IMPLEMENTATION AND OPTIMIZATION OF A MULTI-GPU DISCONTINUOUS GALERKIN SOLVER FOR MAXWELL'S EQUATIONS

Orian Louant, Matteo Cicuttin, Clément Smagghe
and Christophe Geuzaine

LIÈGE université

Applied and Computational Electromagnetics

3rd LUMI-BE User Day

Brussels, Belgium — December 17th, 2025

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

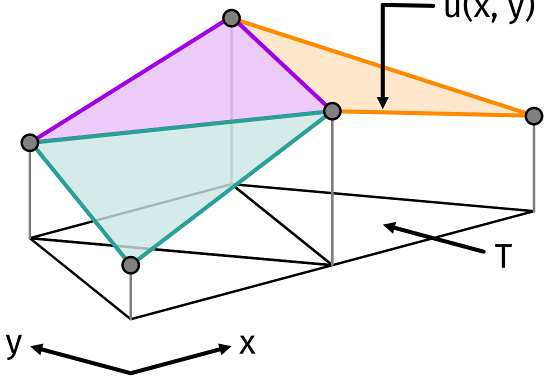
Solve large problem with tenth of billions of degrees of freedom

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

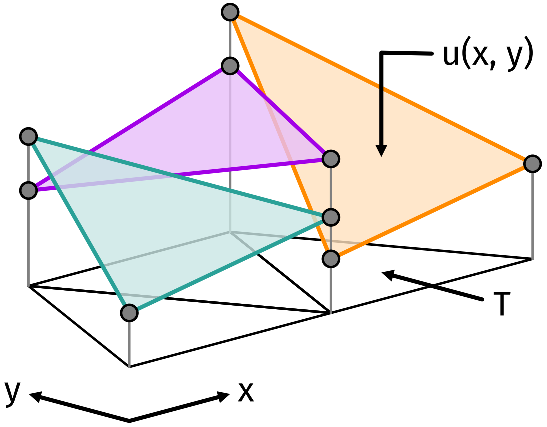
Discontinuous Galerkin

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

Nodal discontinuous Galerkin

In the Discontinuous Galerkin formulation, the solution within each element is represented using a nodal approximation

- 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. The coefficients $E_i^T(t)$ correspond to the values of the field at the nodal points

Nodal discontinuous Galerkin

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

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

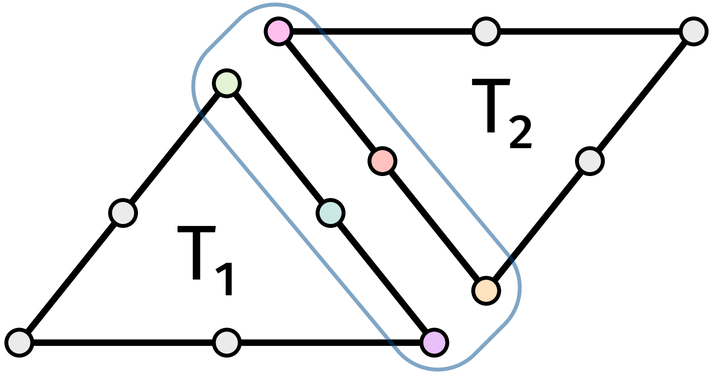
then, we integrate by parts and apply the divergence theorem, we get

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

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

Handling the interfaces

- 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

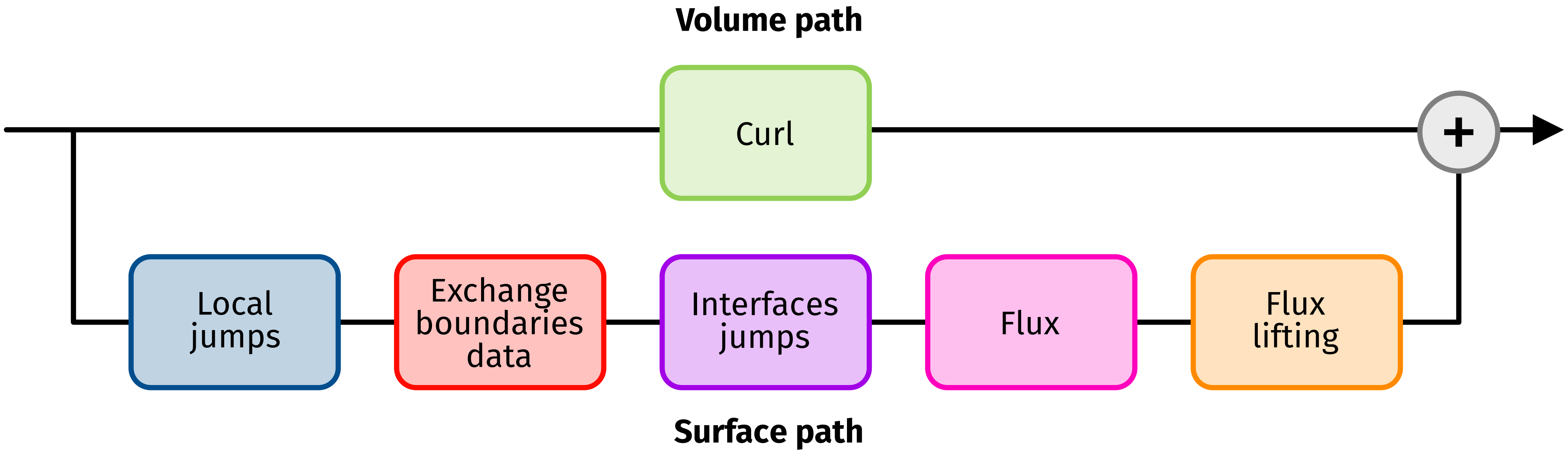


The computation of the fluxes requires the jumps. The jump operator is defined as

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

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

Computational path



The exchange of 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

Implementation choices

The current implementation uses custom, from-scratch kernels rather than established linear algebra libraries. The code runs on both CPUs and GPUs, and the same implementation is used for NVIDIA and AMD GPUs:

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

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

Kokkos has been tested to achieve portability on both CPU and GPU, but the results were mixed: while GPU performance was comparable to native CUDA/HIP, CPU performance was disappointing due to insufficient vectorization

Revisiting the goal

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.

