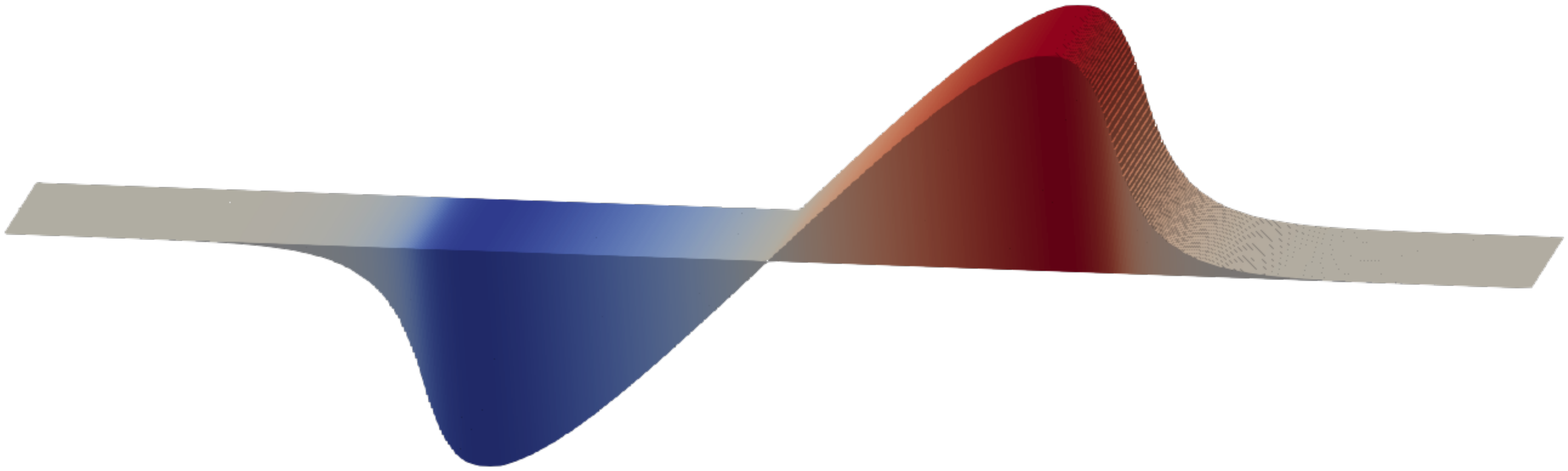
Developing a GRMHD code for heterogeneous computing

Challenges and perspectives

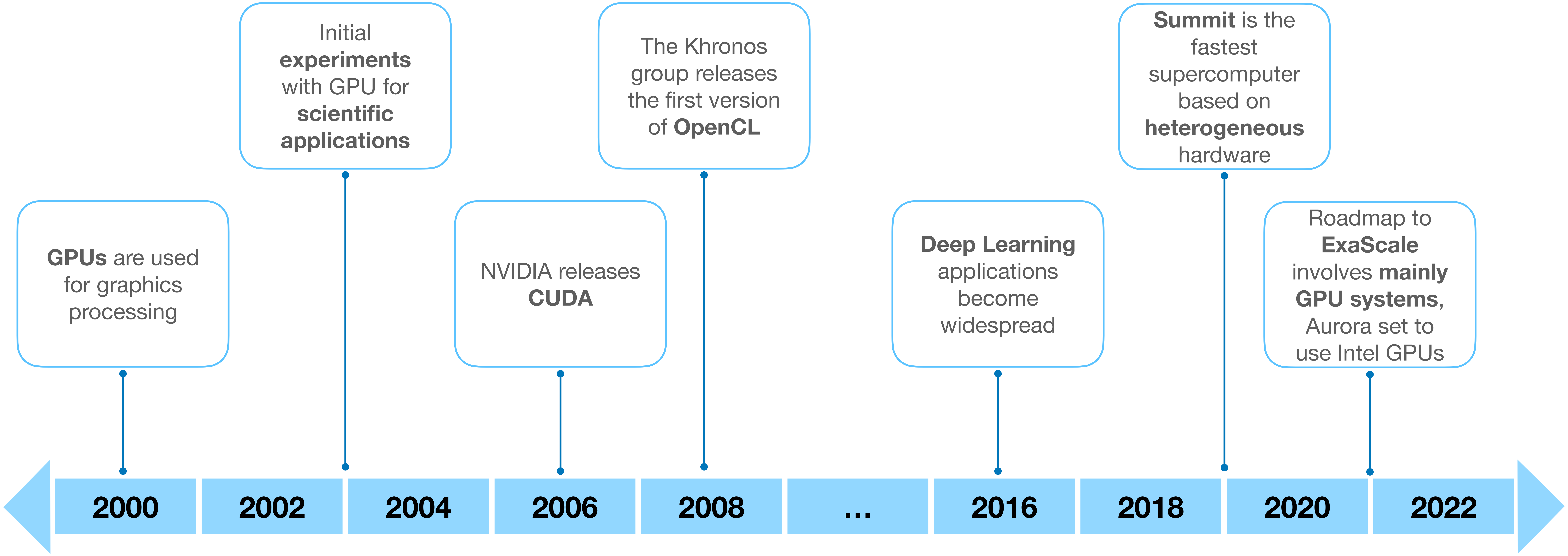


Based on work done in collaboration with L. Rezzolla

Table of Contents

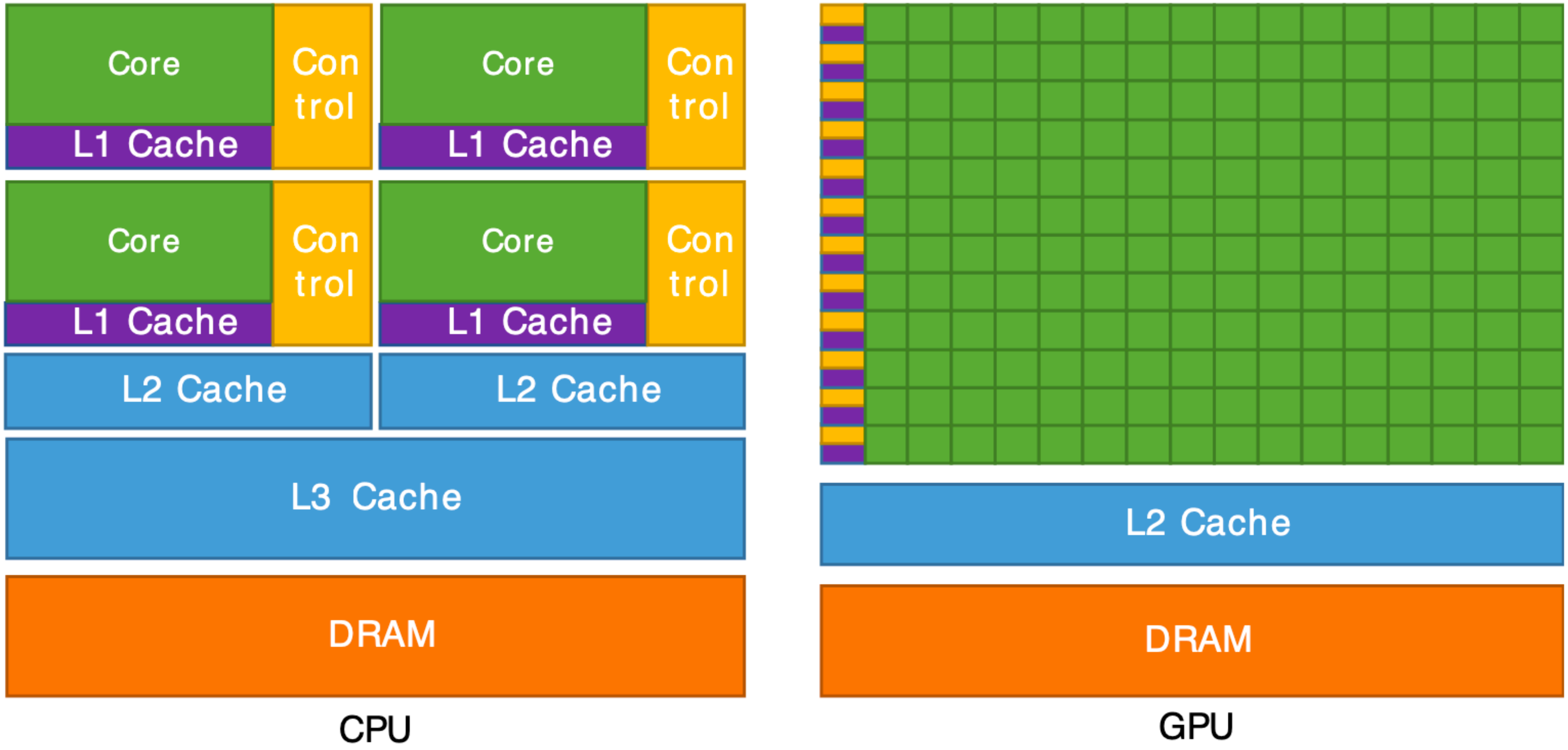
Introduction and motivation

Graphics **P**rocessing **U**nits are becoming prevalent tools for High Performance Computing



