HIDALGO2

CENTRE OF EXCELLENCE

GPU-accelerated FVM for the Navier-Stokes Equations

M. Constans, Z. Horváth

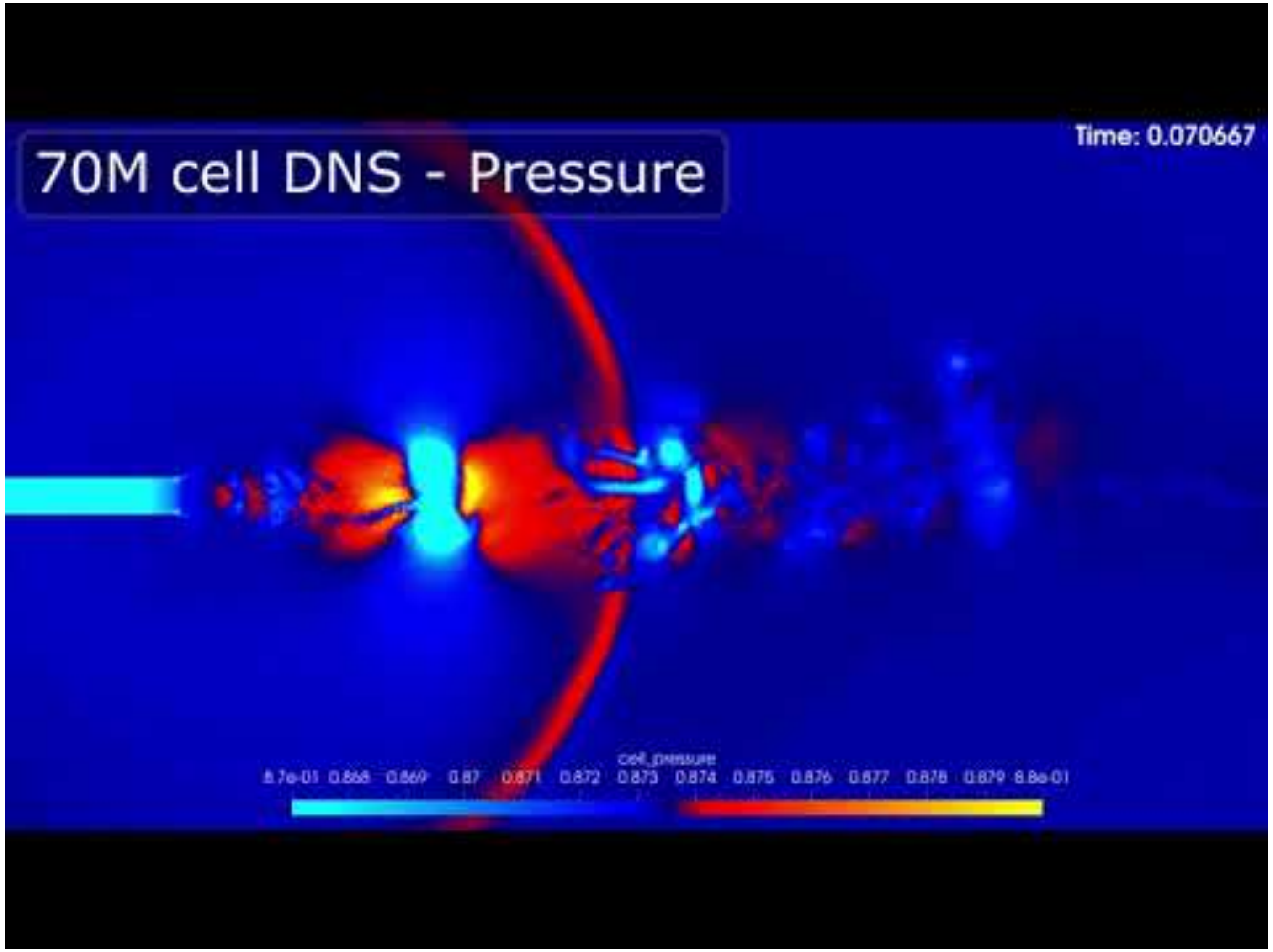
Széchenyi István University (SZE)

