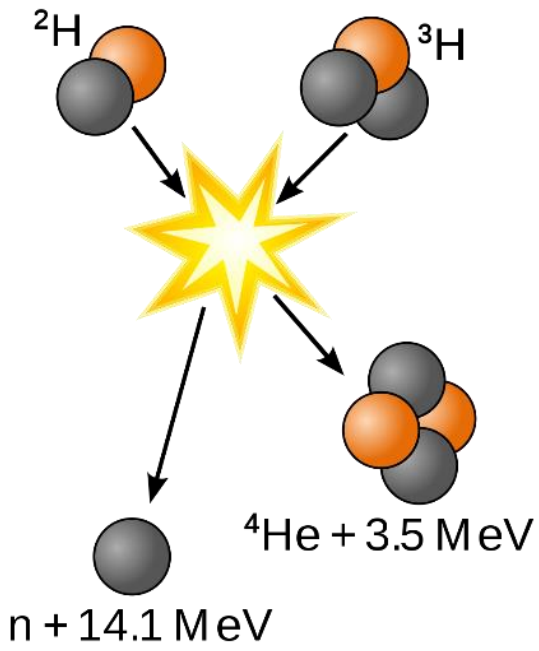
GPU programming in practice - Feltor

M. Wiesenberger

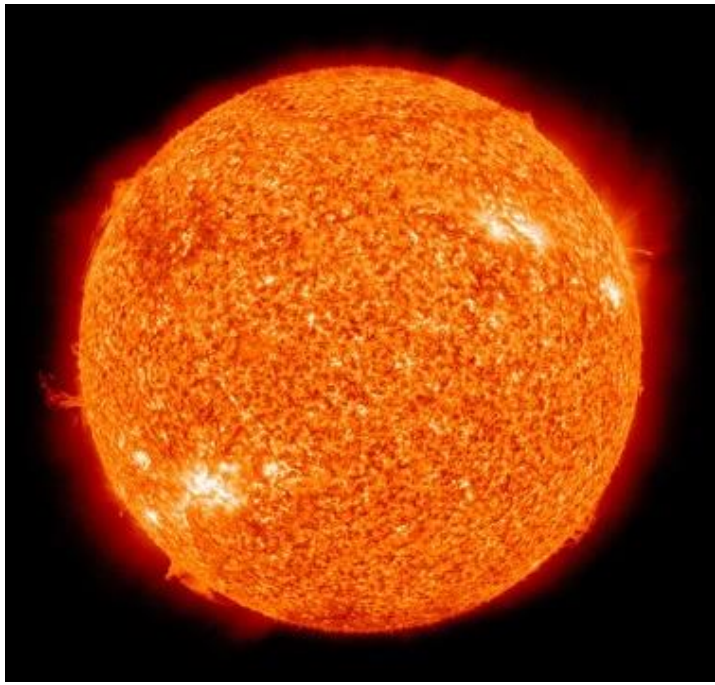
Outline

- Fusion research and GPUs – a **visual tour**
- The Feltor code and the **container free numerical algorithm** design principle in C++
- Performance is **memory bandwidth bound**
- Do we need **binary reproducibility in practice?**

Fusion



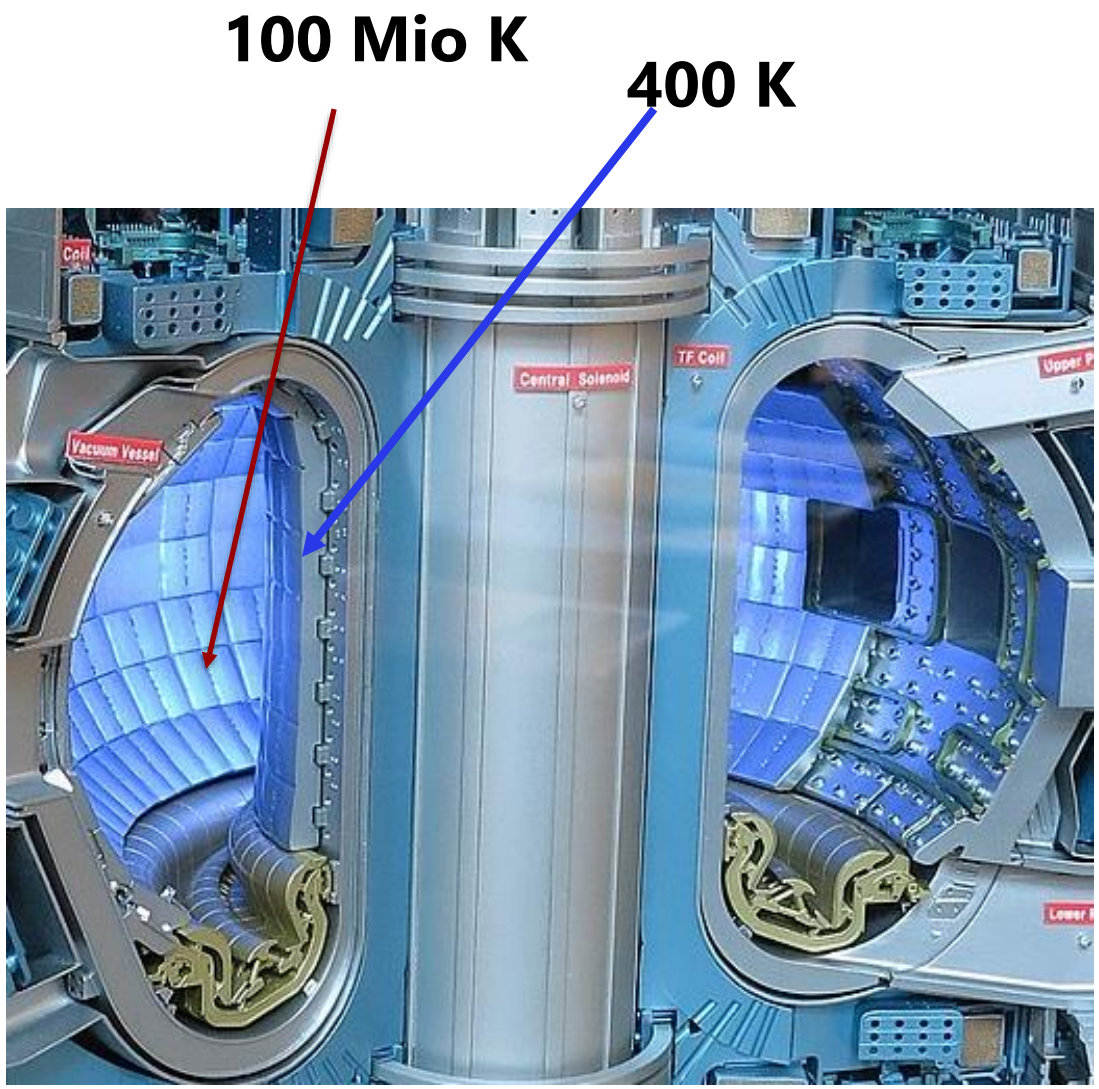
- Merge Hydrogen isotopes to He
- releases **million times more energy** than chemical reactions (coal, oil), due to nuclear vs. molecular binding energies
- Fuel abundance for **world energy production** = 4 Bio. Years with D-D fusion (~ comparable to lifetime of Earth)



Fusion on Earth

Plasma turbulence leads to **unsustainably high heat flux** unto material wall (higher than space shuttle re-entering Earth's atmosphere)

