Multi-GPU Discontinuous Galerkin Solver for Maxwell's Equations

Orian Louant, Matteo Cicuttin, Clément Smagghe
and Christophe Geuzaine

Applied and Computational Electromagnetics

9th International Conference on Advanced Computational Methods in Engineering and Applied Mathematics

Ghent, Belgium — September 17, 2025

Introduction — GmshDG

GmshDG[1] is multi-GPUs nodal Discontinuous Galerkin solver based of Gmsh[2]:

- Currently only support Maxwell equations but will be extended in the future
- Supports both major GPUs vendors (NVIDIA and AMD)
- Still in early development

Goal

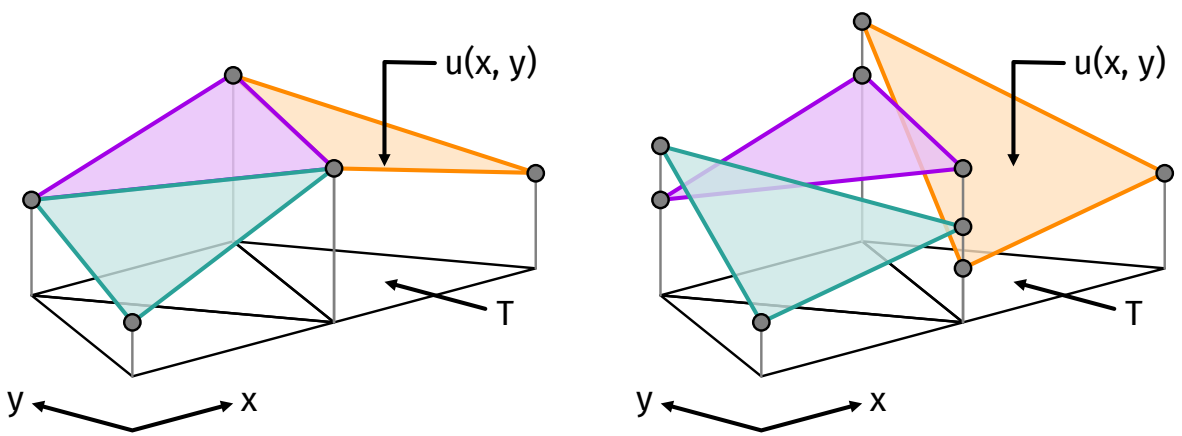
Solve large problem with tenth of billions of degrees of freedom

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

Introduction — discontinuous Galerkin

The Discontinuous Galerkin (DG) method is a numerical technique for solving PDEs that combines features of the finite element and finite volume methods.

- DG represents the solution using basis functions defined on individual elements T of a mesh.
- It does not enforce continuity of the solution u across element boundaries.



Continuous **Discontinuous**

Introduction — nodal discontinuous Galerkin

- Fields are approximated using nodal basis functions located at nodal points
- Within each T element, the fields are represented as

$$\boldsymbol{E}^T(\boldsymbol{x}, t) \approx \sum_{i=0}^{N_p} \boldsymbol{E}_i^T(t)\phi_i(\boldsymbol{x})$$

where ϕ_i are the basis functions and N_p is the number of nodes per elements.

Starting from the Maxwell equations for the electric field, ignoring the source terms:

$$\varepsilon \frac{\partial \boldsymbol{E}}{\partial t} = \nabla \times \boldsymbol{H}$$

and by multiplying by the test function ϕ_i and integrate over the elements

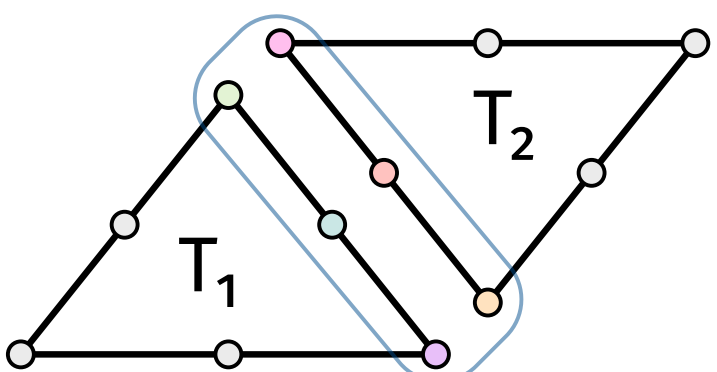
$$\int_T \varepsilon \frac{\partial \boldsymbol{E}}{\partial t} \phi_i dV = \int_T (\nabla \times \boldsymbol{H}) \phi_i dV$$

