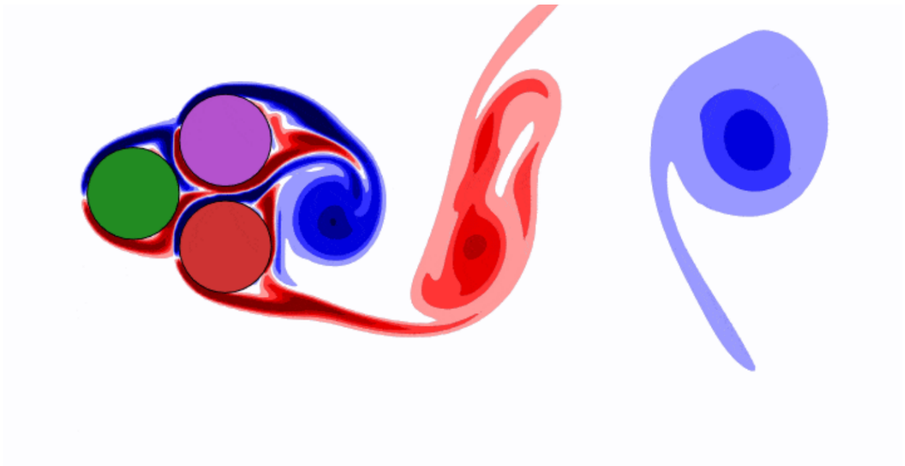
`WaterLily.jl`: A differentiable solver to simulate fluid flow and dynamic bodies with heterogeneous execution

Bernat Font*, Gabriel D. Weymouth

JuliaCon 2024



TU Delft

b.font@tudelft.nl

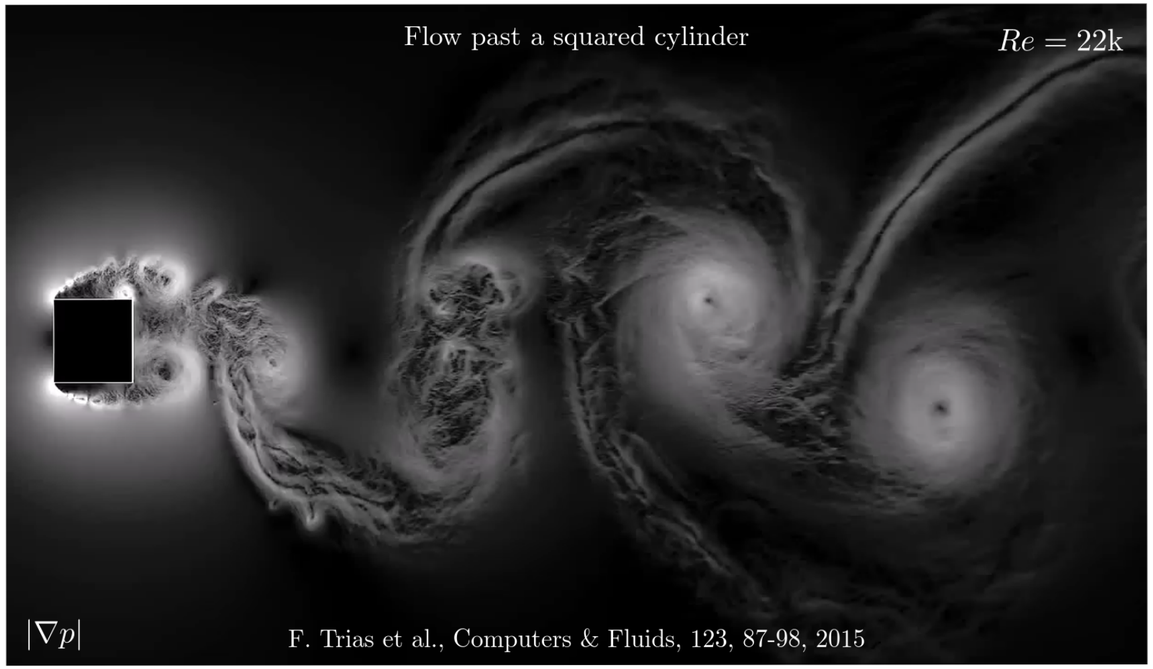
Background | Simulating fluid flows

- **Solving the Navier-Stokes equations, aka. Computational Fluid Dynamics (CFD)**

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\nabla p + Re^{-1} \nabla^2 \boldsymbol{u}$$

  - Designing vehicles (aircrafts, trains, cars, boats...), weather forecasting, combustion, even biological flows

- **But it's costly!** There is a wide range of spatial and temporal scales that need to be resolved → **HPC**



Julia for CFD?

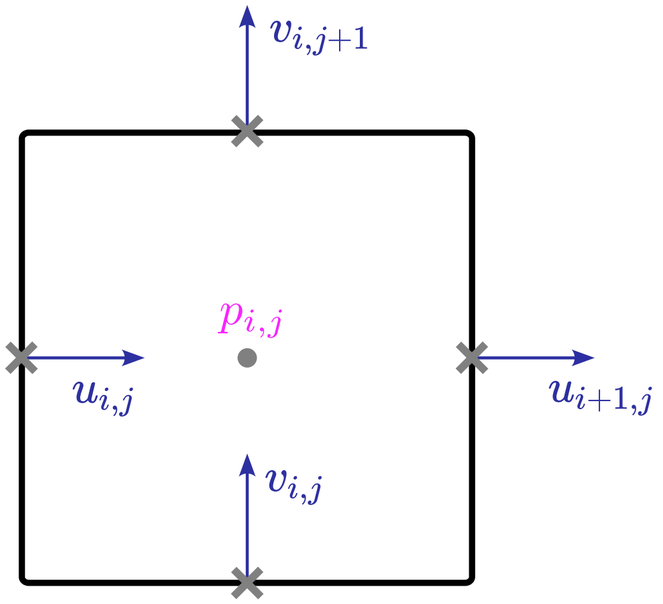
- **CFD solvers must be written in compiled languages.**
- **In CFD, HPC is a must.**
  - Julia is parallel: `LoopVectorization.jl, Polyester.jl, MPI.jl, ...`
- **Modern CFD solvers should be able to run in both CPUs and GPUs.**
  - Julia can run in most architectures: `CUDA.jl, AMDGPU.jl, KernelAbstractions.jl, ...`

Added benefits:

- **Solves the two-language problem.**
  - Easy to integrate with ML libraraies such as `SciML, Flux.jl, ...`
- **Active community.**
  - Engage with core developers and benefit from latest developments

