WaterLily.jl

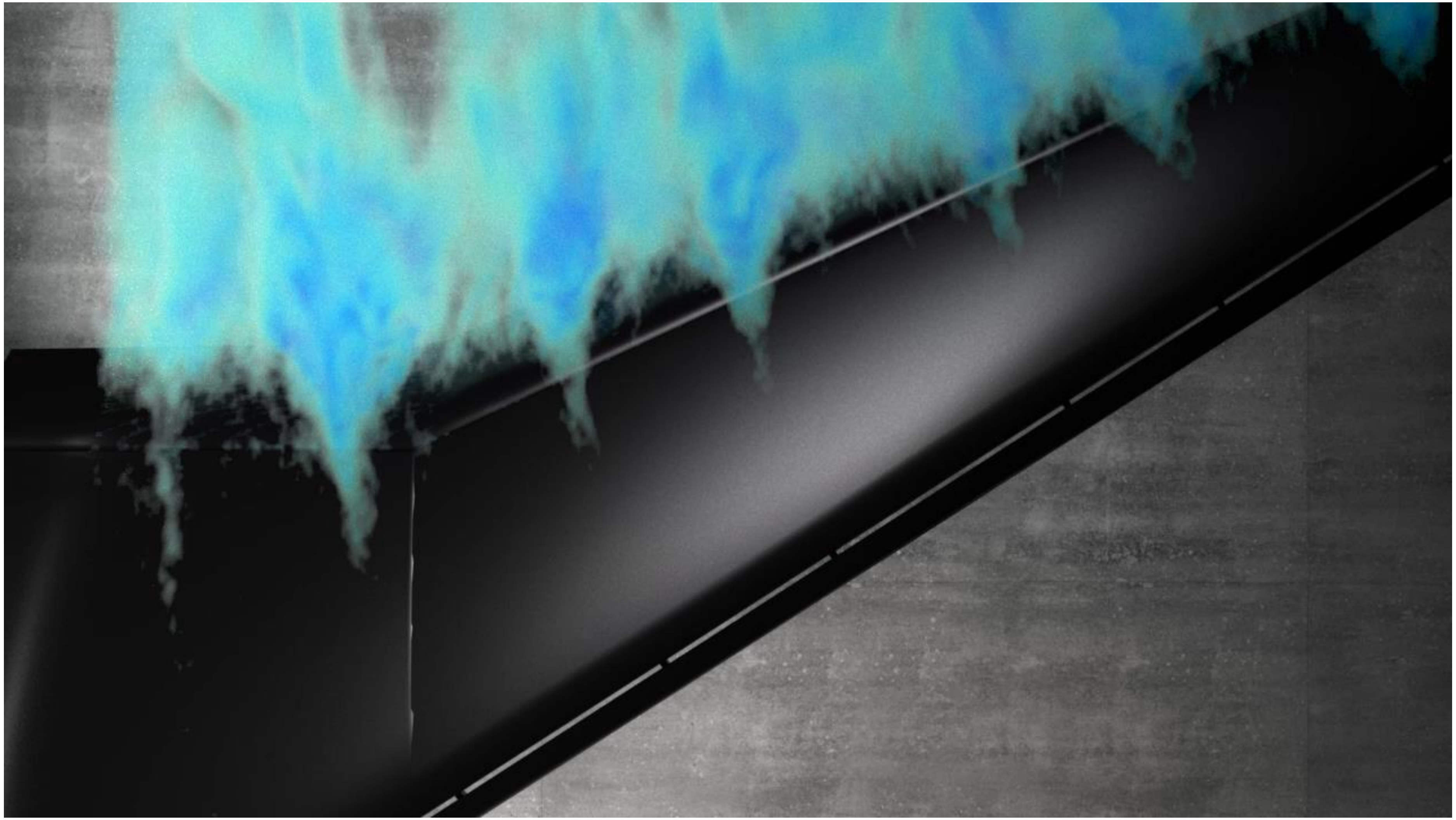
A fast and flexible CFD solver with heterogeneous execution