then, we integrate by parts and apply the divergence theorem, we get

$$\int_T \varepsilon \frac{\partial \boldsymbol{E}}{\partial t} \phi_i dV = \underbrace{\int_{\partial T} (\boldsymbol{n} \times \boldsymbol{H}) \phi_i dS}_{\text{Surface contribution}} - \underbrace{\int_T \boldsymbol{H} \cdot (\nabla \times \phi_i) dV}_{\text{Volume contribution}}$$

The same process can be used to get an expression for the magnetic field.

- The surface contribution $\int_{\partial T}(\boldsymbol{n}\times\boldsymbol{H})\phi_i dS$ is not well defined in DG because of the discontinuities at the interface ∂T
- To solve the problem, numerical fluxes are introduced to ensure unique interface value, stability and conservation

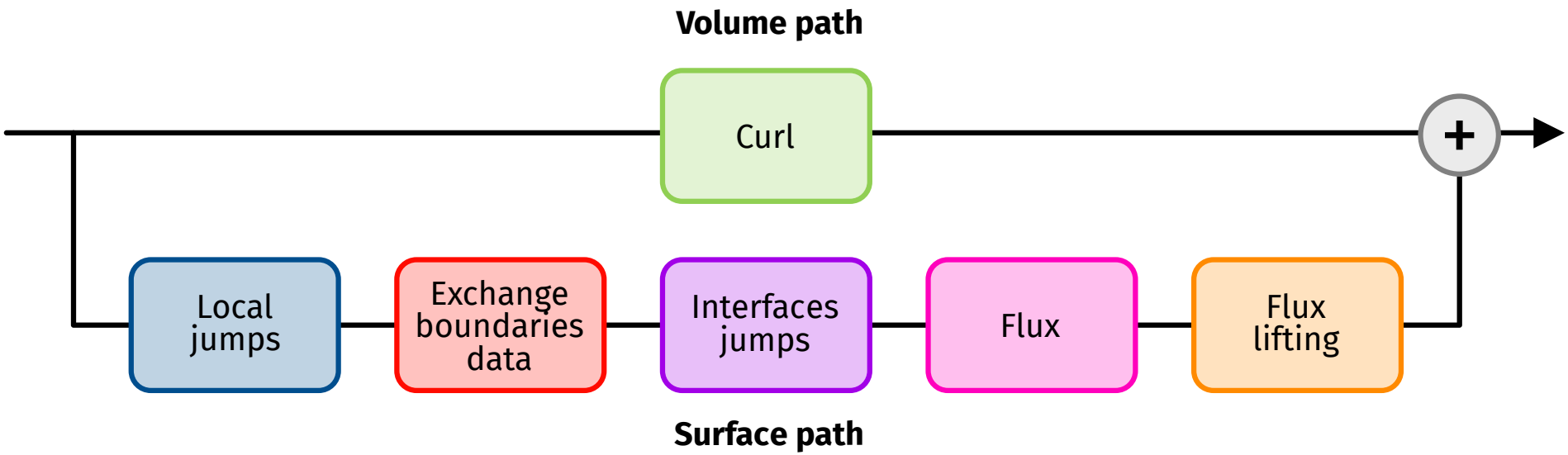


The computation of the fluxes require the jumps. The jump operator is defined as

$$[\![\boldsymbol{u}]\!] = \boldsymbol{u}^+ - \boldsymbol{u}^-$$

where $\boldsymbol{u}^+$ and $\boldsymbol{u}^-$ denote the values of the vector field $\boldsymbol{u}$ on the face shared by elements T^+ and T^-

Introduction — computational path



The exchange data between neighbours is only required for the multi-GPU setup. In that case, the mesh is partitioned using the METIS graph partitioner via GMSH

Goal

Solve large problems with tenth of billions of degrees of freedom

From the characteristics of the DG method, we expect to be able to efficiently use GPU compute power and scale to 100+ GPUs

- **Single GPU:** each nodal point can compute its own contribution, well suited for massively parallel architectures like GPUs.
- **Multi-GPUs strong and weak scaling:** large portions of the calculations are local but we need to minimize the impact of the exchange of data jor the jumps if we want to achieve good parallel performance.

