WaterLily.jl

An **incompressible flow solver** for dynamic bodies
running seamlessly on **CPU/GPU**

Bernat Font, Gabe Weymouth

b.font@tudelft.nl
g.d.weymouth@tudelft.nl

Julia4PDEs 2026

TU Delft

Behind WaterLily

Paella Appreciator

Seafood Appreciator

Paella Aficionado

The requirements: leveraging the Julia ecosystem

1. **Fast**:
   - Compiled: Julia uses **LLVM**
   - Multi-threading on CPU and GPU of any vendor: `KernelAbstractions.jl`
   - Memory-distributed computing: `MPI.jl`

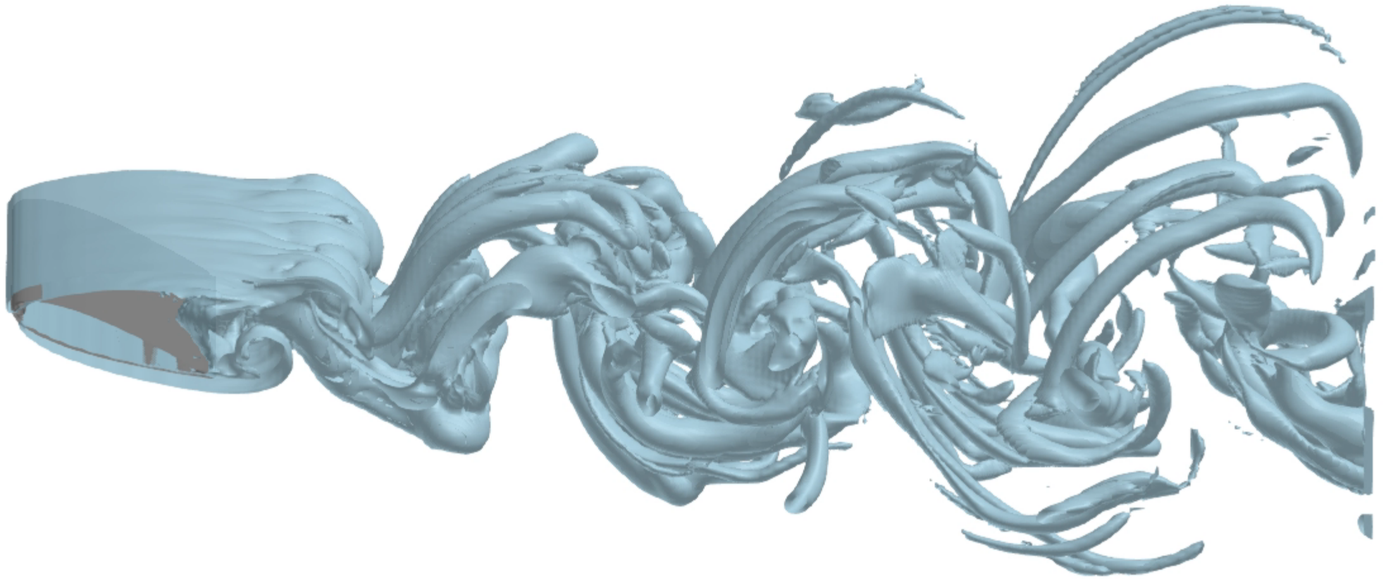
2. **Flexible**
   1. Support of STL-like geometries via `MeshIO.jl` and `GeometryBasics.jl`
   2. `ForwardDiff.jl` to compute body velocity from position and time-mapping function