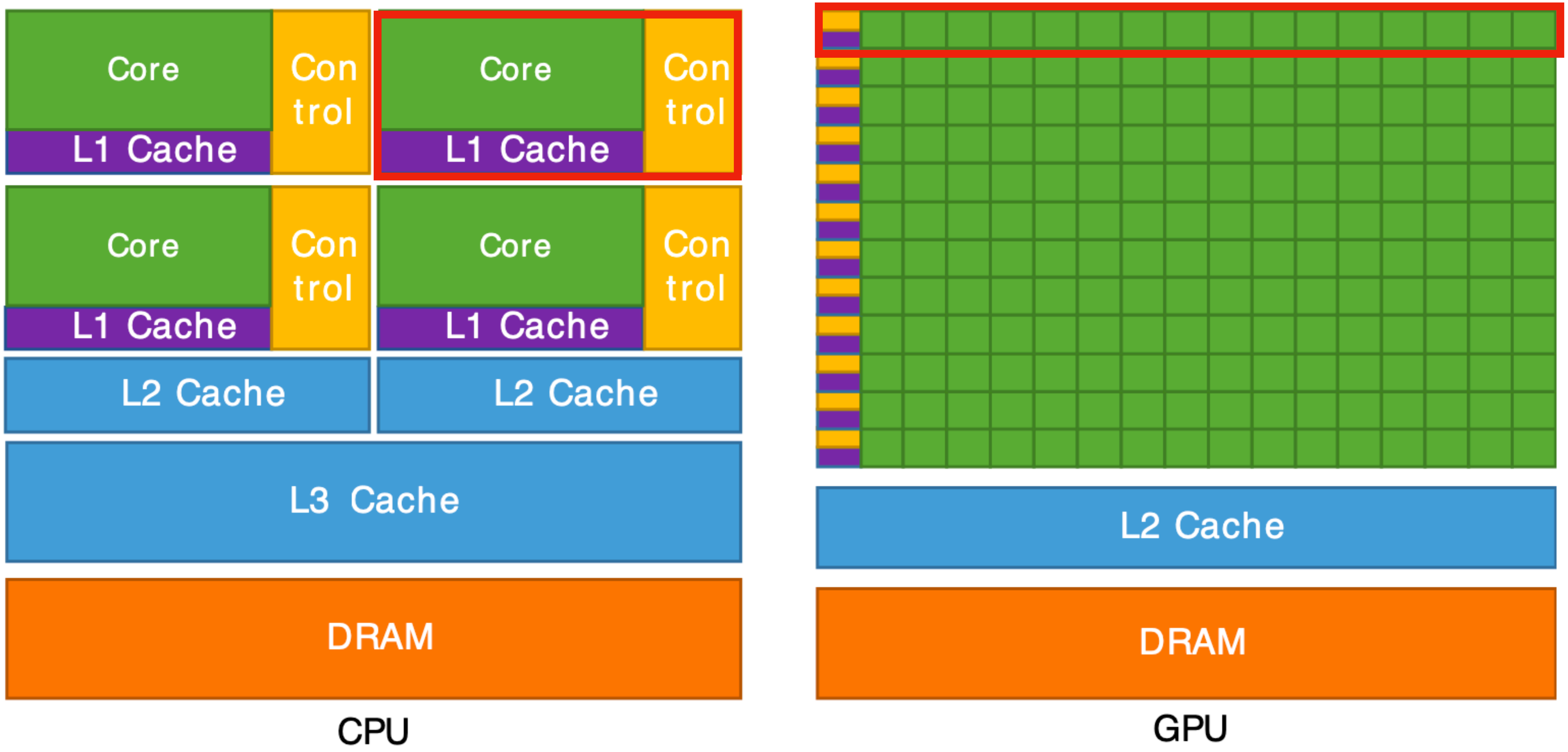
Introduction and motivation

GPUs on paper offer **far more raw compute power** than CPUs



Introduction and motivation

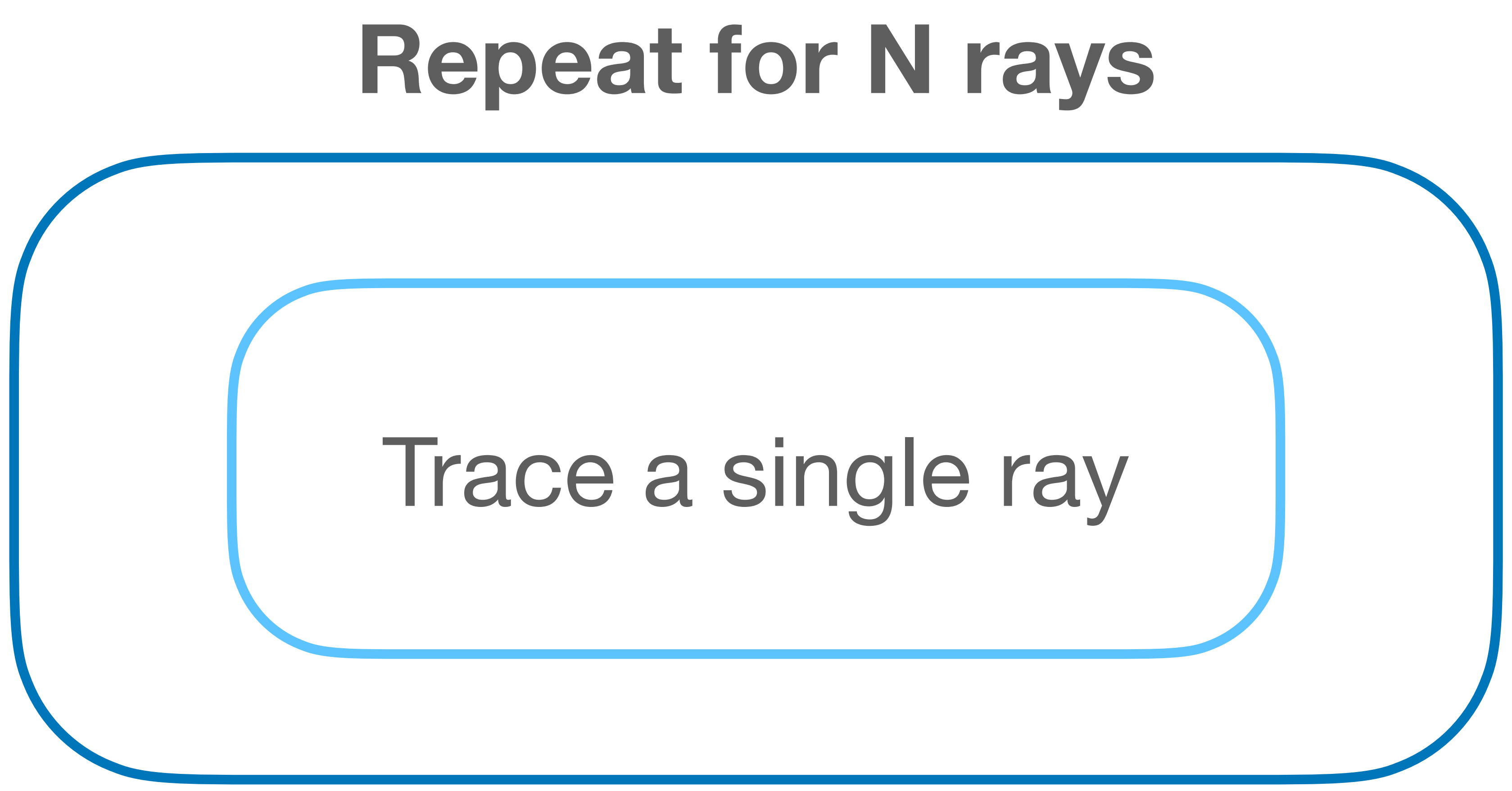
The microarchitecture and data models are **very different.**



What works well on GPUs

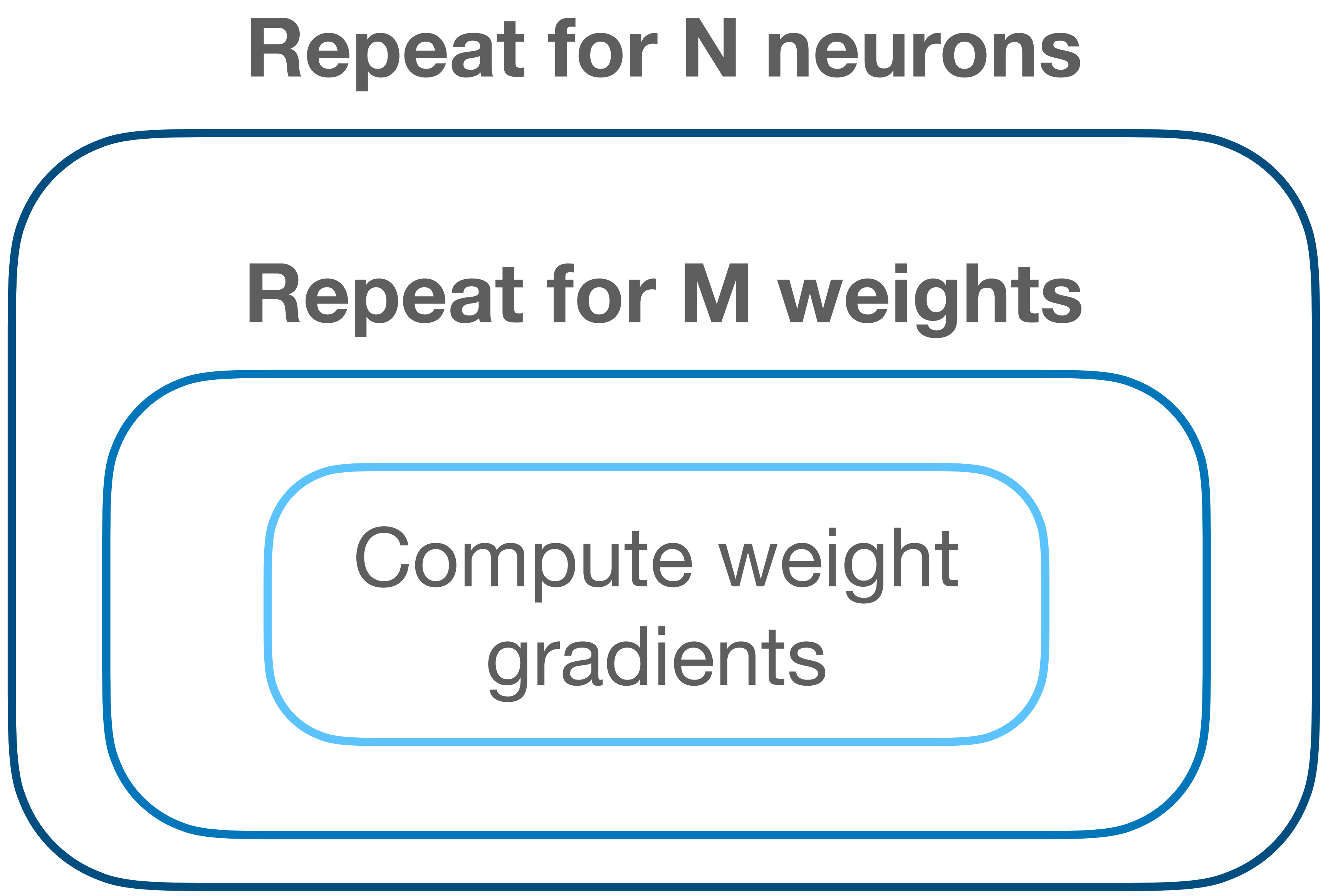
- **Ray tracing** and image processing

  High degree of **parallelism**, little data **dependency**



- **AI** and **Machine Learning** (e.g. deep neural net back propagation)

  Lots of operations can be run **independently**, mostly reliant on **linear algebra kernels**



What about PDE solvers?

CONS

I) **Hyperbolic PDEs** describe the transport of information.

   a. **Causal structure** —> **data dependencies**

   b. Need for **communication** and **synchronization**

II) Large **I/O** and **memory** requirements

III) GPUs are usually **optimized** for FP8-16-32 workloads.

PROS

I) Plenty of **parallelism**, lots of **grid sites** / **particles** to update

II) Can benefit heavily from **SMP** (~shared memory parallelism)

III) Mixed **hyperbolic** / **elliptic** systems could have even **larger** benefits.

What about PDE solvers?

Only **one way to know** for sure.

-> We are developing a **new GRMHD framework** on GPU backends aimed at:

- Exploring the **applicability** of heterogeneous computing to computational astrophysics.

- Building a **modern** and **future-proof** tool for research.

Codename: **G**eneral **R**elativistic **A**strophysics **C**ode for **E**xascale.

Rest of this talk

1. Introduction to GPU computing
2. Application to GRMHD equations
   - Introduction to GRACE
   - Code Tests & Preliminary Results
   - What can we say about performance?

Introduction to GRACE

Two main components:

1. `p4est` AMR library

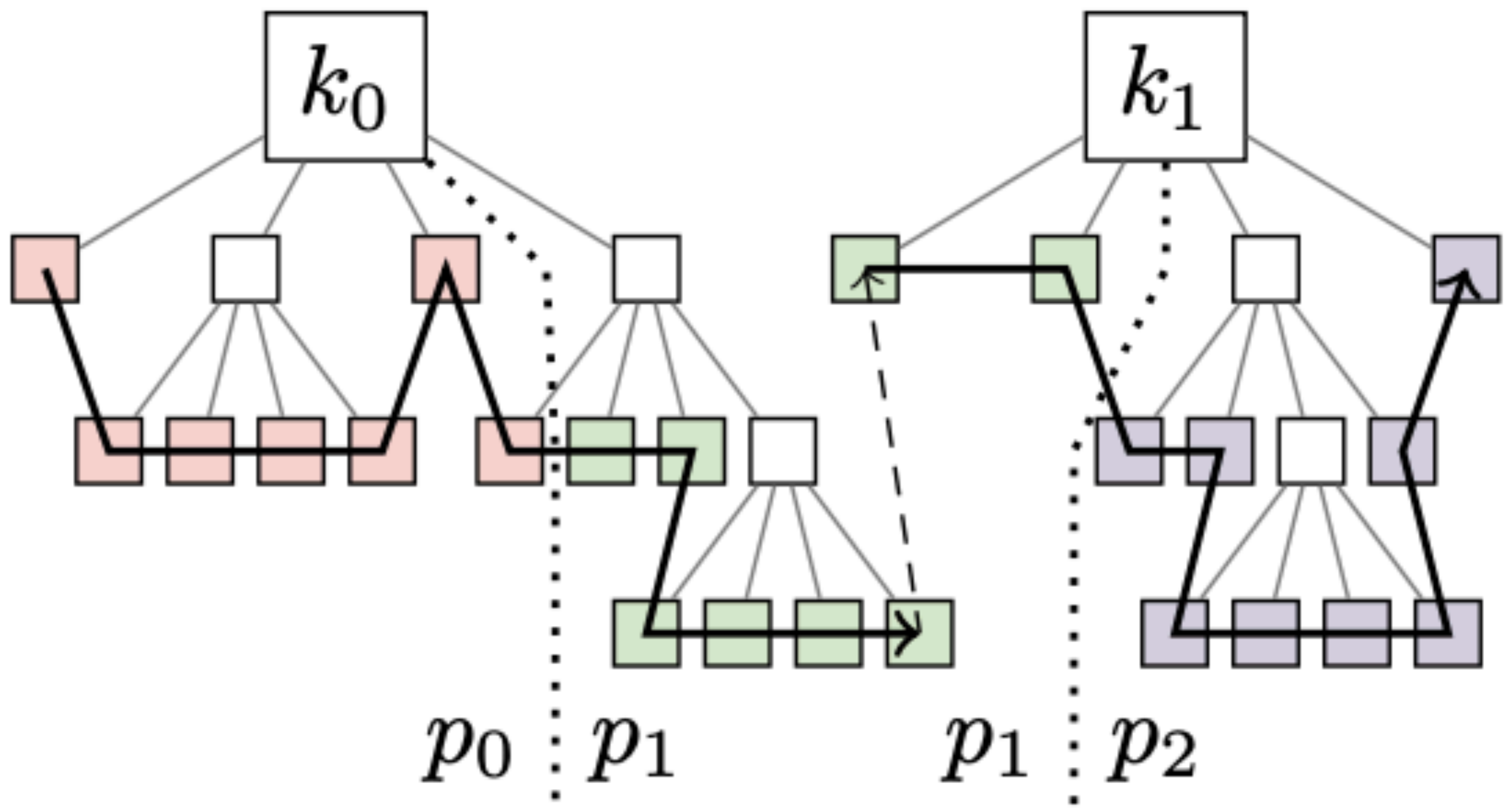
   Grids with adaptive resolution are a fundamental ingredient of any code that aims at serious scientific contributions.