Thursday May 22 2025

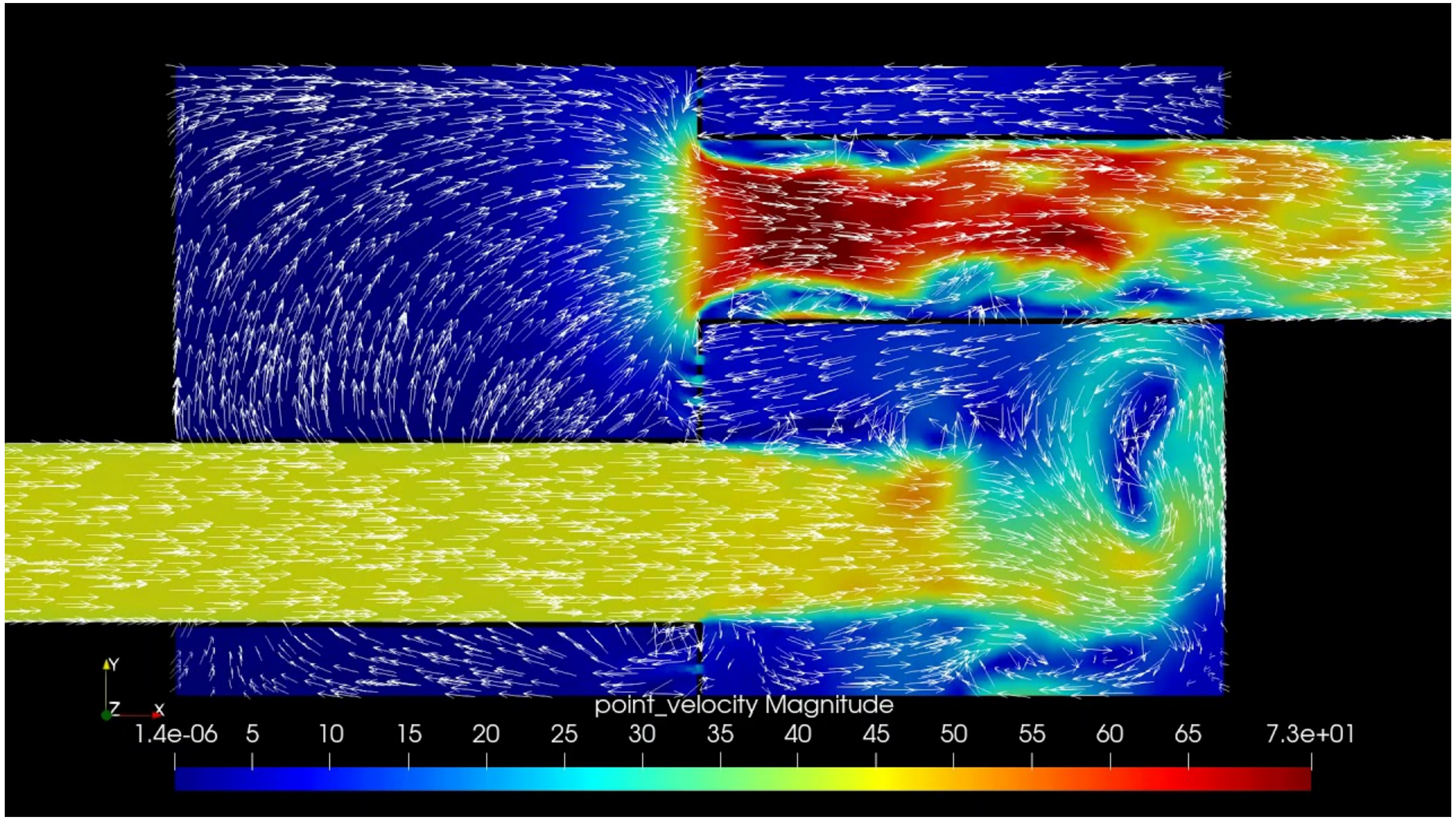
EuroHPC Joint Undertaking

Grant number: 101093457

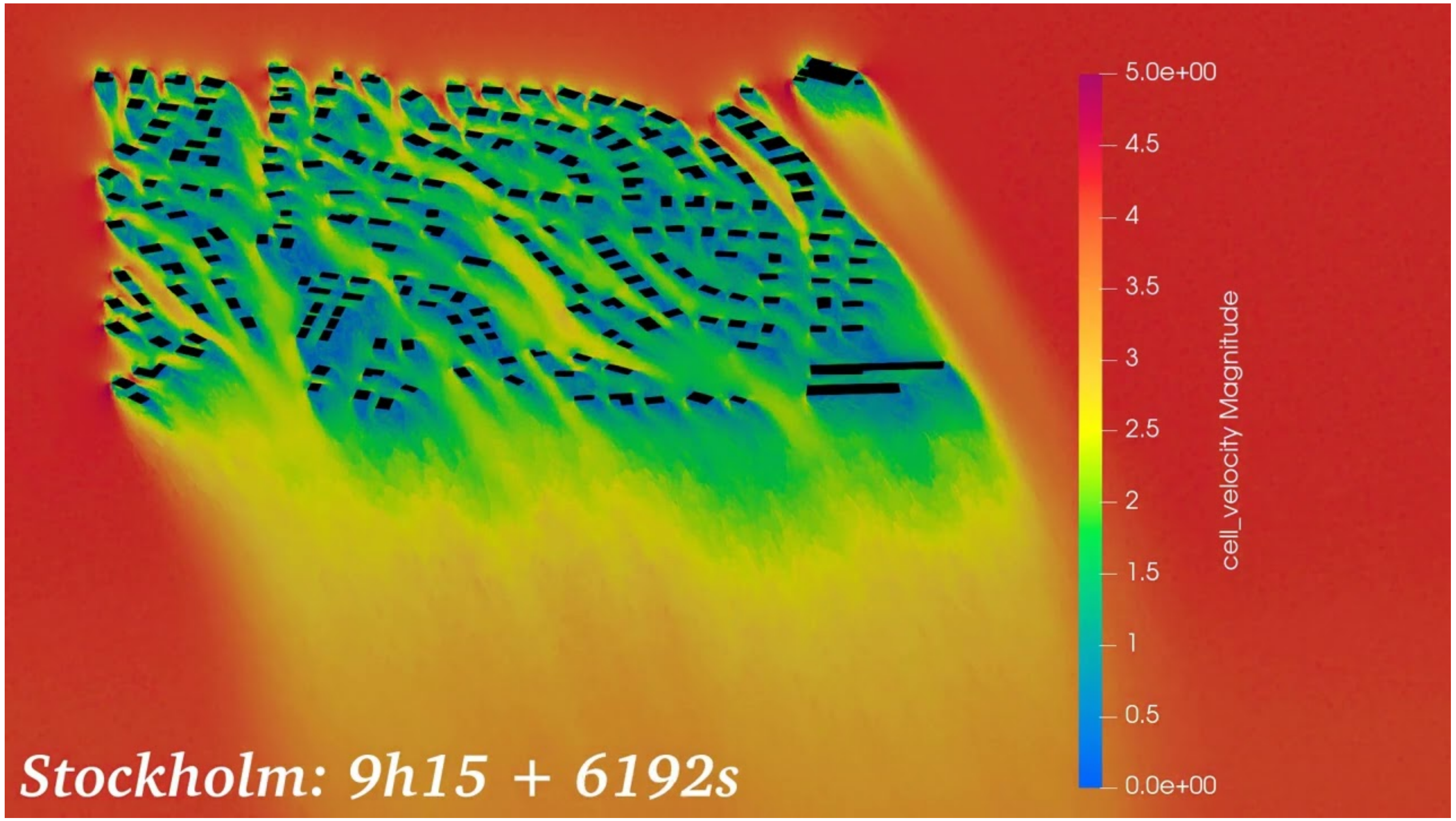
RedSIM - Exhaust Pipe Acoustics (70M Cells)


70M cell DNS - Pressure
Time: 0.070667
cell_pressure
8.7e-01 0.868 0.869 0.87 0.871 0.872 0.873 0.874 0.875 0.876 0.877 0.878 0.879 8.8e-01