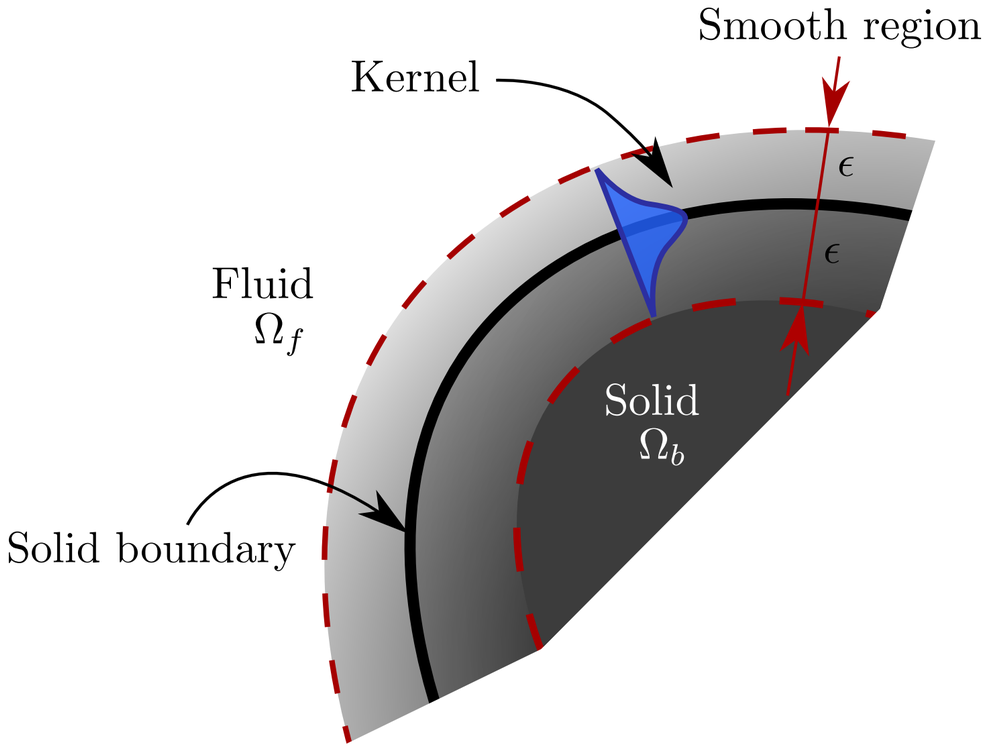
Numerical methods in WaterLily.jl

- Incompressible fluid flow
- Finite volume in a staggered grid



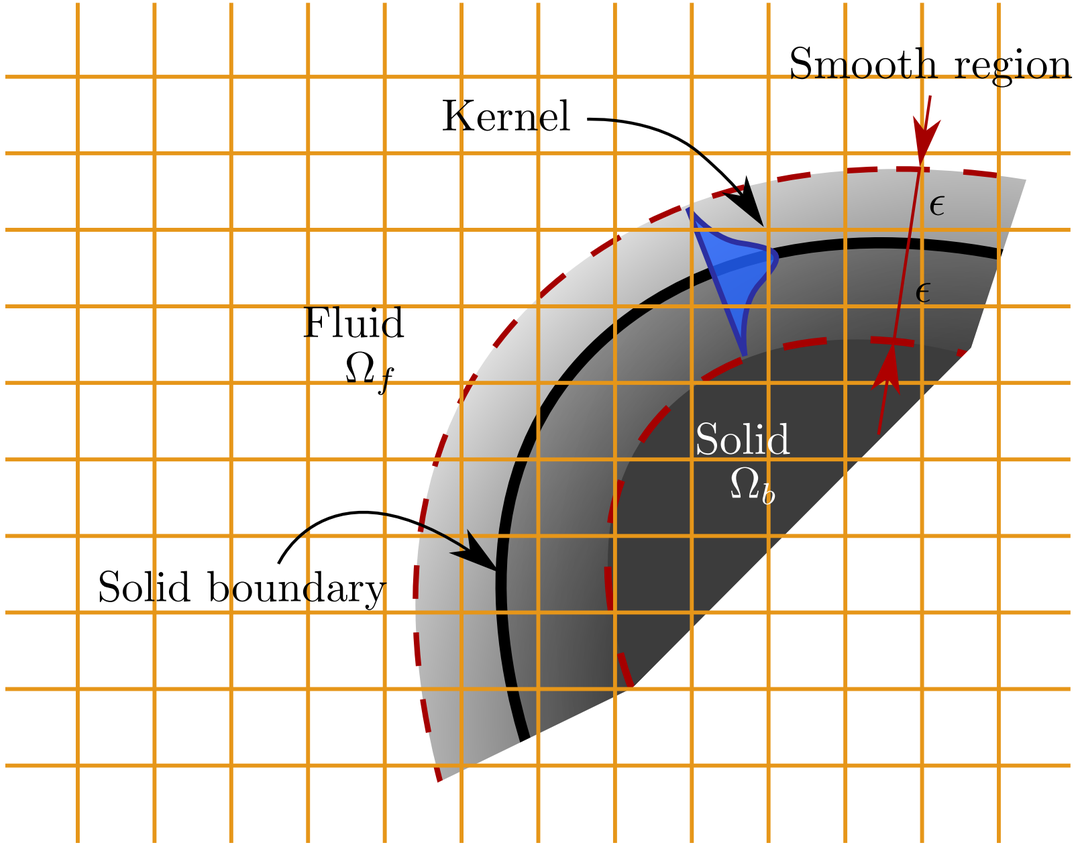
Numerical methods in WaterLily.jl

- Incompressible fluid flow
- Finite volume in a staggered grid
- Boundary data immersion method for solid geometries
  - Signed distance function (SDF) + `ForwardDiff.jl` to compute body normal, curvature, and velocity



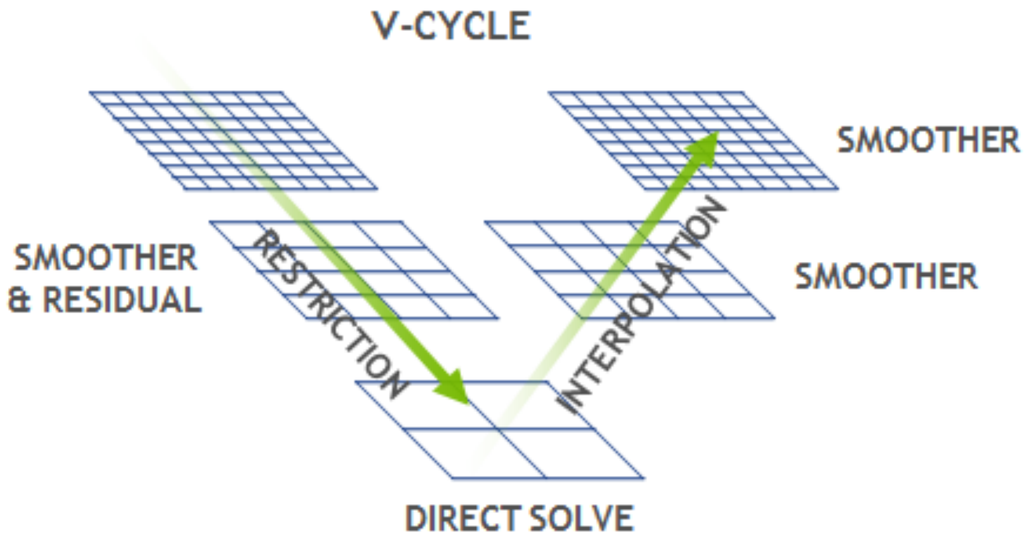
Numerical methods in WaterLily.jl

- Incompressible fluid flow
- Finite volume in a staggered grid
- Boundary data immersion method for solid geometries
  - Signed distance function (SDF) + `ForwardDiff.jl` to compute body normal, curvature, and velocity



