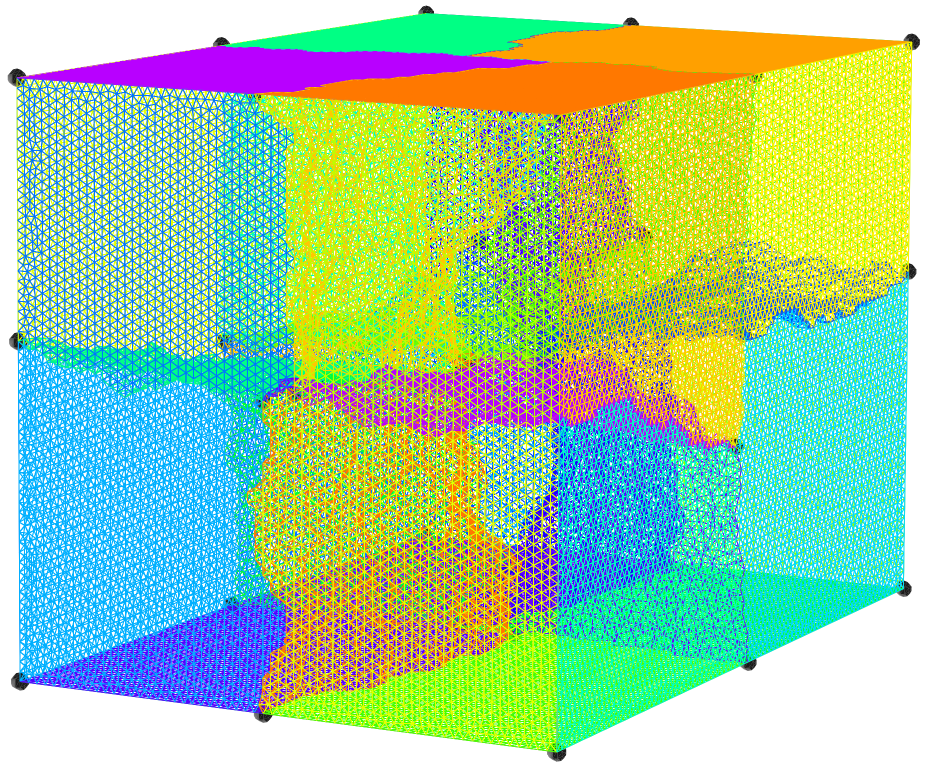
Performance Optimization of a Multi-GPU Discontinuous Galerkin Solver

Université de Liège
Faculté des Sciences Appliquées



Performance Optimization of a Multi-GPU Discontinuous Galerkin Solver

Master's thesis presented to obtain the degree of
Master of Science in Computer Science Engineering

SMAGGHE Clément - s203332

Academic Year 2024-2025

Promotor: Pr. GEUZAINE Christophe
Co-promotor: Pr. CICUTTIN Matteo

Abstract

The Applied and Computational Electromagnetics research group of the University of Liège has previously developed a solver based on the Discontinuous Galerkin (DG) method for solving Maxwell's equations. This master's thesis focuses on its performance analysis and optimization on multi-GPU architectures when solving large-scale problems.

Initial profiling on the Lucia supercomputer revealed that inter-GPU communication, rather than computation, was the primary bottleneck. We addressed this by updating MPI communication from blocking to non-blocking. An extensive set of benchmark tests was conducted on Lucia (NVIDIA GPUs), showing a significant reduced communication overhead and improved scalability — up to 6 times faster execution in some cases. The same tests were then performed on LUMI (AMD GPUs) by using even more computing power (up to 512 GPUs), confirming the robustness of our improvements across architectures, despite a 30% performance penalty due to hardware differences, notably cache size.

A second and final optimization step was carried out in order to mask communication with computation, which needed the application of the DG operation to be restructured. While this step was only benchmarked on Lucia due to time and resource constraints, results showed another considerable performance improvement of up to 11 times faster execution compared to the solver with blocking communication.

Our work only focused on the solver's resolution phase, leaving the initialization phase — currently a major performance and memory bottleneck — largely untouched. Addressing this phase could make it possible to solve even larger problems and support a greater number of GPUs for a given problem in future work.

Acknowledgements

I would like to start by thanking Prof. Geuzaine, his assistant Orian, and Prof. Cicuttin — in no particular order — for all their help and advice throughout this great adventure. Without them, this work would not have been possible. Thank you.

I would also like to deeply thank my parents, who have always listened to me attentively and supported me through thick and thin, as well as for their delicious dishes, which never ceased to fill my belly with motivation. Thank you.

And finally, thank you, reader.

Clément

Table of contents

1 - Introduction

The advent of Graphics Processing Units (GPUs) as a powerful tool for general-purpose computing has deeply transformed High-Performance Computing (HPC), with continuous advancements in programming models facilitating broader adoption. Over the last decade, GPUs have demonstrated their potential to significantly accelerate computations across a wide range of scientific and engineering applications. This evolution is particularly evident in modern supercomputers, where multi-GPU architectures have become increasingly prevalent. In fact, several of the world's most powerful supercomputers leverage GPU acceleration to achieve unprecedented levels of computational performance. However, harnessing the full potential of multi-GPU systems introduces substantial challenges, requiring careful consideration of parallelization strategies, data movement, and resource allocation.

Developing highly efficient multi-GPU applications requires a wide understanding of both the hardware capabilities and the software abstractions provided by modern programming models. A key challenge in optimizing performance is ensuring that computational workloads are effectively distributed across multiple GPUs while minimizing overhead from inter-GPU communication. Developers must take care of these complexities to fully exploit the computational resources available on supercomputers.

This master's thesis focuses on the performance analysis and optimization of a Discontinuous Galerkin (DG) solver designed for execution on multi-GPU architectures. Specifically, the study centers on benchmarking an existing solver to assess its efficiency, identifying bottlenecks, and implementing targeted improvements to enhance its scalability and performance for usages requiring multiple accelerators. The solver under consideration is based on the DG method for solving Maxwell's equations and is developed as part of the Gmsh DG project[1] at the Applied and Computational Electromagnetics research group of the University of Liège. A previous master's thesis was written by Marco D'Antonio on that subject [1]. His work focused on analyzing and optimizing such solver for high-performance computing on modern CPU and GPU architectures. The author proposed and implemented key code optimizations, performed detailed performance and scaling analyses on multi-CPU systems, and extended the solver to support multi-GPU execution with limited testing. The results demonstrated significant performance gains, especially in memory efficiency and kernel-level execution, with insights guiding future improvements and deployment on larger multi-GPU platforms like LUMI. Given the computational intensity of such applications — where trillions of Floating Point Operation per Second (Flops) are performed — optimizing execution on multi-GPU systems is crucial for achieving practical simulation times for large-scale problems.

Thus, the central research question guiding this thesis is:
To what extent can a Discontinuous Galerkin solver efficiently utilize modern multi-GPU architectures?
In order to answer this question, the work was structured in the following stages.

1. **Initial performance benchmarking**
   The first step is to conduct a performance analysis of the existing DG solver, particularly when running on multiple GPUs, in order to get a brief view of what the solver is capable of. This involves executing a series of benchmarks to assess computational efficiency by profiling the solver with official tools provided by NVIDIA. This includes roofline models, kernel execution times, communication latencies and patterns between GPUs.
2. **Identification of bottlenecks and optimization of the solver**
   Based on the results from the previous step, the primary sources of inefficiency can be identified. These insights are critical to determine which aspects of the implementation require optimization. The targeted optimizations will be implemented to improve the solver's efficiency.

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

3. **Performance comparison**
   Once the optimizations have been implemented and validated, a complete new set of benchmark tests must be carried out to quantify the solver's performance improvements. We will use Lucia, a high-performance supercomputer based in Wallonia, which features 50 GPU nodes, each containing 4 NVIDIA A100 GPUs. To extend the study the solver's efficiency, we will also conduct tests on a different architecture: LUMI. LUMI is a supercomputer located in Finland, and one of the fastest in Europe, featuring 2978 GPU nodes, each with 4 AMD MI250X GPUs. This step will be vital as it will allow us to validate results from Lucia as well as to help us predict the solver performance on other supercomputers and architectures.

By following this structured approach, this work aims to provide a deeper understanding of the computational challenges associated with multi-GPU execution of DG solvers and to propose strategies for maximizing the efficiency of such solvers on modern supercomputing architectures.

The first two chapters will introduce the work of this master's thesis, then we will develop the previous three points as separate chapters.

2 - Physical and mathematical context

Before starting the actual work of this master's thesis, which is to analyze and optimize the Gmsh DG solver, it is essential to understand its purpose, applications, mathematical and physical foundations, and computational structure.

In this chapter we briefly recall the mathematical and physical context on which the Gmsh DG solver is based. For a more thorough description, we refer the reader to the master's thesis written by Marco D'Antonio [1], of which this chapter is an extract. As already mentioned in our introduction, the above thesis focused on the study and optimization of the CPU-based part of the solver and, to a lesser extent, on the GPU-based part.

2.1 - Vector calculus

We review 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 $f_x(\cdot)$, $f_y(\cdot)$, $f_z(\cdot)$.

For sufficiently smooth scalar functions $f : \mathbb{R}^3 \to \mathbb{R}$ or sufficiently smooth vector functions $\boldsymbol{f} : \mathbb{R}^3 \to \mathbb{R}^3$ it is possible to compute the quantities ∇f, $\nabla \times \boldsymbol{f}$ and $\nabla \cdot \boldsymbol{f}$, which are the standard *gradient*, *curl* and *divergence* vector calculus operators. We also recall that the following identities hold:

$$\nabla \times (\nabla f) = \boldsymbol{0} \quad \forall f, \tag{1.1}$$

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

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

$$\int_V (\nabla \cdot \boldsymbol{f}) \, \mathrm{d}v = \int_{\partial V} \boldsymbol{f} \cdot \hat{\boldsymbol{n}} \, \mathrm{d}S \tag{2}$$

where the outward unit vector normal to V is denoted by $\hat{\boldsymbol{n}}$. By applying the substitution $\boldsymbol{f} \to \boldsymbol{f}g$, where $g : \mathbb{R}^3 \to \mathbb{R}$, to (2) 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:

$$\int_V \boldsymbol{f} \cdot (\nabla g) \, \mathrm{d}V + \int_V (\nabla \cdot \boldsymbol{f}) g \, \mathrm{d}V = \int_{\partial V} (\boldsymbol{f} \cdot \hat{\boldsymbol{n}}) g \, \mathrm{d}S \tag{3}$$

2.2 - Maxwell's equations

Let $\boldsymbol{x} \in \mathbb{R}^3$ represent 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, in differential form, are written as

$$\nabla \cdot \boldsymbol{d}(\boldsymbol{x}, t) = \rho(\boldsymbol{x}, t) \tag{4.1}$$

$$\nabla \cdot \boldsymbol{b}(\boldsymbol{x}, t) = 0 \tag{4.2}$$

$$\nabla \times \boldsymbol{e}(\boldsymbol{x}, t) = -\frac{\partial \boldsymbol{b}(\boldsymbol{x}, t)}{\partial t} \tag{4.3}$$

$$\nabla \times \boldsymbol{h}(\boldsymbol{x}, t) = \frac{\partial \boldsymbol{d}(\boldsymbol{x}, t)}{\partial t} + \boldsymbol{j}(\boldsymbol{x}, t) + \boldsymbol{j_s}(\boldsymbol{x}, t) \tag{4.4}$$

where

$\boldsymbol{e} : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}^3$ is the *electrical field*, which has units of $\frac{\mathrm{V}}{\mathrm{m}}$ (Volt 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 $\frac{\mathrm{A}}{\mathrm{m}}$ (Ampere per meter),

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

$\boldsymbol{j} : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}^3$ is the *current density* in conductive media,
which has units of $\frac{\mathrm{A}}{\mathrm{m}^3}$ (Ampere per cubic meter),

$\boldsymbol{j_s} : (\mathbb{R}^3 \times \mathbb{R}^+) \to \mathbb{R}^3$ is the *current density* due to the sources, which has the same units as $\boldsymbol{j}$,

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

From now on, the spatial and temporal dependence of all fields will be assumed implicit, and we will therefore omit its indication in the equations.

Maxwell's equations are solved in a domain $\Omega \subset \mathbb{R}^3$ with outward normal $\hat{n}$ 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)*

$\hat{\boldsymbol{n}} \times \boldsymbol{e} = \boldsymbol{0}$, known as *PEC*

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

$\boldsymbol{d} = \varepsilon \boldsymbol{e}$, the *electric constitutive relation* with ε the *electric permittivity*, (5.1)

$\boldsymbol{b} = \mu \boldsymbol{h}$, the *magnetic constitutive relation* with μ the *magnetic permeability*, (5.2)

$\boldsymbol{j} = \sigma \boldsymbol{e}$, the *Ohm's law* with σ the *conductivity*. (5.3)

The quantities ε, μ and σ depend on the considered material and will be referred as *material parameters* later in this section. In the context of this master's thesis however, only materials which are *linear*, *locally homogeneous*, *isotropic* and *nondispersive* will be considered, leading to piecewise constant, scalar material parameters: this setting covers a huge class of practical engineering problems, despite not being fully general.

Generally, ε and μ are decomposed into two parts: first is the absolute parameter for *free space* (ε_0, μ_0), and second is the *relative* parameter (ε_r, μ_r). This gives us $\varepsilon = \varepsilon_0 \varepsilon_r$ and $\mu = \mu_0 \mu_r$. In SI units, the values of the free space parameters are $\varepsilon_0 = 8.8541878128 \times 10^{-12} \frac{\mathrm{F}}{\mathrm{m}}$ and $\mu_0 = 4\pi \times 10^{-7} \frac{\mathrm{H}}{\mathrm{m}}$.

To demonstrate that the electromagnetic field propagates in waves, we consider the source-free setting where $\boldsymbol{j_s} = 0, \rho = 0$ and no conductive media is present, thus $\sigma = 0$. By substituting the constitutive relations (5) into (4.1) and (4.2), then rearranging, one obtains

$$\mu \frac{\partial \boldsymbol{h}(\boldsymbol{x}, t)}{\partial t} = -\nabla \times \boldsymbol{e}(\boldsymbol{x}, t) \tag{6.1}$$

$$\varepsilon \frac{\partial \boldsymbol{e}(\boldsymbol{x}, t)}{\partial t} = \nabla \times \boldsymbol{h}(\boldsymbol{x}, t) \tag{6.2}$$

By taking the divergence of (6.1) using the second identity (1.2), we deduce that (4.1) is automatically satisfied by (6.1). By reminding that we are in a source-free setting, we can also deduce that (4.2) is automatically satisfied by (6.2). As a consequence, we do not need to enforce explicitly (4.1) and (4.2) during the solution process.

Now, by taking the curl of (6.1) and substituting (6.2) in the result, one obtains

$$\nabla \times \mu^{-1}(\nabla \times \boldsymbol{e}) + \varepsilon \frac{\partial^2 \boldsymbol{e}}{\partial t^2} = \mathbf{0} \tag{7}$$

confirming that the electromagnetic field propagates in waves with speed $c = \frac{1}{\sqrt{\mu\varepsilon}}$.

In the context of this thesis we will be interested in solving the *first order formulation* (6) of Maxwell's equations using appropriate numerical methods.

2.3 - The Discontinuous Galerkin method for Maxwell equations

Partial Differential Equations (PDEs) like Maxwell's equations are rarely solvable analytically, 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. Gmsh DG employs the DG method, which is particularly advantageous on massively parallel machines.

As already mentioned, all the details of the numerical methods employed in Gmsh DG are discussed in the master's thesis of Marco D'Antonio [1]; here we propose an extract from that thesis in order to recall the main points. We also refer the reader to [2], [3], [4], [5] for a more in-depth discussion topics related to the DG method.

2.4 - Computational domain and meshes

In this section, some general definitions about meshes will be recalled, see [5, chapter 1] for more details.

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

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

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

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

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 $F = T_1 \cap T_2 \subset \Gamma_{\text{int}}$ is nonempty, and in this case F is called internal face.
- Given an element T, the set $F = T \cap \partial\Omega \subset \Gamma_{\text{bnd}}$ 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
- $F := T^+ \cap T^- \subset \Gamma_{\text{int}}$
- $\hat{\boldsymbol{n}}^+$ and $\hat{\boldsymbol{n}}^-$ normals of T^+ and T^- on F

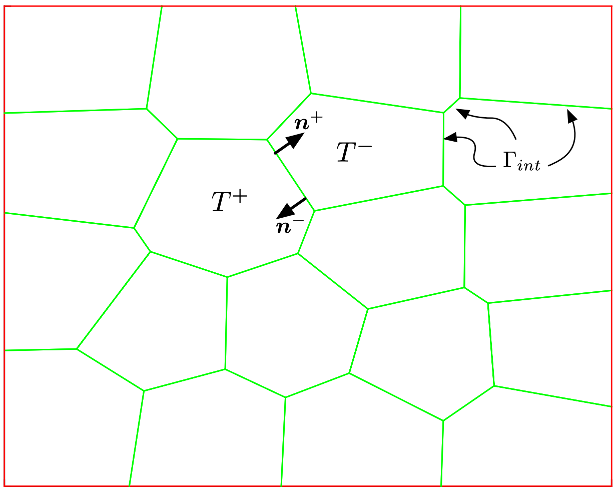


Figure 1 - Visual summary of the symbols introduced to denote mesh elements.

2.5 - Piecewise-polynomial approximation spaces

Let $L^2(\Omega)$ be the standard Lebesgue space of square integrable functions. The discrete approximation space used by DG methods [5, chapter 1], [6] is

$$V_h := \left\{ v \in L^2(\Omega) : v_{|T_i} \in \mathbb{P}_d^k(T_i) \right\}, \qquad i \in \{1, ..., |\mathcal{T}|\}, \tag{8}$$

where $\mathbb{P}_d^k(T_i)$ is a d–variate polynomial of degree k on the element T_i. Intuitively, this definition asserts that the functions of V_h are piecewise polynomial functions that can jump on the interfaces between mesh elements. Clearly, if the DG method needs to be applied to vector-valued problems, V_h has to be adapted accordingly. Note that later in this thesis, we will refer to degree k as the order of approximation.

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

$$p(\boldsymbol{x}) = \sum_{j=1}^{N_d^k} p_j \phi_j(\boldsymbol{x}) = \boldsymbol{p}^T \boldsymbol{\phi}(\boldsymbol{x}), \qquad \boldsymbol{x} \in T \tag{9}$$

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 $\{p_j\}_{j \in \{1,...,N_d^k\}}$ and $\boldsymbol{\phi}(\boldsymbol{x})$ is the column vector $\{\phi_j(\boldsymbol{x})\}_{j \in \{1,...,N_d^k\}}$. 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 $[\boldsymbol{p}_1 \mid ... \mid \boldsymbol{p}_N]$ which is the juxtaposition of all the coefficients of the polynomials attached to the mesh elements. The most straightforward choice for this work is the classical *Lagrange* basis [2].

In order to appropriately describe the DG method, and in particular what happens on the interfaces between the elements of a mesh, we need to introduce a new definition.

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

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

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

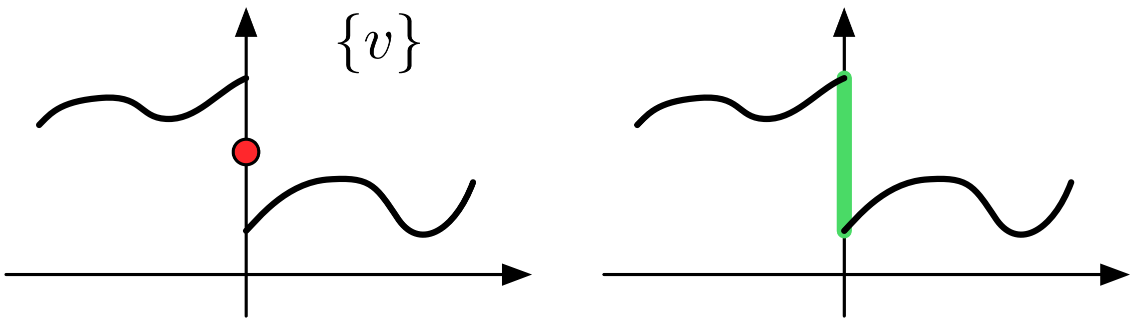


Figure 2 - Graphical representation of the average (left) and jump (right).

2.6 - The Discontinuous Galerkin method for Maxwell equations

In order to apply the DG method, the goal now is to inscribe the Maxwell's equations in the setting of a conservation law of the form

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

where $\boldsymbol{q} \in \mathbb{R}^n, n \in N$ 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 (6) are therefore rewritten as a conservation law by setting

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

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 therefore apply the standard technique of testing the differential equation with a test function $\boldsymbol{v}$ and integrating on Ω, obtaining

$$\mathcal{Q}\frac{\partial}{\partial t}\int_\Omega \boldsymbol{q}\cdot\boldsymbol{v}\,\mathrm{d}V + \int_\Omega (\nabla\cdot\boldsymbol{F})\cdot\boldsymbol{v}\,\mathrm{d}V = \int_\Omega \boldsymbol{g}\cdot\boldsymbol{v}\,\mathrm{d}S.$$

The above equation can be broken in a sum over all elements $T \in \mathcal{T}$ where, on each element T, we apply integration by parts twice and introduce the *numerical flux* [2], [3] $\boldsymbol{F}^*$. Therefore, element-wise:

$$\mathcal{Q}\frac{\partial}{\partial t}\int_T \boldsymbol{q}\cdot\boldsymbol{v}\,\mathrm{d}V + \int_T (\nabla\cdot\boldsymbol{F}(\boldsymbol{q}))\cdot\boldsymbol{v}\,\mathrm{d}V = \int_T \boldsymbol{g}\cdot\boldsymbol{v}\,\mathrm{d}V + \int_{\partial T} (\hat{\boldsymbol{n}}\cdot(\boldsymbol{F}(\boldsymbol{q}) - \boldsymbol{F}^*)\cdot\boldsymbol{v}\,\mathrm{d}S. \tag{10}$$

A rigorous discussion of numerical fluxes is not in the scope of this work, we just recall the existence of *centered fluxes* and *upwind fluxes* [2], [3]. 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

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

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

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

holds and it thus 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, and this has important implications in the implementation of the DG method. Indeed, when using a Lagrange basis, only face-based Degrees of Freedom (DoFs) are involved in the computation of fluxes, and in particular the computation of the jumps is just a subtraction between topologically-adjacent DoFs (Figure 3). This locality can be exploited: because flux computations depend only on neighboring elements through shared faces, the mesh can be partitioned into subdomains with minimal communication overhead. Each subdomain can be processed independently on a separate compute node, with only face-based data exchanged at the interfaces, enabling efficient parallelization.

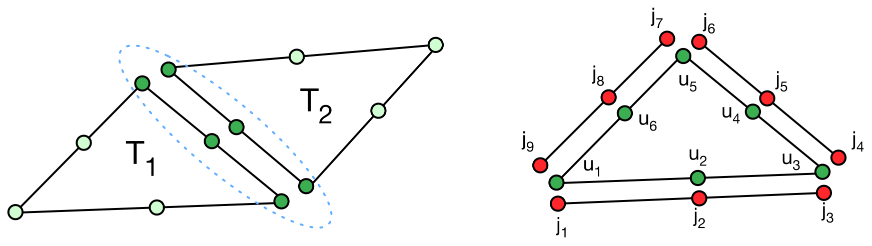


Figure 3 - DoFs involved in the computation of the fluxes between two triangles T_1 and T_2 (left) and association between flux DoFs and element DoFs (right). Both pictures are in the case where Lagrange basis functions are used. While this diagram illustrates the concept in 2D for clarity, the same principles apply in the 3D case, using tetrahedra and faces rather than triangles and edges.

2.7 - Computational properties of DG

By examining (10) it can be noticed that all the volumetric terms (integrals on T) are element-local, therefore in an implementation they can be computed independently and in parallel. The only communication between adjacent elements is due to the numerical flux and takes place only in the computation of boundary integrals. In addition, an explicit discretization of (10) leads to a repeated computation of element-local matrix-vector products [1], exposing another level of parallelism, this time at the level of the single degree of freedom. Taking into account the way in which GPUs work, this enables DG to be extremely efficient on such architectures [1].