RedSIM - Muffler Velocity Field (4M Cells)



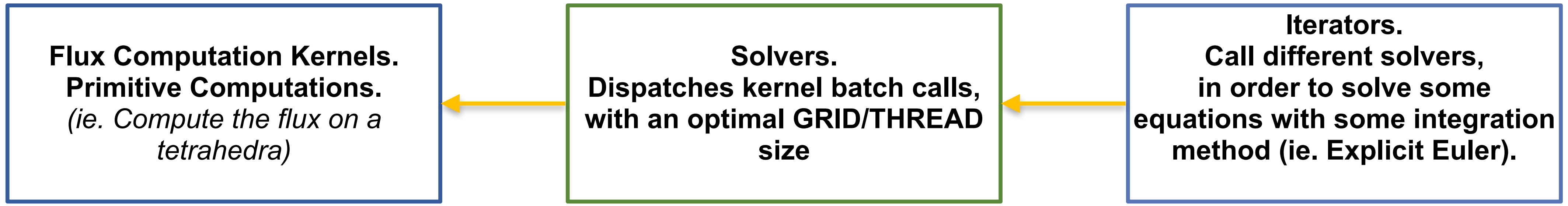
Stockholm - Urban Wind Simulation



RedSIM - Architecture Overview

- **RedSIM** is an **unstructured, polyhedral** Finite Volume Method (FVM) CFD solver, for the **Compressible Navier-Stokes Equations**. It uses the **Vijayasundaram** method. We use the Explicit-Euler method for **simulating unsteady flows.**
- Written in **C/C++**, **CUDA**, **MPI**, targeting **CUDA HPC architectures, with multiple GPU nodes**.
- Designed for GPU architectures from day 1, leveraging the strength of **CUDA GPU-s.**
- Heavily **Data Oriented Design**. We use a custom memory allocation system both on the **CPU** and **GPU** side based on Virtual Memory Paging and build our own custom data structures, like dynamic arrays and hash tables on top of it.
- **Rationale for our own containers:** Current standard library implementations (ie. STL) can suffer heavily from memory fragmentation when multiple containers are declared (placement new can help); having a continuous address space, with allocations back-to-back, allows us to upload **multiple containers in a single cuMemcpy**, by just uploading our entire Arena (Bump Allocator)
- In the same line of thought: **no exceptions**, **no RAII**; Allocations on the GPU are expensive, copies even more expensive. We want to be explicit about when copies happen. In a lot of ways, it's very similar to optimizing resource loads/transfers in video-games (models, textures, …)

RedSIM - Architecture Overview

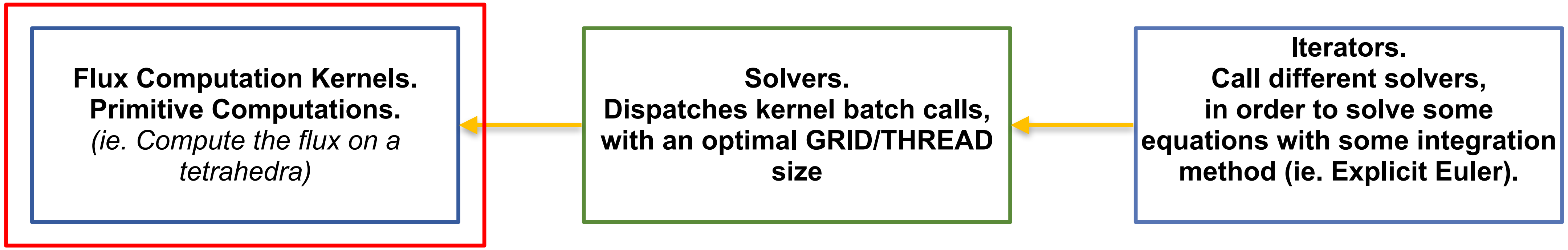


Flux Computation Kernels.
Primitive Computations.
(ie. Compute the flux on a tetrahedra)

Solvers.
Dispatches kernel batch calls, with an optimal GRID/THREAD size

Iterators.
Call different solvers, in order to solve some equations with some integration method (ie. Explicit Euler).

RedSIM - Architecture Overview



Flux Computation Kernels.
Primitive Computations.
(ie. Compute the flux on a tetrahedra)

Solvers.
Dispatches kernel batch calls, with an optimal GRID/THREAD size

Iterators.
Call different solvers, in order to solve some equations with some integration method (ie. Explicit Euler).

Most of our optimization efforts went into the kernels,
since they are the main computational bottleneck.

All kernel code is platform/hardware specific (x86, ARM, CUDA).
On the CPU-side, all code is manually written in **SIMD** (AVX512, if available).
For **CUDA**, **we manually optimize our kernels by viewing PTX disassembly, and using NSight compute on windows.**

Optimizing RedSIM for CUDA Architectures.

- **Memory Access Pattern Optimization** is arguably the most important thing in HPC.

- More often than not applications are not **compute-bound**, but **memory throughput-bound**.

- GPU-s have relatively low **VRAM**, compared to CPU architectures, that often times have terabyte of RAM to work with in HPC.

- In CFD especially, when computing very large Reynolds number simulations with over 100M cells, **proper compression of data** determine whether we can fit everything in VRAM.e

- Proper data locality is also incredivb

Memory Access Pattern Optimization

Memory Optimization Concerns

Keeping Allocations Linear & Data Locality.

Avoid memory fragmentation (thousands of disjoint malloc calls). The more fragmentation, **the more cache misses**.

Make sure that data access **is always local to some extent**.

Efficient Data Compression.

Compressing things into lower-precision variants is really common in GPU programming. This is especially challenging in scientific computing, since heavy lossy compression doesn't play well with Simulation stability.