Numerical simulations needed to predict heat load onto plasma facing components



Full-F (gyro**fluid**) electromagnetic model in toroidal geometry (Feltor)

$$\frac{\partial}{\partial t}N = -\frac{1}{B}[\psi, N]_\perp - \bar{\nabla}_\parallel (NU) - NU\left(\boldsymbol{\nabla}\cdot\hat{\boldsymbol{b}} + \boldsymbol{\nabla}\cdot\boldsymbol{b}_\perp\right) - \tau\mathcal{K}(N)$$
$$- N\mathcal{K}(\psi) - \mu\mathcal{K}_{\boldsymbol{\nabla}\times\hat{\boldsymbol{b}}}(NU^2) - \mu NU^2\boldsymbol{\nabla}\cdot\boldsymbol{\mathcal{K}}_{\boldsymbol{\nabla}\times\hat{\boldsymbol{b}}} + \nu_\perp\Delta_\perp N + \nu_\parallel\Delta_\parallel N + S_N, \tag{98a}$$

$$\frac{\partial}{\partial t}W = -\frac{1}{B}[\psi, U]_\perp - \frac{1}{\mu}\bar{\nabla}_\parallel\psi - \frac{1}{2}\bar{\nabla}_\parallel U^2 - \frac{\tau}{\mu}\bar{\nabla}_\parallel \ln N - U\mathcal{K}_{\boldsymbol{\nabla}\times\hat{\boldsymbol{b}}}(\psi) - \tau\mathcal{K}(U) - \tau U\boldsymbol{\nabla}\cdot\boldsymbol{\mathcal{K}}_{\boldsymbol{\nabla}\times\hat{\boldsymbol{b}}}$$
$$- \left(2\tau + \mu U^2\right)\mathcal{K}_{\boldsymbol{\nabla}\times\hat{\boldsymbol{b}}}(U) - 2\tau U\mathcal{K}_{\boldsymbol{\nabla}\times\hat{\boldsymbol{b}}}(\ln N) - \frac{\eta}{\mu}\frac{n_e}{N}n_e(U_i - u_e)$$
$$+ \nu_\perp\Delta_\perp U + \nu_\parallel\Delta_\parallel U, \tag{98b}$$

$$W := \left(U + \frac{A_\parallel}{\mu}\right) \tag{98c}$$

together with $\bar{\nabla}_\parallel f = \nabla_\parallel f + A_\parallel\mathcal{K}_{\boldsymbol{\nabla}\times\hat{\boldsymbol{b}}}(f) + \frac{1}{B}[f, A_\parallel]_\perp$ and $\boldsymbol{\nabla}\cdot\boldsymbol{b}_\perp = A_\parallel\boldsymbol{\nabla}\cdot\boldsymbol{\mathcal{K}}_{\boldsymbol{\nabla}\times\hat{\boldsymbol{b}}} - \mathcal{K}_{\boldsymbol{\nabla}B}(A_\parallel)$ and

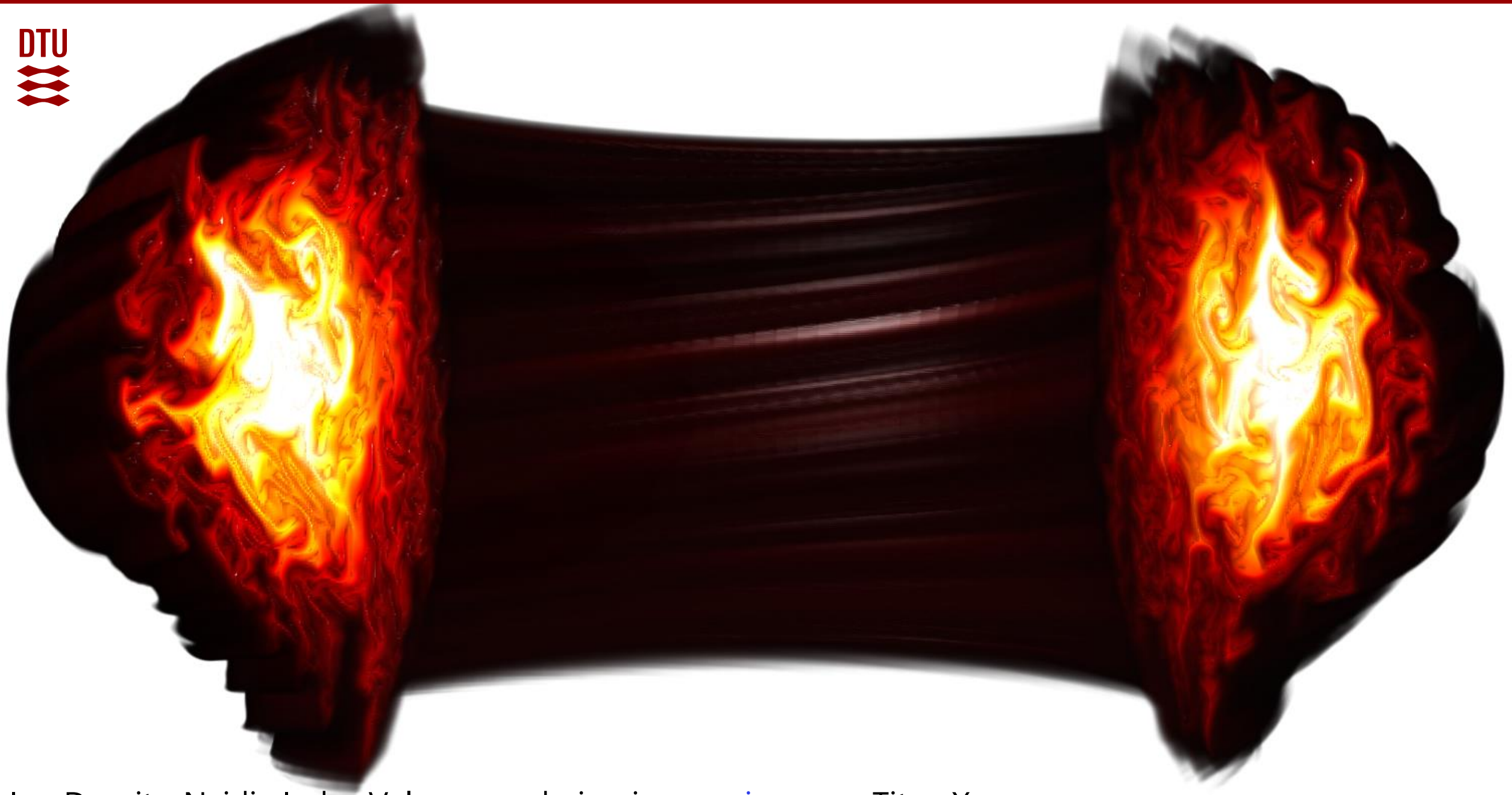
$$-\boldsymbol{\nabla}\cdot\left(\frac{N_i}{B^2}\boldsymbol{\nabla}_\perp\phi\right) = \Gamma_{1,i}N_i - n_e, \qquad \Gamma_{1,i}^{-1} = 1 - \frac{1}{2}\tau_i\mu_i\Delta_\perp \tag{99a}$$

$$\psi_e = \phi, \quad \psi_i = \Gamma_{1,i}\phi - \frac{\mu_i}{2}\frac{(\boldsymbol{\nabla}_\perp\phi)^2}{B^2} \tag{99b}$$

$$\left(\frac{\beta}{\mu_i}N_i - \frac{\beta}{\mu_e}n_e - \Delta_\perp\right)A_\parallel = \beta\left(N_iW_i - n_ew_e\right) \tag{99c}$$

Typical production run

- Run on 16 GPUs (Tesla V100)
- For a week
- Resolution ~ 30Mio points = 250MB
- Output ~ 100 GB