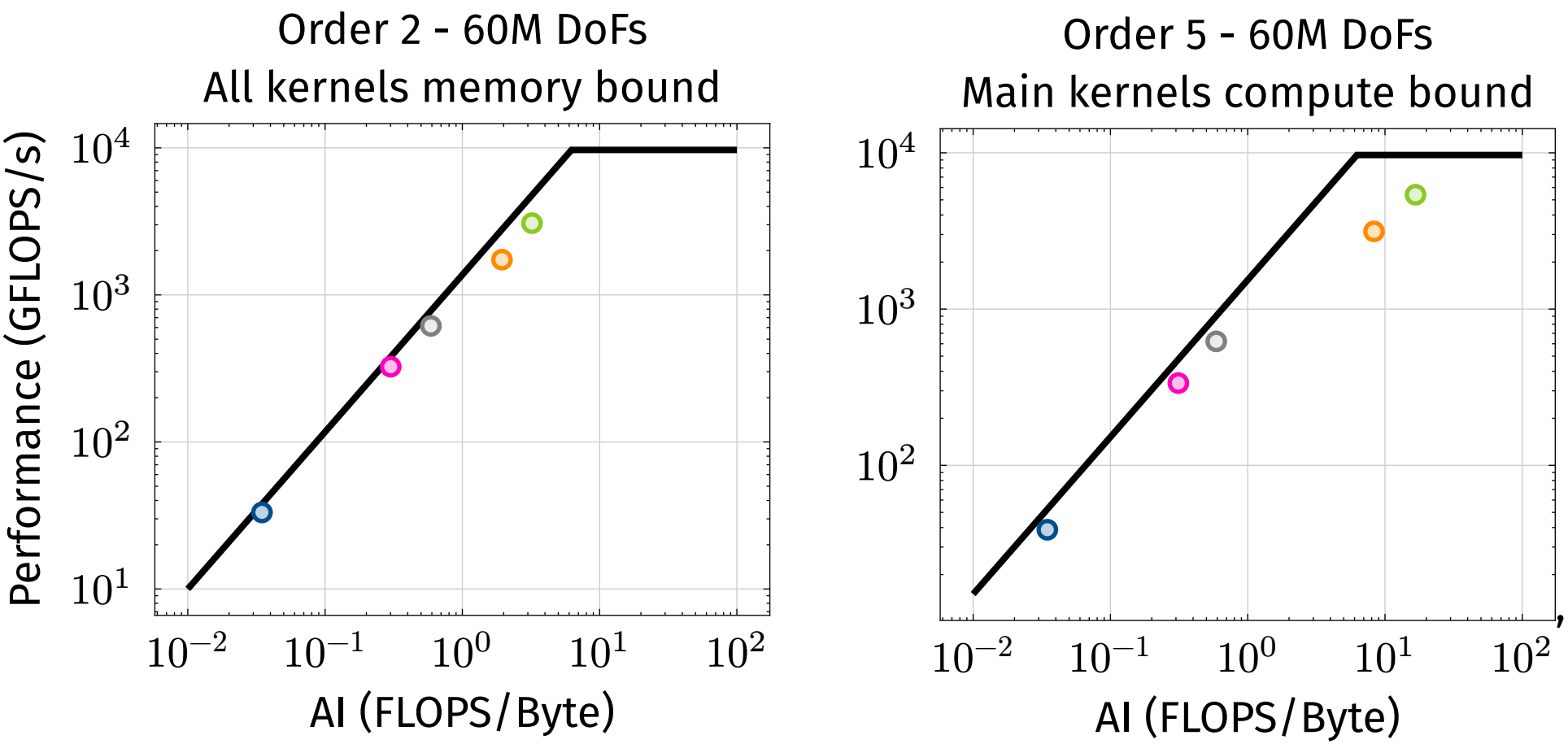
Single-GPU performance

Single GPU performance – Depending to the polynomial order

Order	**Total iteration time** (s)	**Time per iteration** (ms)	**Throughput** (GDoFs/s)
1	251.027	50.195	1.21
2	121.076	24.210	2.51
3	79.442	15.885	3.75
4	67.062	13.410	4.50
5	66.653	13.328	4.50
6	82.288	16.454	3.73