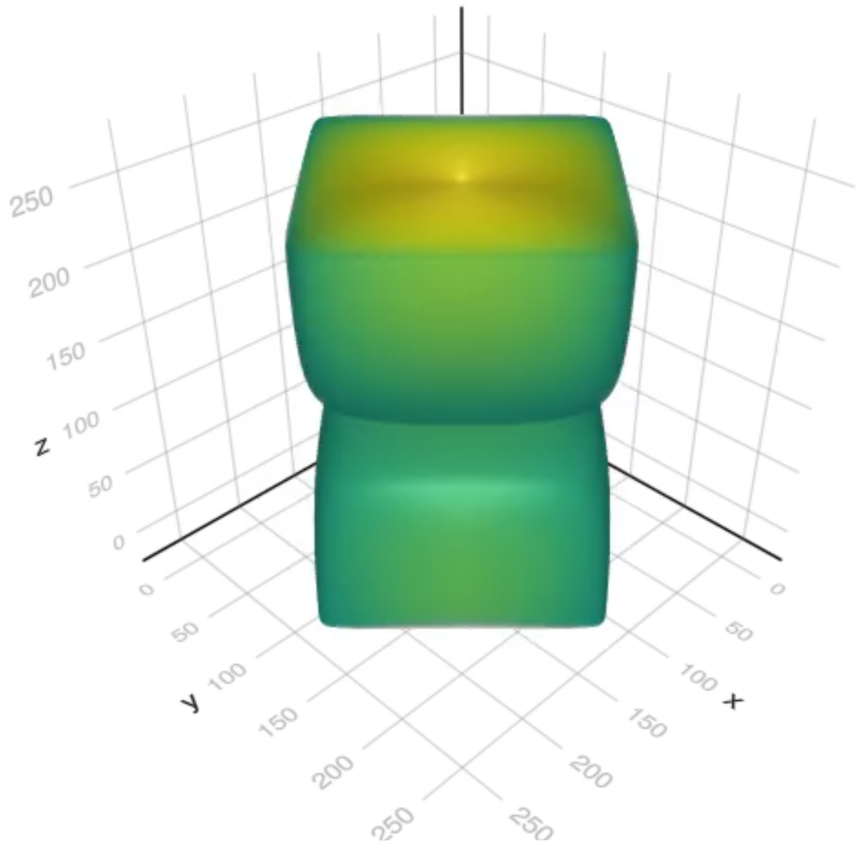
Numerical methods in WaterLily.jl

- Incompressible fluid flow
- Finite volume in a staggered grid
- Boundary data immersion method for solid geometries
  - Signed distance function (SDF) + `ForwardDiff.jl` to compute body normal, curvature, and velocity
- Geometric multigrid for the pressure solver (Poisson equation) + projection method