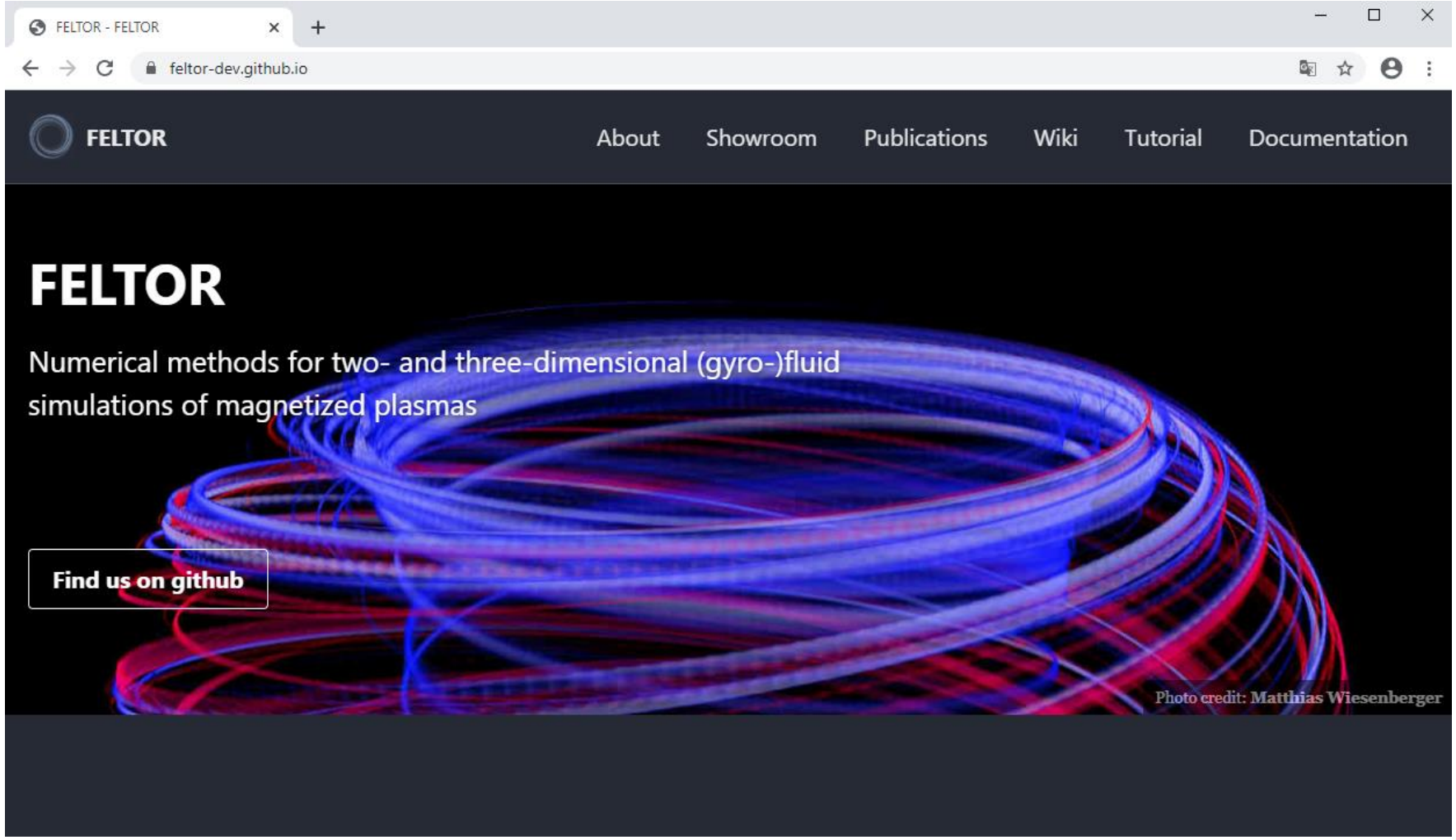


Ion Density. Nvidia Index Volume rendering in paraview on a Titan Xp

Electron velocity: Nvidia Index Volume rendering in paraview on a Titan Xp

Magnetic field. Streamline tracing in paraview and raytracing using Titan Xp

https://feltor-dev.github.io/



FELTOR

Numerical methods for two- and three-dimensional (gyro-) fluid simulations of magnetized plasmas

Matthias Wiesenberger

Matthias started the development of Feltor early 2012 as part of his PhD project in the group of Alexander Kendl at Innsbruck University, Austria. Since summer 2017 Matthias has worked as a postdoctoral researcher in the Plasma Physics and Fusion Energy group at the Technical University of Denmark (DTU).

Markus Held

Markus, a fellow PhD student in Innsbruck, joined the development of Feltor in 2014. Markus currently works as a postdoctoral researcher at the department of Space, Earth and Environment, Astronomy and Plasma Physics at Chalmers University of Technology in Gothenburg, Sweden.

What is FELTOR? - FELTOR

https://feltor-dev.github.io/wiki/

FELTOR

About Showroom Publications Wiki Tutorial Documentation

WIKI

- What is FELTOR
- Contribute
- Design
- Error handling
- Troubleshooting

What is FELTOR?

FELTOR (Full-F ELectromagnetic code in TORoidal geometry) is a modular scientific software package that can roughly be divided into six parts described as follows

6	diagnostics	Physical projects / User zone
5	applications	
4	Advanced numerical schemes	dg library (discontinuous galerkin)/ Developer zone
3	Topology and Geometry	
2	Basic numerical algorithms	
1	Vector and Matrix operations	

The structure of the FELTOR project

User Zone:

A collection of actual simulation projects and diagnostic programs for two- and three-dimensional drift- and gyro-fluid models

What is FELTOR? - FELTOR

https://feltor-dev.github.io/wiki/

FELTOR

About Showroom Publications Wiki Tutorial Documentation

WIKI

- What is FELTOR
- Contribute
- Design
- Error handling
- Troubleshooting

What is FELTOR?

FELTOR (Full-F ELectromagnetic code in TORoidal geometry) is a modular scientific software package that can roughly be divided into six parts described as follows

6	diagnostics	Physical projects / User zone
5	applications	
4	Advanced numerical schemes	dg library (discontinuous galerkin)/ Developer zone
3	Topology and Geometry	
2	Basic numerical algorithms	
1	Vector and Matrix operations	

The structure of the FELTOR project

Hardware abstraction layer – "Where the GPU Kernels hide"

User Zone:

A collection of actual simulation projects and diagnostic programs for two- and three-dimensional drift- and gyro-fluid models

```
1  #include <iostream>
2  #include <array>
3  // include the dg-library
4  #include "dg/algorithm.h"
5
6  int main()
7  {
8
9    std::array<double, 2> x ={2,2}, y = {4,4};
10   double a = 0.5, b = 0.25;
11   // compute a*x+b*y and store it in y
12   dg::blas1::axpby( a, x, b, y);
13   // compute Sum_i x_i y_i
14   double sum = dg::blas1::dot( x,y);
15   // output should be 8
16   std::cout << sum << std::endl;
17   return 0;
18 }
```

Warm-up Question:

What is the value of y_[0] and y[1] in line 13?

```
#include <iostream>
#include <array>
// include the dg-library
#include "dg/algorithm.h"

int main()
{
    //use the thrust library to allocate memory on the GPU
    thrust::device_vector<double> x(1e6, 2), y(1e6, 4);
    double a = 0.5, b = 0.25;
    // compute a*x+b*y and store it in y
    dg::blas1::axpby( a, x, b, y);
    // compute Sum_i x_i y_i
    double sum = dg::blas1::dot( x,y);
    // output should be ... large
    std::cout << sum << std::endl;
    return 0;
}
```

Same code, but executes on GPU!