Dr. Bernat Font, Prof. Gabriel D Weymouth

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

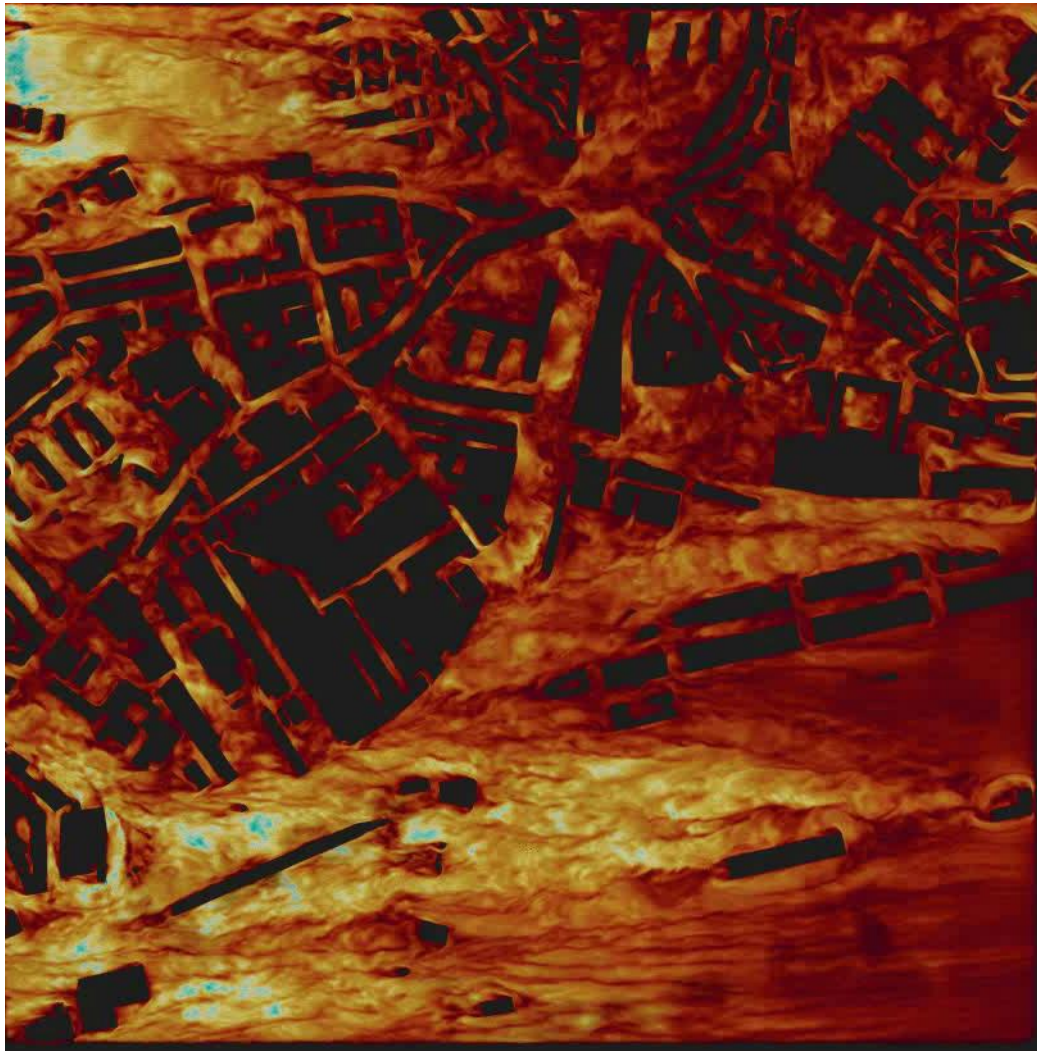
TU Delft

Background: Computational Fluid Dynamics is hard!



Large-scale CFD group @ BSC

Background: Computational Fluid Dynamics is hard!



Large-scale CFD group @ BSC

Background: Computational Fluid Dynamics is hard!

- Simulation of fluid flows for all kind of problems

  - Weather forecasting, designing aircrafts, cars, boats..., combustion, even to simulate blood flow.

- Physics governed by the Navier—Stokes equations

$$\partial_t \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + Re^{-1} \nabla^2 \mathbf{u}$$

$$\nabla \cdot \mathbf{u} = 0$$

- Costly!

  - Wide range of scales in space and time.
  - HPC required to perform simulations with reasonable timeframes.

Julia for CFD?