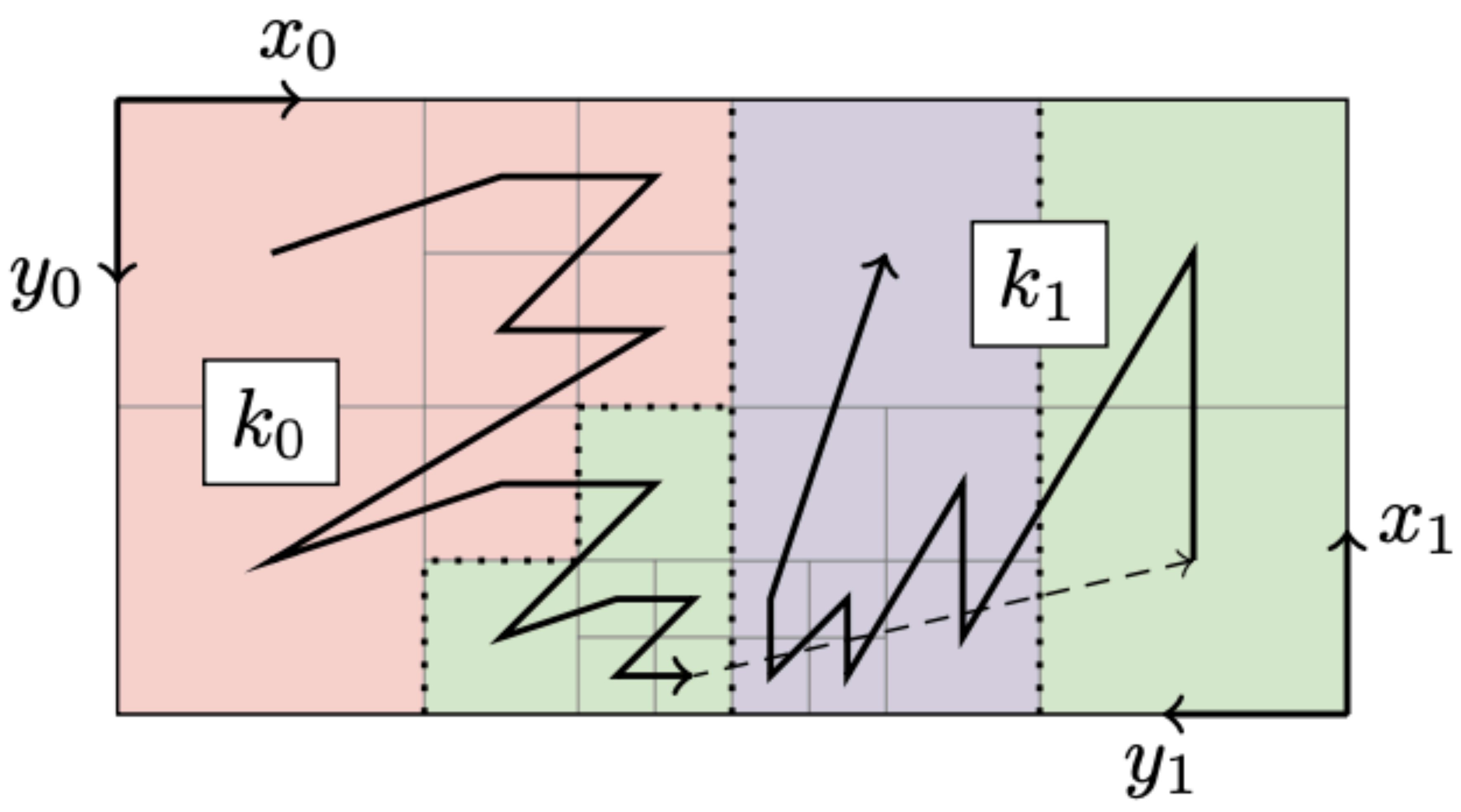
2. `Kokkos` Performance Portability Layer

   GPUs are complex and varied (different vendors, different APIs) and a software layer in between the physics code and the silicon helps to mitigate these challenges.

`p4est` AMR library

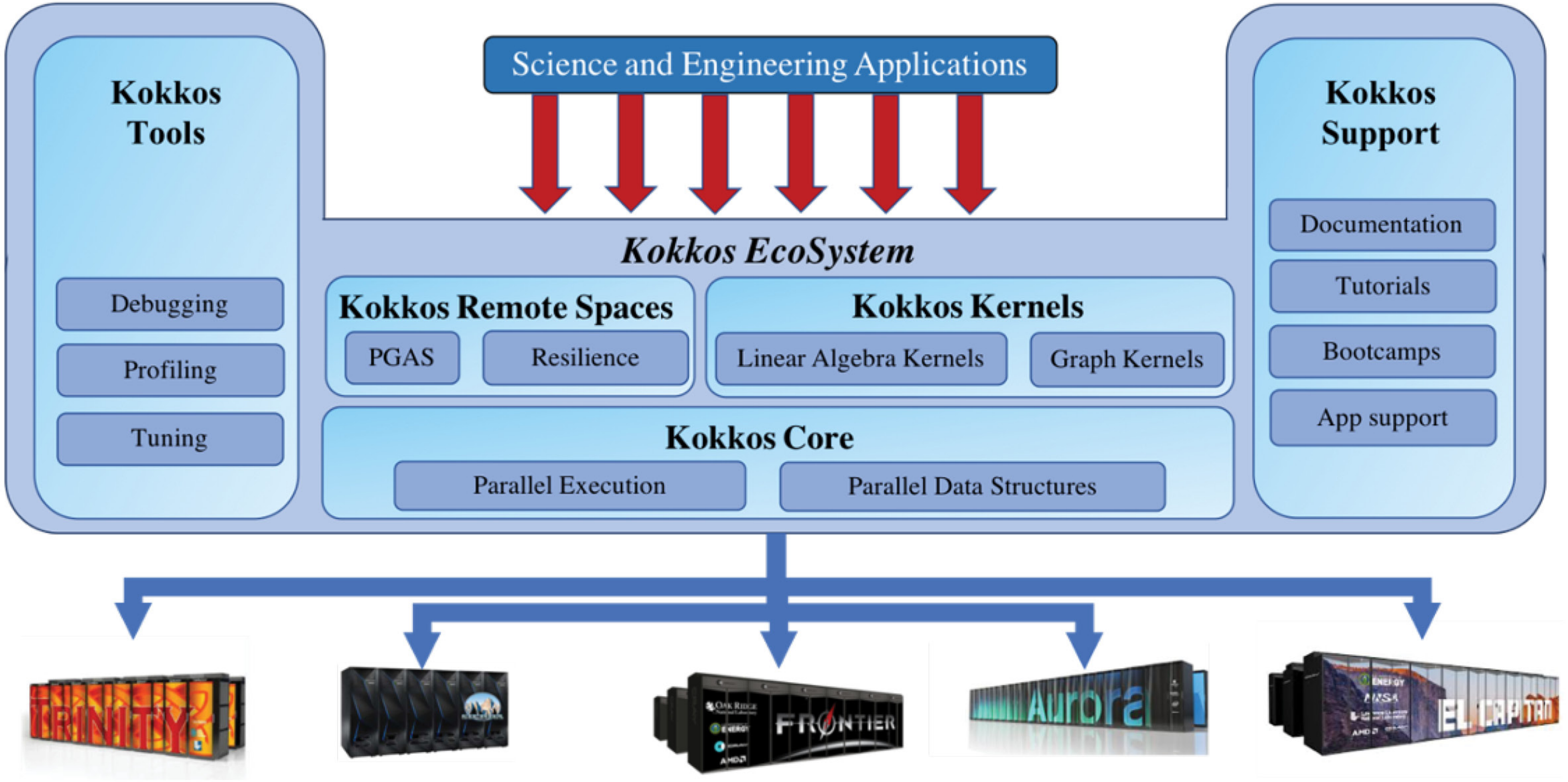
- **`p4est`** only handles the grid, **not** the data
- More dev work but ideal for GPUs
- Data always sits on the GPU
- AMR routines (prolongation, restriction, ghost zones) are custom written **GPU kernels**





`Kokkos` Performance Portability Layer

`Kokkos` is used in GRACE to **offload** work to GPU devices.



Code Tests & Preliminary Results

Two **model** equations:

i) ***Scalar advection***

Simplest **hyperbolic** equation. Test of basic finite-volume + AMR infrastructure.

ii) ***Burgers equation***

Nonlinear hyperbolic PDE —> **H**igh **R**esolution **S**hock **C**apturing methods required to handle discontinuities.

GRMHD module currently under testing

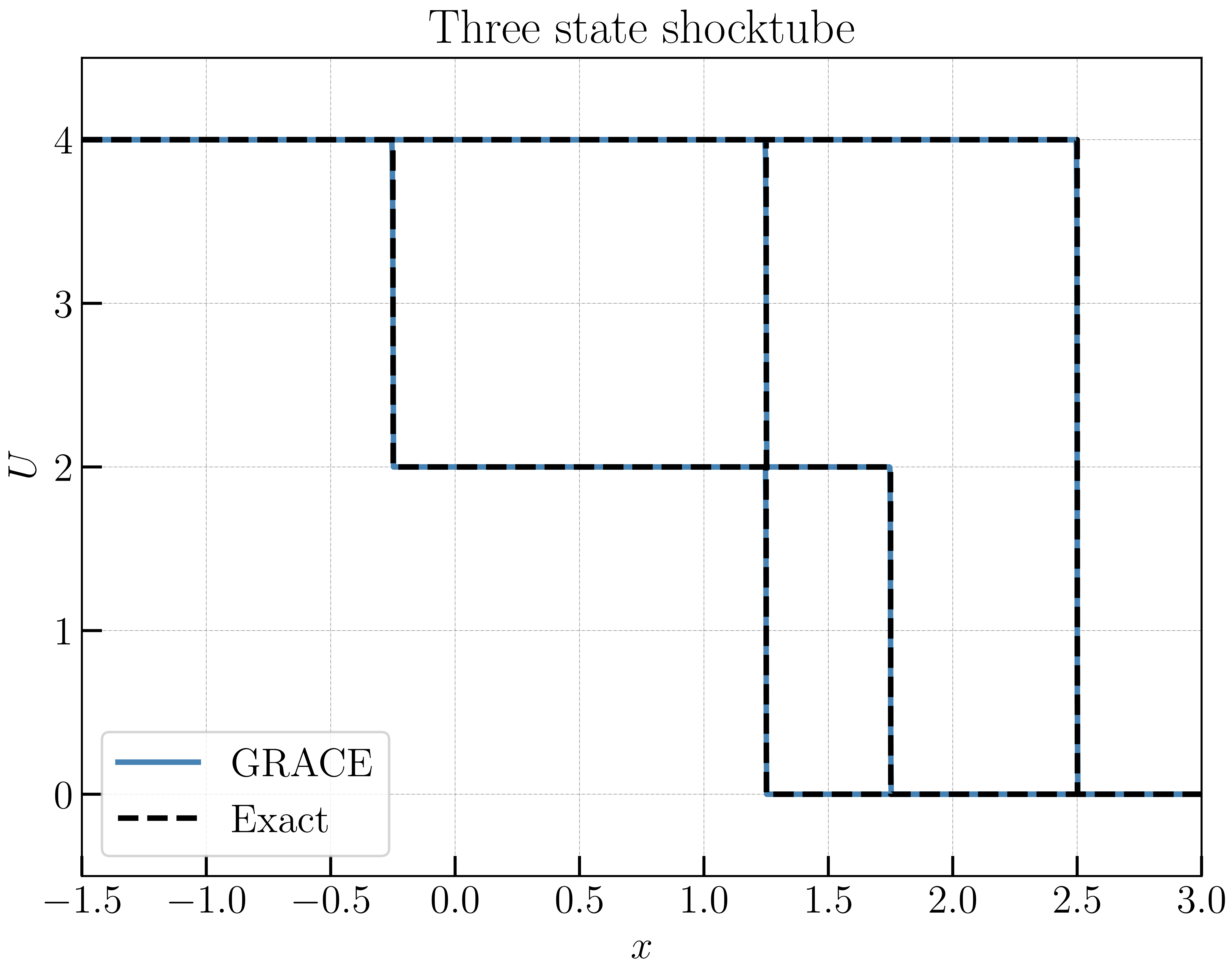
Burgers equation

- **HRSC** solvers for **nonlinear** PDEs implemented on **Cartesian** grid (2D and 3D).
- Currently supported reconstruction algorithms: **minmod**, **monotonized-central, WENO (3rd/5th order).**
- Currently supported Riemann solvers: **HLLE**.
- Prototypical PDE system: **Burgers' inviscid** equation.