MPI+GPU

```cpp
#include <iostream>
//activate MPI in FELTOR
#include "mpi.h"
#include "dg/algorithm.h"

int main(int argc, char* argv[])
{
  //init MPI
  MPI_Init( &argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;
  int np,rank;
  MPI_Comm_size(comm, &np);
  MPI_Comm_rank(comm, &rank);
  //allocate and initialize local memory
  thrust::device_vector<double> x_local( 1e8/np, 2), y_local(1e8/np, 4);
  //combine the local vectors to a global MPI vector
  dg::MPI_Vector<thrust::device_vector<double>> x(x_local, comm);
  dg::MPI_Vector<thrust::device_vector<double>> y(y_local, comm);

  //now repeat the operations from before..
  double a = 0.5, b = 0.25;
  // compute a*x+b*y and store it in y
  dg::blas1::axpby(a, x, b, y);
  // compute Sum_i x_i y_i
  double sum = dg::blas1::dot(x, y);
  // output should be ... large
  if(rank==0)std::cout << sum << std::endl;

  //be a good MPI citizen and clean up
  MPI_Finalize();
  return 0;
}
```

```
MPI_Init( &argc, &argv);
MPI_Comm comm = MPI_COMM_WORLD;
int np,rank;
MPI_Comm_size(comm, &np);
MPI_Comm_rank(comm, &rank);
//allocate and initialize local memory
thrust::device_vector<double> x_local( 1e8/np, 2), y_local(1e8/np, 4);
//combine the local vectors to a global MPI vector
dg::MPI_Vector<thrust::device_vector<double>> x(x_local, comm);
dg::MPI_Vector<thrust::device_vector<double>> y(y_local, comm);

//now repeat the operations from before..
double a = 0.5, b = 0.25;
// compute a*x+b*y and store it in y
dg::blas1::axpby(a, x, b, y);
// compute Sum_i x_i y_i
double sum = dg::blas1::dot(x, y);
// output should be ... large
if(rank==0)std::cout << sum << std::endl;

//be a good MPI citizen and clean up
MPI_Finalize();
```

Platform independent code:

runs unchanged on laptop, GPU, MPI, HPC cluster ....

Design idea - **separate implementation of an algorithm from its parallelization (and optimization)**

1. Define a **set of common functions - the „building blocks“** **(**typically basic Matrix and Vector operations)

2.1 Implement the algorithm in terms of these „building blocks“ (“Lego”, automatically inherits the benefits of optimization efforts)

2.2 parallelize and optimize for GPUs, OpenMP, MPI, … i.e. write Cuda Kernel

Example: Runge Kutta

$$y_{n+1} = y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$
$$t_{n+1} = t_n + h$$

for n = 0, 1, 2, 3, ..., using[3]

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\ k_4 &= f\left(t_n + h, y_n + hk_3\right). \end{aligned}$$

Can be implemented with **2 Kernels**

axpby(a, x, b, y);

y <- a*x+b*y

With scalars a, b and arrays x and y

rhs(t, y, yp);

yp <- f(t,y)

With scalar t and array y

https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods

Exercise, discuss with your neighbour

1. Explain what algorithm / problem you want to implement on GPU(s) or take conjugate gradient (wikipedia) – go to "The resulting algorithm"
2. Identify the basic (parallel) operations (**Kernels**) you need in order to implement your algorithm (e.g. vector addition, sorting, FFT…). **How many Kernels do you need to write?**
3. What advantages/disadvantages do you see with this approach?

https://en.wikipedia.org/wiki/Conjugate_gradient_method#The_resulting_algorithm

Conjugate Gradient

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$

if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result

$\mathbf{p}_0 := \mathbf{r}_0$

$k := 0$

repeat

$$\alpha_k := \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k}$$

$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$

$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$

if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop

$$\beta_k := \frac{\mathbf{r}_{k+1}^\top \mathbf{r}_{k+1}}{\mathbf{r}_k^\top \mathbf{r}_k}$$

$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$

$k := k + 1$

end repeat

return $\mathbf{x}_{k+1}$ as the result

Solves **Ax = b**

A is a real, symmetric, positive definite matrix

x_0 is an input vector (the initial guess)

b is the right hand side

Conjugate gradient – 3 operations

```
template< class Matrix, class Vector, class Preconditioner, class Norm>
unsigned cg( Matrix& A, Vector& x, const Vector& b,
             Preconditioner& P, Norm& S, double eps)
{
    // ... initialize stuff
    for( unsigned i=1; i<max_iter; i++)
    {
        dg::blas2::symv( A, p, ap); //matrix - vector multiplication
        alpha =  nrmzr_old/dg::blas1::dot( p, ap); //dot product
        dg::blas1::axpby( alpha, p, 1.,x); //vector addition
        dg::blas1::axpby( -alpha, ap, 1., r); //vector addition
        if( sqrt( dg::blas2::dot(S,r)) < eps*nrmb)
            return i;
        dg::blas2::symv(P,r,ap); //matrix - vector multiplication
        nrmzr_new = dg::blas1::dot( ap, r); //dot product
        dg::blas1::axpby(1.,ap, nrmzr_new/nrmzr_old, p ); //vector addition
        nrmzr_old=nrmzr_new;
    }
    return max_iter;
}
```

What do you not see ?

```
template< class Matrix, class Vector, class Preconditioner, class Norm>
unsigned cg( Matrix& A, Vector& x, const Vector& b,
             Preconditioner& P, Norm& S, double eps)
{
    // ... initialize stuff
    for( unsigned i=1; i<max_iter; i++)
    {
        dg::blas2::symv( A, p, ap); //matrix - vector multiplication
        alpha =  nrmzr_old/dg::blas1::dot( p, ap); //dot product
        dg::blas1::axpby( alpha, p, 1.,x); //vector addition
        dg::blas1::axpby( -alpha, ap, 1., r); //vector addition
        if( sqrt( dg::blas2::dot(S,r)) < eps*nrmb)
            return i;
        dg::blas2::symv(P,r,ap); //matrix - vector multiplication
        nrmzr_new = dg::blas1::dot( ap, r); //dot product
        dg::blas1::axpby(1.,ap, nrmzr_new/nrmzr_old, p ); //vector addition
        nrmzr_old=nrmzr_new;
    }
    return max_iter;
}
```

Advantages

- Code is **easy to read and use** (parallelization details are hidden from the user, no expertise on GPUs required)
- Code **easy to maintain** (because only a set amount of Kernels)
- **Extensible** to new hardware (by providing an implementation of the building blocks on the new hardware, no need to change user code)
- **Platform independent** (code runs unchanged on various architectures from laptop to HPC system)
- **Flexible** – new algorithms that use the same building blocks can be readily implemented
- **Target optimizable** – Kernels can be heavily optimized for specific architectures

Disadvantages

- The set of primitive functions **defines what algorithms can be implemented** (and what cannot be implemented, e.g. direct solvers)
- **Optimization is restricted to the scope** of each function - sometimes vectors are loaded more often than needed, gives approx factor 2-3 **performance penalty** compared to „ideal“ implementation of the algorithm as a single function