- CFD solvers must been written in compiled languages (eg. Fortran, C/C++).

  ☑ Julia is compiled, so it’s fast!

- In CFD, HPC is a must.

  ☑ Julia is parallel:

  `LoopVectorization.jl, Polyester.jl, MPI.jl`

- Modern CFD solvers should run on both CPUs and GPUs.

  ☑ Julia can run in most architectures:

  `CUDA.jl, AMDGPU.jl, KernelAbstractions.jl`

Julia is CFD! Additional benefits

- It solves the two-language problem:
  - ☑ Scientific computing, machine learning, postprocessing, and visualization, coexist in the same environment.

- Active community:
  - ☑ Engage with core developers and benefit from latest developments.

So what is WaterLily?

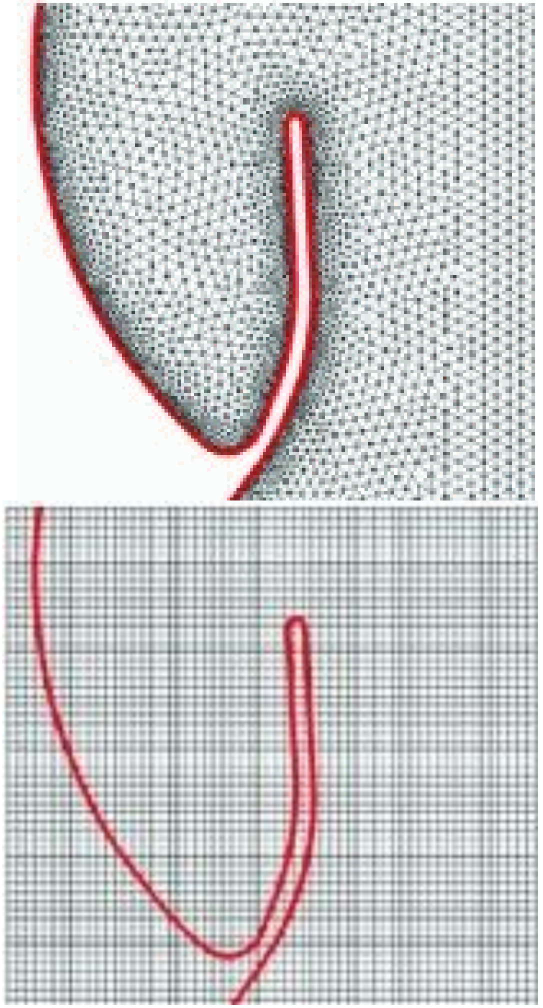
- Incompressible flow, finite-volume, CFD solver
- Immersed boundary

Trivial for computers, developers & users

- Field data is a simple array
- All algorithms simplified
- Grid-design is **unnecessary**
(primary source of user error & learning curve)

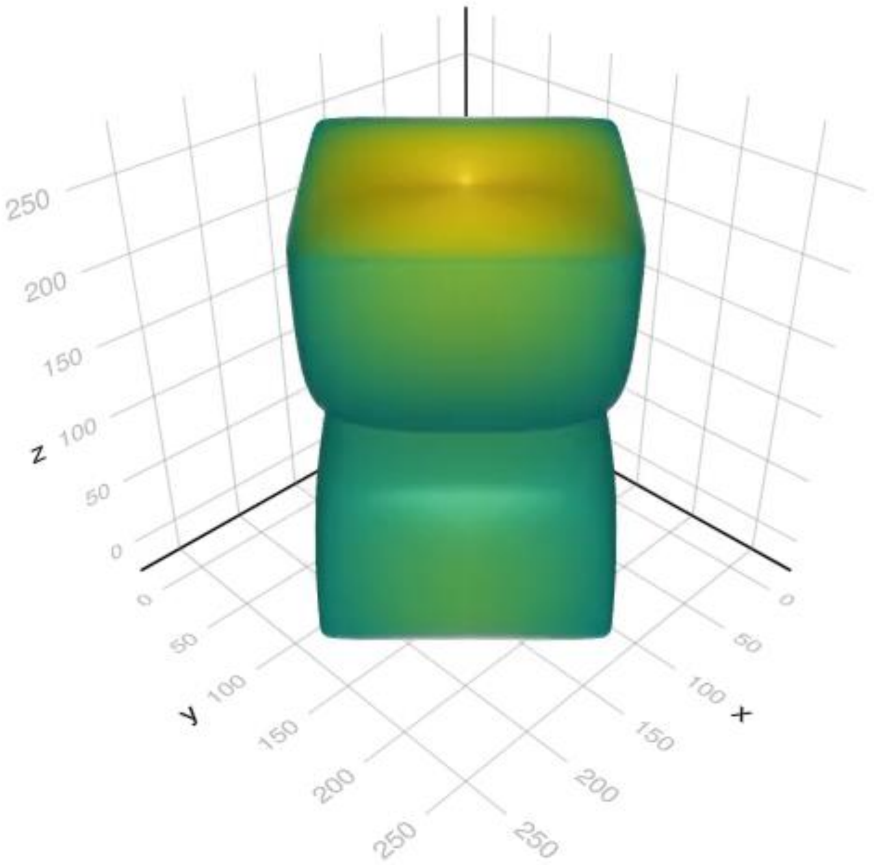
Numerical advantages over fitted grid methods

