

Interactive Multi-GPU Supercomputing with Julia

CSCS User Lab Day – Meet the Swiss National Supercomputing Centre

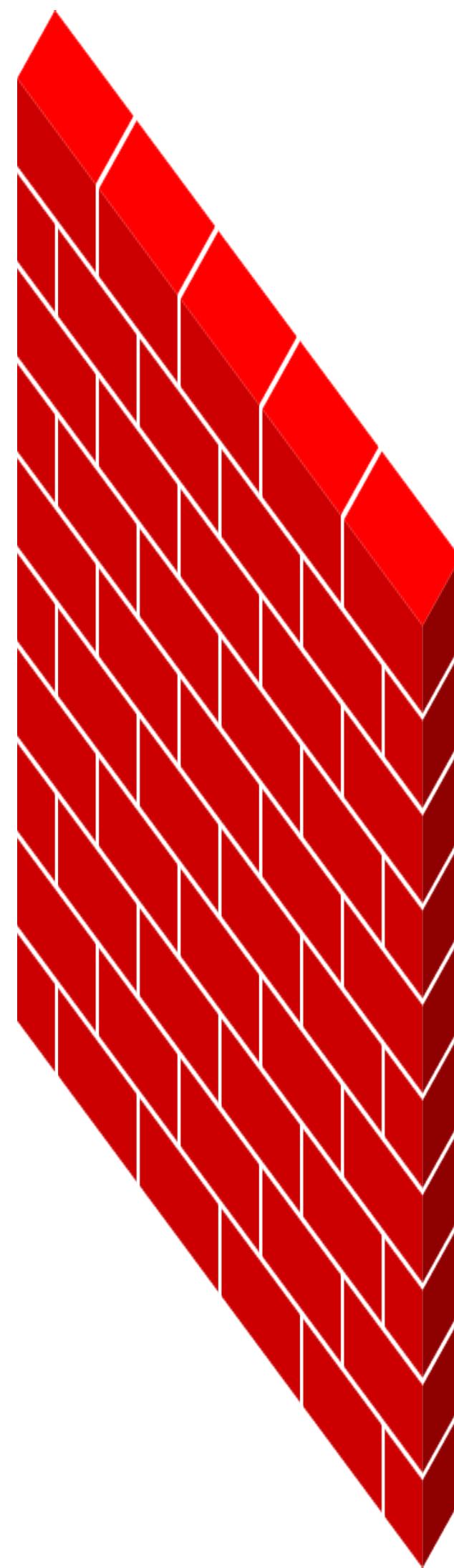
Samuel Omlin

September 9th 2019



Prototype

```
P = rand(4,3)
```



Production code

```
float* P;  
P = malloc(...);  
rand(P,...);
```

The two language problem

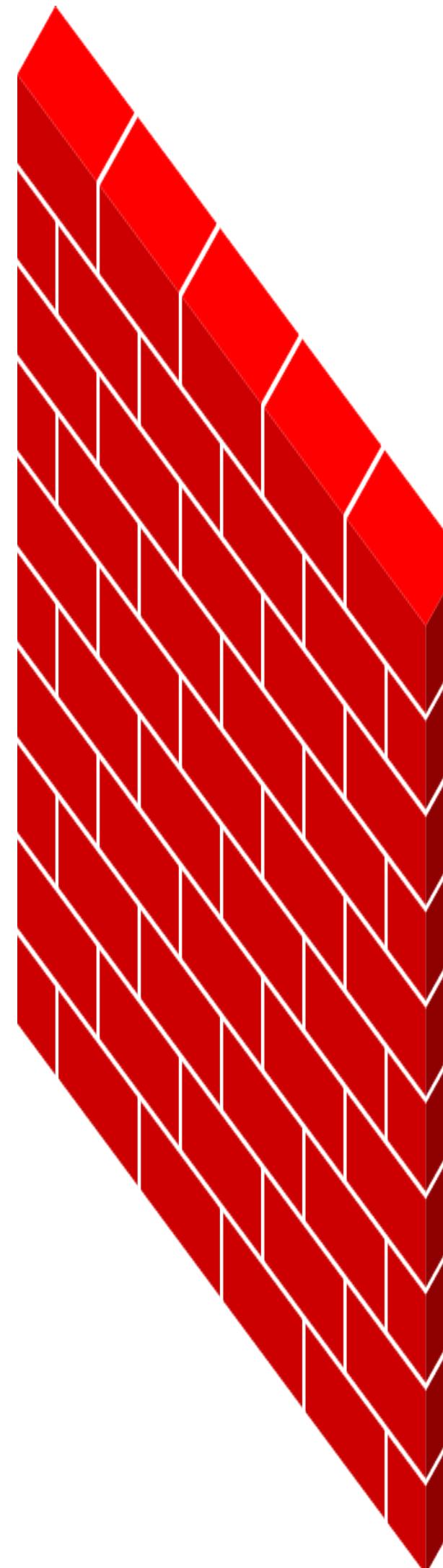
Prototype (MATLAB / Python / ...)

simple & high-level

interactive

low development cost

slow



Production code (C / C++ / Fortran / ...)

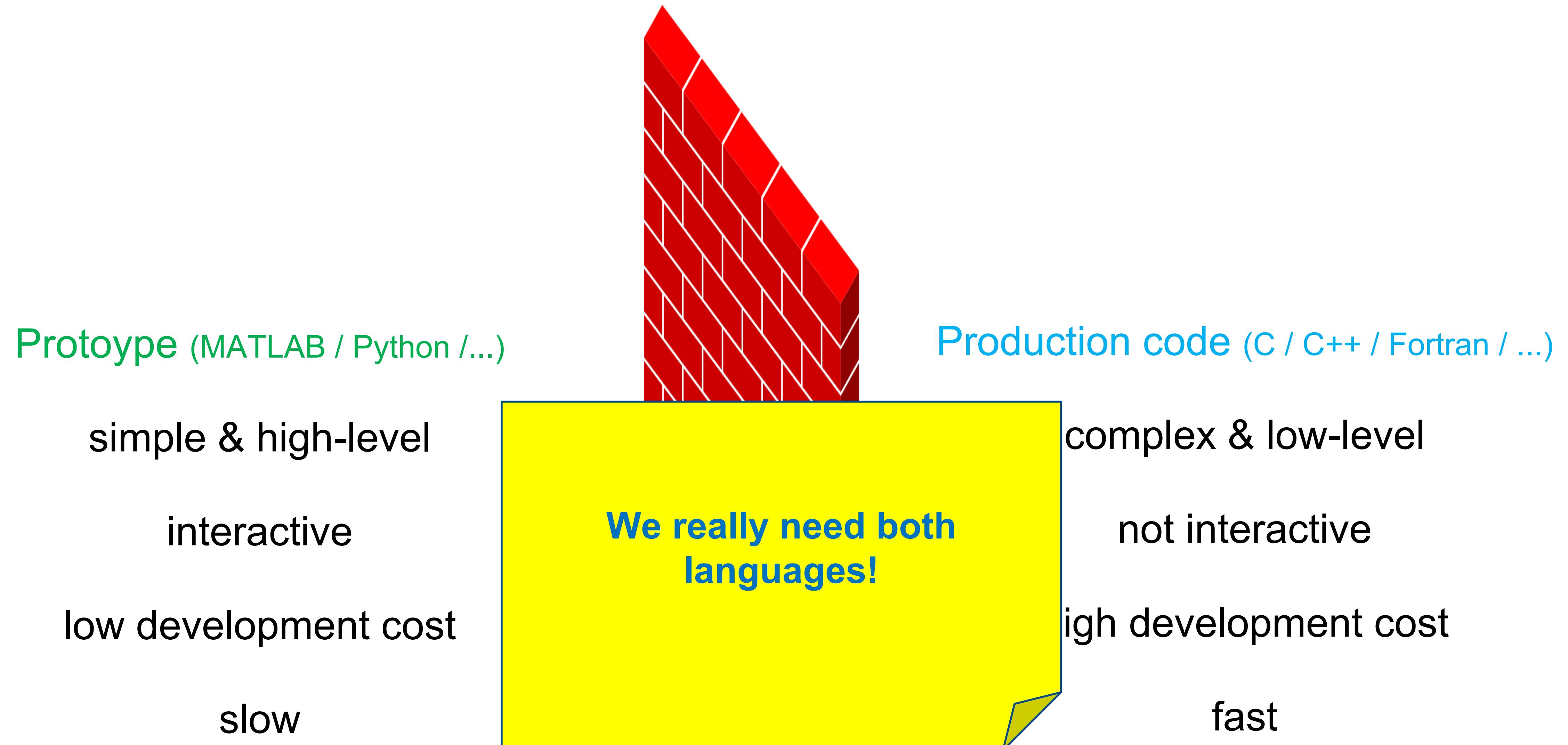
complex & low-level

not interactive

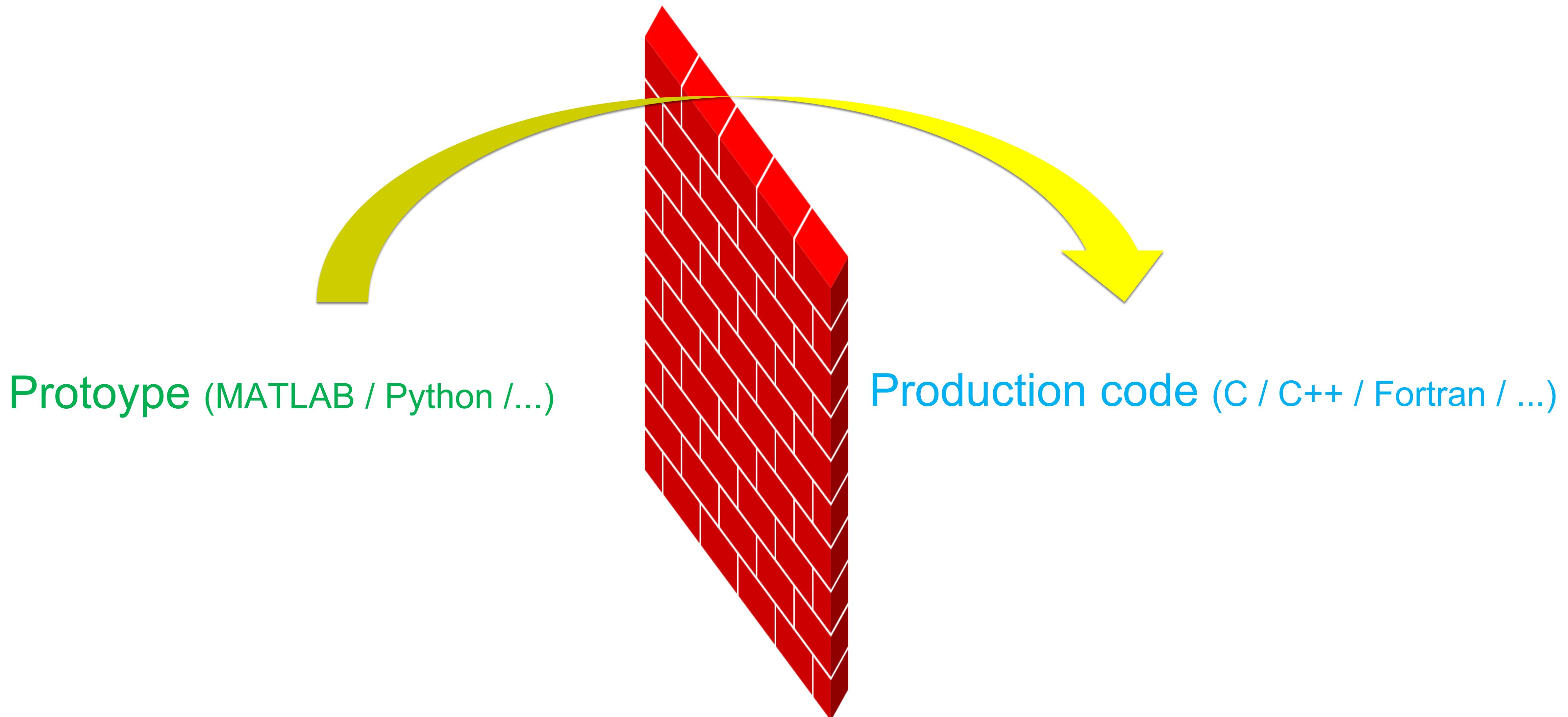
high development cost

fast

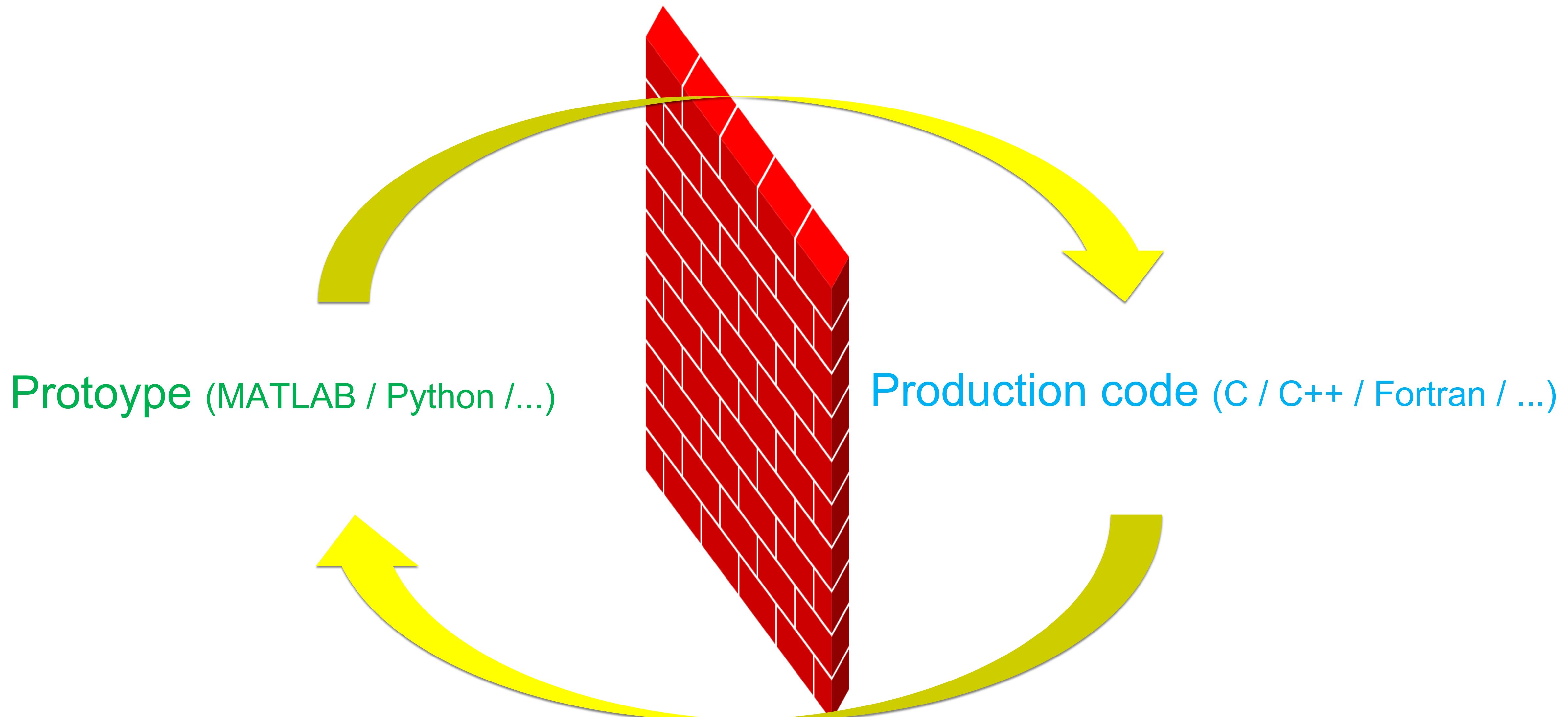
The two language problem



The two language problem



The two language problem



Solution

A language that can be used for both
Prototype & Production code

Solution

A language that can be used for both

Prototype & Production code



Solution



simple & high-level

interactive

low development cost

fast

Solution



simple & high-level
interactive
low development cost
fast

Fast and interactive???