The computational structure resulting from a DG discretization of the Maxwell's equations will be as depicted in Figure 4.

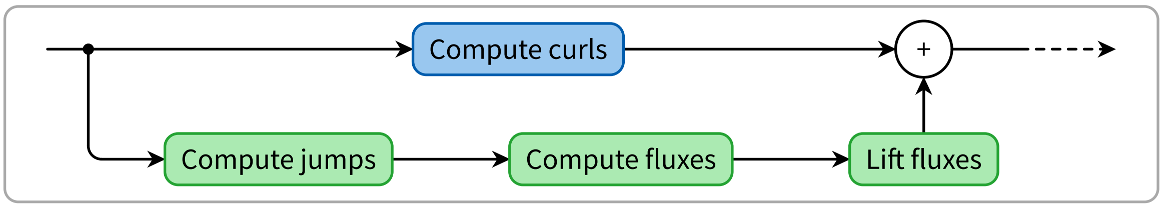


Figure 4 - Computational steps required in order to do one application of the Discontinuous Galerkin operator.

In particular, we recognize a volumetric path and a boundary path. The volumetric path (blue) can be computed element-wise and in parallel, whereas the boundary path (green) requires communication between elements in the computation of jumps. The rest of the boundary path is again computable element-wise in parallel.

2.8 - 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 [3] for the details.

3 - Graphics Processing Units

GPUs were firstly specialized electronics circuits designed to accelerate image and video manipulations. In consequence, they are also called *accelerators*, providing a highly parallel environment compared to standard Central Processing Units (CPUs). But this environment is not exclusive to image processing, and they can be used for general computing: this is called General-Purpose Computing on Graphics Processing Units (GPGPU). Gmsh DG, our solver for the Maxwell's equations, is well-suited for GPUs as it involves high arithmetic intensity with minimal inter-element communication. Each element can be updated independently using local data and only small amounts of data from neighboring faces, making the method highly parallelizable, as we discussed in Section 2.6.

GPUs feature thousands of cores that can be used in parallel, with a Single Instruction Multiple Threads (SIMT) execution model. CPUs on the other hand, while they feature only tens of cores, are equipped with a broader set of instructions and are more dedicated to data caching and control flow.

Developing an application to use GPUs is not as straightforward as it is for the CPUs. But nowadays, several programming models and platforms are available to developers, like OpenMP, CUDA, and HIP, allowing them to ignore the underlying graphical concepts in favor of high-performance programming. They fall into different categories.

- Target-specific.
  The majority of them are designed for a very specific target GPU, generally the brand. This includes CUDA for NVIDIA [7] and ROCm for AMD [8].
- Target-free.
  Some programming models are more generic, but might sometime fail to exploit the target GPU to its fullest due to abstraction. We can find OpenCL [9] and SYCL [10] falling into this category.

Gmsh DG currently supports both CUDA and HIP, allowing it to run efficiently on both AMD and NVIDIA GPUs. This is achieved using C++ preprocessing directives that select the appropriate backend at compile time, ensuring the executable is built with only one Application Programming Interface (API). Since GPU kernels are written in CUDA C++, the *Hipify* tool [11] is used to automatically convert them to support the AMD ROCm platform. These aspects, GPU kernels and the *Hipify* process, will be discussed in more detail later in the thesis.

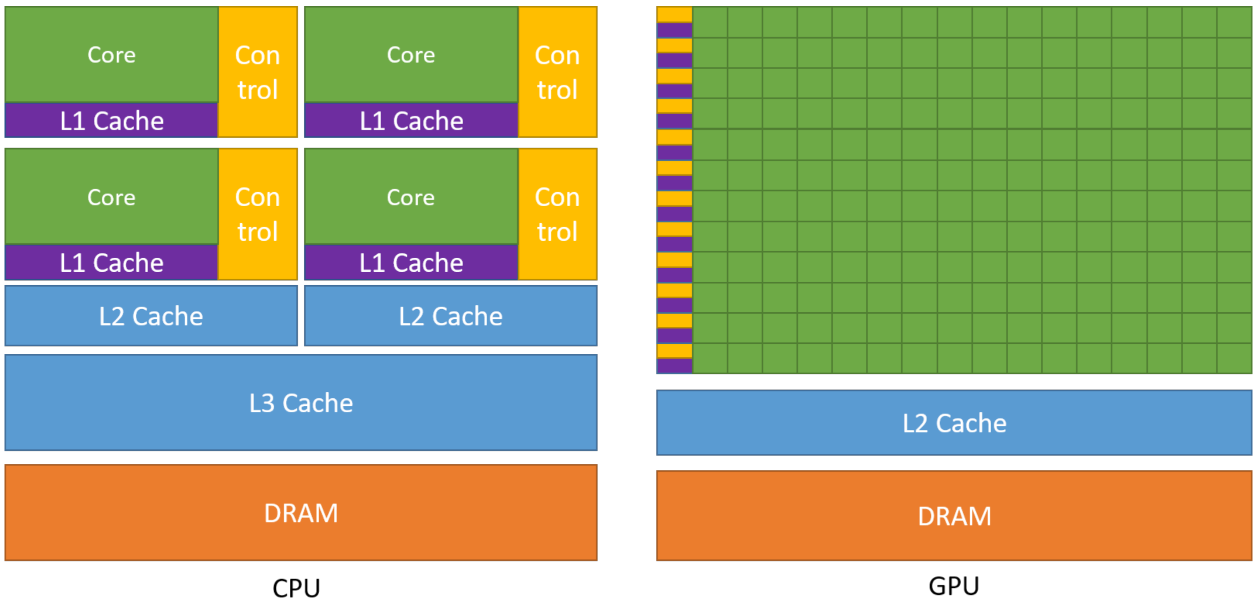


Figure 5 - Comparison of transistors assignment between GPUs and CPUs. The former devote more transistors to data processing (in green) but much fewer for control units (in yellow) and caches (in purple) (NVIDIA Corporation 2022).

3.1 - Architecture fundamentals and terminology

GPU cards are accelerators connected to the CPU via a bus (most often 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

It is worth noting that recent architectures such as AMD's MI300A adopt an Accelerated Processing Unit (APU) design, where the CPU and GPU share the same die, and therefore enable them to share physical memory. This unified memory architecture eliminates the need for explicit data transfers betweenCPU and GPU, potentially simplifying development and improving performance for memory-bound applications. This design, however, is not present in the supercomputers we will use in this master's thesis: Lucia and LUMI (more on them later).

Before going into more details, we should note that the different components of the GPU do *not* have an universal terminology across different programming models. In the context of this thesis however, we will use the CUDA terminology [12].

Each CUDA device (or GPU) is composed of the following elements.
- Several *Streaming Multiprocessors (SMs)*. A SM is an independent processing unit.
- A global memory, residing in DRAM.
- L2 global cache, shared by all SMs. As we could see in Figure 5, GPUs do not have any L3 cache.

Over the years, GPUs have gone from strength to strength. This resulted in changes in hardware features and set of instructions. To differentiate them, CUDA introduced Compute Capability (CC), telling us the SM version used for any NVIDIA GPUs. Knowing the CC used by a card is crucial to fine tune the performance of applications. Even if the CUDA compiler is using that version number to properly optimize them — e.g. by using the best possible set of instructions specific to a version — the developer (us) should be aware of it as it can influence the implementation of algorithms. A CC 8.0 SM is mainly composed of:
- 192 KB of unified data cache, out of which the shared memory is partitioned. The remaining data cache is used as L1 cache;
- 64 CUDA cores FP32 and 32 CUDA cores FP64, each capable of performing single (double) precision arithmetic operations per clock cycle respectively; as well as 64 CUDA cores INT32 for integer math (which is not for our study). The CUDA cores are the one responsible of running the actual computation.

3.2 - CUDA execution model

The developer can define C++ functions, called *kernels*, to be executed in parallel by many threads on the GPU using the CUDA programming model.

A kernel is executed by a *grid*, which can span on all SMs. A grid is composed of a group of one-dimensional, two-dimensional or even three-dimensional thread *blocks* on a single SM. Such blocks are, in their turn, divided into *warps*. A warp consists in a group of 32 consecutive threads, all executing the Single Instruction Multiple Data (SIMD). Each warp has its own instruction address counter and register state, so can branch and execute independently. In a case where the threads in a warp would not execute the same instruction (e.g. because of branching), the *warp scheduler* will execute each branch one after the other while disabling threads that are not taking part in the current path. Finally, a *thread* is the smallest element and extremely lightweight compared to CPU threads. It cannot do any context switch: the resources stay allocated until its execution completion. The above description can be seen in Figure 6.

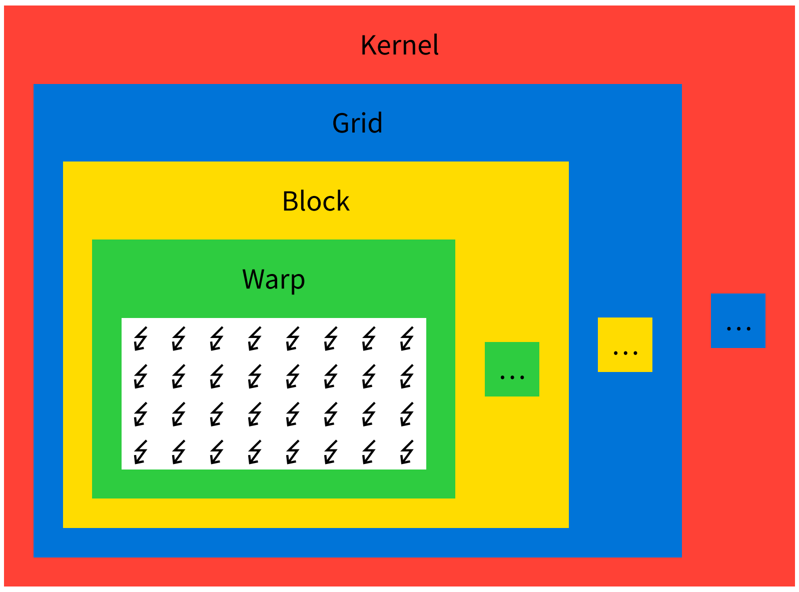


Figure 6 - CUDA thread hierarchy, from kernel to thread. The latter are represented by ⚡, for a total of 32 threads per warp.

The way computations are performed on a GPU is quite different from a CPU. On a CPU, operations on data structures like vectors are typically carried out using loops that process one element at a time (or a few at a time with SIMD instructions). In contrast, a GPU launches many threads simultaneously, and each thread is assigned to handle a specific portion of the data — e.g. one thread per vector element. This massively parallel approach allows GPUs to efficiently process large datasets in parallel.

But the DG method is more complex than a simple vector operation: as already explained in Section 2.3, the method approximates the solution by using polynomial functions attached to mesh elements. The values we need to calculate are the polynomial coefficients, also known as *Degrees of Freedom (DoFs)*. To better work with the SIMT execution model, it is important to distribute the same instruction to the different GPU threads: in our case, each one of them will calculate a single DoFs.

3.3 - CUDA memory model

Before looking further in our use case — the DG method — and how to properly use GPUs, we still have to look at how they deal with memory. Note that *latency* is the amount of clock cycles required for a warp to be ready to execute its next instruction.

Global memory resides in device memory and has a high latency access of hundreds of clock cycles. All threads can access it. It is persistent across kernel launches by the same application. It is the memory where the CPU can send data to and read data from.

Constant memory can be accessed from all kernel as read-only data, and must be set by the host (from the CPU) prior to kernel launch. It also resides in device memory, and is 64 KB, cached in the *constant cache* of 8 KB.

Local memory scope is local the the thread, and it resides in the device memory, thus presenting the same high latency as global memory. The local memory is typically used for large structures, arrays not indexed with constant values, or simply any variable if the kernel uses more registers than available ($\Rightarrow$ *register spilling*).

Shared memory is accessible by all threads in a block and is on-chip, making it more effective than global memory (in terms of latency). The shared memory is divided into words of 32 bits, called *banks*, and is almost as fast as registers as long as there are no bank conflicts. The latter happens when two different addresses of shared memory request fall in the same memory bank. 32-bit banks is a bit inconvenient if threads operates with double floating point numbers, as it requires 2 transactions from shared memory.

Registers are just like CPU registers. They define a private memory for each thread used for local variables in a kernel. They are a limited number of them.

Out of all these different memory, only the registers and shared memory provide the smallest latency and higher bandwidth (all the others are located in the device memory), so the developer should try to limit accesses to others.

Threads should also perform *coalesced memory access*, which is a critical aspect of efficient GPU programming. In simple terms, memory access is considered coalesced when all 32 threads within a warp access aligned and consecutive memory addresses. When this pattern is followed, the GPU can combine these individual accesses into a single memory transaction, significantly reducing memory latency and bandwidth usage. Conversely, uncoalesced accesses — where threads access scattered or misaligned memory locations — lead to multiple memory transactions, which can severely degrade performance. Therefore, organizing data structures and memory access patterns to support coalescing is essential for maximizing throughput on the GPU.

3.4 - Optimal use of GPU

While GPUs advertise a high theoretical throughput (measured in Flops), achieving this peak performance in practice can be challenging. To approach this theoretical maximum, the application must be carefully optimized to maximize computational efficiency. This translates into three key objectives: minimizing latency, maximizing memory throughput, and maximizing arithmetic operation throughput.

3.4.A - Maximize parallel execution

The idea is to reach maximum *occupancy* and utilization. The occupancy is simply a ratio, per SM, of active warps in the SM over the maximum number of active warps supported by the SM. Maximizing the utilization makes it possible to fully hide the latency, and to not “waste” resources (electricity and energy).

Maximum utilization is achieved when all available threads on the GPU are actively used. This typically requires a problem to solve that is embarrassingly parallel — which can be easily divided into many independent subtasks that can be executed in parallel with little or no communication between them. When such conditions are met, one can design an application that effectively exploits the GPU’s massive parallelism.

Note that several kernels can be run concurrently by using *streams*. Streams are queues of commands, such as kernel launches or CUDA API calls, which are executed in sequence. If resources are sufficient, streams can be executed in parallel, otherwise their commands are executed interleaved. Streams also have a priority, allowing certain more important tasks to be executed earlier or faster, by preempting lower-priority work.

But it is not enough to have work allocated to all CUDA threads: it must be meaningful work. Indeed, as we already mentioned before, if a kernel has several execution paths for a single warp, due to branching, then the *warp scheduler* will execute all of them sequentially by turning off threads not taking part in the current execution path. This makes the GPU underutilized, as these threads are not doing anything, but we can’t request them to do another task. It is therefore important to have the exact same instructions inside a same warp.

It should also be noted that the workload must be well-balanced; otherwise, some warps, blocks, or grids may complete their tasks earlier than others. The slower ones become the bottleneck, as the CPU typically waits for all GPU computations to finish before reading the results back from global memory.

3.4.B - Maximize memory throughput

Increasing memory throughput means trying to limit the data transfers with low bandwidth, such as CPU $\leftrightarrow$ GPU or global memory access. In order to do that, the developers must correctly use shared memory and caches. The former is only controlled by the developers (with manual allocation), while the latter is controlled by the system. Just as it is the case for CPU development, they should remain the most local possible in memory in order to have the least amount of cache misses.

As already mentioned earlier, one should access the global memory in a coalescent way, to not require extra transactions. They should also make sure to not avoid bank conflicts when accessing the shared memory, for the same reason.

3.4.C - Maximize operation throughput

The final concept to optimize is the number of operations per second. For this to be maximal, the threads need to always have something to work on. Therefore, the memory latency is completely hidden and a warp should always be ready for each scheduler. Since we want to keep threads busy, any type of halt should be avoided: GPU memory synchronization (or any kind of *barrier*) should be limited, as well as *divergence*. The latter what we referred earlier as branching inside the same warp.

Another simple idea is to prefer instructions offering higher throughput as they will give more results in the same amount of time. Or we can sometimes increase throughput by using less instructions. Any known memory properties should also be specified in the code, such as `const` or `__restrict__`, to help the compiler generate more efficient code (optimizing memory loads, code re-ordering, …). The `__restrict__` keyword can also be used for CPU code. Such a qualifier tells the compiler that the associated pointer does not alias — i.e., no other pointer will access the same memory region. This allows the compiler to apply aggressive optimizations, such as avoiding redundant memory loads. However, it is important to note that the programmer needs to be careful: if aliasing does occur, the compiler will not issue a warning, and the resulting program may behave incorrectly.

Finally, it is also possible to do post-optimizations, by analyzing the code performance with e.g. NVIDIA Nsight. Profiling the application allows the developers to detect which kernel takes the most of the time, the memory throughput, Flops, …; crucial informations that can be used to find bottlenecks.

3.5 - Multi-GPU

In some scenarios, it is desirable to use even more compute power than what a single GPU can deliver. And with the modern architectures or supercomputers, this is now easier than ever.

A CPU can easily use more than one GPU, but the biggest problem is how to split the work among GPUs. In our case, for the DG method, the values computed are the DoFs, including several hundred million variables. An effective way of workload distribution is to separate the volume in (almost) equal parts for each GPU.

Optimizing the use of multiple GPUs is relatively straightforward when the computations assigned to each GPU are completely independent, as no communication is needed. Unfortunately, our case does not fall into this category, as we already discussed in Section 2.6. Because we have chosen to partition the computational domain across multiple GPUs, some mesh faces lie on the interfaces between subdomains and are therefore shared by more than one GPU. This requires the GPUs to exchange data at each time step to ensure consistency across subdomain boundaries.

The first and simplest approach is to rely on the CPU to handle communication of shared interface data between GPUs. However, this method introduces significant overhead due to memory latency: the GPU must first synchronize and transfer its data to the CPU memory, after which the CPU copies the data to the target GPU. This multi-step process can become a major performance bottleneck, especially when performed at every time iteration.

The second, more clever approach takes advantage of modern supercomputing hardware. In the context of CUDA, GPUs can be interconnected using NVLink, a high-bandwidth, low-latency interconnect developed by NVIDIA. NVLink consists not only of physical links, but also includes hardware and software support enabling direct memory access between GPUs, bypassing the CPU entirely. This allows for significantly faster data transfers. When using a CUDA-aware Message Passing Interface (MPI) implementation (such as OpenMPI), developers can perform direct inter-GPU communication using the standard MPI API, and the application will automatically use those NVLinks or other supported interconnects when available [13], [14].

This inter-GPU communication still introduces some latency. Therefore, optimal use of a multi-GPU system requires minimizing the amount of data exchanged between GPUs and overlapping communication with computation whenever possible.

4 - Initial performance benchmarking

Now that the context of our work is well defined and explained, it is now time to benchmark the Gmsh DG solver to try to highlight shadow areas where the solver might be performing inefficiently.

The first step is to find a representative and suitable test case as a basis for our benchmarks. A simple resonant cavity, defined by a rectangular cuboid volume, is an ideal choice. Indeed, its analytical solution is known, which can be compared to the output of our solver in order to estimate its good behavior. Its implementation is also fairly straightforward, and the problem size can be easily scaled by changing the cavity size or the mesh size.

The reason behind wanting to analyze the performance of the solver on multiple GPUs is simple: we want it to be capable of dealing with physically realistic problems with high accuracy over a large range of frequencies. This is particularly relevant for wideband signals — such as pulses — or narrowband signals at high frequencies. Resolving small wavelengths makes it necessary to use fine meshes and high polynomial orders, leading to extremely large problem sizes. In many advanced applications — such as space microwave telescope for electromagnetics, turbofan noise for acoustics or environmental vibrations in the Earth's subsurface for elastodynamics — current solvers are unable to handle the full range of frequencies of technical or scientific interest.

In this context, our primary performance objective is to achieve good weak scaling: we want the solver to take approximately the same execution time as the problem size grows proportionally with the number of GPUs. Strong scaling, on the other hand, is useful to demonstrate how effectively the solver can accelerate the solution of a fixed-size problem by distributing it across an increasing number of GPUs. Strong scaling also makes parallel overhead more visible when the workload per GPU decreases.

The communication between GPUs is handled by the MPI, while the kernels are written in CUDA C++.

For this initial benchmark, it is necessary to cover two aspects.

- The pure calculation part: are we able to reach the advertised Flops from the manufacturer? This should not depend from the communication between MPI ranks at all.
- The data exchange part: is the system network flooded, or is the communication delay abnormally long?

Ideally, the data exchange should be done in the background during calculation to keep the GPUs busy and ready for the next operation(s).

But before actually running tests on the Lucia supercomputer, we should study its infrastructure as some of the solver performance patterns might be a consequence of it.

4.1 - Lucia supercomputer details

Lucia [15], named after Lucia De Brouckère, chemist and professor at the Faculty of Sciences of the Université Libre de Bruxelles, is a supercomputer funded by the Walloon Region and hosted in Charleroi.

Lucia is mainly composed of two partitions: CPU and GPU, with 300 and 50 compute nodes respectively. We will only cover the details of the GPU partition, as we only want to improve the performance of our solver on accelerators. In November 2024, Lucia was ranked 388th in the TOP500 list [16], with a throughput of 2.78 PFlops[2] .

A GPU compute node is composed of the following components [17]:

- one AMD EPYC 7513 32-core CPU @2.6GHz
- 240 GB of DDR4 usable memory @3200MHz
- four NVIDIA A100, each with 40 GB of VRAM, connected with NVLink 3.0
- two Infiniband HDR 200 Gbits/s interconnects

[2] made on a LINPACK benchmark, measuring how fast a computer solves a dense n×n system of linear equations Ax = b

To make the best use of the hardware at our disposal, a more in-depth review of the NVIDIA A100 is needed. One of them theoretically can reach 9.746 TFlops (FP64), using 108 Streaming MultiProcessor (the equivalent of core in CPU). It also features a large L2 cache of 40MB.

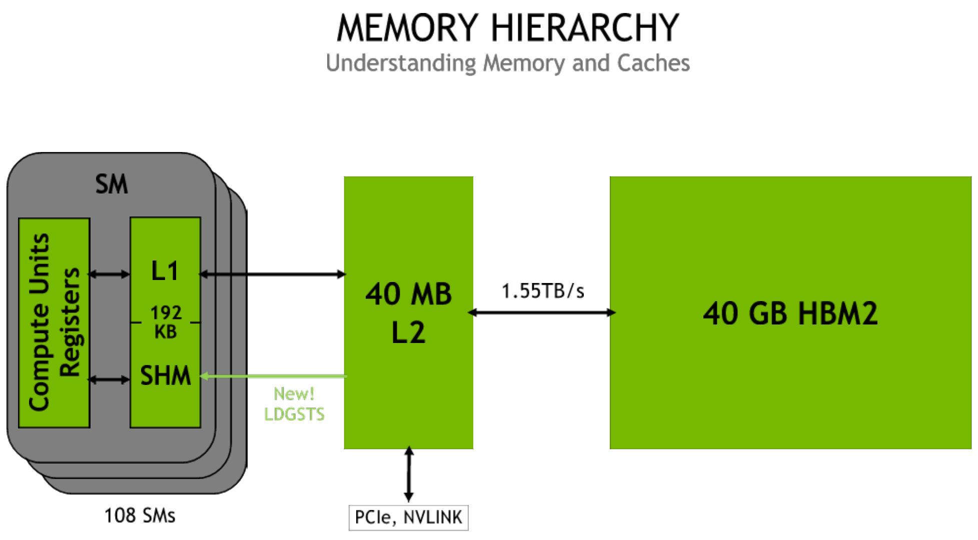


Figure 7 - NVIDIA A100 memory and cache hierarchy [18].

The clusters are managed by the Slurm job scheduler [19].

4.2 - Roofline model

The roofline model [20], [21] gives a great visualization of the performance of GPUs kernels. Indeed, it shows a direct link between performance and hardware limitation, making it easy for us to detect possible issues.

On the x-axis we can find the kernels' *arithmetic intensity*. The latter is corresponds to the number of double-precision floating-point operations (FP64) performed by the kernels divided by the memory traffic they generate in Byte per second. On the other hand, the y-axis is the kernels' performance throughput in GFlops.