How to implement in your own code

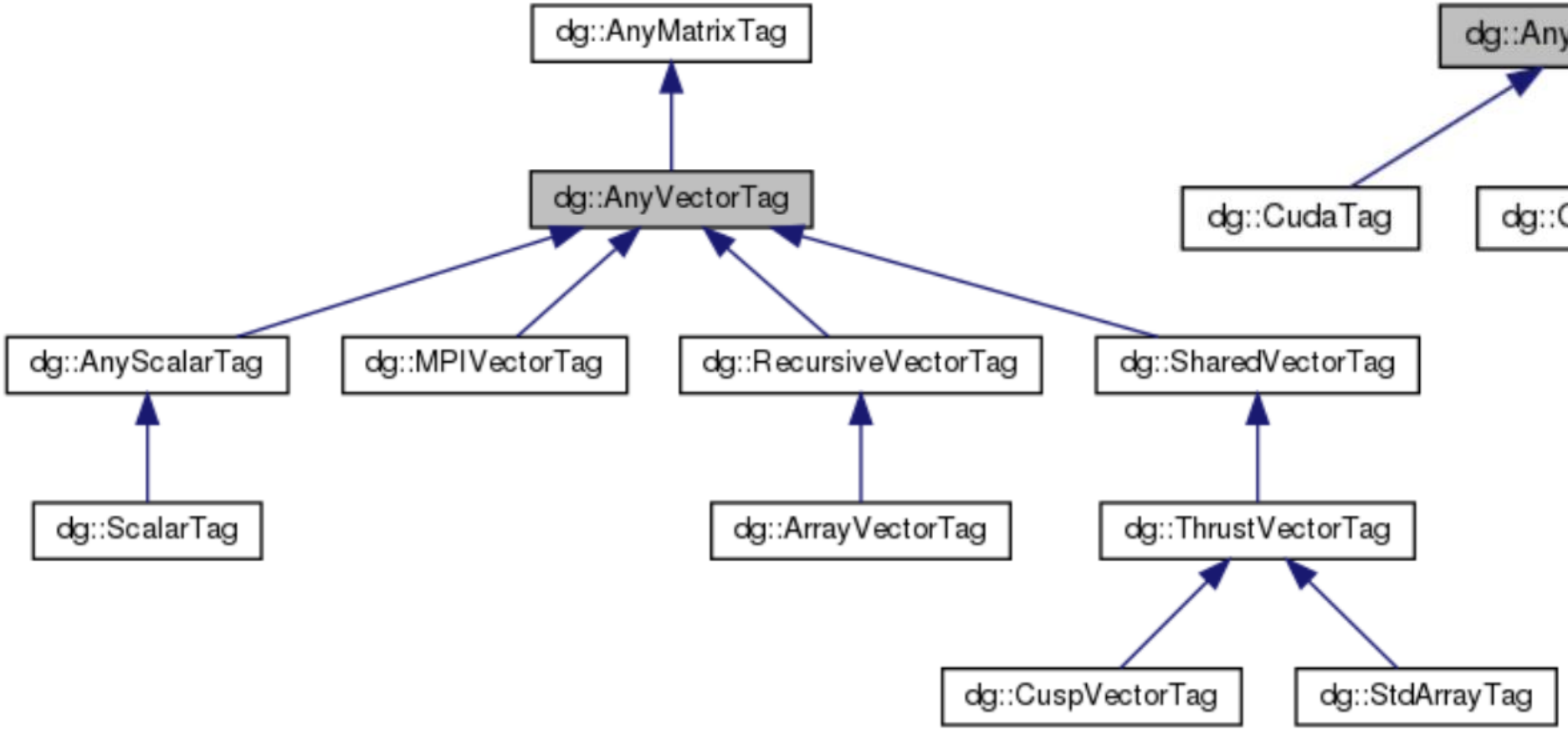
1. Formulate your problem in terms of elementary operations
2. Use a library! Chances are that what you need is already implemented (and optimized)

- https://github.com/thrust/thrust (**included in CUDA toolkit**)-> provides vector classes for memory allocation (**very useful**), sorting, reductions, scatter/gather, search
- https://github.com/cusplibrary/cusplibrary based on thrust, sparse matrix vector formats and preconditioners, matrix-vector multiplications, Krylov subspace solvers
- https://github.com/kokkos/kokkos provides abstractions for both parallel execution of code and data management
- https://github.com/feltor-dev/feltor implements a host of numerical algorithms (all sorts of time-steppers, Krylov subspace, multigrid, …)

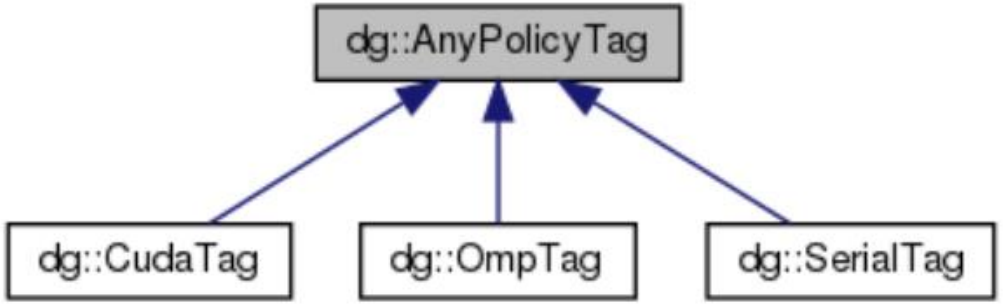
Do it yourself (become a modern C++ expert)

1. Identify the basic functions you need (don't make it too fine-grained, prefer functions with larger workload)
2. Implement for various architectures (i.e. write a CUDA Kernel and at least a serial version for testing)
3. Call the correct version based on the type of the input variables using a **Tag dispatch system (combined with Template Traits)** = compiler inspects the type (of parameters) and chooses the correct implementation for you **at compile time with 0 performance overhead** – advanced yet standard C++ technique, e.g. https://arne-mertz.de/2016/10/tag-dispatch/