Julia code is compiled, yet
only shortly before you use it
the first time.

Solution

Mature?

Is “adult” now:
v1.0.0 released in Aug 2018



simple & high-level

interactive

low development cost

fast

Solution



simple & high-level

interactive

low development cost

fast

1-day fly?

Developed at MIT and by a
fast growing open-source
community

Solution



Too good to be true!

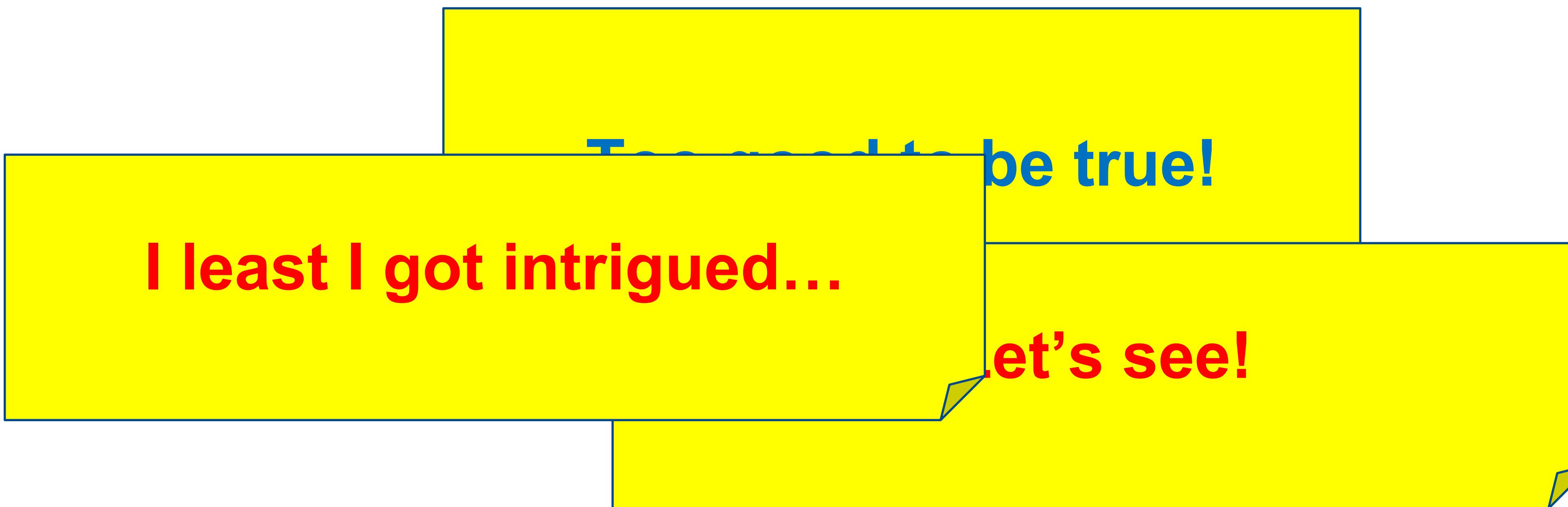
Solution



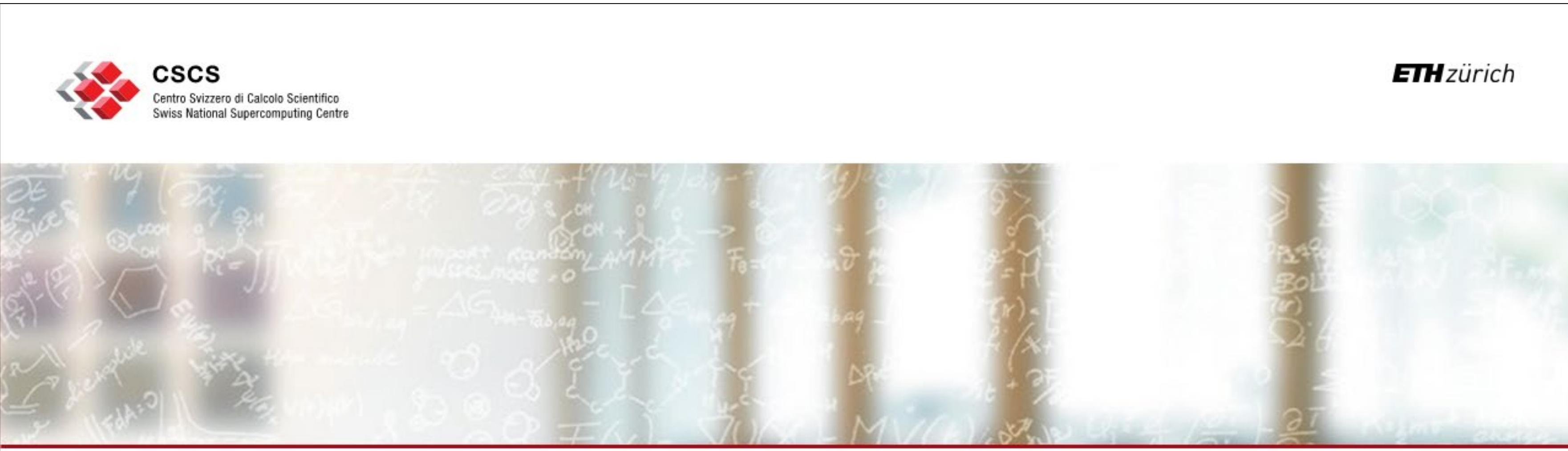
Too good to be true!

Let's see!

Solution



Case study



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETHzürich

A Nonlinear Multi-Physics 3-D Solver: From CUDA C + MPI to Julia

PASC19
Ludovic Räss^{1,2}, **Samuel Omlin**³, Yury Podladchikov²
June 14th 2019

1 Stanford University, Department of Geophysics, Stanford CA, USA
2 Swiss Geocomputing Centre, University of Lausanne, Lausanne, Switzerland
3 CSCS - Swiss National Supercomputing Centre, Lugano, Switzerland

 **Stanford**
University

 **Unil**
UNIL | Université de Lausanne
Swiss Geocomputing
Centre

Case study

Stanford University

PORTING A MASSIVELY PARALLEL MULTI-GPU APPLICATION TO JULIA: A 3-D NONLINEAR MULTI-PHYSICS FLOW SOLVER

Ludovic Räss^{1,2}, Samuel Omlin³, Yury Podladchikov²

¹ Stanford University, Department of Geophysics, Stanford CA, USA

² Swiss Geocomputing Centre, University of Lausanne, Lausanne, Switzerland

³ CSCS - Swiss National Supercomputing Centre, Lugano, Switzerland

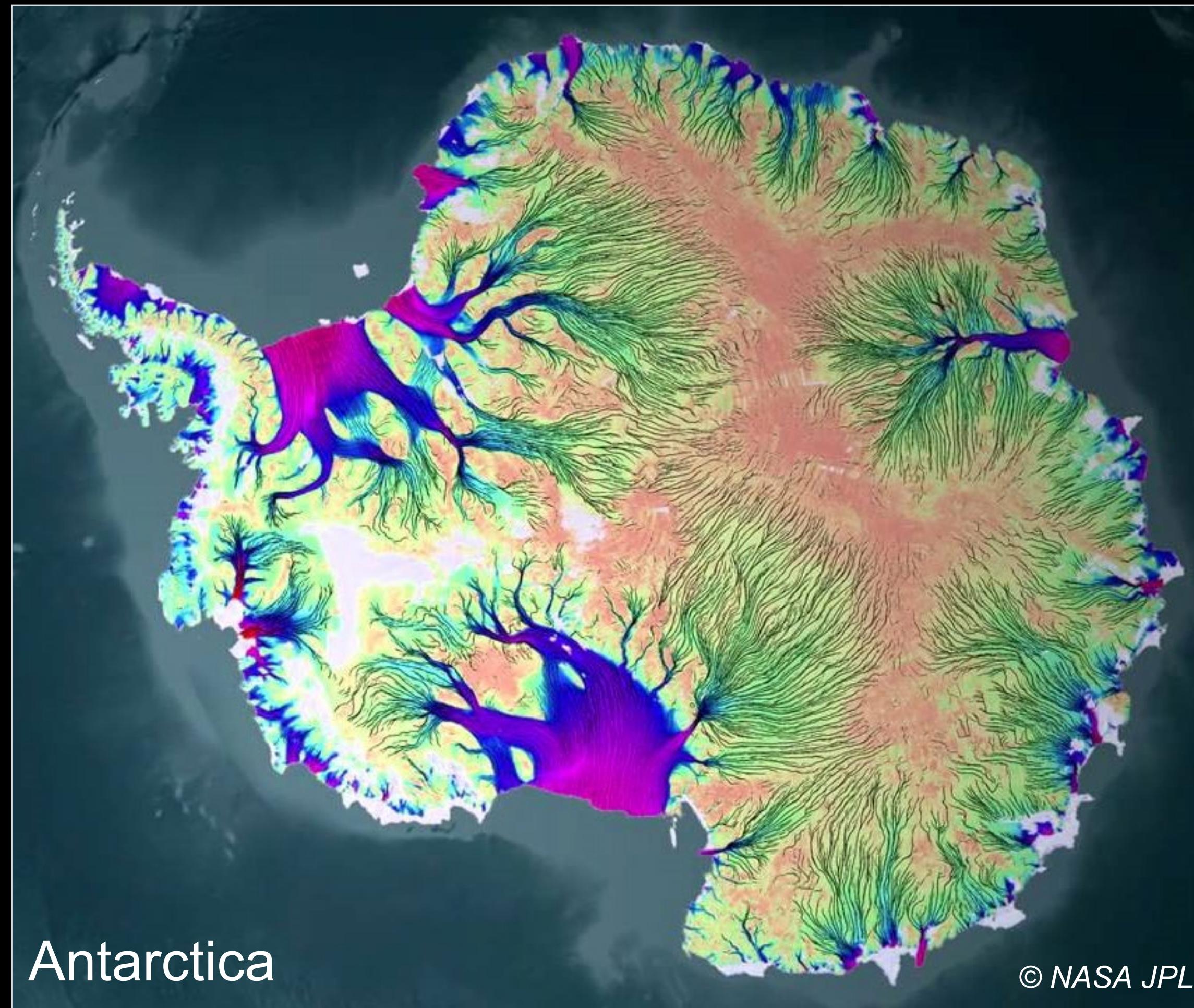
23 July 2019 | JuliaCon 2019 | Baltimore MD

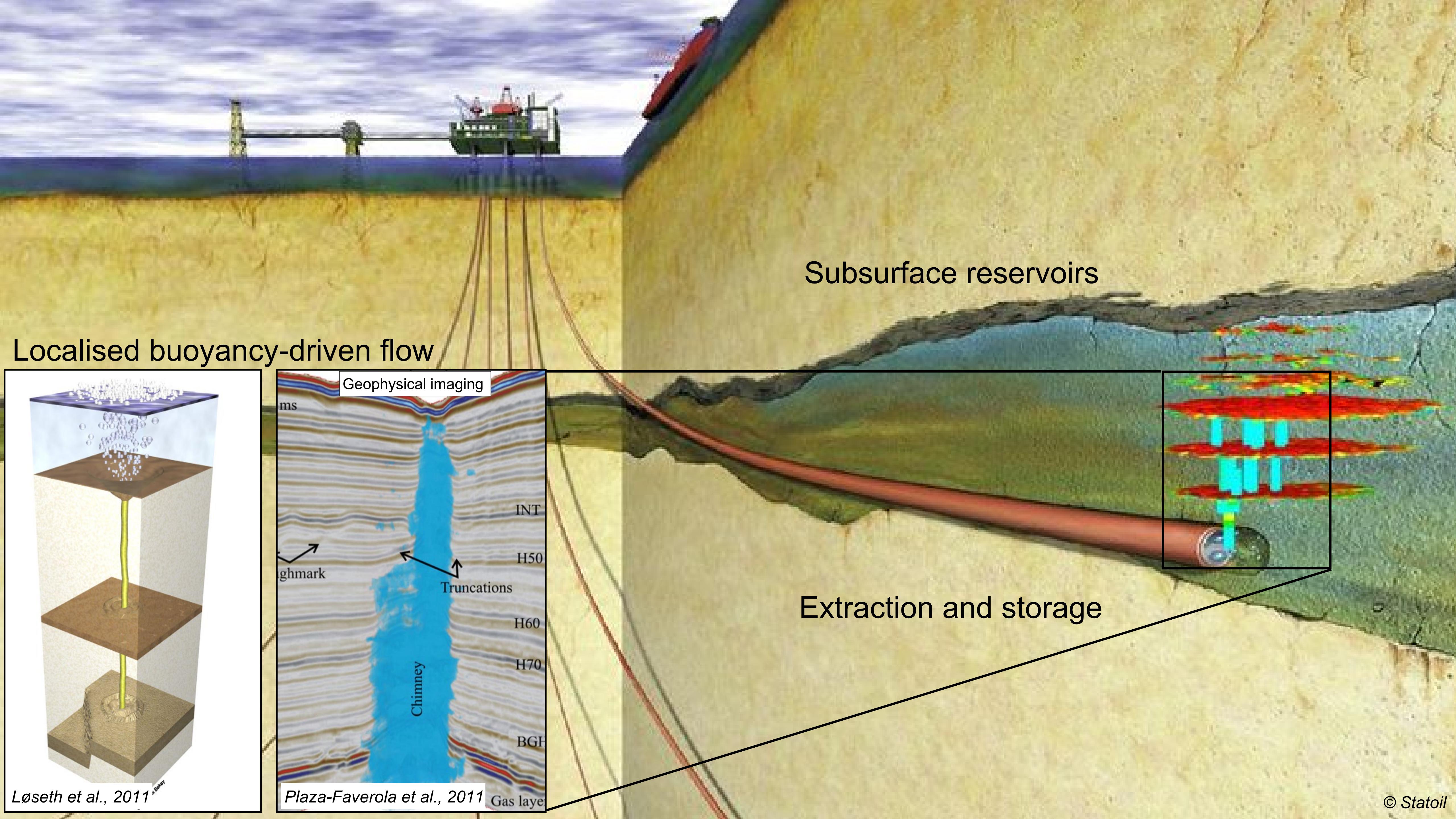


Agenda

- The two language problem 
- Case study: the solver
- Porting to Julia | Multi-GPU Programming in Julia
- Case study: results
- Case study: conclusions
- General conclusions
- Outlook & recommendations