Reachable performance are limited by the hardware: computational performance in Flops and bandwidth in Byte/s. Assuming both computation and memory transfers can overlap, the maximum reachable performance is limited by the former or by the latter. GPUs have a theoretical maximum throughput in Flops, which translates into a roof on the right side of the graph where the computation work covers all memory transfers. On the left of the graph, the roof is represented by a slope: the bandwidth of the cache is the cause of the performance decrease [22].

All values, from maximum hardware capabilities to kernel performance, were obtained with NVIDIA Nsight Compute [23], providing precise measurements.

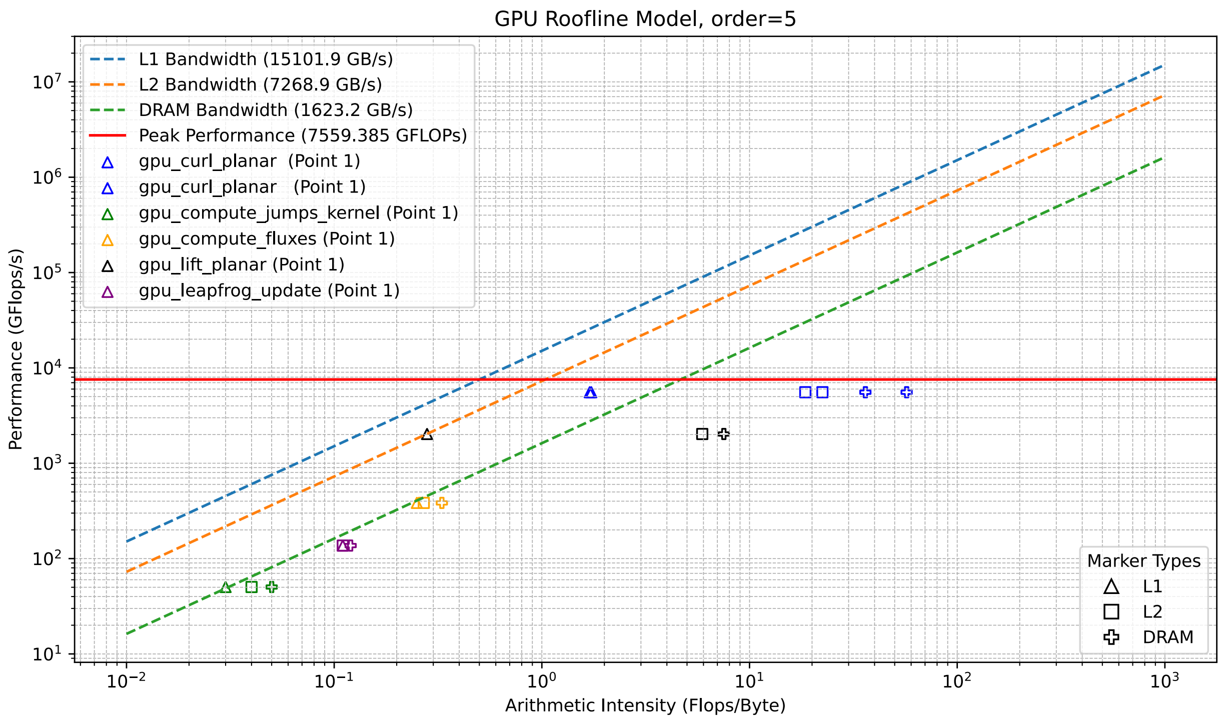


Figure 8 - Roofline model of GPU kernels on Lucia using a single GPU.

As we can see in the roofline model (Figure 8), the performance of the different GPU kernels is fairly close to the *theoretical* maximum. However, it is important to note that this model reflects the throughput of a single GPU in isolation. The roofline model would remain approximately the same even when using multiple GPUs, as it does not account for communication times and delays between GPUs, even if they can significantly impact overall performance.

4.3 - Data exchange

Now that we know that the Gmsh DG solver is great for intra-GPU work, we should highlight its behavior for inter-GPU communication. Fast communication without delays between all GPUs is a key point for any highly parallel workloads [24]. As a matter of fact, it is vital for our solver to present no communication problems for a relatively small number of GPUs. Indeed, if such a problem existed, doubling the amount of resources would have a major impact on execution time due to communication delays: the relationship would not be linear but exponential.

We should study in details the communication pattern of the solver. In the following of this master's thesis, it will facilitate the correct interpretation of results.

As we have seen in the hardware details of Lucia in Section 4.1, NVIDIA A100 GPUs are connected via NVLink 3.0, bringing the intranode bandwidth of up to 600 GBytes/s per direction between GPU pairs [25]. However, when utilizing multiple nodes, internode communication becomes the limiting factor. It can theoretically go up to 50 GBytes/s per direction thanks to the 2 Infiniband HDR-200. We could expect naively that each rank could take a quarter of that bandwidth — 12.5 GBytes/s — if all the 4 GPUs on the node are used. In practive, however, MPI ranks do *not* communicate uniformly with all others. Indeed, since each rank solves the problem over a subdomain of the total volume, communication only occurs across partition interfaces — as discussed in Section 2.6 and Section 3.5. Some ranks may have many neighbors, while others only a few.

To describe this more formally, we introduce the notation N_i to denote the number of neighboring partitions — i.e. MPI ranks — that partition i must exchange data with. The total communication volume and latency for a given rank is closely related to N_i, and it can significantly influence the parallel scalability of the solver. Ideally, the mesh partitioning should minimize both $\max_i N_i$ and the size of the shared boundaries to reduce communication overhead. Our solver already uses the Gmsh library [26] to generate such a mesh. The resonant cavity defined by a rectangular cuboid, can be found partitioned in 8 subdomains thanks to Gmsh in Figure 9. While the volume is a quite common one to deal with, it does not cover all use case — e.g. spheres. In this configuration, the symmetry led to a decomposition into $2 \times 2 \times 2 = 8$ partitions. Each one of them has mainly 3 neighbors, but also share some faces with a fourth to a lesser extent.

More generally, in a 3D structured mesh with many subdomains, a typical interior subdomain — unlike all the partitions from the figure — will have up to *six* main direct neighbors, one along each face, corresponding to the six directions in 3D space (± x, ± y, ± z). Additional connections may exist but involve less communication due to limited interfaces. Note that the actual number of neighbors can vary depending on the partitioning strategy and the location of the subdomain (e.g. on the boundary or inside).

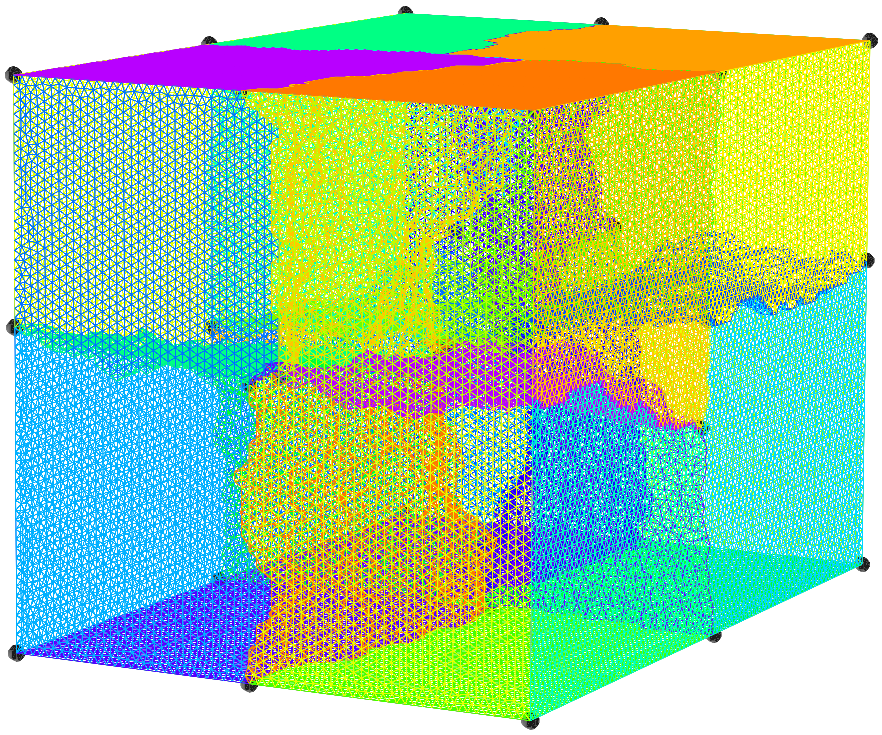


Figure 9 - Partitioning of a rectangular cuboid in 8 MPI ranks. Each color represent a volume belonging to a different rank. Black spheres are used to indicate the separation between partitions.

To deepen our knowledge in this domain, the best to do is to profile the solver and to capture the amounts of data transmitted for each MPI rank pair. At first, the communication time can be ignored. This allows us to not use a dedicated profiler like NVIDIA Nsight, as we can wrap calls to the MPI API [27]. OpenMPI, it its great kindness, gives in fact two names for each function defined in their API: ones with the prefix `MPI_`, others with `PMPI_`. The former is just a symbol for the latter, and their only difference is their names. Using this information, we can implement our own wrappers by declaring and implementing custom `MPI_` functions, in which we will make calls to `PMPI_` functions. This typically intercepts `MPI_` calls from other places in our code, in order to log the amount of data sent, then redirecting the calls to the API using `PMPI_` functions. Luckily, only `MPI_` prefixes are used in the project, but we should be careful to only log the transactions we want to study, that is the ones during the resolution of the problem. There is no need to log received data, as it was sent by another rank. Instead, we should only focus on what is sent.

		MPI receiver rank							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
MPI sender rank	**0**	0	5556	4876	1534	0	134	3902	0
	1	5556	0	96	4999	2120	112	424	3069
	2	4876	96	0	5921	0	4144	610	0
	3	1534	4999	5921	0	3566	319	0	0
	4	0	2120	0	3566	0	7018	32	4773
	5	134	112	4144	319	7018	0	6228	1145
	6	3902	424	610	0	32	6228	0	6416
	7	0	3069	0	0	4773	1145	6416	0

Table 1 - Data sent in kiloBytes (kB) by an MPI rank (on the left) to all other ranks (on the top) during a single iteration for a problem featuring 647,753,400 DoFs. This run used 8 NVIDIA A100 on 2 compute nodes. The black dotted strokes show the separation between nodes: MPI ranks from 0 to 3 are running on the same node, and ranks from 4 to 7 on the other one.

We measured that each rank communicate with mainly 4 other ranks, with the 4th being less important, representing 9% of the total communication payload on average. This correlates perfectly with the partitioning we saw earlier. What we wanted to highlight is the amount of data sent between nodes. Note that the table is symmetric: the same amount of data is sent by both ends in each pair of MPI ranks. In average, 27% of the total exchanged data is internode communication; intranode communication is roughly 3 times as big. The maximum possible bandwidth inside a node can reach up to 600 GBytes/s, while it is only 50 GBytes/s between nodes, giving a ratio of 12 which is far greater than the above ratio of 3. If we used more MPI ranks — thus more partitions — the ratio would be even smaller, since we would have up to 6 main neighbors to communicate with. This confirms that intranode communication is the only one that could have an impact on our solver performance.

To continue to test the Gmsh DG solver under heavy load in order to more easily detect communication issues (if any), we should we use NVIDIA's profiling tools (just as for the roofline model). It is preferable to run the profiling on the solver when it uses a significant amount of MPI ranks (which is the number of GPU). In the case of the Lucia supercomputer, the practical maximum allocation we can make is 16 nodes (note that 50 GPU nodes are available, but we are *not* the only user and even acquiring 16 nodes is quite difficult). This gives us a total of 64 NVIDIA A100, which is good for what we are trying to see.

Profiling, as always, introduces some overhead. This overhead comes from the inserted function calls from the profiler, which are needed to log useful informations such as the time spend by a certain part of the solver, or the different API function calls (e.g. MPI and Compute Unified Device Architecture (CUDA)). To limit the impact of profiling our solver, monitoring a single MPI rank is sufficient. In our testings, profiling in this configuration only increases the total execution time by 1% on average.

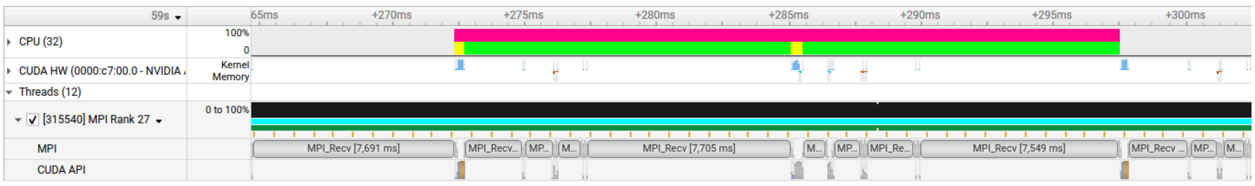


Figure 10 - Profiling from NVIDIA Nsight Systems Systems of the old solver on the 27th MPI rank, for a run using 64 GPUs spread over 16 nodes on Lucia. Colors were manually inserted in the "CPU (32)" line: pink, yellow and green represent the a complete single iteration (with 2 DG operations), computational time and communication time — respectively.

The Figure 10 highlight a single iteration of the solver using leapfrog as the time integration method, which features 2 Discontinuous Galerkin (DG) operations, each mainly composed of four `MPI_` API function calls (either for send or receive operations). We measured that the computational part took 1 ms, while the communication part took 24 ms. In other words, all the GPUs we allocate are only used $\frac{1}{25} = 4\%$ of the time spent to solve the entire problem. This is a considerable waste of time and energy.

Several causes are possible.

1. Slow communication because of a flooded network (full bandwidth used).
2. Slow communication due to blocking sends from one rank to many others, causing delays as receiving ranks must wait before forwarding their own data.
3. Imbalanced workload distribution causing some ranks to lag behind, delaying synchronization points.

We can directly remove the third one, as the solver cleverly and evenly distribute the volume among MPI ranks, providing a balanced workload.

To know whether or not the network is saturated, we need once again to profile our application. This time though, we implemented the logic directly in the code of the solver, in order to track the amount of data exchanged from MPI and the time it took. As a result, the peak bandwidth for outgoing data from one MPI rank is 32 GBytes/s. Note that this is the total outgoing data, meaning that it comprises multiple neighboring ranks. But this is far from the (theoretical) maximum bandwidth of 200 GBytes/s *on a single link*. We can therefore rule out this hypothesis.

The only possible cause that remains is that some MPI ranks are waiting for receiving data, but as they are not the first to be served, they further delay other ranks, aggravating the problem. To find out if this assumption is correct, we need to take a more in-depth look at the actual code of the solver.

5 - Identification of bottlenecks and optimization of the solver

As we have seen in the previous section, we are searching for evidences of bad MPI communication where a rank blocks several others, directly and indirectly.

In the actual state of the solver, all MPI communication is done in a rendez-vous manner: one rank waits the other one it needs to send data to — or receive data from — until they are both ready. This is called *blocking* communication, as ranks wait for their peer in the main thread: no other work/operation is permitted during that time. Again, in the context of this master's thesis, we are only interested in the solving part, not about the initialization. This lets us with only blocking MPI calls that are used to exchange boundaries data between neighboring ranks. As a reminder, the only data exchanged between partitions are face-based DoFs, which are used to compute jumps, then fluxes (Section 2.6).

The current general idea for sending boundaries data is the following:

```
FUNCTION exchange_boundary_data()
    FOR EACH neighbor IN this.neighbors
        IF this.rank < neighbor.rank THEN
            // CALL gpu_jumps_to_ipc():
            COPY internal_data TO send_buffer
            SYNCHRONIZE GPU

            SEND buffer TO neighbor
            RECEIVE buffer FROM neighbor

            // CALL gpu_ipc_to_jumps():
            internal_data -= buffer
            SYNCHRONIZE GPU
        ELSE
            RECEIVE buffer FROM neighbor

            // CALL gpu_jumps_swap_ipc():
            COPY internal_data TO temporary
            internal_data -= buffer
            MOVE temporary TO buffer
            SYNCHRONIZE GPU

            SEND buffer TO neighbor
        END IF
    END FOR
END FUNCTION
```

Listing 1 - Pseudocode representing the global idea behind boundaries data exchanges between neighboring MPI ranks for the solver with blocking communication.

The different `CALL` s in comments are CUDA kernels, but their operations were described in pseudocode. A `SYNCHRONIZE GPU` instruction is also necessary, to wait for the GPUs to finish their work.

In other words, for each pair of neighboring MPI ranks, the one with the smallest rank starts by calling a CUDA kernel while the other one waits patiently to receive data from it. Then, the latter does some data processing from what it received, prepares its buffer and finally sends its boundary data to the lower rank.

The analysis confirms the assumption we made in Section 4.3, as we observe that the current communication strategy uses blocking MPI calls, meaning a rank can easily prevent another one from advancing. We can easily illustrate this with the help of a small example (see Table 2). We should also note that in the current implementation, the list `this.neighbors` is sorted by rank, with the smallest being the first.

0					
🟨 1	2	3	7	8	9

9					
🟥 0	7	11	12	13	14

...

0					
🟩 1	🟩 2	🟩 3	🟩 7	🟩 8	🟨 9

9					
🟨 0	7	11	12	13	14

0					
🟩 1	🟩 2	🟩 3	🟩 7	🟩 8	🟩 9

9					
🟩 0	🟨 7	11	12	13	14

...

Table 2 - Example showing MPI ranks blocking others. Number on top is the rank that needs to exchange data — send and receive — with all ranks on the bottom. Green, yellow or red cells means the exchange is done, in progress or blocked (waiting for the other rank to be ready), respectively.

Through this example, the blocking can easily be seen. Here, the rank 9 was the last one to exchange data with rank 0, but since the former tries to make a rendez-vous directly with the latter, it simply cannot do anything else than wait its turn.

A possible fix to this problem would be to rearrange the list of `this.neighbors`, so it is not sorted by rank anymore. If built correctly, this would avoid ranks blocking others. It is possible to build such list, and it can be computed at the initialization part. Sorting based on the difference between `this.rank` and `neighbor.rank`, with the smallest different first, is a good start.

Unfortunately, the above solution would not be the best, as ranks are still blocked in some cases due to the communication pattern. For example, in the previous pseudocode, one can see that the smaller rank has no delay between its `SEND` and `RECEIVE` instructions. But the greater rank has work to do with its buffer between the 2 calls, making the smaller rank wait. This blocking, albeit much shorter, is still present and introduces some delays in the future communication with other ranks.

One possibility to fully solve this is to "simply" use non-blocking MPI communication. Non-blocking communication means that it is done in the *background*, and the main program continues its execution. That way, a rank can now post several `SEND` and `RECEIVE` *requests* at the same time.

As for the implementation, it now needs two buffers per neighbor: one for receiving, one for sending. Indeed, they need to be separated since we don't know if the data is sent or received first. or if they are done simultaneously, as we don't have control anymore on these requests (except checking if they are done). All buffers can be allocated during the initialization part, thus the buffers used to receive data from the neighbors are already ready to use. On the other hand, buffers for sending data to neighbors need to be filled appropriately before posting a `SEND` request.

With the intention of starting the greatest possible amount of transactions as soon as possible to not delay communication, it is clear that `RECEIVE` requests should be done first. Whenever a request completes, we have update the internal state using the content from buffers. GPU kernels also need to be adjusted to accommodate the buffer changes.

The newly implemented version can therefore be seen as the following pseudocode:

```
FUNCTION exchange_boundary_data_nonblocking()
    // Phase 1: Post all receives immediately
    FOR EACH neighbor IN this.neighbors
        POST NON-BLOCKING RECEIVE receive_buffer_for(neighbor) FROM neighbor
    END FOR

    // Phase 2: Prepare and send data
    FOR EACH neighbor IN this.neighbors
        // CALL gpu_jumps_to_ipc():
        COPY internal_data TO send_buffer_for(neighbor)
        SYNCHRONIZE GPU

        POST NON-BLOCKING SEND send_buffer_for(neighbor) TO neighbor
    END FOR

    // Phase 3: Process completed receives as they arrive
    WHILE there are pending receive operations
        WAIT FOR ANY receive operation to complete
        neighbor = identify_neighbor_from_completed_operation()
        // CALL gpu_ipc_to_jumps():
        internal_data -= receive_buffer_for(neighbor)
    END WHILE
    SYNCHRONIZE GPU
    WAIT FOR ALL pending send operations
END FUNCTION
```

Listing 2 - Pseudocode representing the global idea behind boundaries data exchanges between neighboring MPI ranks for the new solver with non-blocking communication.

The pseudocode for the new solver using non-blocking communication consists of 3 phases: `RECEIVE` requests, `SEND` requests, and finally a synchronization phase where the solver waits for the completion of these requests to read boundary data sent by neighbors. Note that we wait for any `RECEIVE` transaction to end, one at a time. This allows us to update the `internal_data` while some transactions are still running, effectively lowering the time taken by the communication part of the solver.

Now that the new algorithm is implemented, the solver performance should be completely reevaluated. This is done in the next section.

6 - Performance and scalability analysis

In order to study the solver's scalability, as well as to quantify the improvements from the optimizations of the previous chapter, several benchmarks were needed.

In this section, we will continue to use the test case from Section 4, the resonant cavity defined by a rectangular cuboid volume. Weak scaling analysis requires different total problem sizes: each MPI ranks should have the same amount of DoFs to work with. To change the size of the problem, we refer back to Equation (8) in Section 2.5, where we discussed the use of polynomial functions of degree k to discretely approximate the solution space in the DG method. This degree, also known as the approximation order, can be increased or decreased to control the number of Degrees of Freedom (DoFs) per element. A higher k leads to more DoFs and greater accuracy, but also increases computational and memory requirements. Gmsh DG only supports $k \in [1, 6]$.

Other parameters can also be adjusted: the mesh size, and the volume size. Modifying the mesh size requires us to also update the time step accordingly, in order to maintain the stability of the numerical scheme. Because of this, we preferred not to touch it (as adjusting others is easier). We want to test the solver on a "per iteration" basis (just like the previous profiling in Figure 10), to see if the communications can be completed in the background during the computational work, or to see whether the communication between MPI ranks takes the majority of the time or not.

For the above reasons, we decided to only tweak the volume size and approximation order. The size of the problem should be big enough to represent real-life applications, but it is worth recalling that we are also limited by the amount of memory. We therefore need to find a "just-in-the-middle" case. Strong and weak scaling analyses are a first-choice solution: they are perfect to demonstrate how our solver scale (with respect to the size of the problem and the number of MPI ranks), but also allows us to easily compare the performance between the old blocking communication solver to the non-blocking one.

As a reminder, a strong scaling analysis [28] consists in running the executable with increasing amount of processors (in our case, GPUs and MPI ranks) with the same total problem for each of these runs. Ideally, for n total processes, T_i the execution time of the process i and T_{single} the one of the run using a single process, we have:

$$\sum_{i=1}^{n} T_i = T_{\text{single}}$$

On the other hand, a weak scaling analysis [29] requires us to modify the total problem size in order to get the same problem size per process. A perfectly scalable application would then take the exact same execution time for all runs. In other words:

$$\forall i \in [1, n], \quad T_i = T_{\text{single}}$$

To better evaluate the scalability of our solver for real-life problems, we chose to make the scaling benchmarks for each of the 6 orders, as some use cases require high precision results, while others might not. In consequence, the only parameter that can be tweaked for the weak scaling analysis is the problem volume size. This is easier to predict the number of DoFs, but also brings far better precision: if we double the number of GPUs, we want the number of DoFs to double too, so each GPU gets the same amount of computational work. Doubling the volume of the problem seems to be the easiest and most straightforward solution, but the number of DoFs is not exactly doubled and is in fact a bit less than that. Let's show this with a small example: let's define a 2x2x2 cube with 27 DoFs (each dimension has 3 of them). Doubling only one dimension doubles its volume, e.g. 4x2x2, and we now have 5x3x3=45 DoFs, which is less than 27x2=54. DoFs of the new problem are doubled *minus the DoFs of one face of the original cube*. The same logic can be applied to cubes (multiplying each dimension by $\sqrt[3]{2}$). We still doubled the volume size for twice the number of GPUs for weak scaling analysis, but therefore a single

GPU does not have the exact same computational work from run to run. Therefore, we should not look at the execution or iteration time but rather the rate at which we solve the problem: “DoFs/s/GPU”.