Memory Tag

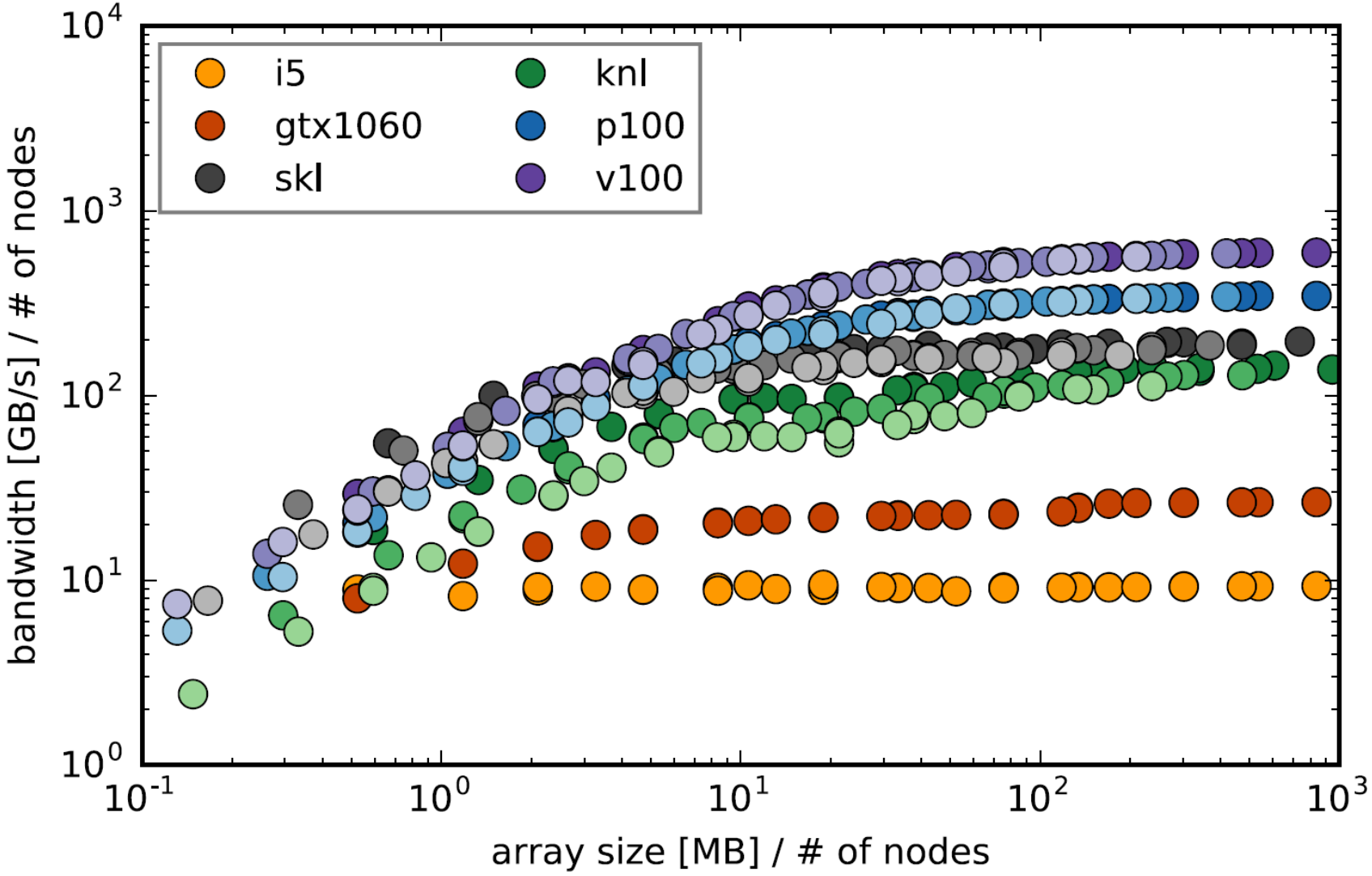


Parallelization Tag

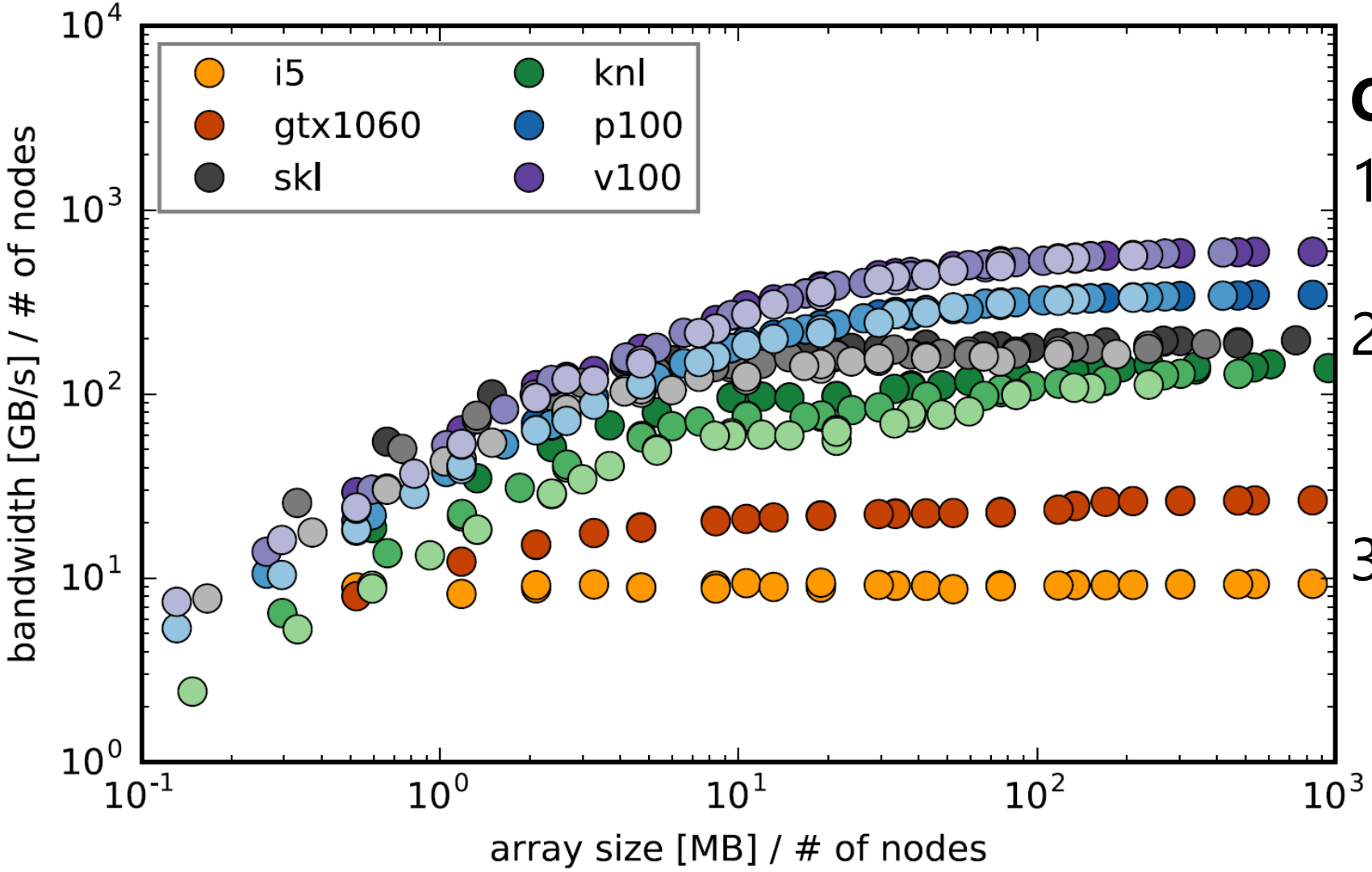


Array size / runtime – dg::blas1::dot(x,y)

$$\sum_i x_i y_i$$



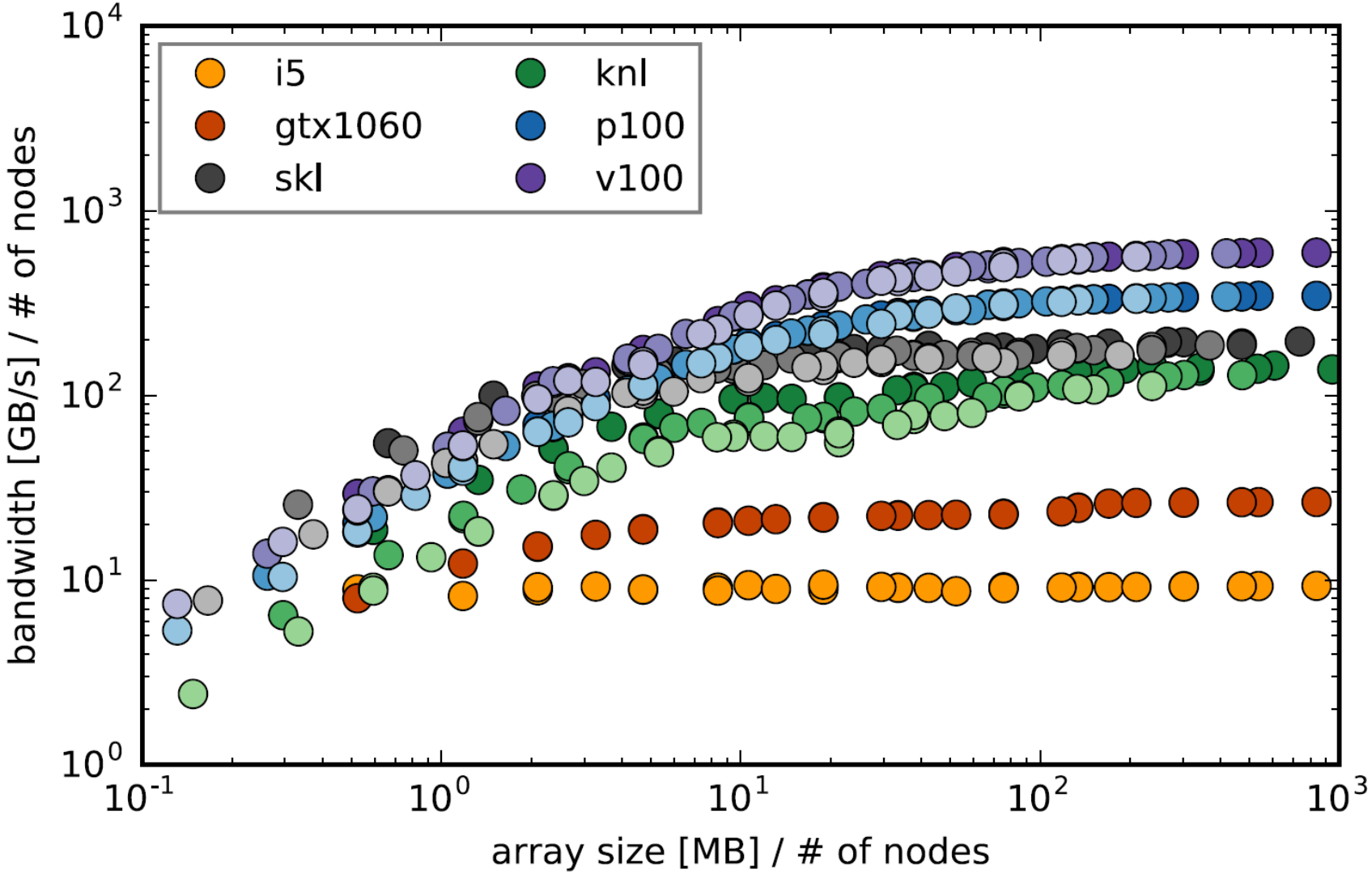
Array size / runtime – dg::blas1::dot(x,y)



