



# TrixiCUDA.jl: CUDA Support for Solving Hyperbolic PDEs on GPU

Huiyu Xie (@huiyuxie)  
JuliaCon 2025

*\* We are using Trixi.jl v0.11.17 and  
TrixiCUDA.jl v0.1.0-rc.3 for this talk.*

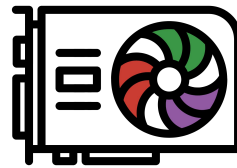


# Julia Programming and CUDA

PDEs with GPU acceleration: [MFEM](#), [deal.II](#), [libparanumal](#), etc.

## Why Julia?

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



JuliaGPU

## Why CUDA?

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



# Introduction to TrixiCUDA.jl



Trixi-Framework



Trixi-GPU

Core acceleration: Semidiscretization (i.e., spatial discretization)

Why semidiscretization?

- Discontinuous Galerkin (massive, neighbor-only GPU parallelism).
- Flux computations (custom kernels and optimization, not library-friendly).
- \*Weak task dependencies (CUDA stream concurrency).

*\* It is weak compared to time discretization (time stepping process).*

Public API design principles: Similar and intuitive

```
# Set up equation
equations = LinearScalarAdvectionEquation1D(...)

# Create DGSEM solver
solver = DGSEM(...)

# Set up tree mesh
mesh = TreeMesh(...)

# Create semidiscretization
semi = SemidiscretizationHyperbolic(...)

# Create ODE
ode = semidiscretize(...)

# Set up callbacks
callbacks = CallbackSet(...)

# Solve ODE
sol = solve(...)
```

**Trixi.jl APIs**

```
# Set up equation
equations = LinearScalarAdvectionEquation1D(...)

# Create DGSEM solver (GPU-enabled)
solver = DGSEMGPU(...)

# Set up tree mesh
mesh = TreeMesh(...)

# Create semidiscretization (GPU-enabled)
semi = SemidiscretizationHyperbolicGPU(...)

# Create ODE (GPU-enabled)
ode = semidiscretizeGPU(...)

# Set up callbacks
callbacks = CallbackSet(...)

# Solve ODE
sol = solve(...)
```

