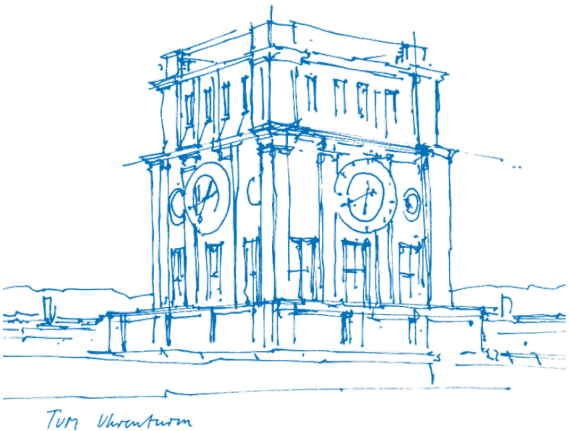
Chair of Scientific Computing
TUM School of Computation, Information and Technology
Technical University of Munich

ExaHyPE on GPUs



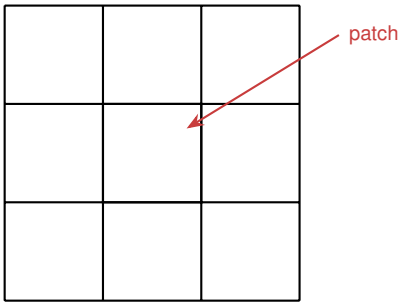
Mario Wille

Chair of Scientific Computing
TUM School of Computation, Information and Technology
Technical University of Munich

Joint SeisSol & ExaHyPE User Workshop,
LRZ Garching, June 11th, 2026

ExaHyPE in One Slide

- An engine for **hyperbolic PDE systems** (Euler, shallow water, seismic waves, astrophysics, . . .).
- Solvers: **Finite Volumes** and **ADER-DG** (+ a-posteriori limiting), explicit time stepping.
- Built on **Peano**: dynamically adaptive Cartesian meshes organised as a *spacetree*, traversed along a space-filling curve.

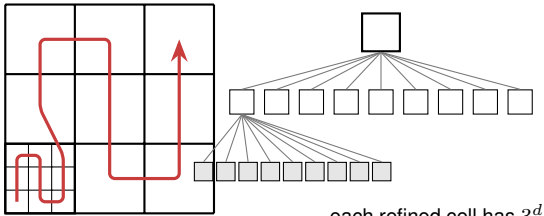


Each cell holds a patch of unknowns
(e.g. finite volumes or DG polynomials)

Goal

Run the whole ExaHyPE simulation on the GPU with **zero host↔device transfers**.
Exception: file output and grid adapt.

Background: How ExaHyPE Computes

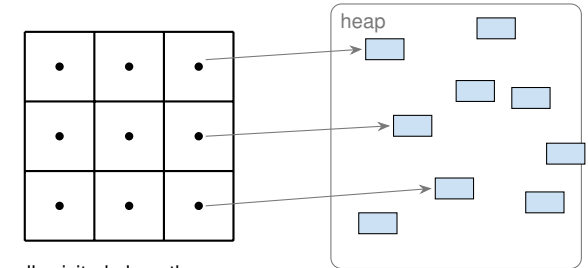


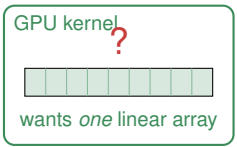
adaptive Cartesian grid; the space-filling curve orders all cells (depth-first through refinements)

each refined cell has 3^d children

- Per time step, **one or more tree traversals** walk the whole mesh and call the solver's compute routines **one cell at a time**, in curve order. Multi-sweep schemes (a-posteriori limiting, predictor-corrector) need several traversals per step, because neighbour data travels only with the traverse.
- Everything happens inside this traversal: mesh management, neighbour exchange, *and* the numerics. Data travels with the traversal on stacks; there is no global index or array.
- Parallelism: the mesh splits into subtrees, one task per subtree.