Questions:

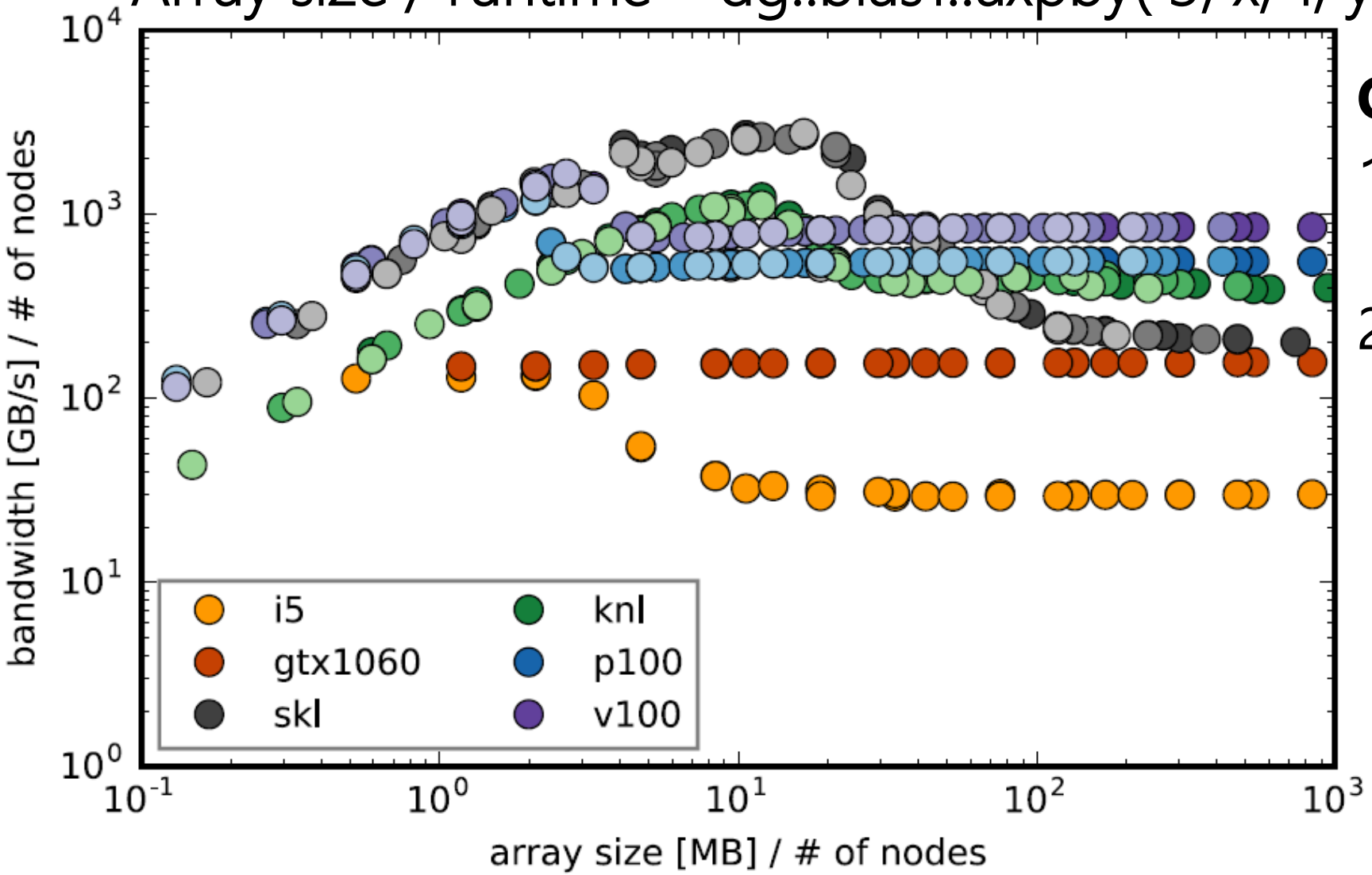
1. Which architecture is fastest?
2. What does a linear increase in throughput signify?
3. What limits the performance for large array sizes?

Array size / runtime – dg::blas1::dot(x,y)



$$t = T_{\mathrm{lat}}(n) + \frac{mS}{nB}$$

Array size / runtime – dg::blas1::axpby(3, x, 1, y)



Questions:

1. Which architecture is fastest and why?
2. What can you tell about the difference between GPUs and CPUs?

	Peak	Axpby			Dot		
	Bandwidth [GB/s]	Bandwidth [GB/s]	$T_{lat}(1)$ [μs]	$T_{lat}(4)$ [μs]	Bandwidth [GB/s]	$T_{lat}(1)$ [μs]	$T_{lat}(4)$ [μs]
i5	34	30 ± 01	00 ± 02	n/a	10 ± 01	05 ± 01	n/a
gtx1060	192	158 ± 01	00 ± 01	n/a	27 ± 01	93 ± 09	n/a
skl	256	207 ± 06	00 ± 01	00 ± 01	193 ± 19	18 ± 03	38 ± 05
knl	>400	394 ± 23	06 ± 01	10 ± 01	142 ± 07	55 ± 02	120 ± 06
p100	732	554 ± 01	01 ± 01	03 ± 01	347 ± 02	49 ± 01	49 ± 01
v100	898	849 ± 01	02 ± 01	03 ± 01	594 ± 03	34 ± 02	35 ± 01

GPUs are fast due to their fast memory

Simple performance model

Runtime of a simulation given by:

$$t(P,S,n) = N\left[T_{lat}(n) + 3.3\frac{S}{nB(P)}\right]$$

Where

S = array size

B = hardware bandwidth (of a single GPU e.g.)

P = hardware identifier

N = number of function calls

n = number of nodes / GPUs

Strong scaling efficiency

$$\varepsilon(P,S,n) := \frac{t(P,S,1)}{nt(P,S,n)}$$

Exercise: Compute Minimum array size per GPU $(S/n)_{\min}$ to get strong scaling above 50%! What happens if S is very large?

GPUs require a minimum amount of work in order to be used efficiently!