To determine the size of the problem we should run the scaling benchmarks on, we made several attempts at the order 6, the highest, therefore the one using the largest amount of memory. We ended up reaching approximately 650 millions DoFs (that we used for the strong scaling analysis), but we managed to go even higher thanks to the memory taken from *multiple* nodes (for the weak scaling analysis, as the amount of memory is too low on single node for that much DoFs). To have a perfect and fair distribution of the volume across all MPI ranks, it has to be a cube.

At first, the problem volume size was the same for the strong scaling benchmarks, and the same for the “starting point” of weak scaling ones. Unfortunately, we only found later that we should have tuned the volume size for each order so that the number of DoFs would be roughly the same between the different benchmarks. This would have allowed us to better compare the performance and scalability between approximation orders. Lacking time, we managed to revise the weak scaling benchmarks, but not the strong ones.

To test our solver, two different process bindings are possible. The first one is the “close” mapping policy: it uses GPUs that are physically close together by filling up one compute node before moving to others, maximizing locality and reducing communication latency. The second one is the “spread” mapping policy: as its name suggests, it distributes GPUs allocation as widely as possible — so simply *one* GPU per node — to provide a more balanced load.

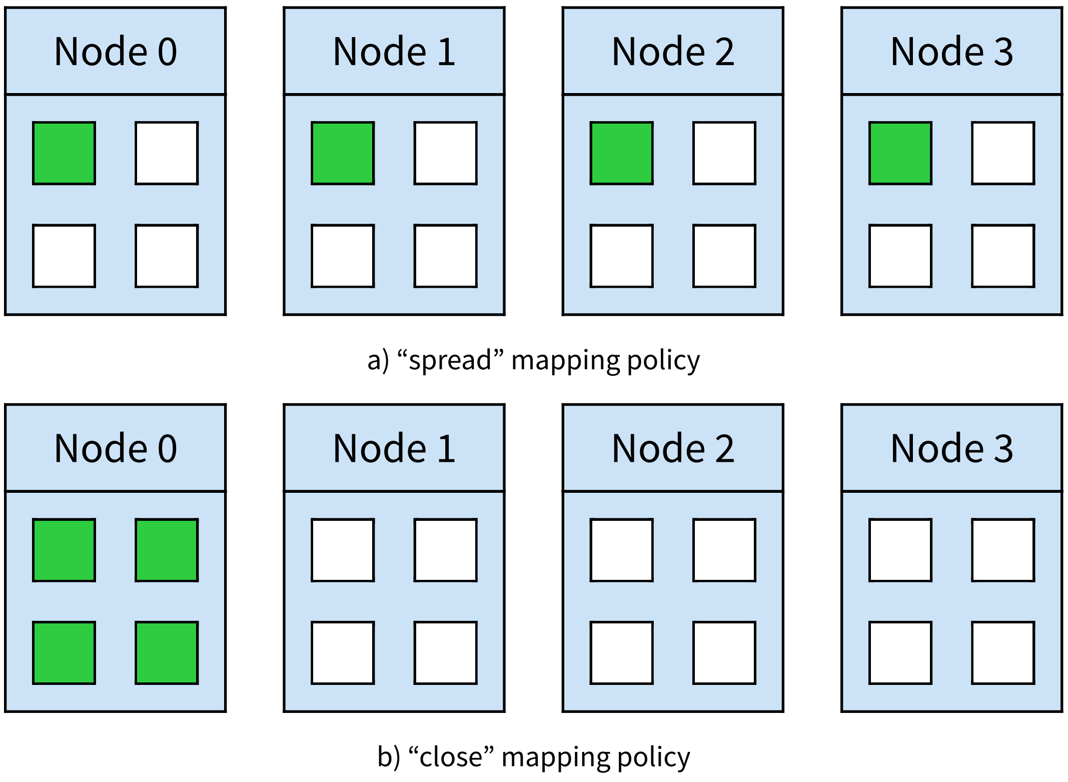


Figure 11 - Assignment of 4 GPUs in compute nodes with two different mapping policies: “spread” and “close”. The boxes in green represent the assigned GPUs.

All scaling benchmarks were made with the six orders of approximation, and using both of “spread” and “close” mapping policies. The strong scaling benchmarks were pushed even further than these two mapping policies, as we tested the solver with all possible powers of 2 for the number of GPUs *per node*. The compute nodes allocation was *exclusive* to us, meaning that we requested the nodes for ourself only, without any other program interfering with our solver timings.

The different plots for the scalability benchmarks can be found at the end of each subsection.

6.1 - Pre- and post-optimization comparison

Before looking at the strong and weak scaling graphs, it is important to validate the good behavior of the new solver. Since the main change was communication part, we should confront the new solver to what was discussed in Section 4.3.

More precisely, let's profile the solver with NVIDIA Nsight Systems again, as shown in Figure 12. Just like previously, the computational part takes a total of 1 ms. However, the communication part now takes 3.65 ms, which translates into a total GPU use of $\frac{1}{4.65} \approx 21\%$ over the whole time spent to solve the entire problem. This is a considerable improvement, as we were at 4% with the blocking communication, but 79% of the GPU time is still spent idle, waiting on communication rather than performing computations, representing a significant underutilization of computational resources.

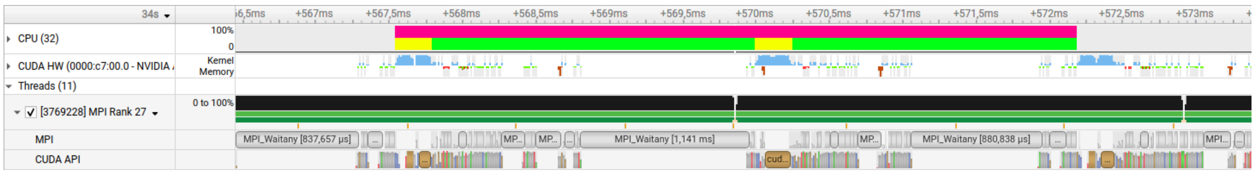


Figure 12 - Profiling from NVIDIA Nsight Systems Systems of the non-blocking communication solver on the 27th MPI rank, for a run using 64 GPUs spread over 16 nodes on Lucia. Colors were manually inserted in the "CPU (32)" line: pink, yellow and green represent the a complete single iteration (with 2 DG operations), computational time and communication time — respectively.

With this simple test, we can already attest to the superiority of the new solver. But this only cover a specific unique configuration: it is needed to analyze the new implementation performance on a wide variety of settings and tests, and that is where the strong and weak scaling analyses come in.

6.1.A - Strong scaling

The first thing that stands out, without even comparing the two versions of the solver, is that a lot of MPI ranks (i.e. 32+) — which is equal to the number of GPUs — should only be used for huge problems, or we lose efficiency significantly. Indeed, for a fixed number of MPI ranks, increasing the DoFs brings us closer to the ideal speedup. This is due to the communication latencies introduced by MPI. When the problem is quite small, each rank spends more time to communicate to its "neighbors" rather than actually computing. Therefore, increasing the number of ranks for the same problem size worsen the effect, as the system now has even more communication but less computation to do: this explains the sudden exponential grow in iteration times after a certain number of ranks (that depends on the order). On the other hand, for bigger problems, the GPUs are most of the time computing (instead of waiting and transferring data), and the communication delays start to be negligible. We can therefore expect somewhat good results from the weak scaling analysis.

We should also notice a few outliers in these graphs, for example the old solver when using 16 GPUs, one per node, at the order 5. All these outliers, which don't follow the general curve very well, have been manually tested afterwards to validate them. But almost all of them disappear with the non-blocking communication solver.

Unfortunately, our test case was requiring too much memory using a single MPI rank at the order 6, and the related graph's axes are therefore adjusted and are different than the other ones.

If we now compare the original solver scalability with the optimized one, we can easily find out that the latter gives impressive results and up to 8 times faster iteration (for the order 3 with 16 nodes and 2 GPUs per node), but on average between 2 and 4 times faster for all the orders with 16 GPUs and more, while the speedup is almost negligible for 4 GPUs and less.

We also get a performance boost with smaller orders (thus less degrees of freedoms) when using a modest amount of GPUs. We can see that by looking at the graphs with 8 GPUs: we come from ~1.8 times faster at order 1 (~30M DoFs), to only ~1.1 at order 6 (~648M DoFs).

These findings are linked to the proximity of the ideal speed (see the “Ideal run” green line in the graphs). While we can see that there is room for improvements for lower orders (from 1 to 3), our new solver is really close to being ideal with up to 16 GPUs, and relatively close for 32 GPUs. On the other hand, with 64 GPUs, the iteration time increases significantly, even with consequent problem size.

We are also quite happy to see that in any situation, our new version of the solver is always faster than the old one. Changing the blocking communication to non-blocking could only have one “issue”: when using a lot of GPUs (e.g. 64), they all might flood the system at the same time with their respective calls to the MPI library saying they are ready to receive. The system, compared to the blocking version, has more work to do in a tight interval to map all non-blocking sends to non-blocking receives. Dealing with all these requests at the same time introduces some delay, but is negligible from the delays due to the blocking communication of the old version of the solver.

Overall, spreading the use of GPUs across compute nodes does not necessarily boosts efficiency (not even slightly).

All the constatations were done on the iteration time, throughput (in DoFs/s), and efficiency, which are all closely linked together. Since our work in Section 5 consisted in a significant change of MPI communication, more detailed results would be welcome. As a consequence, we performed another profiling with NVIDIA Nsight Systems. This time however, the goal is to show the repartition of calculation and communication. To obtain brief information, as we already have a good amount with the graphs, we choose to only conduct the profiling at the order 5. This choice is quite simple: the best example to show the communication/calculation ratio is to use the largest problem we used in these strong scaling analyses. Unfortunately, Lucia did not have enough memory to deal with the problem at an order of 6 when using a single GPU. The biggest problem remaining was thus the one at order 5.

Number of GPUs	1	2	4	8	16	32	64
Iteration time (ms)	136.8	68.7	35	18.8	10.6	7.12	9.28
Communication time (ms)	/	0.39	0.63	1.66	2.06	2.77	7.02
Calculation time (ms)	136.8	68.31	34.37	17.14	8.54	4.35	2.26

Table 3 - Communication and calculation times from NVIDIA Nsight Systems profiling given a number of MPI ranks, for a single iteration of the non-blocking communication solver at the order 5 and 431,835,600 DoFs, using the “close” mapping policy with the same problem for all runs on Lucia. Equivalent to the strong scaling analysis in Figure 15, but slightly different timings due to the profiler overhead and measurements imprecision.

While the calculation time scales very well, as it is roughly 2 times smaller when the number of GPUs is doubled, the communication time grows with the number of GPUs and drastically increases at 64. This shows and confirms that the heavy-computational steps from Figure 4 have a really limited room for performance enhancement, as we saw in Figure 8.

Meanwhile, the ratio communication over calculation keeps increasing. The explanation should be obvious: using more GPUs will divide the global problem into smaller pieces, therefore making calculation time smaller too. As the communication time does not increase anymore (after a certain point), it is logical for the ratio to increase continuously.

To put the above timings into perspective, the strong analysis is put in table form in Table 4.

	Number of GPUs	1	2	4	8	16	32	64
Blocking communication	Iteration time (ms)	136.8	69.1	35.9	22	14.8	15.1	40.8
	Average GDoFs	3.15	6.25	12.03	19.56	29.15	28.54	10.58
	Efficiency (%)	100	98.7	95.3	77.8	57.9	27.4	5.8
Non-blocking communication	Iteration time (ms)	136.8	68.8	34.74	18.75	10.58	7.05	9.36
	Average GDoFs	3.16	6.28	12.43	23.03	40.82	61.25	46.16
	Efficiency (%)	100	99.3	98.2	92.1	80.8	60.6	22.4

Table 4 - Strong scaling analysis of the solver on Lucia for a comparison between blocking and non-blocking communication at order 5. This is a transcript of Figure 15 in table form. For each version of the solver, the table show respectively: iteration time (in milliseconds), average degrees of freedom per second - DoFs - (in GigaDoFs) and efficiency (in %), given the number of GPUs with a "closed" mapping policy.

Graphs have a strong advantage of being easily interpreted and present things nicely, while tables offer a more precise view of the same results but with certainly less visual appeal and intuitive clarity. We now have the choice to compare the two versions of the solver: graph and table. While the former features more different run settings (orders, numbers of MPI ranks, …), the above table confirms our interpretation of the graphs: the non-blocking implementation is better in all cases, reaching between 2~4 times as much as throughput (DoFs) when using the most resource (32+ GPUs).

Overall, this non-blocking MPI communication solver performers better than with blocking communication, but still performs poorly at strong scaling.

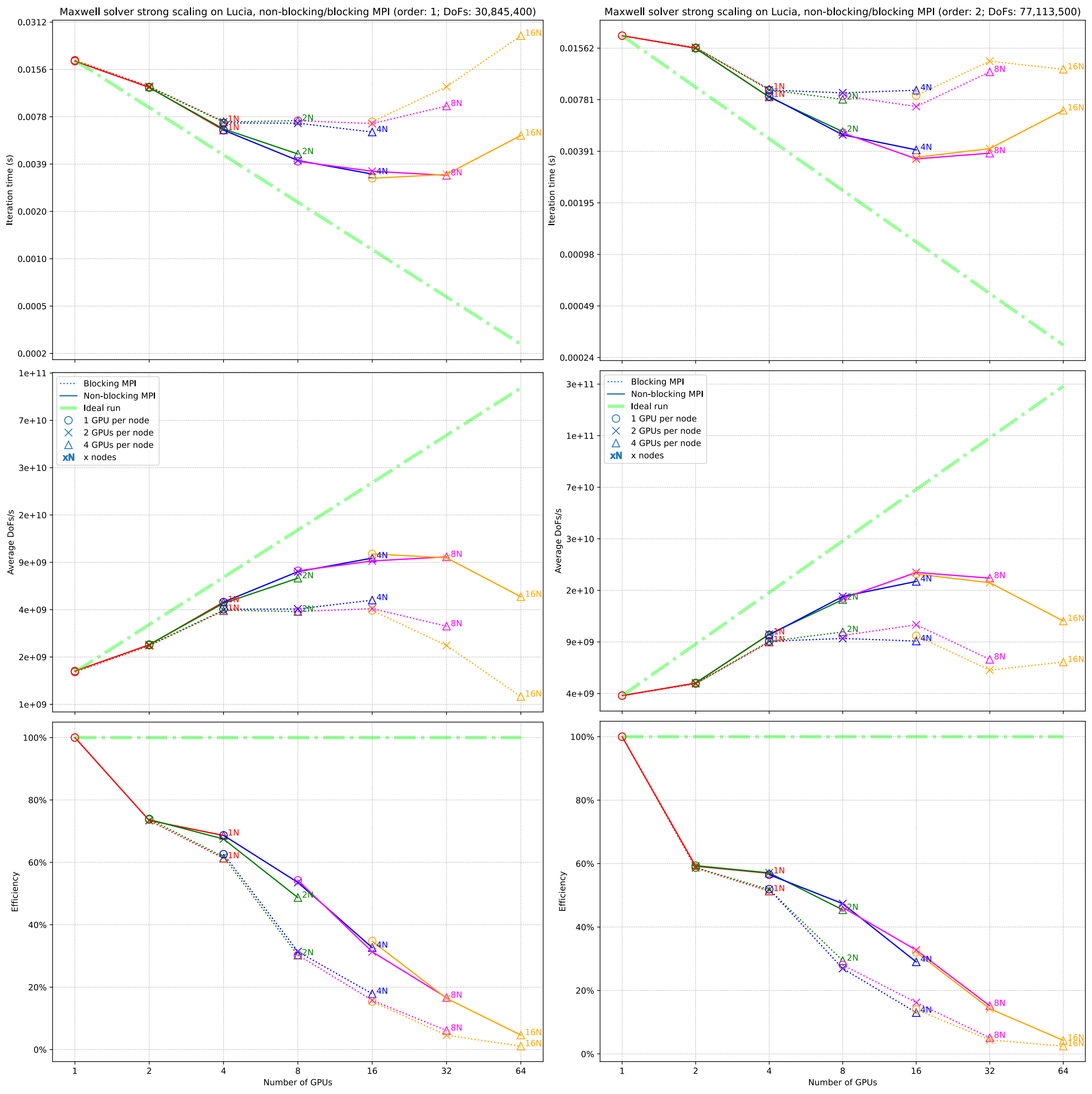


Figure 13 - Strong scaling analysis of the solver on Lucia for a comparison between blocking and non-blocking communication (orders 1 & 2). The plots show respectively: iteration time, average Degrees of Freedom (DoFs) per second and efficiency, given a number of GPUs.

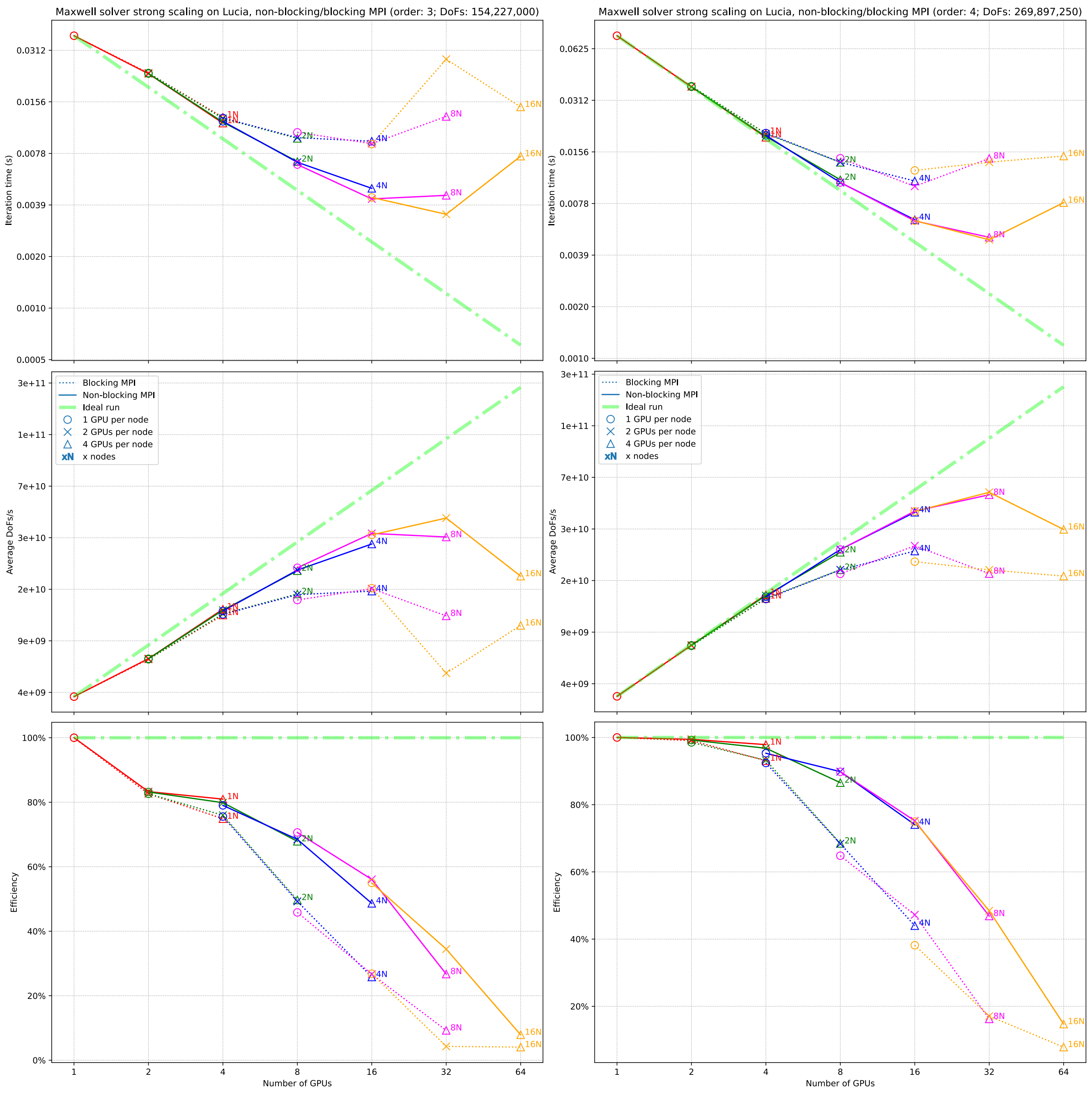


Figure 14 - Strong scaling analysis of the solver on Lucia for a comparison between blocking and non-blocking communication (orders 3 & 4). The plots show respectively: iteration time, average Degrees of Freedom (DoFs) per second and efficiency, given a number of GPUs.

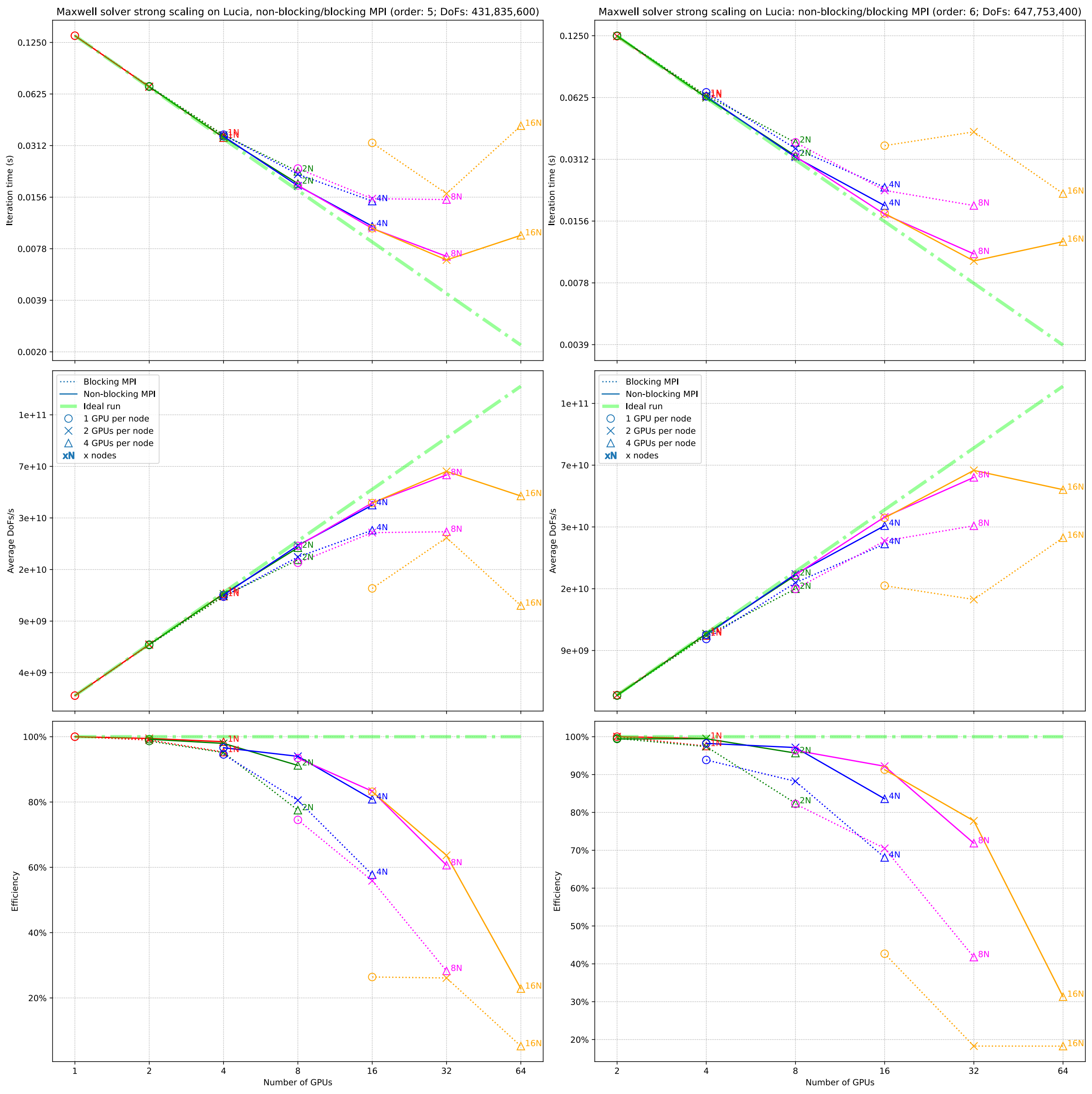


Figure 15 - Strong scaling analysis of the solver on Lucia for a comparison between blocking and non-blocking communication (orders 5 & 6). The plots show respectively: iteration time, average Degrees of Freedom (DoFs) per second and efficiency, given a number of GPUs.