Single NVIDIA A100 GPU - constant problem size (60M DoFs) - 5 000 iterations

Single GPU performance — Roofline plot



Single GPU performance — GPU *vs* CPU throughput

Order	**Full CPU node throughput** (GDoFs/s)	**Single GPU throughput** (GDoFs/s)	**Speedup**
1	0.82	1.21	1.48
2	1.01	2.51	2.49
3	1.15	3.75	3.26
4	1.17	4.50	3.84
5	1.17	4.50	3.84
6	1.17	3.73	3.18

Single NVIDIA A100 GPU vs 2x AMD EPYC 7763 64-Core CPU
constant problem size (60M DoFs) - 5 000 iterations

Single GPU performance — GPU *vs* CPU power and CO_2 emissions

It's important to consider the simulation power use as it impact the economic costs (electricity cost to run the computation and cool the data center) as well as the environmental footprint.

	Energy per timestep (J)	**Average Power** (W)	**CO_2eq per hour**[3] (g)
CPU	25.147	526	58
GPU	4.026	302	33

Single NVIDIA A100 GPU vs 2x AMD EPYC 7763 64-Core CPU - 60M DoFs

[3] https://www.nowtricity.com : the average for the Belgian energy mix in 2024 was 110 gCO_2eq/kWh

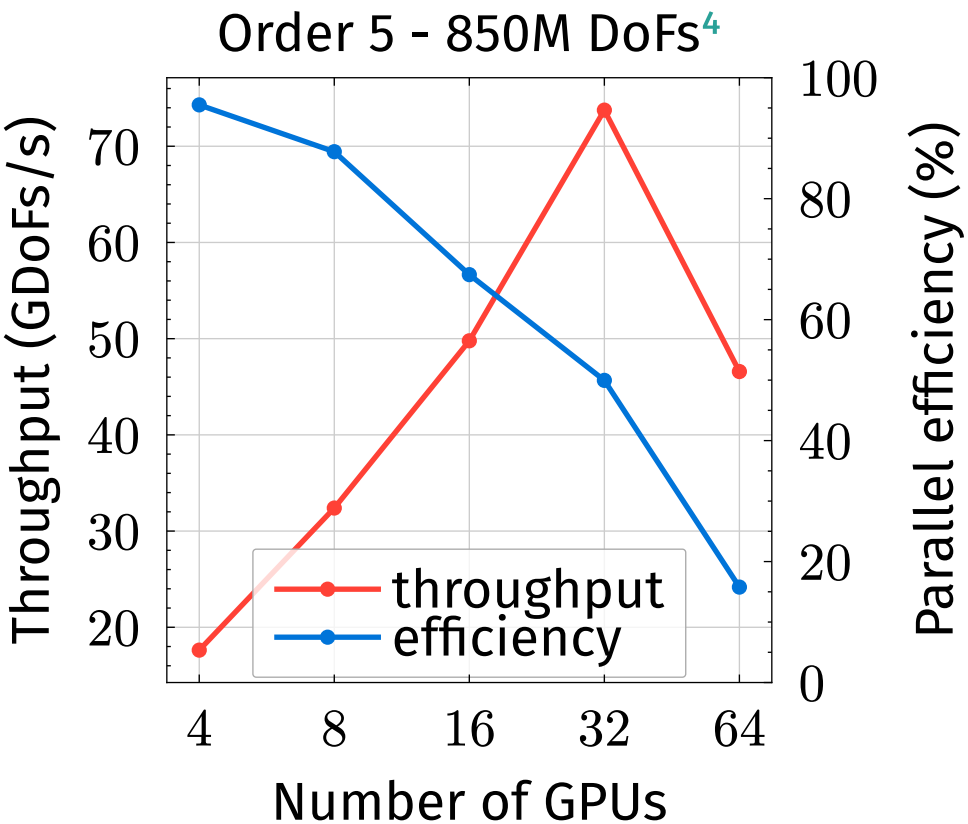
Multi-GPU performance

Multi-GPU — communication algorithm overview

The multi-GPU implementation leverages MPI, assigning one rank per GPU, to handle data exchange between neighbors, which is necessary for computing *jumps*.

```
EXCHANGE_BOUNDARY_DATA():
1 for neighbor in NeighborsList:
2     indexes ← neighborIndexes[neighbor]
3     packedDataSend ← PACK_BOUNDARY_DATA(fields, indexes)
4
5     SEND(packedDataSend, neighbor)
6     RECV(packedDataRecv, neighbor)
7
8     COMPUTE_JUMPS(fields, indexes)
```

