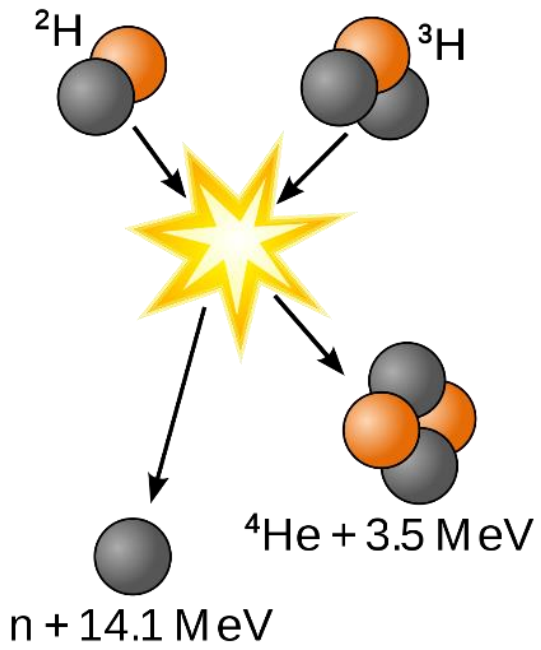
GPU programming in practice - Feltor

M. Wiesenberger PRACE winter school, Innsbruck, 11.3.2020

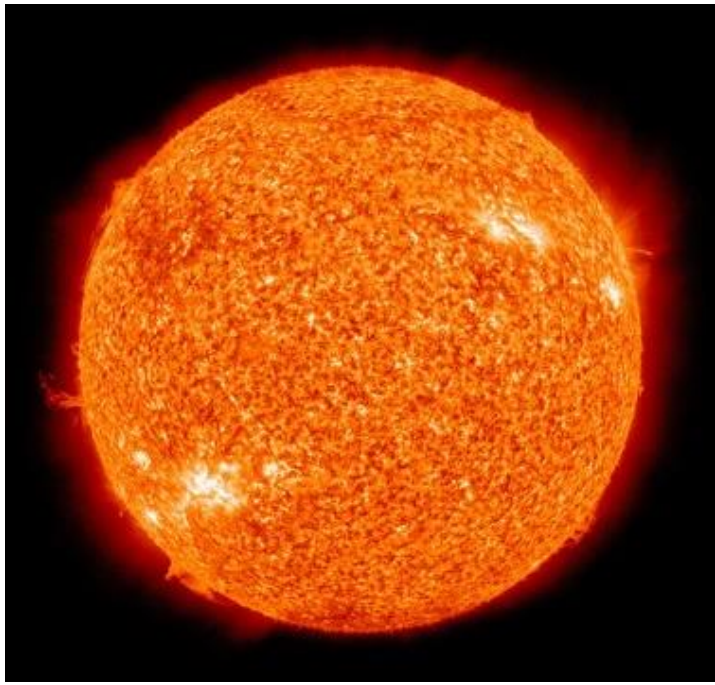
Outline

- 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**

Fusion



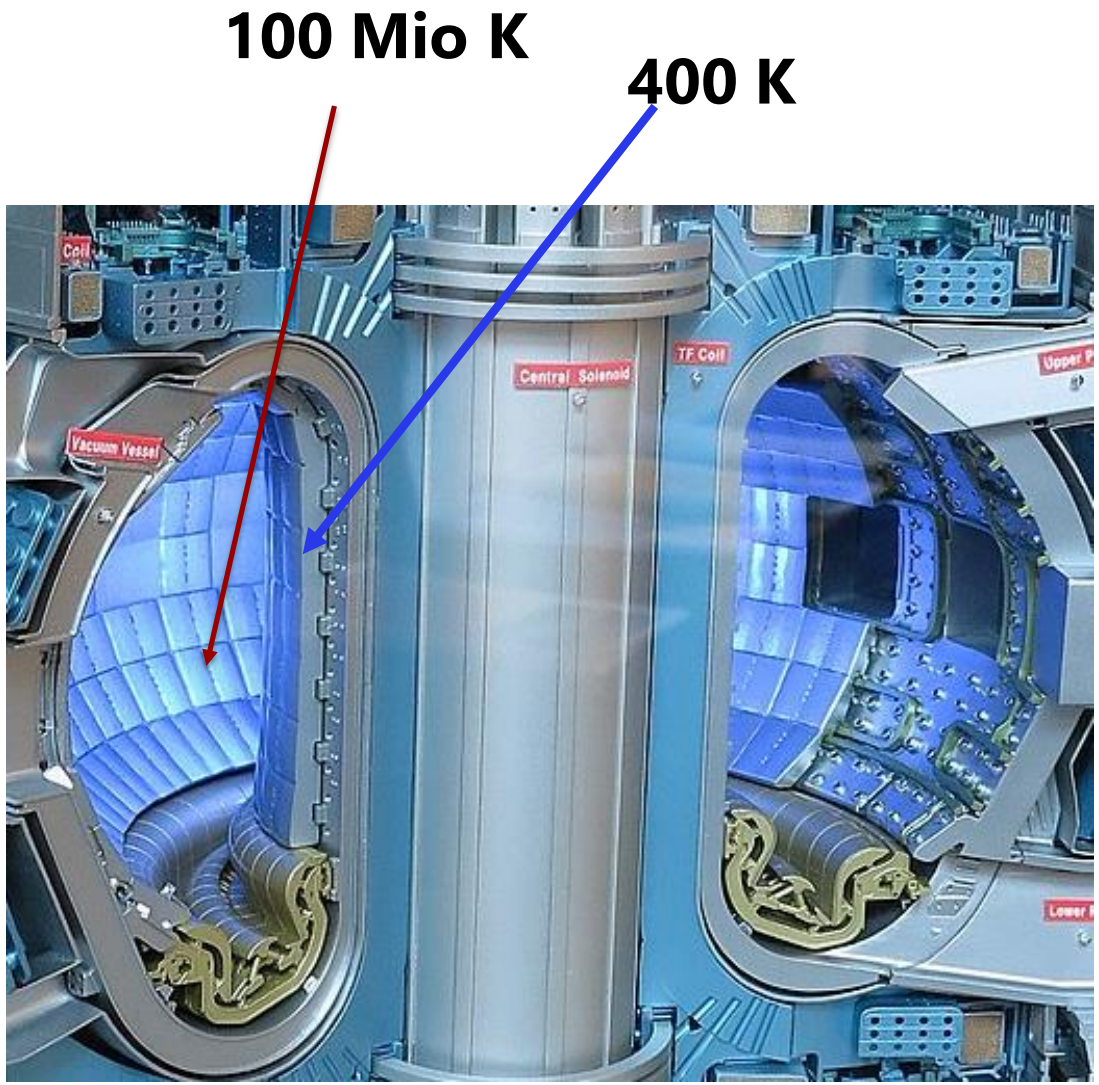
- Merge Hydrogen isotopes to He
- releases **million times more energy** than chemical reactions (coal, oil), due to nuclear vs. molecular binding energies
- Fuel abundance for **world energy production** = 4 Bio. Years with D-D fusion (~ comparable to lifetime of Earth)