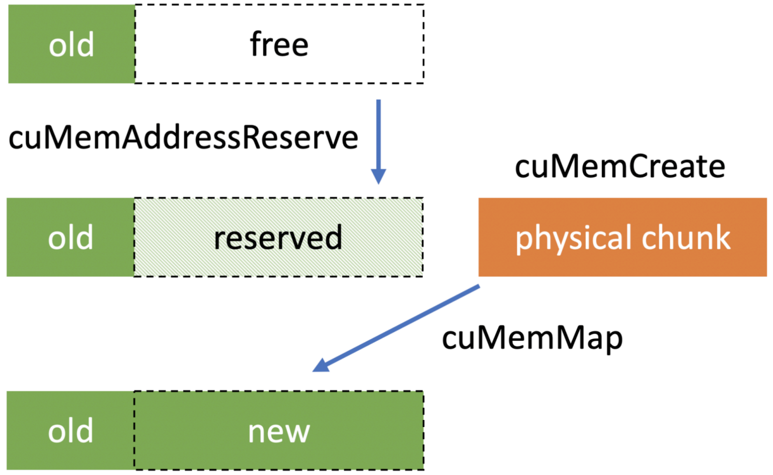
Communication Latency

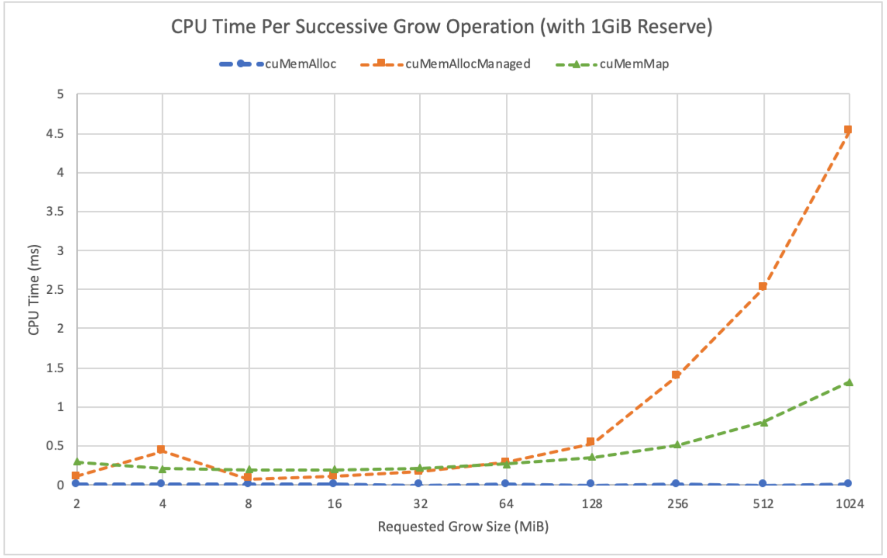
CPU <-> GPU transfers are very expensive, stall the pipeline, and should be minimized. **Moreover**, inter-node communication between two GPU-s on different nodes is even MORE expensive, and work dispatch has to be very carefully considered.

Keeping Allocations Linear

- **On the CPU-side:** We manage memory ourselves, using virtual memory reserves / commits. (**VirtualAlloc** on win32, **mmap** on linux), with a combination of our own custom allocation structures: Arenas (also known as Bump Allocators), **Pools, Free Lists**. This allows us to have complete control over memory fragmentation, and pass large chunks of data around between the CPU/GPU with **CUDA**, CPU/CPU with **MPI**.

- **On the GPU-side:** We do the exact same thing - thanks to **cuMemAddressReserve, cuMemCreate, cuMemMap,** we can manage virtual memory in a similar but slightly different way.



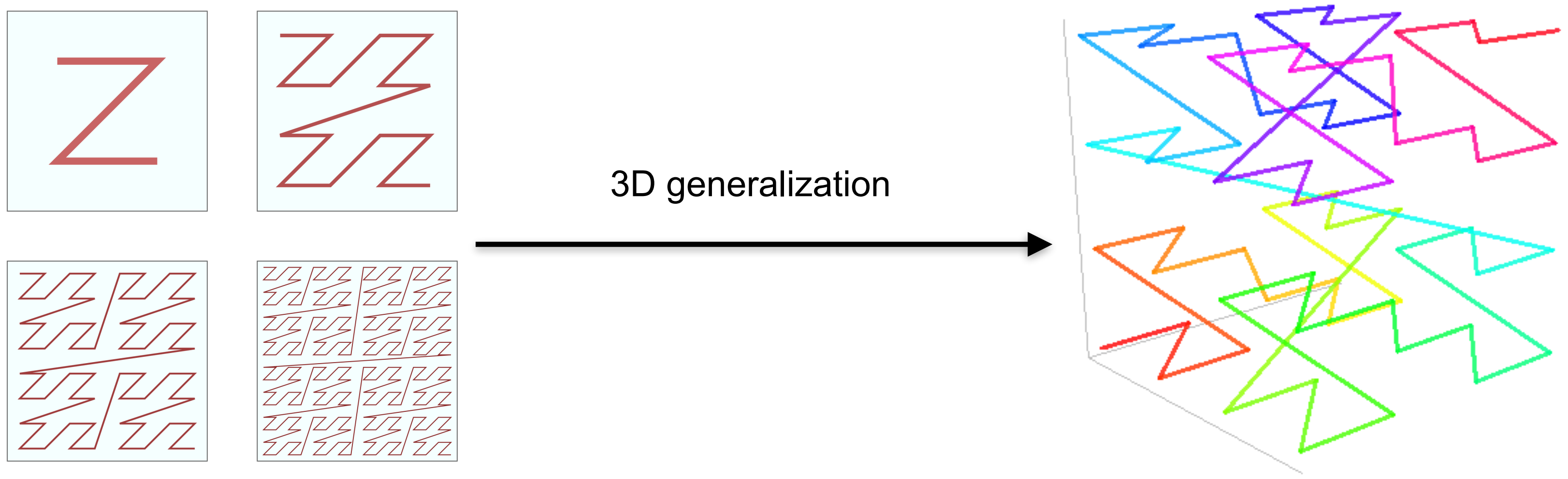


Memory Access Locality - Inside a single Node.

- Inside a single partition, we use **space filling curves** in order to guarantee some form of data locality, and avoid cache misses.