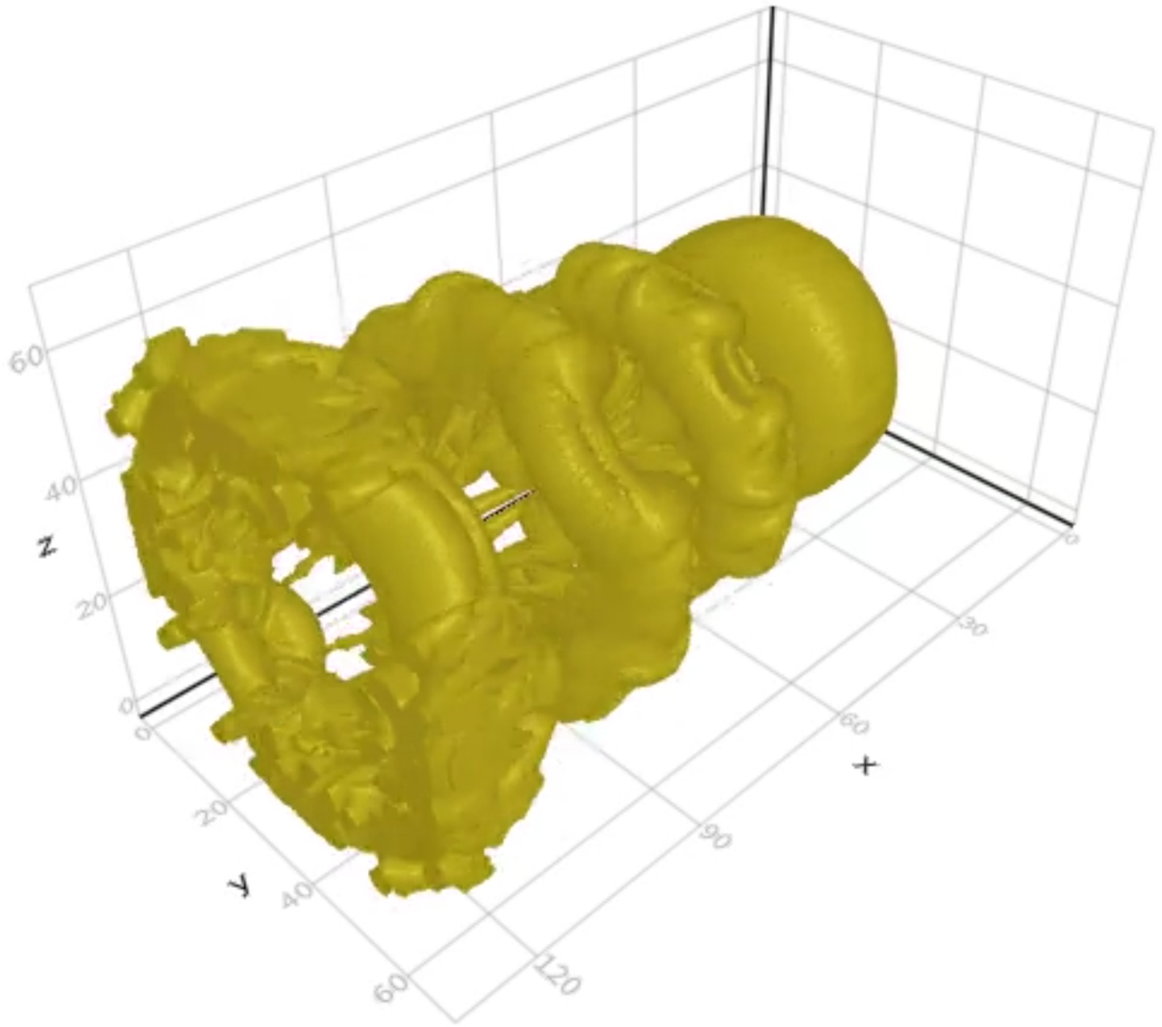


Easy simulations setup

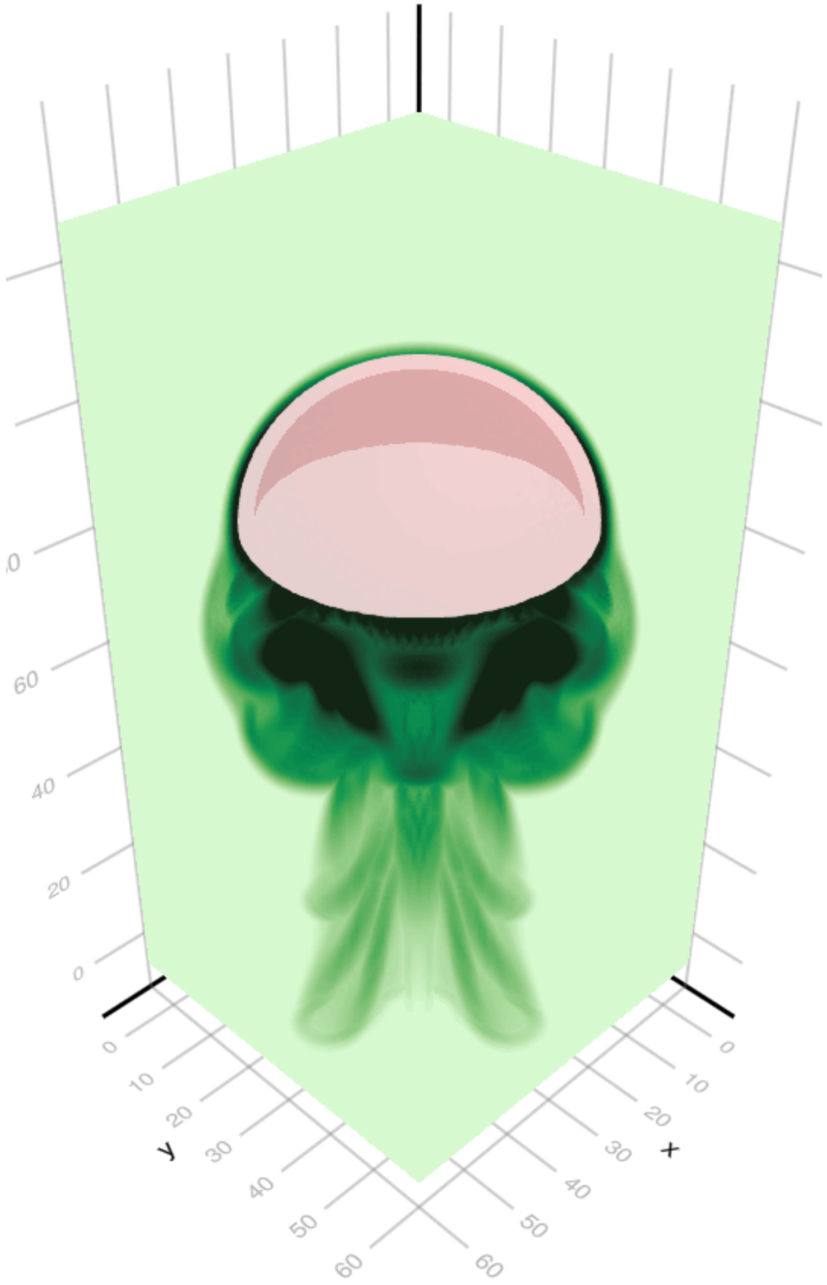
- The `Simulation` struct holds information of both `Fluid` and `Body`
- The user defines the initial condition of the problem, the size of the simulation, and the body SDF (if any).
- The simulation can be run on CPU or GPU just specifying the array type (`Array, CuArray, ROCArray, ...`)




```julia
using WaterLily
function TGV(; pow=6, Re=1e5, T=Float32, mem=Array)
    # Define vortex size, velocity, viscosity
    L = 2^pow; U = 1; ν = U*L/Re
    # Taylor-Green-Vortex initial velocity field
    function uλ(i,xyz)
        x,y,z = @. (xyz-1.5)*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_z
    end
    # Initialize simulation
    return Simulation((L, L, L), (0, 0, 0), L; U, uλ, ν, T, mem)
end
```




```julia
using WaterLily
using StaticArrays
function donut(p=6;Re=1e3,mem=Array,U=1)
    # Define simulation size, geometry dimensions, viscosity
    n = 2^p
    center,R,r = SA[n/2,n/2,n/2], n/4, n/16
    v = U*R/Re

    # Apply signed distance function for a torus
    norm2(x) = √sum(abs2,x)
    body = AutoBody() do xyz,t
        x,y,z = xyz - center
        norm2(SA[x,norm2(SA[y,z])-R])-r
    end

    # Initialize simulation and return center for flow viz
    Simulation((2n,n,n),(U,0,0),R;v,body,mem),center
end
```

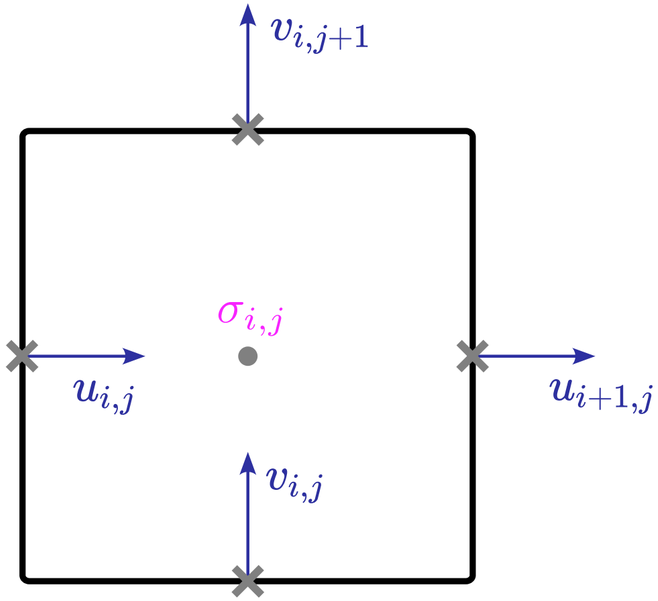



```julia
using WaterLily
using StaticArrays
function jelly(p=5;Re=5e2,mem=Array,U=1)
    # Define simulation size, geometry dimensions, & viscosity
    n = 2^p; R = 2n/3; h = 4n-2R; ν = U*R/Re

    # Motion functions
    ω = 2U/R
    @fastmath @inline A(t) = 1 .- SA[1,1,0]*0.1*cos(ω*t)
    @fastmath @inline B(t) = SA[0,0,1]*((cos(ω*t)-1)*R/4-h)
    @fastmath @inline C(t) = SA[0,0,1]*sin(ω*t)*R/4

    # Build jelly from a mapped sphere and plane
    sphere = AutoBody((x,t)->abs(√sum(abs2,x)-R)-1, # sdf
                      (x,t)->A(t).*x+B(t)+C(t))     # map
    plane = AutoBody((x,t)->x[3]-h,(x,t)->x+C(t))
    body =  sphere-plane

    # Return initialized simulation
    Simulation((n,n,4n),(0,0,-U),R;ν,body,mem,T=Float32)
end
```

Behind the scenes | Loop abstractions

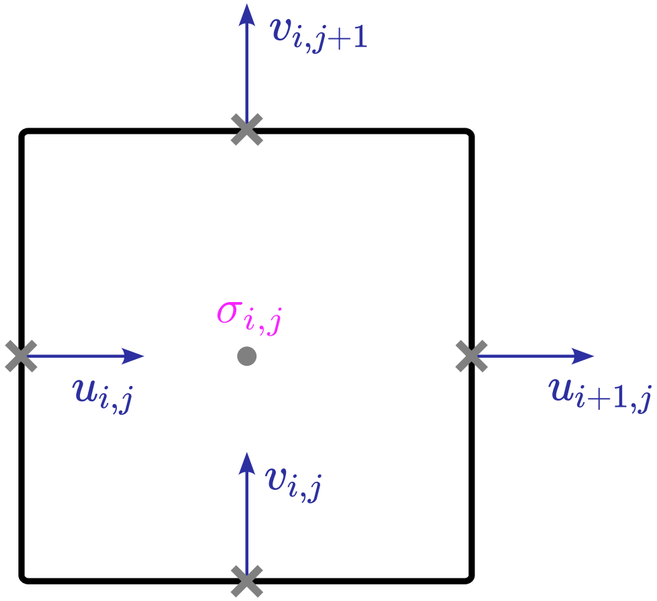
$$\sigma_{i,j} = \iiint \nabla \cdot \boldsymbol{u} \, \mathrm{d}V = \iint \boldsymbol{u} \cdot \hat{\boldsymbol{n}} \, \mathrm{d}S = (u_{i+1,j} - u_{i,j}) + (v_{i,j+1} - v_{i,j})$$

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

# serial loop macro
macro loop(args...)
    ex,_,itr = args # gets expression and iterator info
    op,I,R = itr.args # from iterator info, get index and range
    @assert op ∈ (:(∈),:(in))
    return quote
        for $I ∈ $R
            $ex # contains I
        end
    end |> esc
end
```



Behind the scenes | Loop abstractions + KernelAbstractions.jl

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



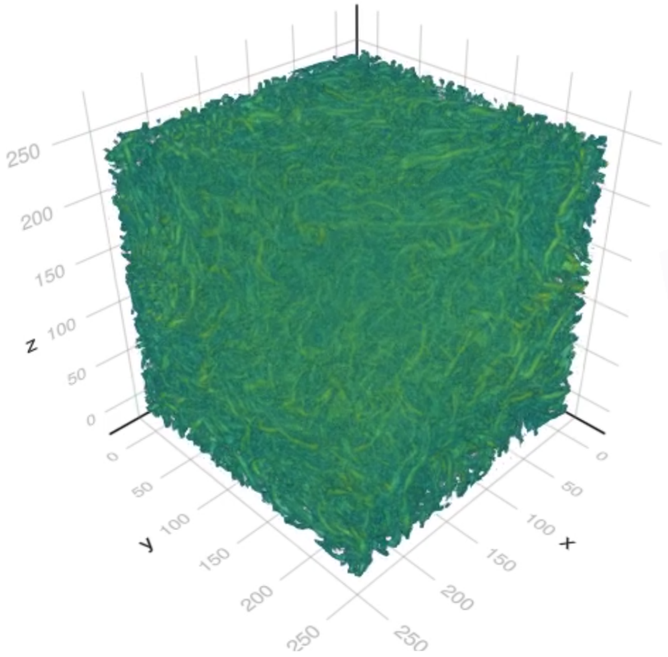

