# 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.*

# juliacon — 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).

NVIDIA CUDA

juliacon

# 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).*

# Side-by-Side API Comparison

[Try examples](#)

Public API design principles: Similar and intuitive

```julia
# 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**

```julia
# 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(...)
```

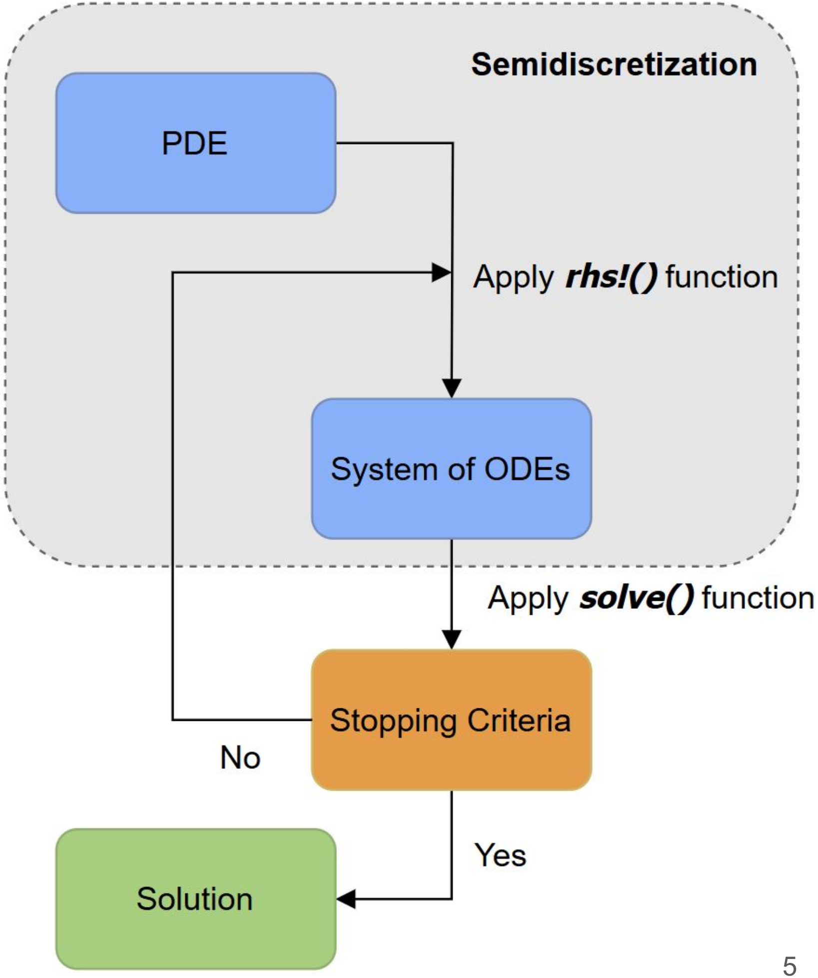
**TrixiCUDA.jl APIs**

# Workflow Skeleton

**`rhs_gpu!()`**: Semidiscretization on GPU, encapsulating all GPU kernels.

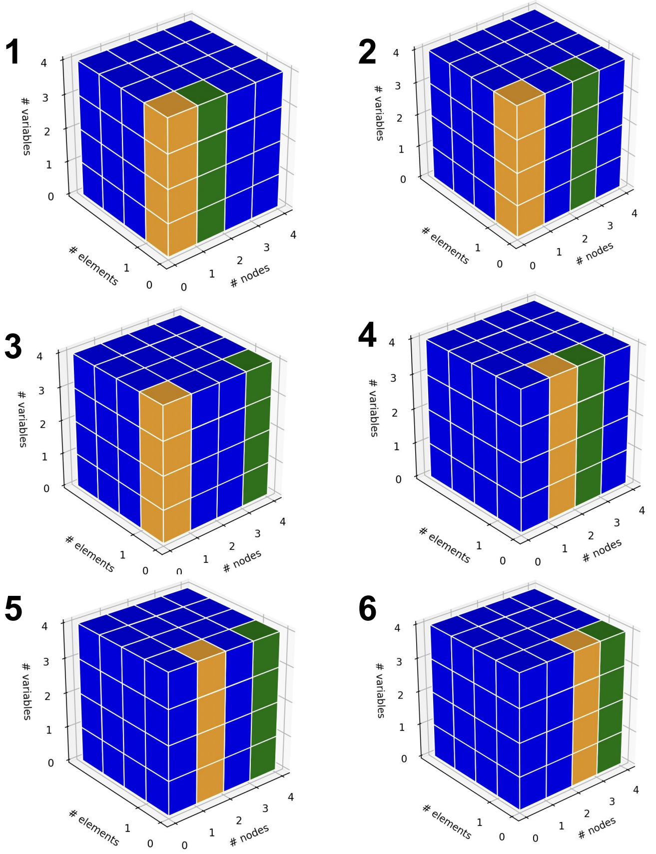
- **`cuda_volume_integral!()`** computes volume integral on GPU.
- **`cuda_interface_flux!()`** computes interface flux on GPU.
- **`cuda_boundary_flux!()`** computes boundary flux on GPU.
- Other GPU kernels…

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



# GPU Kernel Optimization

Original CPU code

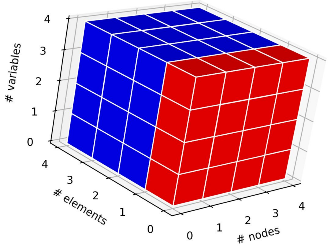



**Data: Current state u (4×4×4 data grid)**

Example: Volume integral with flux differencing for 1D problems

Dependent loops!