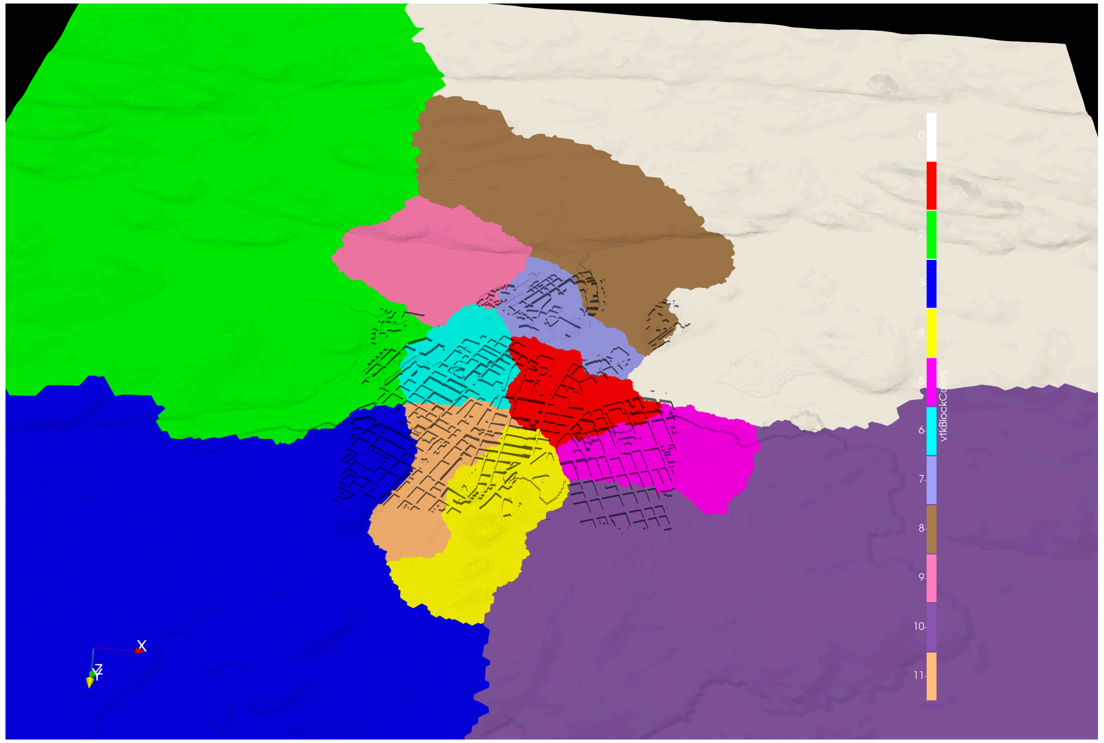
- Morton Curves are really great because they only take a couple of cycles to compute (bit interleaving), yet produce optimal results (better than Hilbert curves for our application).

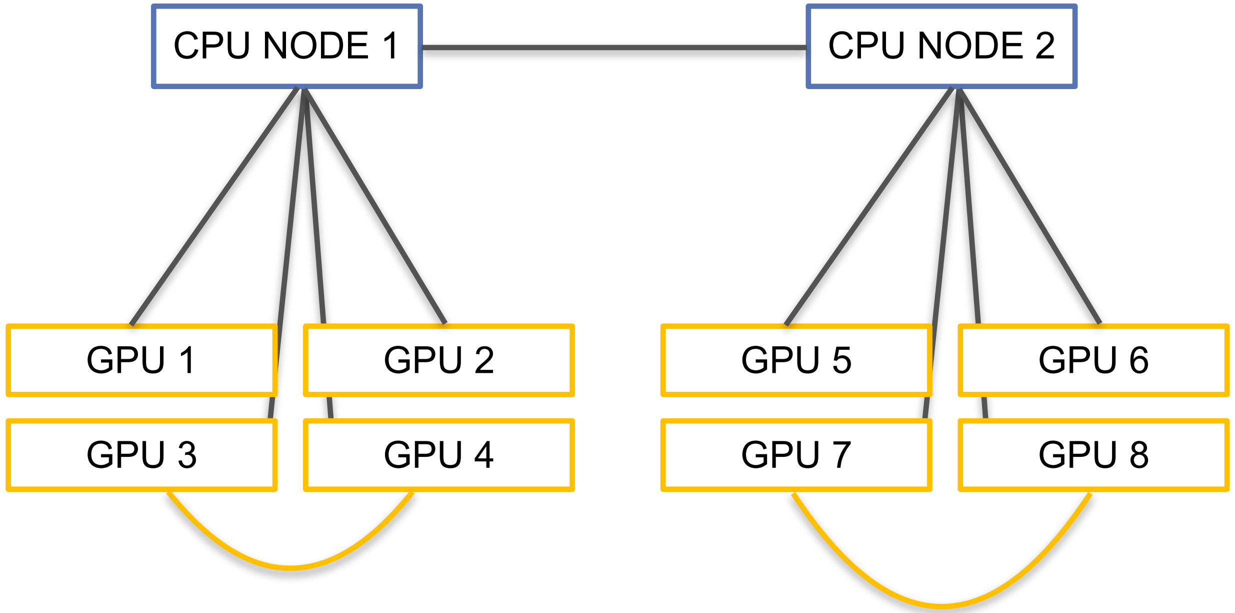
- Fun Fact: For graphics API-s, texture lookups usually using Morton curves for minimizing cache misses, alongside chunks (consider how many cache misses there would be if data was stored in rows).