6.1.B - Weak scaling

Before interpreting the results, we should pay attention to the "Efficiency" axis (and therefore the "DoFs/s/GPU" one too), as it is not consistently scaled across the plots, which can lead to misleading visual comparisons and misinterpretation of trends.

Due to the limited amount of memory, some runs could not complete. This only happened within the "close" mapping policy benchmarks: using 64 GPUs at the order 5 and when using 32 or more GPUs at the order 6.

Starting with an overall view, we can see that lowering the number of GPUs per node lowers the iteration time for some cases while it increases it in others just as in the strong scaling analysis. There is therefore no mapping policy that is better than the other one in terms of performance.

Continuing with the similarities with the strong scaling, the new non-blocking communication solver always performs on par or better than the old one, except 2 runs using the order 6. These are negligible, being only 2 out of 69 runs, and their efficiency difference just is 3~4%.

In the global tendency, we can see that higher orders (and thus bigger problems) scale far better, as they reach between 86% and 95% of efficiency with 16 GPUs at the orders 4, 5 and 6, while the two lower orders reach only 40%~50%.

Number of MPI ranks	1	2	4	8	16	32
Total iteration time (ms)	15	15.8	16.02	16.06	16.49	20.38
Communication time (ms)	/	0.22	0.55	0.7	1.27	4.97
Calculation time (ms)	15	15.58	15.47	15.36	15.22	15.41

Table 5 - Communication and calculation times from NVIDIA Nsight Systems profiling given a number of MPI ranks, for a single iteration of the non-blocking communication solver at the order 5, using the "close" mapping policy, with the same problem size per MPI rank on Lucia. Equivalent to weak scaling analysis in Figure 17, but slightly different timings due to the profiler overhead and measurements imprecision.

Just like in the strong scaling, a look at the communication and calculation parts is another great way to see the performance gain, see Table 5. As in the strong scaling, the communication time keeps increasing. Unfortunately, due to lack of memory, the test could not be performed with more than 32 MPI ranks, but enough data is available. We can expect the communication time to continuously increase with the number of GPUs: as the size of the problem solved by each GPU is constant, utilizing more of

	Number of GPUs	1	2	4	8	16	32
Blocking communication	GDoFs/s/GPU	3.15	3.08	2.92	2.8	2.6	1.85
	Efficiency (%)	100	97.8	92.8	90.2	84.1	59.8
Non-blocking communication	GDoFs/s/GPU	3.15	3.11	3.07	3.01	2.81	2.61
	Efficiency (%)	100	98.9	97.8	96.7	94.3	84.9

Table 6 - Weak scaling analysis of the solver on Lucia for a comparison between blocking and non-blocking communication at order 5 using the "closed" mapping policy. This is a transcript of Figure 15 in table form. For each version of the solver, the table show respectively: iteration time, average degrees of freedom — in GigaDoFs — per second per GPU, and efficiency, given a number of GPUs

them raises the total amount of transactions needed, while the amount of data per transaction remains mostly constant. On the other hand, computation time remains virtually constant and scales almost perfectly, confirming what we had observed with strong scalings. Small timing errors can be present, due to profiling overhead, or the fact that the number of DoFs per GPU is not *exactly* the same. Again, communication time is the only cause for the worse performance when using consequent amount of GPUs.

While we are at it, a more precise table can be made (see Table 6), in contrast to the more appealing graphs. This new resource can help to better interpret the results, but there is nothing more to add that what was already said for the graphs.

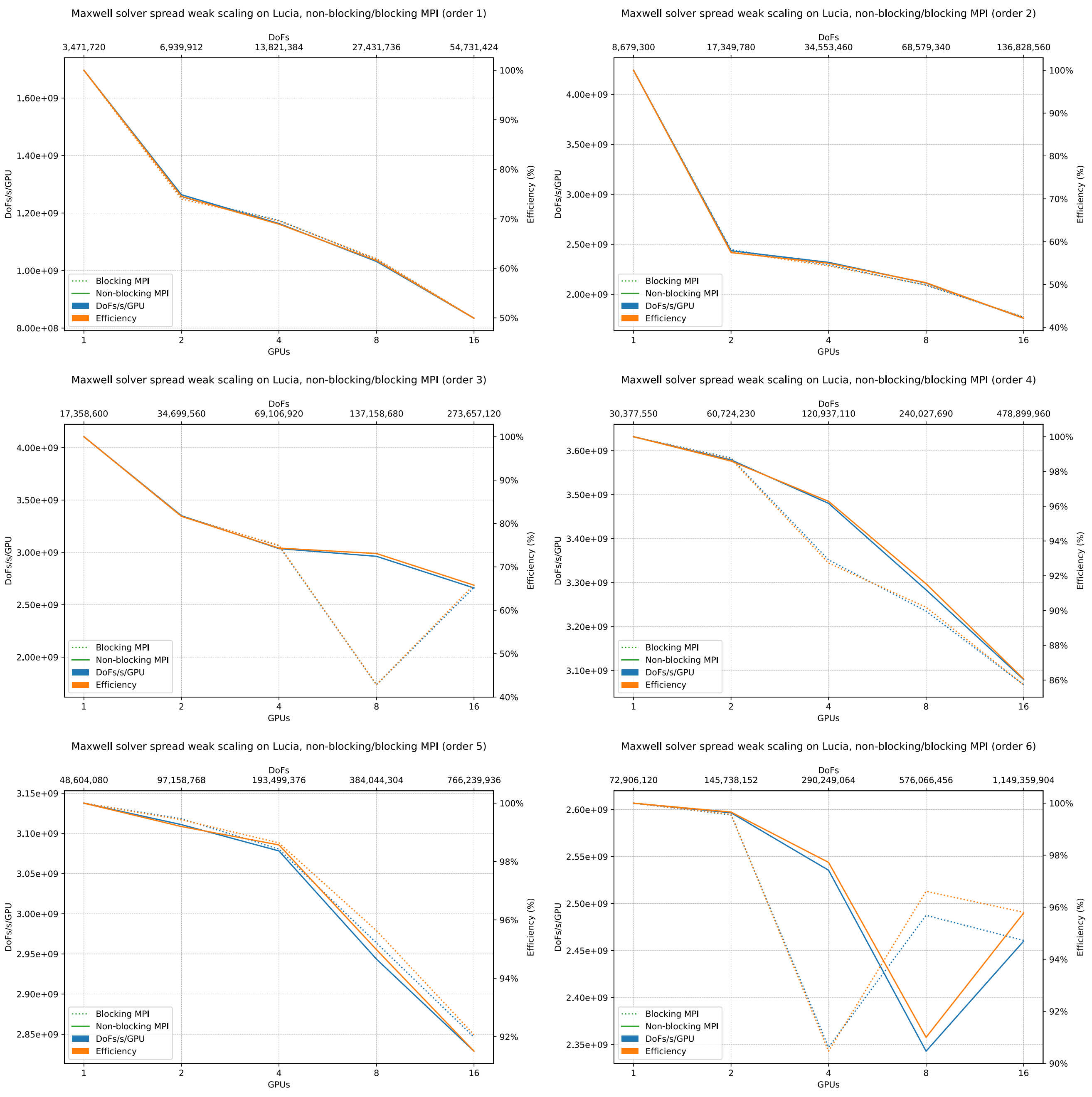


Figure 16 - Weak scaling analysis of the solver on Lucia using the "spread" mapping policy for a comparison between blocking and non-blocking communication. The plots show: Degrees of Freedom (DoFs) per second per GPU (on the left axis) and efficiency (on the right axis), given a number of GPUs (on the bottom axis) and problem size in DoFs (on the top axis).

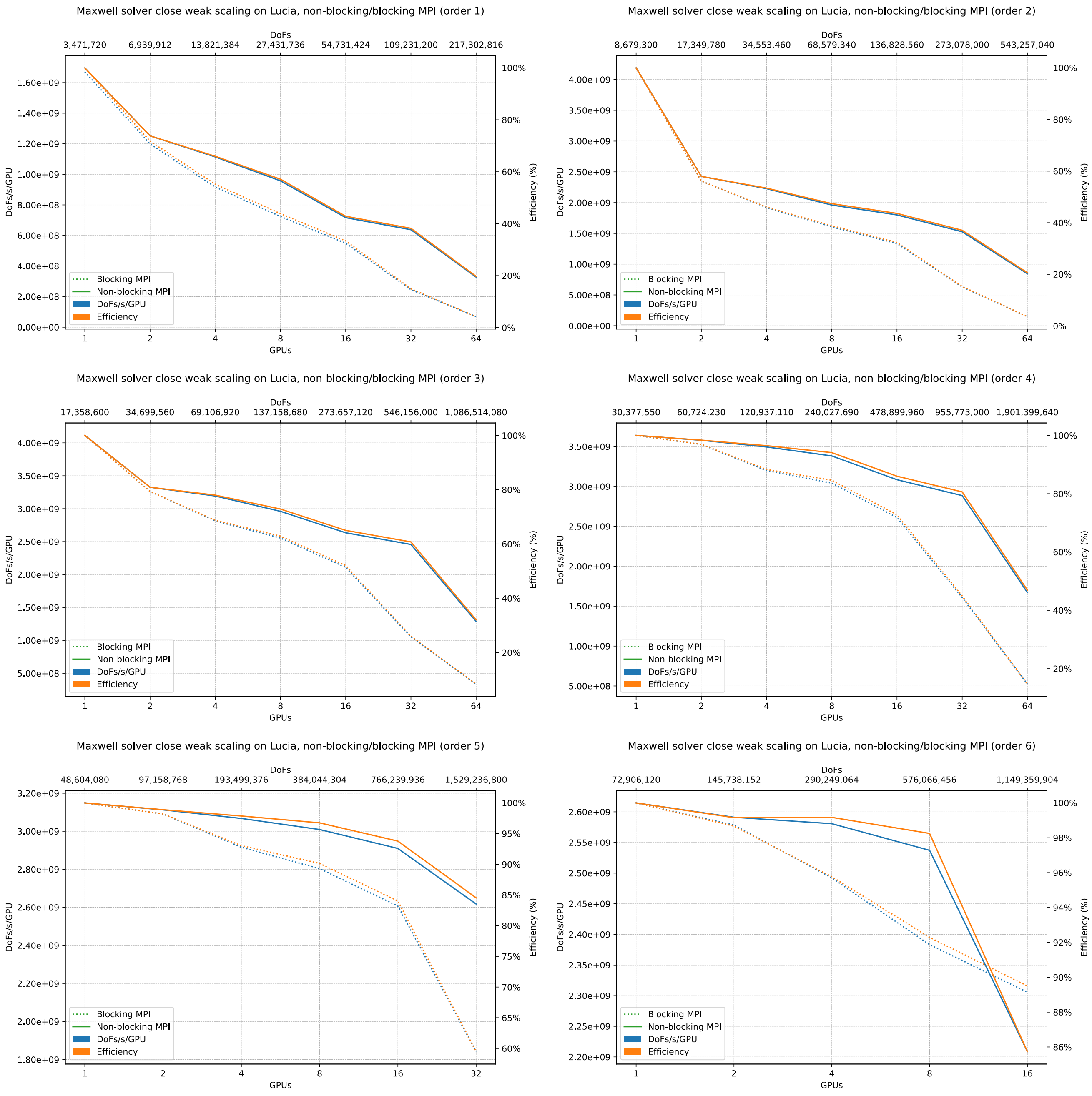


Figure 17 - Weak scaling analysis of the solver on Lucia using the “close” mapping policy for a comparison between blocking and non-blocking communication. The plots show: Degrees of Freedom (DoFs) per second per GPU (on the left axis) and efficiency (on the right axis), given a number of GPUs (on the bottom axis) and problem size in DoFs (on the top axis).

6.1.C - Revised weak scaling

The revised weak scaling shows pretty much the same pattern that we observed earlier: the solver scales better with bigger problems. This series of tests confirms the previous ones, but we do not compare anymore the two versions of the compiler. Instead, we only focused on the performance of the non-blocking MPI one, containing all our improvements.

The goal was to benchmark the solver on the (roughly) same problem size for the different approximation orders. As said earlier, the order 6 provided a problem big enough to require substantial compute work, while still small enough to fit in memory. Therefore, only the five other orders were revised. We played with the volume size to approach the number of DoFs of the order 6 below a 7% difference. The weak scaling for the order 6 can be found on the 2 previous pages.

The problem of limited memory happens once again, as none of the runs in the “close” mapping policy managed to reach 64 MPI ranks. Changing the mapping policy from “spread” to “close” has little impact on the efficiency, with only 1% of difference on average.

The graphs speak for themselves: only the three upper orders (4, 5 and 6) reach an efficiency of 90% and more for GPUs, while lower ones only 58%~78%. The order 2 gives particularly poor results compared to others. From what we see, increasing the number of GPUs (beyond the limits of what is shown on these graphs) should keep the same efficiency for lower orders (1, 2 and 3). Hopefully, this will be demonstrated by later benchmarks on the LUMI supercomputer.

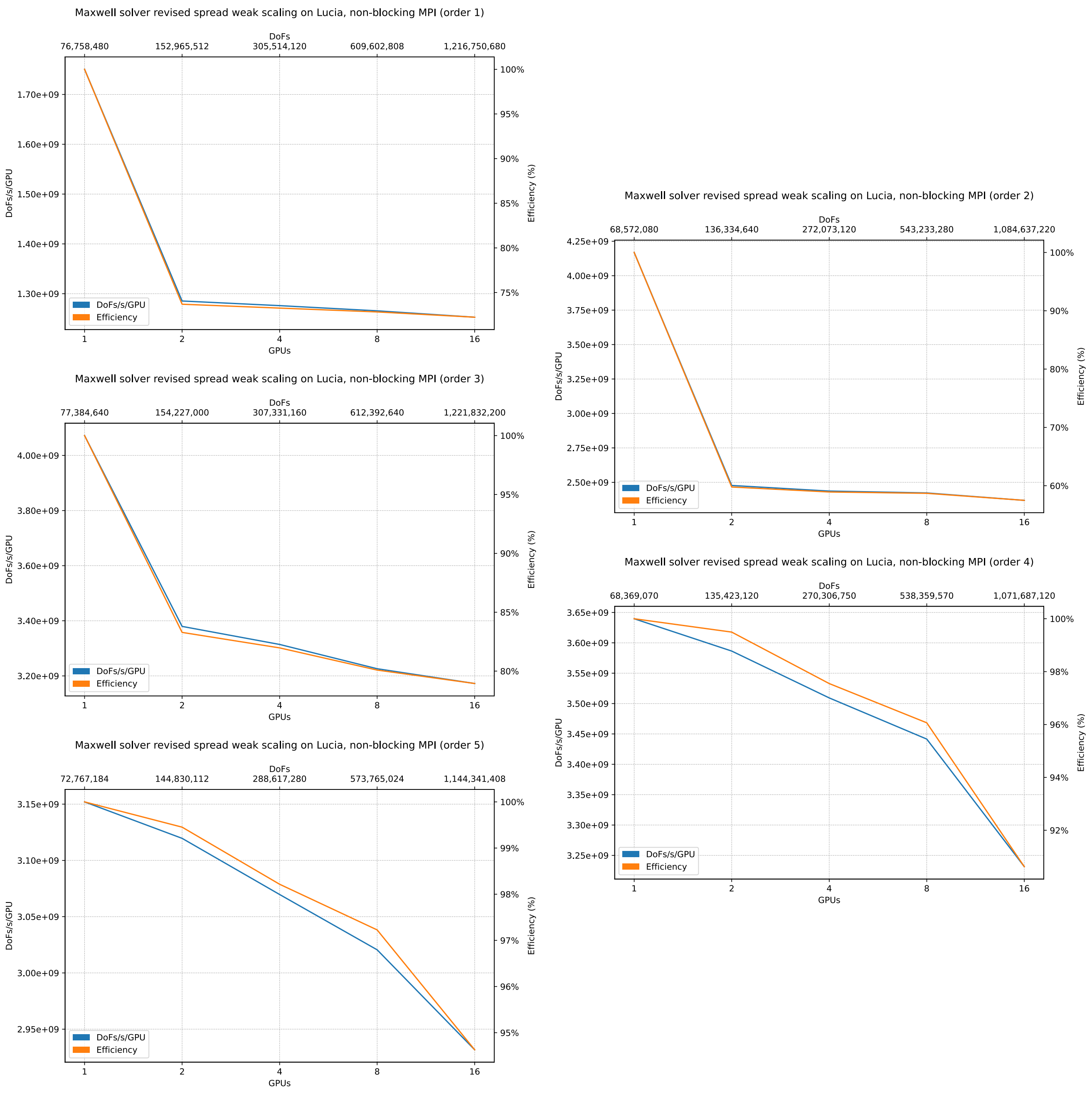


Figure 18 - Revised weak scaling analysis of the solver on Lucia using the "spread" mapping policy for a comparison between blocking and non-blocking communication. The plots show: Degrees of Freedom (DoFs) per second per GPU (on the left axis) and efficiency (on the right axis), given a number of GPUs (on the bottom axis) and problem size in DoFs (on the top axis).

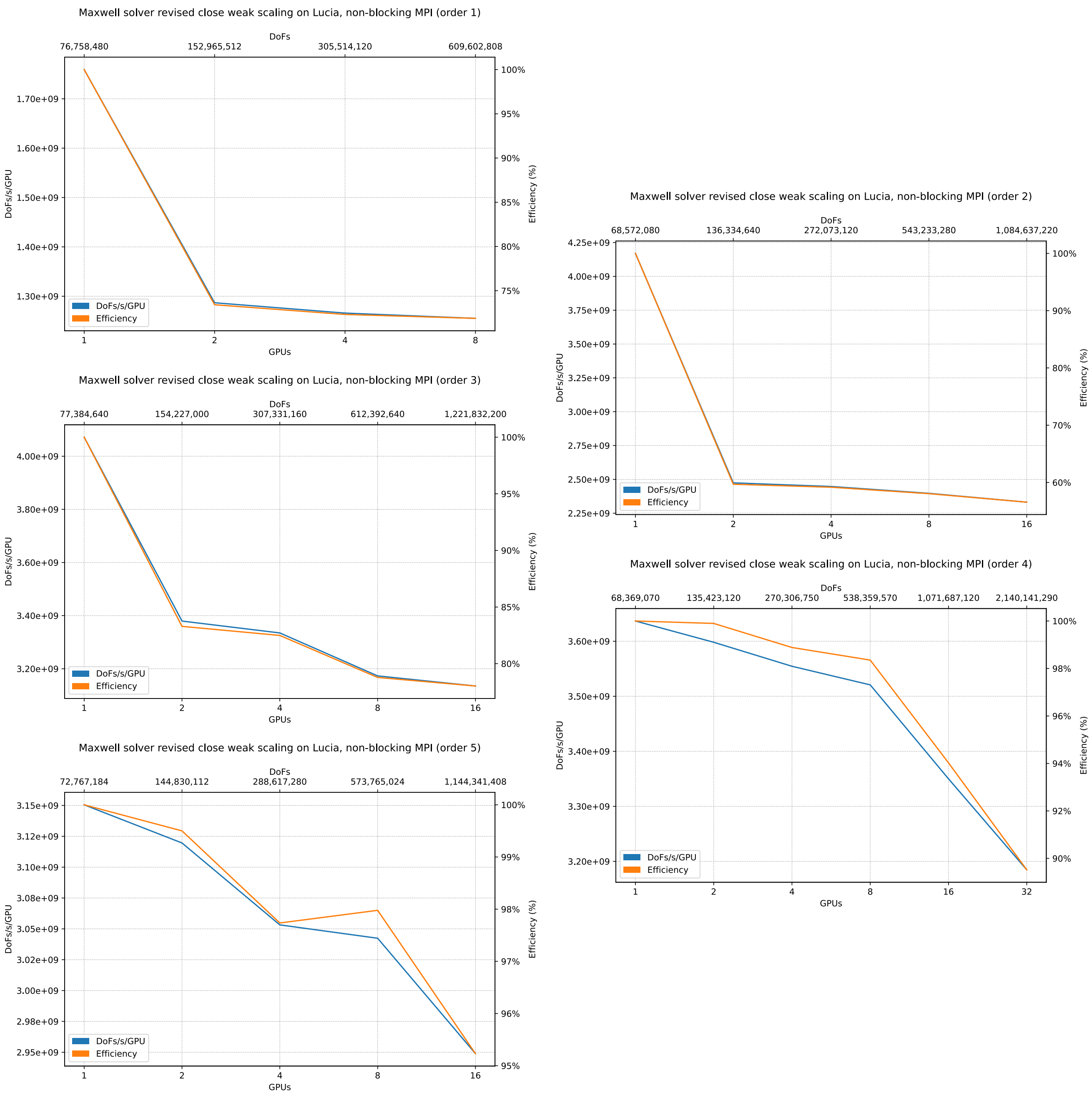


Figure 19 - Revised weak scaling analysis of the solver on Lucia using the “close” mapping policy for a comparison between blocking and non-blocking communication. The plots show: Degrees of Freedom (DoFs) per second per GPU (on the left axis) and efficiency (on the right axis), given a number of GPUs (on the bottom axis) and problem size in DoFs (on the top axis).

6.2 - Performance comparison on LUMI

Now that we have extensively tested our new solver and shown its performance improvements, the next step is to test it on another platform. This way, we can confirm (hopefully) our previous finding and interpretations of the different scaling analyses. This step is crucial, because if our solver performs correctly on two different architectures, we can predict its correct behavior on a wider set of platforms. This time, we only focused on the non-blocking MPI solver, as it was proven beneficial in all circumstances.

In the context of this master's thesis, we managed to get access to the European LUMI supercomputer. But before testing our solver on it, we should at least cover its differences with the Belgian Lucia supercomputer.

6.2.A - LUMI supercomputer details

LUMI, the Large Unified Modern Infrastructure based in Finland, is a supercomputer pooled by European resources with the aim of developing high-end exascale supercomputers for processing big data. In November 2024, LUMI was the world's 8th best supercomputer, reaching a throughput of 379.7 PFlops[2] from its LUMI-G partition [30] in the TOP500 list [31].

LUMI is composed of several partitions, each for different purposes, including (between others) one x86-CPU oriented (LUMI-C), one for quantum computing (LUMI-Q), and one GPU oriented (LUMI-G) [32]. We only used the latter one of course, as we are only interested in testing our solver on GPUs.