3. **Natively linked to ML**
   - Can be coupled with `Lux.jl`

The requirements: leveraging the Julia ecosystem

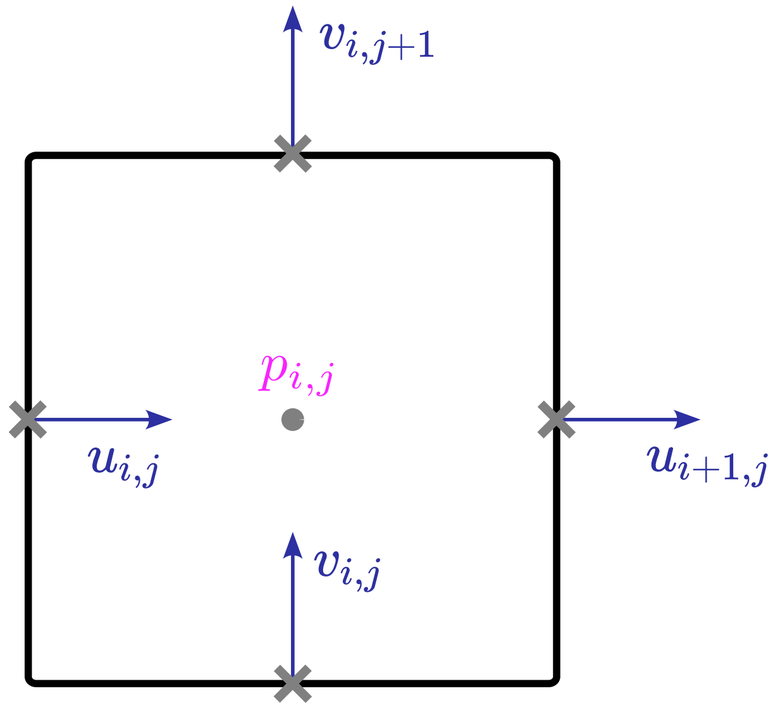
4. **Differentiable**
   - Full solver differentiability via `ForwardDiff.jl`

5. **Visualization**
   - Fast rendering of 2D or 3D flow fields using `Makie.jl`



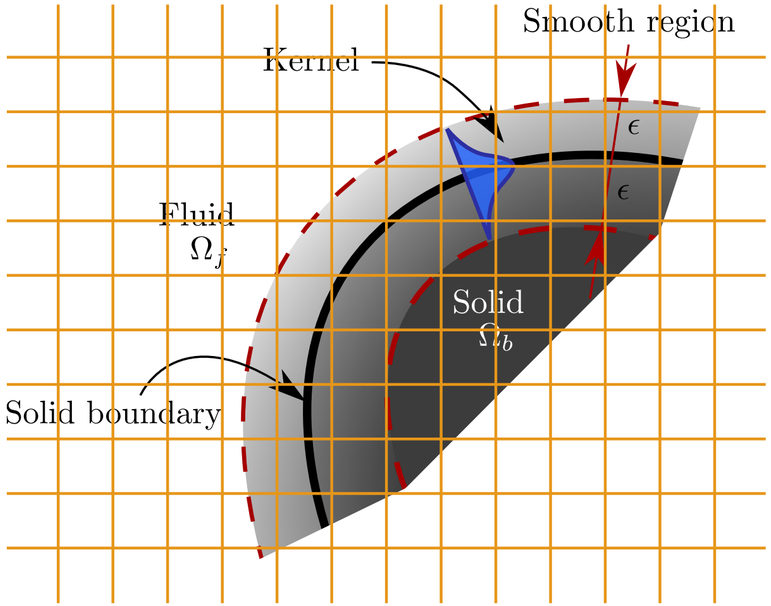
WaterLily | Numerical methods

- Incompressible flow
- Finite volume in a staggered grid
  - QUICK reconstruction scheme in space, predictor-corrector in time (2nd order)