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 cmoponents



Full-F (gyrofluid) 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

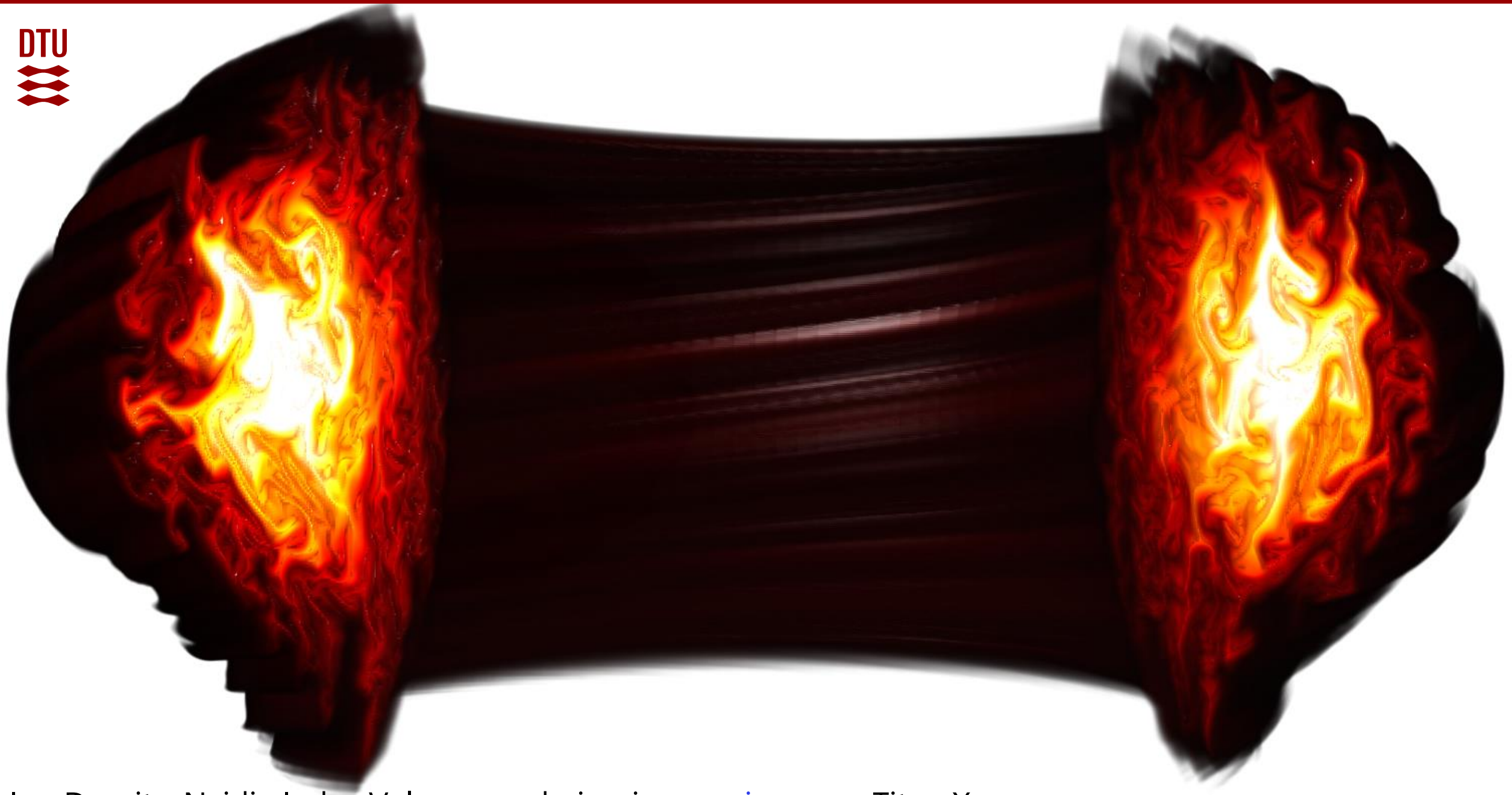
$$-\boldsymbol{\nabla}\cdot\left(\frac{N_i}{B^2}\boldsymbol{\nabla}_\perp\phi\right) = \Gamma_{1,i}N_i - n_e, \qquad \Gamma_{1,i}^{-1} = 1 - \frac{1}{2}\tau_i\mu_i\Delta_\perp \tag{99a}$$