Multi-GPU — initial implementation strong scaling

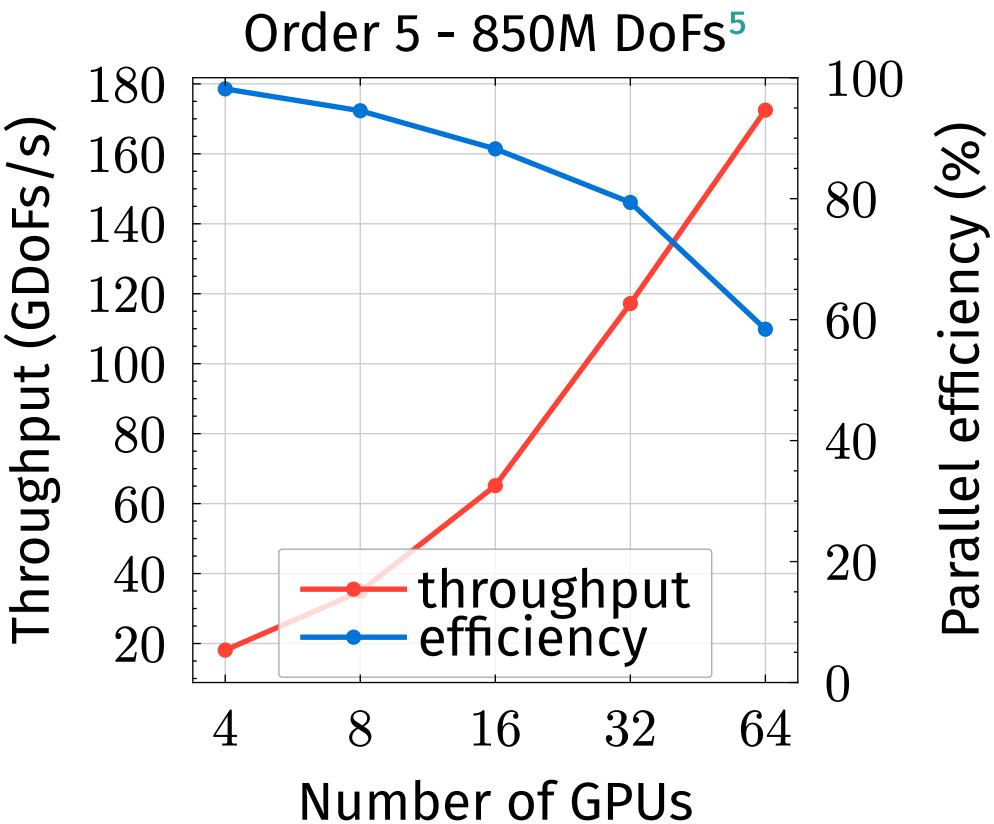


- Simple “naive” implementation using MPI blocking Send and Recv
- The parallel efficiency quickly drop bellow 80% (16 GPUs)
- Throughput don’t improve when using 64 GPUs

Switch to MPI non-blocking

[4] Lucia - 4x NVIDIA A100 40GB GPUs - 1x AMD EPYC 7513 32-Core CPU - 256 GB of RAM, 2x 200 Gbps Infiniband NICs

Multi-GPU — non-blocking MPI communication strong scaling



- Massive improvement compared to the blocking implementation
- 80% parallel efficiency up to 32 GPUs
- Throughput keep improving when using 64 GPUs

❌ **Still, this is not enough to use more than 100+ GPUs efficiently**

[5] Lucia - 4x NVIDIA A100 40GB GPUs - 1x AMD EPYC 7513 32-Core CPU - 256 GB of RAM, 2x 200 Gbps Infiniband NICs