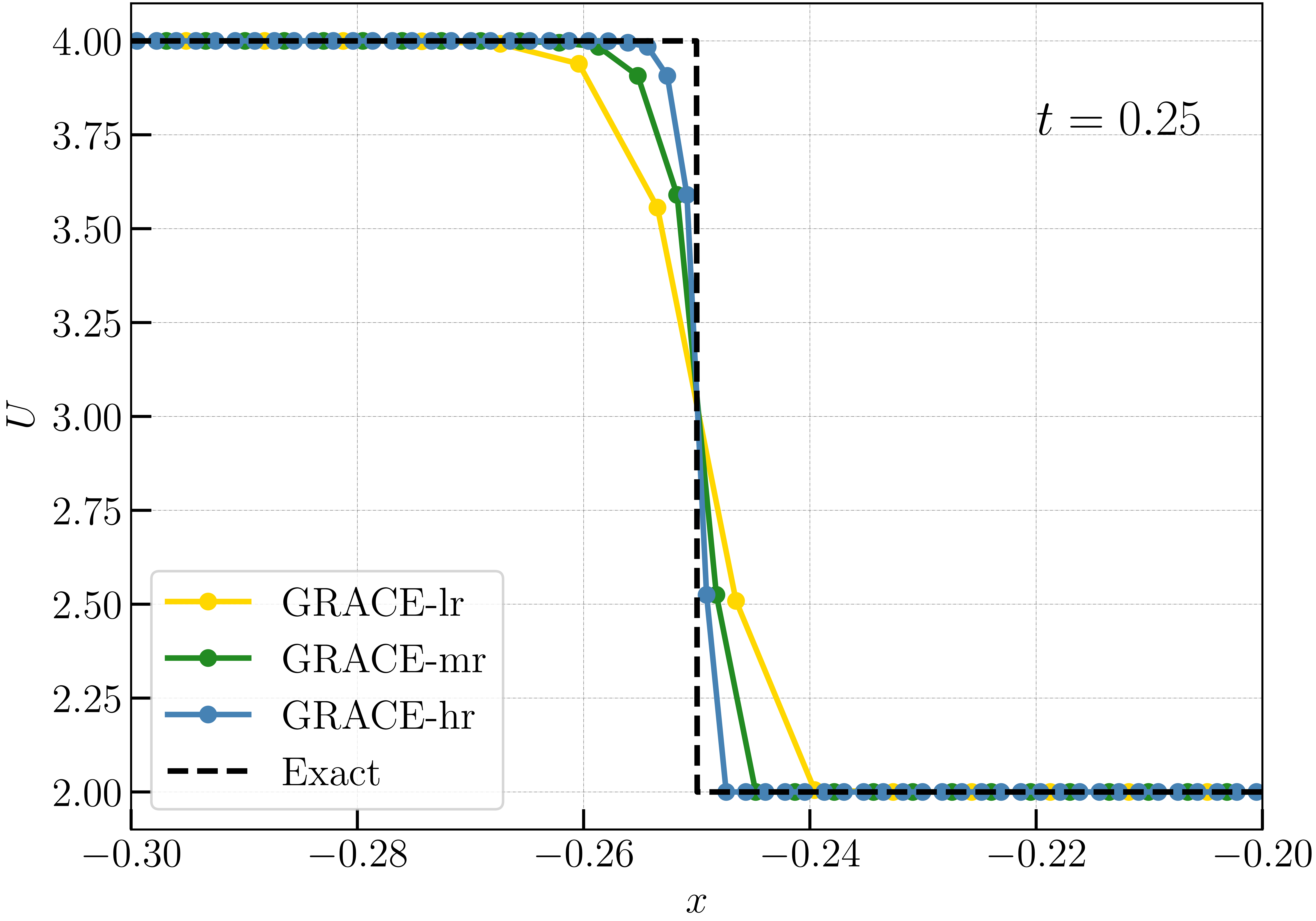
$$\boxed{\partial_t U(\mathbf{x}, t) + \frac{1}{2}\partial_x U(\mathbf{x}, t)^2 = 0}$$

Code Tests & Preliminary Results: Burgers equation

- Three-state **shock-tube** for Burgers' equation
- Solved in 3D with **uniform mesh** refinement and Runge-Kutta 2 time-stepping.
- The reconstruction method is **MC2**.
- This initial data leads to a double shockwave.



Three state shocktube



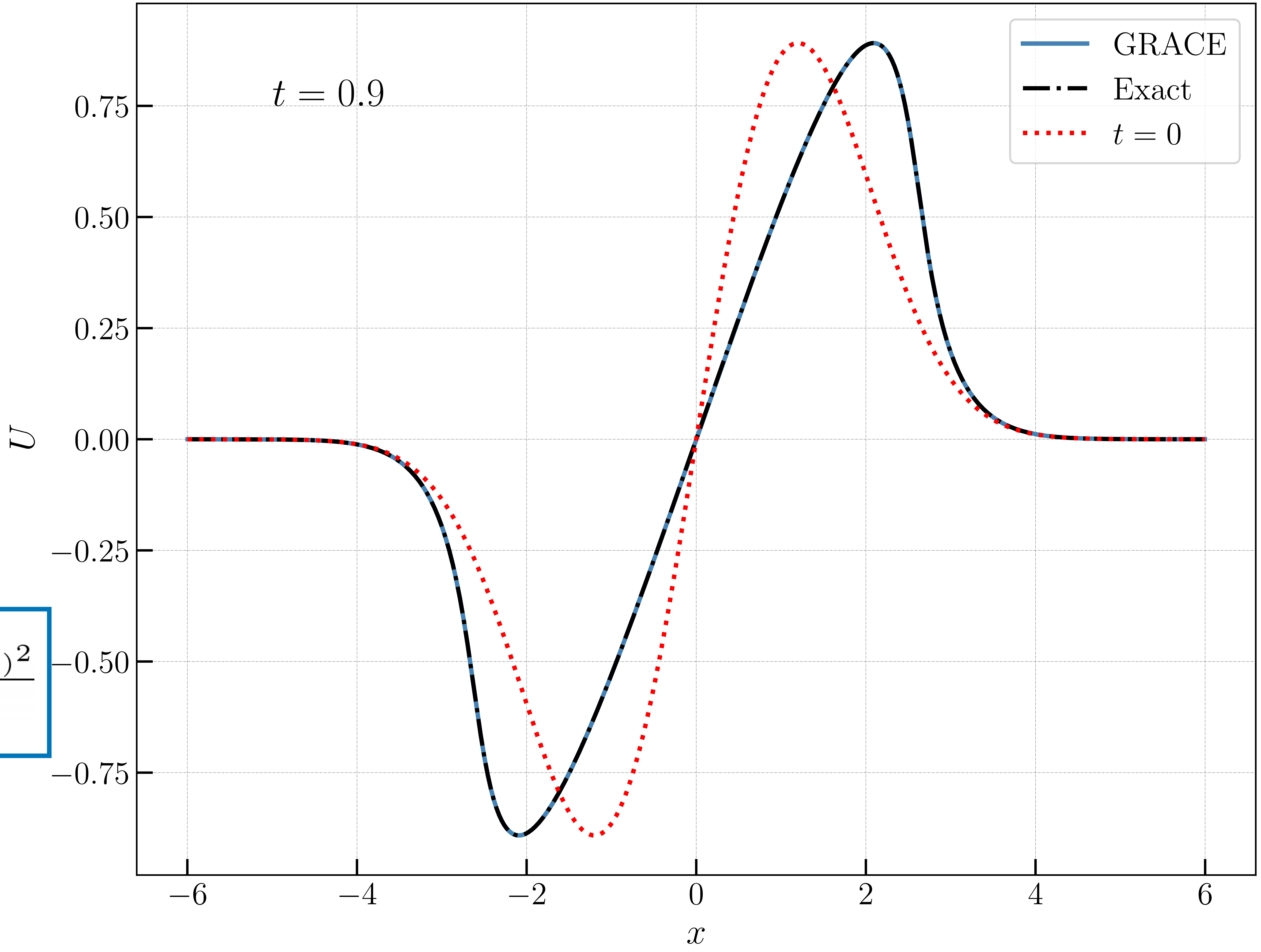
Code Tests & Preliminary Results

- **N-wave** test
- **Initial data**

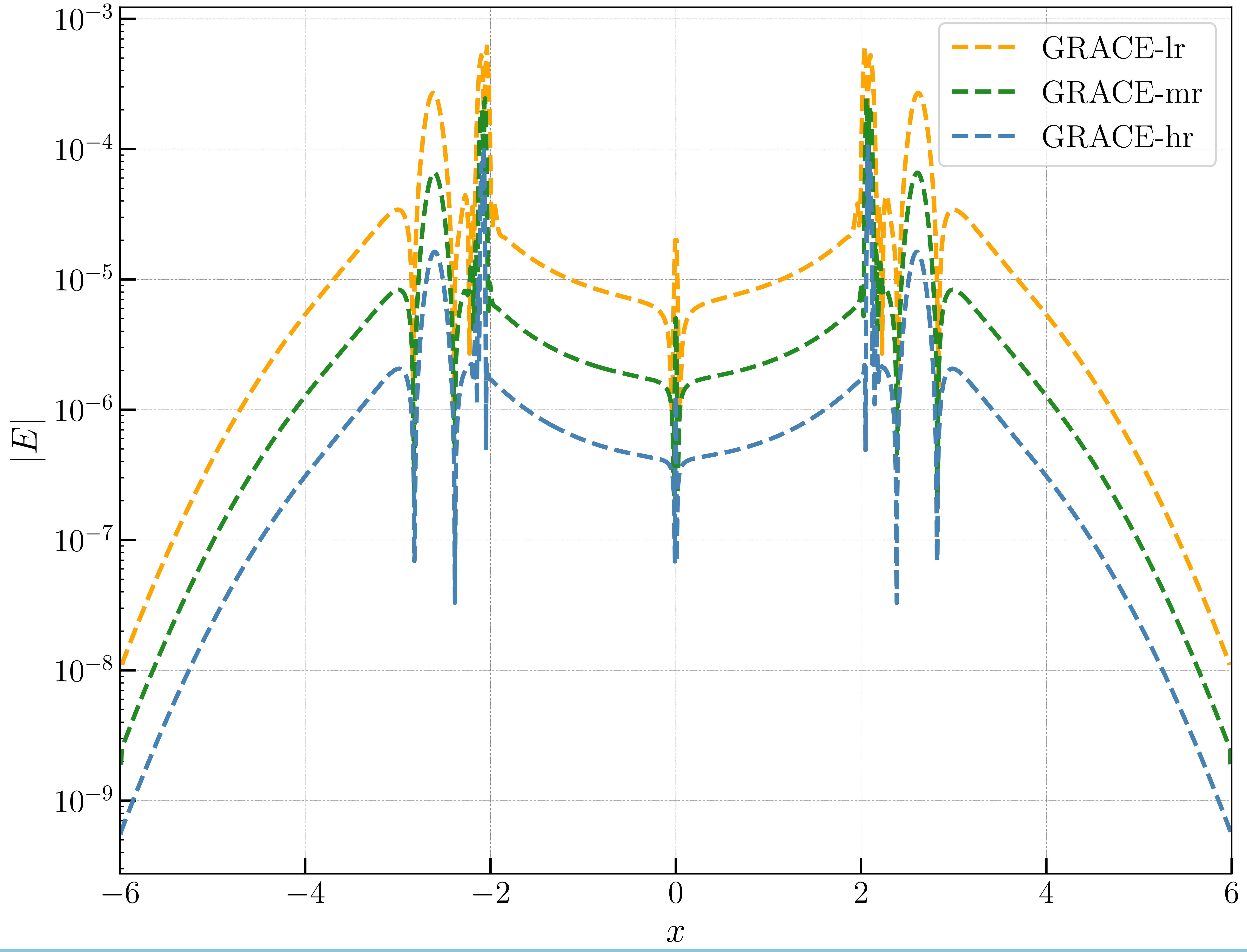
$$U(x,0) = e^{-\frac{(x-1)^2}{2}} - e^{-\frac{(x+1)^2}{2}}$$