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

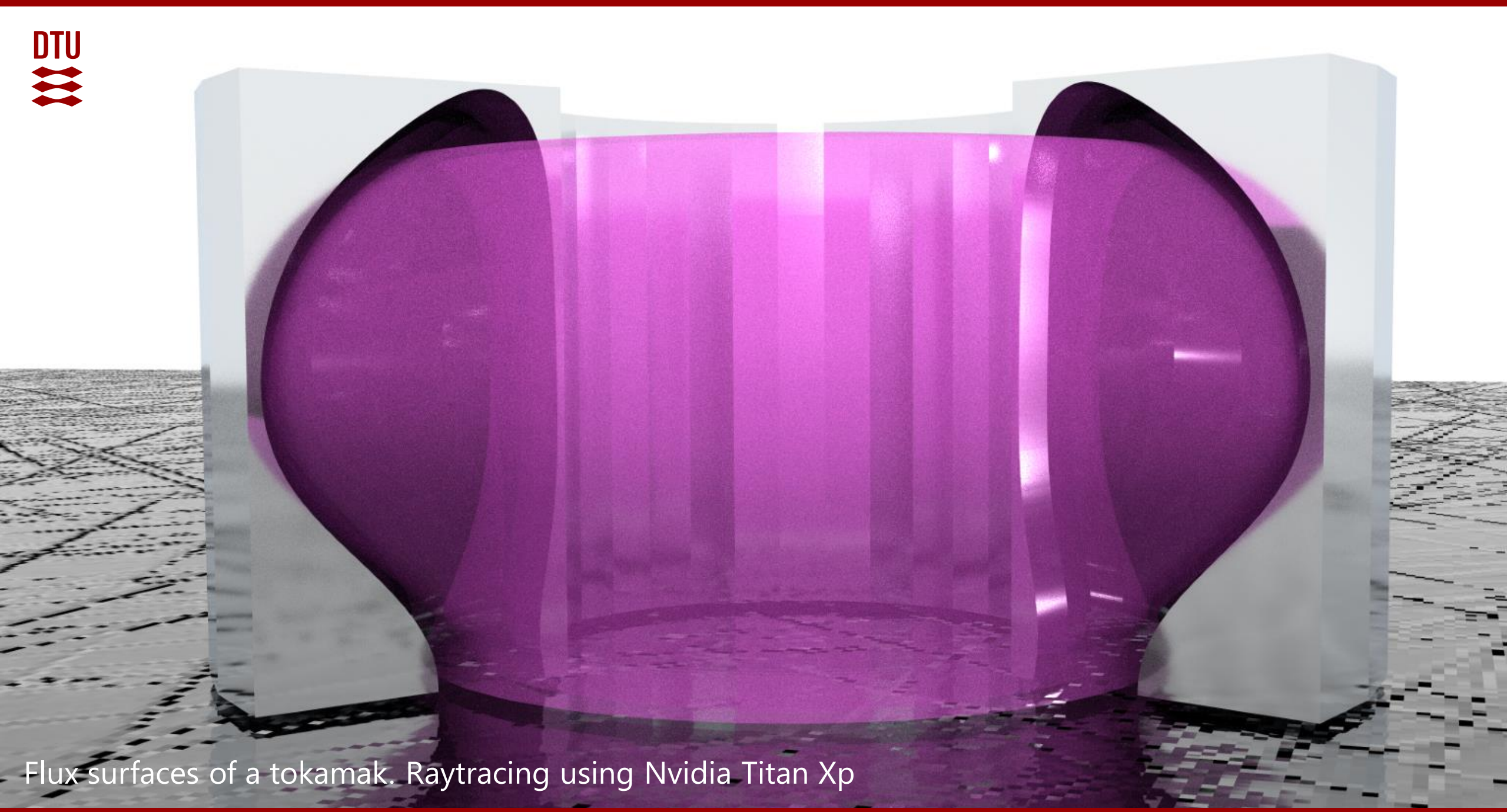
$$\left(\frac{\beta}{\mu_i}N_i - \frac{\beta}{\mu_e}n_e - \Delta_\perp\right)A_\parallel = \beta\left(N_iW_i - n_ew_e\right) \tag{99c}$$