```julia
function divergence!(σ, u)
    for d ∈ 1:ndims(σ)
        @loop σ[I] += ∂(d, I, u) over I ∈ inside(σ)
    end
end
```

Codebase from 1000 LOC to... 1000LOC!

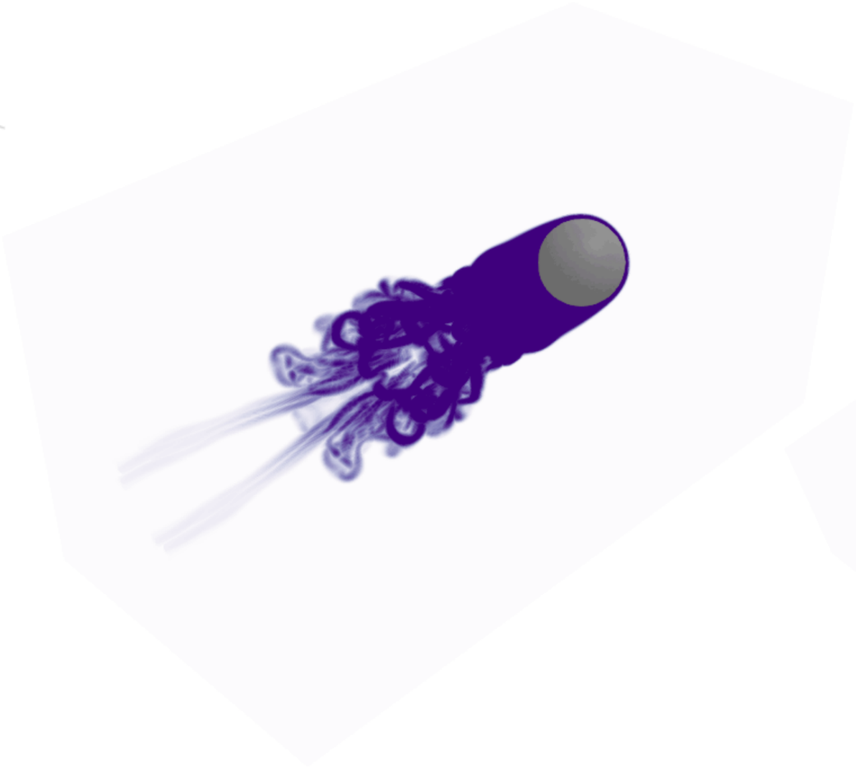
```julia
# KA-adapted loop macro
macro loop(args...)
    ex,_,itr = args
    _,I,R = itr.args; sym = []
    grab!(sym,ex)     # get arguments and replace composites in `ex`
    setdiff!(sym,[I]) # don't want to pass I as an argument
    @gensym kern      # generate unique kernel function name
    return quote
        @kernel function $kern($(rep.(sym)...),@Const(I0)) # replace composite arguments
            $I = @index(Global,Cartesian)
            $I += I0 # offset