Multi-GPU — scalability limitations

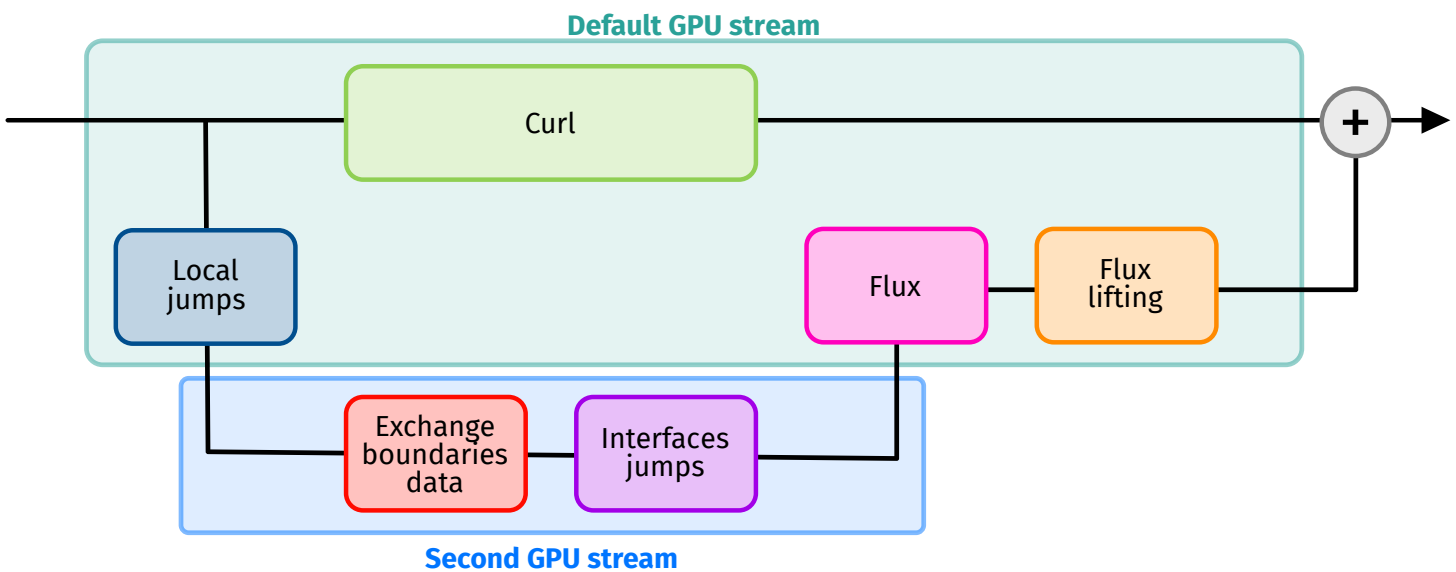
The limit to the scalability has two main reasons:

- As we increase the number of GPUs, the number of DoFs assigned to each GPU decreases, leaving fewer computations per device. This underutilization prevents the hardware from being fully saturated, with the impact being most pronounced in the compute-intensive *Curl* and *Lifting* kernels
- As the number of GPUs increases, the communication-to-computation ratio also rises

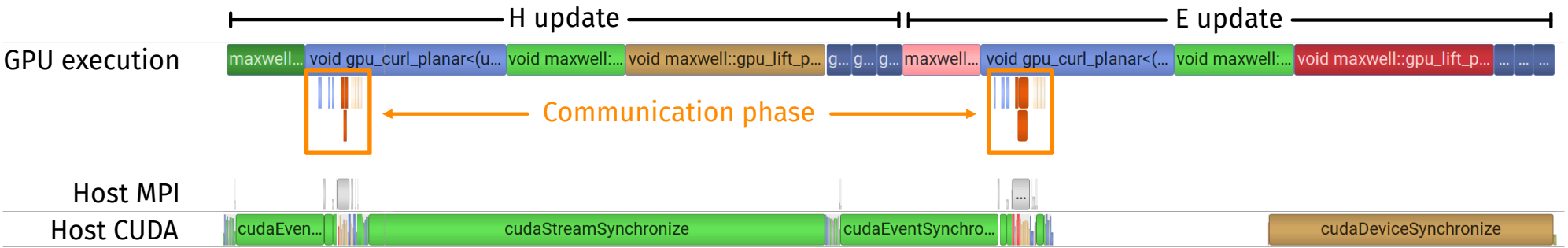
➡️ While the first limitation can be overcome with a larger problem size, we need to address the second limitation