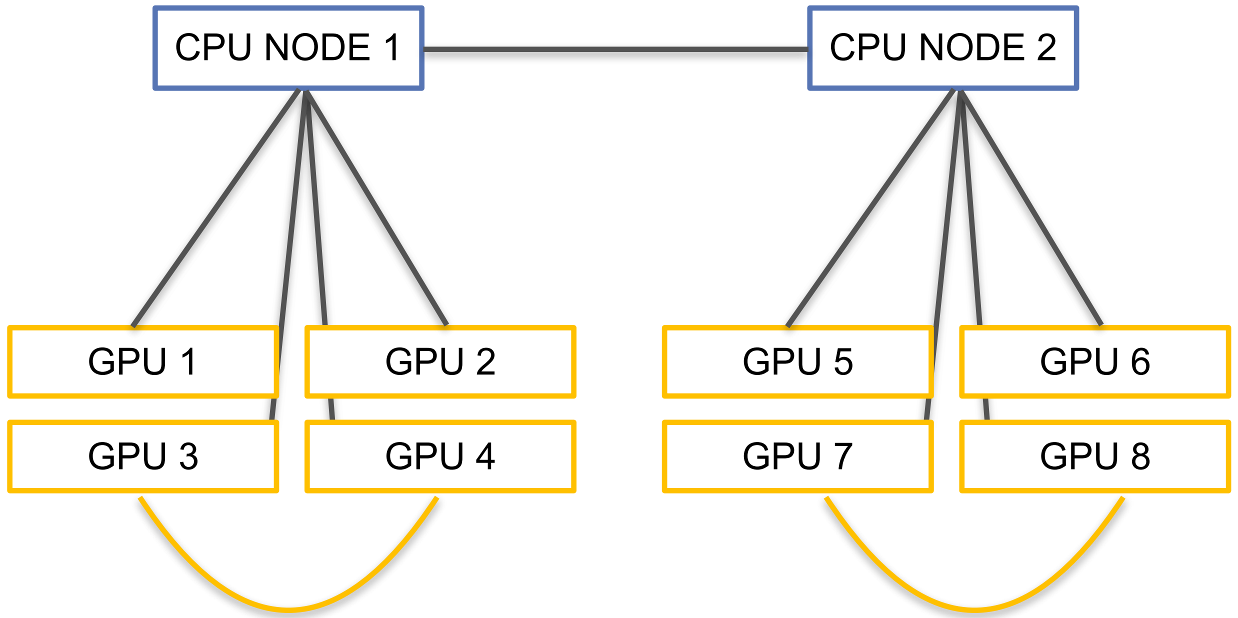
Memory Access Locality - Multi-Node communication

- In order to communicate between multiple **GPU nodes** (1 CPU is attached to 8 GPU-s, typically A100-s), we have to decompose our domain into **multiple partitions.** Correct decomposition is key for optimal performance. A good decomposition must have a minimal contact surface, and minimize the amount of data exchange needed between GPU-s. It should also cluster GPU-s on the same node in an adjacent way, in order to avoid unnecessary communication between different nodes.