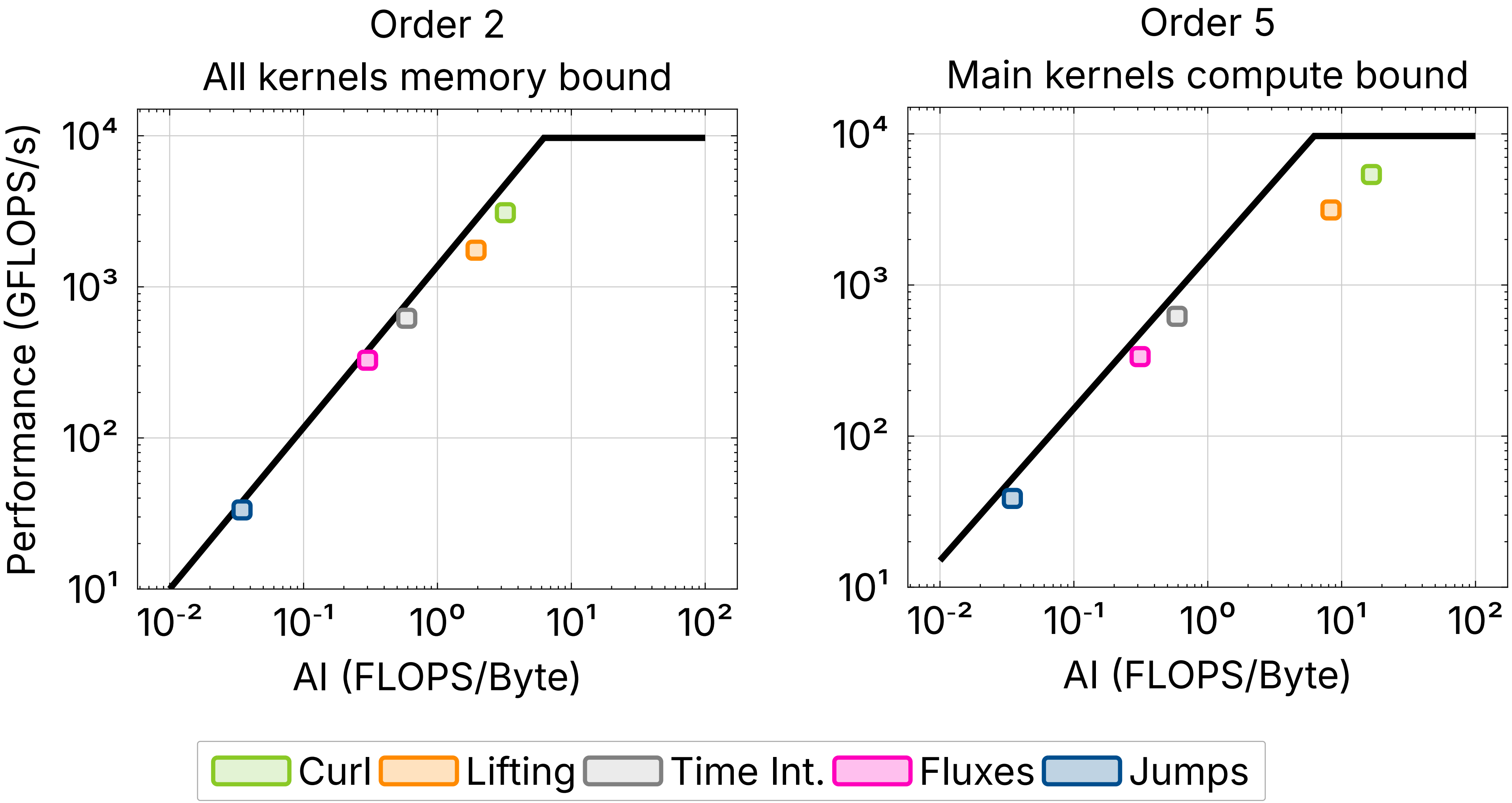
Single-GPU performance

GPU *vs* CPU throughput

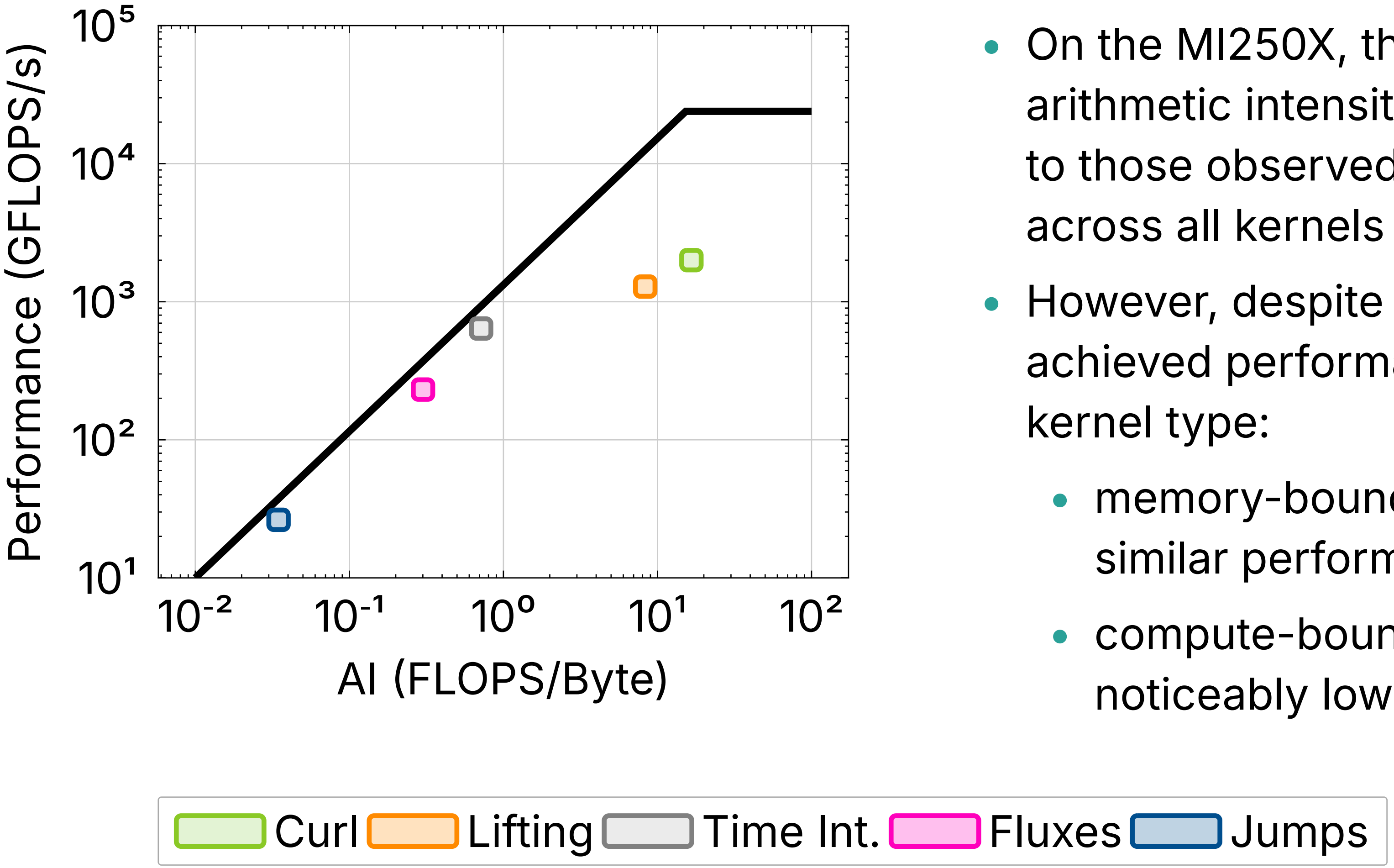
Order	Full CPU node throughput (GDoFs/s)	NVIDIA A100 throughput (GDoFs/s)	AMD MI250X throughput (GDoFs/s)	NVIDIA H100 throughput (GDoFs/s)
1	0.82	1.21 (1.5x)	2.46 (3.0x)	1.83 (2.2x)
2	1.01	2.51 (2.5x)	4.33 (4.3x)	3.55 (3.5x)
3	1.15	3.75 (3.3x)	5.60 (4.9x)	5.15 (4.5x)
4	1.17	4.50 (3.8x)	5.12 (4.4x)	5.88 (5.0x)
5	1.17	4.50 (3.8x)	4.63 (4.0x)	5.82 (5.0x)
6	1.17	3.73 (3.2x)	3.81 (3.2x)	5.23 (4.5x)