            $ex # contains I
        end
        $kern(get_backend($(sym[1])),64)($(sym...),$R[1]-oneunit($R[1]),ndrange=size($R))
    end |> esc
end
```

Cases | Validation, Benchmarking & Profiling

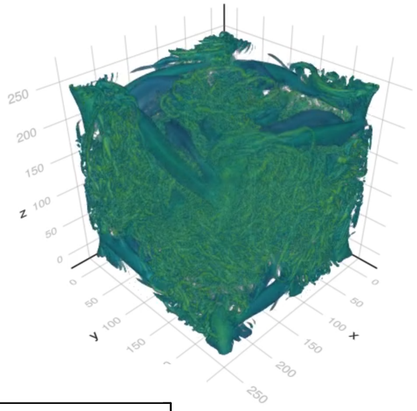
TGV



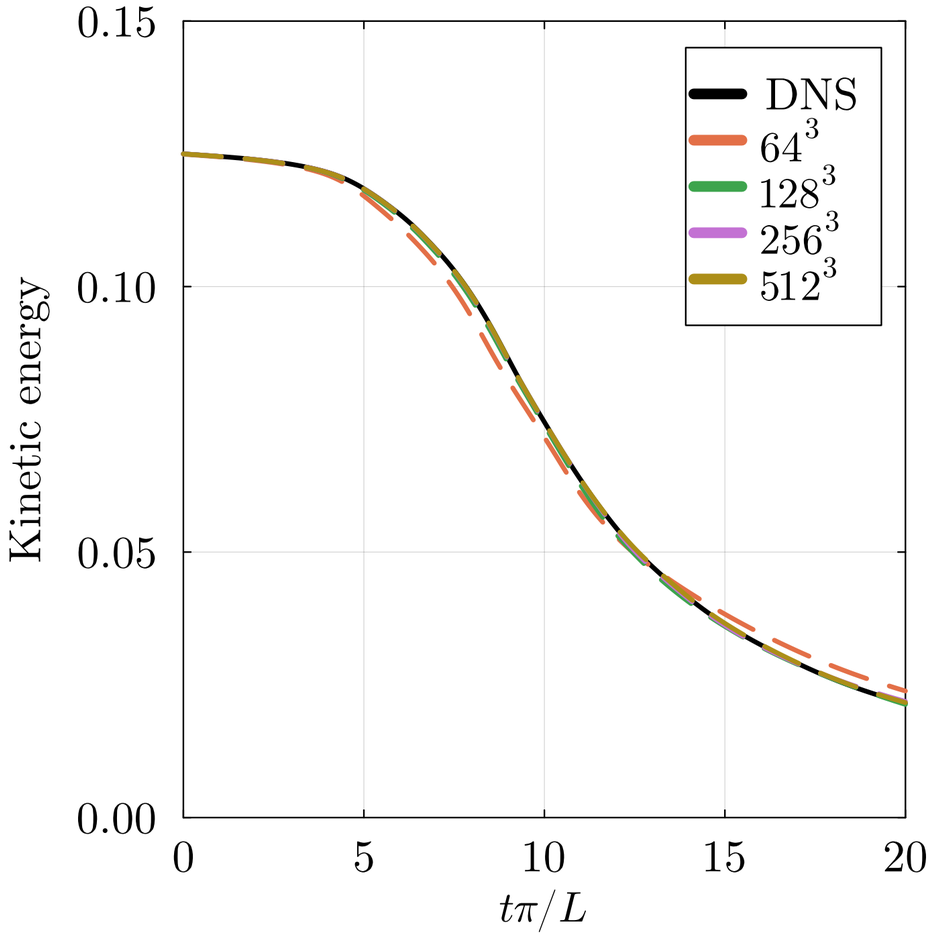
Sphere

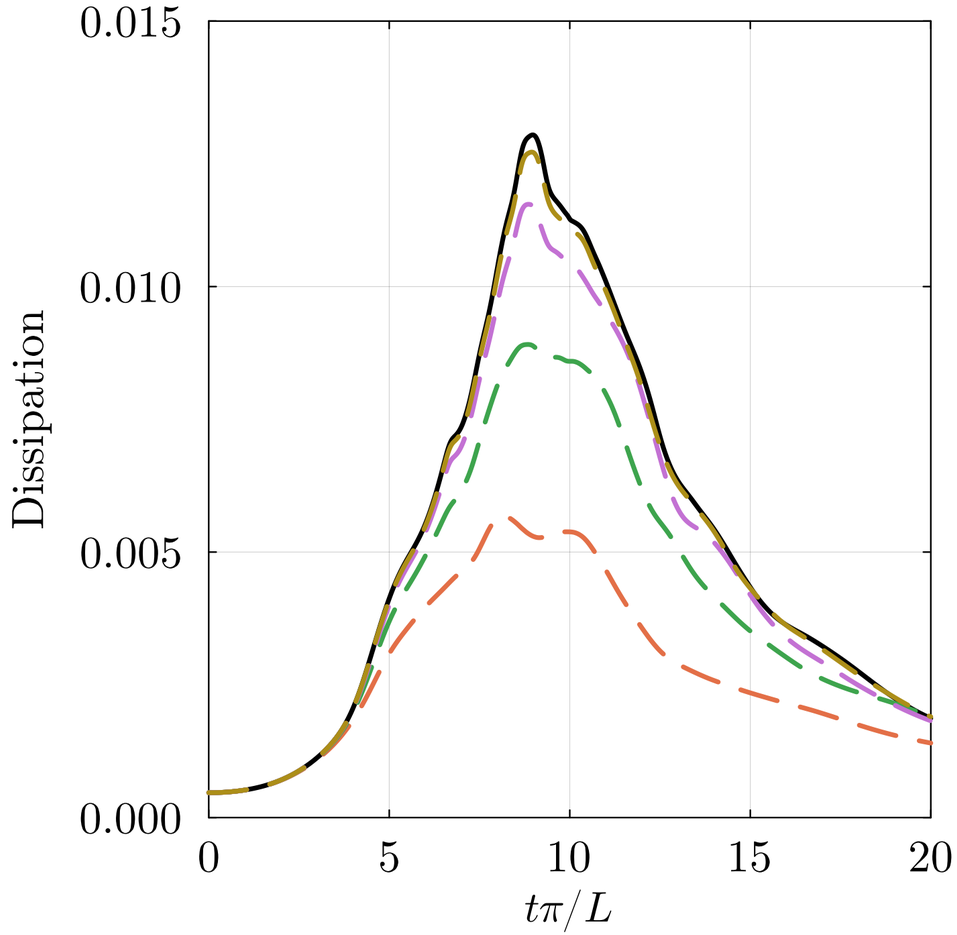


Moving cylinder



TGV

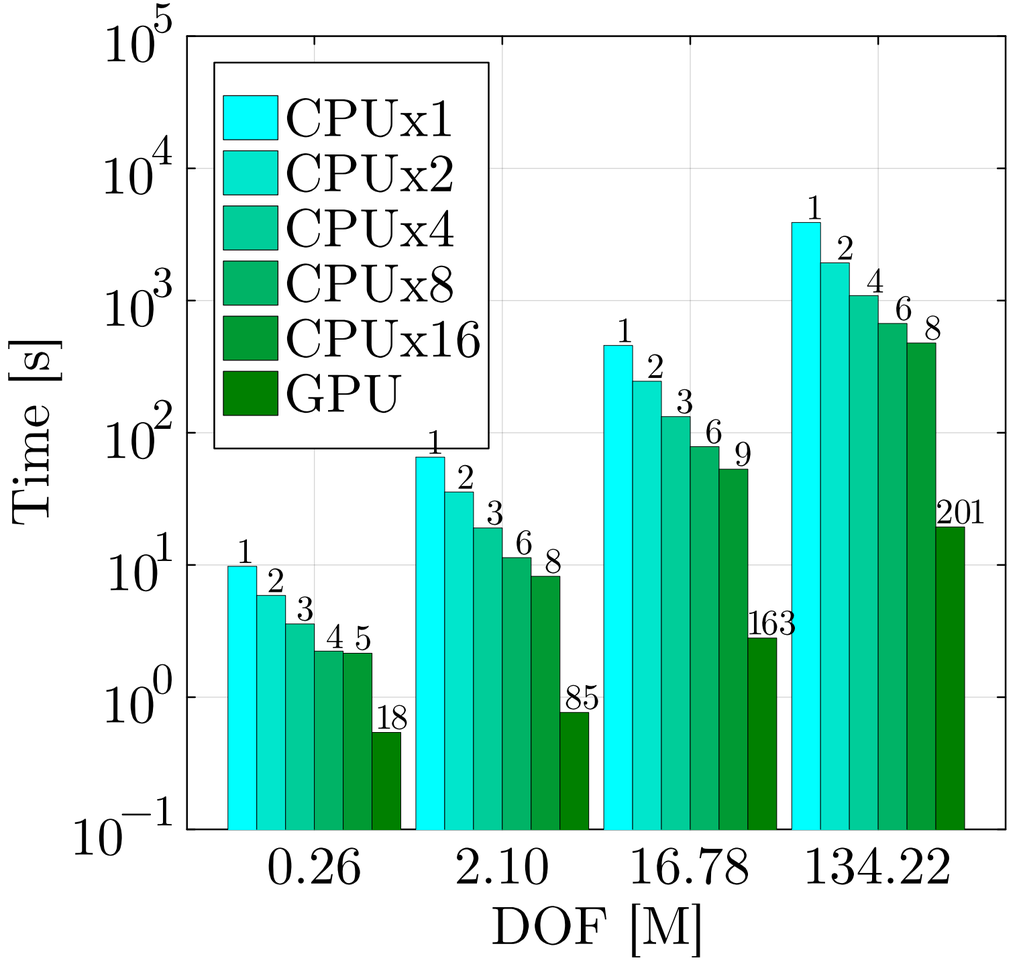

0.15
0.10
0.05
0.00
Kinetic energy
0
5
10
15
20
$t\pi/L$
DNS
64^3
128^3
256^3
512^3


0.015
0.010
0.005
0.000
Dissipation
0
5
10
15
20
$t\pi/L$

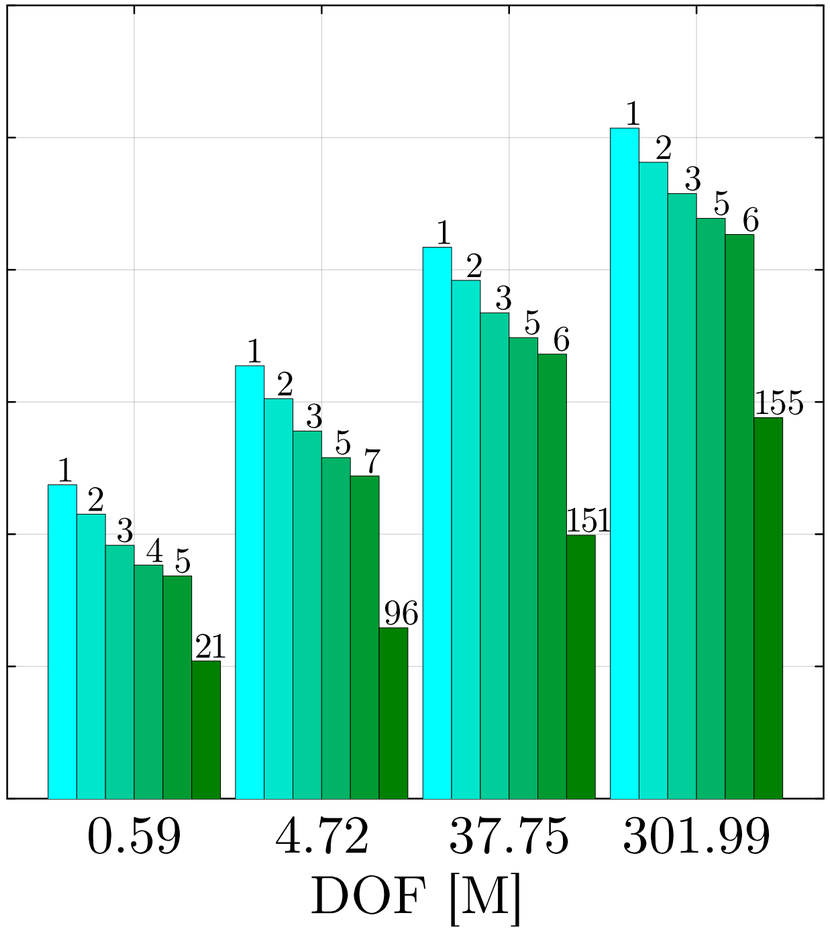
Float32, 100 dts, NVIDIA H100, Marenostrum5

302M DOF consume 39GB VRAM