The 2978 nodes composing the GPU partition LUMI-G features the following characteristics [30], [33]:
- one 64-core AMD Trento CPU @2GHz
- four AMD MI250X GPUs
- four 200 Gbit/s network interconnect cards (= 800 Gbit/s injection bandwidth)
- eight 64 GB DDR4 memory (= 512 GB)

To correctly exploit the MI250x and obtain an optimal behavior, it is necessary to study it in details [34]. Each MI250x GPU is a multi-chip module, made of 2 Graphics Compute Die (GCD), for a total of 11,912 AMD GPUs for the whole partition (seen from a software perspective). A single GCD presents 110 usable

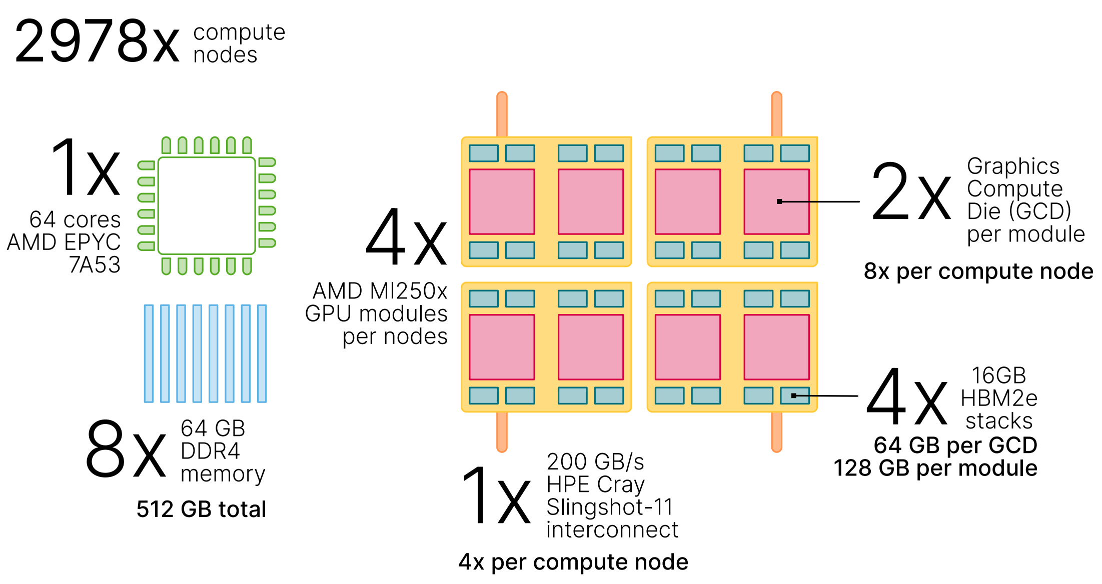

2978x compute nodes
1x 64 cores AMD EPYC 7A53
8x 64 GB DDR4 memory
512 GB total
4x AMD MI250x GPU modules per nodes
2x Graphics Compute Die (GCD) per module
8x per compute node
4x 16GB HBM2e stacks
64 GB per GCD
128 GB per module
1x 200 GB/s HPE Cray Slingshot-11 interconnect
4x per compute node

Figure 20 - Overview of a LUMI-G compute node [33].

compute units (2 are reserved) and 64 GB of memory. It contains a global shared L2 cache of 8MB, and can theoretically perform 24 TFlops.

Four 16-bit Infinity Fabric links the GPU dies in a MI250x, reaching 200 GB/s per direction, while a single one is used between different MI250x accelerators, making it 50 GB/s per direction. We should note that all MI250x GPUs in the compute node are all connected together, but this is not the case for the GPUs dies: they are only connected to 3 other dies.

Thanks to one Network Interface Card (NIC) per MI250x with an Ethernet connection of 25 GB/s per direction, internode communication can go up to 100 GB/s in each direction.

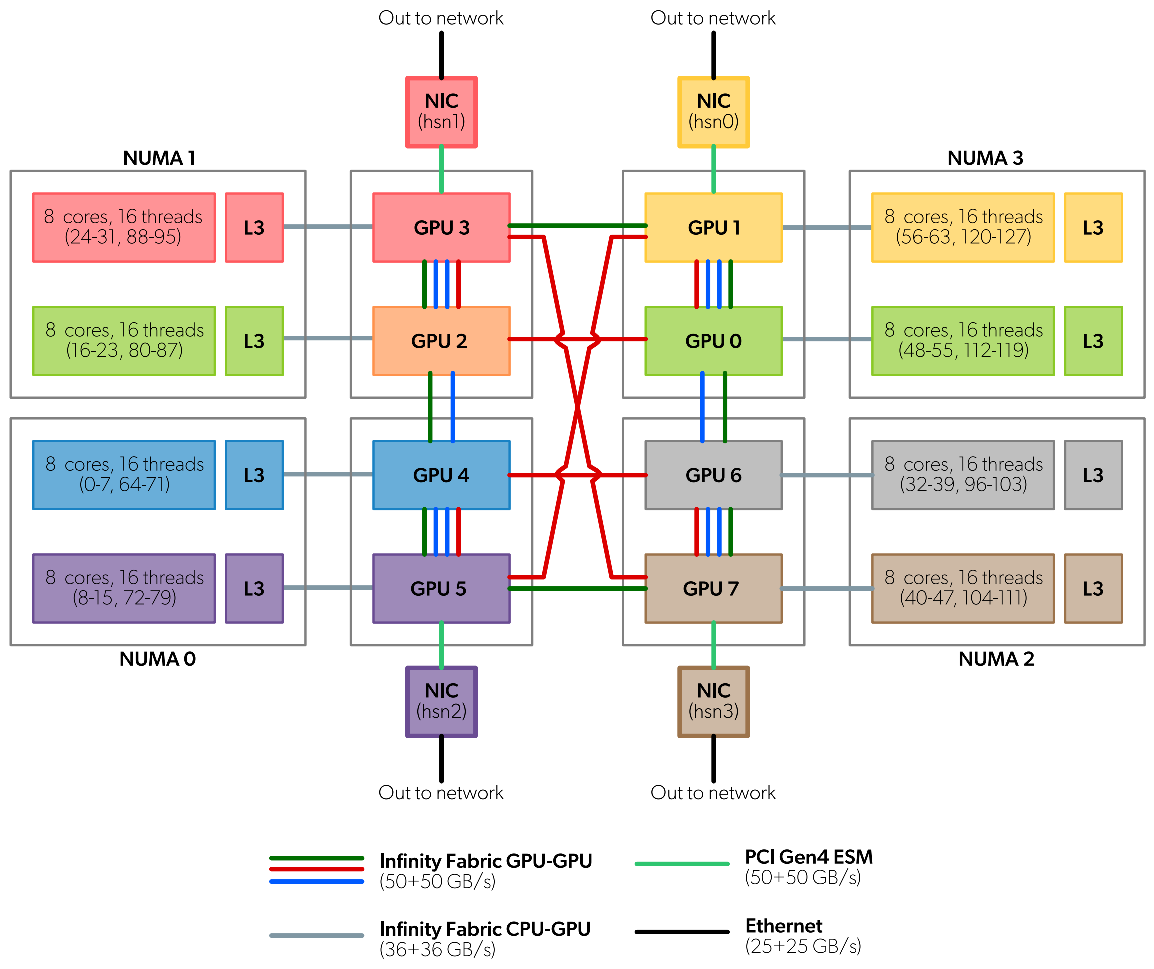

Out to network
Out to network
NIC (hsn1)
NIC (hsn0)
NUMA 1
NUMA 3
8 cores, 16 threads (24-31, 88-95)
L3
GPU 3
GPU 1
8 cores, 16 threads (56-63, 120-127)
L3
8 cores, 16 threads (16-23, 80-87)
L3
GPU 2
GPU 0
8 cores, 16 threads (48-55, 112-119)
L3
8 cores, 16 threads (0-7, 64-71)
L3
GPU 4
GPU 6
8 cores, 16 threads (32-39, 96-103)
L3
8 cores, 16 threads (8-15, 72-79)
L3
GPU 5
GPU 7
8 cores, 16 threads (40-47, 104-111)
L3
NUMA 0
NUMA 2
NIC (hsn2)
NIC (hsn3)
Out to network
Out to network
Infinity Fabric GPU-GPU (50+50 GB/s)
PCI Gen4 ESM (50+50 GB/s)
Infinity Fabric CPU-GPU (36+36 GB/s)
Ethernet (25+25 GB/s)

Figure 21 - LUMI-G node architecture and rings [34].

We should recall that our solver's GPU code is written in CUDA C++, which is compiled by the CUDA Runtime and targets NVIDIA GPUs. In order to be able to run our code on AMD GPUs, the use of *Hipify* is needed [11]. *Hipify* allows us to port the code to the AMD ROCm platform with an automatic conversion of languages. Since we, as the developer, do not write natively the code for that latter platform in HIP, this "blind" operation could be the cause of a small performance decrease.

In short, LUMI has twice as much GPUs per node as Lucia (Slurm sees GCDs as separate graphics cards), and same for the main Random Access Memory (RAM) memory. The amount of memory per GCD has raised from 40 to 64 GB. By only looking at these numbers, we thought we would be able to go further for the weak scalings, using the "close" mapping policy. But we should be reminded that our solver initialization is far from good and holds a lot of duplicate data across MPI ranks, which might introduce some bottlenecks because we can now have twice the MPI ranks per compute node, so the memory needs are more than just twice. We should also note that a L2 cache in a GCD on LUMI is significantly smaller than the L2 cache in a NVIDIA A100 (on Lucia). We came from 40MB to 8MB, so a ratio of 5. This will bring in significantly more cache misses.

Just like Lucia, it uses the Slurm job scheduler [19].

6.2.B - Strong scaling

The first thing that catches the eye in our opinion is that the runs made on LUMI are noticeably slower than on Lucia, with an average speed difference of 30%. As we warned in the previous subsection, this issue mainly comes from the limited amount of L2 cache on LUMI, which is 5 times smaller than on Lucia. The automatic conversion from CUDA to HIP also contributes to the speed decrease, but to a much lesser extent.

Then, we finally can see the behavior of our solver when it uses up to 512 MPI ranks and GPUs. We were quite limited on the Lucia supercomputer, as it was already quite hard to get an access to 16 compute nodes simultaneously, but we do not have this problem anymore with the LUMI supercomputer has it has a few thousands of them. This allows us to confirm its pretty good scalability on LUMI, but also to notice that the data gathered from LUMI is more consistent: we no longer have any outliers, and its efficiency has increased drastically. It should be noted that the efficiency reports were drawn up separately for the Lucia and LUMI runs, in order to eliminate the 30% speed decrease from the “Efficiency” graph — as it is already shown in “Average DoFs/s” and “Iteration time (s)” graphs. The “Ideal run” green line was cut at 64 MPI ranks, otherwise the meaningful data would have been more difficult to read and correctly interpret as they would have appeared smaller.

Unlike the tests conducted on Lucia, using the “spread” mapping policy (or in other words using more nodes for the same number of MPI ranks) is *slightly* beneficial on LUMI: up to 2% of gain in efficiency.

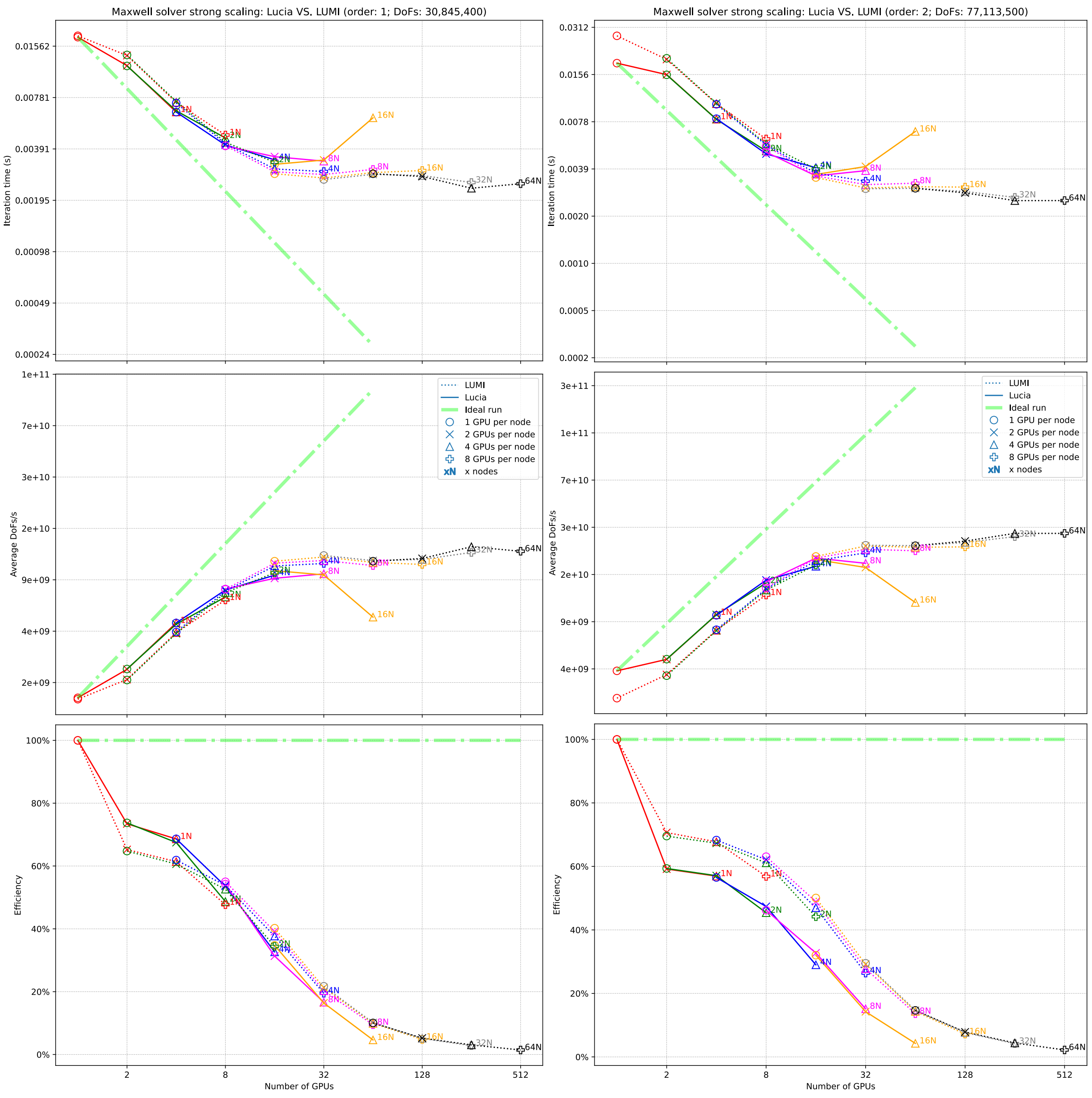


Figure 22 - Strong scaling analysis of the non-blocking communication solver for a comparison between LUMI and Lucia (orders 1 & 2). The plots show respectively: iteration time, average Degrees of Freedom (DoFs) per second and efficiency, given a number of GPUs.

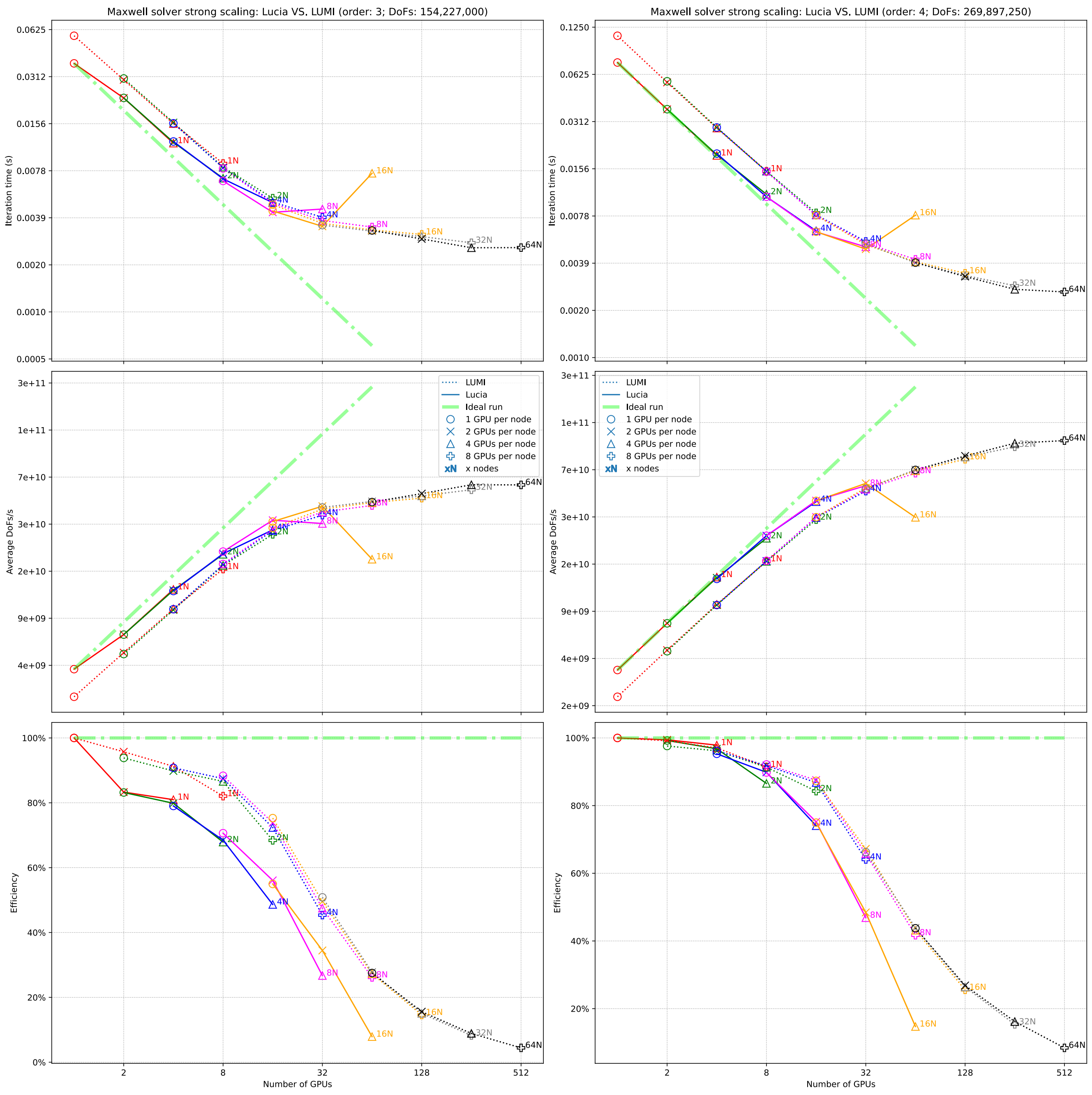


Figure 23 - Strong scaling analysis of the non-blocking communication solver for a comparison between LUMI and Lucia (orders 3 & 4). The plots show respectively: iteration time, average Degrees of Freedom (DoFs) per second and efficiency, given a number of GPUs.

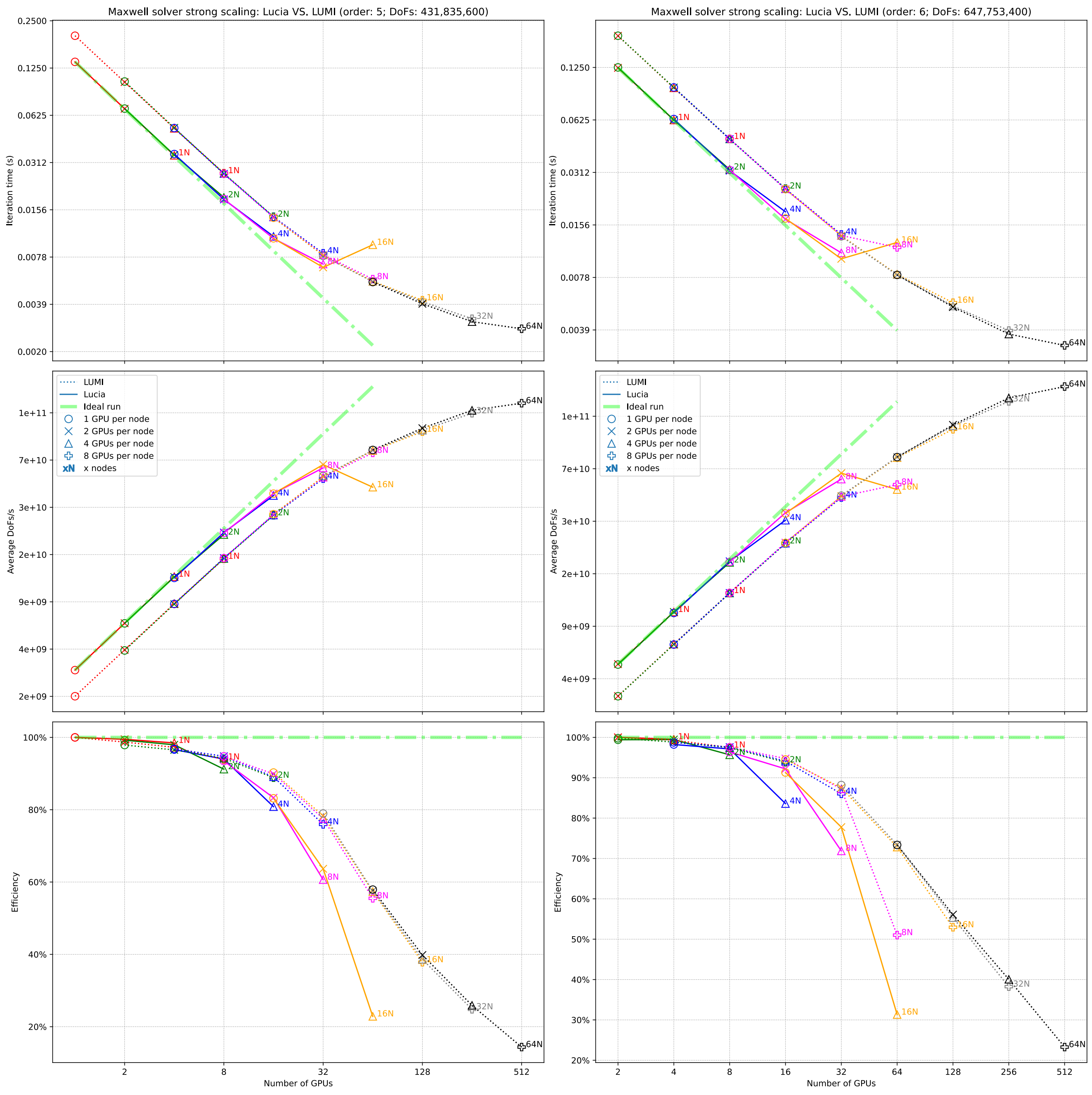


Figure 24 - Strong scaling analysis of the non-blocking communication solver for a comparison between LUMI and Lucia (orders 5 & 6). The plots show respectively: iteration time, average Degrees of Freedom (DoFs) per second and efficiency, given a number of GPUs.

6.2.C - Weak scaling and revised weak scaling

Most of the points mentioned in the strong scaling analysis can be seen in the weak scaling one too. Again, LUMI presents a performance hit of about 30% in throughput.

Overall, our solver scales better on LUMI than on Lucia, as it gives an increase of 4 to 10% in the efficiency for 8+ GPUs using the “spread” mapping policy. The efficiency difference is even greater with MPI ranks thanks to the “close” mapping policy: up to 20%. Taken alone, our solver’s efficiency on LUMI reaches quite good levels for real-life applications, as long as the problem is big enough. For higher orders 4, 5 and 6, it is above the 80% threshold using 16+ GPUs.

Also, the problem of the solver taking up too much memory resurfaces. Indeed, the different graphs showing the weak scaling using the “close” mapping policy not all show data up to 512 GPU — 128 nodes, each with 8 GPUs. For the same reason, the revised weak scaling was unsuccessful: as we supposed in Section 6.2.A, doubling the volume size and GPUs (= MPI ranks) requires more than twice the amount of memory, for which case LUMI did not have enough — except for three benchmarks.

The revised scaling are shown just for confirmation of all the information we managed to extract from previous conducted tests.

In the end, benchmarks on LUMI confirmed our solver’s good behavior and scaling.