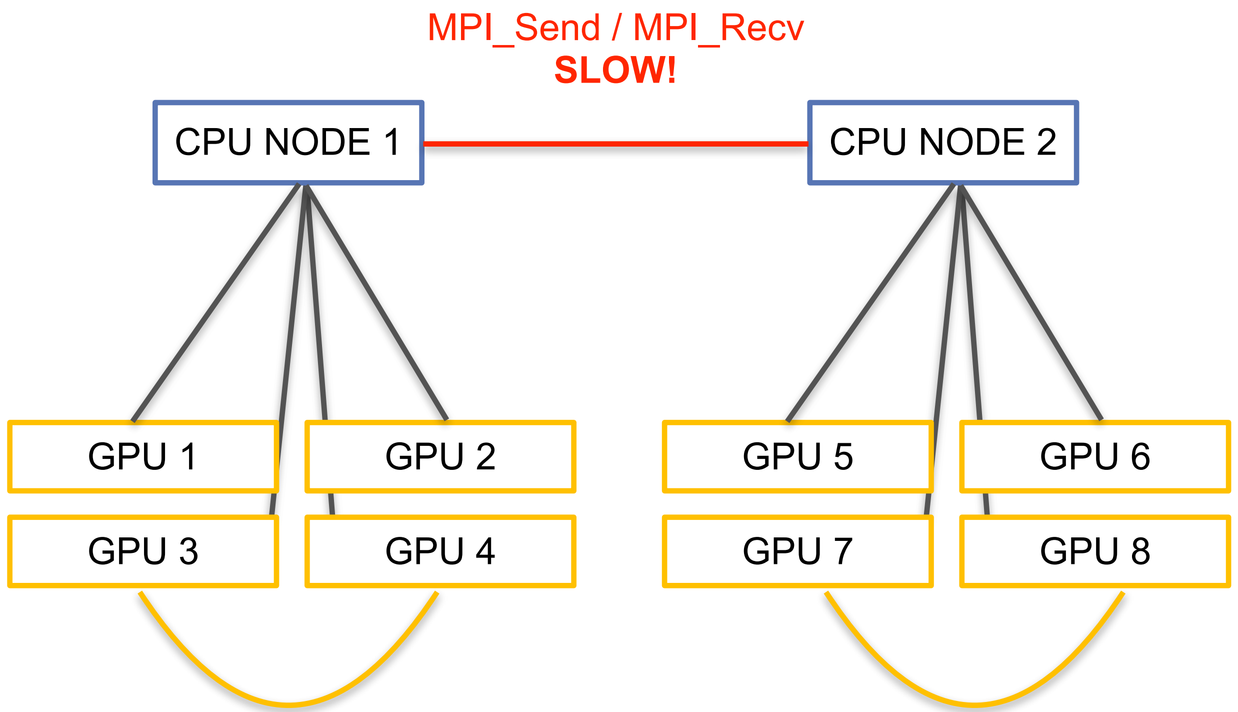




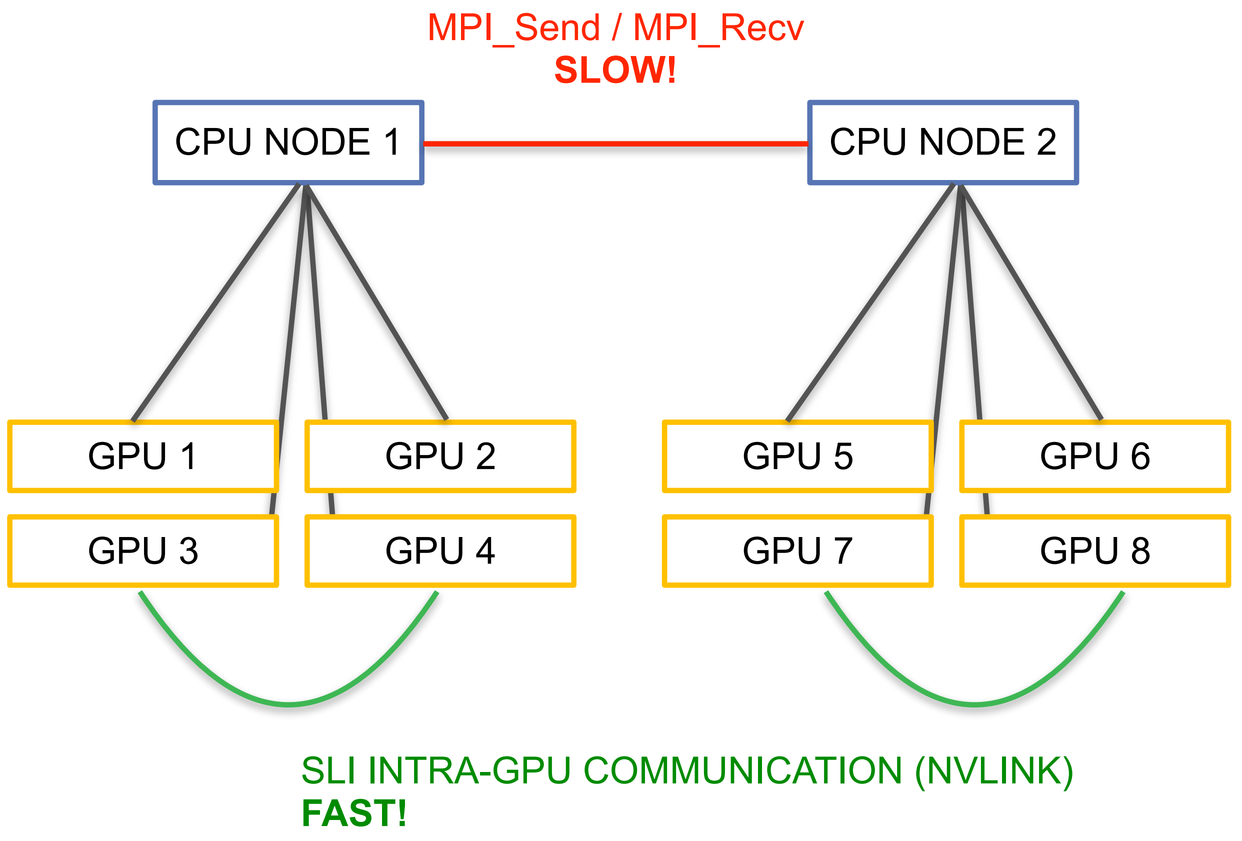
Typical HPC Architecture



Typical HPC Architecture



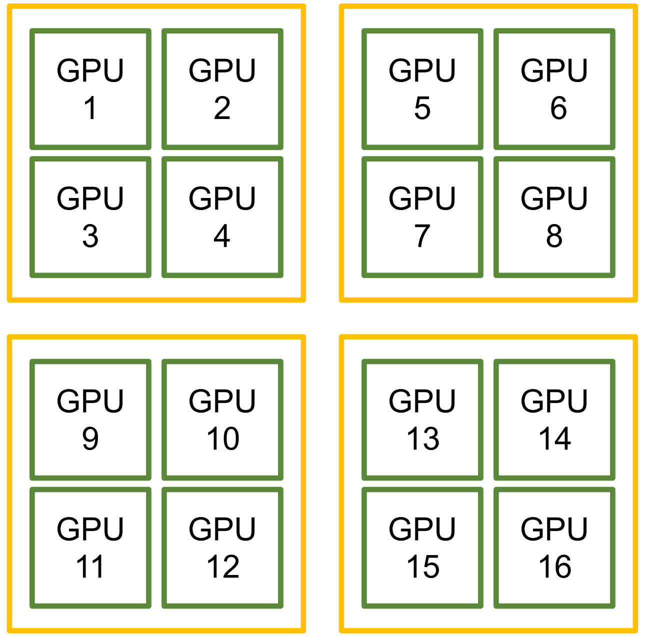
Typical HPC Architecture



Partitioning Strategy - CUDA + MPI

- We partition with ZOLTAN:
  - **STEP1:** Zoltan_LB_Partition With **ParMETIS**, for **NODES**.
  - STEP2: Zoltan_LB_Partition with Morton Space Fill Curves, for
  - **EACH THREAD** on the CPU, **EACH BLOCK/THREAD FOR CUDA**.
- MPI_GRAPH partitioning communicator for describing topology.
- NVLINK for GPU/GPU communication via SLI. (We use the CUDA API).
- MPI for CPU/CPU communication.

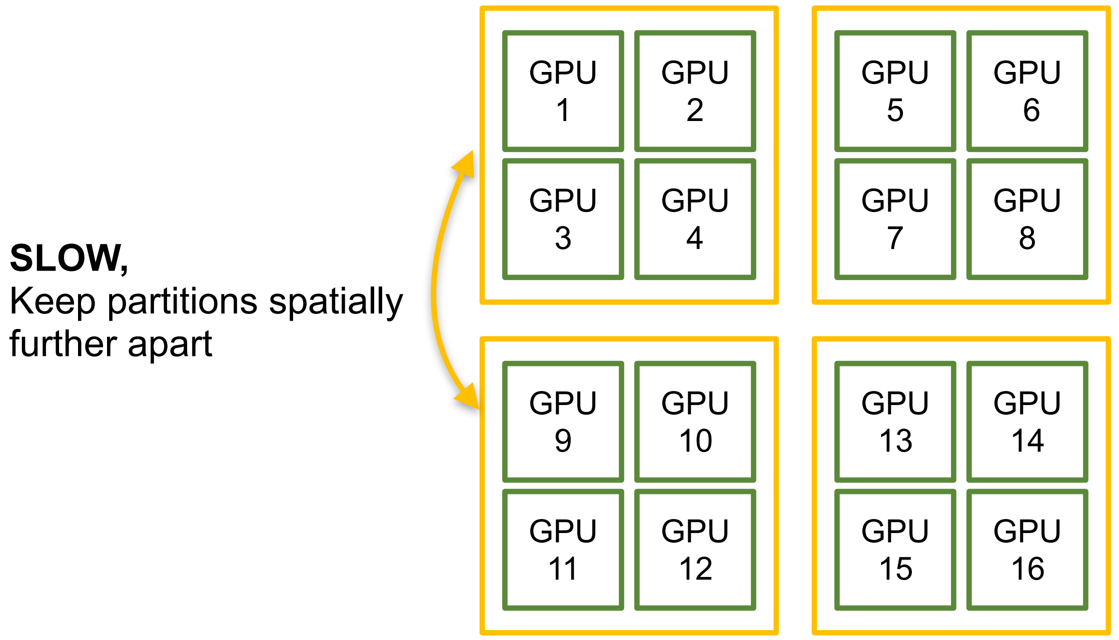
__Partitioning Strategy__



Partitioning Strategy - CUDA + MPI

- We partition with ZOLTAN, keeping these facts in mind.
- MPI_GRAPH partitioning communicator for describing topology.
- NVLINK for GPU/GPU communication via SLI. (We use the CUDA API).
- MPI for CPU/CPU communication.