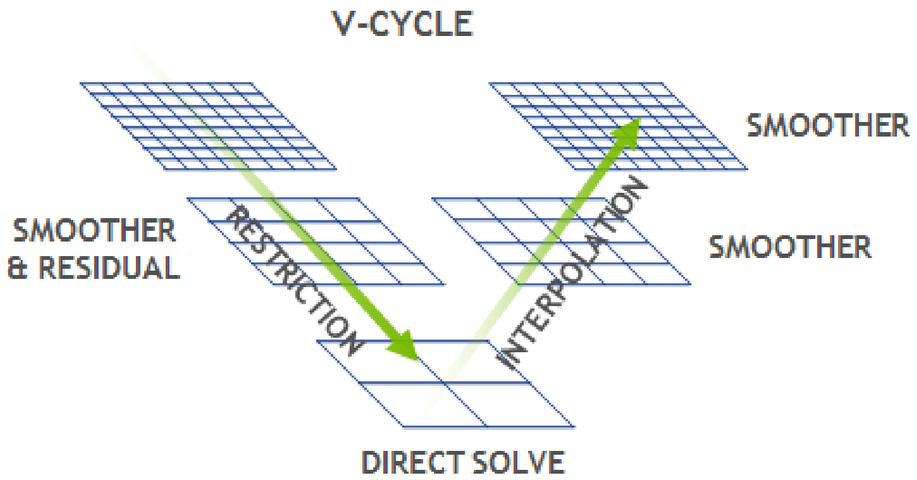
WaterLily | Numerical methods

- Incompressible flow
- Finite volume in a staggered grid
  - QUICK reconstruction scheme in space, predictor-corrector in time (2nd order)
- Boundary data immersion method for solid geometries
  - Signed distance function, **ForwardDiff.jl** (normal, curvature, velocity)



`WaterLily` | Numerical methods

- Incompressible flow
- Finite volume in a staggered grid
  - QUICK reconstruction scheme in space, predictor-corrector in time (2nd order)
- Boundary data immersion method for solid geometries
  - Signed distance function, `ForwardDiff.jl` (normal, curvature, velocity)
- Geometric multigrid (GMG) for the pressure solver + projection method



Porting to GPU using KernelAbstractions.jl

- Consider the divergence of a 2D vector field $\vec{u}$ in a staggered and Cartesian grid where $\Delta x = 1$

$$\sigma = \iiint (\nabla \cdot \vec{u}) \, \mathrm{d}V = \iint \vec{u} \cdot \hat{n} \, \mathrm{d}S \rightarrow \sigma_{i,j} = (u_{i+1,j} - u_{i,j}) + (v_{i,j+1} - v_{i,j})$$

- This can be written as

```
N = (10, 10) # domain size
σ = zeros(N) # scalar field
u = rand(N..., length(N)) # 2D vector field with ghost cells

for d ∈ 1:ndims(σ), I ∈ inside(σ)
    σ[I] += ∂(d, I, u)
end
```

Porting to GPU using KernelAbstractions.jl

- **KernelAbstractions.jl** allows to write a simple kernel function that is then **specialized** for different GPU backend:
  - **CUDA.jl**, **AMDGPU.jl**, **oneAPI.jl**, and **Metal.jl**

```
using KernelAbstractions: get_backend, @index, @kernel

@kernel function _divergence!(σ, u)
    I = @index(Global, Cartesian) # define a parallel Cartesian index in Global memory
    σ_sum = zero(eltype(σ)) # initialize accumulation
    for d ∈ 1:ndims(σ)
        σ_sum += ∂(d, I, u)
    end
    σ[I] = σ_sum # assign result
end
function divergence!(σ, u)
    _divergence!(get_backend(σ), 64)(σ, u, ndrange=size(inside(σ))) # generate & call the kernel
end
```

Porting to GPU using KernelAbstractions.jl

```
for d ∈ 1:ndims(σ)
    @loop σ[I] += ∂(d, I, u) over I ∈ inside(σ)
end
```

- Generating a kernel for every **loop** in our solver

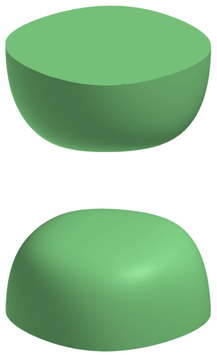
```
macro loop(args...)
    ex,I,R,sym = args, [] # get expression, iterator, range
    grab!(sym, ex)        # get kernel arguments (symbols)
    @gensym kern          # generate unique kernel function name
    return quote
        @kernel function $kern($(rep.(sym)...)) # define the kernel
            $I = @index(Global, Cartesian)
            $ex
        end
        $kern(get_backend($(sym[1])), 64)($(sym...), ndrange=size($R)) # generate & call the kernel
    end |> esc
end
```