Multi-physics flow localisation





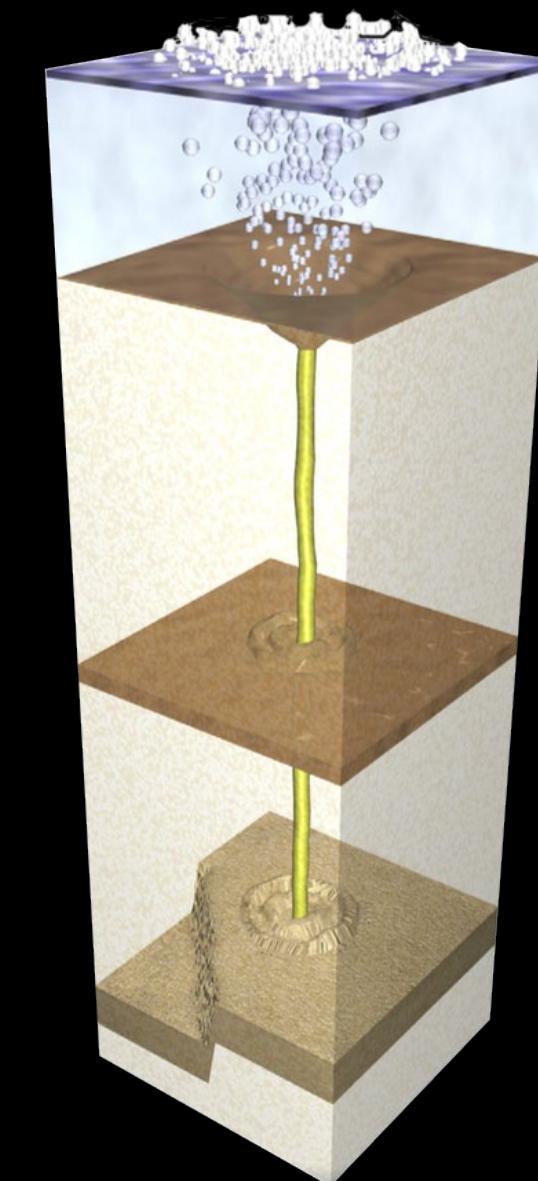
The multiphysics solver

...resolves complex meso-scale dynamics

- includes nonlinear multi-physics feedbacks
- captures *spontaneous* localisation in space + time
- handles highly localised action in large domains
- and is computationally very challenging:

Extremely high-resolution 3-D forward models are mandatory

Requires a supercomputing approach

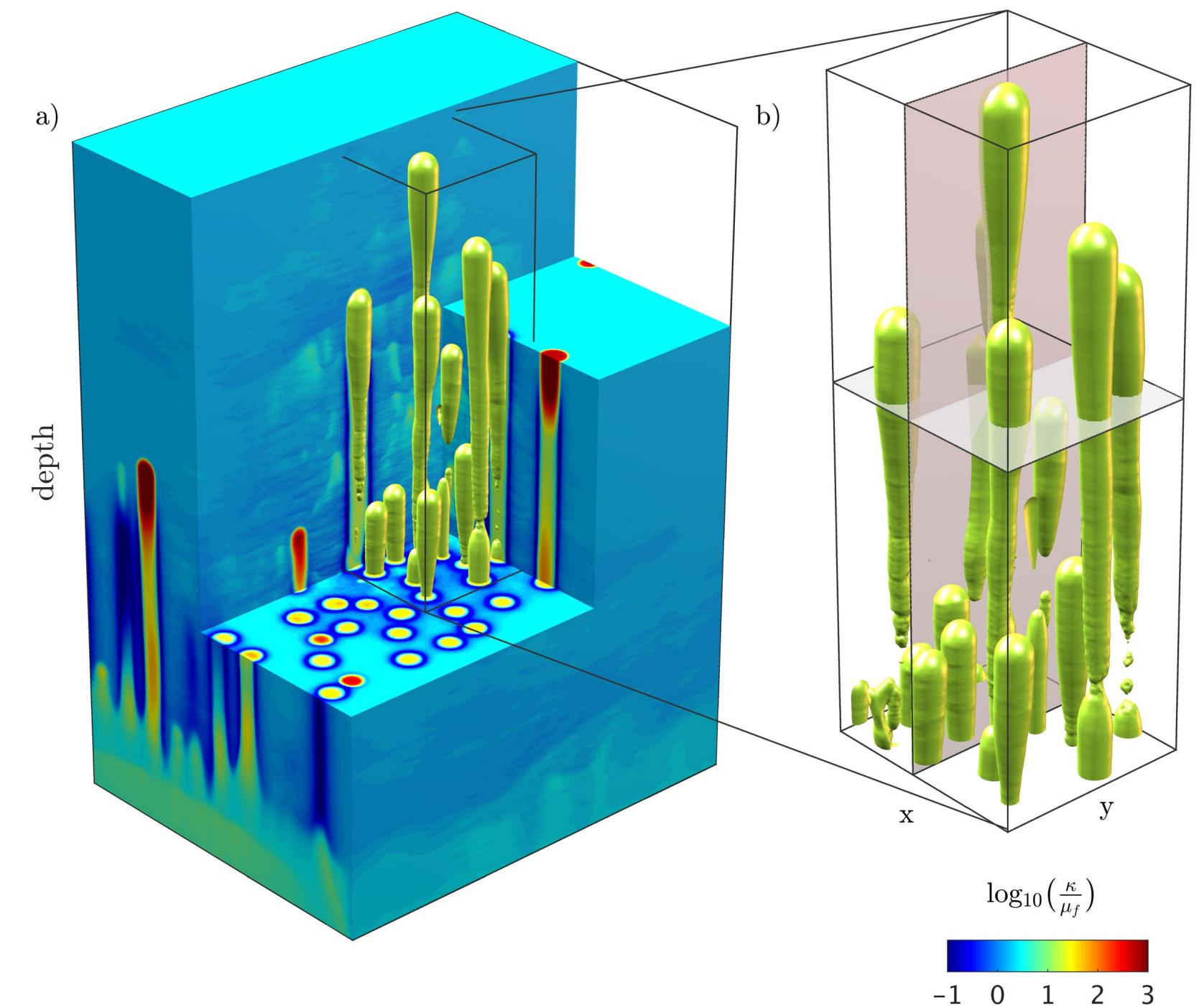




The movie making-of

Resolution:

- space: $1000 \times 1000 \times 2000$ grid points in 3-D
- time: 20'000 implicit time steps



Räss et al., 2018. *Nature Scientific Reports*

Numerical method & code

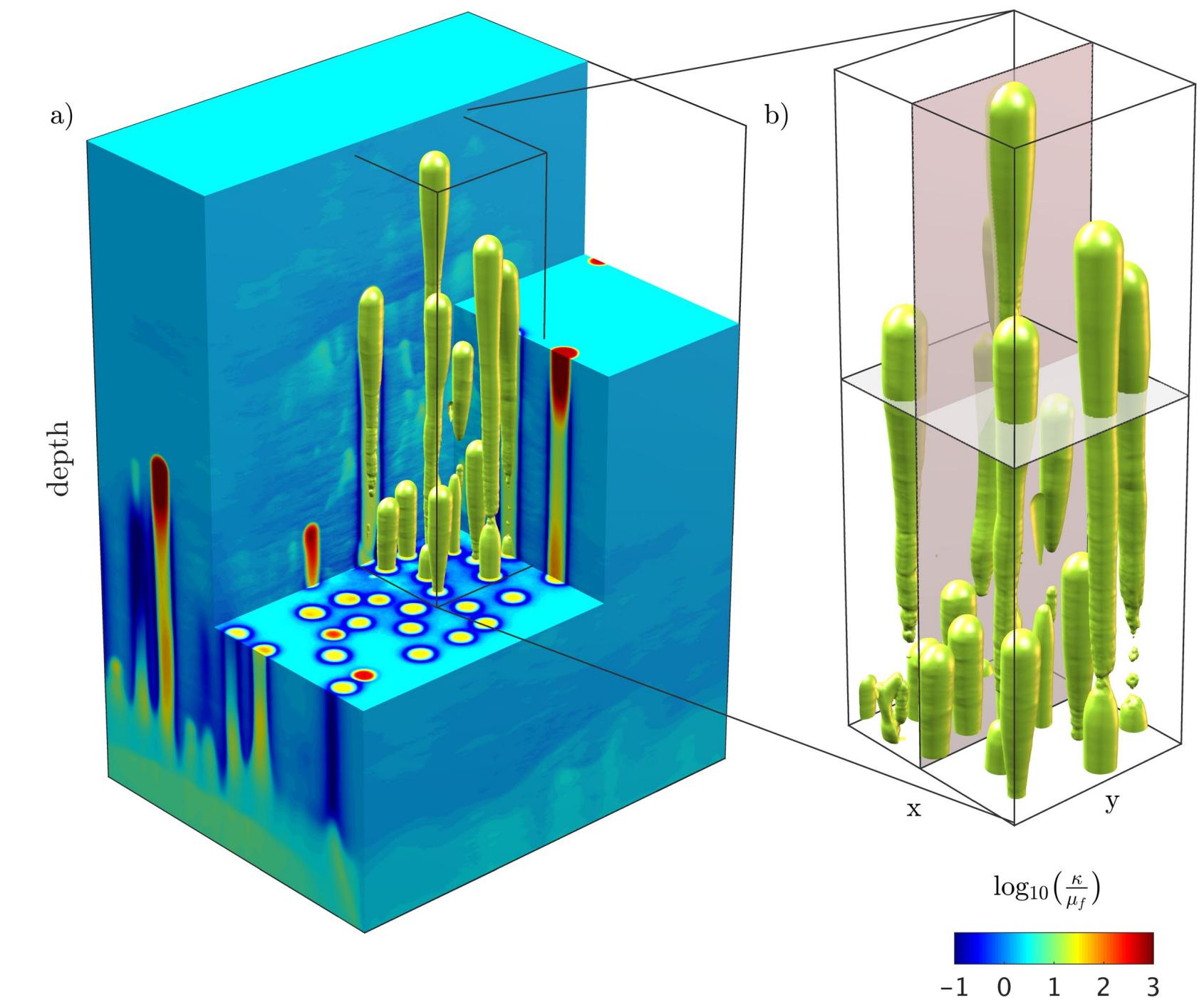
Numerical method:

- Finite-Difference stencils on Cartesian, staggered grid
- Iterative implicit time stepping: pseudo transient method

Code:

- CUDA C + MPI

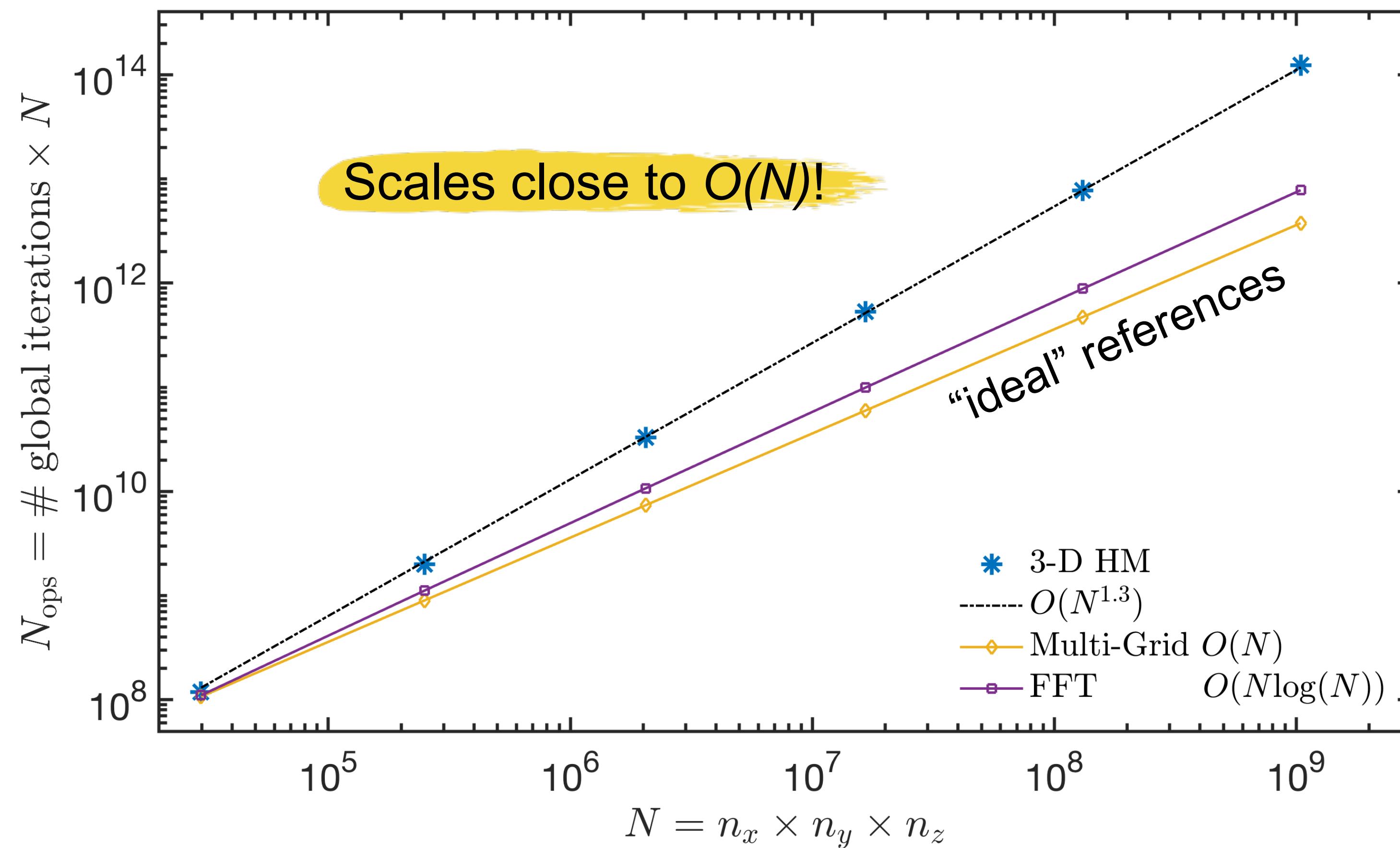
Runs close to hardware limit !