Partitioning Strategy

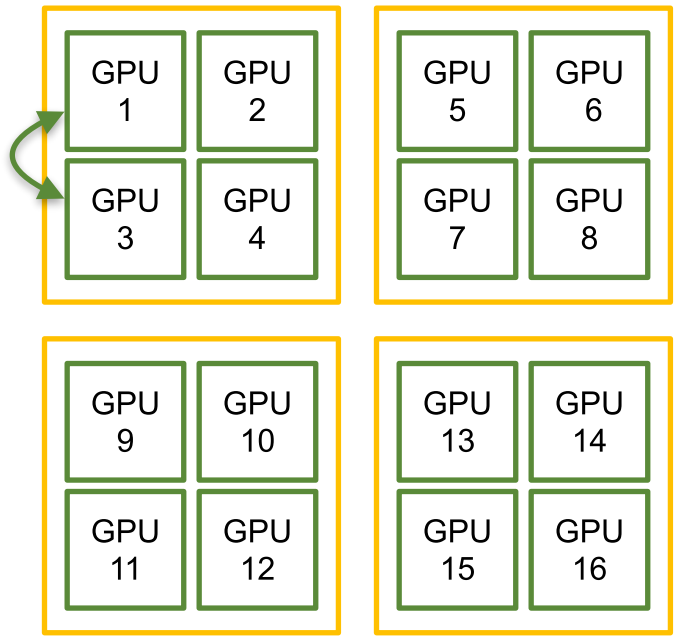


Partitioning Strategy - CUDA + MPI

- We partition with ZOLTAN, keeping these facts in mind.
- MPI_GRAPH partitioning communicator for describing topology.
- NVLINK for GPU/GPU communication via SLI. (We use the CUDA API).
- MPI for CPU/CPU communication.

Partitioning Strategy

FAST,
Keep partitions spatially close to each other.



Minimum Reduction - Global Time Stepping

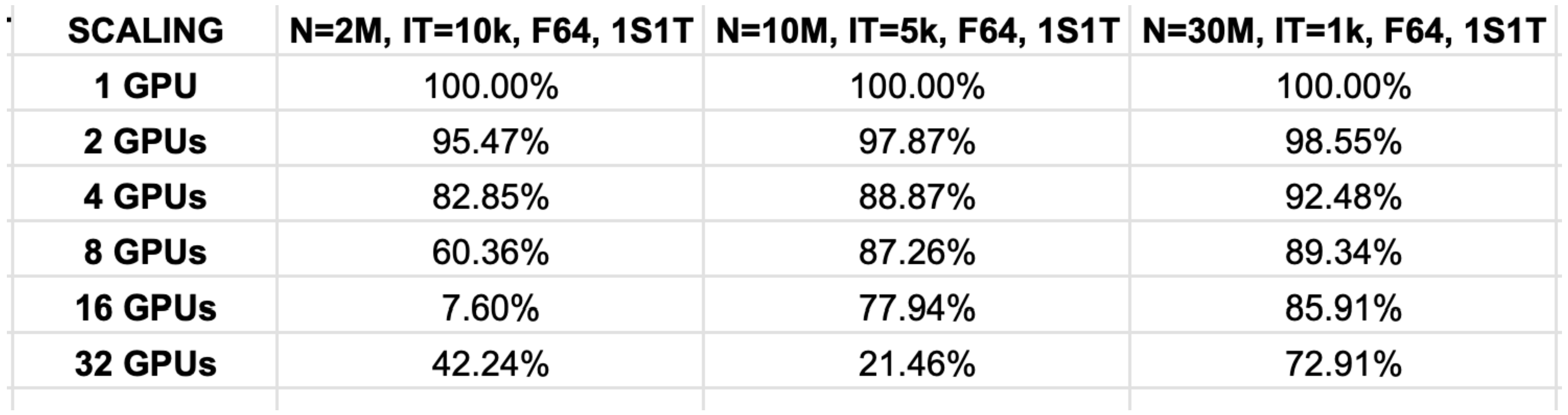
- Since we’re using a Global Time Stepping Scheme, we need to find a minimum tilmestep to use. To painful part is that in order to determine the global minimum, each **GPU, GPU NODE** has to find a global minimum.
- In order to parallelize this, we use **BLOCK BASED minimum-reduction in CUDA.**
- Once we find the minimum on each GPU on a given node, we use **MPI_Reduce** to find the global minimum.

RedSIM - Scaling

- Current Scaling Numbers on **KAROLINA HPC, Nvidia A100s**.'

- For N=**2M**, and N=**10M,** we relatively bad scaling numbers for 8, 16, 32GPU-s. The reason is, 2M cells starts being a problem **too small** for 16-32 GPUs, so we start stalling with communication time.

- As can be seen, with N=30M, our scaling numbers are much better, since each GPU is busy all the time with compute, and the GRID/THREAD distribution is much better.



SCALING	N=2M, IT=10k, F64, 1S1T	N=10M, IT=5k, F64, 1S1T	N=30M, IT=1k, F64, 1S1T
1 GPU	100.00%	100.00%	100.00%
2 GPUs	95.47%	97.87%	98.55%
4 GPUs	82.85%	88.87%	92.48%
8 GPUs	60.36%	87.26%	89.34%
16 GPUs	7.60%	77.94%	85.91%
32 GPUs	42.24%	21.46%	72.91%

Q&A