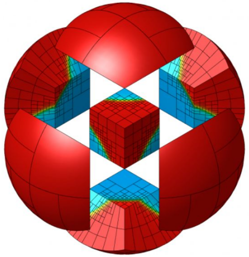
TrixiCUDA.jl: CUDA Support for Solving Hyperbolic PDEs on GPUs

Huiyu Xie (@huiyuxie)
JuliaCon 2025

juliacon

PDEs with GPU Acceleration



MFEM: Scalable Finite Element Discretization Library (C++)

- Supports CUDA, HIP, OCCA, etc.
- Heavily relies on batched matrix operations (e.g., cuBLAS) for acceleration.

deal.II: Library for solving PDEs with adaptive finite elements (C++)

- Supports CUDA and SYCL via Taskflow.
- Provides custom kernels for core algorithms, including merge, reduce, sort, and matrix multiplication.

Julia Programming and CUDA

Why Julia?

- Scientific Computing: Good FP, arrays, and parallelism.
- Users: Easy to program (compared to C++).
- Developers: JuliaGPU, rapid development, strong ecosystem.

Why CUDA?

- Mature support through CUDA.jl.
- Fine-grained control over kernel optimization.
- Strong package ecosystem (e.g., cuBLAS).

juliacon

Brief Introduction: TrixiCUDA.jl

Acceleration Sketch:

- Potential acceleration beyond matrix operations.
- Requires custom kernels due to flux computations, etc. in PDE solvers.
- Semidiscretization is heavy in computation but highly parallelizable.

Trixi-Framework

SciML

Trixi-GPU

JuliaGPU

Semidiscretization in PDEs

What is semidiscretization?

- Semidiscretization is a high-level description of spatial discretizations specialized for PDEs.

How to solve PDEs?

- Spatial discretization (i.e., semidiscretization) first, which reduces PDEs into a system of ODEs.
- Time integration is applied afterward to solve the ODE system.

Workflow Skeleton

Core acceleration happens in phase 3.

PDE Problem

`ode=semidiscretize(...)`

Semidiscretization is wrapped, turning the *PDE problem* into an *ODE problem*.

ODE Problem

Phase 1

`semi=SemidiscretizationHyperbolic(...)`

Pack equations, mesh, solver, initial and boundary conditions, and possible source terms.

Phase 2

Phase 3

`sol=OrdinaryDiffEq.solve(...)`

Solve *ODE problem* through a time integration loop. Semidiscretization takes effect here.

`ODEProblem(rhs!, u0_ode, tspan, semi)`

GPU Kernel Optimization

Common optimization methods:

- Maximizing occupancy
- Enabling coalesced global memory access
- Minimizing control divergence
- Tiling of used data with shared memory (avoid bank conflicts)
- Privatization (works well when there are plenty of atomic operations)
- Thread coarsening

Checkout one example here if you are interested!

juliacon

Takeaways about TrixiCUDA.jl

- **Julia**: Intuitive and accessible for both beginners and experts in scientific computing.
- **CUDA**: CUDA acceleration for semidiscretizations in PDE solvers.
- **Precision**: Support both single-precision and double-precision floating-point operations.
- **Optimization**: Specialized optimizations beyond matrix operations.
- **Expansion**: Researchers can add new features/algorithms to Trixi.jl and easily get GPU acceleration in TrixiCUDA.jl.

Big Open Problem

GPU Libraries like cuBLAS, cuSPARSE, cuSOLVER, cuDNN, cuFFT, etc.

Why no good PDE package?

- Too Complex and requires huge effort due to various problem types and GPU architectures (profiling/benchmark-based optimization is not good enough).

<u>How to give a fast implementation (parallel and optimized) based on the complexity?</u>

Q&A Session

Feel free to try some existing examples to solve the PDEs with GPU acceleration.

Any questions or concerns so far?

juliacon

Acknowledgment

Project Advisors

- Prof. Hendrik Ranocha (Johannes Gutenberg University Mainz, Germany)
- Prof. Jesse Chan (Rice University, U.S.)
- Prof. Michael Schlottke-Lakemper (University of Augsburg, Germany)

Upstream Developers

- Dr. Tim Besard (Lead Developer, JuliaGPU)
- Dr. Christopher Rackauckas (Lead Developer, SciML)

Julia Community

References