Räss et al., 2018. Nature Scientific Reports

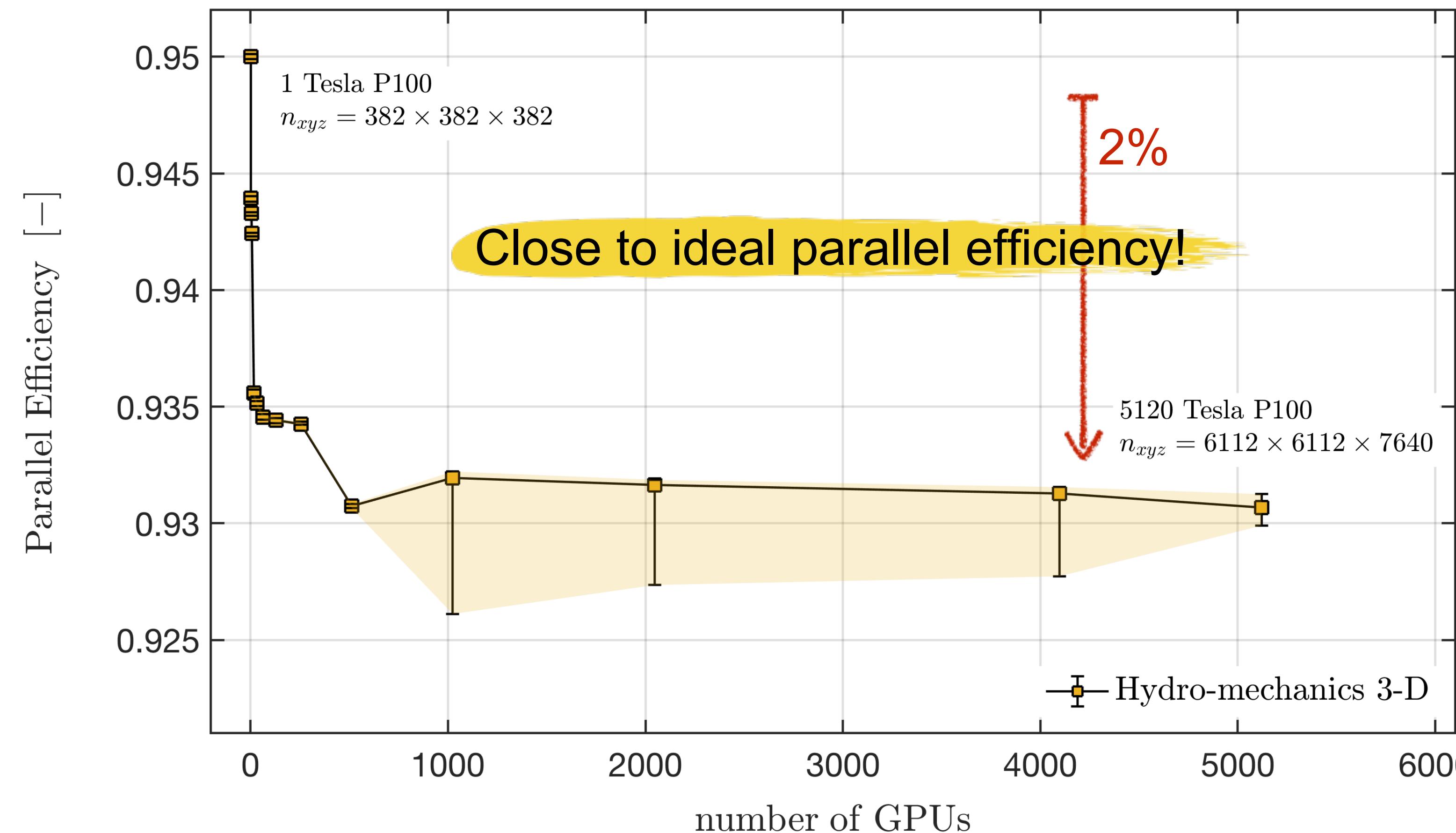
Pseudo-Transient solver

Scaling of our iterative GPU solver with numerical grid resolution N



Hide MPI communication

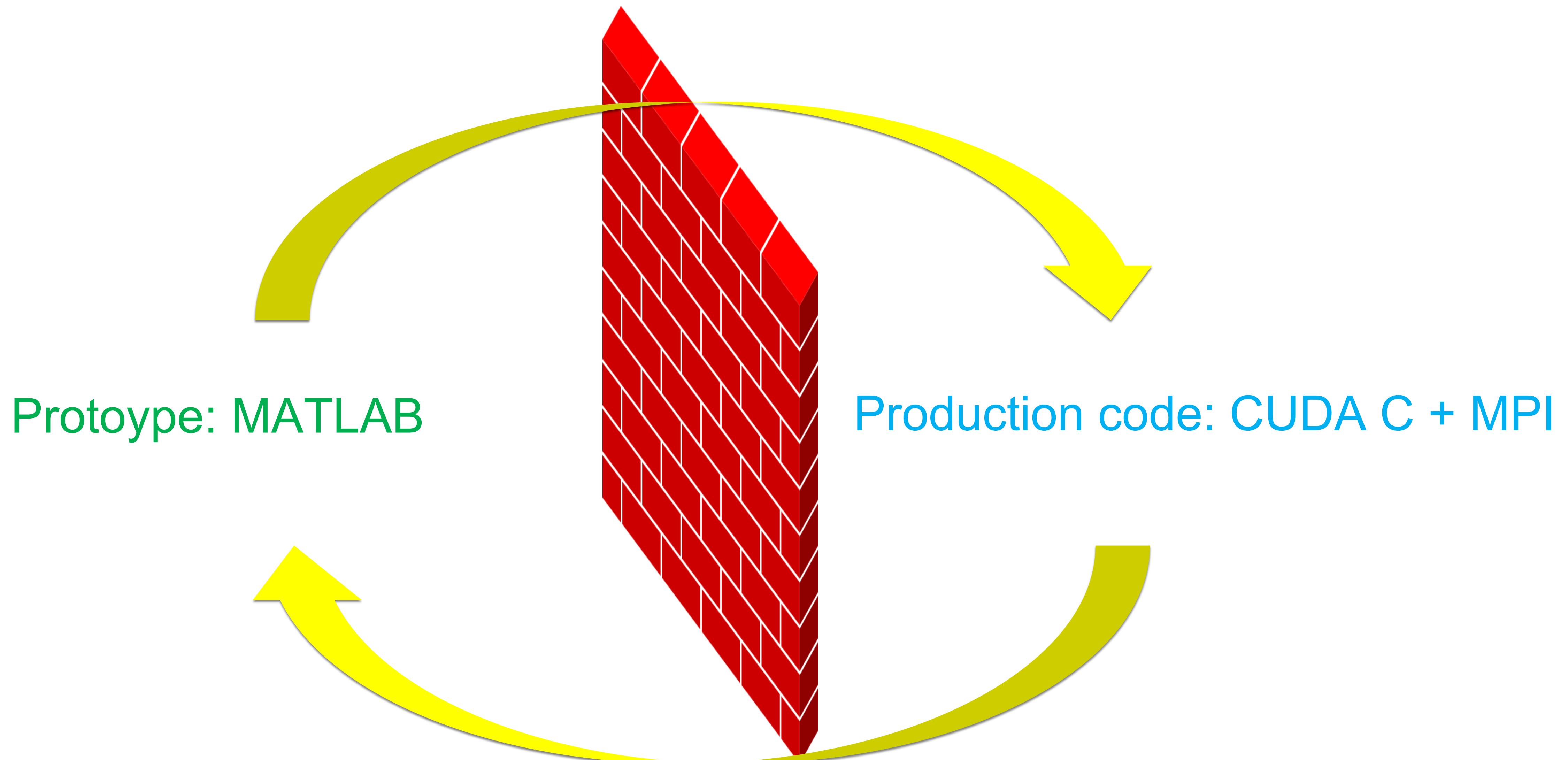
Weak scaling on Cray XC50 “Piz Daint” @ CSCS | 5120 Nvidia P100 GPUs



Agenda

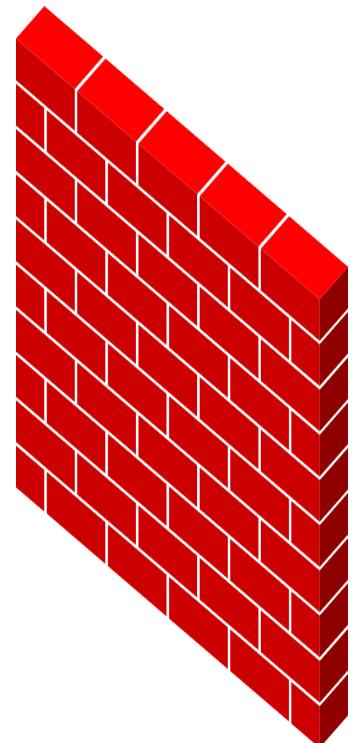
- The two language problem ✓
- Case study: the solver ✓
- Porting to Julia | Multi-GPU Programming in Julia
- Case study: results
- Case study: conclusions
- General conclusions
- Outlook & recommendations

The two language problem for our solver

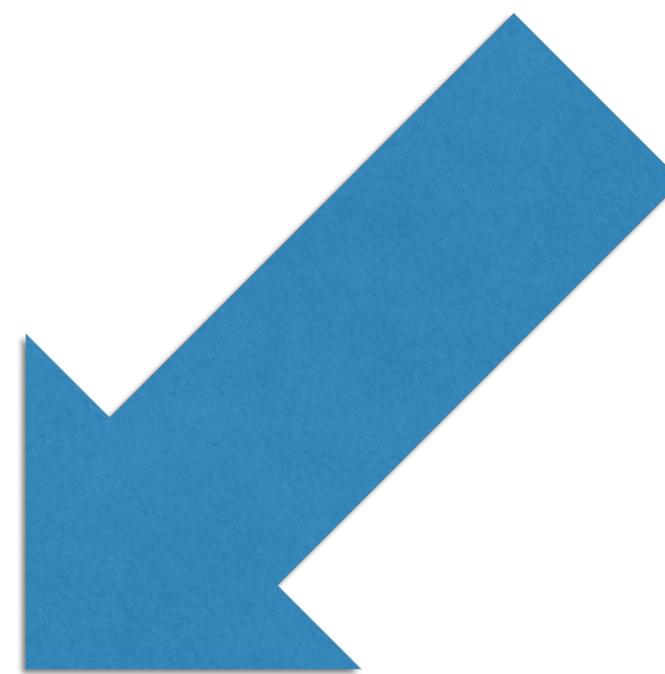


Porting approach

Prototype: MATLAB



Production code: CUDA C + MPI

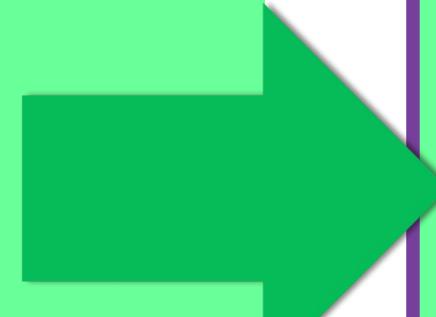


Code structure

```
# physical and numerical parameters  
  
# initializations of fields  
  
# time loop  
  
# nonlinear iterations  
  
# convergence check  
  
# postprocessing & visualization
```

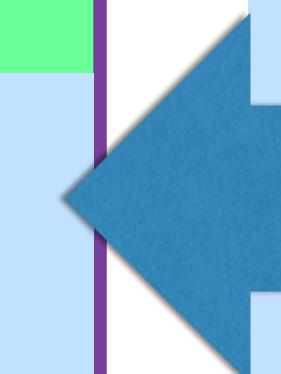
the **julia** code

Prototype
MATLAB



```
# physical and numerical parameters  
  
# initializations of fields  
  
# time loop  
  
# nonlinear iterations  
  
# convergence check  
  
# postprocessing & visualization
```

Production
code
CUDA C
(+ MPI)



the **julia** code

Prototype
MATLAB

```
# physical and numerical parameters
phi0 = 1e-2
nx   = 128
# initializations of fields

# time loop

# nonlinear iterations

# convergence check

# postprocessing & visualization
```

Production
code
CUDA C
(+ MPI)

the **julia** code

Prototype
MATLAB

```
# physical and numerical parameters
phi0 = 1e-2
nx   = 128
# initializations of fields
phi = phi0*cuones(nx,ny,nz)
# time loop

# nonlinear iterations

# convergence check

# postprocessing & visualization
```

Production
code
CUDA C
(+ MPI)

the **julia** code

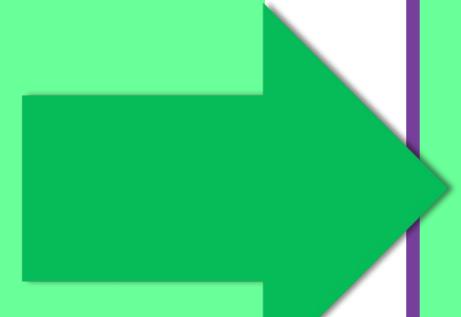
Prototype
MATLAB

```
# physical and numerical parameters
phi0 = 1e-2
nx   = 128
# initializations of fields
phi = phi0*cuones(nx,ny,nz)
# time loop
for it = 1:nt
    # nonlinear iterations
    # convergence check
    # postprocessing & visualization
end
```

Production
code
CUDA C
(+ MPI)

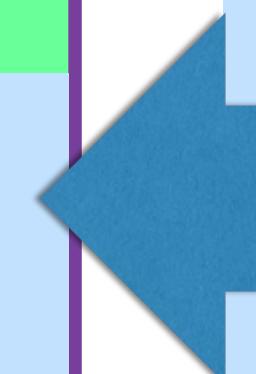
the **julia** code

Prototype
MATLAB



```
# physical and numerical parameters
phi0 = 1e-2
nx   = 128
# initializations of fields
phi = phi0*cuones(nx,ny,nz)
# time loop
for it = 1:nt
    # nonlinear iterations
    while (err > eps)
        @cuda ... compute_phi_etc...
        # convergence check
    end
    # postprocessing & visualization
end
```

Production
code
CUDA C
(+ MPI)



the **julia** code

CUDA kernel example (1-D heat diffusion)

```
function diffusion3D_kernel!(Te2, Te, Ci, lam, dt, dx)
    ix = (blockIdx().x-1) * blockDim().x + threadIdx().x # thread ID
    if ix>1 && ix<size(Te2,1)
        Te2[ix] = Te[ix] + dt*(Ci[ix]*(- (-lam*(Te[ix+1] - Te[ix]))/dx
                                         - (-lam*(Te[ix] - Te[ix-1]))/dx
                                         )/dx
                                         )
    end
    return nothing
end
```

the **julia** code

CUDA kernel example (1-D heat diffusion)

```
function diffusion3D_kernel1!(Te2, Te, ci, lam, dt, dx)
    ix = (blockIdx().x-1) * blockDim().x + threadIdx().x # thread ID
    if ix>1 && ix<size(Te2,1)
        Te2[ix] = Te[ix] + dt*(ci[ix]*(- (-lam*(Te[ix+1] - Te[ix-1])/dx)
                                            - (-lam*(Te[ix] - Te[ix-1])/dx))
                               )/dx
    end
    return nothing
end
```

Fourier's law of heat conduction:

$$\mathbf{q}_x = -\lambda \frac{\partial T}{\partial x}$$

the julia code

CUDA kernel example (1-D heat diffusion)

```
function diffusion3D_kernel1!(Te2, Te, Ci, lam, dt, dx)
    ix = (blockIdx().x-1) * blockDim().x + threadIdx().x # thread ID
    if ix>1 && ix<size(Te2,1)
        Te2[ix] = Te[ix] + dt*(Ci[ix]*(- (-lam*(Te[ix+1] - Te[ix-1])/dx)
                                            - (-lam*(Te[ix] - Te[ix-1])/dx))
                               )/dx
    end
    return nothing
end
```