TGV

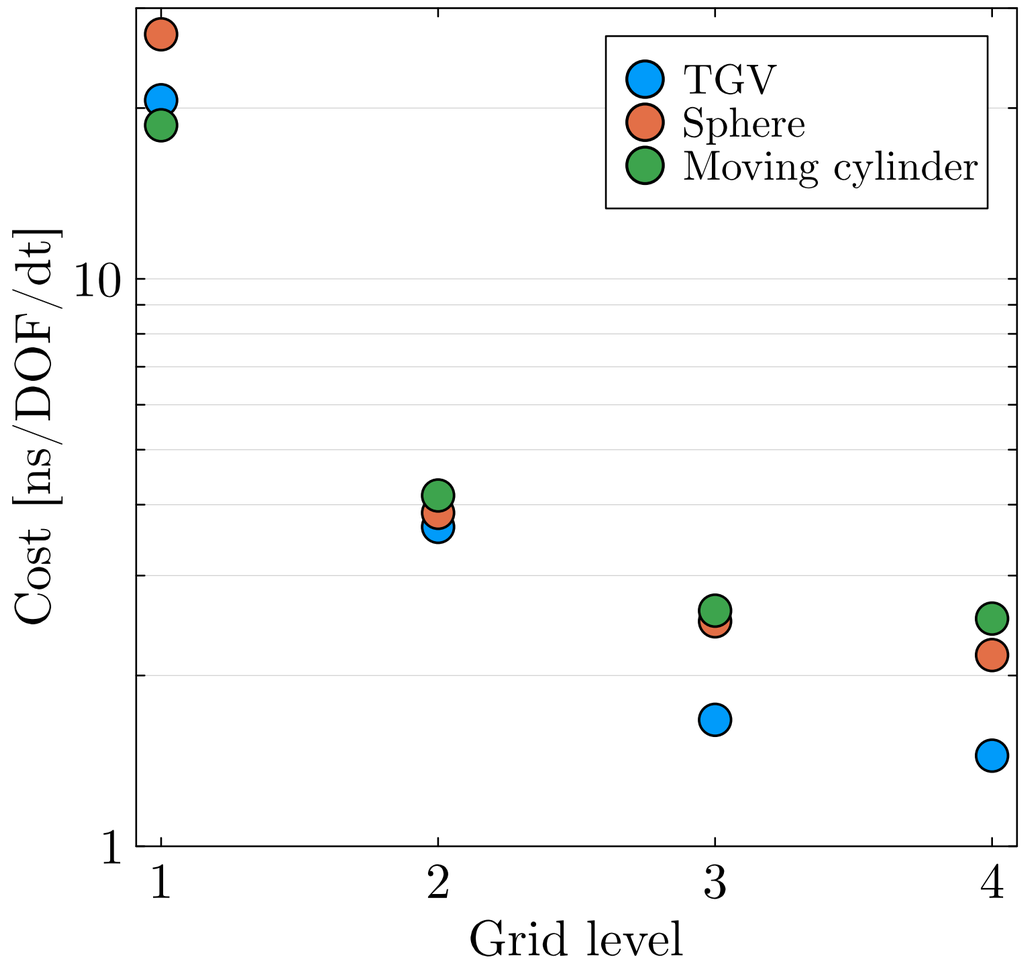


Moving cylinder



Float32, 100 dts, NVIDIA H100, Marenostrum5

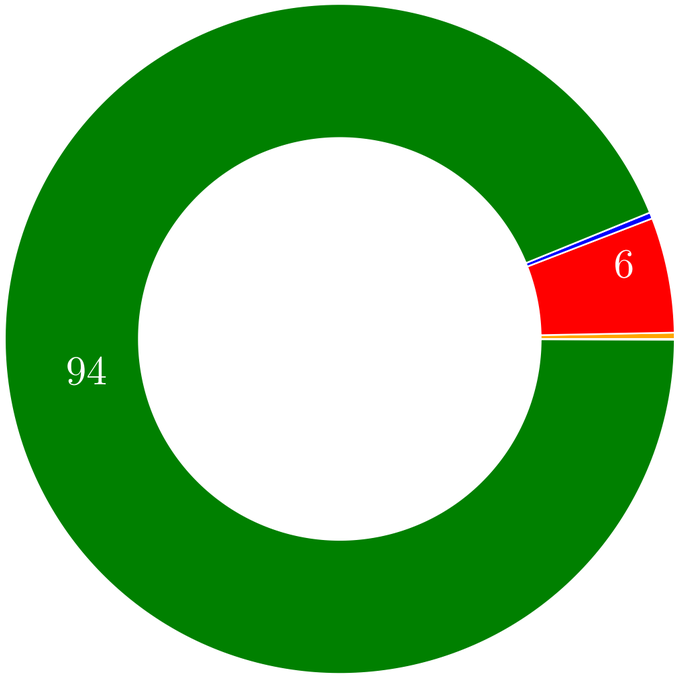
302M DOF consume 39GB VRAM



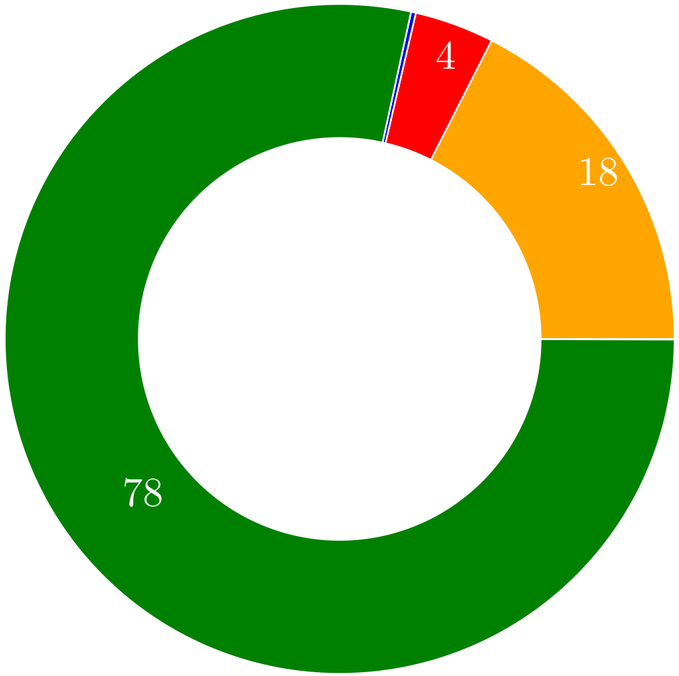
Float32, 1000 dts, NVIDIA GeForce RTX 4060 Laptop GPU

NVTX.jl

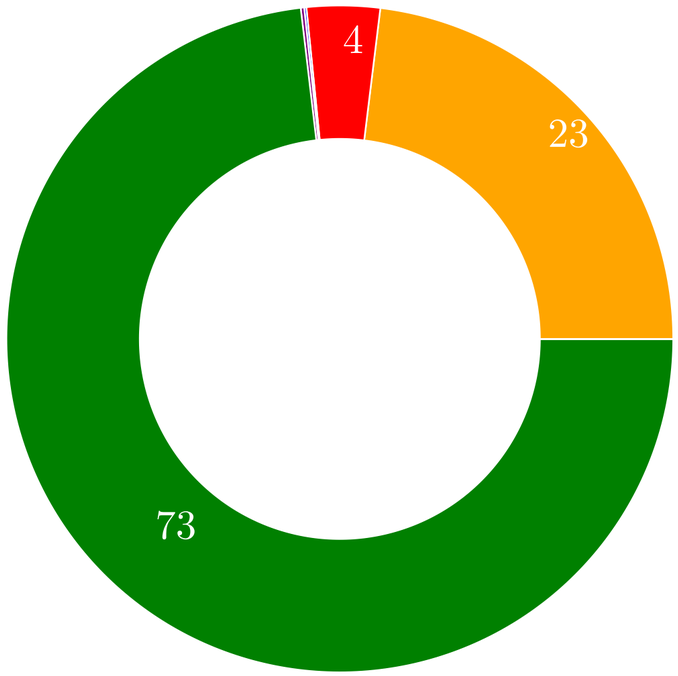
TGV

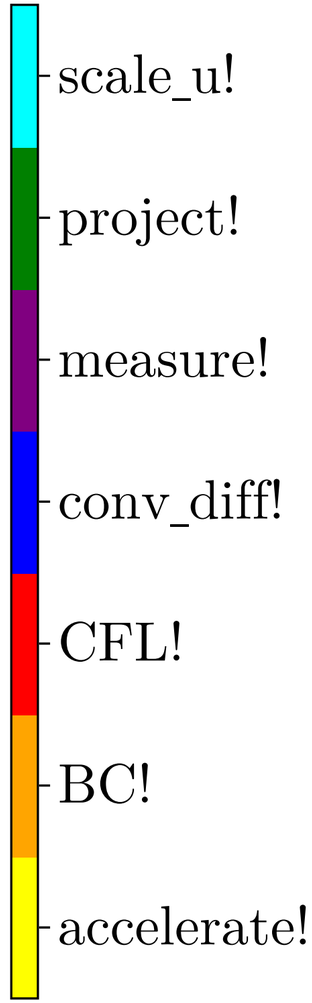


Sphere



Moving cylinder



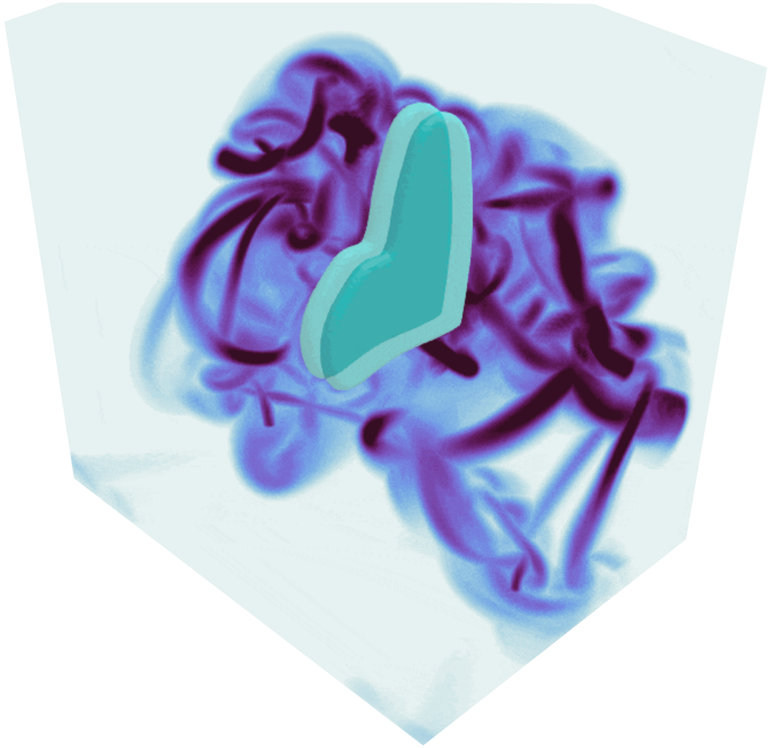


`WaterLily.jl` recap

- CFD solver with backend-agnostic execution thanks to `KernelAbstractions.jl`
- Simulation of dynamic bodies using immersed boundary method and signed distance function