- Accurate & perfectly conservative fluxes
- No limitations on boundary topologies/motion
- Low cost/memory enables more DOF
- **Geometric** multi-grid pressure solver



- And we can do very cool simulations…

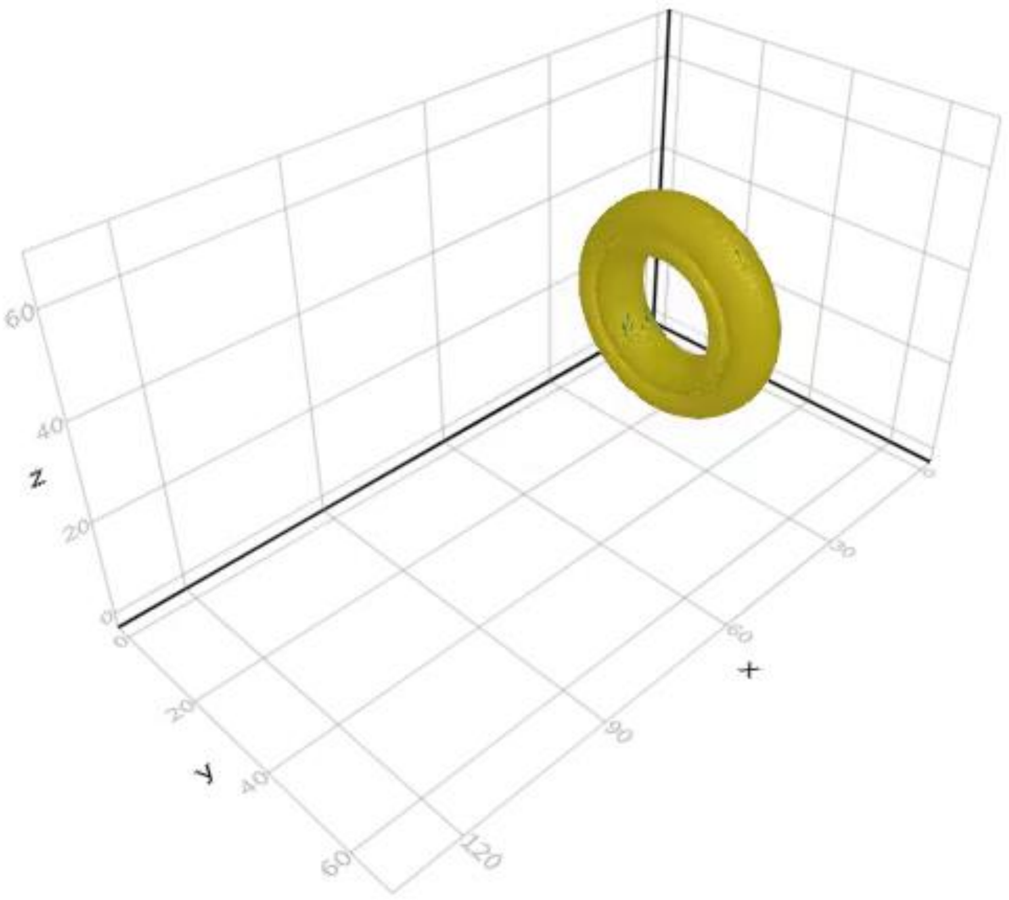
This is WaterLily! Taylor—Green Vortex




```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                # scaled coordinates
        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
```

- 50M DOFs.
- 30min to run and render on a laptop with NVIDIA card.
- 3ns/dt/DOF on RTX 3080Ti (top CFD solvers are around 5-10 ns/it/DOF in a similar setup).