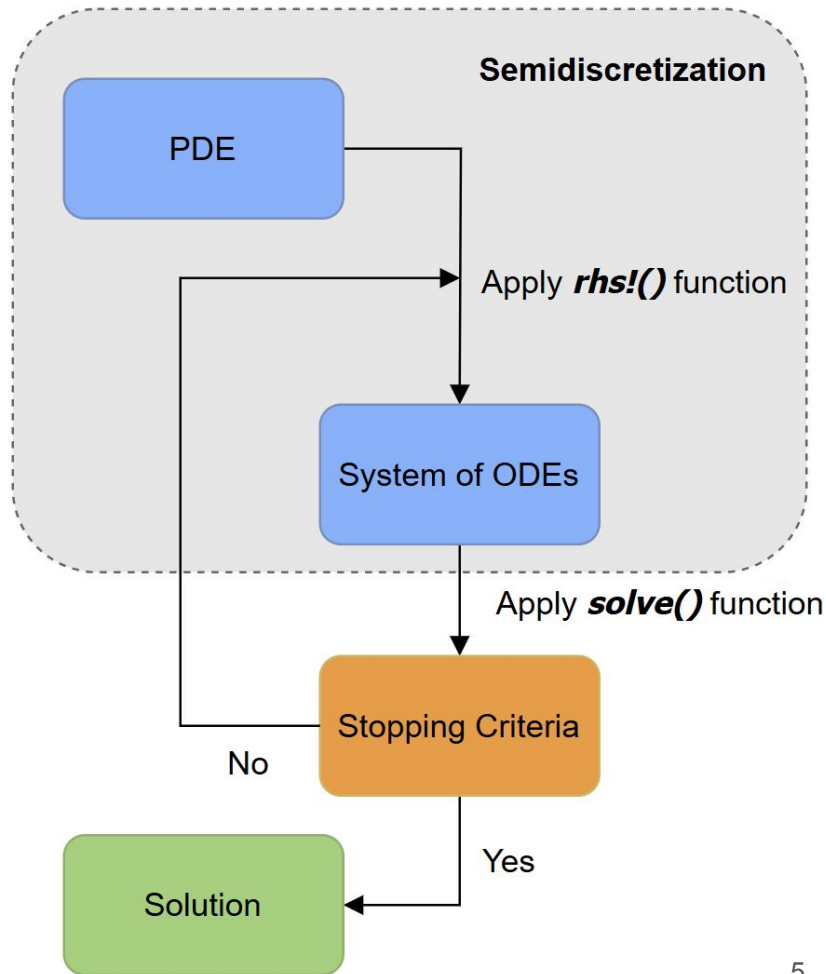
**TrixiCUDA.jl APIs**

# juliacon Workflow Skeleton

**rhs\_gpu!()**: Semidiscretization on GPU, encapsulating all GPU kernels.

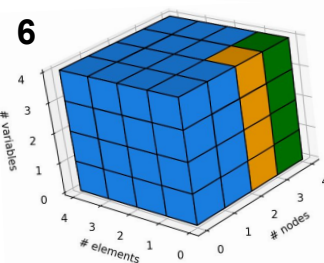
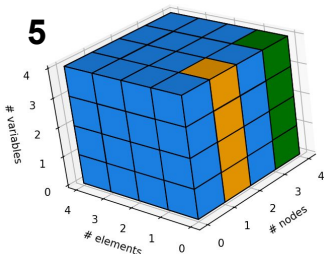
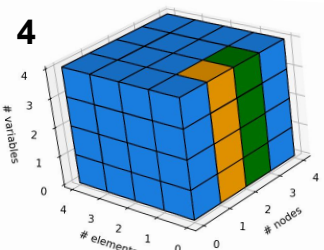
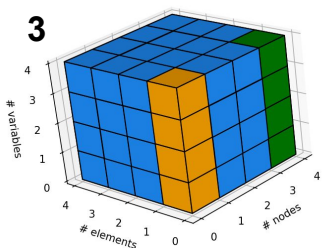
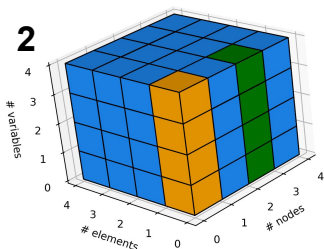
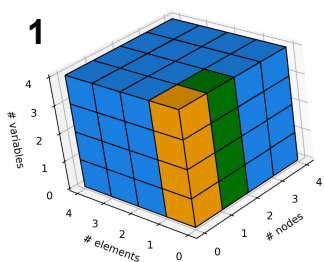
- **cuda\_volume\_integral!()** computes volume integral on GPU.
- **cuda\_interface\_flux!()** computes interface flux on GPU.
- **cuda\_boundary\_flux!()** computes boundary flux on GPU.
- Other GPU kernels...

\* See tutorial: [Introduction to DG methods](#) for volume integral, interface flux, boundary flux, etc.



# GPU Kernel Optimization

Example: Volume integral with flux differencing for 1D problems



Data: Current state  $u$  ( $4 \times 4 \times 4$  data grid)

## Original CPU code

# Pseudocode

for element in 1: #(elements):

for idx1 in 1: #(nodes):

for idx2 in (idx1+1): #(nodes):

compute volume flux between nodes  
idx1 and idx2

accumulate volume flux into node idx1

end

end

end

\* See tutorial: [DGSEM with flux differencing](#) for flux differencing.

```
# Pseudocode  
for element in 1: #(elements):
```

```
  for idx1 in 1: #(nodes):  
    for idx2 in (idx1+1): #(nodes):  
      // compute flux...
```

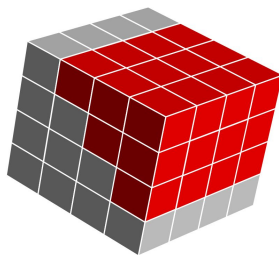
```
    end
```

```
  end
```

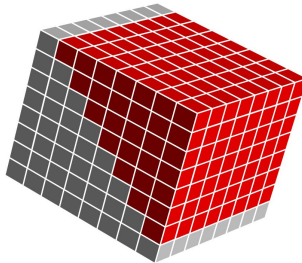
```
end
```

Naive approach: Launch thread grid  
 $\#(\text{elements}) \times \#(\text{nodes}) \times \#(\text{nodes})$ .