The highlighted portion of the code corresponds to the following mathematical expression:

$$\frac{\partial T}{\partial x} = -\lambda \frac{\partial q_x}{\partial x}$$

Fourier's law of heat conduction:

$$q_x = -\lambda \frac{\partial T}{\partial x}$$

Conservation of energy:

$$\frac{\partial T}{\partial t} = \frac{1}{C_p} \left(-\frac{\partial q_x}{\partial x} \right)$$

the julia code

CUDA kernel example (1-D heat diffusion)

```
function diffusion3D_kernel1!(Te2, Te, ci, lam, dt, dx)
    ix = (blockIdx().x-1) * blockDim().x + threadIdx().x # thread ID
    if ix>1 && ix<size(Te2,1)
        Te2[ix] = Te[ix] + dt*(ci[ix]*(- (-lam*(Te[ix+1] - Te[ix-1])/dx)
                                            - (-lam*(Te[ix] - Te[ix-1])/dx))
                                         )/dx
    end
    return nothing
end
```

Fourier's law of heat conduction:

$$\mathbf{q}_x = -\lambda \frac{\partial T}{\partial x}$$

Conservation of energy:

$$\frac{\partial T}{\partial t} = \frac{1}{C_p} \left(- \frac{\partial q_x}{\partial x} \right)$$

Update of Temperature: $T_{t+1} = T_t + \frac{\partial T_t}{\partial t}$

the **julia** code

CUDA kernel example (1-D heat diffusion)

```
function diffusion3D_kernel!(Te2, Te, Ci, lam, dt, dx)
    ix = (blockIdx().x-1) * blockDim().x + threadIdx().x # thread ID
    if ix>1 && ix<size(Te2,1)
        Te2[ix] = Te[ix] + dt*(Ci[ix]*(- (-lam*(Te[ix+1] - Te[ix]))/dx
                                         - (-lam*(Te[ix] - Te[ix-1]))/dx
                                         )/dx
                                         )
    end
    return nothing
end
```

the code

Versus normal CPU function (1-D heat diffusion)

```
function diffusion3D_kernel1!(Te2, Te, ci, lam, dt, dx)
    for ix = 1:size(Te,1)
        if ix>1 && ix<size(Te2,1)
            Te2[ix] = Te[ix] + dt*(ci[ix]*(- (-lam*(Te[ix+1] - Te[ix]))/dx
                                              - (-lam*(Te[ix] - Te[ix-1]))/dx
                                              )/dx
                                              )
        end
    end
end
return nothing
end
```

the code

CUDA kernel example (1-D heat diffusion)

```
function diffusion3D_kernel!(Te2, Te, Ci, lam, dt, dx)
    ix = (blockIdx().x-1) * blockDim().x + threadIdx().x # thread ID
    if ix>1 && ix<size(Te2,1)
        Te2[ix] = Te[ix] + dt*(Ci[ix]*(- (-lam*(Te[ix+1] - Te[ix]))/dx
                                         - (-lam*(Te[ix] - Te[ix-1]))/dx
                                         )/dx
                                         )
    end
    return nothing
end
```

the **julia** code

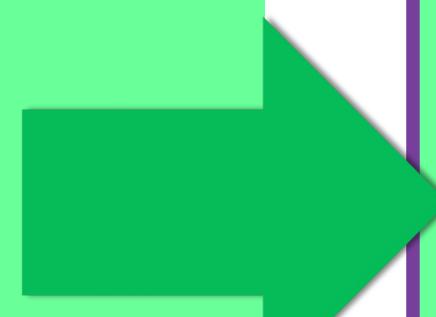
Prototype
MATLAB

```
# physical and numerical parameters
phi0 = 1e-2
nx   = 128
# initializations of fields
phi = phi0*cuones(nx,ny,nz)
# time loop
for it = 1:nt
    # nonlinear iterations
    while (err > eps)
        @cuda ... compute_phi_etc...
        # convergence check
    end
    # postprocessing & visualization
end
```

Production
code
CUDA C
(+ MPI)

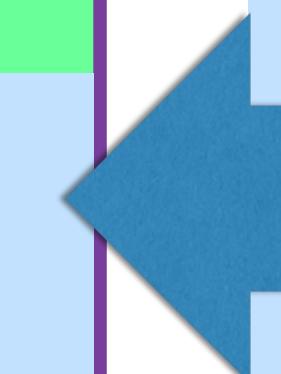
the **julia** code

Prototype
MATLAB



```
# physical and numerical parameters
phi0 = 1e-2
nx   = 128
# initializations of fields
phi = phi0*cuones(nx,ny,nz)
# time loop
for it = 1:nt
    # nonlinear iterations
    while (err > eps)
        @cuda ... compute_phi_etc...
        # convergence check
        if mod(iter,10)==0 err = ...; end
    end
    # postprocessing & visualization
end
```

Production
code
CUDA C
(+ MPI)



the **julia** code

Prototype
MATLAB

Production
code
CUDA C
(+ MPI)

```
# physical and numerical parameters
phi0 = 1e-2
nx   = 128
# initializations of fields
phi = phi0*cuones(nx,ny,nz)
# time loop
for it = 1:nt
    # nonlinear iterations
    while (err > eps)
        @cuda ... compute_phi_etc...
        # convergence check
        if mod(iter,10)==0 err = ...; end
    end
    # postprocessing & visualization
    heatmap(phi[:,ny/2,:])
end
```

the **julia** code

Prototype
MATLAB

```
# physical and numerical parameters
phi0 = 1e-2
nx   = 128
# initializations of fields
phi = phi0*cuones(nx,ny,nz)
# time loop
for it = 1:nt
    # nonlinear iterations
    # ...
    Computed automatically on the GPU!
    # ...
    # postprocessing & visualization
    phi_change .= phi .- phi_init
end
```

Production
code
CUDA C
(+ MPI)

Multi-GPU – implicit domain decomposition

We created a small Julia module for Cartesian **staggered** grids:

GG: implicit **G**lobal **G**rid

Multi-GPU – implicit domain decomposition

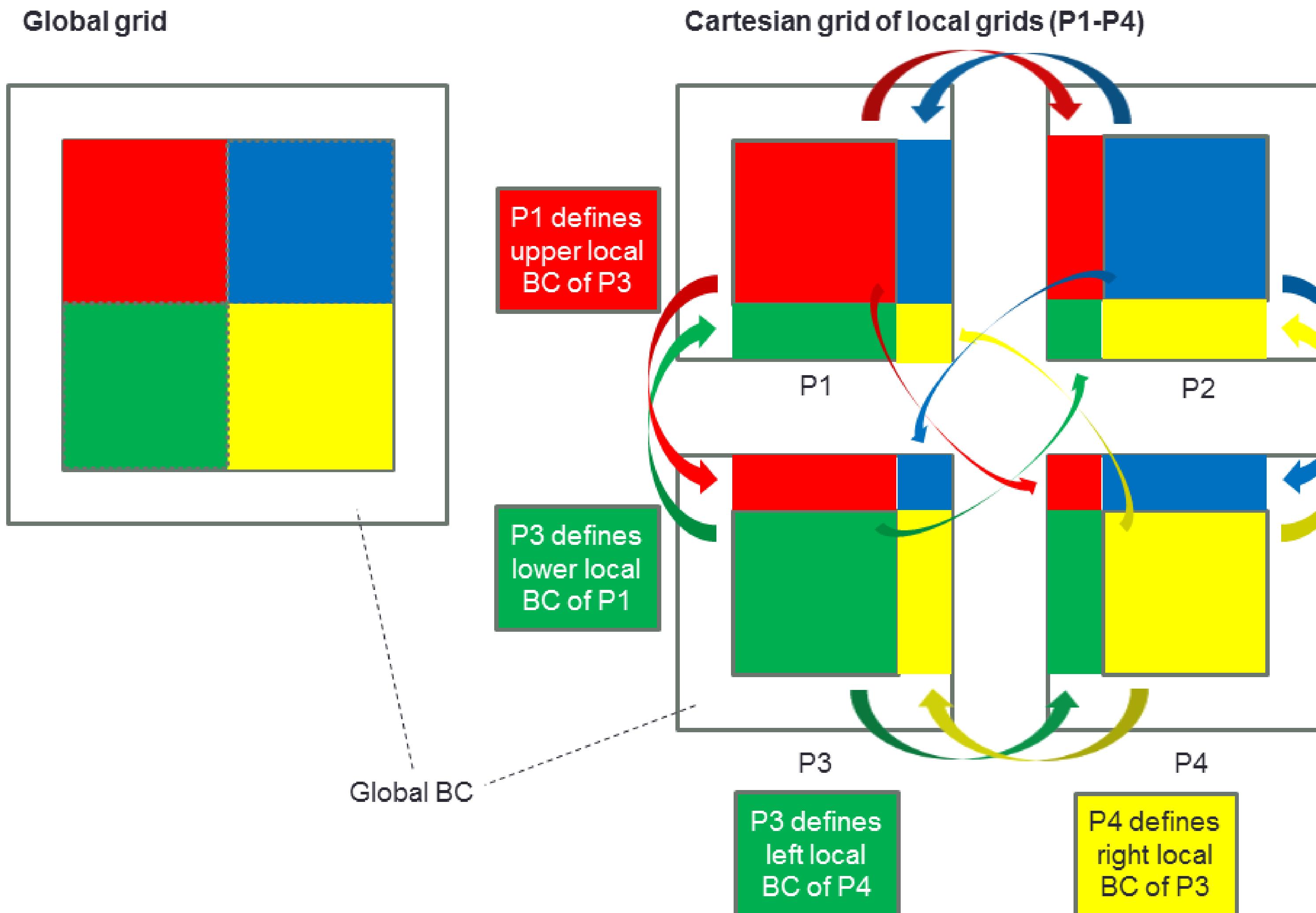
We created a small Julia module for Cartesian **staggered** grids:

GG: implicit **G**lobal **G**rid

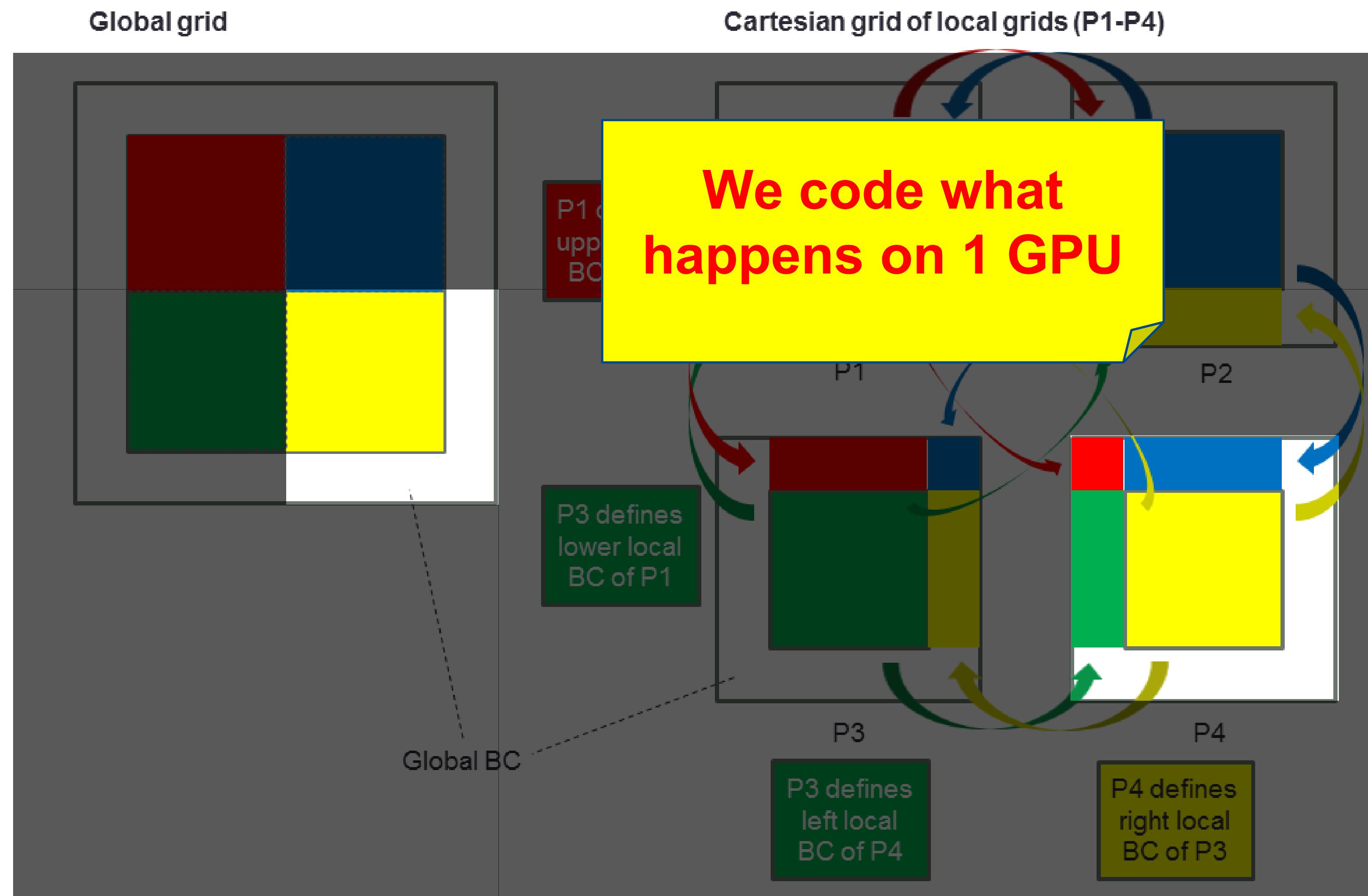
Uses MPI¹!