This is WaterLily! Donut




```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
```

This is WaterLily! Jellyfish

```
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
```

Porting from serial CPU to backend-agnostic

- Main objectives:
  - ☑ **KISS** (keep it simple, stupid!): Both developers and users should be able to quickly adapt to the ported solver. Avoid complexity if possible.
  - ☑ **Performance**: Benchmark (`@btime`) and test all the new additions, look for edge cases.
  - ☑ **General**: Ability to run in different architectures in a unified framework.

- Architecture-targeted options (**JuliaGPU**):
  - `CUDA.jl`: NVIDIA devices.
  - `AMDGPU.jl`: AMD devices.
  - `oneAPI.jl`: Intel devices.
  - `Metal.jl`: Apple devices.

Porting from serial CPU to backend-agnostic

a) Using the intrinsic kernels: Array programming, Linear algebra, FFT

```
using CUDA
using AMDGPU

A = rand(3, 3) |> CUDA.CuArray # or AMDGPU.ROCArray
B = rand(3, 3) |> CUDA.CuArray # or AMDGPU.ROCArray

C = A * B # C is a CuArray or ROCArray, depending on A, B
```

b) Writing custom kernels

```
N = 2^20
x_d = CUDA.fill(1.0f0, N)  # a vector stored on the GPU filled with 1.0 (Float32)
y_d = CUDA.fill(2.0f0, N)  # a vector stored on the GPU filled with 2.0

function gpu_add!(y, x)
    index = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    stride = blockDim().x * gridDim().x
    for i = index:stride:length(y)
        y[i] += x[i]
    end
    return nothing
end

numblocks = ceil(Int, N/256)
@cuda threads=256 blocks=numblocks gpu_add!(y_d, x_d)
```

Porting from serial CPU to backend-agnostic

- Custom kernels are hard to unify for all the different backends…

AMDGPU	CUDA
`workitemIdx`	`threadIdx`
`workgroupIdx`	`blockIdx`
`workgroupDim`	`blockDim`
`gridItemDim`	No equivalent
`gridGroupDim`	`gridDim`
`groupsize`	`threads`
`gridsize`	`blocks`
`stream`	`stream`

`KernelAbstractions.jl` to the rescue!

- Writing custom kernels that are multi-threaded on CPU execution, and can run on all the different devices (NVIDIA, AMD, Intel, Apple). Exports `@kernel`

```julia
using CUDA
using CUDA.CUDAKernels
CUDA.allowscalar(false) # prevents against serial execution on CPU
using KernelAbstractions

@kernel function matmul_kernel!(a, b, c)
    i, j = @index(Global, NTuple)
    # creating a temporary sum variable for matrix multiplication
    tmp_sum = zero(eltype(c))
    for k = 1:size(a)[2]
        tmp_sum += a[i,k] * b[k, j]
    end
    c[i,j] = tmp_sum
end

a = rand(256, 123) |> CUDA.CuArray
b = rand(123, 45) |> CUDA.CuArray
c = zeros(256, 45) |> CUDA.CuArray

backend = KernelAbstractions.get_backend(a) # CPU(), CUDABackend(), ROCBackend()
kernel! = matmul_kernel!(backend, 256) # specialise kernel for the backend and workgroup size
kernel!(a, b, c, ndrange=size(c)) # launch kernel
```

WaterLily approach: Abstracting loops, divergence kernel

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

```
δ(d,::CartesianIndex{D}) where {D} = CartesianIndex(ntuple(j -> j==d ? 1 : 0, D)) # returns (1, 0) or (0, 1) for 2D
@inline ∂(a,I::CartesianIndex{D},u::AbstractArray{T,n}) where {D,T,n} = u[I+δ(a,I),a]-u[I,a] # finite difference
inside(a) = CartesianIndices(ntuple(i-> 2:size(a)[i]-1,ndims(a))) # exclude boundary elements

# 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

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