- **Solution:**

$$U(x,t) = e^{-\frac{(x-U(x,t)t-1)^2}{2}} - e^{-\frac{(x-U(x,t)t+1)^2}{2}}$$

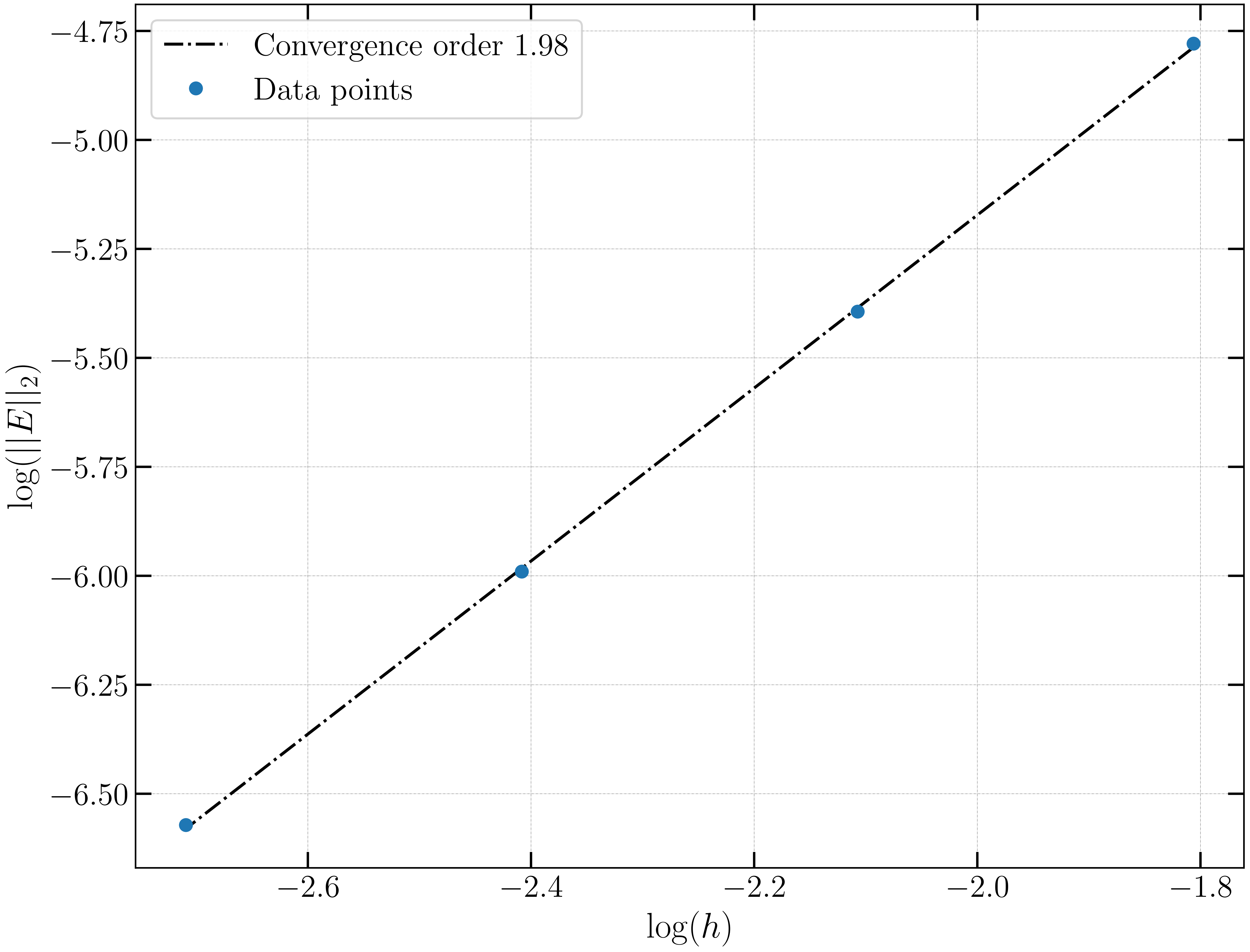


Results



GRACE-lr
GRACE-mr
GRACE-hr
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
$|E|$
-6
-4
-2
0
2
4
6
x

Order of Convergence



What can we say about performance?

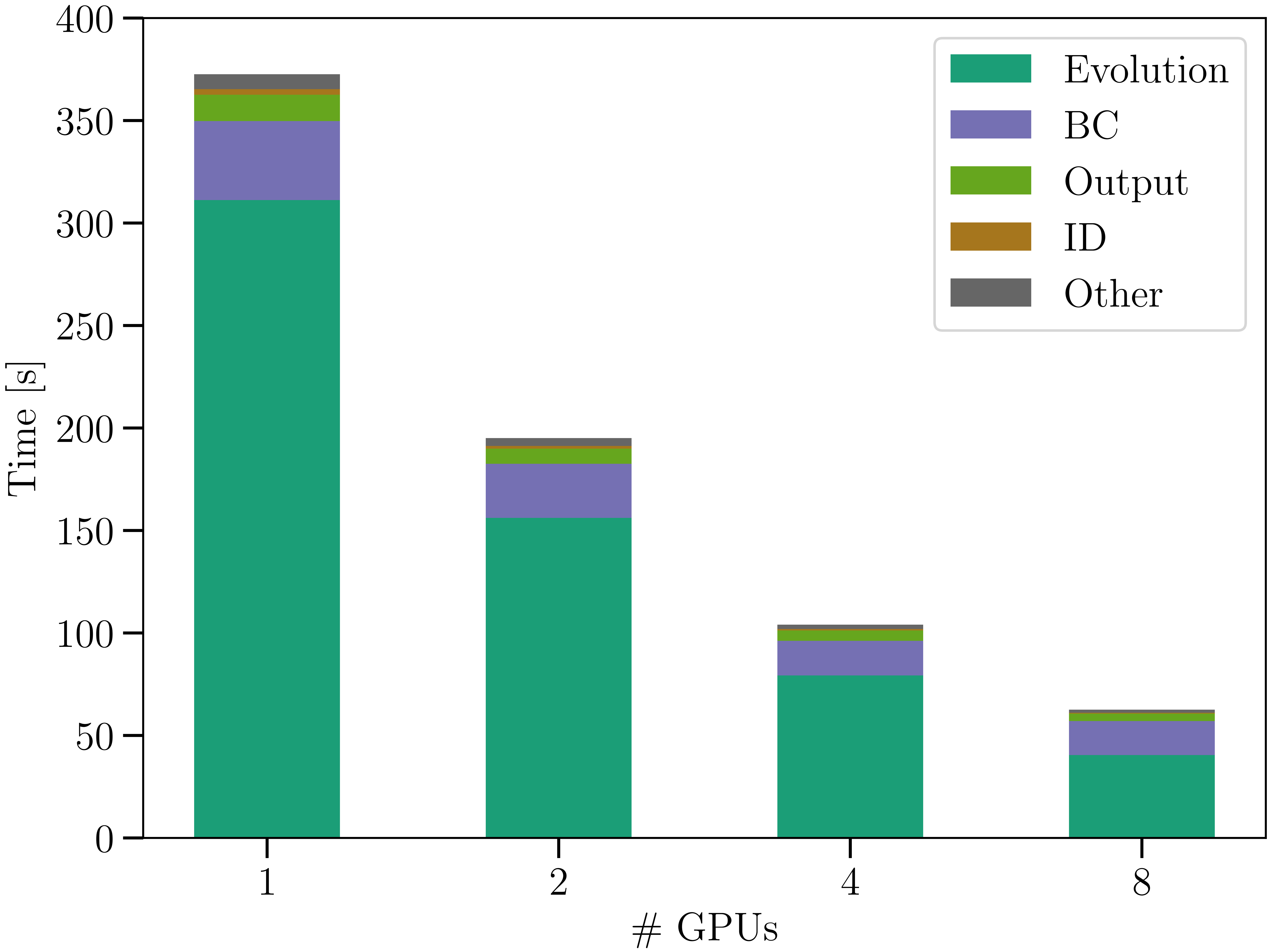
Profiling codes on heterogeneous systems is **hard.**

- Scaling is a measure of how well an application performs on a **large number** of compute resources
- **Strong scaling**: fixed problem size, increasing resources
- **Weak scaling**: problem size grows proportionally to the available resources
- Scaling alone is not always a good proxy for performance.

- Code **efficiency** on a single compute unit is largely **uncorrelated** to **scaling**
- Measuring efficiency can be very **challenging**
- Having a grip on “single-core” code performance is key for **effective optimization.**

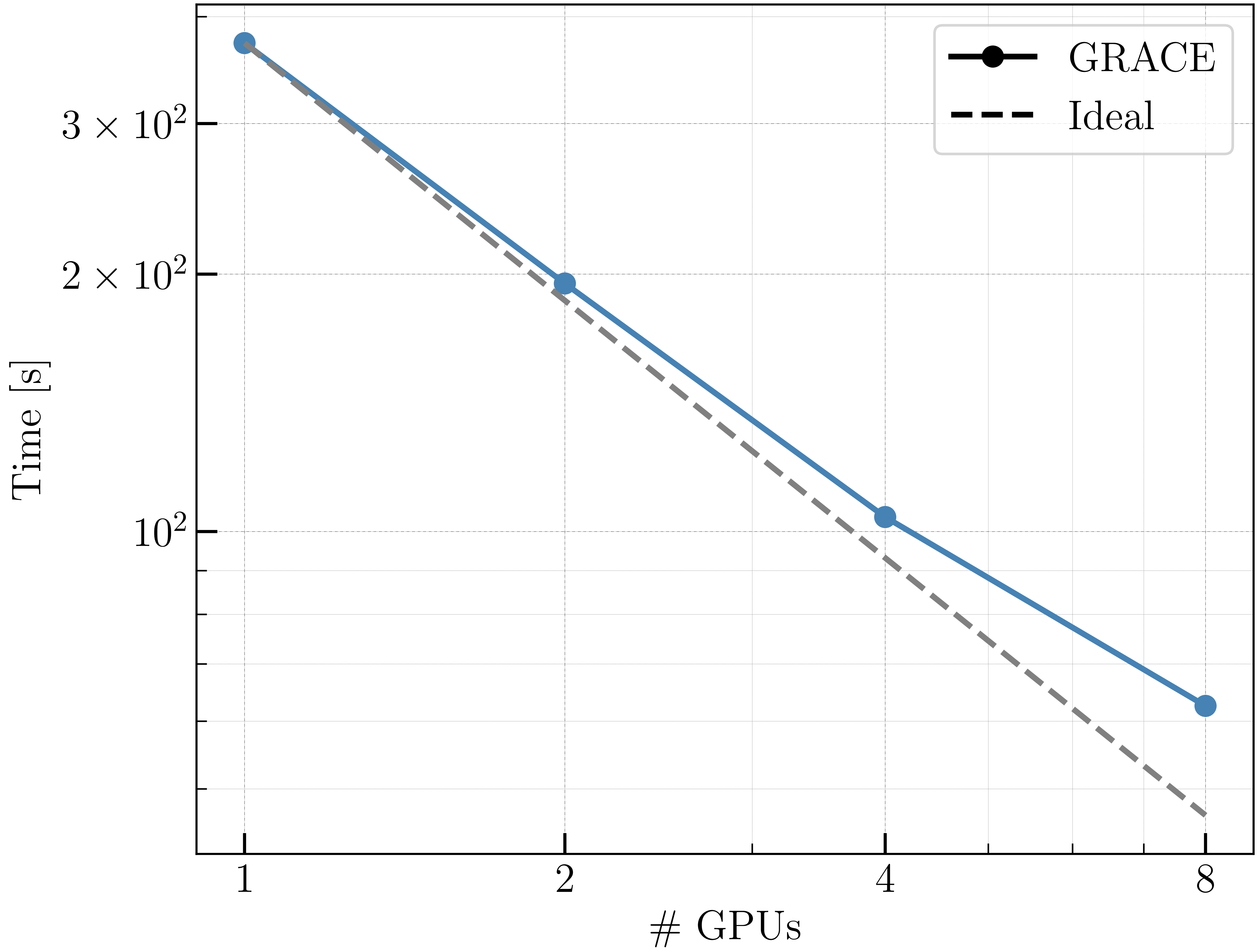
Profiling case study: Unigrid simulation

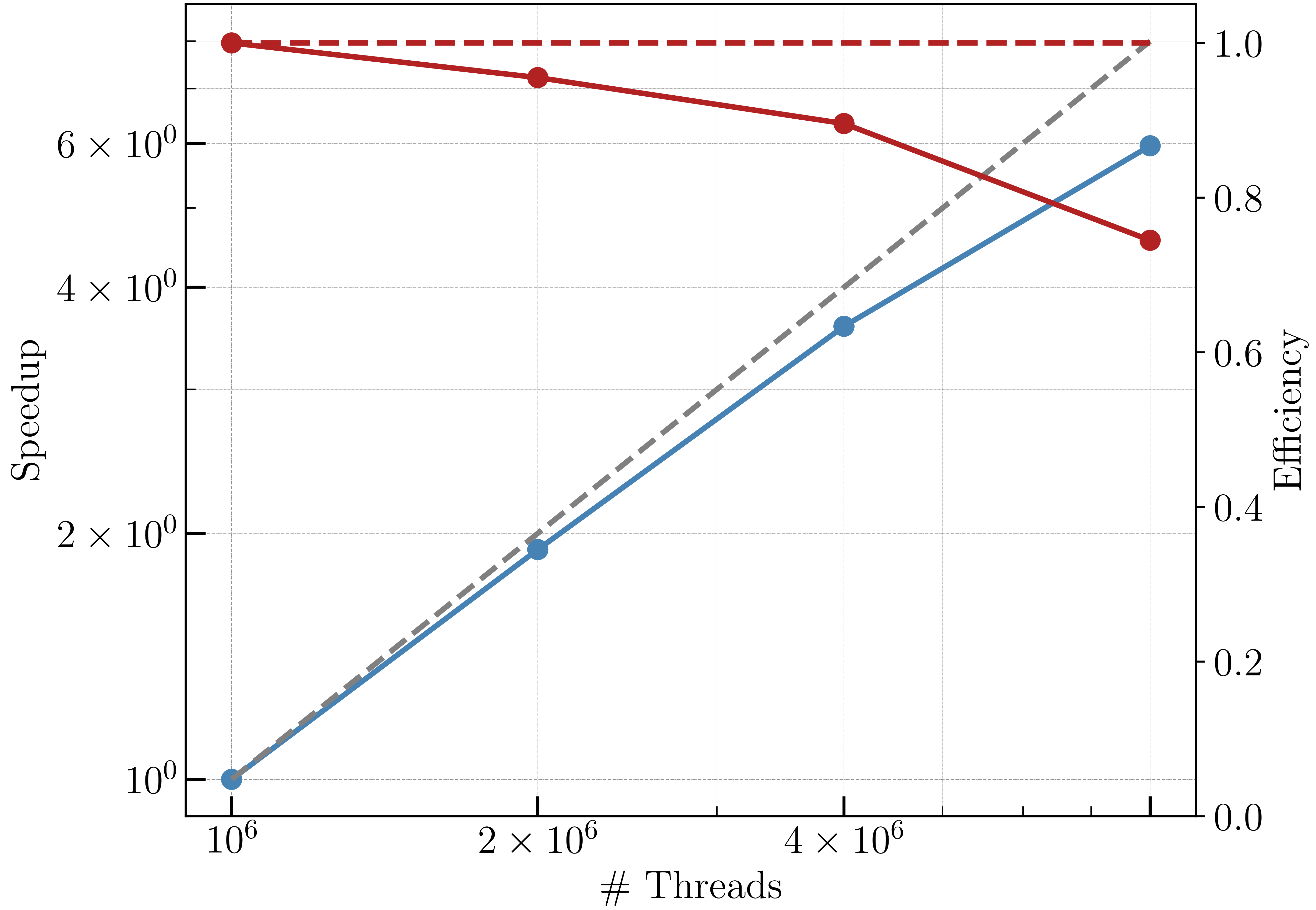
- Unigrid simulation of Burgers equation N wave test case.
- 5 levels of refinement with 16x16x16 points / block + ghost zones.
- **Take-away**: Over 80% of the time is spent doing useful calculations



