PDEs meet GPUs

Using Julia to tackle the hardware limit

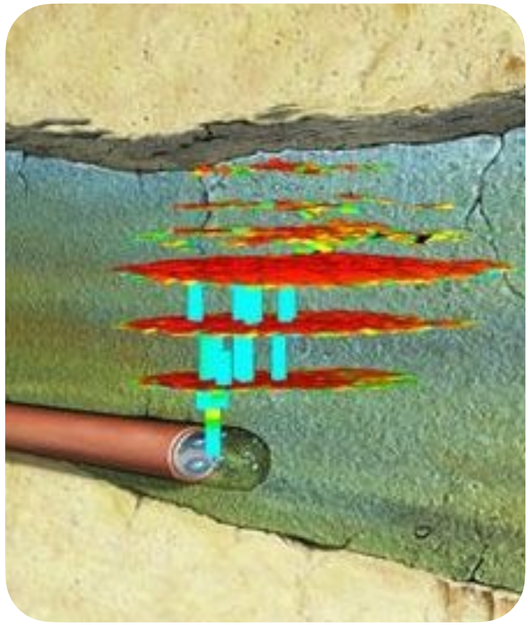
Unil.

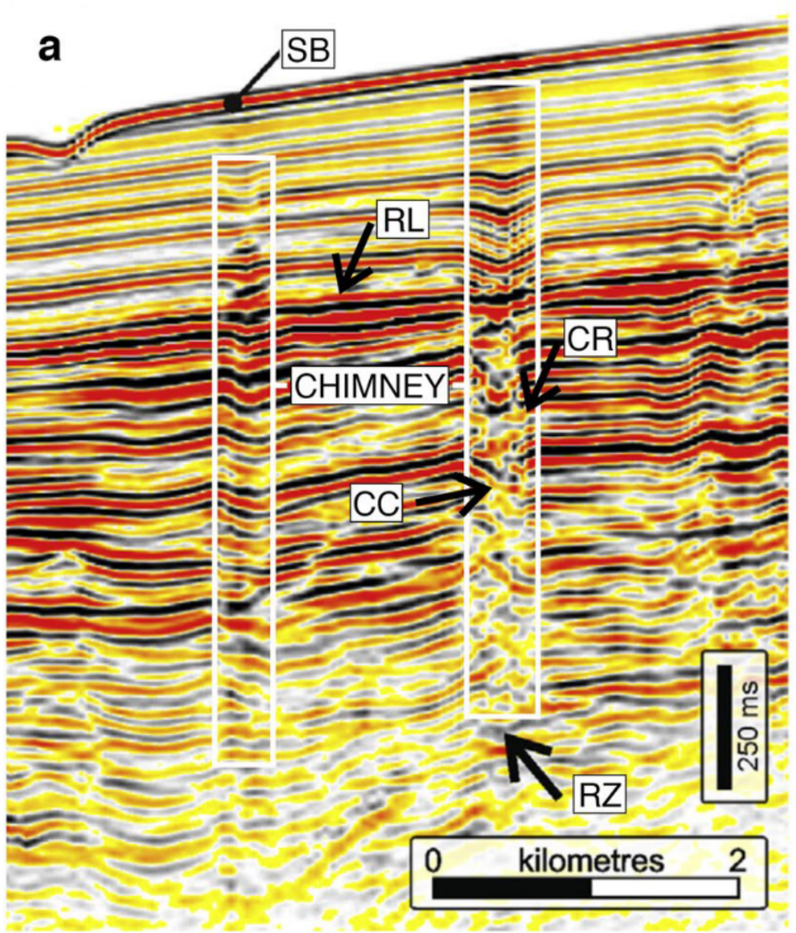
Ludovic Räss 27.03.2026

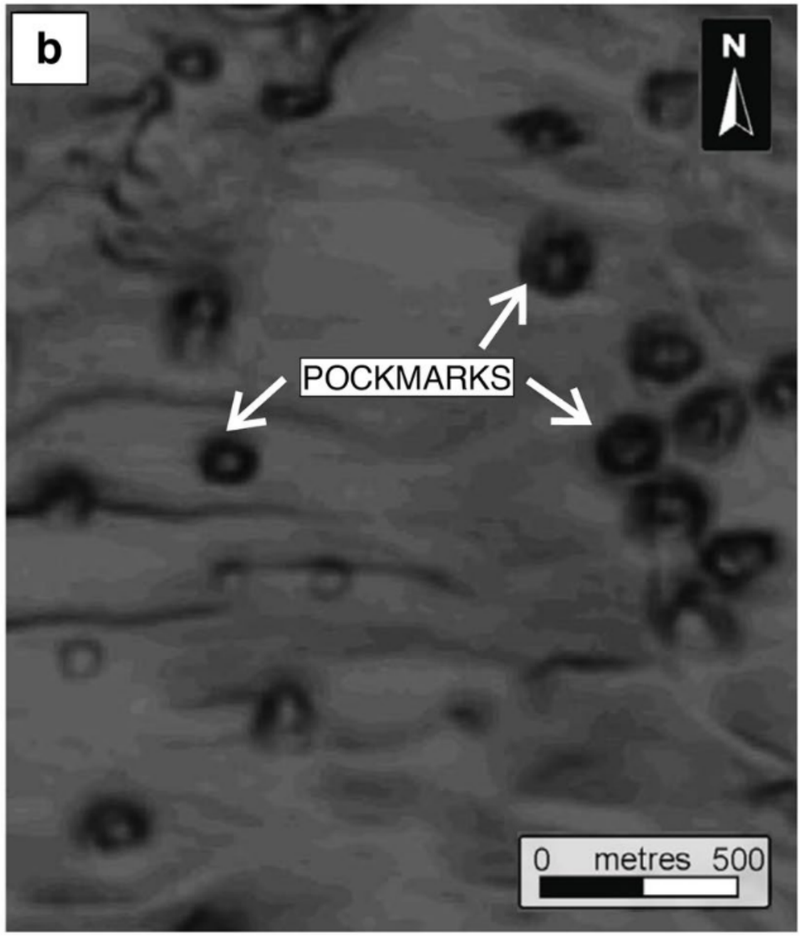
julia4PDEs 2026

GPU computing

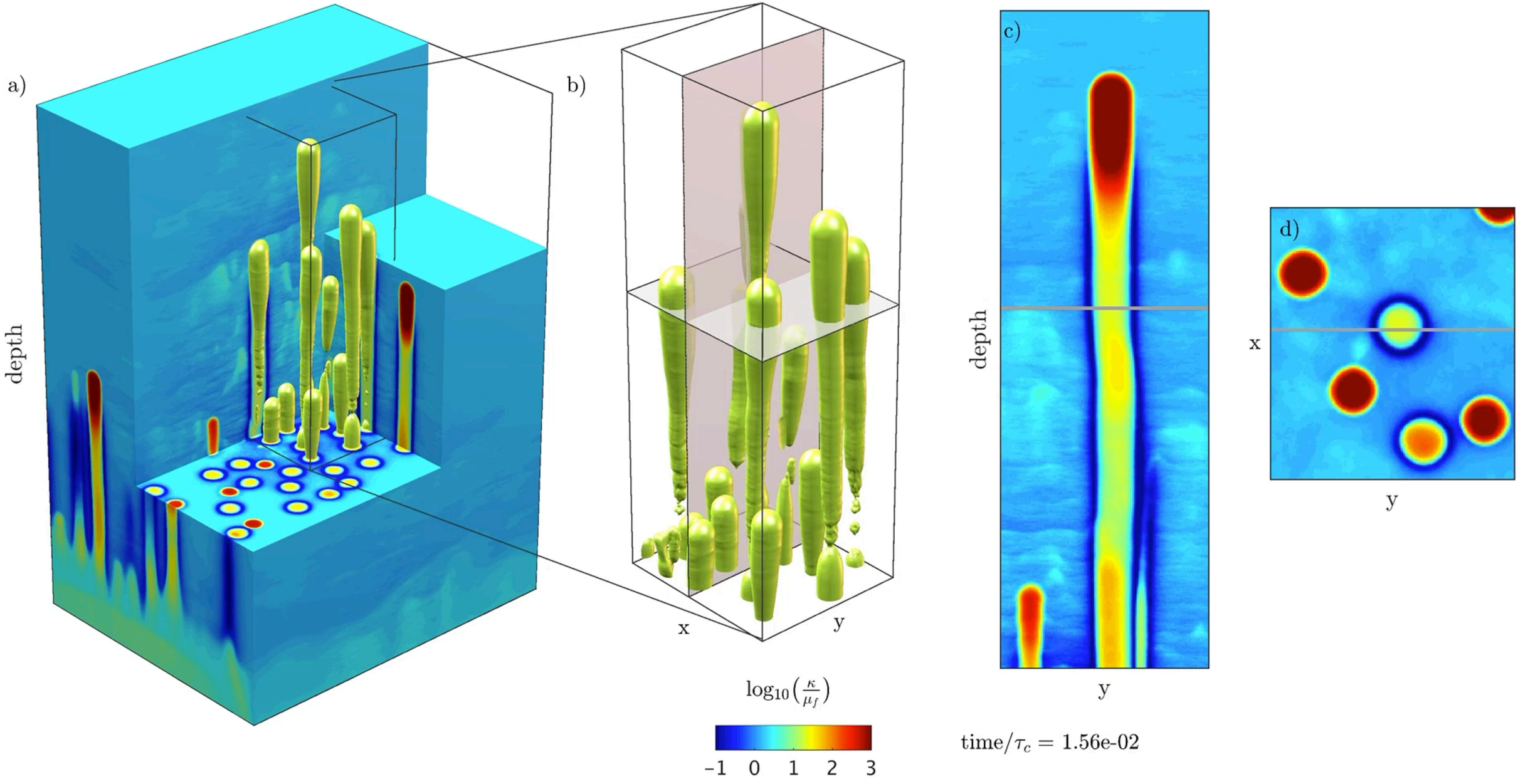
In earth sciences







http://www.nature.com/articles/s41598-018-29485-5

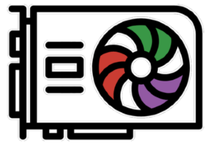


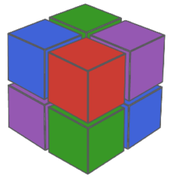
http://www.nature.com/articles/s41598-018-29485-5

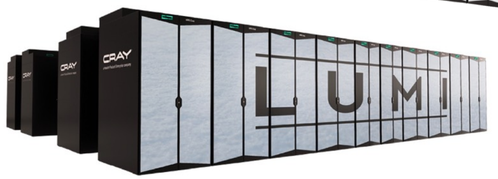
Agenda