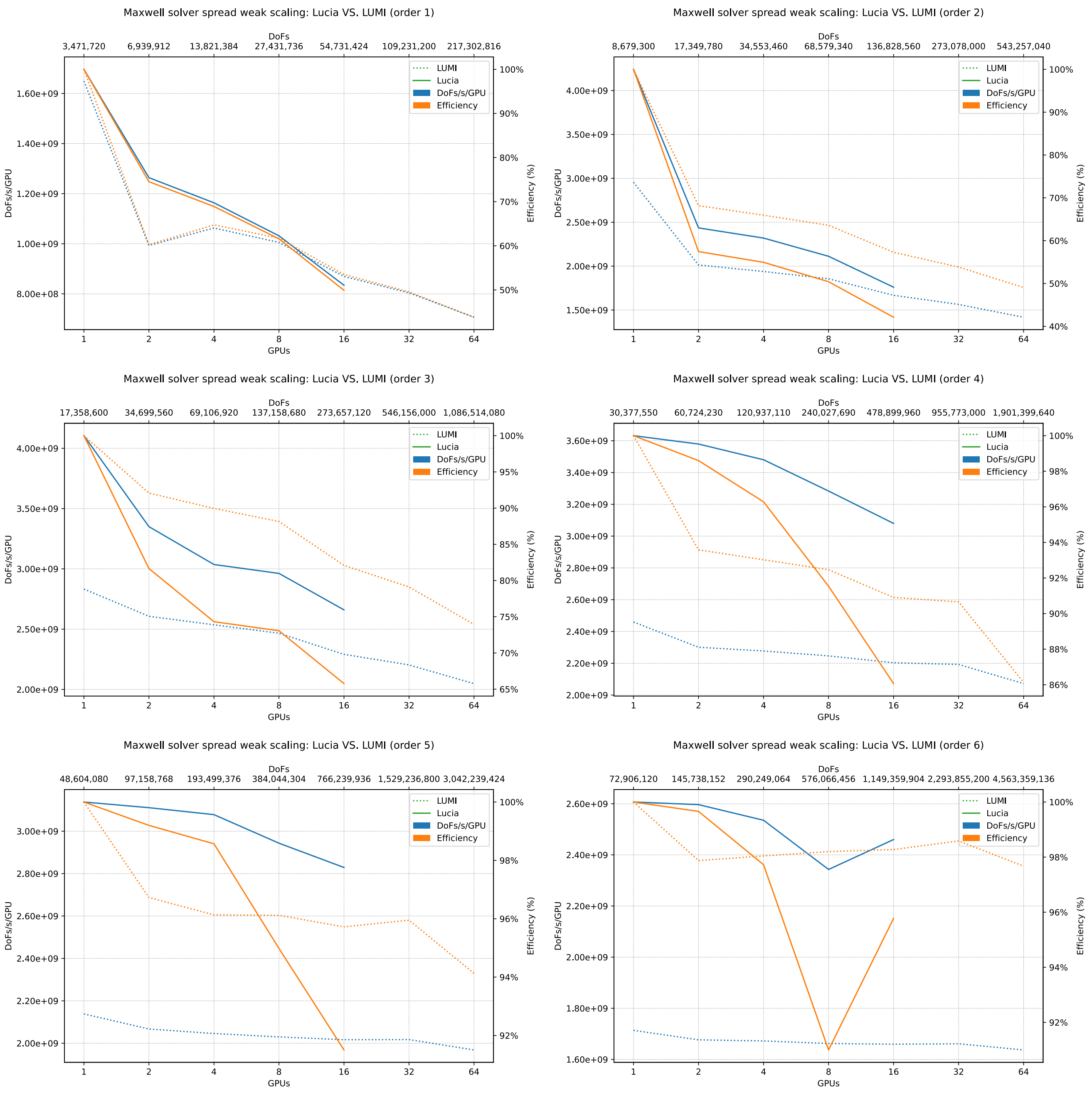


Figure 25 - Weak scaling analysis of the non-blocking communication solver using the "spread" mapping policy for a comparison between LUMI and Lucia. The plots show: Degrees of Freedom (DoFs) per second per GPU (on the left axis) and efficiency (on the right axis), given a number of GPUs (on the bottom axis) and problem size in DoFs (on the top axis).

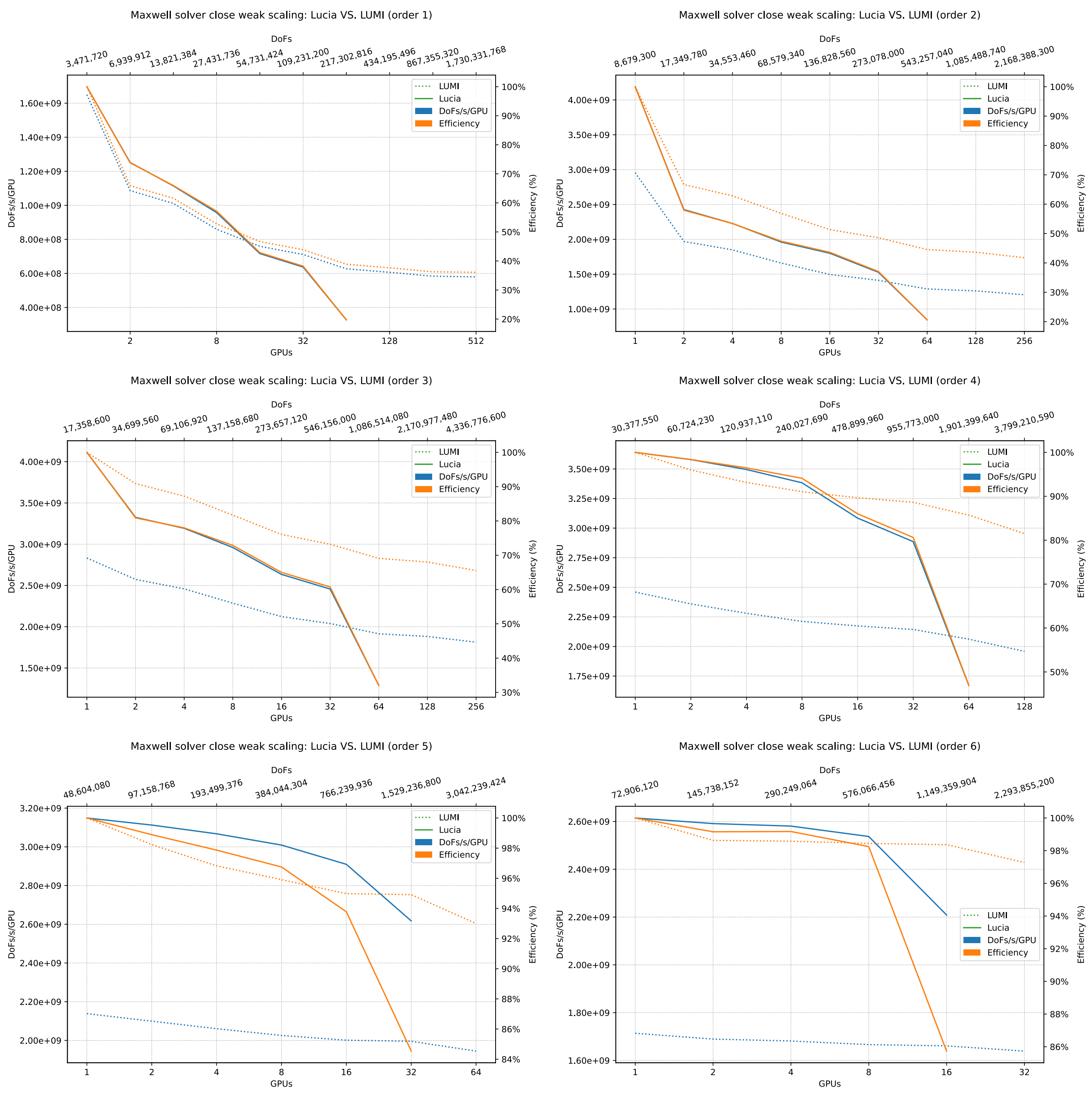
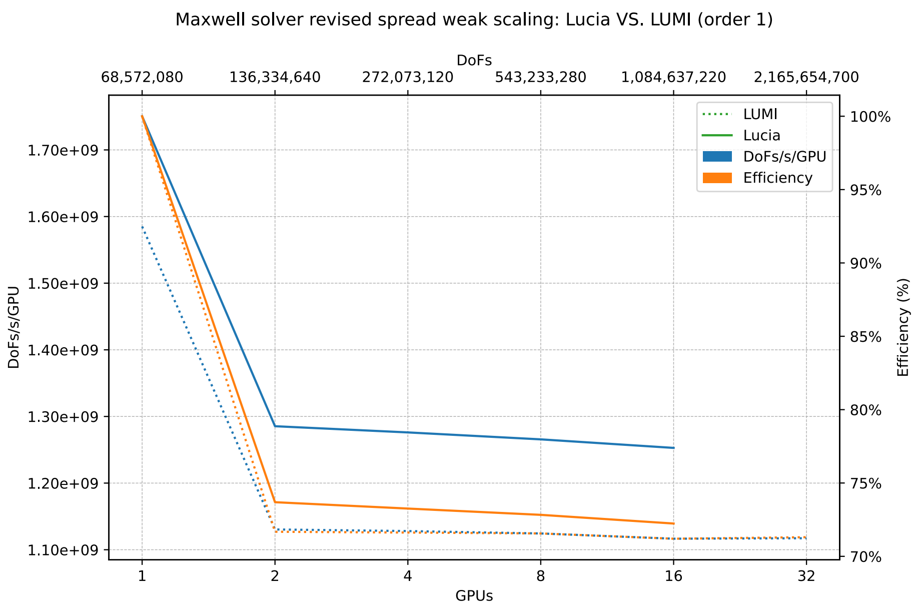
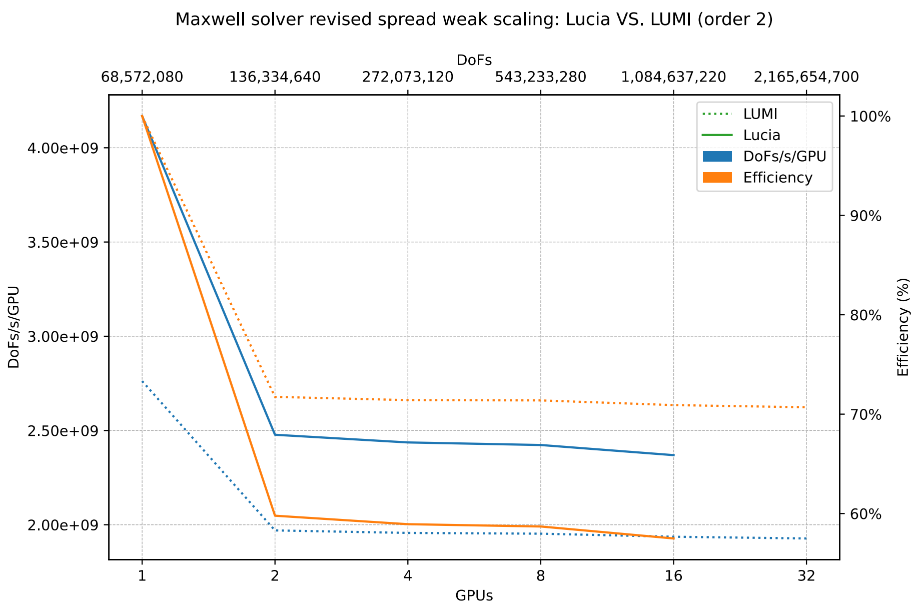


Figure 26 - Weak scaling analysis of the non-blocking communication solver using the “close” mapping policy for a comparison between LUMI and Lucia. The plots show: Degrees of Freedom (DoFs) per second per GPU (on the left axis) and efficiency (on the right axis), given a number of GPUs (on the bottom axis) and problem size in DoFs (on the top axis).





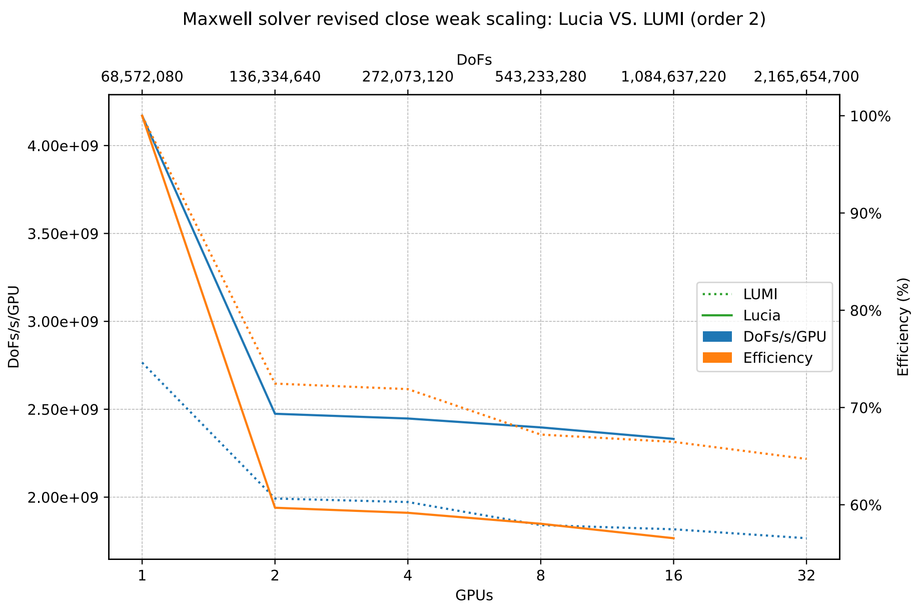


Figure 27 - Revised weak scaling analysis of the non-blocking communication solver for a comparison between LUMI and Lucia. The plots show: Degrees of Freedom (DoFs) per second per GPU (on the left axis) and efficiency (on the right axis), given a number of GPUs (on the bottom axis) and problem size in DoFs (on the top axis).

6.3 - Possible additional performance gains

While the previous performance and scalability analyses demonstrated a great overall improvement, there is still room for significant enhancement. Indeed, up until now, we only changed the blocking MPI communication to non-blocking, allowing ranks to exchange data with several others ranks at the same time. This does not change the order of operations, as it was shown in Figure 4. There are different ways of making the solver more efficient (i.e. shorter iteration times), but some of them might be negligible.

One example would be to more cleverly exchange boundaries data between MPI ranks. Indeed, a rank could not wait to send its data to its neighbors until the start of the next transactions (after just one more iteration), since the data ready to be sent is located in buffers, untouched by the compute-intensive part of an iteration. This would allow the rank to wait less, as it would start the calculation of the next iteration directly after receiving data from neighboring ranks. It is still required to wait for its “send” transactions to finish right before the buffers are modified at the next iteration.

One could also imagine changing the order in which the transactions are carried out, as they are currently sorted according to receiver rank. Different heuristics could be adopted, for example prioritizing communication to ranks running on another node. Changing the order would even be more beneficial in combination with the previously explained optimization.

The previous possible enhancements are only related to the communication itself, and should have a relatively small impacts on the current performance. It is true that any attempt at optimizing the computational throughput of GPUs would lead to small or even negligible performance gains (as the Figure 8 shows that we are close to hardware limitations), while our solver does not *perfectly* scale. In practice, a perfect scaling where the program’s efficiency is always 100% with any number of threads is not possible. But we should try to get as close as it is possible. Right now, in any configuration, the efficiency already drops below 90% when just using 32 GPUs, and more drastically when using even more GPUs (as in Figure 24).

Scalability problems can be seen in Figure 12: for the duration of the communication process, ranks simply wait for data exchanges to finish, calculating nothing (or almost nothing). This results in wasted computational power. To overcome this, we need to find useful work for them to do during these transactions, without delaying the communication (too much). We will refer to this as *masked communication*: it will be done in the “background”. At the time of writing this, we still had some time to spare. Enough time to actually delve deeper into the subject and try our hand at masked communication.

7 - Masking communication

Now that we have implemented non-blocking communication, the next significant performance gain can be achieved by masking it. The whole problem lies in finding useful intensive work to do during these data exchanges. As a reminder, we have 4 main steps in a single iteration of the solver, as we can see in Figure 4: computing curls, jumps and fluxes, then lifting fluxes. We should recall that there are two different paths: a volumetric path (blue) and a boundary path (green). They do not have any data dependency between each other *within the same iteration*, so the paths can be interleaved in any way, as long as each of them is done sequentially. The jumps computation involves communication between the elements inside the same partition, then partitions exchanges their boundaries data before computing fluxes. We can therefore compute curls from the other path when data is exchanged between MPI ranks, as shown in Figure 28.

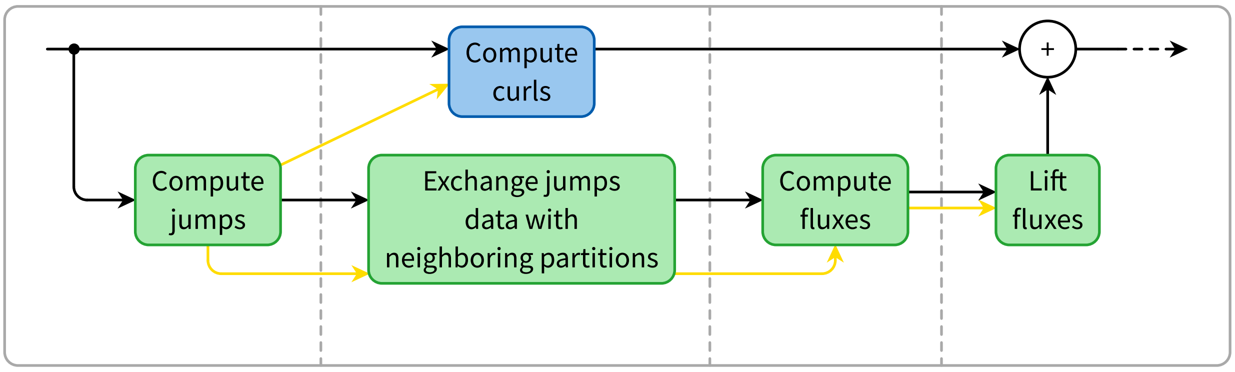


Figure 28 - Steps required in order to do one application of the Discontinuous Galerkin operator with a partitioned volume; GPU execution path in yellow.

Thus, some arrangements in the solver implementation are needed, in order to compute curls while exchanging data with other MPI ranks. This is not straightforward, as both jumps communication (remember Listing 2) and curls computation need some GPU kernels to be executed. In order to execute kernels concurrently, we should use different GPU streams for these two operations, as we have seen in Section 3.4.A.

Since we want to use multiple streams, we should also define their relative priority. When we launch curls computation and jumps data exchanges concurrently, two outcomes are possible. The first is when curls computation finishes *before* communication. In that case, the only work left is on the boundary path, and we should prioritize the kernels related to communication to start computing fluxes (then lifting them) as soon as possible. The second outcome is when curls computation takes *more* time than communication. Here, the purpose of curls computation is still to use GPU resources when the communication kernels does not need them (thus should have less priority). On the other hand, fluxes computation and lifting should not have a higher priority than curls, so the latter are not preempted or slowed down.

In the end, we can create 2 streams: a high-priority one for communication-related kernels, and a low-priority one for all other kernels.

7.1 - First performance results

Once the previous idea is implemented, let's profile once again our solver using NVIDIA Nsight Systems when using a significant amount of GPUs (e.g. 64), to show how communication is going as it is the limiting factor on iteration time. This is shown in Figure 29. The performance has increased compared to the non-blocking MPI version, as we expected: the figure shows that the GPU is working for approximately 76% of the time of an iteration. The 24% left are used to simply wait for the termination of jumps data exchange, meaning that the curls computing was, in this case, too short to completely mask communication. We should remind that we came from a long way: the original solver — with blocking communication — spent 96% of the time in communication, while the non-blocking version 79%. Our new version of the solver already shows breathtaking improvements in performance.

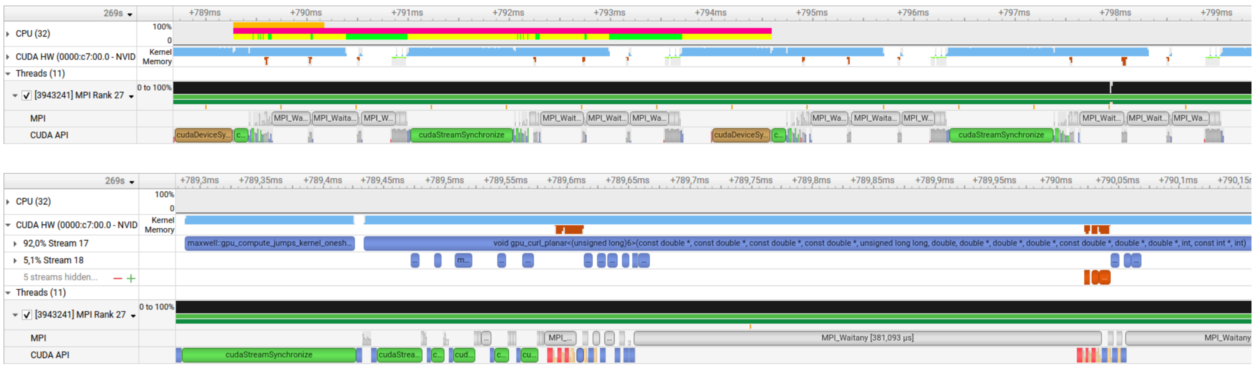

CPU (32)
CUDA HW (0000:c7:00.0 - NVID
Threads (11)
[3943241] MPI Rank 27
MPI
CUDA API
92,0% Stream 17
5,1% Stream 18
5 streams hidden...
cudaStreamSynchronize
cudaDeviceSy...
MPI_Waitany [381,093 µs]
maxwell::gpu_compute_jumps_kernel_onesh...
void gpu_curl_planar<(unsigned long)6>(const double *, const double *, const double *, const double *, unsigned long long, double, double *, double *, double *, const double *, double *, double *, int, const int *, int)

Figure 29 - Profiling from NVIDIA Nsight Systems of the masked communication solver on the 27th MPI rank, for a run using 64 GPUs spread over 16 nodes on Lucia. The image underneath is a zoom of the one on top and shows different CUDA streams. Colors were manually inserted in the “CPU (32)” line: orange, pink, yellow and green represent the interval of the image underneath, a complete single iteration (with 2 DG operations), computational time and communication time — respectively.

We also have the confirmation that the 2 streams defined earlier are running concurrently: the one with ID 18 for kernels used to send and receive jumps data, and the one with ID 17 for everything left.

A performance boost is present for this specific run, but like in previous analyses we should test the solver with different configurations and problem sizes to confirm the improvement. This time however, we will only analyze the strong scaling. This choice was done because the strong scaling of the solver with non-blocking communication showed significant worse results, with an efficiency difference of e.g. at least 20% when using 32 GPUs if we compare Table 6 and Table 4. Focusing on strong scaling helps us better visualize and address the communication bottlenecks that become critical at high GPU counts, which are not as visible in weak scaling scenarios.

7.2 - Strong scaling

The strong scaling plots, which can be found at the end of this subsection, show the difference between masked (non-blocking) communication and the previous non-blocking communication implementation from Listing 2.

In the general tendency, masking communication has a benefit in all different settings, and particularly when using numerous GPUs with up to 2.5 times faster iteration time.

We can also notice that the solver now has an almost perfect scalability for big enough problems (orders 5 and 6) with up to 32 GPUs: we have more than 95% of efficiency. The previous NVIDIA Nsight Systems profiling on 64 GPUs (Figure 29) revealed that the curls computation was too short to cover all data exchanges. This is the only main cause for runs that are relatively far from the ideal run. For a fixed problem size, increasing the amount of GPUs results in smaller partitions, which translates into a reduction in communication time and, more significantly, in curls computation time. This can be illustrated with a small example: double the number of GPUs, and the volume of a single partition will be twice as small, while its surface area will be less than twice as small. For a rectangular cuboid, only 4 faces have their surfaces area divided by 2, others remain untouched. A unit cube (1x1x1) split in two will have its surface area divided by $\frac{1*1*6}{0.5*1*4+1*1*2} = 1.5$, which is the same for any size of rectangular cuboid. This explains why curls computation time decreases faster than jumps data communication.

To demonstrate in another way the improvement brought about by communication masking, let’s look again at the time distribution between calculation and communication, with greater precision than what graphs offer, using NVIDIA Nsight Systems profiling. It is not possible to scale perfectly, because even if communication is masked, it still slows down the environment. Indeed, jumps data exchanges involve kernel launches to copy to and read from buffers. These kernels use part of the GPU’s computing power, so it will have fewer resources for computing curls. In the following Table 7, communication time refers to the time taken for all communication-related operations, even if they are performed in parallel with other calculations (since they slow down the latter): we need to highlight the extra time required compared with a run without communication but with the same partition properties (e.g. size and DoFs). It is important to note that from 32 GPUs, the curls calculation starts to terminate before the communication is fully completed, which increases the communication time more substantially.

Number of GPUs	1	2	4	8	16	32	64
Iteration time (ms)	136.8	68.6	34.28	17.18	8.68	4.81	4.19
Communication time (ms)	/	0.09	0.2	0.31	0.48	0.78	1.11
Calculation time (ms)	136.8	68.51	34.08	16.87	8.2	4.03	3.08

Table 7 - Calculation communication times from NVIDIA Nsight Systems profiling given a number of MPI ranks, for a single iteration of the masked communication solver at the order 5 and 431,835,600 DoFs, using the “close” mapping policy with the same problem as Table 3 for all runs on Lucia.