Challenge (1): Solution Data is Scattered Across the Heap



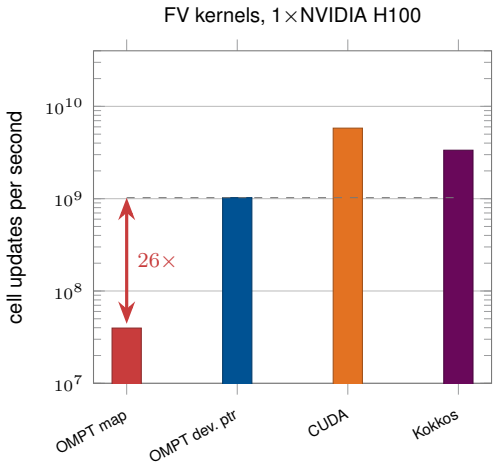


- Every cell/face allocates its own patch buffer on first use.
- The numerics run *inside* the traversal events, cell by cell, on the CPU.
- Moving a cell between subtrees or ranks moves its buffer through the traversal stacks.

Challenge (2): Why This Defeats the GPU

1. **No batch.** A GPU kernel wants one raw pointer and a (embarrassingly) parallel loop. The degrees of freedom live in many separate, scattered heap allocations: there is nothing to launch a kernel over.
2. **Numerics are part of the traversal.** The update fires inside a sequential, recursive tree walk, as tiny per-cell callbacks. Offloading each callback means a kernel launch *and* a host↔device copy per cell.
3. **Host pointers mean nothing on the device.** The cell's `DoFType*` cannot be offloaded; the mesh structure, halo exchange and load balancing all use host pointers.

Challenge (3): CPU↔GPU Data Transfers Hurt Performance

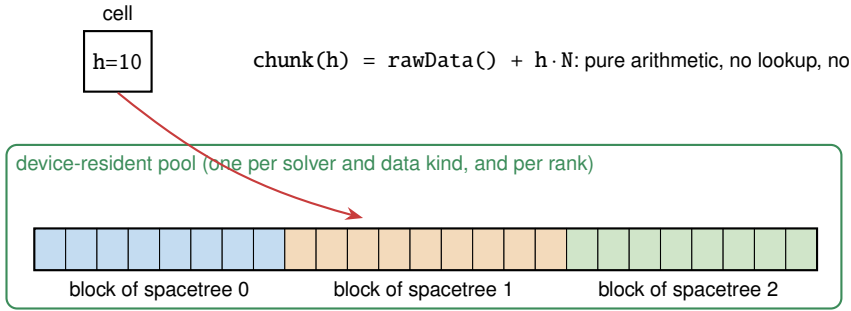


- Identical generated finite-volume kernels (shallow water, one full time step per measurement).
- **OMPT map**: the patch data is *copied* to the GPU and back around every kernel.
- **OMPT device pointer**: the same kernel reading data that already *lives* on the GPU.
- The copy alone costs a factor of $\mathbf{26}$, before any kernel optimisation.

Consequence

Data must *reside* on the device and must not be moved around.

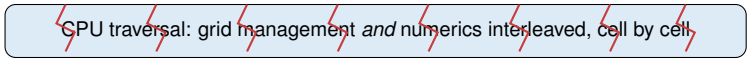
Solver-Owned Memory Pools



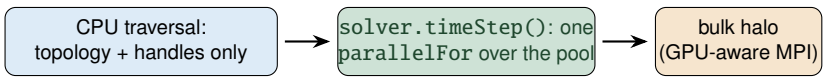
- All unknowns of one kind live in **one contiguous array**; a cell or face carries only a 32-bit **handle**.
- Each spacetree fills a **disjoint block**: no atomics, no locks, and every page is first-touched by the thread that owns it (NUMA).
- Filled once at initialisation, then **frozen**; only mesh adaptation rebuilds it (grow only).

Split the Traversal from the Numerics

Before:



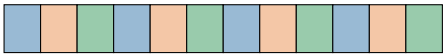
Now:



- The traversal keep the mesh: building, adapting, partitioning. It moves **handles**, never unknowns.
- All numerics (time step, initial conditions, AMR interpolation, limiting, error norms) become **kernels over the pool**, inherently batched; reductions via `parallelReduce`.
- Per-cell geometry and face connectivity are recorded into small arrays at initialisation, so kernels need no mesh.