PDEs meet GPUs

1. **Performance limiters**
   Hardware-aware methods and discretisations
2. **GPUs and Julia**
   Portability, performance and backend abstraction
3. **GPUs and Julia in HPC**
   Scaling from laptops to supercomputers





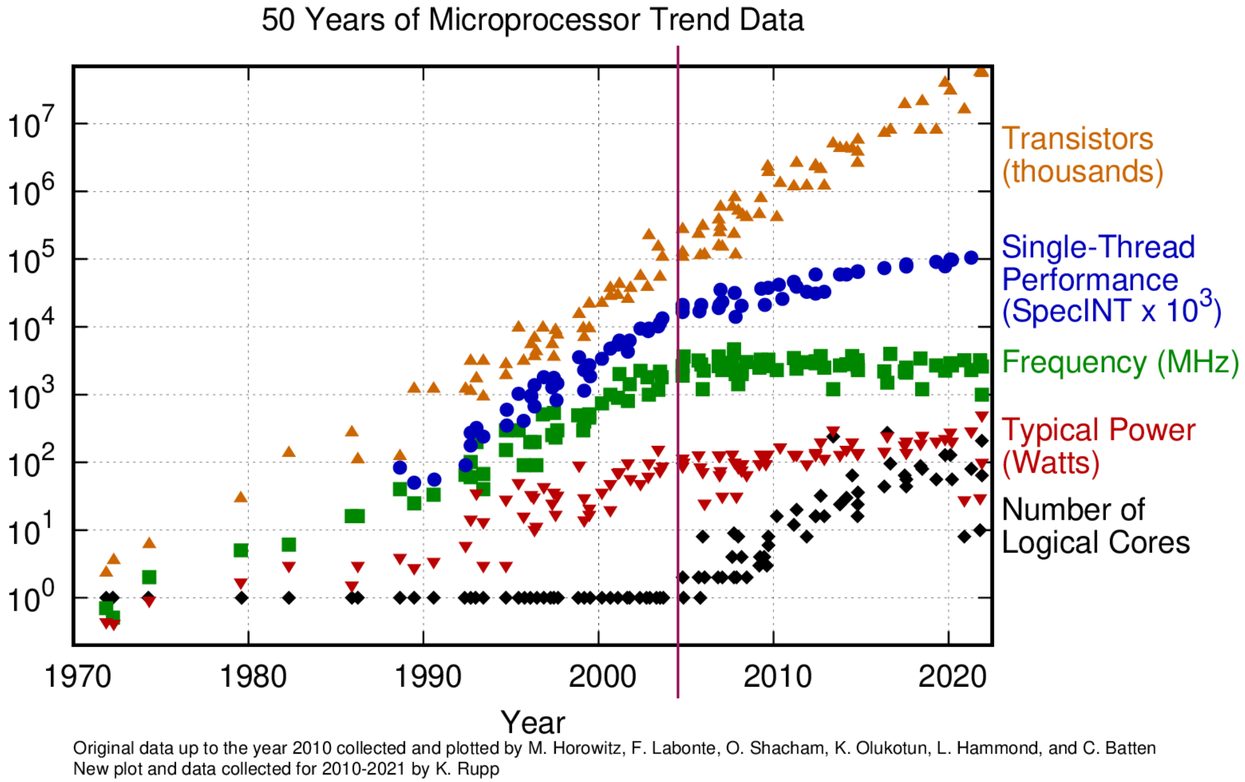


Performance limiters

Parallel computing

The memory wall

- Hitting the memory wall in 2004.
- Recent CPUs and GPUs have many cores.
- They use their parallelism to hide latency (i.e. overlapping execution times).
- Multi-core CPUs and GPUs share similar challenges.