CPU: *Lucia (2x AMD EPYC 7763) —* ***NVIDIA A100:*** *Lucia —* ***NVIDIA H100:*** *Marenostrum 5 —* ***AMD MI250X:*** *LUMI*
Constant problem size (60M DoFs) - 5 000 time steps

Roofline plot — NVIDIA A100

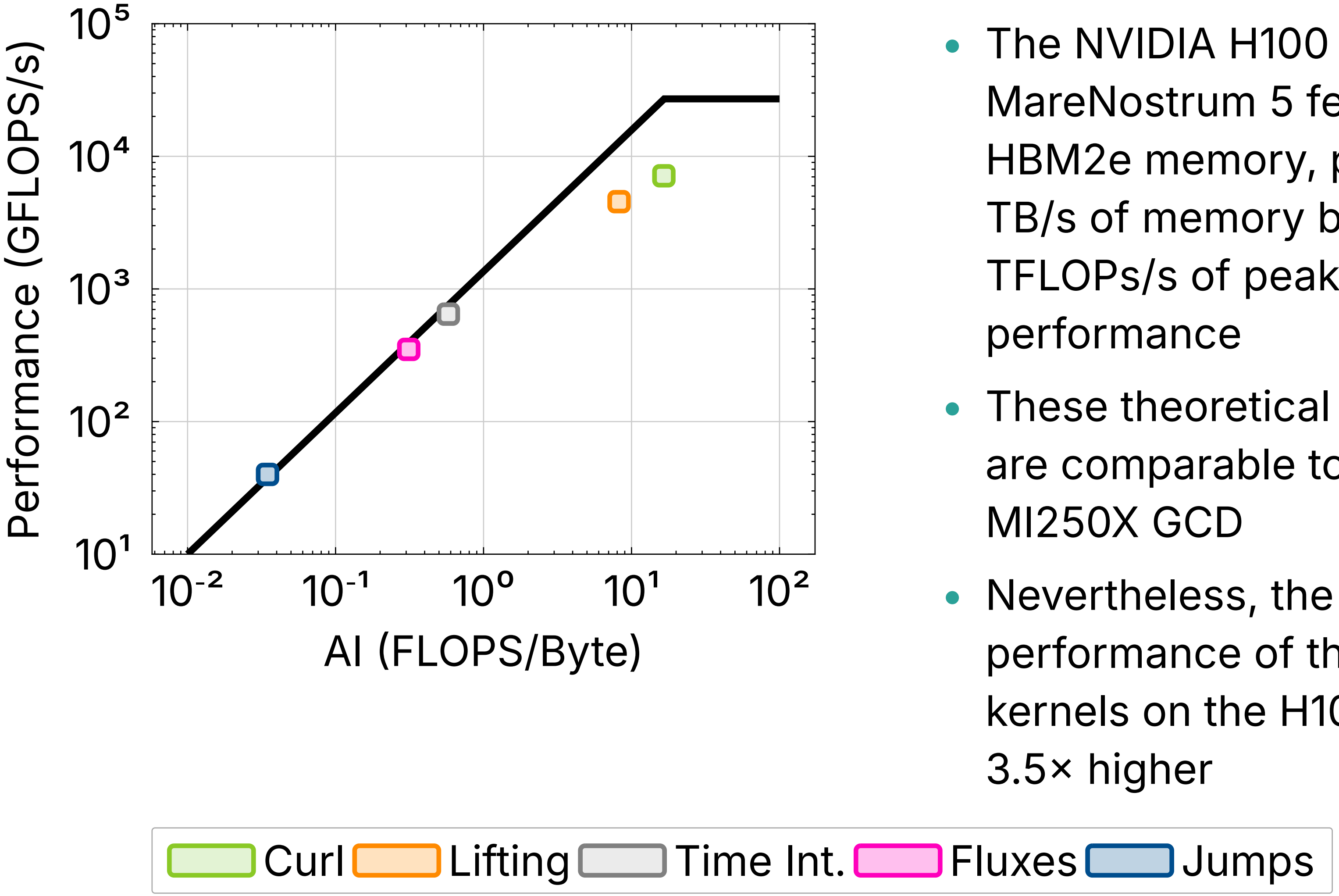


Roofline plot — AMD MI250X



- On the MI250X, the measured arithmetic intensities are comparable to those observed on A100 GPUs across all kernels
- However, despite this similarity, the achieved performance differs by kernel type:
  - memory-bound kernels exhibit similar performance levels
  - compute-bound kernels show noticeably lower performance

Roofline plot — NVIDIA H100



- The NVIDIA H100 GPUs on MareNostrum 5 feature four stacks of HBM2e memory, providing up to 1.6 TB/s of memory bandwidth and 27 TFLOPs/s of peak compute performance
- These theoretical performance figures are comparable to those of a single MI250X GCD
- Nevertheless, the measured performance of the *flux* and *curl* kernels on the H100 is approximately 3.5× higher

Does the performance difference make sense?