Profiling case study: Unigrid simulation

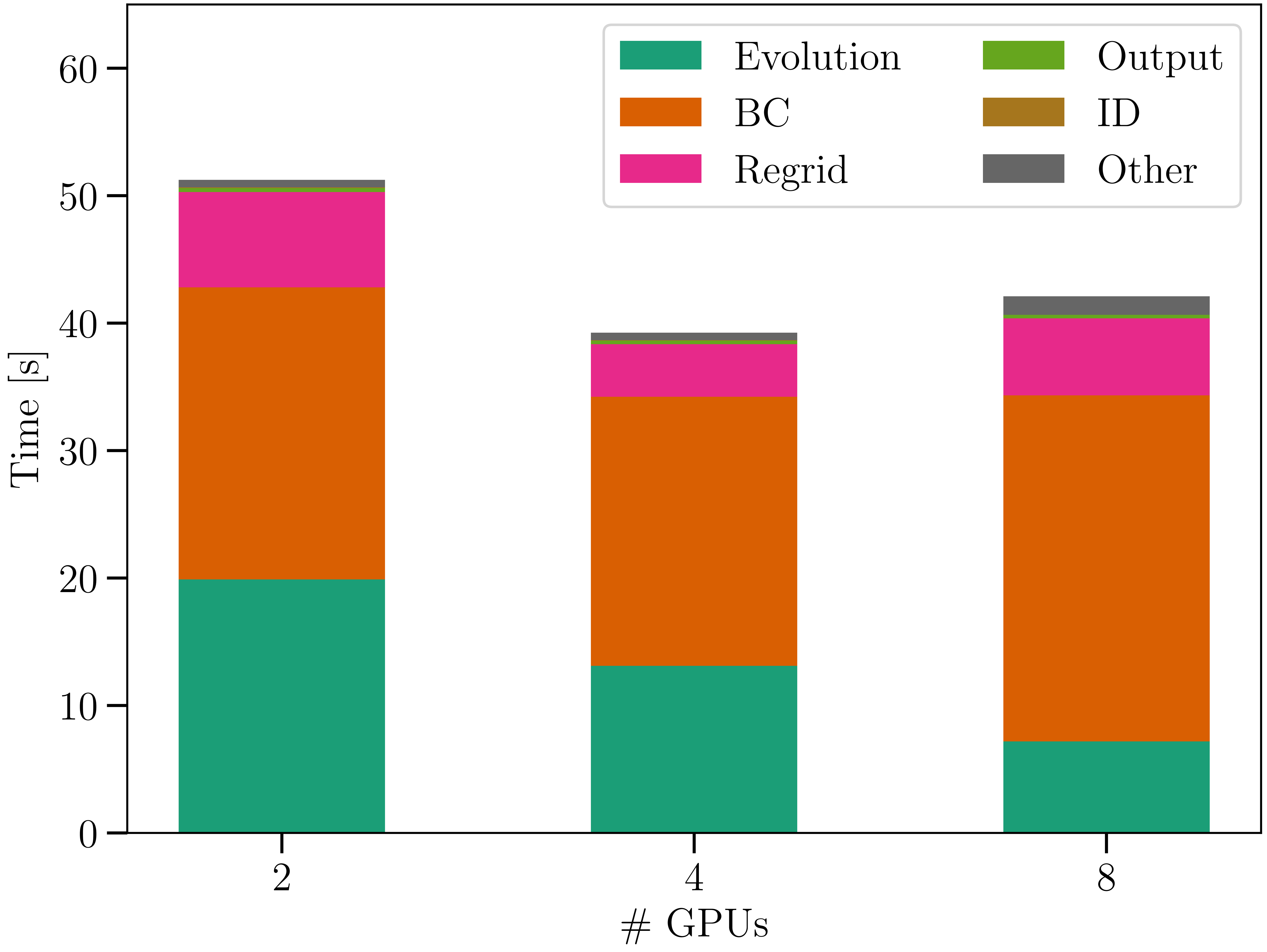
- The code scales well on the (limited) available resources.
- **Caveat**: MPI not properly fine-tuned for the system.
- **Take-away**: need a production environment to properly assess performance.





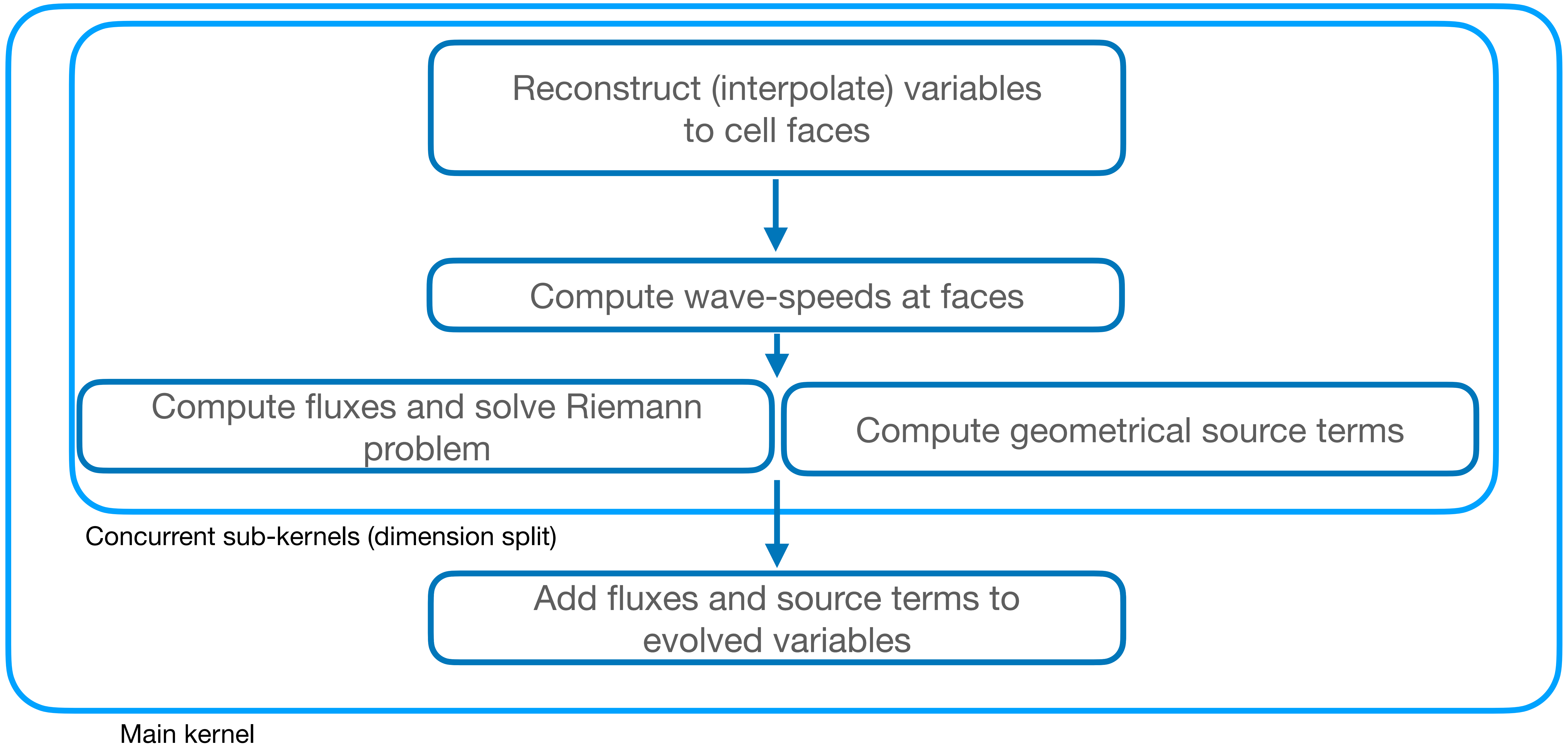
Profiling case study: AMR simulation

- AMR simulations of Gaussian pulse advection with periodic boundaries
- 5 levels of refinement with 16x16x16 points / block + ghost zones.
- **Take-away**: Hanging interfaces are costly to handle



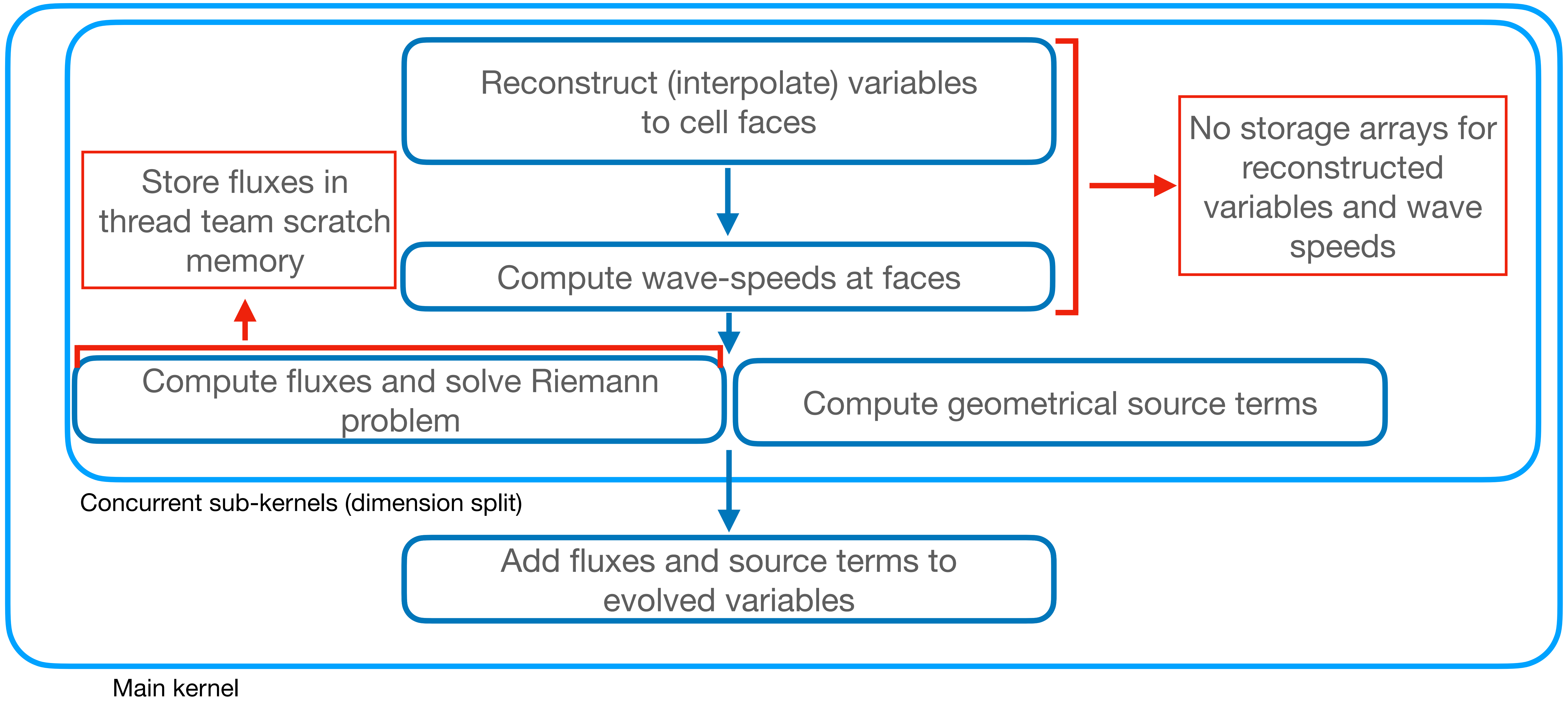
Code performance: evolution kernel

- Evolution: most **time-intensive** section.
- **Schematically** consists of a series of directional loops and a final loop to add sources
- Performance counters sampled with low-level device profilers