- Speedups up to 200x on a cluster-grade GPU
- Comptetitive cost [ns/DOF/dt] wrt well-stablished CFD solvers written in C/C++/Fortran
- AD-ready on CPU backend using `ForwardDiff.jl`

Coming soon (and not so soon)

- Memory-distributed parallelism (`MPI.jl`)
- AD on both CPU and GPU through `Enzyme.jl`
- Optimized Biot-Savart boundary conditions for confined domains
- Volume-of-fluid (VOF) for multiphase flows
- Grid stretching

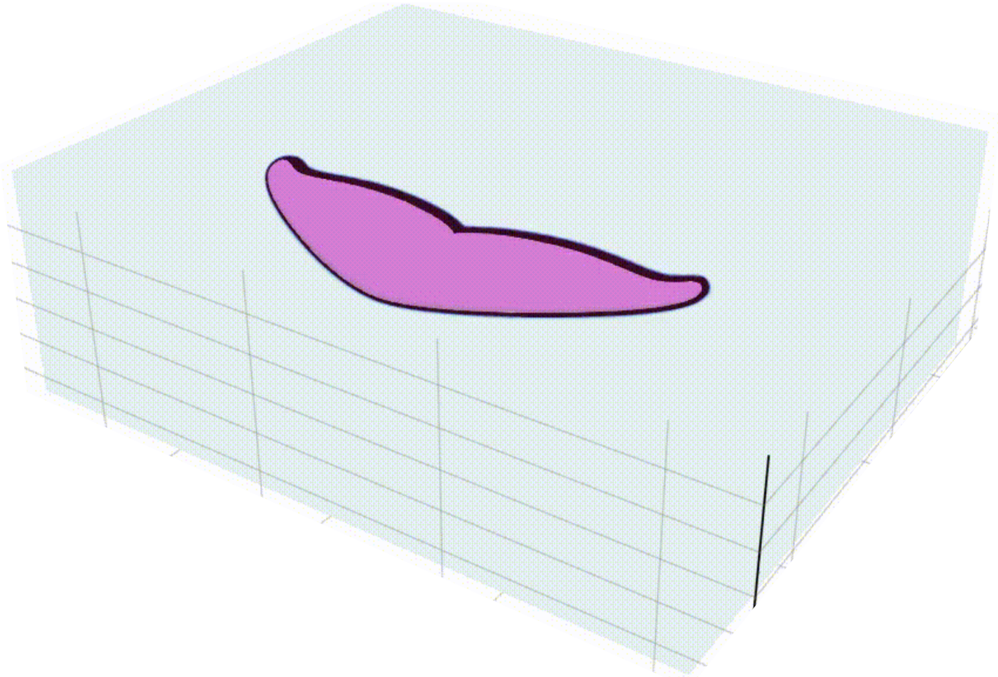


WaterLily.jl: A differentiable solver to simulate fluid flow and dynamic bodies with heterogeneous execution

Bernat Font*, Gabriel D. Weymouth

JuliaCon 2024

WaterLily.jl



Slides

TU Delft

b.font@tudelft.nl