To explain the performance differences between NVIDIA and AMD hardware, we need to consider the following:

- The curl and lifting kernels are the main performance bottlenecks, accounting for approximately 60% of the total execution time
  - The curl kernel computes spatial derivatives by applying differentiation matrices to the degrees of freedom within each element
  - The lifting kernel applies precomputed lifting operators to the fluxes to incorporate interface contributions into the element volume

In practice, both kernels largely reduce to dense matrix–vector or matrix–matrix multiplications with small, fixed-size matrices: **data reuse is the key to get good performance**

Sorry, I lied... We are not compute bound

By investigating further, we can conclude that our performance bottleneck is the L1 cache. For example, for the *curl* kernel:

- NVIDIA GPUs benefit from significantly larger L1 caches, higher L1 bandwidth, and higher hit rates
- In contrast, the MI250X, with much smaller L1 size, exhibits lower arithmetic intensity, and a reduced L1 hit rate. This indicates less data reuse and results in substantially lower achieved throughput

	L1 Size (kiB)	**L1 BW** (TB/s)	**L1 AI** (FLOPS/Bytes)	**L1 hit rate** (%)	**Throughput** (TFLOPS/s)
A100	192	15.1	0.37	95	6.126
H100	256	27.0	0.37	96	8.661
MI250X	16	11.9	0.24	81	2.354

We should be able to improve performance on LUMI

While we have not yet achieved performance on the MI250X that is close to the GPU theoretical capabilities, we plan to rework the flux and curl kernels in the near future.

Based on the numbers presented on the previous slide, we expect to achieve significantly better performance on LUMI by leveraging the Local Data Share (LDS). For example, for the *curl* kernel on a single GCD, the achievable performance can be estimated as

$$0.37\ \mathrm{FLOPS/Byte} \cdot 23.9\ \mathrm{TB/s} = 8.8\ \mathrm{TFLOPS}$$

which is comparable to the performance observed on the NVIDIA H100

GPU *vs* CPU power consumption and CO_2 emissions

In addition to the compute performance, it's also 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)
2x AMD EPYC 7763	25.147	526	58
1x NVIDIA A100	4.026	302	33
1x NVIDIA H100	4.540	442	48
1x AMD MI250X	6.620	409	45

60M DoFs - order 5 - 5 000 time steps

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

Multi-GPU performance

Communication algorithm overview

The only part of the code that requires data exchange between GPUs is the computation of numerical fluxes, and in particular the evaluation of the jumps across element interfaces

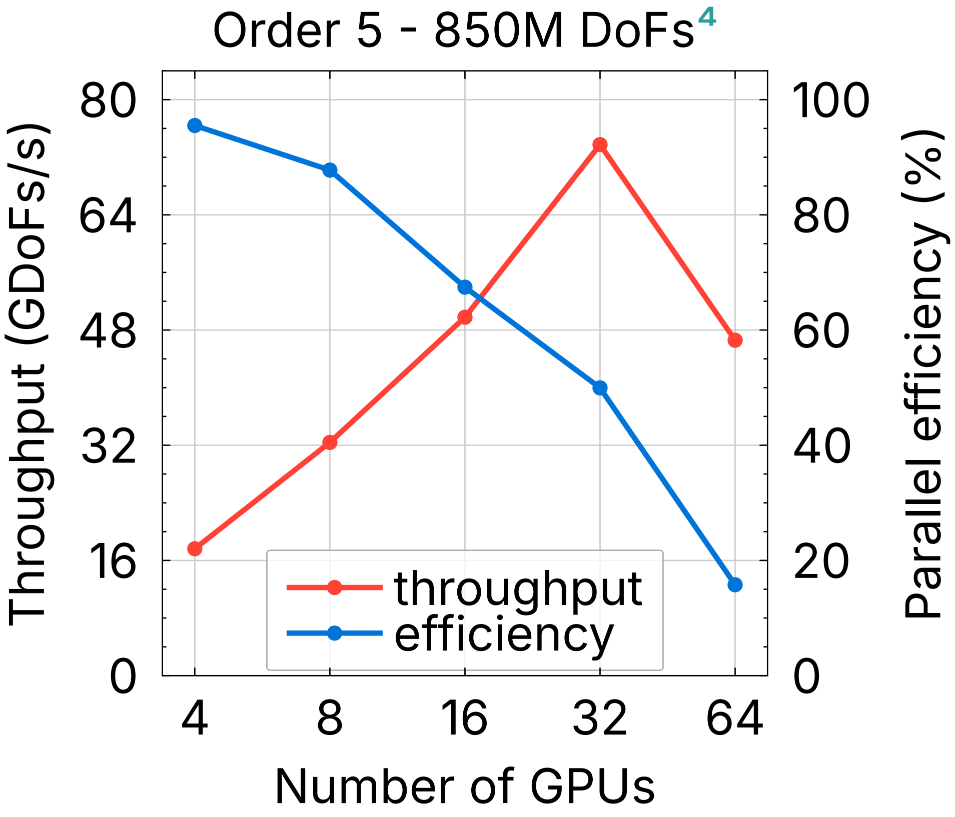
The multi-GPU implementation leverages MPI to handle the exchange of boundary data between neighboring subdomains:

```
exchange_boundary_data():
for neighbor in NeighborsList:
    indexes $\leftarrow$ neighborIndexes[neighbor]
    packedDataSend $\leftarrow$ pack_boundary_data(fields, indexes)

    Send(packedDataSend, neighbor)
    Recv(packedDataRecv, neighbor)

    compute_jumps(fields, indexes)
```