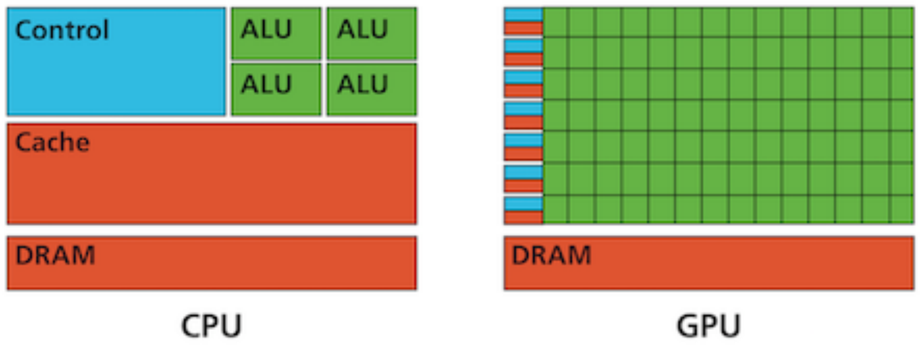
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

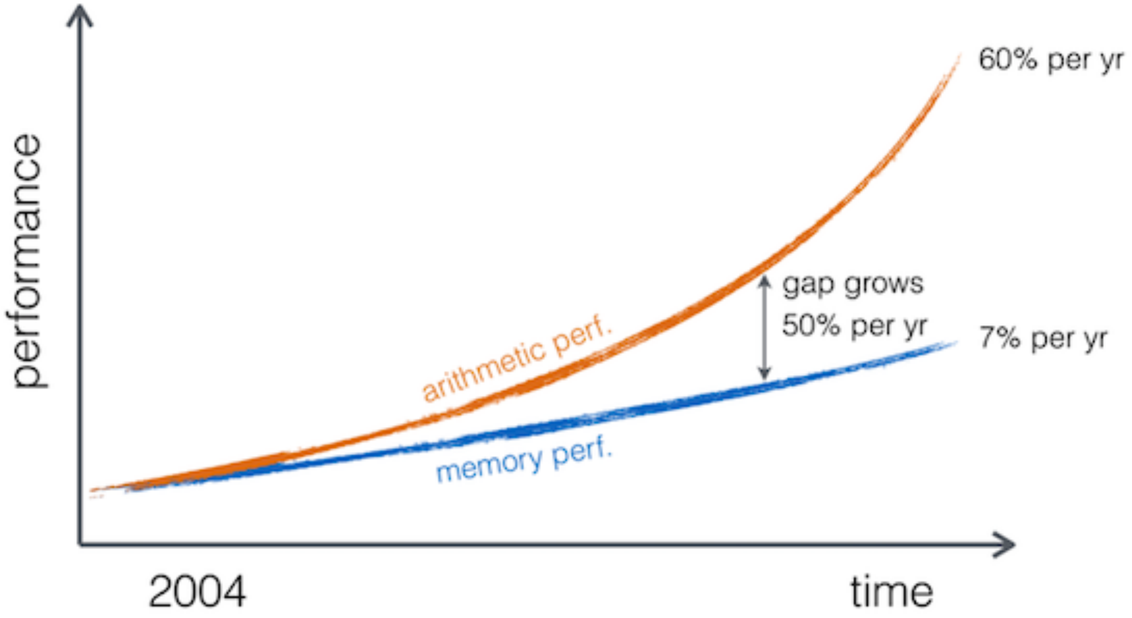
https://github.com/karlrupp/microprocessor-trend-data

GPU vs CPUs

Program hardware accelerators

- GPUs are massively parallel devices
- SIMD machine (programmed using threads - SPMD)





Further increases the FLOPs vs Bytes gap

Hardware limiters

CPU and GPU capabilities

- Recent GPUs and CPUs are memory bound (can do much more compute vs accessing numbers from RAM).
- Imbalance: we can do >50 FLOPs per number accessed from main memory.

Imbalance:

$$\frac{\text{Computation peak performance [TFLOP/s]}}{\text{Memory access peak performance [TB/s]}} \times \text{size of a number [Bytes]}$$

Device	TFLOP/s (FP64)	Memory BW TB/s	Imbalance
Nvidia GH200	34	4	34/4 x 8 = 68
Nvidia A100	9.7	1.55	9.7 / 1.55 x 8 = 50
AMD EPYC 7282 “Rome” 16 cores	0.7	0.085	0.7 / 0.085 x 8 = 65

FLOPs are for free in memory bound regimes

Scientific applications

Memory bandwidth limited

- Most algorithms require only a few FLOPs ...
- Compared to the amount of bytes accessed from main memory.
- FLOPs metric is not the most adequate for reporting application performance.

First derivative example $\partial A/\partial x$:

The “cost” of an isolated evaluation of a finite-difference first derivative (e.g. computing a flux q):

$$q = -D\frac{\partial A}{\partial x}$$

Which in discrete form reads

```
q[ix] = -D*(A[ix+1] - A[ix])/dx
```

1 read + 1 write => 2 x 8 = 16 bytes transferred

1 (fused) muladd => 1 (or 2) FLOPs

assuming $D, \partial x$ are scalars and q, A are Float64 arrays read from main memory

Performance metric

The effective memory throughput

- Need a memory throughput-based performance evaluation metric: T_{eff} [GB/s].
- The upper-bound of T_{eff} is T_{peak}
- Defining T_{eff} we assume:
  - We evaluate an iterative stencil solver.
  - The problem size is much larger then the cache size.