```
N = (2^9, 2^9, 2^9)
σ = zeros(N)
u = rand(N..., length(N))
divergence!(σ, u)
```

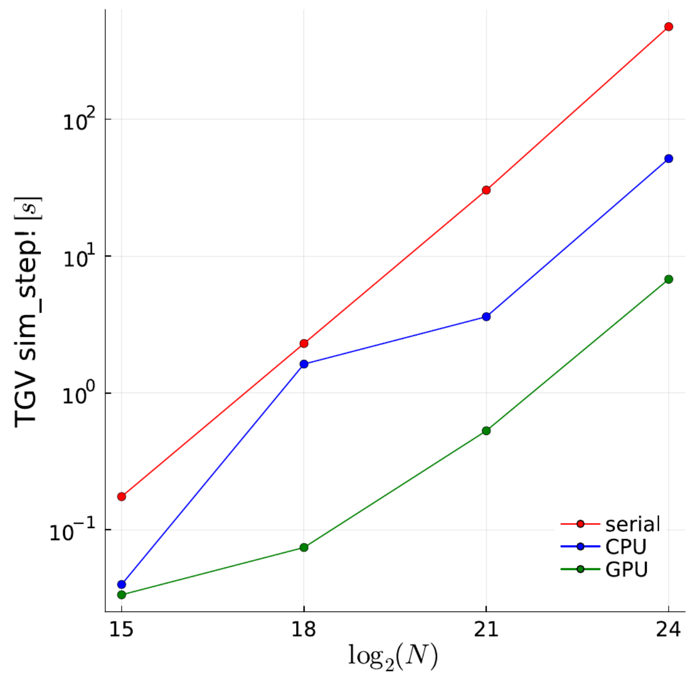
WaterLily approach: Re-writting the `@loop` macro

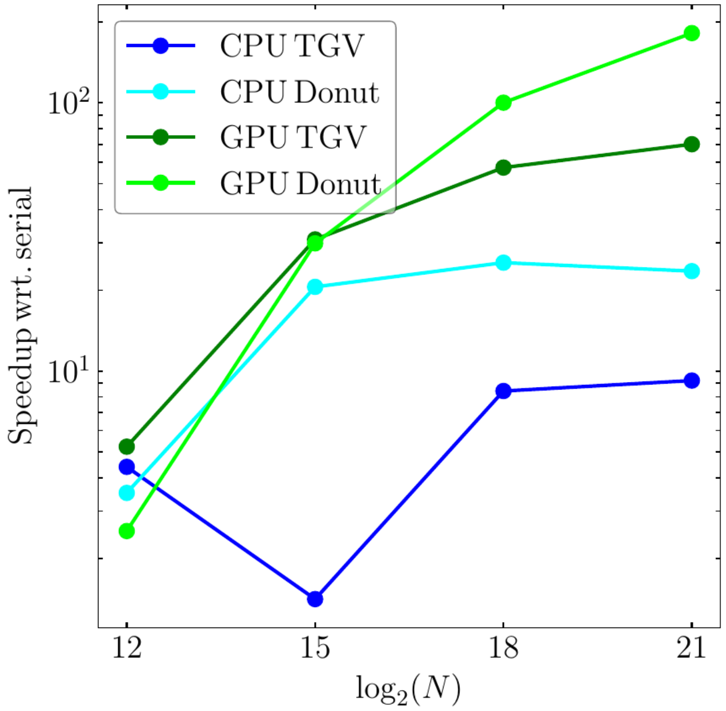
```julia
using KernelAbstractions: get_backend, synchronize, @index, @kernel, @groupsize
using CUDA: CuArray