Typical production run

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



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

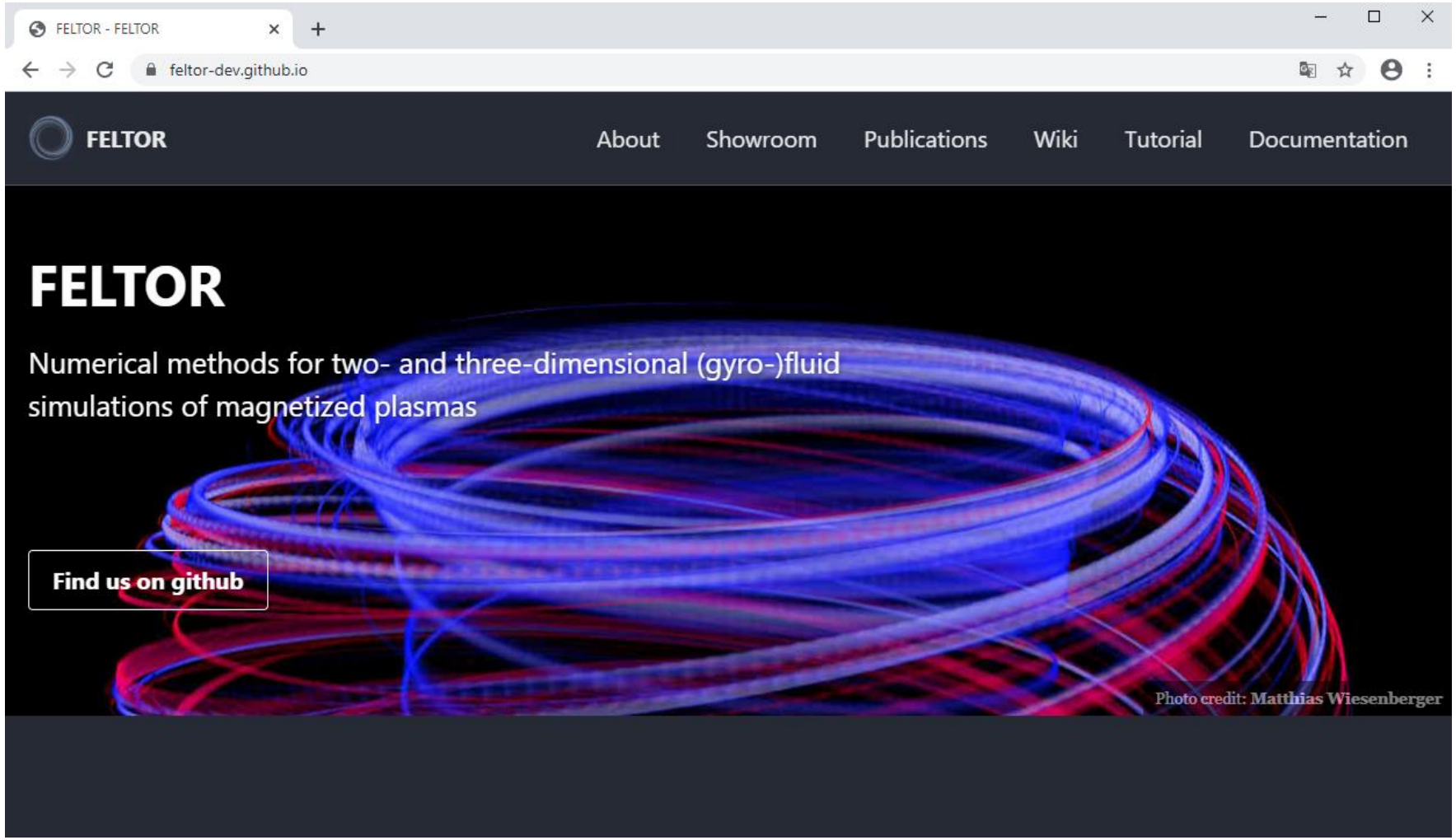


Flux surfaces of a tokamak. Raytracing using Nvidia Titan Xp

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

Electron velocity: Nvidia Index Volume rendering in paraview on a 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.

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

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

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

int main()
{

    std::array<double, 2> x ={2,2}, y = {4,4};
    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 8
    std::cout << sum << std::endl;
    return 0;
}
```

Warm-up Question:

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

```cpp
#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;
}
```

```
 9    MPI_Init( &argc, &argv);
10    MPI_Comm comm = MPI_COMM_WORLD;
11    int np,rank;
12    MPI_Comm_size(comm, &np);
13    MPI_Comm_rank(comm, &rank);
14    //allocate and initialize local memory
15    thrust::device_vector<double> x_local( 1e8/np, 2), y_local(1e8/np, 4);
16    //combine the local vectors to a global MPI vector
17    dg::MPI_Vector<thrust::device_vector<double>> x(x_local, comm);
18    dg::MPI_Vector<thrust::device_vector<double>> y(y_local, comm);
19
20    //now repeat the operations from before..
21    double a = 0.5, b = 0.25;
22    // compute a*x+b*y and store it in y
23    dg::blas1::axpby(a, x, b, y);
24    // compute Sum_i x_i y_i
25    double sum = dg::blas1::dot(x, y);
26    // output should be    large
27    if(rank==0)std::cout << sum << std::endl;
28
29    //be a good MPI citizen and clean up
30    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

Exercise

1. Explain what algorithm / problem you want to implement on GPU(s) (if you don't know take conjugate gradient (wikipedia))
2. Identify and discuss the basic (parallel) operations you need to implement your algorithm (e.g. vector addition, sorting, FFT…)
3. What advantages/disadvantages do you see?

Conjugate gradient – 3 operations

```
5   {
6       ... //initialize
7       //main loop, iteratively solve A*x = b
8       for( unsigned i=1; i<max_iter; i++)
9       {
10          blas2::symv( A, p, ap);
11          alpha =  nrmzr_old/blas1::dot( p, ap);
12          blas1::axpby( alpha, p, 1.,x);
13          blas1::axpby( -alpha, ap, 1., r);
14          if( sqrt( blas2::dot(S,r)) < eps*(nrmb + nrmb_correction))
15              return i;
16          blas2::symv(P,r,ap);
17          nrmzr_new = blas1::dot( ap, r);
18          blas1::axpby(1.,ap, nrmzr_new/nrmzr_old, p );
19          nrmzr_old=nrmzr_new;
20      }
21      return max_iter;
22  }
```