The effective memory access A_{eff} [GB] is the sum of:

- Twice the memory footprint of unknown fields D_u (that depend on their own history)
- Known fields D_k that do not change during iterations

$$A_{\text{eff}} = 2D_u + D_k$$

The effective memory access divided by the time per iteration t_{it} [sec] defines the effective memory throughput [GB/s]:

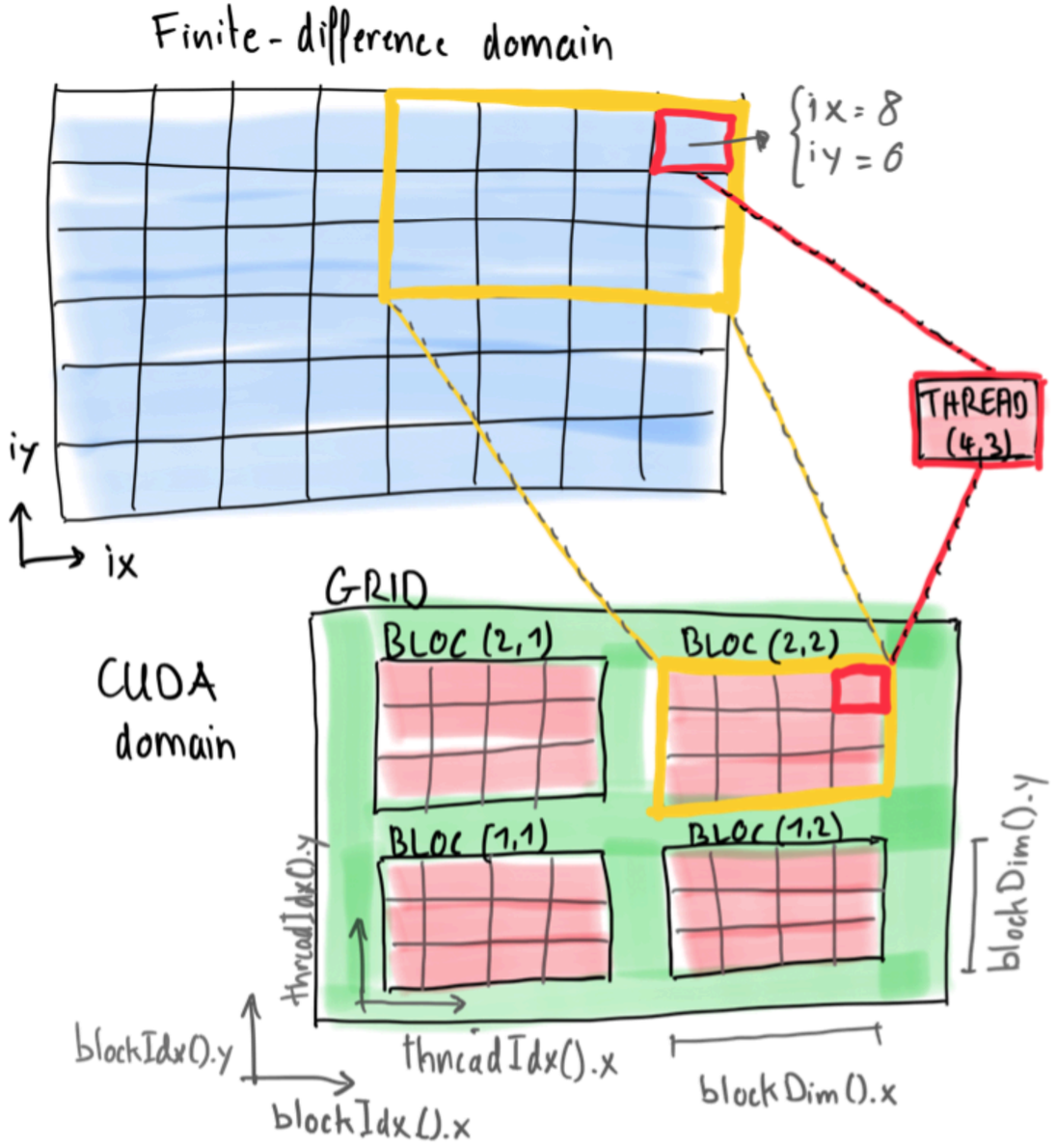
$$T_{\text{eff}} = \frac{A_{\text{eff}}}{t_{\text{it}}}$$

https://doi.org/10.5194/gmd-15-5757-2022

Matching the grids

Hardware-aware discretisation

- GPU architecture matching the computational domain.
- Finite-differences naturally map the underlying hardware layout.
- Use lower-order methods to limit memory traffic.
- Ensure accuracy by high spatial and temporal resolution.



https://pde-on-gpu.vaw.ethz.ch/lecture7/

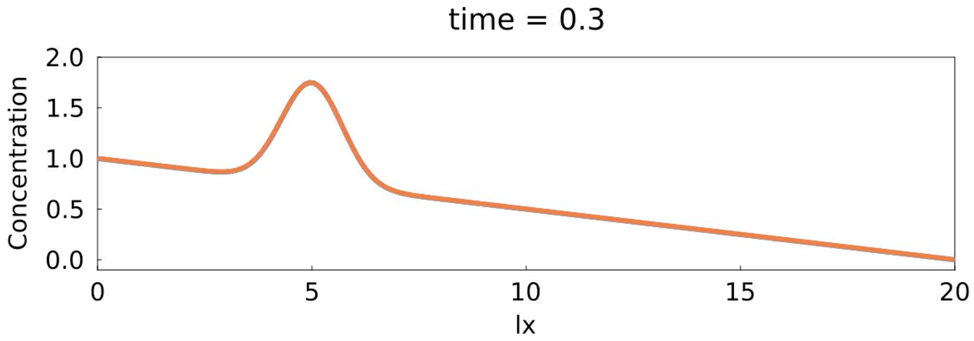
Pseudo-transient

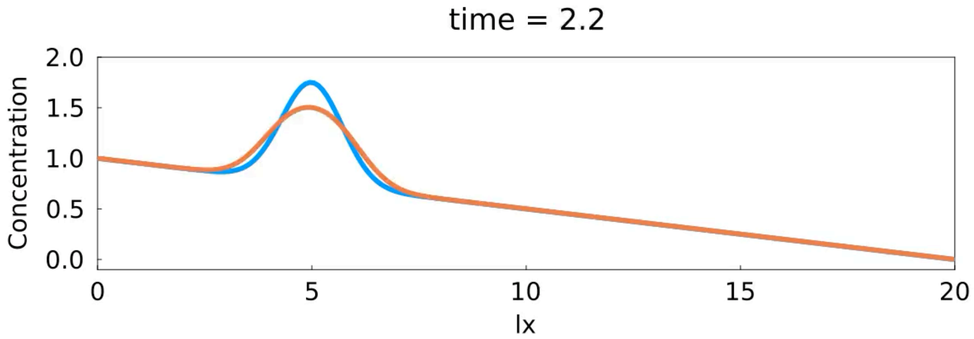
Matrix-free solvers

- Fast and implicit elliptic solvers.
- Analogy to transient physics with damping.
- Local algorithms, no global reductions.

$$\frac{\partial C}{\partial \tau} = \frac{\partial^2 C}{\partial x^2} \qquad \rho\frac{\partial^2 C}{\partial \tau^2} + \frac{\partial C}{\partial \tau} = \frac{\partial^2 C}{\partial x^2}$$

- Challenge on finding the iteration parameter ρ (estimating $\lambda_{\min}, \lambda_{\max}$)





https://doi.org/10.5194/gmd-15-5757-2022

https://doi.org/10.5194/egusphere-2025-5641

https://pde-on-gpu.vaw.ethz.ch/lecture3/

GPUs and Julia

Julia GPU

Portability and performance

- Supports array and kernel programming.
- Integration with vendor libs such as cu/rocBLAS, cu/rocFFT, cu/rocSPARSE and more ...
- Integration with dev tools like Nsight, CUPTI, rocprof.
- Support various toolkit versions with automatic installation through JLLs.