Initial implementation strong scaling

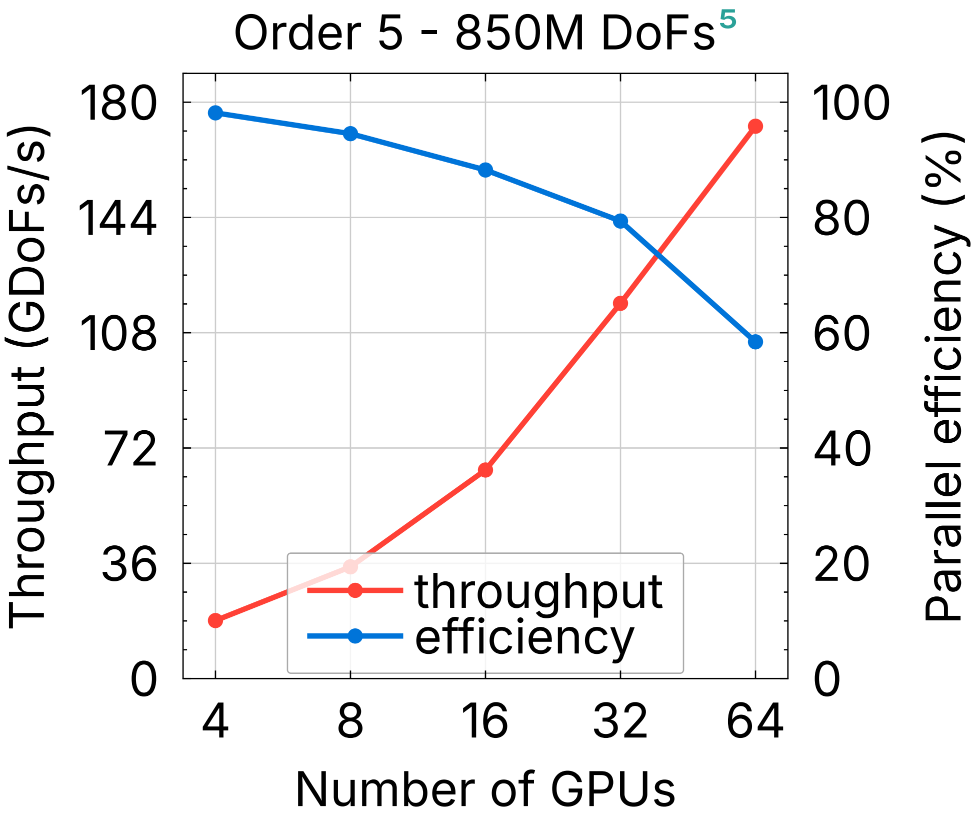


- Simple "naive" implementation using MPI blocking Send and Recv
- The parallel efficiency quickly drops below 80% (16 GPUs)
- Throughput doesn't improve when using 64 GPUs

➡ **Switch to MPI non-blocking**

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

Introducing non-blocking communication



Order 5 - 850M DoFs[5]

- Massive improvement compared to the blocking implementation
- 80% parallel efficiency up to 32 GPUs
- Throughput keeps 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, 2× 200 Gbps Infiniband NICs

Scalability limitations

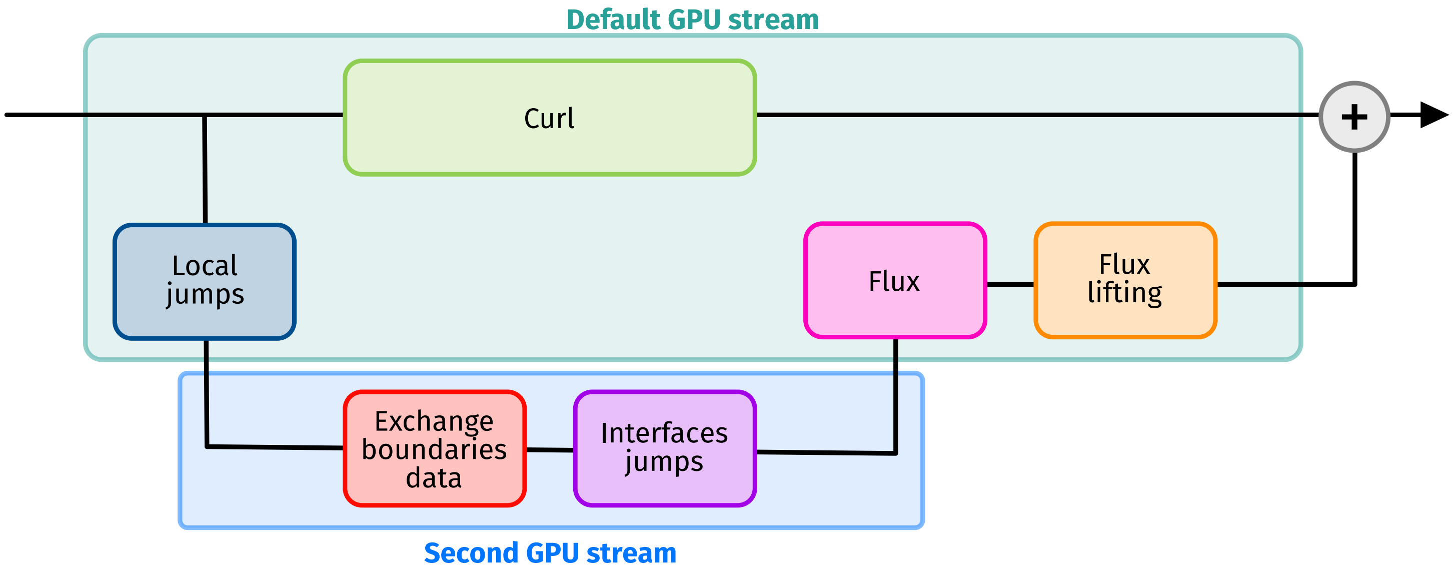
The limit to the scalability has two main reasons:

- 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

A 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



Strategy for hiding communication

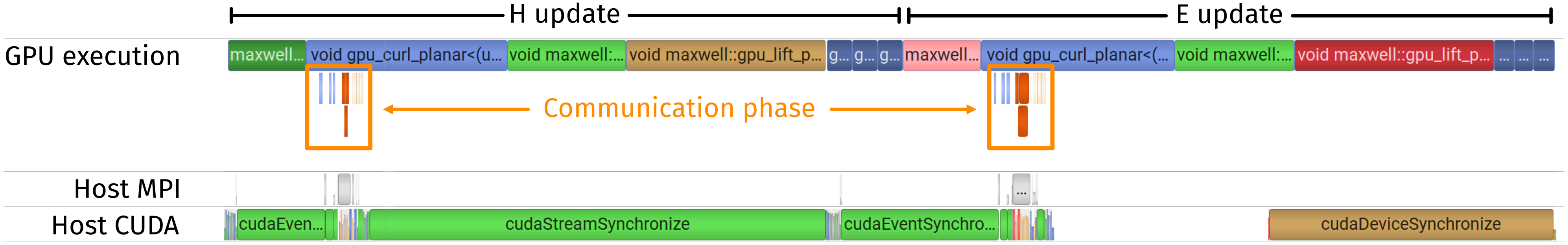
To hide communication, the implementation relies on two GPU streams that execute independently:

- A computation stream, where the majority of the computational kernels are launched
- A communication stream (high priority), where the packing and unpacking kernels (interface jumps) are executed

The kernels are submitted in a single batch to minimize kernel launch overhead:

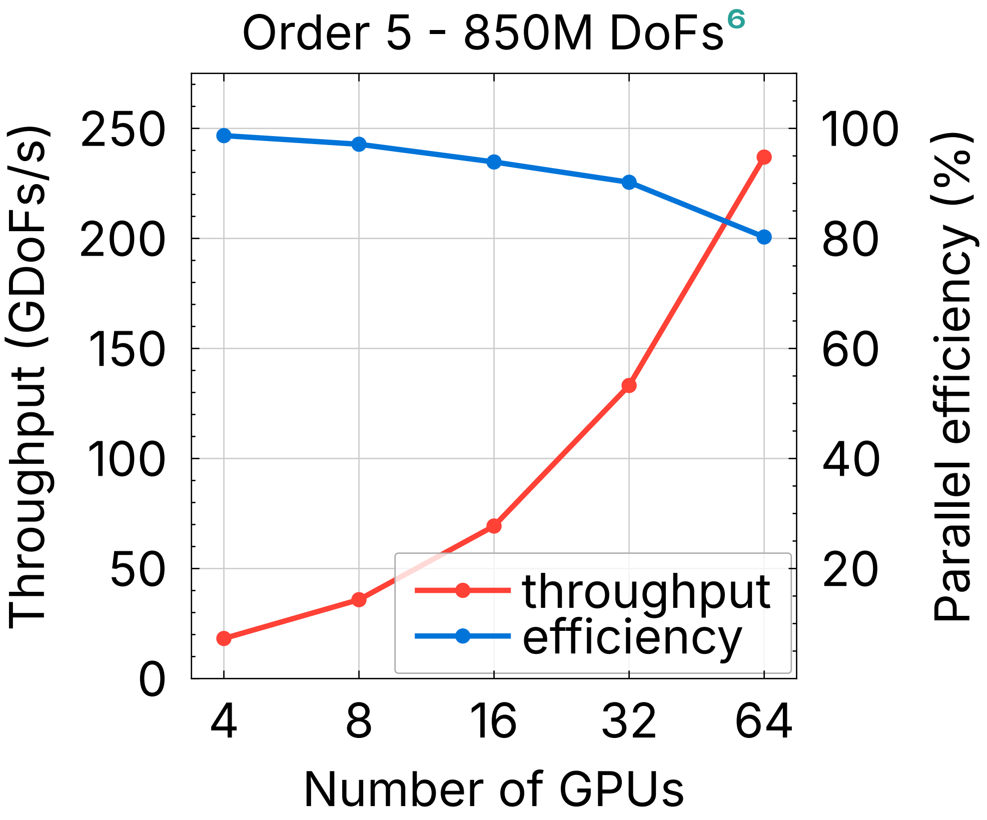
- Coordination between the two streams is handled using CUDA/HIP events
- Events are also used to synchronize GPU and CPU work, for example by triggering MPI send operations once a packing kernel has completed

Execution profile with hidden communication



- The profiling confirms that the communication is overlapped with the execution of the *Curl* kernel
- If the problem size is large enough, the execution time of the *Curl* kernel is sufficient to completely mask the communication

Hiding communication - Lucia strong scaling

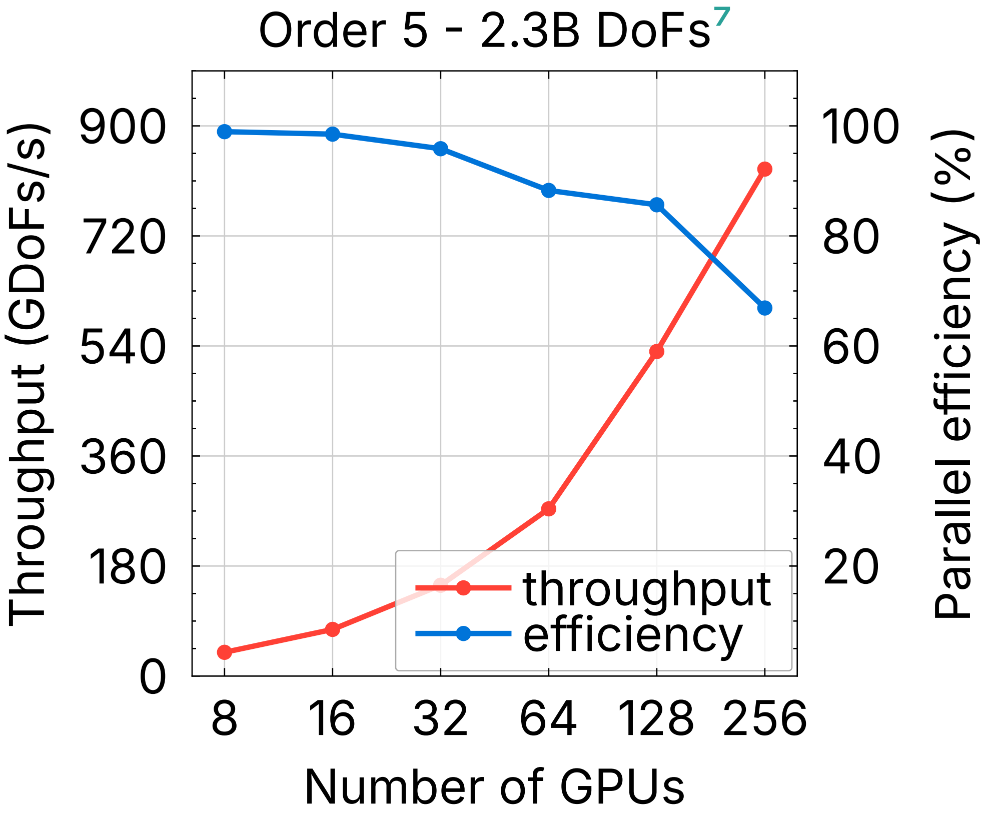


- Hiding communication improves scaling and the improvement is more visible as we increase the number of GPUs

- 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, 2× 200 Gbps Infiniband NICs