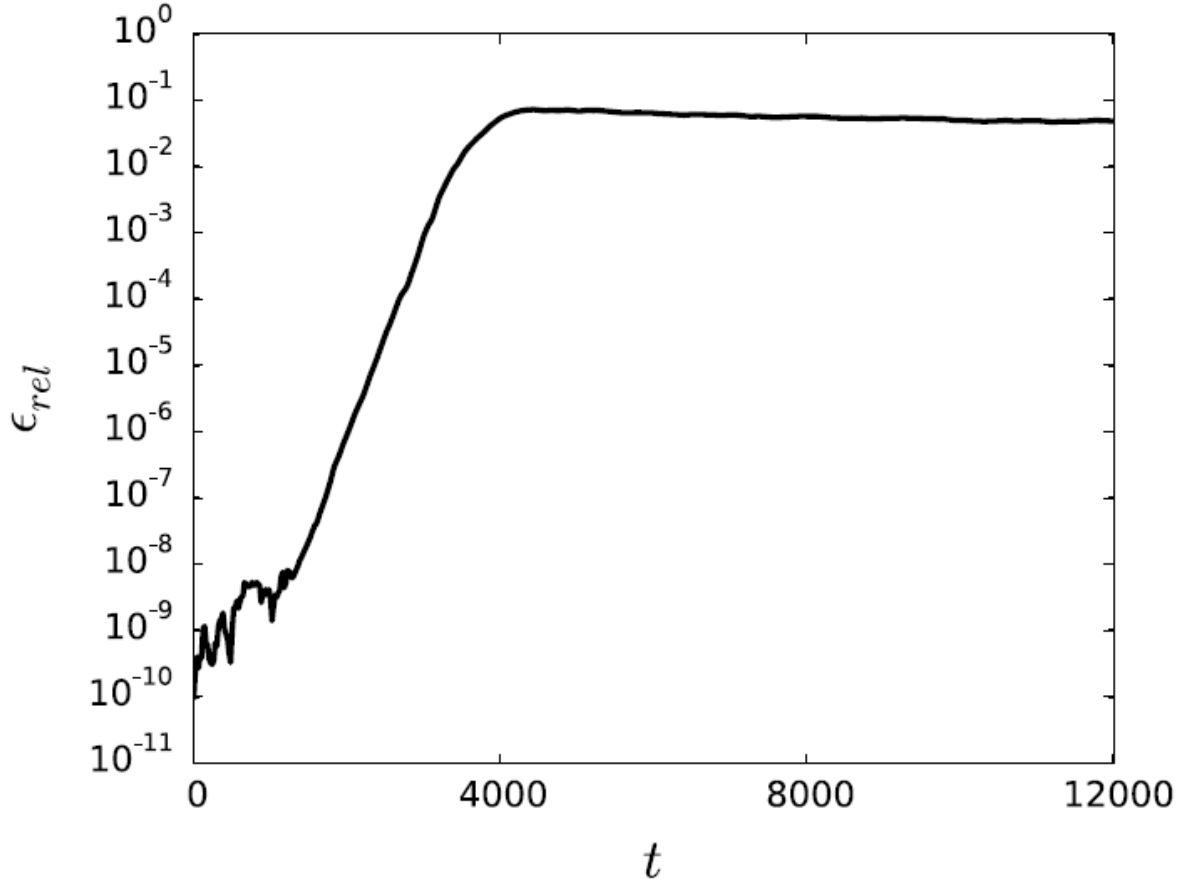
Minimum array size for (strong/weak) scaling above 50%:

$$(S/n)_{\min} \approx 0.3 T_{\mathrm{lat}}(n) B(P)$$

Paradox: for fixed array size fast hardware (high B) scales worse than slow hardware

Curious

- Implement equations modelling plasma turbulence
- On a parallel computer architecture
- Run the same binary with the same input twice
- Plot difference
- **Question: Why is it not the same result?**



M. Wiesenberger, L. Einkemmer, M. Held et al. / Computer Physics Communications 238 (2019) 145–156

R. Iakymchuk, S. Collange, D. Defour, S. Graillat, ExBLAS (exact blas) library, 2018, Available on the WWW, https://exblas.lip6fr/. (Accessed 10 March 2018).

Reproducibility

In parallel environment order of execution is not guaranteed

Numerical addition is not associative

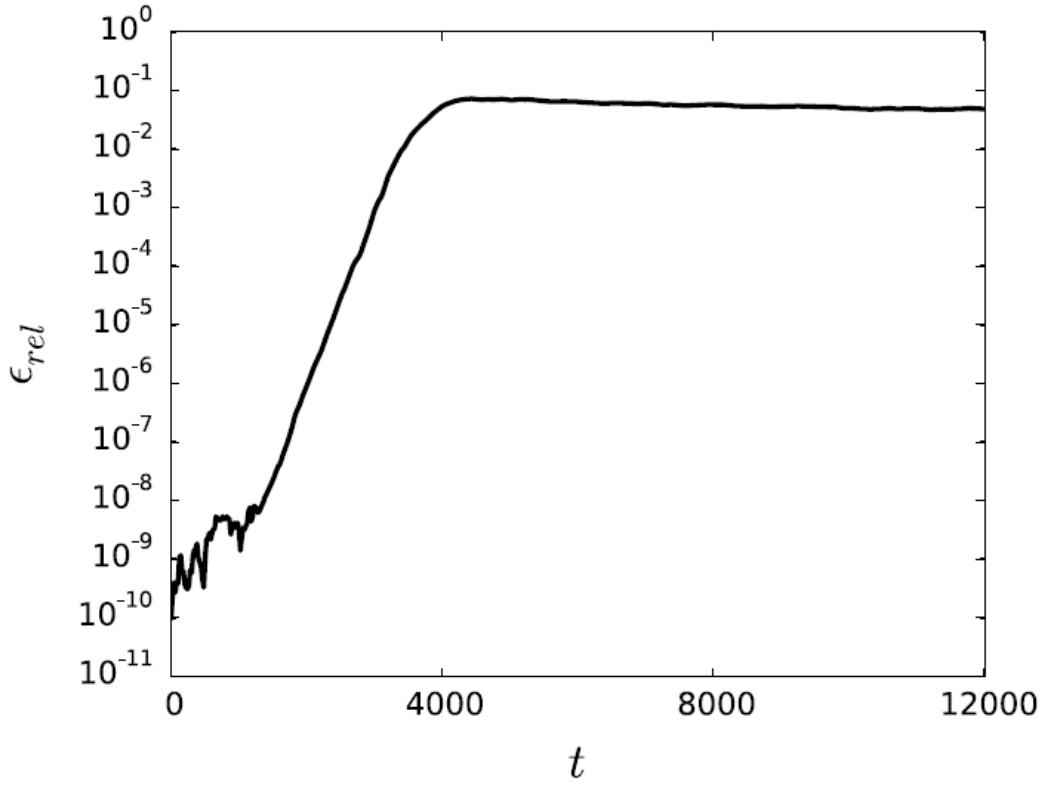
$$(-1\oplus 1)\oplus 2^{-53} \neq -1\oplus(1\oplus 2^{-53})$$

Solution for dot product: $\sum_i x_i y_i$

Long (fixed-point) **accumulators**

-> Preserves every bit of information and yields **exactly rounded result**

FELTOR provides binary reproducible scalar product



Concerns

- **Condition**

Small perturbations grow exponentially over time **physically** (turbulence): necessarily reflects in its numerical representation

➡ **Pointwise convergence** over (long enough) time is **lost**

If we need the exact result we cannot simulate beyond a certain short timespan (cf. Weather forecast)

- **Bitwise Reproducibility**

We can reproduce parallel simulations bit-by-bit, but this just hides the condition problem, **Do we need it?** Probably not, statistically reproduce

- **Accuracy**

In traditional reduction algorithms **error grows with** $\sqrt{a}$ (or worse) with a = array size; a particular concern for simulations in single precision

Conclusion

- GPUs are good for **visualisations**
- Use a **separation of concerns** approach to separate your GPU Kernels from the user
- Performance of vector operations is largely **memory bandwidth bound** -> GPUs excel due to their very fast onboard memory
- Reproducibility can be achieved bitwise with little performance overhead but in many situations **statistical reproducibility is enough** (weather forecast vs climate model)

Answer to questions

- A1: 0.5*2+0.25*4 = 1+1 = 2
- A2: 1) depends, V100 is fastest in general but, skl is slightly faster than V100 for small array sizes, 2) constant runtime (independent of size) 3) the bandwidth
- A3: skylake (CPU) because it has the fastest cache

  The CPUs have larger and faster caches but connection to RAM is slower

- A4: For large array sizes the strong scaling efficiency tends to unity