Multi-GPU — computational path for hiding communication

- If we want to be able to scale to a large number of GPUs, we need to hide the communication
- As the volume and surface path are independent, we can overlap the communication with the execution of the *Curl* kernel

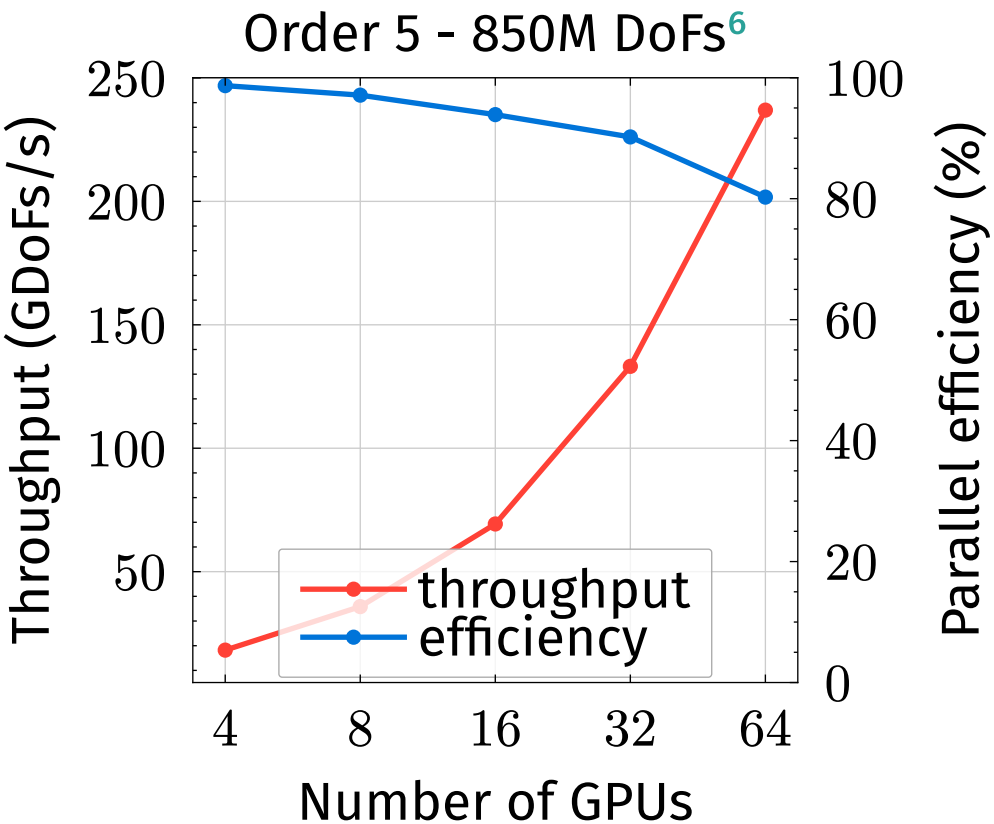


Multi-GPU — execution profile for hidden communication



- The profiling confirms that the communication is overlapped with the execution of the *Curl* kernel
- The execution time of the *Curl* kernel is sufficient to completely mask the communication

Hiding communication - Strong scaling



- Hiding communication improves scaling and the improvement is more visible as we increase the number of GPUs
- Using 64 GPUs, overlapping the communication with the *Curl* kernel execution allows to go from 172 to 236 GDoFs/s

[6] Lucia - 4x NVIDIA A100 40GB GPUs - 1x AMD EPYC 7513 32-Core CPU - 256 GB of RAM, 2x 200 Gbps Infiniband NICs