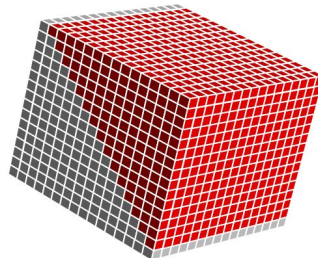
In the innermost loop, SIMD disables threads  
 $1:\text{idx1}$ , and enables other  $(\text{idx}+1):\#(\text{nodes})$   
threads.



4×4×4 threads



8×8×8 threads



16×16×16 threads

...

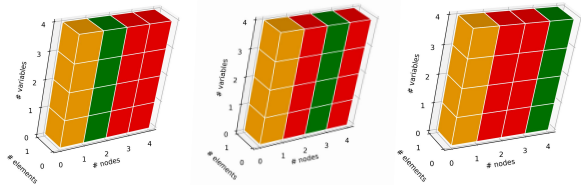
$n \times n \times n$  threads

$\# \text{grey} / (\# \text{grey} + \# \text{red}) = (n-1)/2n$  Nearly half of threads are doing nothing!

## Optimizations:

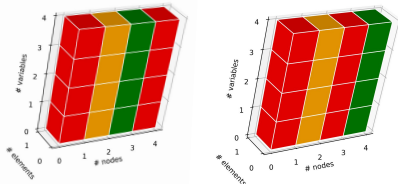
- Tiling with shared memory
- Thread coarsening

T1



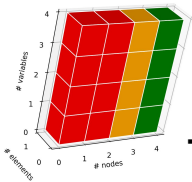
+ data loading time  $\approx 4$  time units

T2



+ data loading time  $\approx 3$  time units

T3

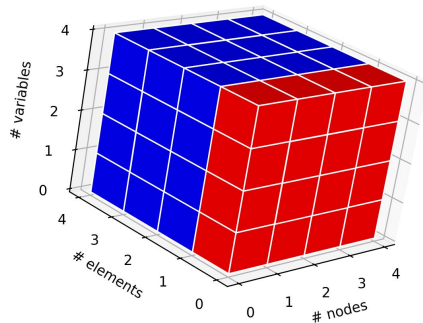


+ data loading time  $\approx 2$  time units

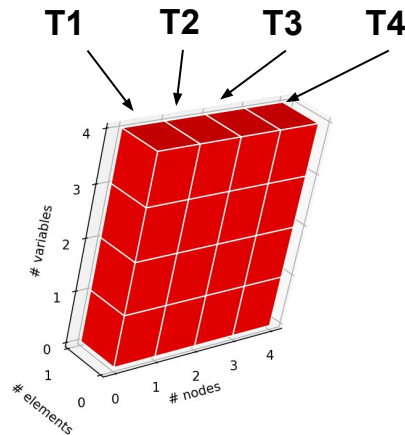
T4  $\approx 1$  time unit

Step 1: Tile the data grid by element layers and load each layer into shared memory.

Step 2: In each tile, map each dependent loop to a single thread (i.e., thread coarsening).



Red: Single shared memory tile



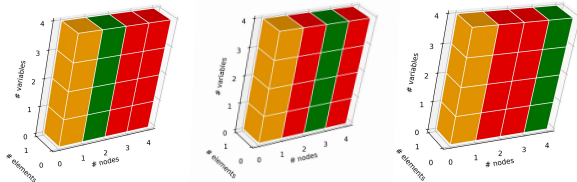
Thread mapping per block

Nearly half the time is idle!



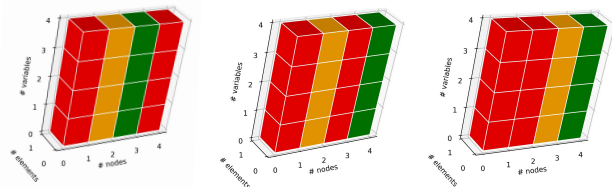
Improvement #1: Consolidate each complementary pair of threads into a single thread.

$$*T1 = T1 + T4$$



+ data loading time  $\approx 5$  time units

$$*T2 = T2 + T3$$

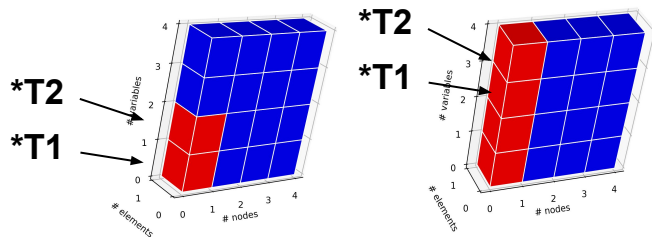


+ data loading time  $\approx 5$  time units

This frees up computing resources for other jobs!