- Automatically **specialized for any computing backend**

WaterLily | Easy simulation setup

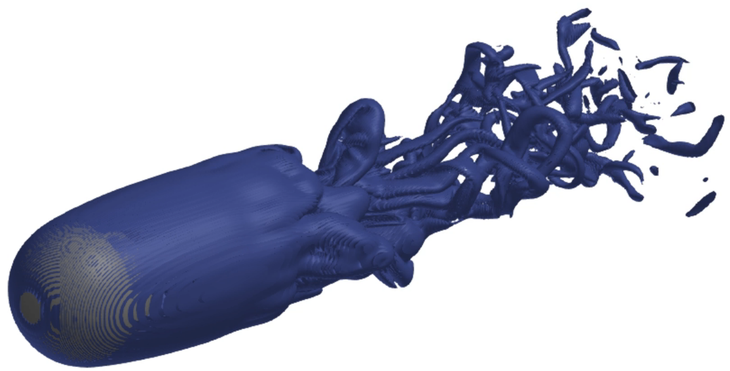
- The **Simulation** type holds information of **Fluid** and **Body** types
- User defines the IC, BC, simulation size, and the body SDF (if any)
- **Simulation** can be run on CPU/GPU changing the array type: **Array**, **CuArray**, **ROCArray**

```julia
using WaterLily, CUDA

function TGV(; p=6, Re=1600, T=Float32, mem=Array)
    # Define vortex size, velocity, viscosity
    L = 2^p; U = T(1); κ=T(π/L); ν = T(1/(κ*Re))
    function uλ(i,xyz) # initial velocity field
        x,y,z = @. xyz*π/L
        i==1 && return -U*sin(x)*cos(y)*cos(z) # u_x
        i==2 && return  U*cos(x)*sin(y)*cos(z) # u_y
        return 0*U                             # u_z
    end
    return Simulation((L, L, L), (0, 0, 0), L; U, uλ, ν, T, mem)
end
sim = TGV(mem=CuArray)
sim_step!(sim, 20)
```



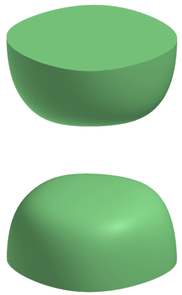
WaterLily | Easy simulation setup

```julia
using WaterLily, CUDA, StaticArrays

function sphere(D; Re=3700, U=1, L=(7,3,3), center=(1.5,1.5,1.5), mem=CuArray, T=Float32)
    center = SA{T}[center...]
    body = AutoBody((x,t)-> √sum(abs2, x .- (center .* D)) - D÷2)
    Simulation(L.*D, (U, 0, 0), D; U, ν=U*D/Re, body, mem, T, exitBC=true)
end

sim = sphere(128)
sim_step!(sim, 10)
```

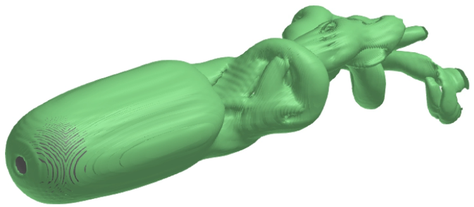


Benchmarking | Not only cool, but fast too!