Code performance: evolution kernel

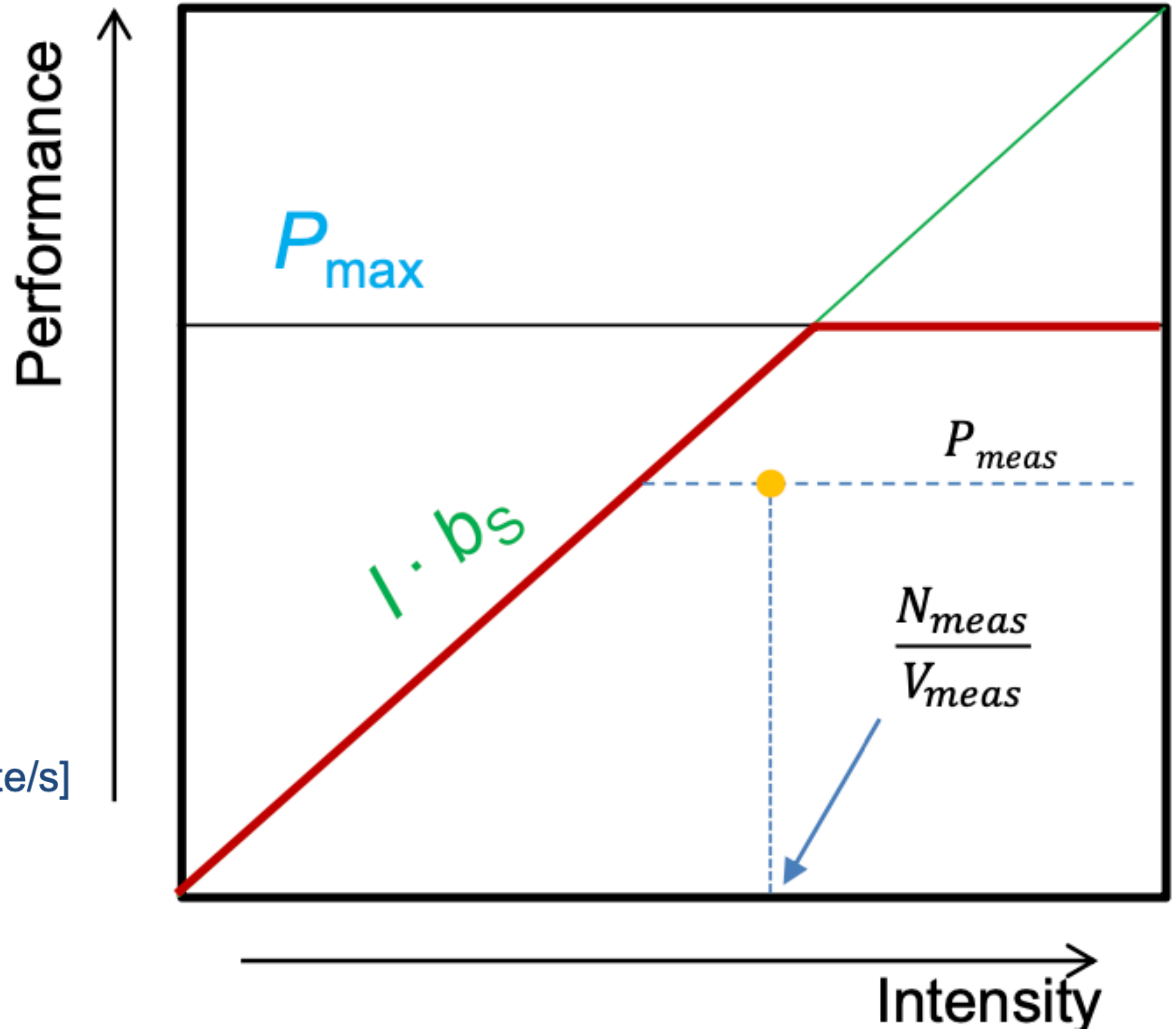
- **Heuristic** optimization based on **saving memory** transfers where possible
- **Remove** intermediate **storage** at the price of **extra computations**
- **Directional dependence** reduces vectorization performance



Code performance: roofline model

- **Optimistic** "speed of light" model for resource utilization.
- Applies to a **single** computational **kernel**.
- Bottleneck either:
  - **Execution of work** → P_{peak} [flop/s]
  - **Data path** → $I \cdot b_S$ [flop/byte x byte/s]

$$P = \min(P_{\text{peak}}, I \cdot b_S)$$

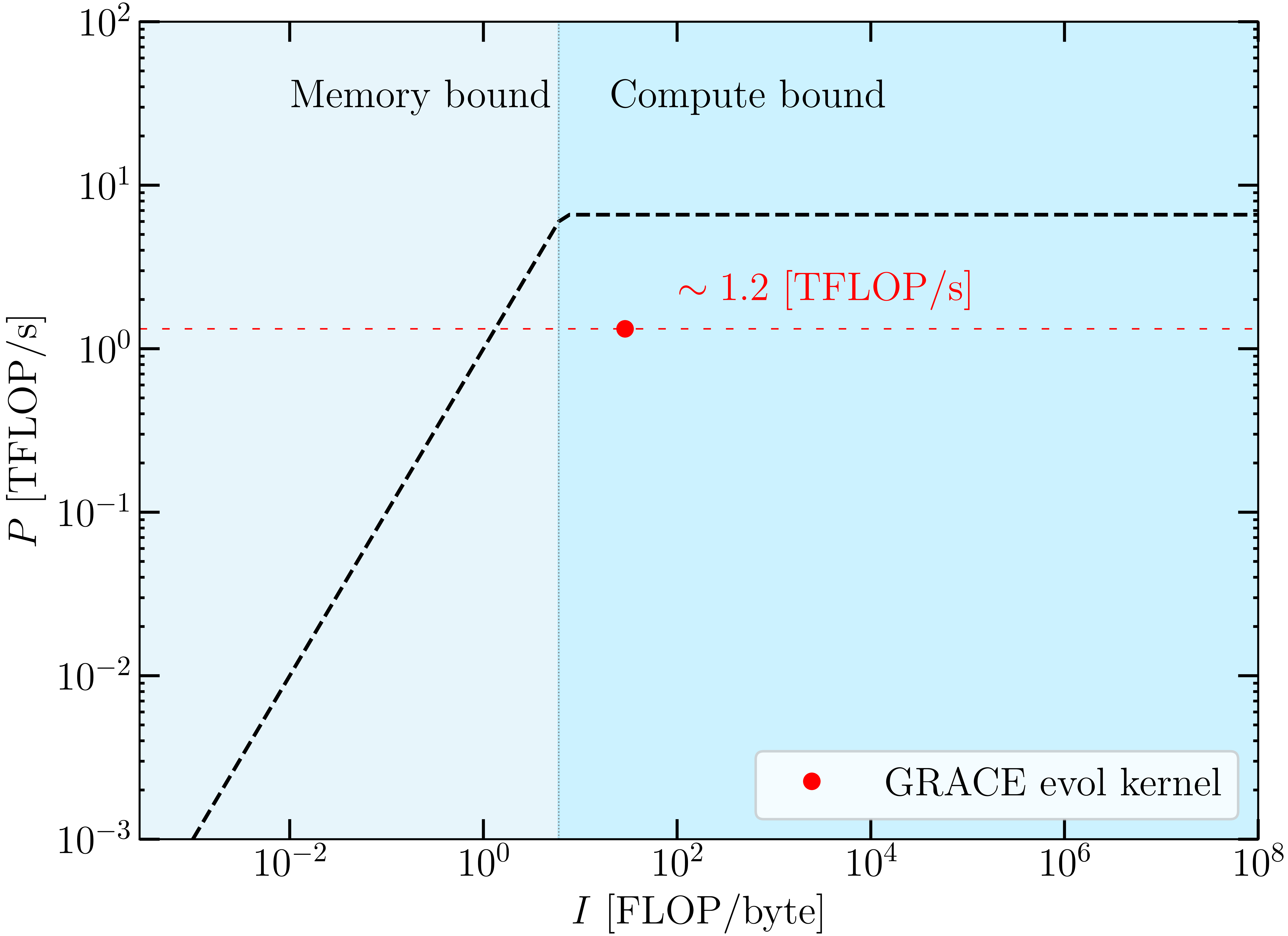


Code performance: roofline model

- Evolution kernel **peaks** at **~ 1.2 TFLOP/s**
- Tested on a single AMD Mi50 card
- Theoretical peak for **FP64** workload:

