbenchmarking". In: *arXiv preprint arXiv:1804.06826* (2018).

[34] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[35] John L Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[36] Sasko Ristov et al. "Superlinear speedup in HPC systems: Why and when?" In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2016, pp. 889–898.

[37] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[38] Samuel Naffziger et al. “2.2 AMD Chiplet Architecture for High-Performance Server and Desktop Products”. In: *2020 IEEE International Solid- State Circuits Conference - (ISSCC).* 2020, pp. 44–45. DOI: 10.1109/ISSCC19947.2020.9063103.

[39] AMD. *High Performance Computing: Tuning guide for AMD EPYC™ 7002 Series Processors.* Online. 2020. URL: https://www.amd.com/system/files/documents/amd-epyc-7002-tg-hpc-56827.pdf.

[40] David Suggs, Mahesh Subramony, and Dan Bouvier. “The AMD “Zen 2” Processor”. In: *IEEE Micro* 40.2 (2020), pp. 45–52. DOI: 10.1109/MM.2020.2974217.

[41] Markus Velten et al. “Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors”. In: *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering.* 2022, pp. 165–175.

[42] Thomas D Cook, Donald Thomas Campbell, and Arles Day. *Quasi-experimentation: Design & analysis issues for field settings.* Vol. 351. Houghton Mifflin Boston, 1979.

[43] Claes Wohlin et al. *Experimentation in software engineering.* Springer Science & Business Media, 2012.

[44] Ang Li et al. “Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect”. In: *IEEE Transactions on Parallel and Distributed Systems* 31.1 (2020), pp. 94–110. DOI: 10.1109/TPDS.2019.2928289.

[45] Dimitrios Ziakas et al. “Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures”. In: *2010 18th IEEE Symposium on High Performance Interconnects.* 2010, pp. 1–6. DOI: 10.1109/HOTI.2010.24.

[46] NVIDIA. “Whitepaper Nvidia ® Nvlink Tm High-speed Interconnect: Application Performance”. In: URL: https://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf.

[47] Denis Foley and John Danskin. “Ultra-performance Pascal GPU and NVLink interconnect”. In: *IEEE Micro* 37.2 (2017), pp. 7–17.

[48] NVIDIA. *NVSwitch: The World’s Highest-Bandwidth On-Node Switch.* Online. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/nvswitch-technical-overview.pdf.

[49] ORNL. *Summit: The next leap in leadership-class computing systems for open science.* Online. 2018. URL: https://www.olcf.ornl.gov/for-users/system-user-guides/summit/.

[50] NVIDIA. *NVIDIA GPUDirect RDMA*. Online. URL: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[51] LUMI User Support Team. *LUMI-G Early Access Platform*. Online. URL: https://docs.lumi-supercomputer.eu/eap/.

I would like to thank Prof. Ferrucci and Prof. Geuzaine
for the support they provided me during my stay abroad
and for the valuable advices I received during the writing of my thesis.
I would also like to express my sincere gratitude to Matteo,
as his guidance was essential during all stages of my work.
A thank you from the bottom of my heart goes to my family,
who have always supported me in my choices and stood by me,
despite my shy, peculiar and sometimes unfriendly character.
A thank you and a kiss goes to all my friends of "La Fasanara",
with whom I have spent the most light-hearted moments of my life,
and whom I hope to always have by my side.
An extra hug, however, goes to Pier Paolo and Vittorio,
for the good times, for their affection and for their support in difficult times.
I want to thank all the members of "Mano Grigia,"
who have now been a constant companion to me for years,
allowing me to blow off some steam and have fun in serenity.
A due hug then goes to all my university friends,
for easing my daily routine of classes and studying,
I especially want to thank two of my fellow travelers:
Lorenzo and Roberto, without you I would be less than half of what I am now.
Finally, thanks to all those who, during my university years,
and in the seven months in Liège, in one way or another have left me something.

Marco

Vorrei ringraziare la Prof.ssa Ferrucci e il Prof. Geuzaine
per il supporto che mi hanno fornito nel mio soggiorno estero
e per i preziosi consigli ricevuti durante la stesura della mia tesi.
Vorrei inoltre esprimere la mia sincera gratitudine a Matteo,
poiché la sua guida è stata essenziale durante tutte le fasi del mio lavoro.
Un grazie dal profondo del mio cuore va alla mia famiglia,
che mi ha sempre supportato nelle mie scelte e che mi è sempre stata accanto,
nonostante il mio carattere timido, particolare e a tratti scostante.
Un grazie e un bacino va a tutti i miei amici della "Fasanara",
con i quali ho passato i momenti più spensierati della mia vita
e che mi auguro di avere sempre al mio fianco.
Un abbraccio in più però va a Pier Paolo e Vittorio,
per i bei momenti, per l'affetto e per il sostegno nei momenti difficili.
Voglio ringraziare tutti i membri della "Mano Grigia",
che sono ormai una costante che da anni mi accompagna,
permettendomi di staccare la spina e divertirmi in serenità.
Un dovuto abbraccio va poi a tutti gli amici universitari,
per avermi allegerito la routine quotidiana di lezioni e studio,
in particolare voglio ringraziare due miei compagni di viaggio:
Lorenzo e Roberto, senza di voi sarei meno della metà di quanto sono ora.
Infine, un grazie a tutti coloro che, durante i miei anni universitari
e nei sette mesi a Liège, in un modo o nell'altro mi hanno lasciato qualcosa.

Marco