¹ <https://github.com/JuliaParallel/MPI.jl>

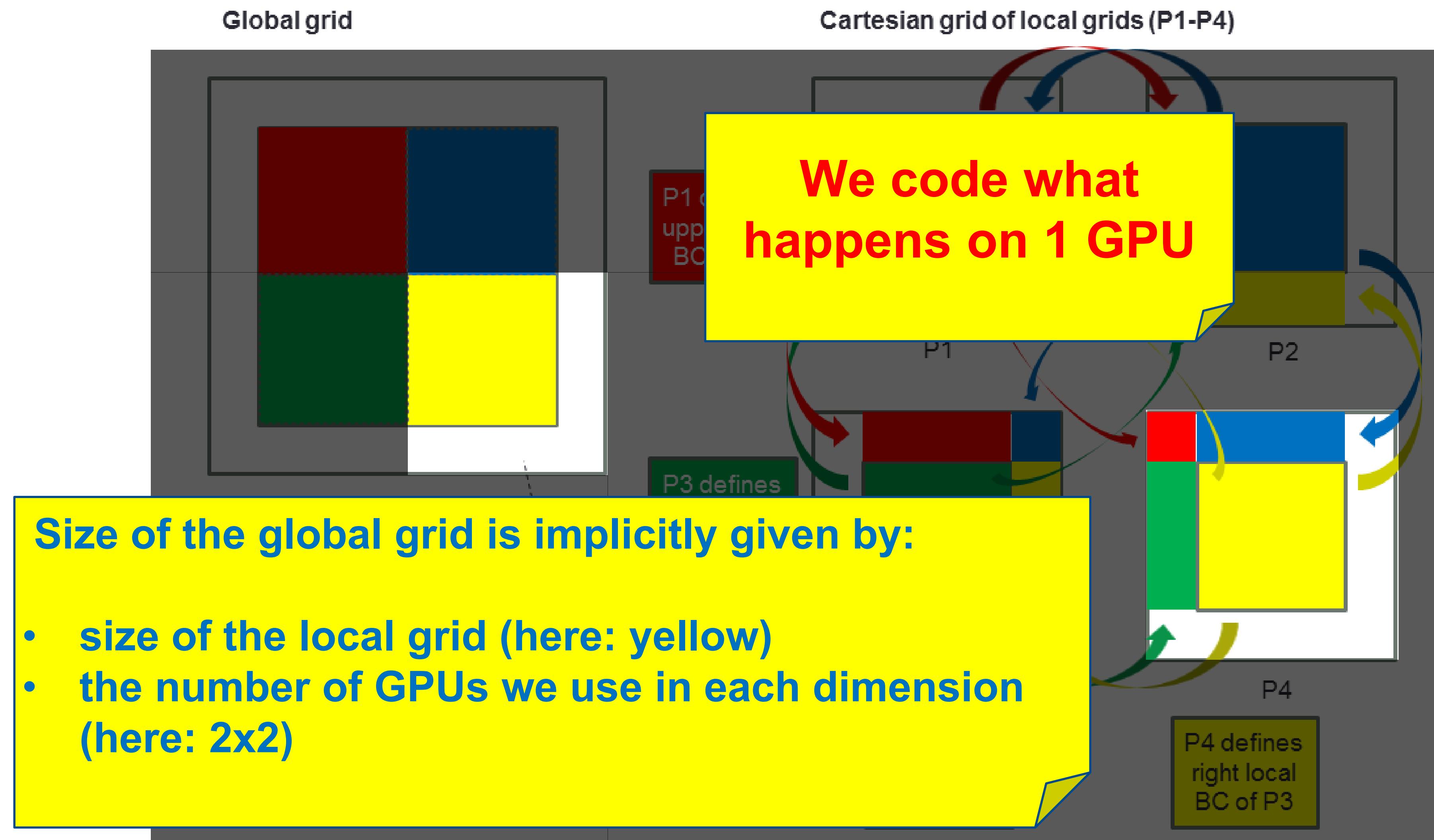
Multi-GPU – implicit domain decomposition?



Multi-GPU – implicit domain decomposition?



Multi-GPU – implicit domain decomposition?



GG: implicit Global Grid

Multi-GPU parallelization with only 3 “buttons”!

- `init_global_grid()`
- `update_halo!()`
- `finalize_global_grid()`

GG: implicit Global Grid

Multi-GPU parallelization with only 3 “buttons”!

- `init_global_grid()`
- `update_halo!()`
- `finalize_global_grid()`

...and functions providing **global coordinates (`x_g()`, `y_g()`, `z_g()`)**...

GG: implicit Global Grid

Multi-GPU parallelization with only 3 “buttons”!

- `init_global_grid()`
- `update_halo!()`
- `finalize_global_grid()`

Keeps the logic for Multi-GPU parallelization separate from your numerical algorithm!

=

No “IT noise” in your numerical algorithm!

...and functions providing **global coordinates (`x_g()`, `y_g()`, `z_g()`)**...

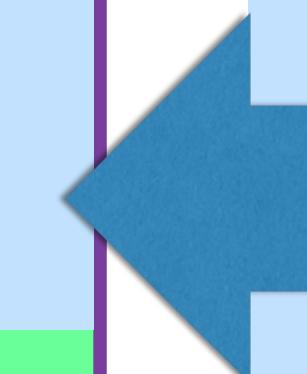
the **julia** code

Prototype
MATLAB



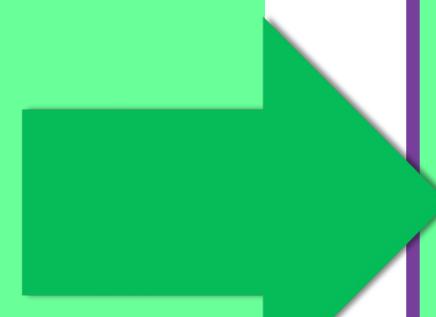
```
init_global_grid(nx,ny,nz)
# initializations of fields
phi = phi0*cuones(nx,ny,nz)
# time loop
for it = 1:nt
    # nonlinear iterations
    while (err > eps)
        @cuda ... compute_phi_etc(...)
        update_halo(phi,vx,vy,vz,...)
    # convergence check
    if mod(iter,10)==0 err = ...; end
    end
    # postprocessing & visualization
    ...
end
finalize_global_grid()
```

Production
code
CUDA C
(+ MPI)



the **julia** code

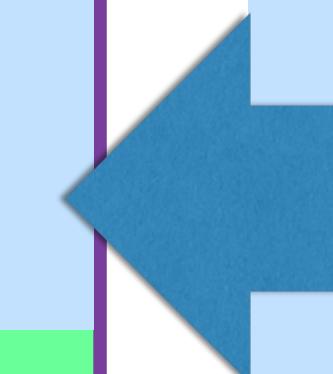
Prototype
MATLAB



```
init_global_grid(nx,ny,nz)
# initializations of fields
phi = phi0*cuones(nx,ny,nz)
# time loop
for it = 1:nt
    # nonlinear iterations
    while (err > eps)
        @cuda ... compute_phi_etc(...)
        update_halo(phi,vx,vy,vz,...)
    # convergence check
    if mod(iter,10)==0 err = ...; end
    end
    # postprocessing & visualization
    ...
end
finalize_global_grid()
```

+ some initializations using
x_g(), y_g(), z_g()

Production
code
CUDA C
(+ MPI)



A single code for single-thread CPU to Multi-GPU?

Would be fantastic for development and debugging...

...and decrease development and maintenance costs.

Solution (part 1)

```
USE_GPU = true

@static if USE_GPU
    using CuArrays, CUDAnative, CUDAdrv1
    myzeros(dims...) = cuzeros(dims...)
    myones(dims...) = cuones(dims...);
else
    myzeros(dims...) = zeros(dims...)
    myones(dims...) = ones(dims...)
end
```

```
macro kernel(cublocks, cuthreads, kernel)
    @static if USE_GPU
        esc(:(@cuda blocks=$cublocks threads=$cuthreads $kernel))
    else
        esc(:($kernel))
    end
end
```

@static: evaluated at parse time – no if statement in the final code!

¹ <https://github.com/JuliaGPU/>

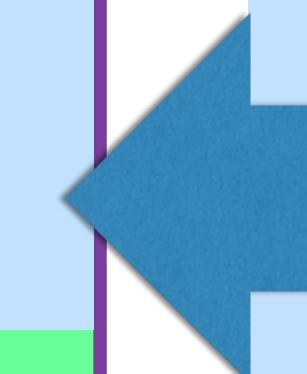
the **julia** code

Prototype
MATLAB



```
init_global_grid(nx,ny,nz)
# initializations of fields
phi = phi0*myones(nx,ny,nz)
# time loop
for it = 1:nt
    # nonlinear iterations
    while (err > eps)
        @kernel ... compute_phi_etc(...)
        update_halo(phi,vx,vy,vz,...)
    # convergence check
    if mod(iter,10)==0 err = ...; end
    end
    # postprocessing & visualization
    ...
end
finalize_global_grid()
```

Production
code
CUDA C
(+ MPI)



Solution (part 2)

```
macro threadids_or_loop(sizemax, block)
    @static if USE_GPU
        esc(quote
            ix = (blockIdx().x-1) * blockDim().x + threadIdx().x
            iy = (blockIdx().y-1) * blockDim().y + threadIdx().y
            iz = (blockIdx().z-1) * blockDim().z + threadIdx().z
            $block
        end)
    else
        esc(quote
            for iz=1:$sizemax[3]
                for iy=1:$sizemax[2], ix=1:$sizemax[1]
                    $block
                end
            end
        end)
    end
end
```

Solution (part 2)

```
macro threadids_or_loop(sizemax, block)
    @static if USE_GPU
        esc(quote
            ix = (blockIdx().x-1) * blockDim().x + threadIdx().x
            iy = (blockIdx().y-1) * blockDim().y + threadIdx().y
            iz = (blockIdx().z-1) * blockDim().z + threadIdx().z
            $block
        end)
    else
        esc(quote
            @threads for iz=1:$sizemax[3]
                for iy=1:$sizemax[2], ix=1:$sizemax[1]
                    $block
                end
            end
        end)
    end
end
```

**@threads: activates
multithreading if
JULIA_NUM_THREADS > 1**

the **julia** code

CUDA kernel example (1-D heat diffusion)

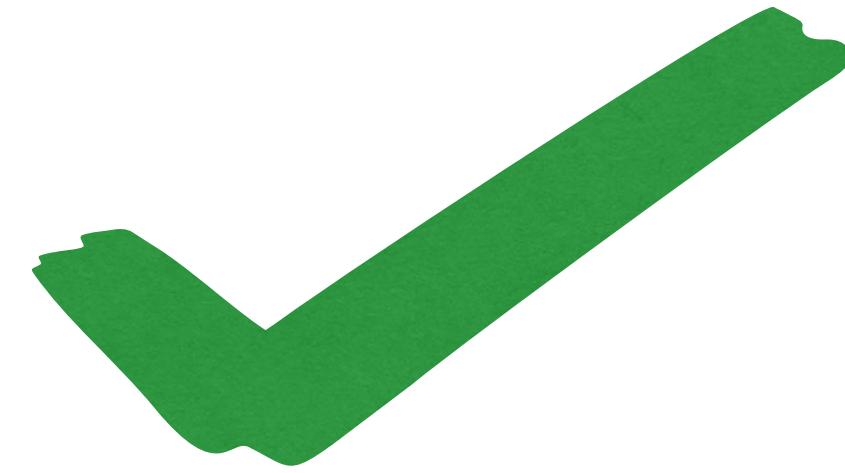
```
function diffusion3D_kernel!(Te2, Te, Ci, lam, dt, dx)
    ix = (blockIdx().x-1) * blockDim().x + threadIdx().x # thread ID
    if ix>1 && ix<size(Te2,1)
        Te2[ix] = Te[ix] + dt*(Ci[ix]*(- (-lam*(Te[ix+1] - Te[ix]))/dx
                                         - (-lam*(Te[ix] - Te[ix-1]))/dx
                                         )/dx
                                         )
    end
    return nothing
end
```

the code

CUDA kernel example (1-D heat diffusion)

```
function diffusion3D_kernel1!(Te2, Te, ci, lam, dt, dx)
    @threadids_or_loops size(Te2) begin # (assuming 1D version of macro)
        if ix > 1 && ix < size(Te2, 1)
            Te2[ix] = Te[ix] + dt * (ci[ix] * (- (-lam * (Te[ix+1] - Te[ix])) / dx
                                                - (-lam * (Te[ix] - Te[ix-1])) / dx
                                                ) / dx
                                                )
        end
    end
    return nothing
end
```

A single code for single-thread CPU to Multi-GPU?



GPU:

`USE_GPU = true | false`

CPU threads:

`export JULIA_NUM_THREADS = 12`

MPI:

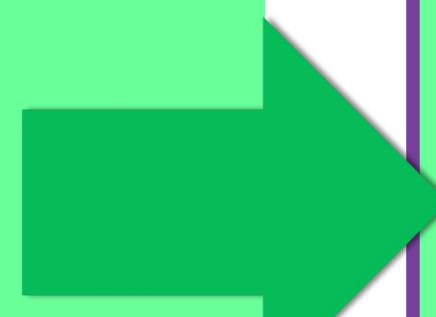
`srun -n 5120 ... julia ... PoroVP3D.jl`

Simple in-situ visualization?

Wouldn't it be great to see the results while computing on multiple GPUs?

the **julia** code

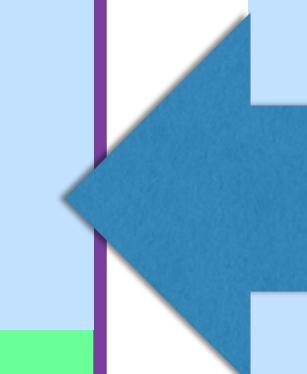
Prototype
MATLAB



```
init_global_grid(nx,ny,nz)
# initializations of fields
phi = phi0*myones(nx,ny,nz)
# time loop
for it = 1:nt
    # ... (time loop code)
    # Gather (reduced size) data on MPI rank 0 and visualize it!
    # postprocessing & visualization
    gather!(phi, phi_v)
    if me==0 heatmap(phi_v[:,ny/2,:]) end
end
finalize_global_grid()
```

**Gather (reduced size) data on
MPI rank 0 and visualize it!**

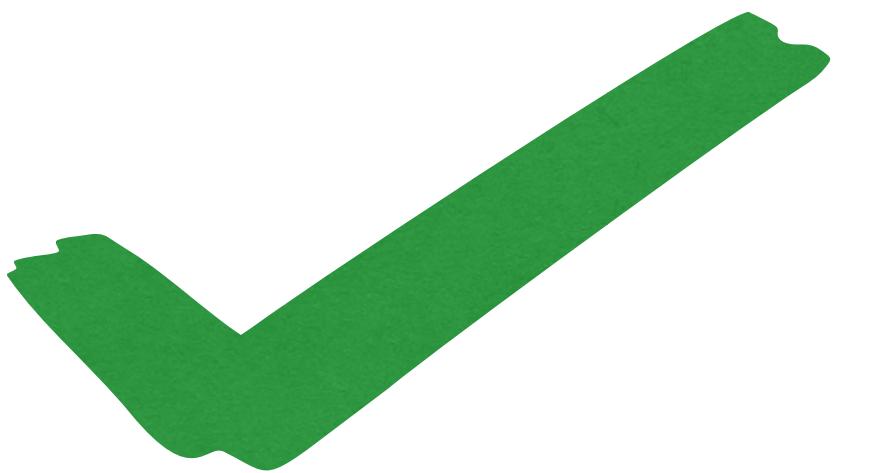
Production
code
CUDA C
(+ MPI)



Simple in-situ visualization?



Simple in-situ visualization?