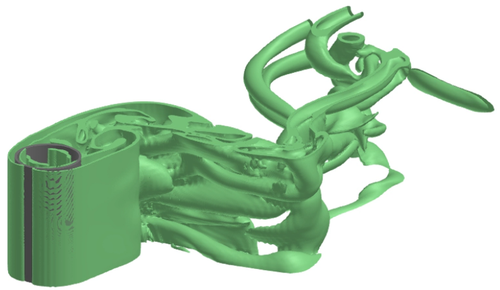
TGV



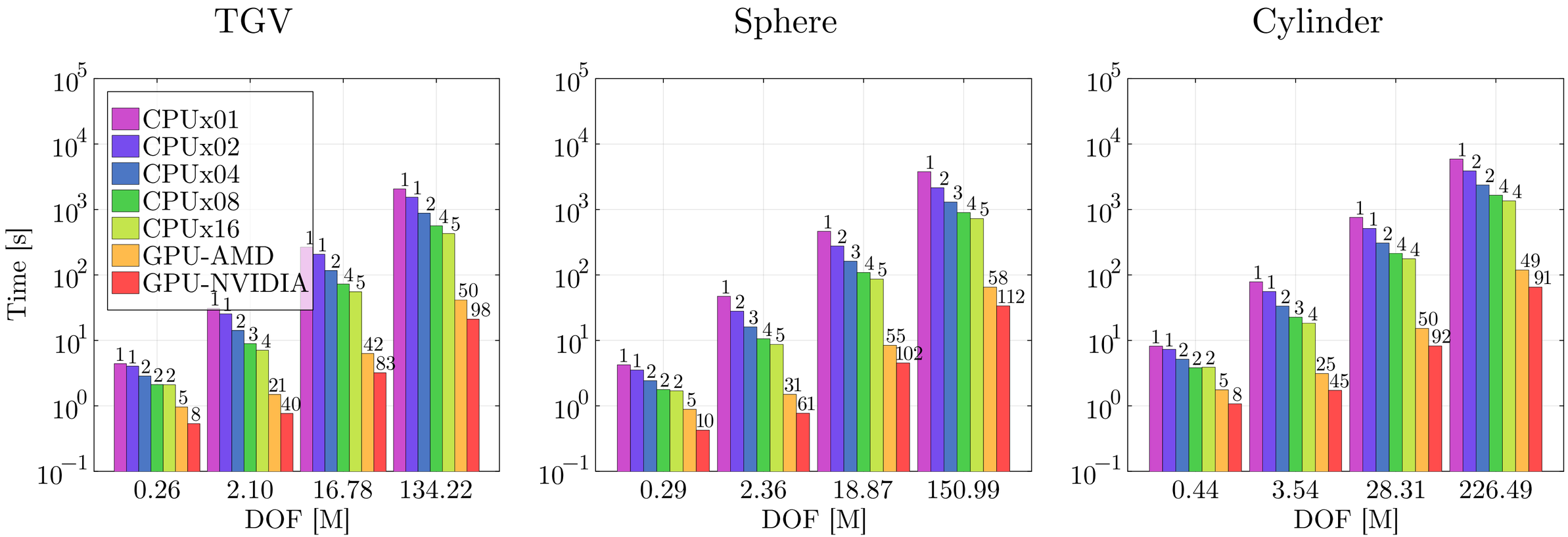
Sphere



Cylinder

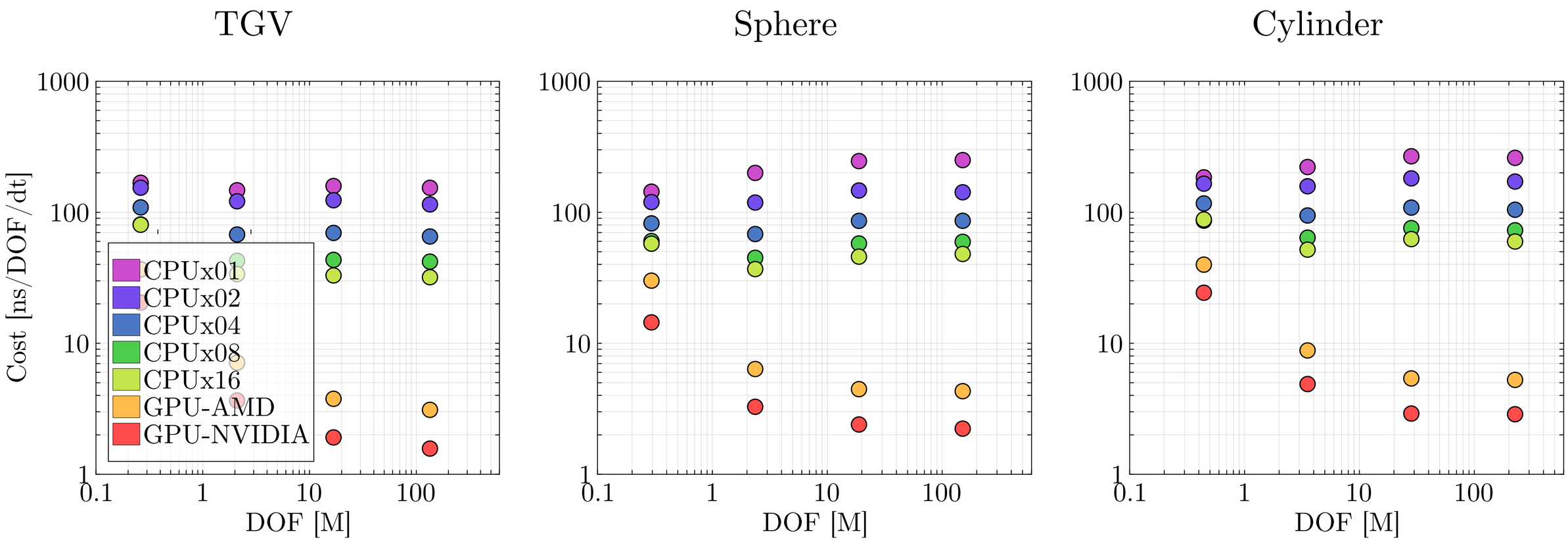