# 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

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

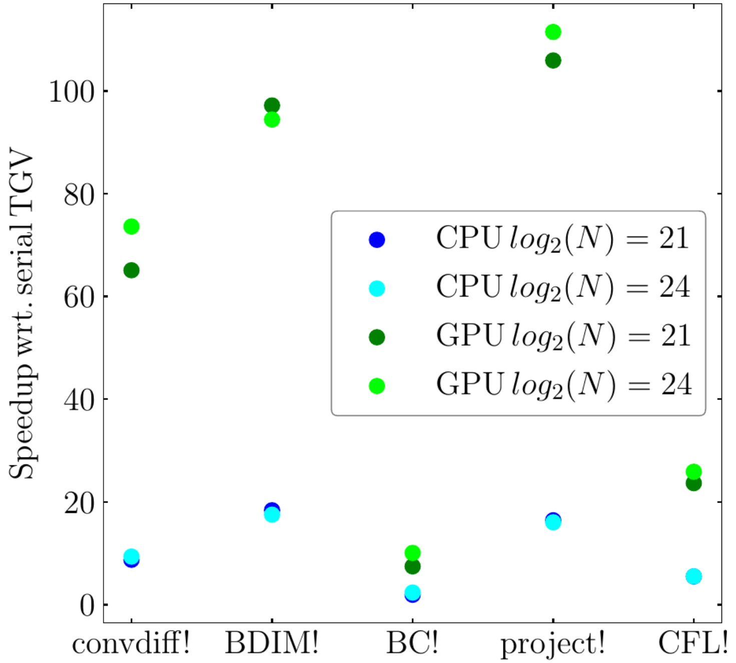
```julia
N = (2^9, 2^9, 2^9)
σ = zeros(N) |> CuArray
u = rand(N..., length(N)) |> CuArray
divergence!(σ, u)
```

✅ Refactor implied going from LOC < 1000 to... LOC < 1000

Benchmarks



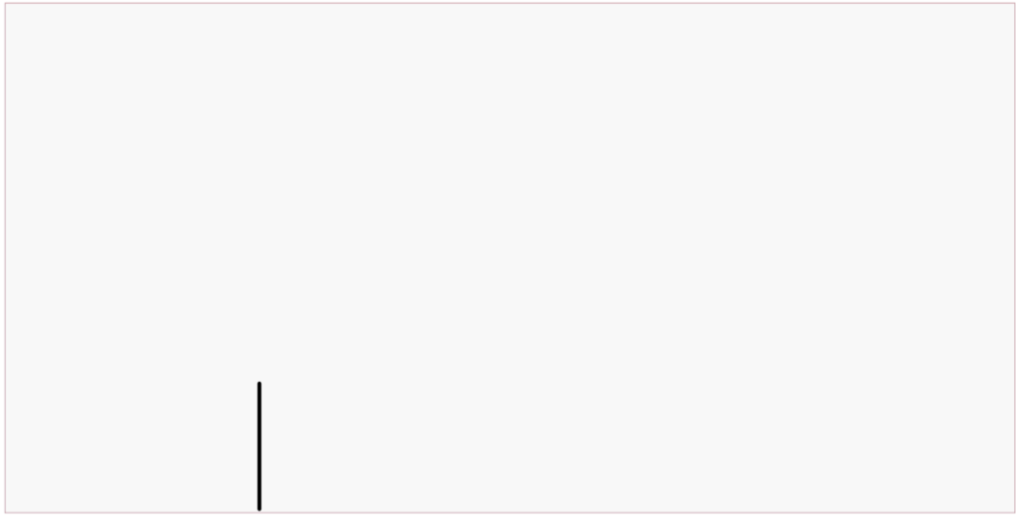




- The more loaded the GPU is, the better performance it achieves.
- On the larger Donut mesh, an speedup > 180x can be observed on the GPU.
- CPU multi-threading performance increase saturates with mid-size grids.
- Expensive kernels are the ones that benefit the most.

Next steps in WaterLily

- Full solver differentiability
- Distributed parallelisation: multi-GPU (with `MPI.jl`), matrix-free geometric multi-grid
- NURBS surfaces
- Multi-phase flow (VOF)
- FEA solver



By Dr. Marin Lauber (TU Delft)

Relevant Links

- `WaterLily.jl`: github.com/weymouth/WaterLily.jl
- `KernelAbstractions.jl`: github.com/JuliaGPU/KernelAbstractions.jl
- Numerial methods in WaterLily: doi.org/10.1016/j.jcp.2011.04.022
- Porting WaterLily to heterogeneous backend: b-fg.github.io/2023/05/07/waterlily-on-gpu
- ParCFD preprint: arxiv.org/abs/2304.08159

WaterLily.jl

A fast and flexible CFD solver with heterogeneous execution

Dr. Bernat Font, Prof. Gabriel D Weymouth

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TU Delft