~ 6.5 TFLOP/s

$$P = \min(P_{\text{peak}}, I \cdot b_s)$$



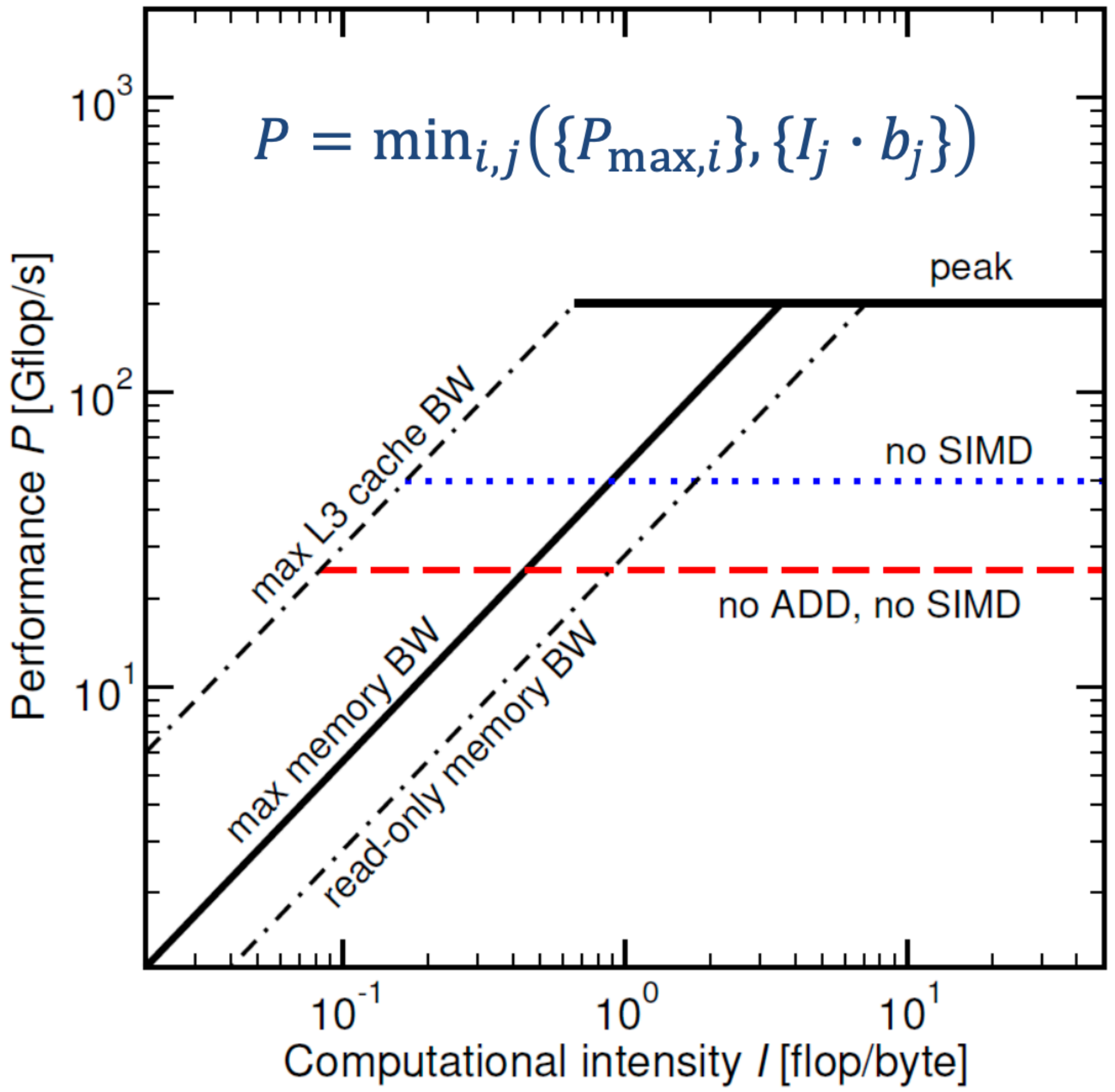
Code performance: roofline model

Pinning down compute bottlenecks is **complex.**

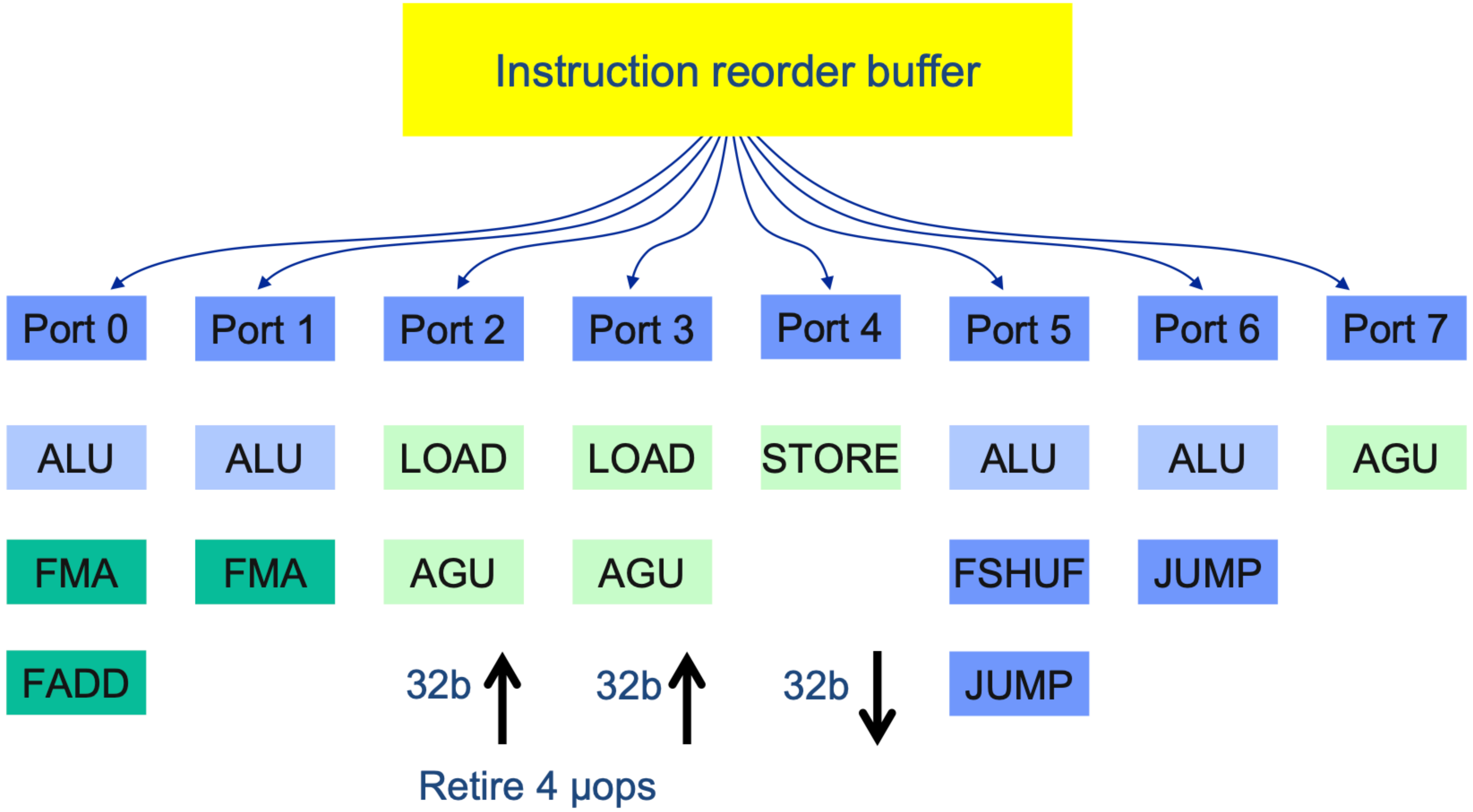
i. L2 **cache misses not** accounted for in simple model.

ii. Likely suffering from **port contention**

iii. **Vectorization** might be limited by loop dependencies



Example of more **realistic** roofline model diagram



Model port scheduler diagram for Intel Haswell chip

Conclusions

- We have a **functional**, oct-tree **AMR enabled framework** for solving non-linear hyperbolic PDEs on GPU backends.
- The code shows **promising single-GPU performance** and **good scaling** on the available in-situ resources.
- The tool is fully built in-house and can be **easily extended.**
- Further developments are in progress and we hope to be production-ready in the Fall.

Current landscape

BACKUP

- Theoretical Astrophysics relies on **HPC** for realistic and accurate simulations of complex systems.
- Overwhelming majority of existing codes are **complex** and built upon years **iterative improvements / modifications.**
- The basic algorithms can often be **streamlined** by careful reviewing of existing codebases.
- The basic computing **paradigm** has **shifted** since these codes where designed.
- **GPUs** are an (almost) obvious candidate for modern simulation codes infrastructures.

Current landscape: my take

BACKUP

- GPU offloading is **not** a drop-in replacement for traditional parallel programming paradigms.
- Current grid-based codes typically perform **very poorly** with a large number of **threads** (why?).
- Efficiency and performance require **knowledge** of both the basic **hardware architecture** and the **algorithm**.
- **Low-level** routines likely need to be revised to better suit the execution model of target machines.
- **High-level** algorithms also require modifications achieve the best possible performance on modern systems.

Introducing GRACE

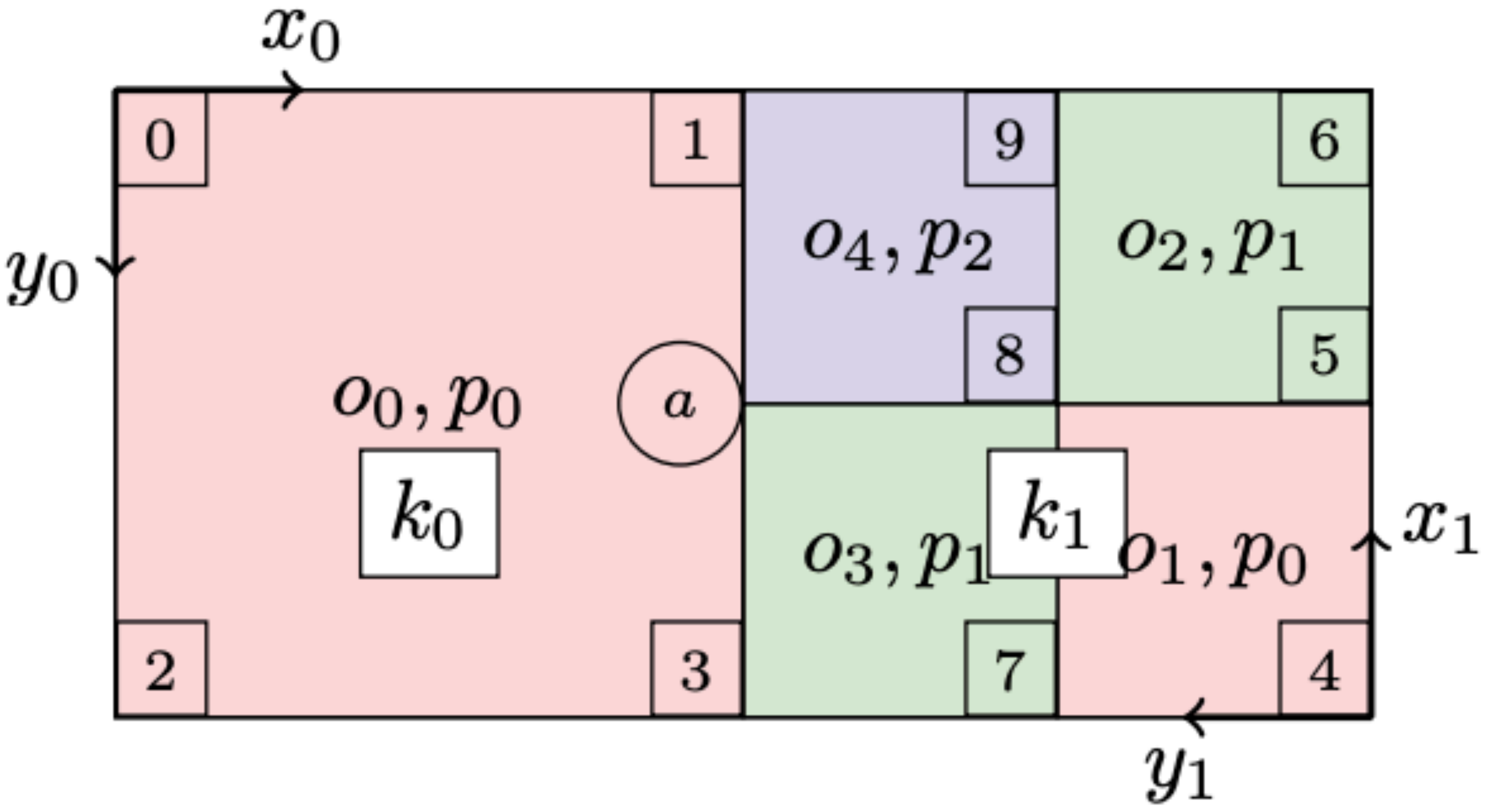
BACKUP

- Development of a **scalable**, **efficient** and **portable** GRMHD code using **GPUs** and **Discontinuous Galerkin** methods.
- Based on the **p4est** library for an efficient, low level forest-of-oct-trees AMR infrastructure.
- GPU offloading handled by the **Kokkos** library, with full support for **HIP** and **CUDA** and experimental support for **Sycle.**
- **Custom** AMR routines and algorithmic flow of the code tailored to target system architectures.
- **Low-level** components of the code (memory allocation, numeric kernels…) redesigned to perform well on new systems.

Example algorithm: ghost-zone exchange

BACKUP

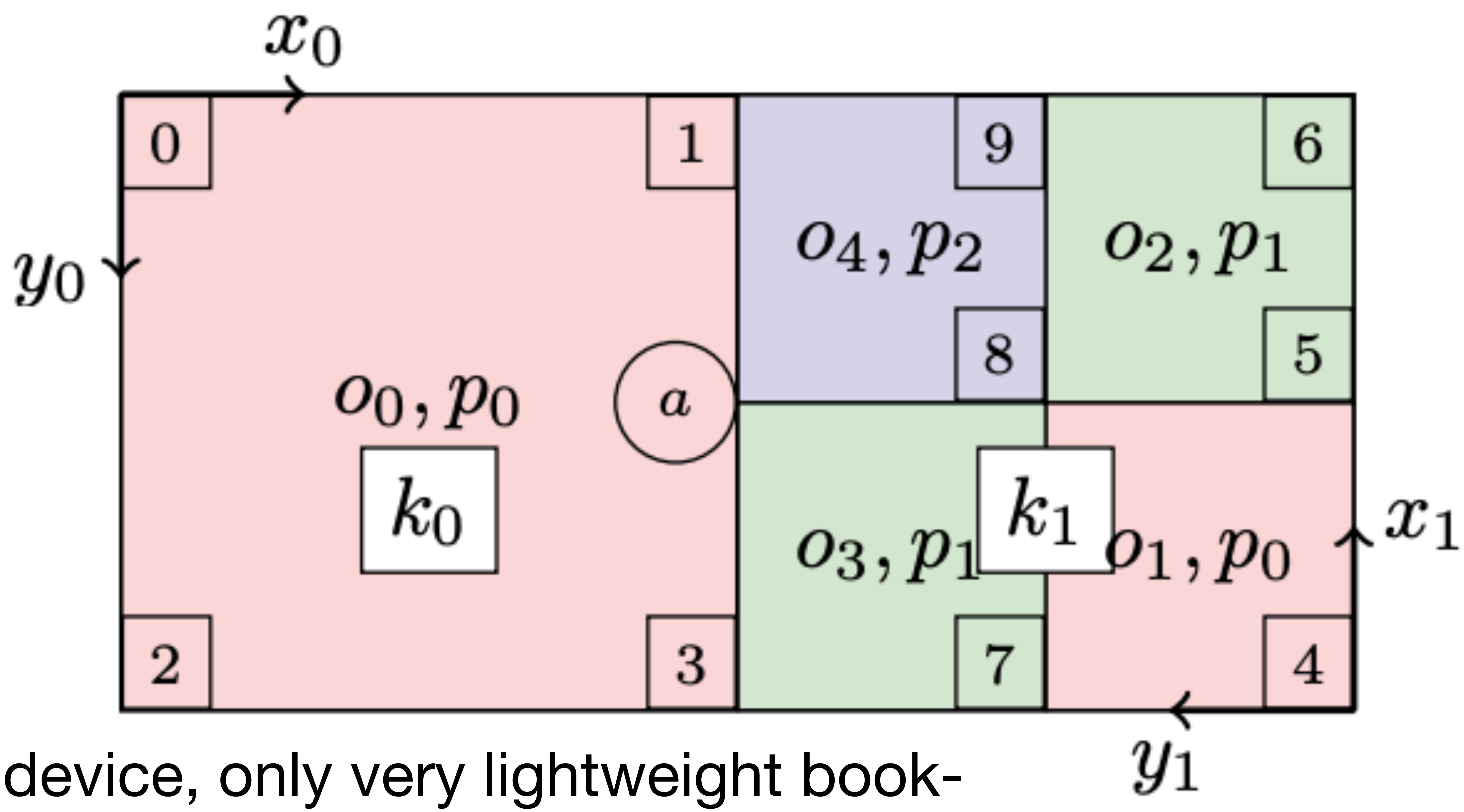
- **1 (Host):** p4est loop over quadrant faces to find neighbors
- **2 (Host-Device):** initiate asynchronous data exchange for halo quads
- **3 (Device):** Apply physical BCs while waiting for data



Example algorithm: ghost-zone exchange

BACKUP

- **4 (Device):** copy interior simple ghost zones and fill coarse ghost cells from fine data
- **(Host-Device):** Exchange coarse data with filled ghost zones
- **6 (Device):** Fill fine ghost zones from coarse data



Variables **always** sit on device, only very lightweight book-keeping data is exchanged (integers)