Benchmarking | x100 speedup on modern GPUs vs serial CPU



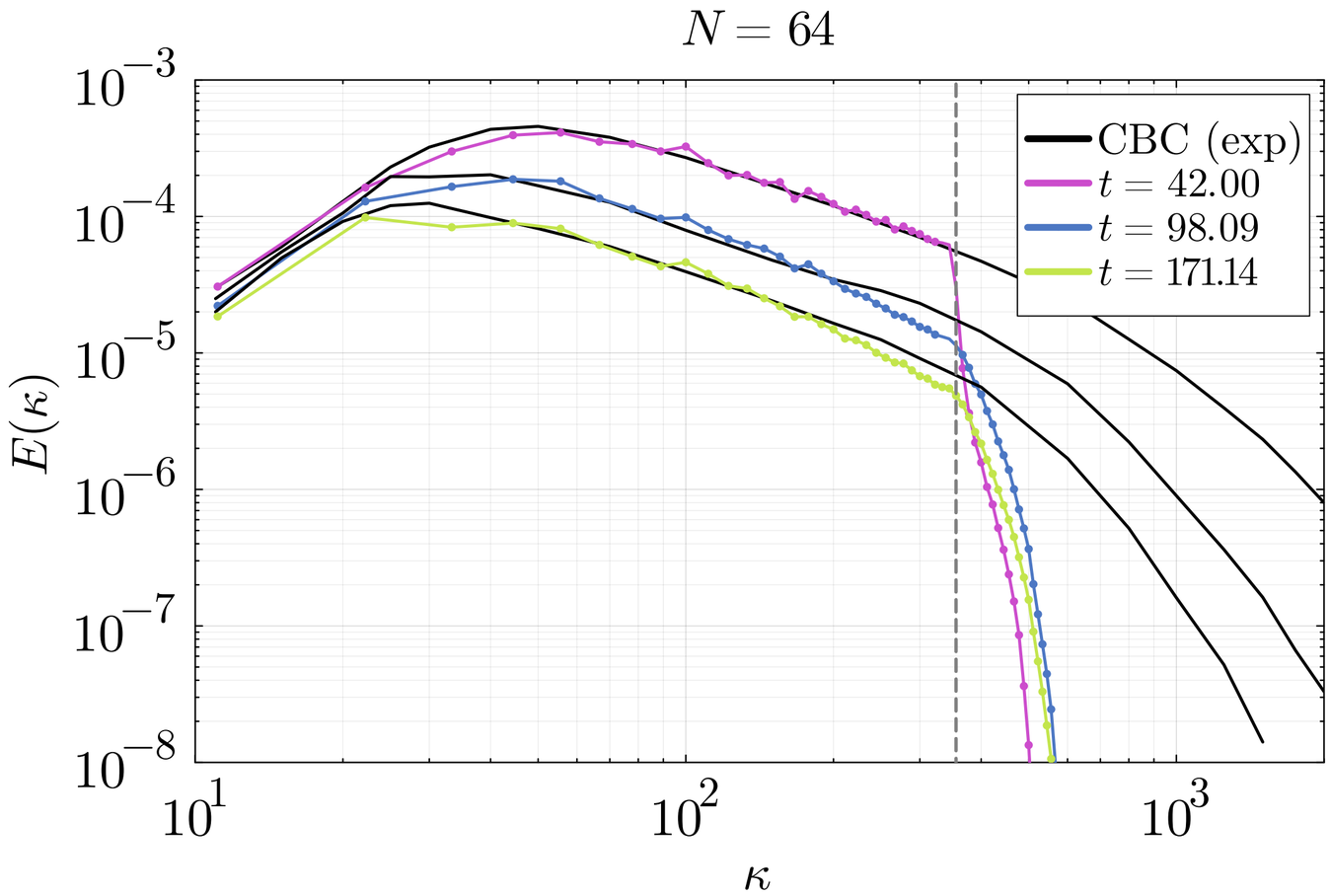
- FP32, 100 dts, NVIDIA H100 GPU, AMD MI250X GPU
- 302M DOF consume 39GB VRAM