AMD GPUs: Implementation and performance

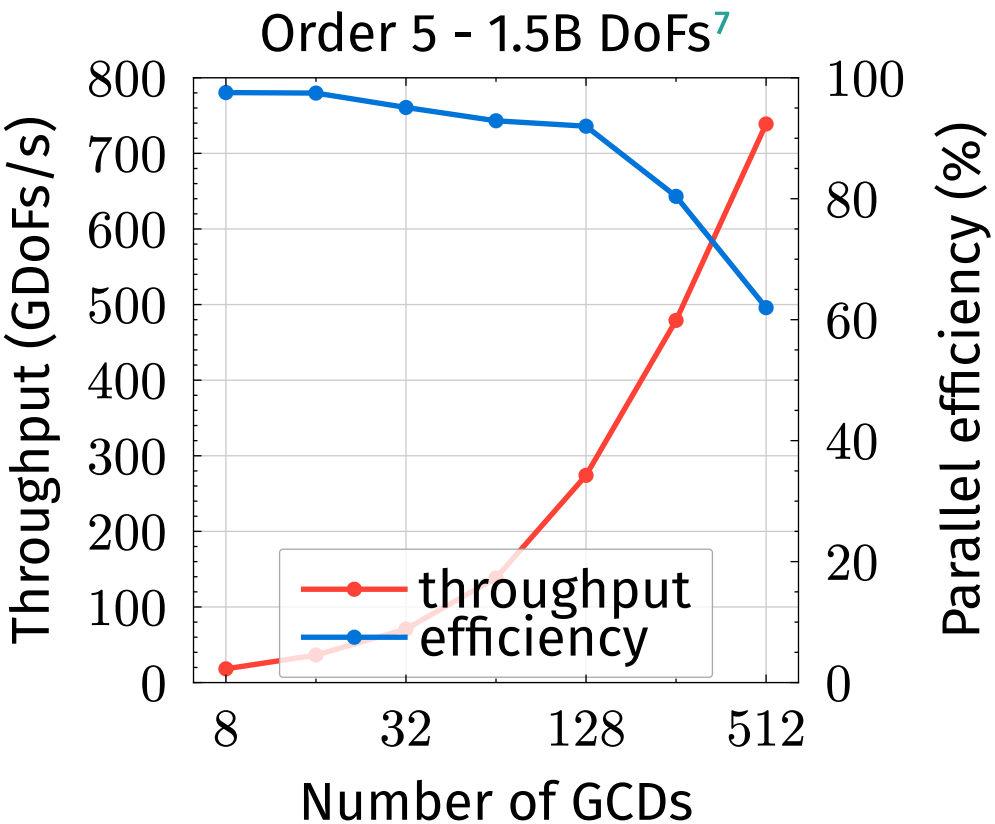
Same implementation for AMD GPUs is the same as the on for NVIDIA GPUs:

- Use of macros to translate API the CUDA and HIP API calls
- Kernels are exactly the same

```
#ifdef USE_HIP
  #define gpuMalloc hipMalloc
#else
  #define gpuMalloc cudaMalloc
#endif
```

- The throughput on AMD MI250x GPUs is about ~50% of what is achieved on a NVIDIA A100: 2.33 GDoFs/s *vs* 4.61 GDoFs/s
- However, we have easy access to a cluster with thousands of AMD MI250x GPUs

AMD GPUs - LUMI strong scaling



- Efficiency > 90% up to 128 GCDs[8]
- After 128 GCDs the amount of computation is not sufficient to hide communication and the parallel efficiency starts to drop

[7] LUMI - 4x AMD MI250x 128GB GPUs - 1x AMD EPYC 7A53 64-Core CPU - 512 GB of RAM - 4x 200 Gbps Slingshot NICs

[8] GCD = Graphics Compute Die. AMD MI250x modules have 2 GCDs with 64 GB of memory each

Key Takeaways

- Using GPUs instead of CPUs for a DG Maxwell solver delivers significant improvements in both throughput and energy efficiency, especially at higher polynomial order
- The strong locality of the DG method makes it well-suited for multi-GPU implementations, as it does not require a lot of communication between devices
- Good scaling performance can be achieved by overlapping the communication phase with the execution of a compute kernel

Thank you for your attention

Contact: orian.louant@uliege.be

www.enmieux.be

Cofinancé par l'Union européenne

Wallonie

We acknowledge LUMI-BE for awarding this project access to the LUMI supercomputer through a Development Access call

LUMI-BE is joint effort from BELSPO, SPW Économie, Emploi, Recherche, Department of Economy, Science & Innovation and Innoviris

The present research benefited from computational resources made available on Lucia, the Tier-1 supercomputer of the Walloon Region, infrastructure funded by the Walloon Region under the grant agreement n°1910247