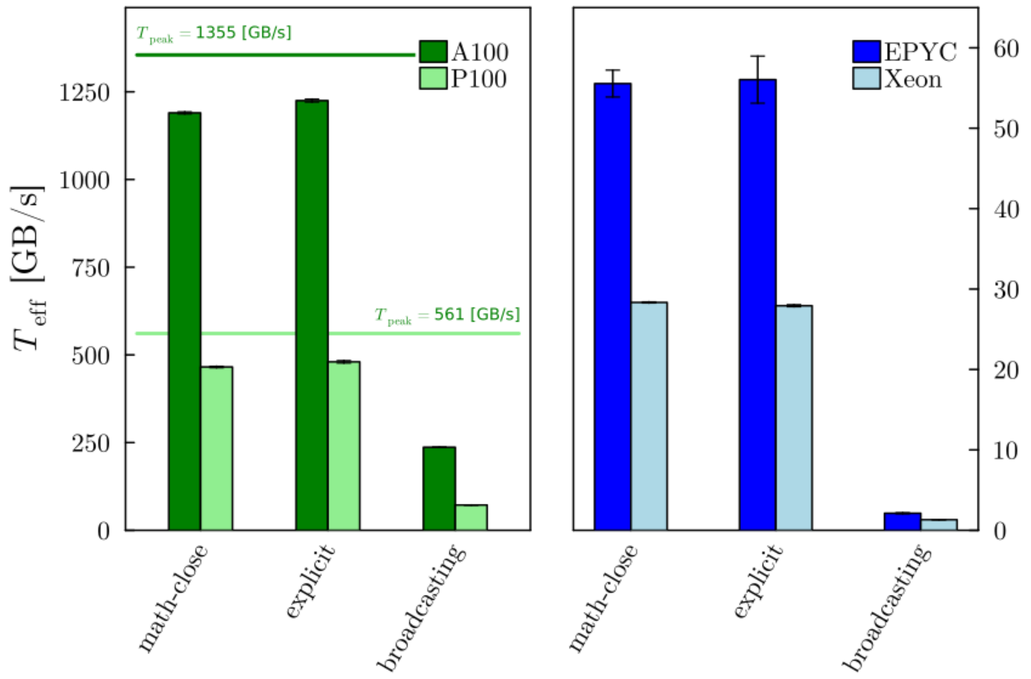
```
using CUDA
const dim = 100_000_000
const a = 3.1415

x = CUDA.ones(dim)
y = CUDA.ones(dim)
z = CUDA.zeros(dim)

# (a) SAXPY via high-level broadcasting
CUDA.@sync z .= a .* x .+ y

# (b) SAXPY via CUBLAS
CUDA.@sync CUBLAS.axpy!(dim, a, x, y)

# (c) SAXPY via CUDA kernel
function saxpy_gpu_kernel!(z, a, x, y)
    i = (blockIdx().x - 1) * blockDim().x +
        threadIdx().x
    if i <= length(z)
        @inbounds z[i] = a * x[i] + y[i]
    end
    return nothing
end

# launch configuration
nthreads = 1024
nblocks = cld(dim, nthreads)

# execute the kernel
CUDA.@sync @cuda(
    threads = nthreads,
    blocks = nblocks,
    saxpy_gpu_kernel!(z, a, x, y)
)
```

https://arxiv.org/abs/2211.02740

Backend abstraction

Stencil computations

ParallelStencil.jl enables parallel stencil computations on GPUs & CPUs.



https://github.com/omlins/ParallelStencil.jl

Benchmark application: 3D heat diffusion solver

```julia
using ParallelStencil
using ParallelStencil.FiniteDifferences3D
@init_parallel_stencil(CUDA, Float64, 3)

@parallel memopt=true optvars=T function step!(
    T2, T, Ci, lam, dt, _dx, _dy, _dz)
    @inn(T2) = @inn(T) + dt*(
        lam*@inn(Ci)*(@d2_xi(T)*_dx^2 +
                      @d2_yi(T)*_dy^2 +
                      @d2_zi(T)*_dz^2 ) )
    return
end

function diffusion3D()
    # Physics
    lam       = 1.0           #Thermal conductivity
    c0        = 2.0           #Heat capacity
    lx=ly=lz  = 1.0           #Domain length x|y|z

    # Numerics
    nx=ny=nz  = 512           #Nb gridpoints x|y|z
    nt        = 100           #Nb time steps
    dx        = lx/(nx-1)     #Space step in x
    dy        = ly/(ny-1)     #Space step in y
    dz        = lz/(nz-1)     #Space step in z
    _dx, _dy, _dz = 1.0/dx, 1.0/dy, 1.0/dz

    # Initial conditions
    T  = @ones(nx,ny,nz).*1.7 #Temperature
    T2 = copy(T)              #Temperature (2nd)
    Ci = @ones(nx,ny,nz)./c0  #1/Heat capacity

    # Time loop
    dt = min(dx^2,dy^2,dz^2)/lam/maximum(Ci)/6.1
    for it = 1:nt
        @parallel memopt=true step!(
            T2, T, Ci, lam, dt, _dx, _dy, _dz)
        T, T2 = T2, T
    end

end

diffusion3D()
```