```
# 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
```




**Data: Single layer**

** See tutorial: DGSEM with flux differencing for flux differencing.*

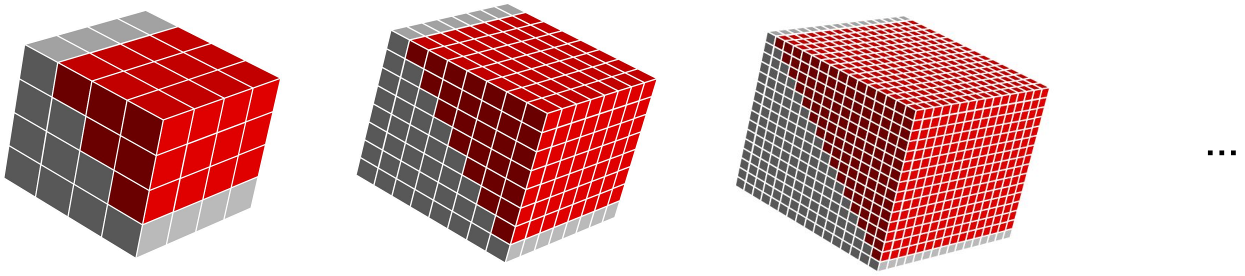
# GPU Kernel Optimization

```
# Pseudocode
for element in 1:#(elements):
    for idx1 in 1:#(nodes):
        for idx2 in (idx1+1):#(nodes):
            // compute flux…
        end
    end
end
```

Dependent loops!

Naive approach: Launch thread grid #(elements) × #(nodes) × #(nodes).

SIMD disables threads corresponding to 1:idx1 in the innermost loop, and enables the remaining threads.




**4×4×4 threads** **8×8×8 threads** **16×16×16 threads** **n×n×n threads**

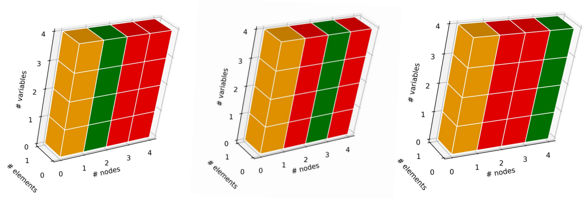
#grey/(#grey+#red) = (n-1)/2n Nearly half of threads are doing nothing!

juliacon

# GPU Kernel Optimization

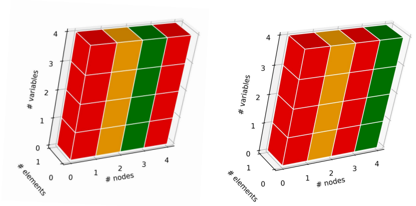
Optimizations:
- Tiling with shared memory
- Thread coarsening

T1



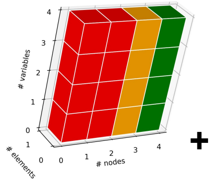

**+ data loading time ≈ 4 time units**

T2




**+ data loading time ≈ 3 time units**

T3

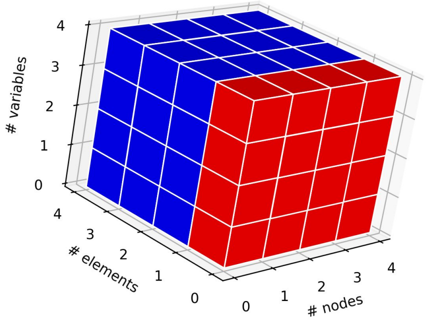



**T4 ≈ 1 time unit**

**+ data loading time ≈ 2 time units**

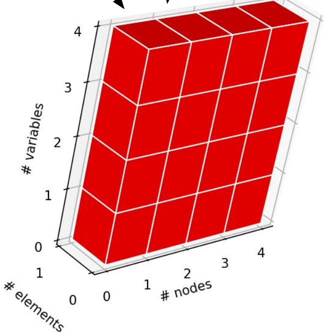
Step 1: Tile the data grid by element layers and load each layer into shared memory.

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




**Red: Single shared memory tile**

**T1 T2 T3 T4**




**Thread mapping per block**

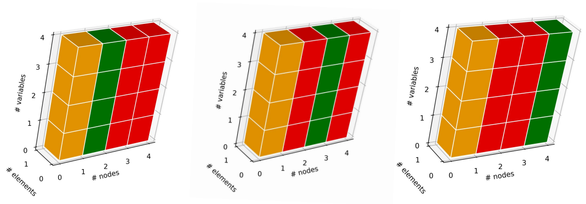