Beyond Tier-1: Leonardo strong scaling



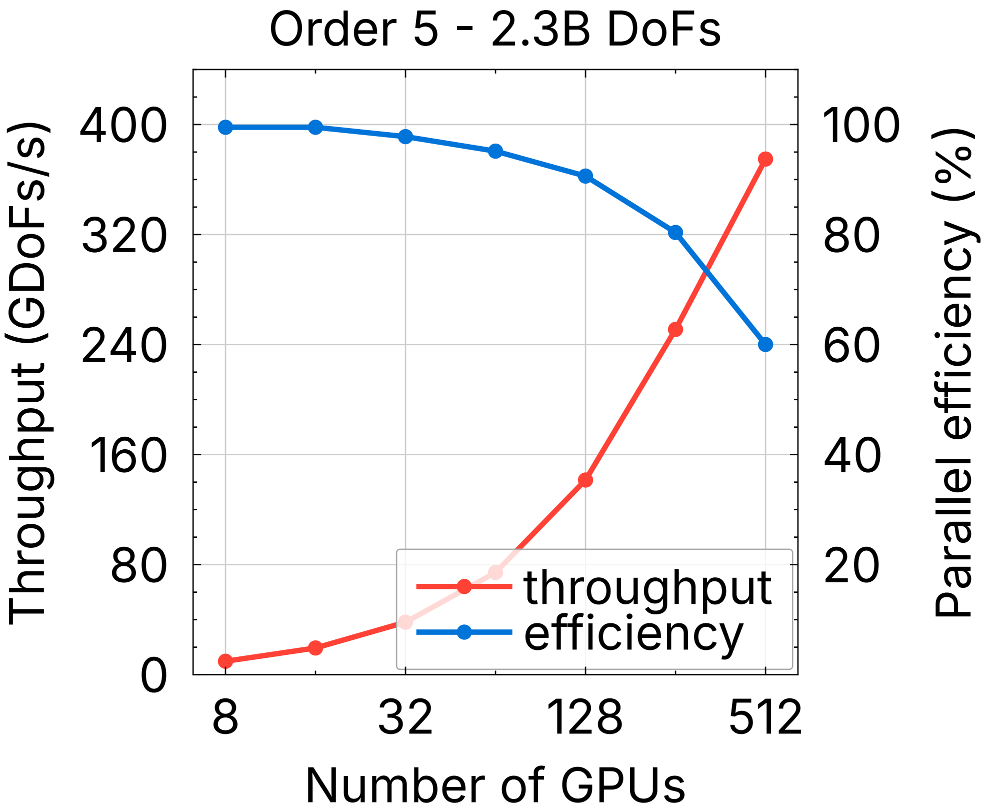
Order 5 - 2.3B DoFs[7]

- On Lucia, the maximum job size is 64 GPUs. To validate our scaling results, we need to extend beyond this limit
- On Leonardo, we observe good parallel efficiency up to 128 GPUs, but it drops beyond that

🗒️ The GPUs on Leonardo have a higher maximum power limit (450 W *vs.* 400 W on Lucia), which slightly improves single-GPU performance: 4.85 GDoFs/s compared to 4.61 GDoFs/s for Lucia

[7] Leonardo - 4x NVIDIA A100 64GB GPUs - 1x Intel Xeon Platinum 8358 32-core CPU - 512 GB of RAM, 4 × 100 Gbps Infiniband NICs

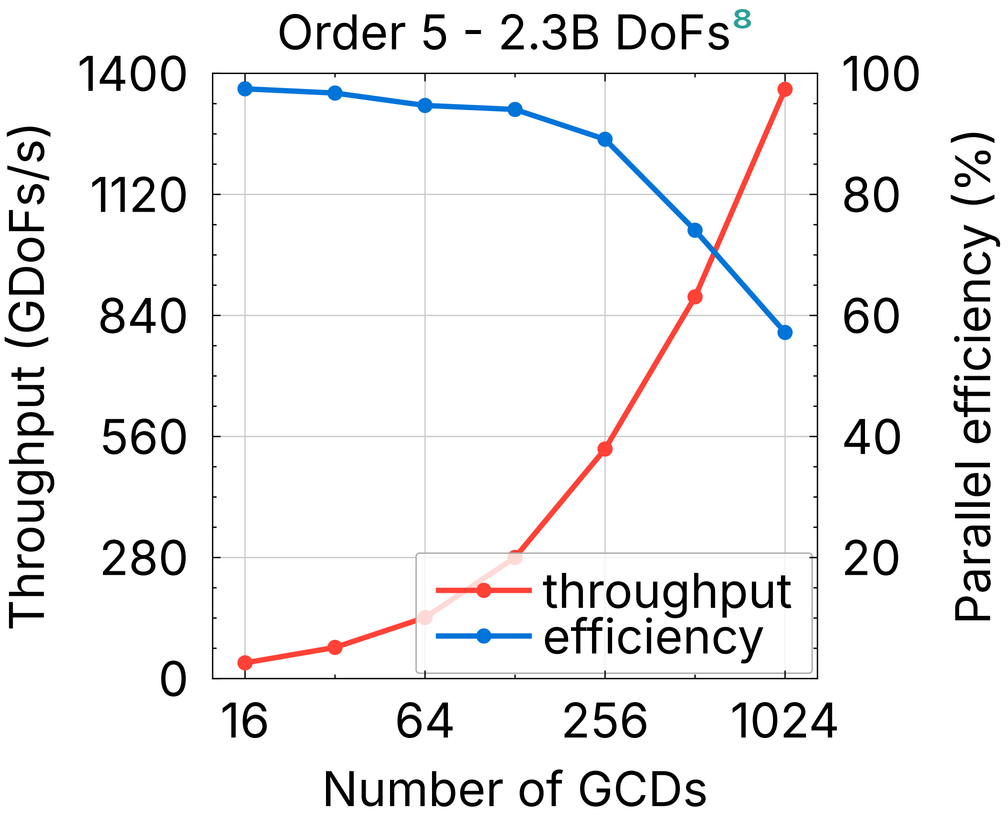
Beyond Tier-1: Leonardo strong scaling (RK4)



- In previous scaling experiments, we used Leapfrog for time integration, which updates $\boldsymbol{E}$ and $\boldsymbol{H}$ in separate steps, leaving limited opportunity to hide communication
- By switching to fourth-order Runge-Kutta, we gain more opportunities to hide communication, which improves strong scaling performance up to 256 GPUs

➡ As long as the execution time of the *Curl* kernel is sufficient to hide communication, we can expect strong scaling to remain good