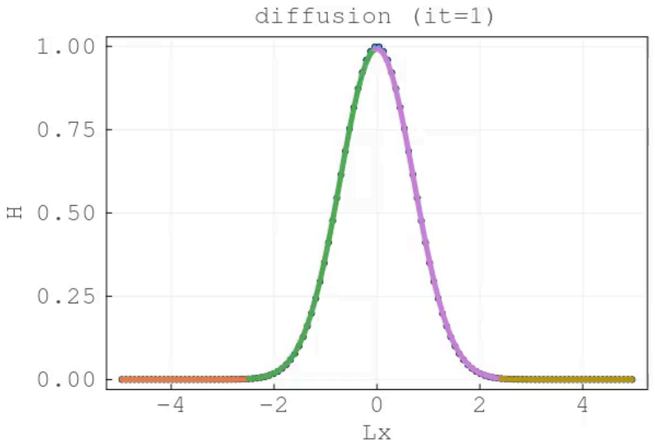
https://doi.org/10.21105/jcon.00138

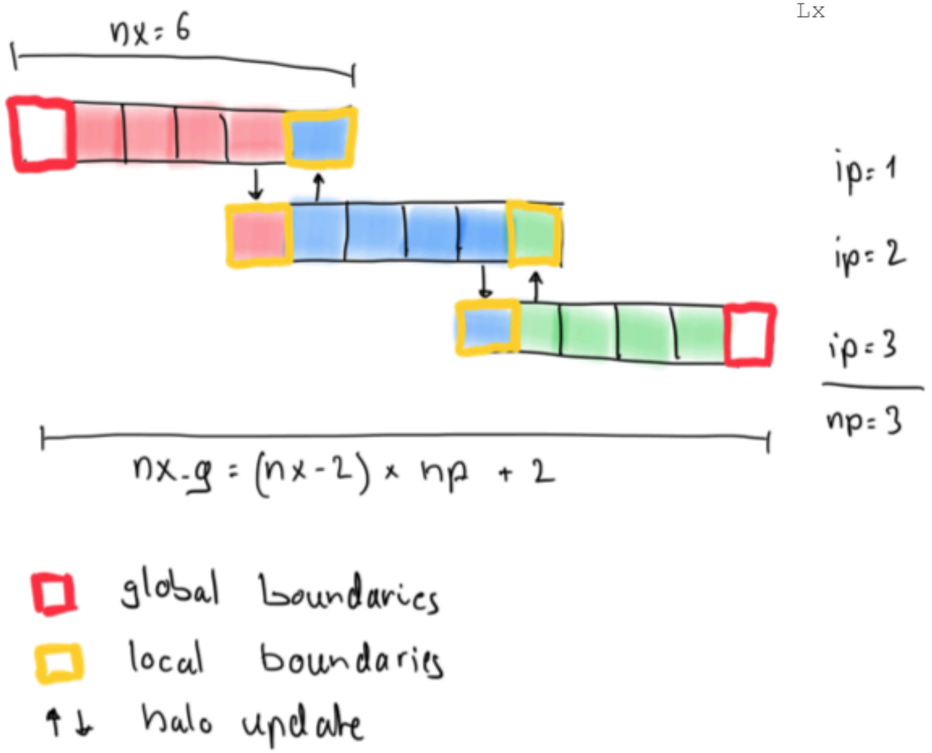
GPUs and Julia in HPC

Multi-GPUs

Julia and distributed memory computing

- Leveraging more then a single GPU
- Distributed memory computing
- GPU-aware MPI
- Implicit global grid
- Halo exchange
- Latency hiding (communication — computation overlap)





https://pde-on-gpu.vaw.ethz.ch/lecture9/

Distributed computing

Julia and distributed memory computing

ImplicitGlobalGrid.jl

- Relies on Julia MPI wrapper MPI.jl (could also use NCCL or other transport library).
- Only 3 additional lines needed to massively parallelise the previous example!
- Supports GPU-aware MPI (CUDA, ROCm).

Benchmark application: 3D heat diffusion solver

```
using ImplicitGlobalGrid
using ParallelStencil
using ParallelStencil.FiniteDifferences3D
@init_parallel_stencil(CUDA, Float64, 3)

@parallel function step!(T2,T,Ci,lam,dt,dx,dy,dz)
    @inn(T2) = @inn(T) + dt*(
        lam*@inn(Ci)*(@d2_xi(T)/dx^2 +
                      @d2_yi(T)/dy^2 +
                      @d2_zi(T)/dz^2 ) )
    return
end

function diffusion3D()
    # Physics
    lam       = 1.0           #Thermal conductivity
    c0        = 2.0           #Heat capacity
    lx=ly=lz  = 1.0           #Domain length x|y|z

    # Numerics
    nx=ny=nz  = 512           #Nb gridpoints x|y|z
    nt        = 100           #Nb time steps
    me,       = init_global_grid(nx, ny, nz)
    dx        = lx/(nx_g()-1) #Space step in x
    dy        = ly/(ny_g()-1) #Space step in y
    dz        = lz/(nz_g()-1) #Space step in z

    # Initial conditions
    T  = @ones(nx,ny,nz).*1.7 #Temperature
    T2 = copy(T)              #Temperature (2nd)
    Ci = @ones(nx,ny,nz)./c0  #1/Heat capacity

    # Time loop
    dt = min(dx^2,dy^2,dz^2)/lam/maximum(Ci)/6.1
    for it = 1:nt
        @hide_communication (16, 2, 2) begin
            @parallel step!(T2,T,Ci,lam,dt,dx,dy,dz)
            update_halo!(T2)
        end
        T, T2 = T2, T
    end

    finalize_global_grid()
end

diffusion3D()
```