We are happy to finally see the first strong scaling results where the iteration time keeps decreasing with more GPUs, at orders 5 and 6. Just like for the previous implementation change (from blocking to non-blocking communication), performance has improved significantly. To better see the improvements, a summary of the 3 different solver implementations is given in Table 8.

Number of GPUs		1	2	4	8	16	32	64
Iteration time (ms)	Blocking MPI	136.8	69.1	35.9	22.1	14.8	15.1	40.8
	Non-blocking MPI	136.8	68.7	35	18.8	10.6	7.12	9.28
	Masked non-blocking MPI	136.8	68.6	34.3	17.2	8.7	4.4	3.5

Table 8 - Average iteration time in milliseconds given a number of MPI ranks at the order 5 and 431,835,600 DoFs, using the “close” mapping policy on Lucia for all 3 different versions of the solver: blocking, non-blocking, and masked communication.

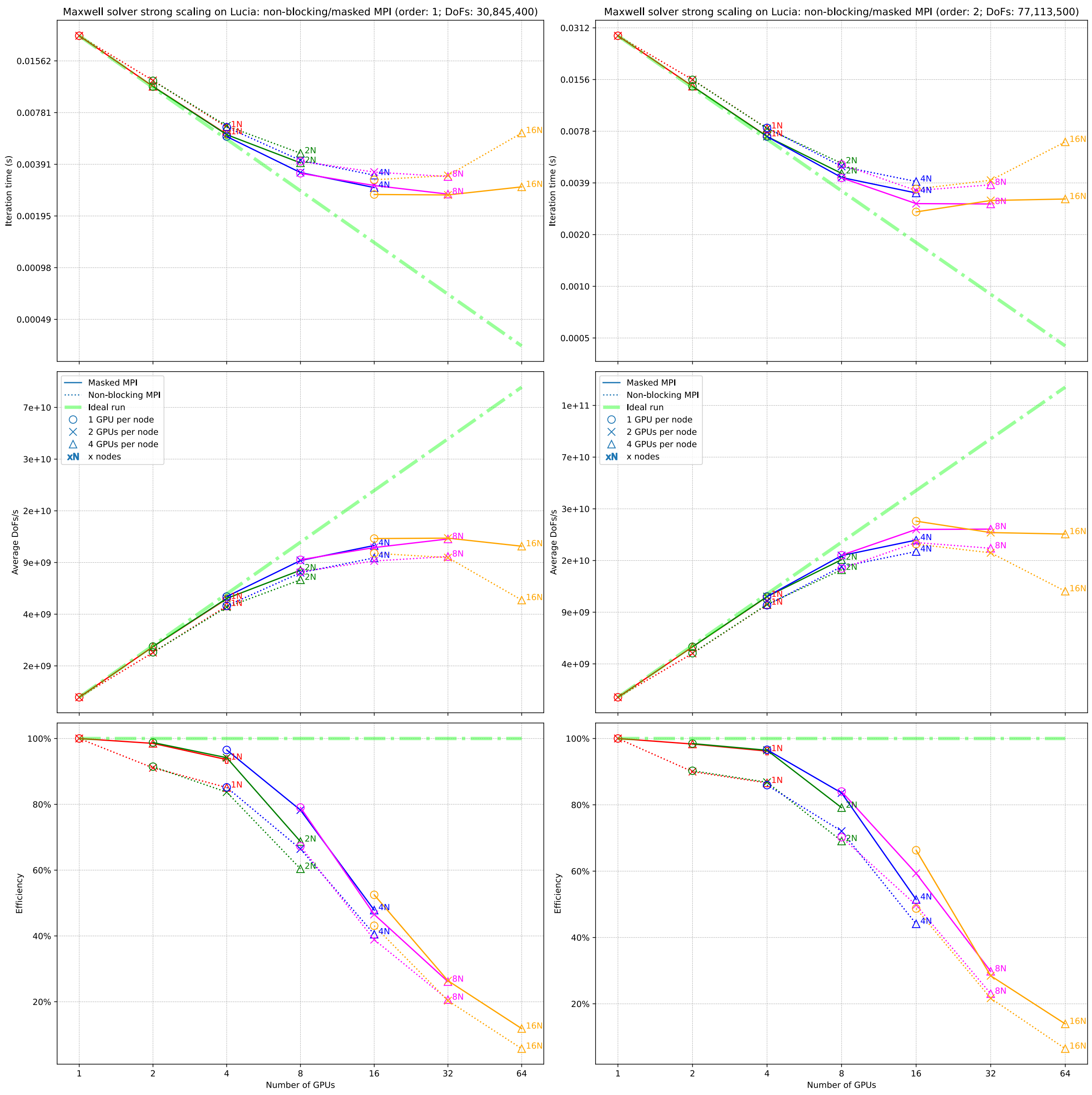


Figure 30 - Strong scaling analysis of the solver on Lucia for a comparison between non-blocking and masked communication (orders 1 & 2). The plots show respectively: iteration time, average Degrees of Freedom (DoFs) per second and efficiency, given a number of GPUs.

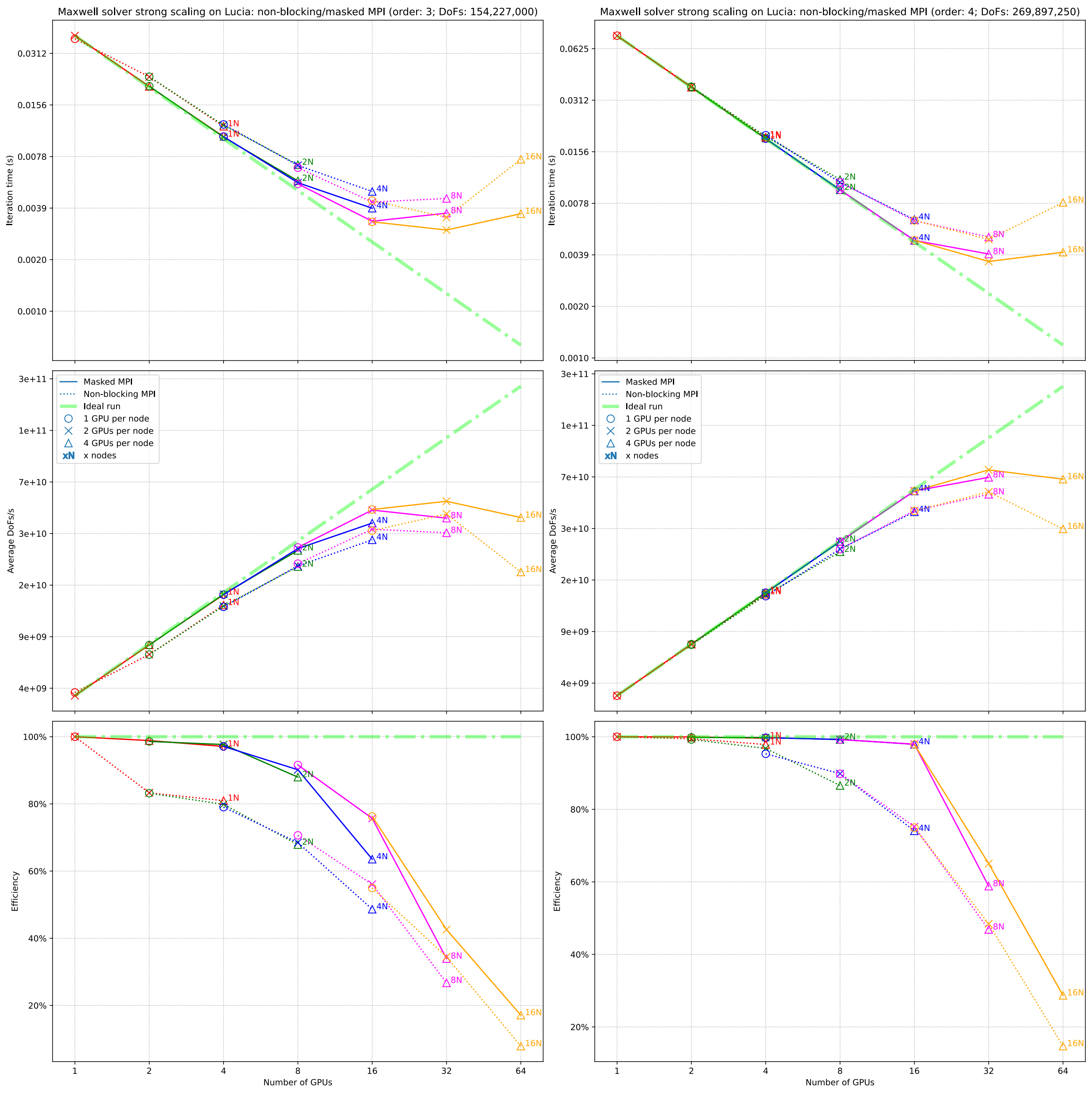


Figure 31 - Strong scaling analysis of the solver on Lucia for a comparison between non-blocking and masked communication (orders 3 & 4). The plots show respectively: iteration time, average Degrees of Freedom (DoFs) per second and efficiency, given a number of GPUs.

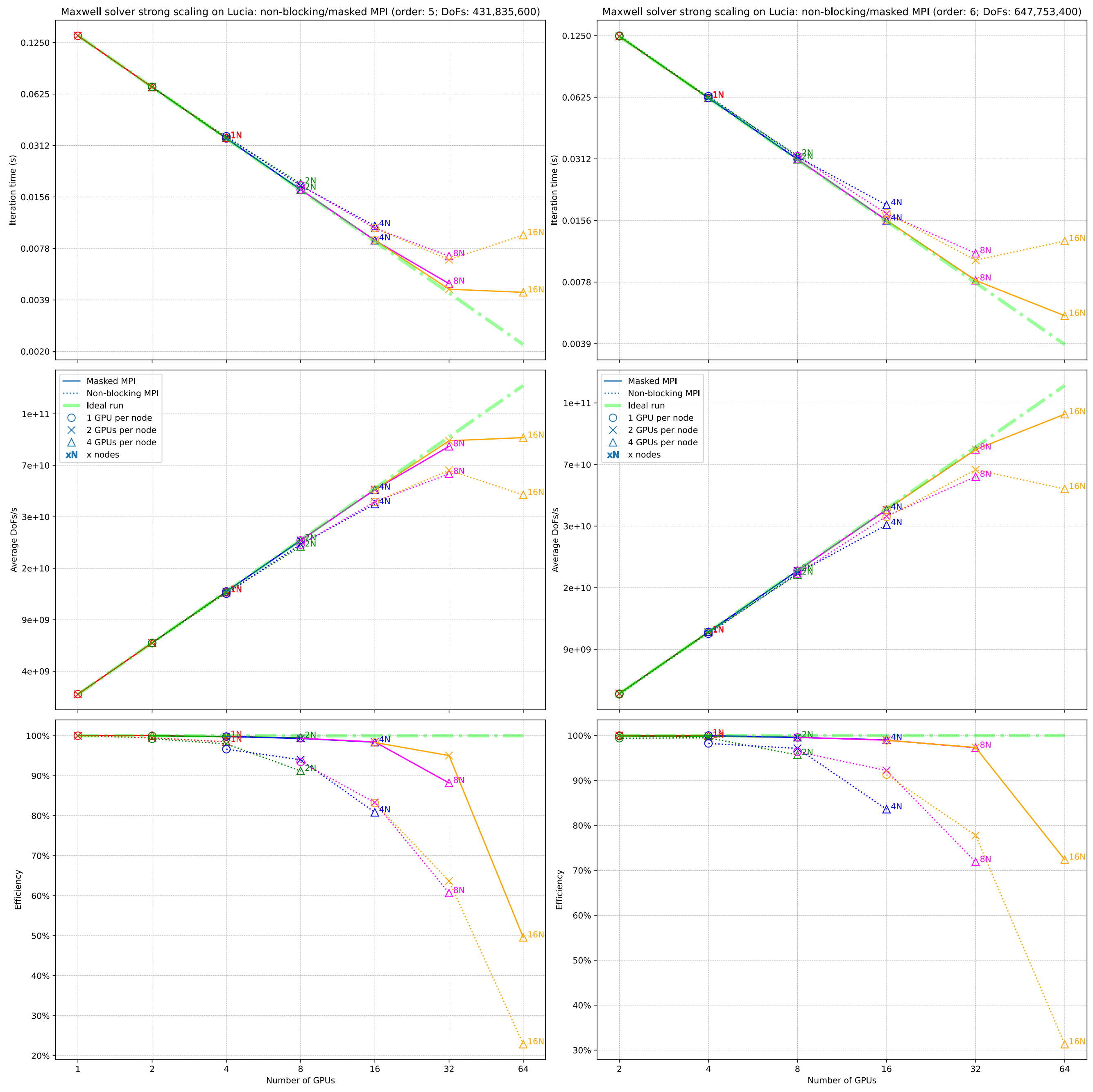


Figure 32 - Strong scaling analysis of the solver on Lucia for a comparison between non-blocking and masked communication (orders 5 & 6). The plots show respectively: iteration time, average Degrees of Freedom (DoFs) per second and efficiency, given a number of GPUs.

8 - Conclusion

In this master's thesis, we started with the Gmsh DG solver developed at ULiège, and refined it in order to answer the question "*To what extent can a Discontinuous Galerkin solver efficiently utilize modern multi-GPU architectures?*". The Figure 32 is a great and brief answer to it showing how the solver performs on large problems. This master's thesis was governed by 4 major stages.

The first step was to benchmark the solver. The ensuing results guided the direction of this work, making it very crucial. For the solver to perform well on multi-GPU systems, we needed to ensure that both the calculation and communication parts are efficient. A roofline model analysis was performed to study the former to find out if any increase in throughput was possible. The latter has been investigated with the help of the NVIDIA Nsight Systems profiler, to better understand what was happening in each iteration under the hood, and see if the communication was either excessive or poorly managed with MPI ranks blocking each others. For both parts, a link with the hardware limitations from the Lucia supercomputer has been established. In the end, we suspected that there was room for improvements in the communication as it presented blocking issues with ranks waiting for each other. As for the calculation part, it showed very limited potential for improvement, and was negligible compared to the communication part.

The second step was to find the bottlenecks of the solver in the code based on our assumptions from previous stage. We found that all the transactions between MPI ranks were "blocking": both the sender and the receiver have to be ready at the same time, otherwise they will wait for the other end until it is ready without doing anything else. Most of the work therefore involved radically modifying their mode of communication, using non-blocking operations that allow data exchanges to take place in the background and one MPI rank to communicate with several peers at the same time. It is important to recall that we need to make sure transactions are done before moving on to other calculations.

The third step was to conduct another performance analysis. This time however, the benchmarks were more extensive, enabling a clear and wide comparison between the initial implementation of the solver and the one featuring non-blocking communication. Their scalability was studied deeply, on the Lucia supercomputer firstly. The results showed a great overall improvement with a significantly reduced communication time, up to 6 times. This translates into greater scalability, bringing us closer to the ideal 100% of efficiency. We confirmed that the computation had really small room for enhancements, unlike communication. Secondly, the strong and weak scaling analyses were performed on another supercomputer: LUMI. This was a vital part of our work, as it was needed to demonstrate the performance of the solver on other architectures (NVIDIA for Lucia, AMD for LUMI), in order to confirm the results as well as to predict its behavior on a different supercomputers.

In the end, the scalability analysis showed interesting results, for both strong and weak scalings. The benchmarks performed on the LUMI supercomputer can be summarized in two key points.

- The overall scalability performed even better than on the Lucia supercomputer, showing that our solver hopefully behaves correctly on any super computer architecture.
- An average performance hit of 30% compared to Lucia, mostly due to the smaller L2 cache of 8 MB from the AMD MI250X GPU, as opposed to 40 MB from the NVIDIA A100.

A fourth and last step was conducted once we found that there was still considerable room for improvement in communication, by masking communication. The order of calculations in the application of the Discontinuous Galerkin operator had to be changed, to compute curls while sending and receiving data. Scalability analyses were then performed to quantify the gains made, but we were running out of time, so we only carried out strong scalings. Unfortunately, we also ran out of computing time on LUMI from previous benchmarks. Therefore, this limited scaling analysis was only performed on Lucia. The results showed another big speed improvement when using numerous GPUs, making our solver even more scalable. Truly high-efficiency executions (i.e. >95%) benefit from a sufficiently long curls calculation, completely masking the communication. The Table 8 provides a brief and precise summary of the performance results for the different versions of the solver.

Of course, there are still ways of optimizing the Gmsh DG solver, as mentioned in Section 6.3. But the two optimizations we brought throughout this work were the ones that led to the biggest performance increase. Our work only focused on the problem resolution part of the solver. But one of its part has been completely ignored until now: the initialization process, which is responsible of partitioning the volume and distributing the work across MPI ranks. In the current state of Gmsh DG solver, it performs really poorly, reaching up to *9* times the time needed to actually solve a problem when using 64 GPUs. It also features data redundancy, making the solver more memory-hungry than necessary. The solver would highly benefit from improved initialization: the current performance of this step is one of the main reasons why we were so limited in our tests on the LUMI supercomputer, as initialization took a large part of our our resource allocation. Fixing the high memory usage would also be profitable to solve larger problems and/or use even more compute nodes: the weak scaling analysis would not be limited to only 16 ranks in some cases.

Appendix

Table of acronyms

DG:	Discontinuous Galerkin
PEC:	Perfect Electric Conductor
PMC:	Perfect Magnetic Conductor
IBC:	Impedance Boundary Condition
PDE:	Partial Differential Equation
SIMT:	Single Instruction Multiple Threads
SIMD:	Single Instruction Multiple Data
SM:	Streaming Multiprocessor
APU:	Accelerated Processing Unit
CC:	Compute Capability
GPGPU:	General-Purpose Computing on Graphics Processing Units
CUDA:	Compute Unified Device Architecture
OS:	Operating System
API:	Application Programming Interface
RAM:	Random Access Memory
DRAM:	Dynamic Random Access Memory
CPU:	Central Processing Unit
GPU:	Graphics Processing Unit
HPC:	High-Performance Computing
NIC:	Network Interface Card
DoFs:	Degrees of Freedom
MPI:	Message Passing Interface
GCD:	Graphics Compute Die
Flops:	Floating Point Operation per Second

List of figures

List of tables

List of listings

Bibliography

[1] M. D'Antonio, “Performance analysis and optimization of a GPU-enabled Discontinuous Galerkin solver,” 2022.

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

[3] L. Angulo, J. Alvarez, M. Pantoja, S. Garcia, and A. Bretones, “Discontinuous Galerkin Time Domain Methods in Computational Electrodynamics: State of the Art,” *Forum for Electromagnetic Research Methods and Application Technologies (FERMAT)*, vol. 10, p. , 2015.

[4] R. 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. in Mathématiques & Applications, vol. 69. Berlin: Springer-Verlag, 2012.

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

[7] N. Corporation, “CUDA Toolkit Documentation 12.9.” Accessed: May 11, 2025. [Online]. Available: https://docs.nvidia.com/cuda/

[8] I. Advanced Micro Devices, “AMD ROCm documentation &#x2014; ROCm Documentation.” Accessed: May 11, 2025. [Online]. Available: https://rocm.docs.amd.com/en/latest/

[9] T. K. G. Inc., “OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems.” Accessed: May 24, 2025. [Online]. Available: https://www.khronos.org/opencl/

[10] T. K. G. Inc., “SYCL - C++ Single-source Heterogeneous Programming for Acceleration Offload.” Accessed: May 24, 2025. [Online]. Available: https://www.khronos.org/sycl/

[11] A. R. Software, “GitHub - ROCm/HIPIFY: HIPIFY: Convert CUDA to Portable C++ Code.” Accessed: Apr. 18, 2025. [Online]. Available: https://github.com/ROCm/HIPIFY

[12] Z. Patt, *The Concepts Behind CUDA Optimization*. 2018. Accessed: Apr. 18, 2025. [Online]. Available: https://corecppil.github.io/Meetups/2018-11-27_FastFurious/TheConceptsBehindCUDAOptimization.pdf

[13] T. O. M. Project, “FAQ: Running CUDA-aware Open MPI.” Accessed: May 12, 2025. [Online]. Available: https://www.open-mpi.org/faq/?category=runcuda

[14] J. Kraus, “An Introduction to CUDA-Aware MPI | NVIDIA Technical Blog.” Accessed: May 12, 2025. [Online]. Available: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/

[15] Lucia, “Cenaero HPC | User Documentation.” Accessed: Apr. 18, 2025. [Online]. Available: https://doc.lucia.cenaero.be/

[16] TOP500, “Lucia - HPE Proliant XL675d Gen10 Plus, AMD EPYC 7513 32C 2.6GHz, NVIDIA A100 SXM4 40 GB, Infiniband HDR, RHEL 8.6 | TOP500.” Accessed: Apr. 18, 2025. [Online]. Available: https://top500.org/system/180122/

[17] Lucia, “Compute Nodes - Cenaero HPC | User Documentation.” Accessed: Apr. 18, 2025. [Online]. Available: https://doc.lucia.cenaero.be/system_details/compute_nodes/

[18] NVidia, *Optimizing Applications for NVidia Ampere GPU Architecture*. 2020. Accessed: Apr. 18, 2025. [Online]. Available: https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf

[19] A. B. Yoo, M. A. Jette, and M. Grondona, “SLURM: Simple Linux Utility for Resource Management,” in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[20] B. Van Straalen, T. Ligocki, S. Williams, and C. Yang, “Empirical Roofline Tool (ERT) v1.1.0.” [Online]. Available: https://doi.org/10.11578/dc.20210423.2

[21] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, “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.

[22] A. Ilic, F. Pratas, and L. Sousa, “Cache-aware Roofline model: Upgrading the loft,” *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014, doi: 10.1109/L-CA.2013.6.

[23] NVidia, “2. Kernel Profiling Guide; NsightCompute 12.8 documentation.” Accessed: Apr. 29, 2025. [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline-charts

[24] A. Li *et al.*, “Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2020, doi: 10.1109/TPDS.2019.2928289.

[25] R. Smith, “NVIDIA Ampere Unleashed: NVIDIA Announces New GPU Architecture, A100 GPU, and Accelerator.” Accessed: May 08, 2025. [Online]. Available: https://www.anandtech.com/show/15801/nvidia-announces-ampere-architecture-and-a100-products

[26] C. Geuzaine and J.-F. Remacle, “Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities,” *International Journal for Numerical Methods in Engineering*, vol. 79, pp. 1309–1331, 2009, doi: 10.1002/nme.2579.

[27] T. O. M. Project, “Open MPI FAQ: Performance analysis tools.” Accessed: May 08, 2025. [Online]. Available: https://www.open-mpi.org/faq/?category=perftools

[28] G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California,” *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 3, pp. 19–20, 2007, doi: 10.1109/N-SSC.2007.4785615.

[29] J. L. Gustafson, “Reevaluating Amdahl's law,” *Commun. ACM*, vol. 31, no. 5, pp. 532–533, 1988, doi: 10.1145/42411.42415.

[30] anni.jakobsson@csc.fi, “LUMI’s full system architecture revealed - LUMI.” Accessed: Apr. 16, 2025. [Online]. Available: https://www.lumi-supercomputer.eu/lumis-full-system-architecture-revealed/

[31] TOP500, “LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11 | TOP500.” Accessed: Apr. 16, 2025. [Online]. Available: https://top500.org/system/180048/

[32] LUMI, “LUMI supercomputer - LUMI.” Accessed: Apr. 16, 2025. [Online]. Available: https://www.lumi-supercomputer.eu/lumi_supercomputer/

[33] LUMI, “Documentation - GPU nodes - LUMI-G.” Accessed: Apr. 16, 2025. [Online]. Available: https://docs.lumi-supercomputer.eu/hardware/lumig/

[34] LUMI, “LUMI training materials - LUMI Architecture.” Accessed: Apr. 17, 2025. [Online]. Available: https://lumi-supercomputer.github.io/LUMI-training-materials/1day-20230921/01_Architecture/