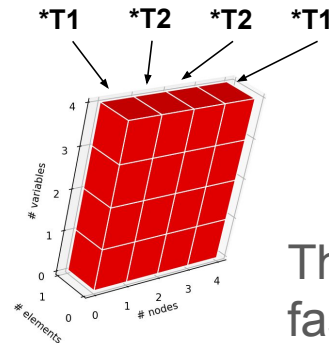
Improvement #2: Coalesce data transfers from global to shared memory.

Load data



*\* Data loading is typically coalesced across 32 threads in practice.*

Compute data



*\* Transpose all the threads when computing the data.*

This makes data transfer faster!

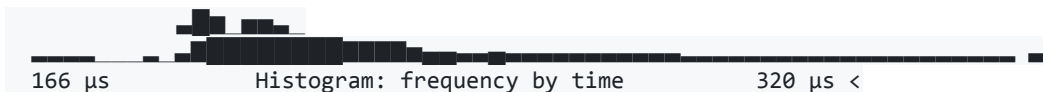
Apply optimizations similarly to 1D, 2D, and 3D problems...

## Benchmarks: 3D Euler equations with entropy conservative flux

BenchmarkTools.Trial: 10000 samples with 1 evaluation.

Range (min ... max):	166.000 $\mu$ s ... 21.309 ms	GC (min ... max):	0.00% ... 18.31%
Time (median):	207.700 $\mu$ s	GC (median):	0.00%
Time (mean $\pm$ $\sigma$ ):	223.133 $\mu$ s $\pm$ 412.949 $\mu$ s	GC (mean $\pm$ $\sigma$ ):	0.68% $\pm$ 0.36%

### Before the optimizations

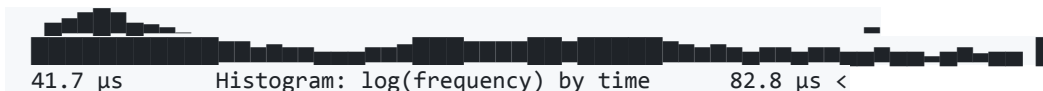


Memory estimate: 8.58 KiB, allocs estimate: 124.

BenchmarkTools.Trial: 10000 samples with 1 evaluation.

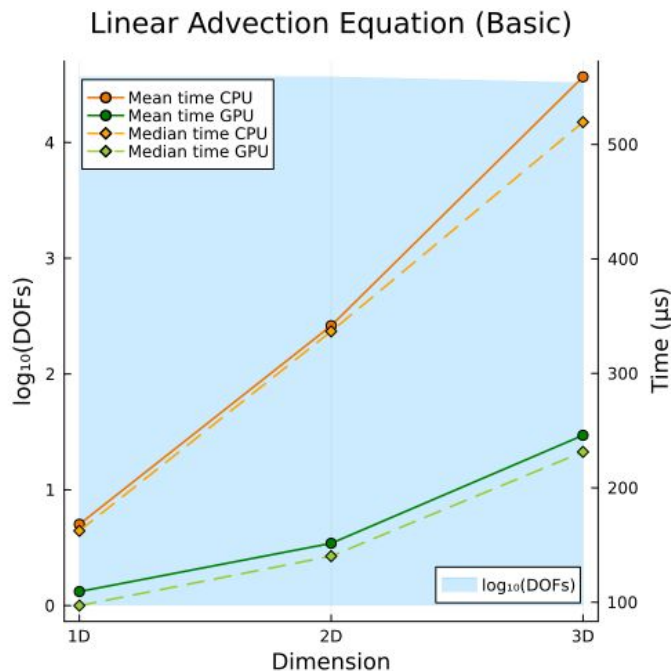
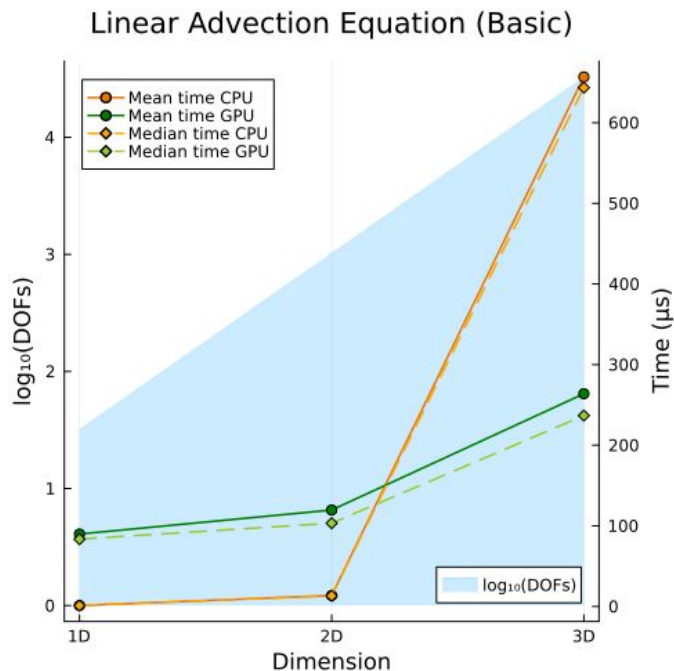
Range (min ... max):	41.700 $\mu$ s ... 260.800 $\mu$ s	GC (min ... max):	0.00% ... 0.00%
Time (median):	44.900 $\mu$ s	GC (median):	0.00%
Time (mean $\pm$ $\sigma$ ):	47.063 $\mu$ s $\pm$ 9.970 $\mu$ s	GC (mean $\pm$ $\sigma$ ):	0.00% $\pm$ 0.00%