https://github.com/eth-cscs/ImplicitGlobalGrid.jl

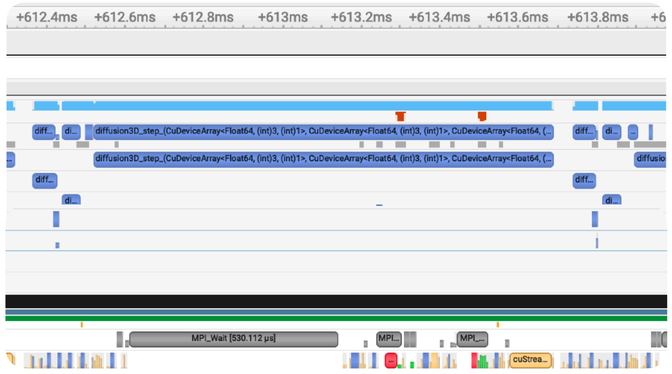
https://doi.org/10.21105/jcon.00137

Julia on Nvidia GPUs

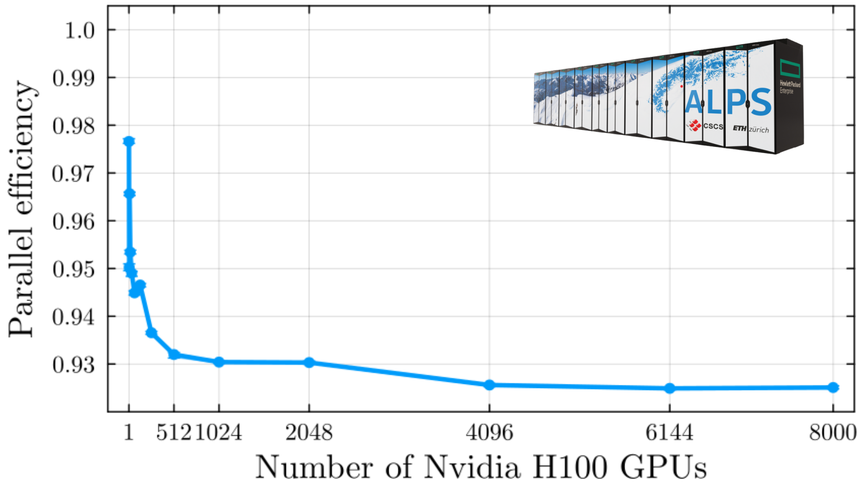
Scaling on ALPS

ImplicitGlobalGrid.jl

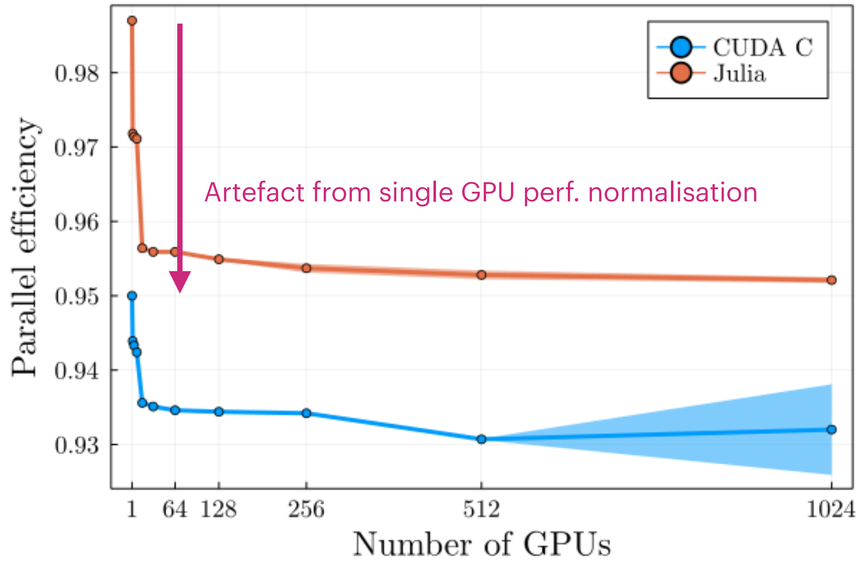
- Parallel efficiency close to 1
- Weak scaling on CSCS supercomputers ALPS (Nvidia GH200) & previous Piz Daint (Nvidia P100)
- Communication latency hiding



Benchmark application: 3D heat diffusion solver



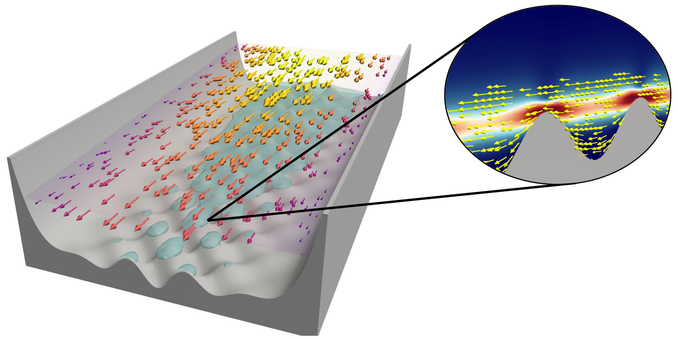
Real world application: 3D poro-visco-elastic two-phase flow solver



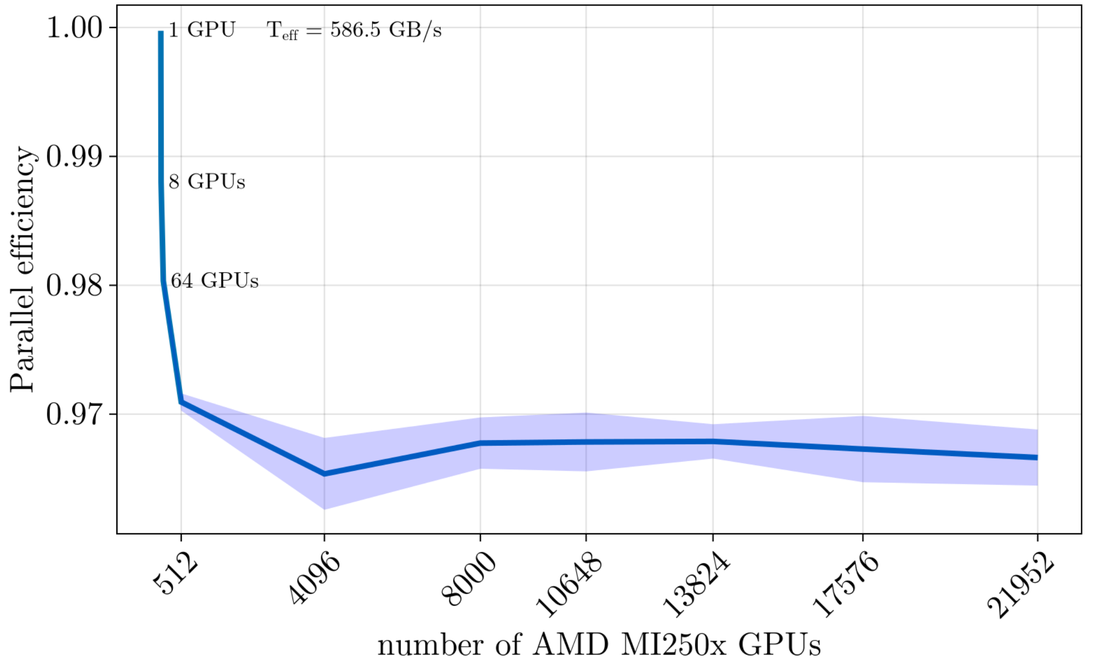
https://github.com/eth-cscs/ImplicitGlobalGrid.jl

Julia on AMD GPUs

Scaling on LUMI

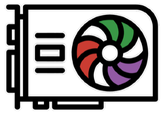


- 97% of parallel efficiency in weak scaling on 21’952 AMD MI250x GPUs.
- **Challenge:** AMDGPU.jl requirements to use task local approach.
- Chmy.jl — Overlapping communication and computation using cooperative multitasking in Julia.
- Each task gets local GPU context and stream.



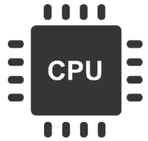
https://github.com/PTsolvers/Chmy.jl

Outlook



Backends

CPUs vs GPUs vs TPUs



CPUs: ARM, x86, ...

GPUs: AMD, Nvidia, Apple, ...

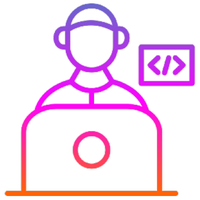
Design

Prototyping vs Production

Modularity

Abstractions

julia

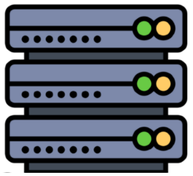


Users

Learning vs Advanced

Teaching vs Research

Timescales (PhD, project)