Nearly half the time is idle!

# GPU Kernel Optimization

Julia: Column-major layout!

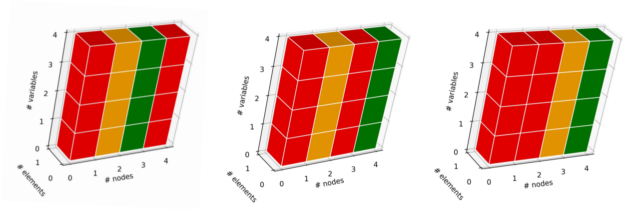
Improvement #1: Consolidate each complementary pair of threads into a single thread.

***T1 = T1 + T4**




**+ data loading time ≈ 5 time units**

***T2 = T2 + T3**

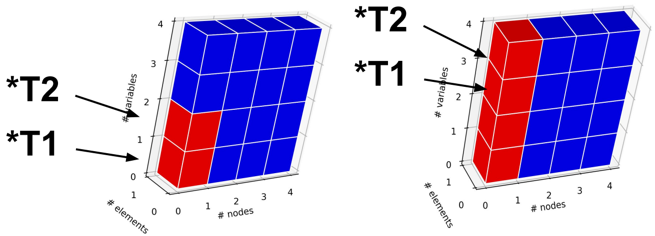



**+ data loading time ≈ 5 time units**

This frees up computing resources for other jobs!

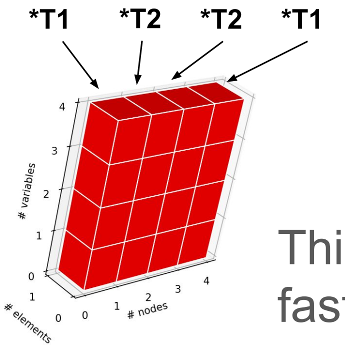
Improvement #2: Coalesce data transfers from global to shared memory.

**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!

# GPU Kernel Optimization

[Optimized GPU code]

## Apply optimizations similarly to 1D, 2D, and 3D problems…

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

**Before the optimizations**

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  166.000 μs …  21.309 ms  ┊ GC (min … max): 0.00% … 18.31%
 Time  (median):     207.700 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   223.133 μs ± 412.949 μs  ┊ GC (mean ± σ):  0.68% ±  0.36%
```

```
  166 μs           Histogram: frequency by time          320 μs <

Memory estimate: 8.58 KiB, allocs estimate: 124.
```

**After the optimizations**

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  41.700 μs … 260.800 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     44.900 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   47.063 μs ±   9.970 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