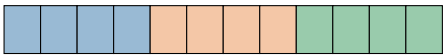
Memory Layout

AoS (CPU): one patch after another



patch 0

SoA (GPU): one unknown after another



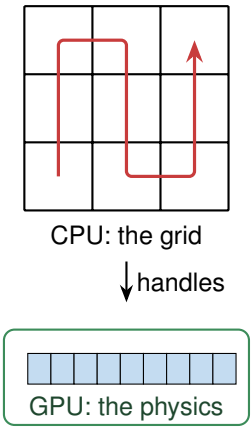
ρ of all volumes

same pool, same handles:
only the index formula changes

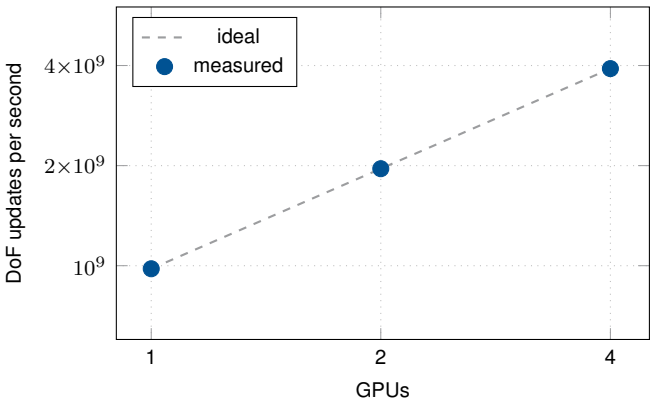
- The pool is **layout-agnostic**: it manages fixed-size blocks; the layout is just the indexing convention.
- Kernels are generated against an **enumerator** $(\text{cell}, \text{volume}, \text{unknown}) \mapsto \text{index}$; the code generator instantiates it per build: **AoS for CPU caches, SoA for coalesced GPU access**.
- Possible *because* the CPU never touches unknowns.

Summary

- The physics lives on the GPU, in **one contiguous array per solver**.
- The CPU **manages the mesh**. It never touches the solution; only lightweight handles.
- On a static grid the time loop **never traverses the mesh**: the GPU simply computes.
- Between GPUs, **only boundary data** is communicated via GPU-aware MPI.
- **The mesh traversals disappear.** Static grids compute time-step without any traversal; the mesh is traversed on demand when the solution asks for refinement, or plotting.



Strong Scaling Across GPUs



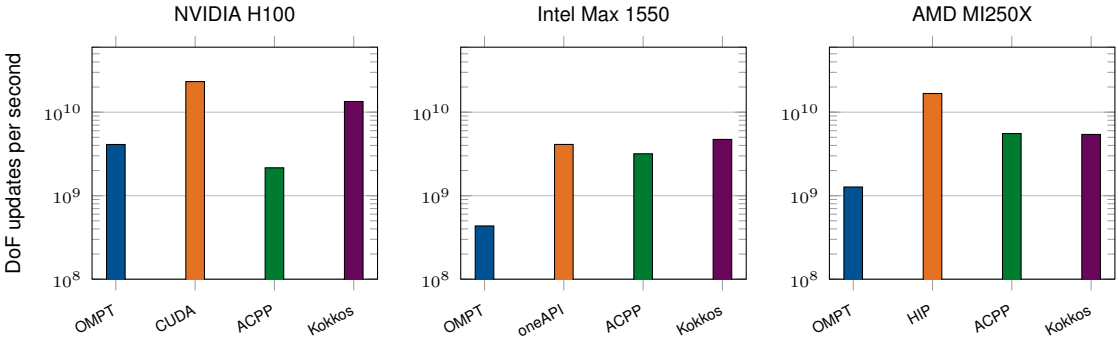
3D Euler point explosion, one rank per GPU.

Halos move device to device; the volume data never moves.

HomeOne, 4 x RTX3080

Zero data transfers between CPU and GPU!
Full App; *not* Mini-App!

Performance Portability: One Code, Three GPU Vendors



- Finite-volume kernel benchmarks (shallow water) on all three vendors; throughput of one full time step.
- Native backends (CUDA, oneAPI, HIP) deliver the highest performance; **Kokkos** is the most consistent portable backend, with **AdaptiveCpp** close behind.

Next Steps

- Kernel tuning & kernel fusion.
- Dynamic AMR and dynamic load balancing.
- CPU traverses and rebuilds the grid while the GPU keeps computing.
- $\rightarrow$ Overlap communication and computation.

Thank you for your attention

Questions?