### After the optimizations



Memory estimate: 2.52 KiB, allocs estimate: 41.

*\* Actual implementations vary by data size. See [PR #105](#) for details and more benchmarks.*



GPU advantage grows with problem complexity.

\* We are using *Trixi.jl* v0.11.17 and *TrixiCUDA.jl* v0.1.0-rc.3 for the benchmark.

Takeaways about TrixiCUDA.jl:

- Julia and CUDA: Combination of the best of both worlds.
- Package: GPU accelerator package, single precision support, intuitive and consistent APIs.
- Acceleration optimization: Semidiscretizations, Discontinuous Galerkin methods, speedups on high-dimension or high-resolution problems.

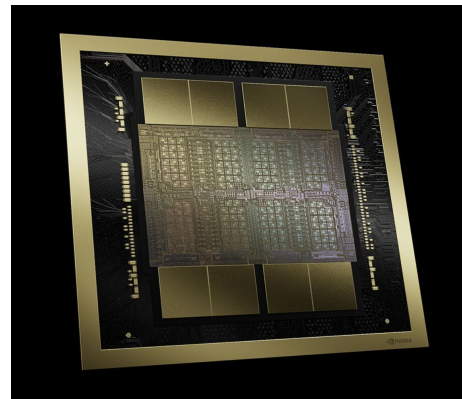


Key challenge:

General optimization across diverse problems and their inputs (related to autotuning, but not exactly), and GPU architectures.

Future directions:

1. Extend meshes, callbacks, time-stepping, and more to the GPU for full acceleration.
2. Adopt matrix-free computing (semidiscretization and more).
3. Implement mixed-precision algorithms (semidiscretization and more).



**Blackwell architecture**

Are there any questions or comments so far?





# Acknowledgment

## Project Advisors

- Prof. Hendrik Ranocha (Johannes Gutenberg University Mainz, Germany)
- Prof. Jesse Chan (Oden Institute - University of Texas at Austin, U.S.)
- Prof. Michael Schlottke-Lakemper (University of Augsburg, Germany)

## Upstream Developers

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

## Julia Community

Hwu, W. W., Kirk, D. B., El Hajj, I. (2022). *Programming Massively Parallel Processors: A Hands-on Approach*. Netherlands: Morgan Kaufmann.

Larsson, S., Thomee, V. (2008). *Partial Differential Equations with Numerical Methods*. Germany: Springer Berlin Heidelberg.

Trixi.jl Documentation, <https://trixi-framework.github.io/TrixiDocumentation/stable/>, accessed June 2025.

CUDA.jl Documentation, <https://cuda.juliagpu.org/stable/>, accessed June 2025.

SciML Documentation, <https://docs.sciml.ai/Overview/stable/>, accessed June 2025.





Thank You!

*Seeking Ph.D. Position for Fall 2026*  
Contact at [huiyuxie.sde@gmail.com](mailto:huiyuxie.sde@gmail.com)