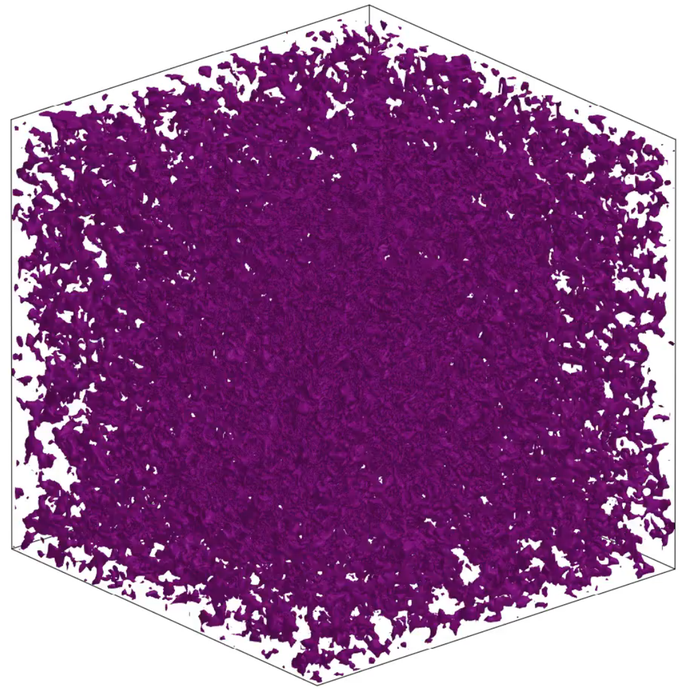
Benchmarking | Top speed: 1.5 ns/DOF/dt



Validation | Homogeneous isotropic turbulence

- $Re = 34\text{k}$, $N = 64^3$ (periodic), CDS + Smagorinky model $C_s = 0.17$





Validation | Sphere

- $Re = 3700$, $D = \{88, 128, 128\}$, $L = (7, \{6, 3\}, \{6, 3\})D$
  DOFs $= 300$M, QUICK (iLES)