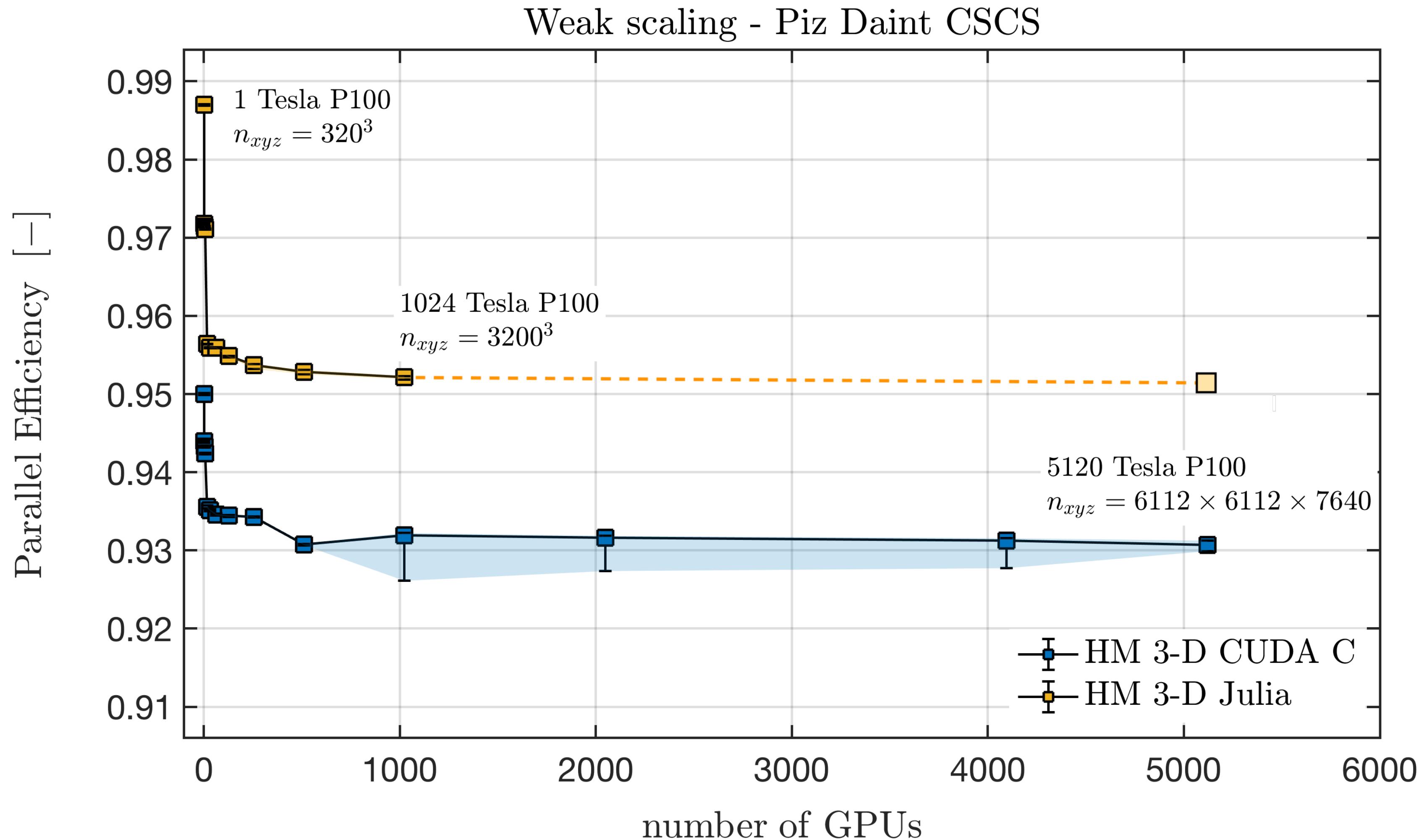
Agenda

- The two language problem ✓
- Case study: the solver ✓
- Porting to Julia | Multi-GPU Programming in Julia ✓
- Case study: results
- Case study: conclusions
- General conclusions
- Outlook & recommendations

Results

Single GPU performance:

93% of the the CUDA C code

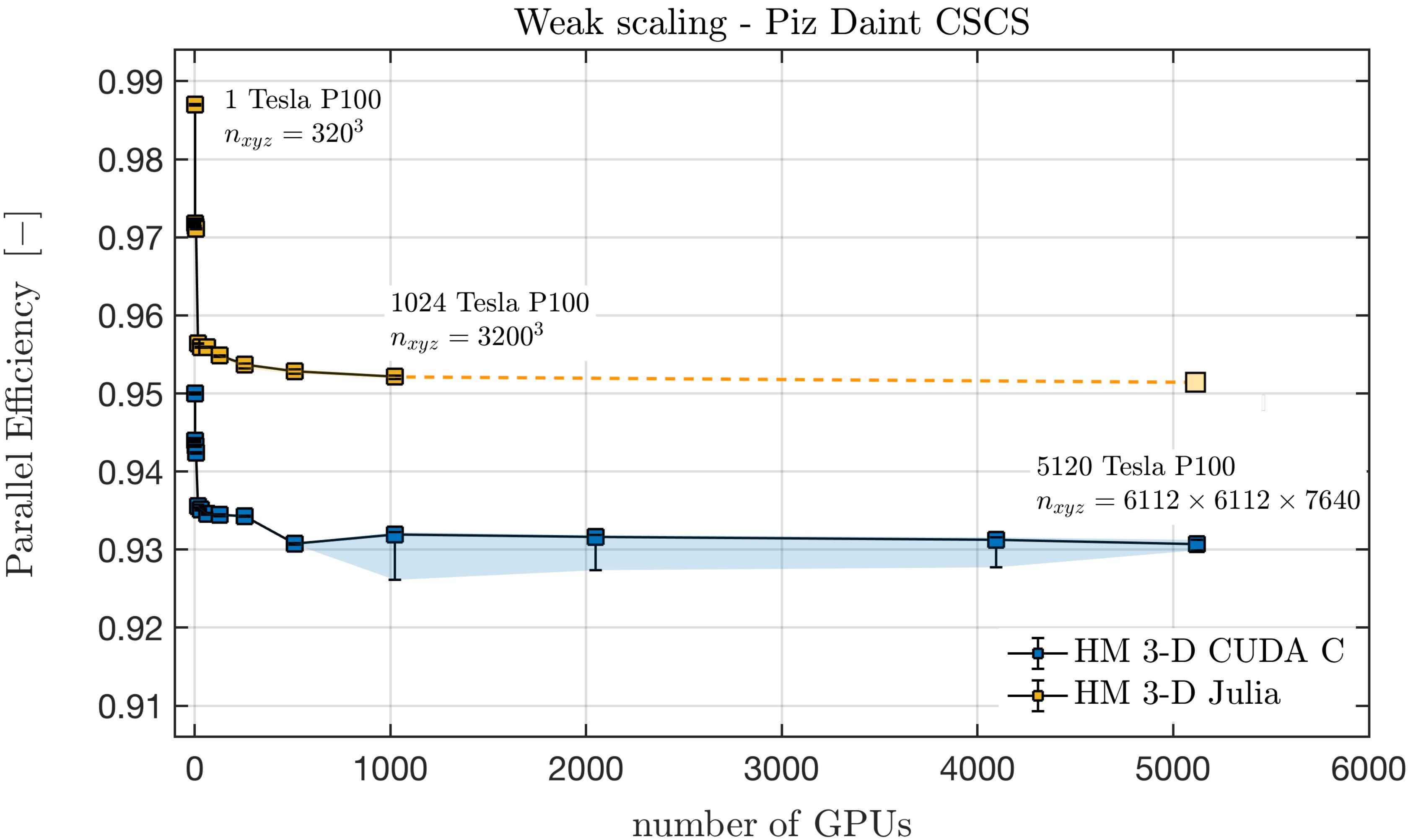


Results

Single GPU performance:

93% of the the CUDA C code

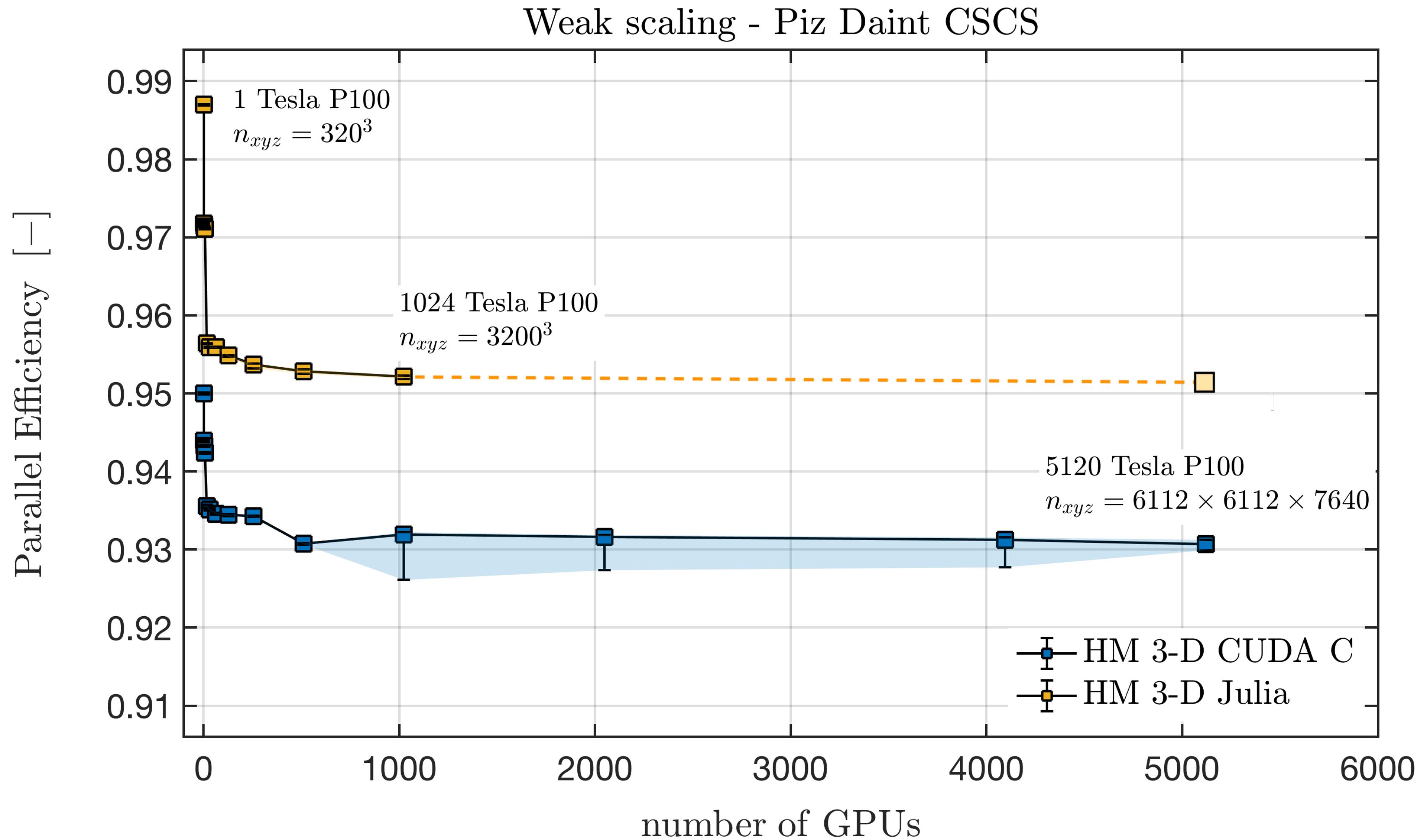
Difference to be analysed



Results

Single GPU performance:

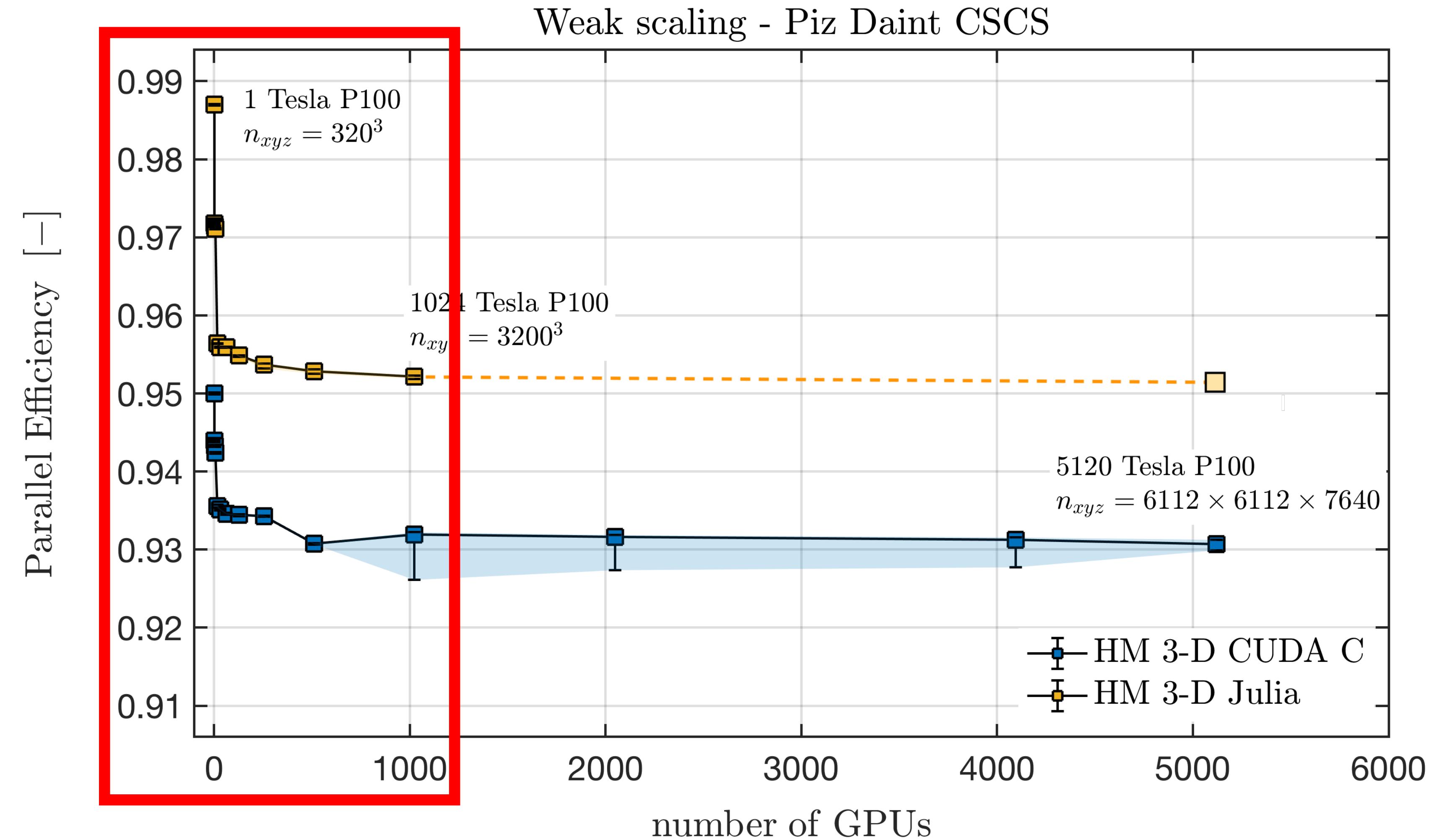
93% of the the CUDA C code



Results

Single GPU performance:

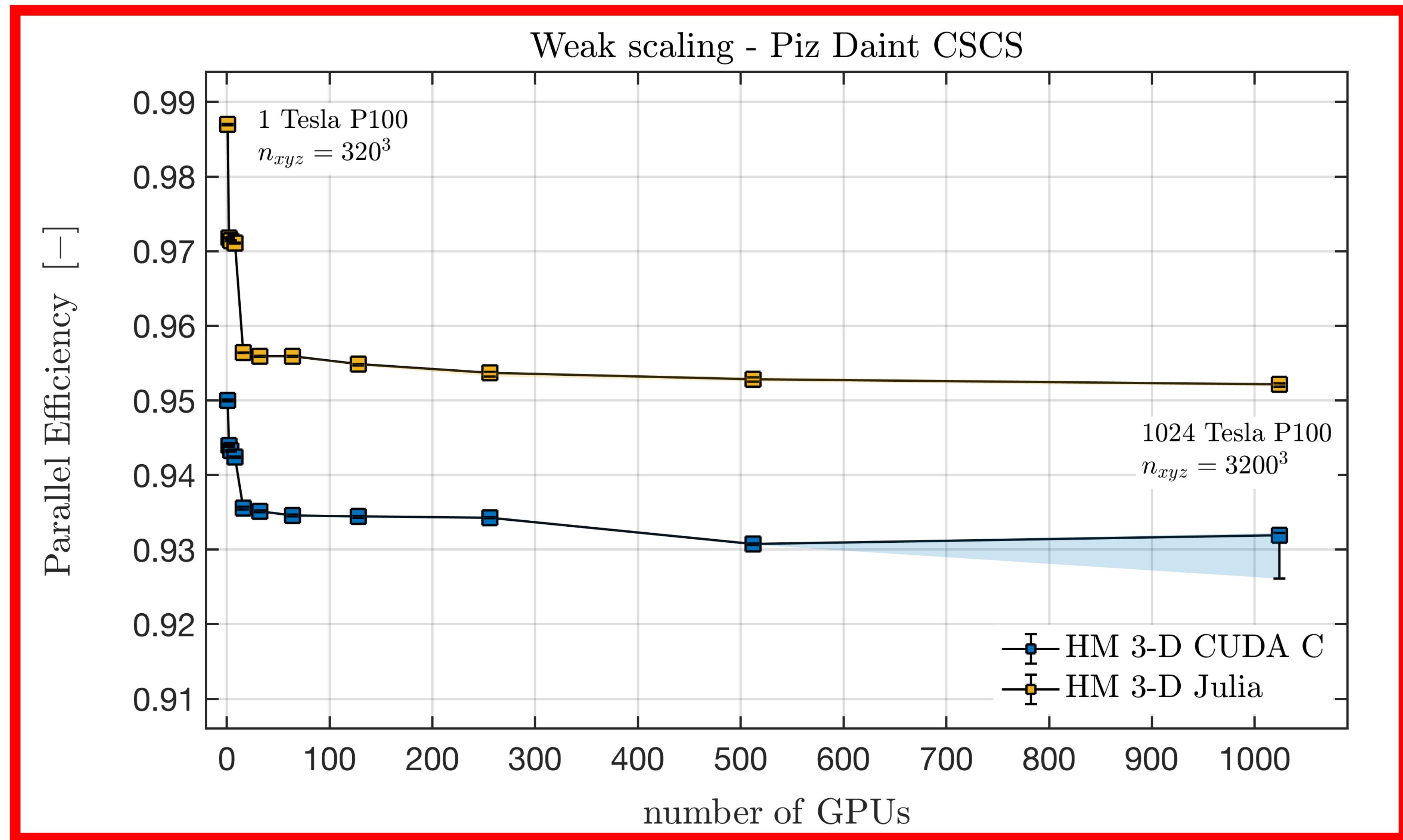
93% of the the CUDA C code



Results

Single GPU performance:

93% of the the CUDA C code



Agenda

- The two language problem ✓
- Case study: the solver ✓
- Porting to Julia | Multi-GPU Programming in Julia ✓
- Case study: results ✓
- Case study: conclusions
- General conclusions
- Outlook & recommendations

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.
- Only the performance critical part (kernels) comes from CUDA C.

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.
- Only the performance critical part (kernels) comes from CUDA C.

Kernels can be written also very nice with
macros/functions - not shown here...

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.
- Only the performance critical part (kernels) comes from CUDA C.
- Adding MPI requires only 3 “buttons” thanks to the GG module!

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.
- Only the performance critical part (kernels) comes from CUDA C.
- Adding MPI requires only 3 “buttons” thanks to the GG module!

**GG is reusable for other
staggered grid applications!**

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.
- Only the performance critical part (kernels) comes from CUDA C.
- Adding MPI requires only 3 “buttons” thanks to the GG module!

**GG is reusable for other
staggered grid applications!**

**So easy...
it can be used for teaching
Earth Science Bachelors!**

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.
- Only the performance critical part (kernels) comes from CUDA C.
- Adding MPI requires only 3 “buttons” thanks to the GG module!
- A single code for all: from single-thread CPU to Multi-GPU!

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.
- Only the performance critical part (kernels) comes from CUDA C.
- Adding MPI requires only 3 “buttons” thanks to the GG module!
- A single code for all: from single-thread CPU to Multi-GPU!
- Super simple in-situ visualization!

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.
- Only the performance critical part (kernels) comes from CUDA C.
- Adding MPI requires only 3 “buttons” thanks to the GG module!
- A single code for all: from single-thread CPU to Multi-GPU!
- Super simple in-situ visualization!
- Performance is nearly as good as with CUDA C after straightforward port to Julia (93%)!

Case study: conclusions

- 90% of the code comes from the prototype: all is nice and high-level.
- Only the performance critical part (kernels) comes from CUDA C.
- Adding MPI requires only 3 “buttons” thanks to the GG module!
- A single code for all: from single-thread CPU to Multi-GPU!
- Super simple in-situ visualization!
- Performance is nearly as good as with CUDA C after straightforward port to Julia (90%)!
- >95% parallel efficiency was shown on up to 1024 GPUs.

Case study: conclusions

We solved our two language problem!



Agenda

- The two language problem ✓
- Case study: the solver ✓
- Porting to Julia | Multi-GPU Programming in Julia ✓
- Case study: results ✓
- Case study: conclusions ✓
- General conclusions
- Outlook & recommendations

General conclusions

With Julia we can:

- write nice and high-level Multi-GPU code.
- write explicit CUDA kernels to get optimal performance in performance critical parts.
- use MPI efficiently without adding a lot of “IT noise” with an approach as taken with the GG module.
- write a single code for all: from single-thread CPU to Multi-GPU.
- do super simple in-situ visualization.
- get similar performance as with CUDA C.
- scale nearly ideally on the fastest Supercomputers in the world.

General conclusions

With Julia we can solve the two language problem!



Agenda

- The two language problem ✓
- Case study: the solver ✓
- Porting to Julia | Multi-GPU Programming in Julia ✓
- Case study: results ✓
- Case study: conclusions ✓
- General conclusions ✓
- Outlook & recommendations

Today's news: Julia in production on Piz Daint

Julia modules:

```
$> module load daint-gpu (or daint-mc)  
$> module load Julia  
$> module load JuliaExtensions
```

<- includes MPI + CUDA packages
<- Plots, PyCall & HDF5 packages
(more to come)

Available packages:

```
julia> versioninfo()
```

Note on the Julia package manager manager:

julia> Pkg.status() shows only the packages installed by the user by default,
but you can load the above packages normally, e.g.:

```
julia> using MPI
```

Start an interactive Julia session with GPU:

```
$> srun -C gpu --time=04:00:00 --pty bash  
$> julia
```

Solving other PDEs in Julia leveraging a GPU supercomputer

Other multi-GPU PDE solvers created with the presented numerical methods (during PhD):

Shallow water (20 462 x 10 354)



Shear heating (16 378 x 16 378)

Glacier Flow (20 462 x 10 354)



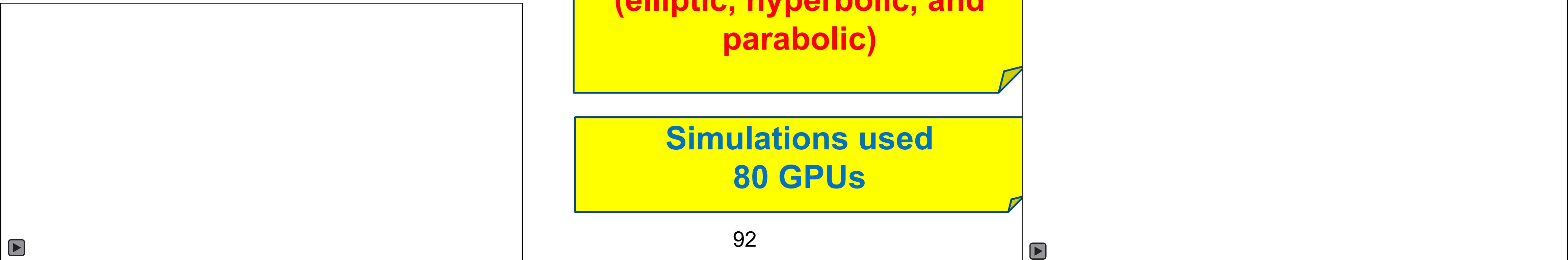
Reactive porosity waves (506 x 506 x 2042)



**Cover all types of PDEs!
(elliptic, hyperbolic, and
parabolic)**

**Simulations used
80 GPUs**

Thermal convection (32 762 x 16 378)



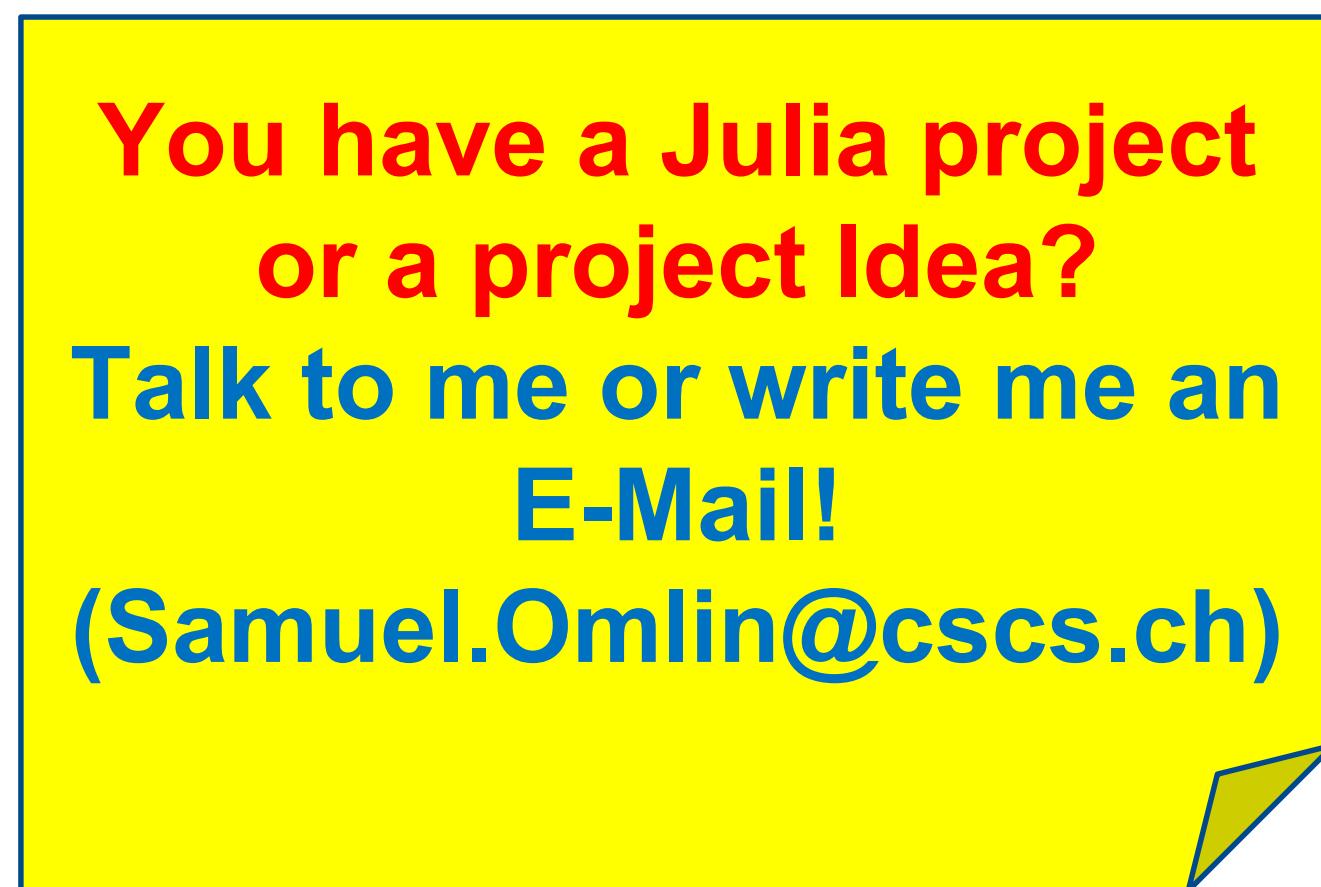
Solving other problems in Julia leveraging a GPU supercomputer

- **General advice:** for best intra and inter-GPU parallelization and scalability, **use local algorithms** (doing most the computations with data stored close in memory; e.g. stencil algorithms)!

Thank you!

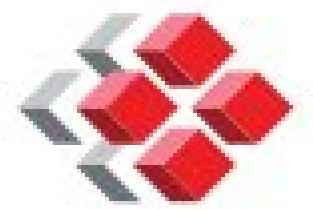


Thank you!



References

- Räss, L., Omlin, S., & Podladchikov, Y. Y. (2019). A Nonlinear Multi-Physics 3-D Solver: From CUDA C + MPI to Julia. PASC19 Conference, Zurich, Switzerland.
- Räss, L., Omlin, S., & Podladchikov, Y. Y. (2019). Porting a Massively Parallel Multi-GPU Application to Julia: a 3-D Nonlinear Multi-Physics Flow Solver. JuliaCon Conference, Baltimore, US.
- Räss, L., Simon, N. S. C., & Podladchikov, Y. Y. (2018). Spontaneous formation of fluid escape pipes from subsurface reservoirs. *Scientific Reports*, 8(1), 11116.
- Løseth, H., Wensaas, L., Arntsen, B., Hanken, N.-M., Basire, C., & Graue, K. (2011). 1000 m long gas blow-out pipes. *Marine and Petroleum Geology*, 28(5), 1047–1060.
- Plaza-Faverola, A., Bünz, S., & Mienert, J. (2011). Repeated fluid expulsion through sub-seabed chimneys offshore Norway in response to glacial cycles. *Earth and Planetary Science Letters*, 305(3–4), 297–308.
- Frankel, S. P. (1950). Convergence rates of iterative treatments equations of partial differential. Mathematical Tables and Other Aids to Computation, 4(30), 65–75.
- Besard, T., Foket, C., & De Sutter B. (2019). Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 30(4), 827-841



CSCS

Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Thank you for your kind attention