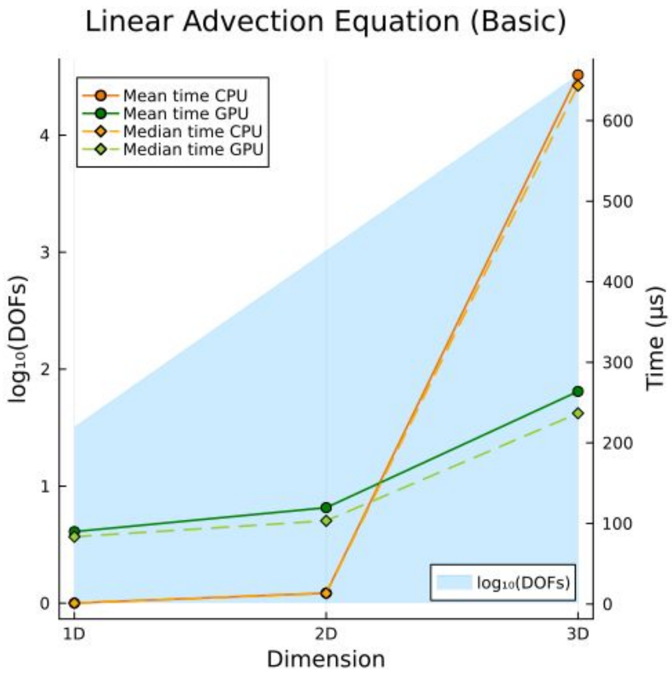
```
  41.7 μs       Histogram: log(frequency) by time      82.8 μs <

Memory estimate: 2.52 KiB, allocs estimate: 41.
```

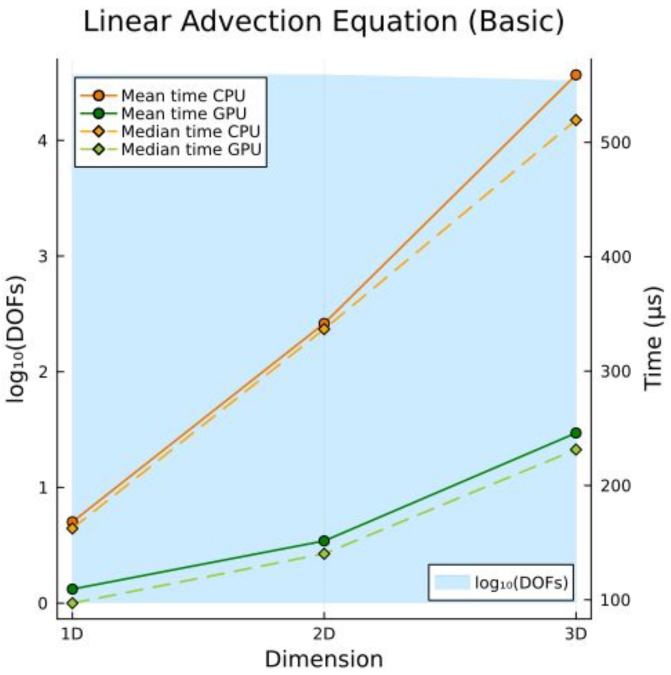
** Actual implementations vary by data size. See [PR #105] for details and more benchmarks.*

# Benchmark on Float32

[Full benchmark results]

[Reproduce the results]




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.*

# Key Takeaways

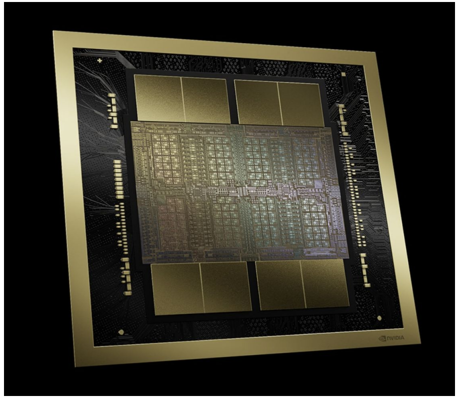
Takeaways about TrixiCUDA.jl:

- 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.

# Challenge and Future Directions

Key challenge:

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



**Blackwell architecture**

Future directions:

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

# Q&A Session

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

# References

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

Hesthaven, J. S., Warburton, T. (2008). *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications.* United Kingdom: Springer.

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

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*