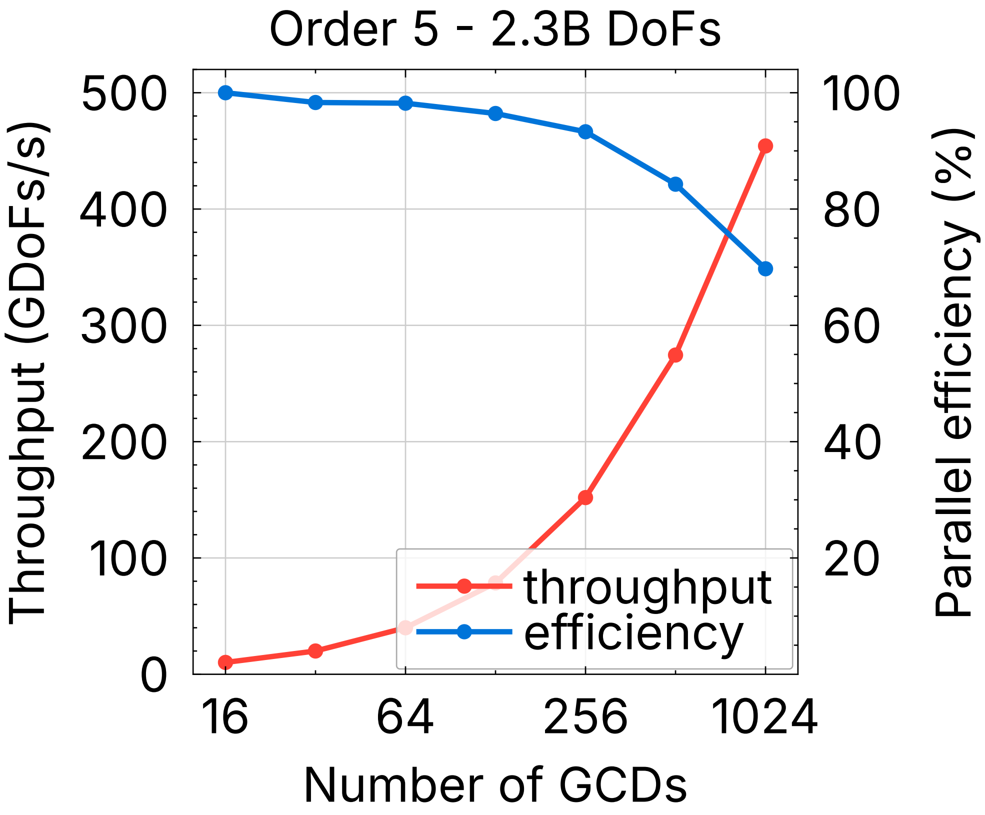
LUMI strong scaling



- Efficiency > 90% up to 256 GCDs
- After 256 GCDs the amount of computation is not sufficient to hide communication and the parallel efficiency starts to drop
- LUMI scales better than Leonardo which can be explained by the difference in interconnect bandwidth (4× 200 Gbits/s vs 2× 200 Gbits/s)

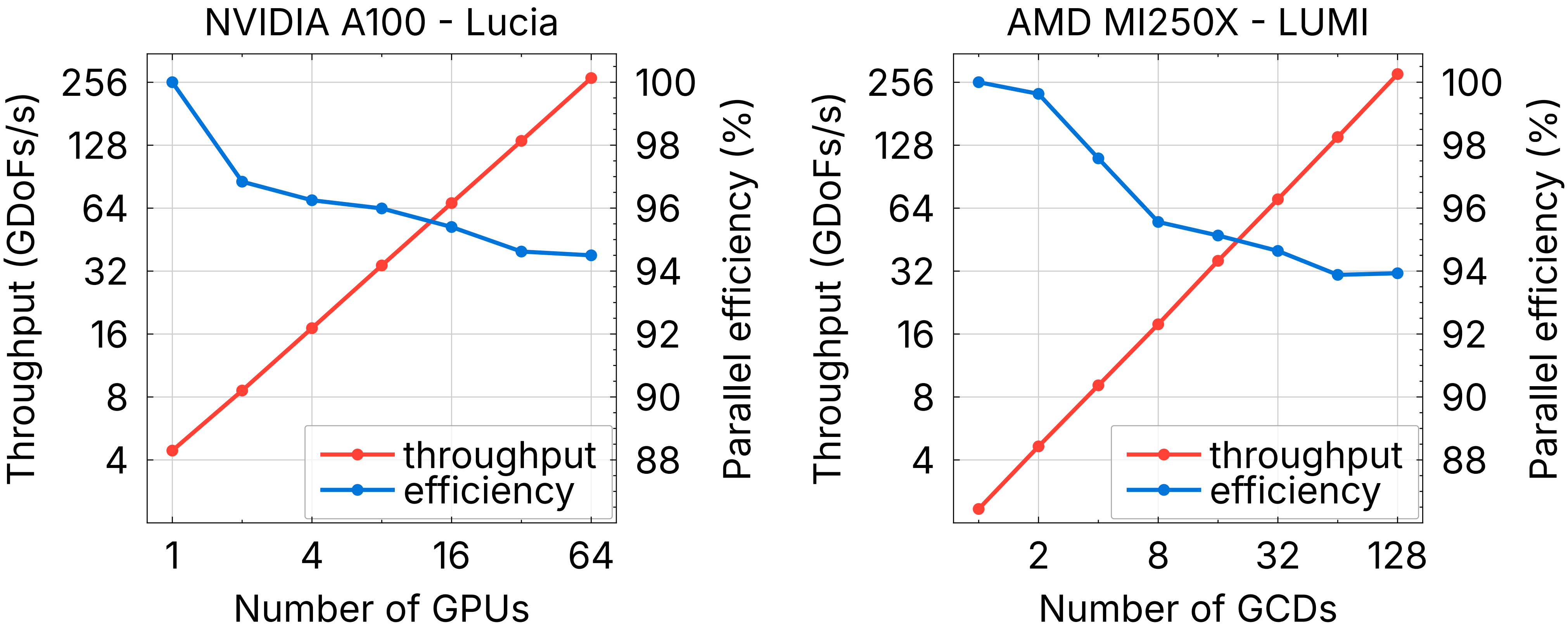
[8] LUMI - 4x AMD MI250X 64GB GPUs - 1x AMD EPYC 7A53 64-core CPU - 512 GB of RAM, 4× 200 Gbps Slingshot NICs

LUMI strong scaling (RK4)



- Using RK4, the parallel efficiency exceeds 90% up to 256 GCDs
- Beyond 256 GCDs, the efficiency begins to decrease but remains above, or close to, 70%
- For runs on 128 compute nodes, both throughput and parallel efficiency are higher on LUMI (454 GDoFs/s, 69%) than on Leonardo (374 GDoFs/s, 60%)

Weak scaling



Perspectives

While we have achieved our goal of efficiently scaling to hundreds of GPUs, there is still room for further improvement in our implementation:

- On LUMI, we plan to experiment with stream- and GPU-triggered communication implemented in Cray MPICH. Initial tests were conducted but had to be halted because some features described in the documentation are not yet implemented in the libraries currently installed on LUMI
- We also plan to evaluate NVSHMEM/rocSHMEM to allow kernels to directly incorporate both communication and computation, which could reduce the need for synchronization with the CPU

Thank you for your attention

Contact: orian.louant@uliege.be

www.enmieux.be

Cofinancé par
l'Union européenne

Wallonie

LUMI

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

EuroHPC
Joint Undertaking

We acknowledge EuroHPC Joint Undertaking for awarding us access to Leonardo at CINECA, Italy and Marenostrum5 at BSC, Spain under grant EHPC-DEV-2025D10-013