Compute infrastructure

Laptop, Workstation

University clusters

Supercomputers

Code conciseness

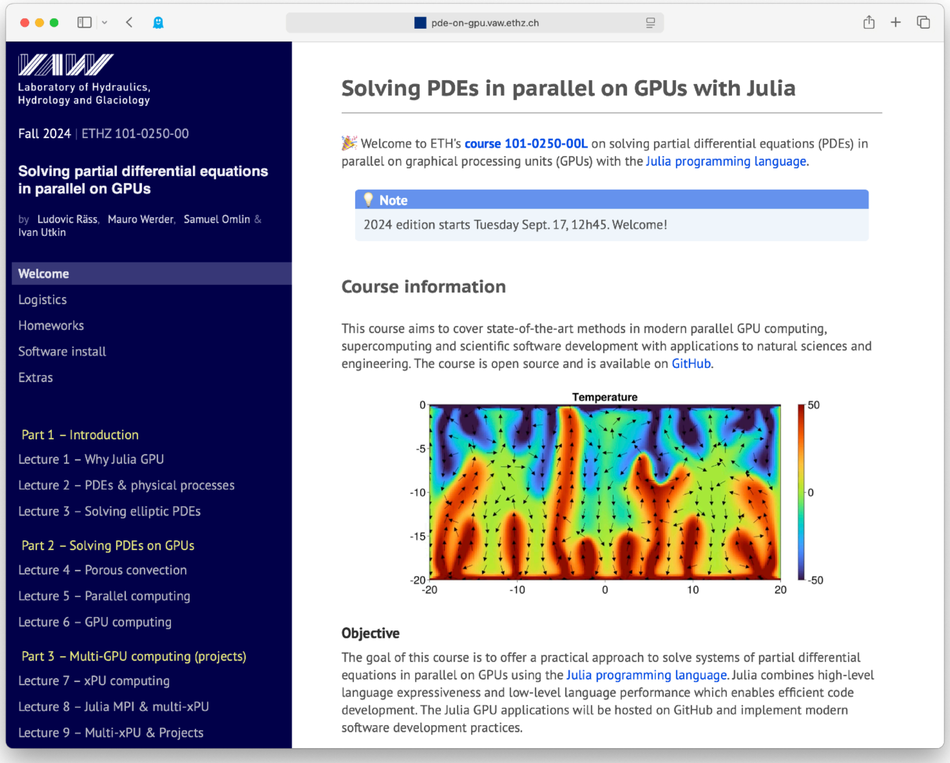
Monster codes: terrible to maintain

Worse; rewrite?

Education

Lateral spreading of supercomputing

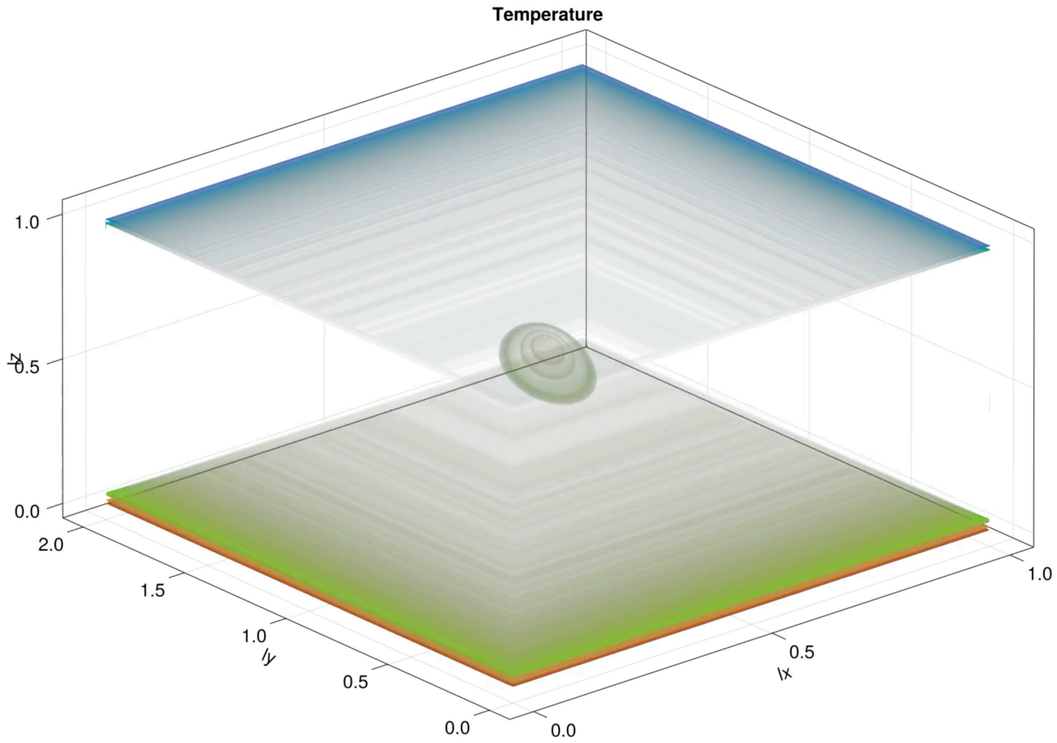
- ETHZ course on PDEs, GPUs & HPC with Julia



https://pde-on-gpu.vaw.ethz.ch/

ETH zürich

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre



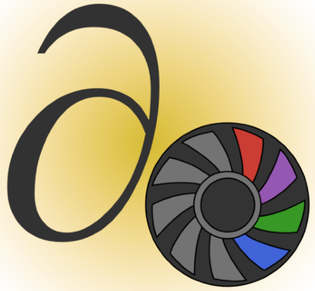
22 ETH Learning and Teaching Journal, Vol 4, No 1, 2023

Teaching supercomputing and software engineering skills to science and engineering students

https://learningteaching.ethz.ch/index.php/lt-eth/article/view/235

∂GPU4GEO Differentiable multi-physics solvers

for extreme-scale geophysics simulations on GPUs



Platform for Advanced Scientific Computing

ETH zürich

CSCS Centro Svizzero di Calcolo Scientifico Swiss National Supercomputing Centre

julia

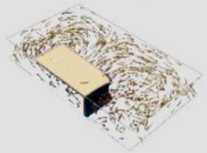
Gradient-based optimisation

GPU4GEO — forward modelling

Flagship Applications

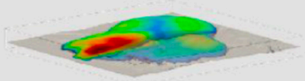
Geodynamics

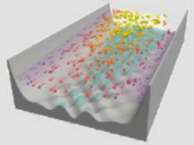
JustRelax.jl



Ice flow

FastIce.jl





Forward PT solvers

$$\frac{\partial u}{\partial \tau} + R(u) = 0$$

Forward model outputs

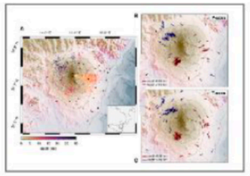
- Velocities
- Stresses
- Geometry evolution

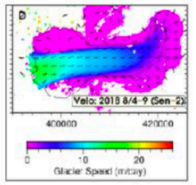
Autodiff (AD)

∂GPU4GEO — sensitivity and inverse modelling

Data

- In-situ field observations
- Remote sensing products
- Meteorological time series





Enzyme.jl

Adjoint PT solvers

$$\frac{\partial \psi}{\partial \tau} + \left[\frac{\partial R(u)}{\partial u}\right]^{\mathrm{T}} \psi = \frac{\partial J}{\partial u}$$

Thank you

For your attention

https://gpu4geo.org/

https://juliagpu.org/

https://github.com/JuliaGeodynamics

https://github.com/PTsolvers

ludovic.raess@unil.ch

luraess

Unil.