What do you not see ?

```
5   {
6       ... //initialize
7       //main loop, iteratively solve A*x = b
8       for( unsigned i=1; i<max_iter; i++)
9       {
10          blas2::symv( A, p, ap);
11          alpha =  nrmzr_old/blas1::dot( p, ap);
12          blas1::axpby( alpha, p, 1.,x);
13          blas1::axpby( -alpha, ap, 1., r);
14          if( sqrt( blas2::dot(S,r)) < eps*(nrmb + nrmb_correction))
15              return i;
16          blas2::symv(P,r,ap);
17          nrmzr_new = blas1::dot( ap, r);
18          blas1::axpby(1.,ap, nrmzr_new/nrmzr_old, p );
19          nrmzr_old=nrmzr_new;
20      }
21      return max_iter;
22  }
```

Advantages

- Code is **easy to read** (parallelization details are hidden from the user)
- **Easy to use** (no expertise on GPUs required)
- **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
- **Fast** – functions can be heavily optimized for specific architectures (it's hard to write slow code but you won't get the full performance either see Downsides)

Downsides

- 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 penality 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 (e.g. vector addition, sorting, matrix-vector multiplication …)
2. Use a library! Chances are that what you need is already implemented (and optimized)

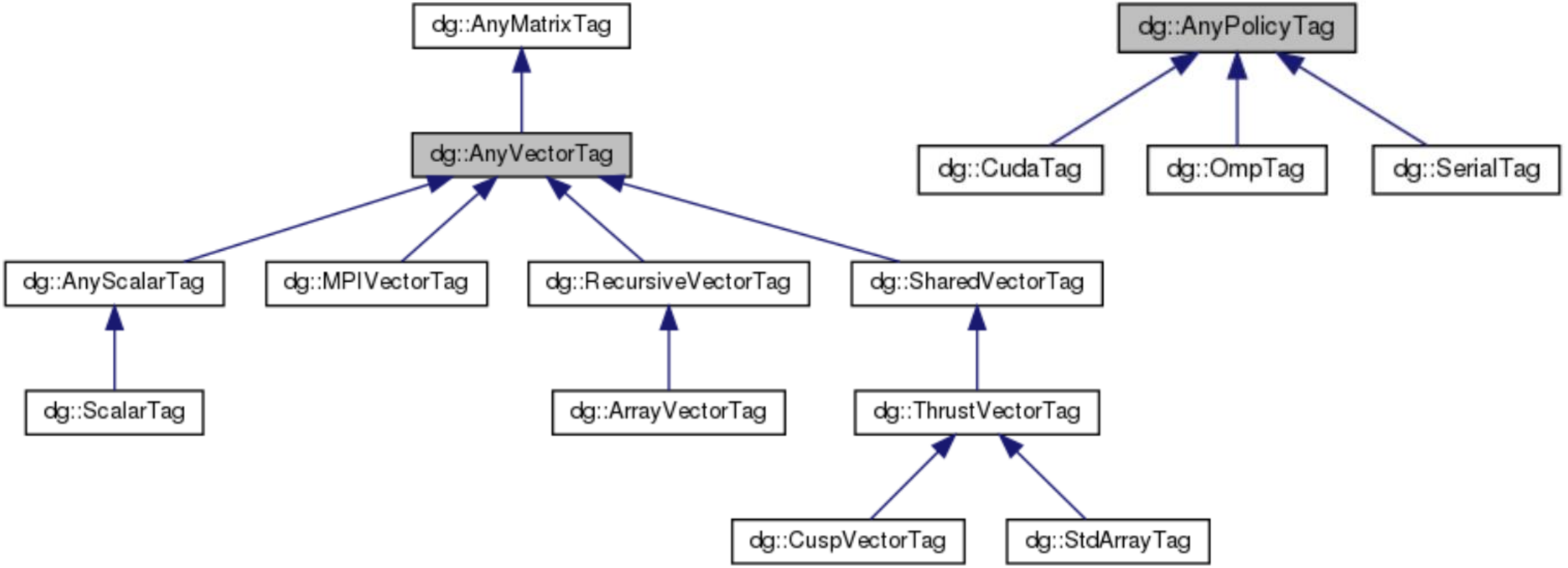
- https://github.com/thrust/thrust (included in CUDA toolkit)-> provides vector classes for memory allocation (very useful), sorting, reductions, scatter/gather, search
- https://github.com/cusplibrary/cusplibrary based on thrust, sparse matrix vector formats and preconditioners, matrix-vector multiplications,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

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

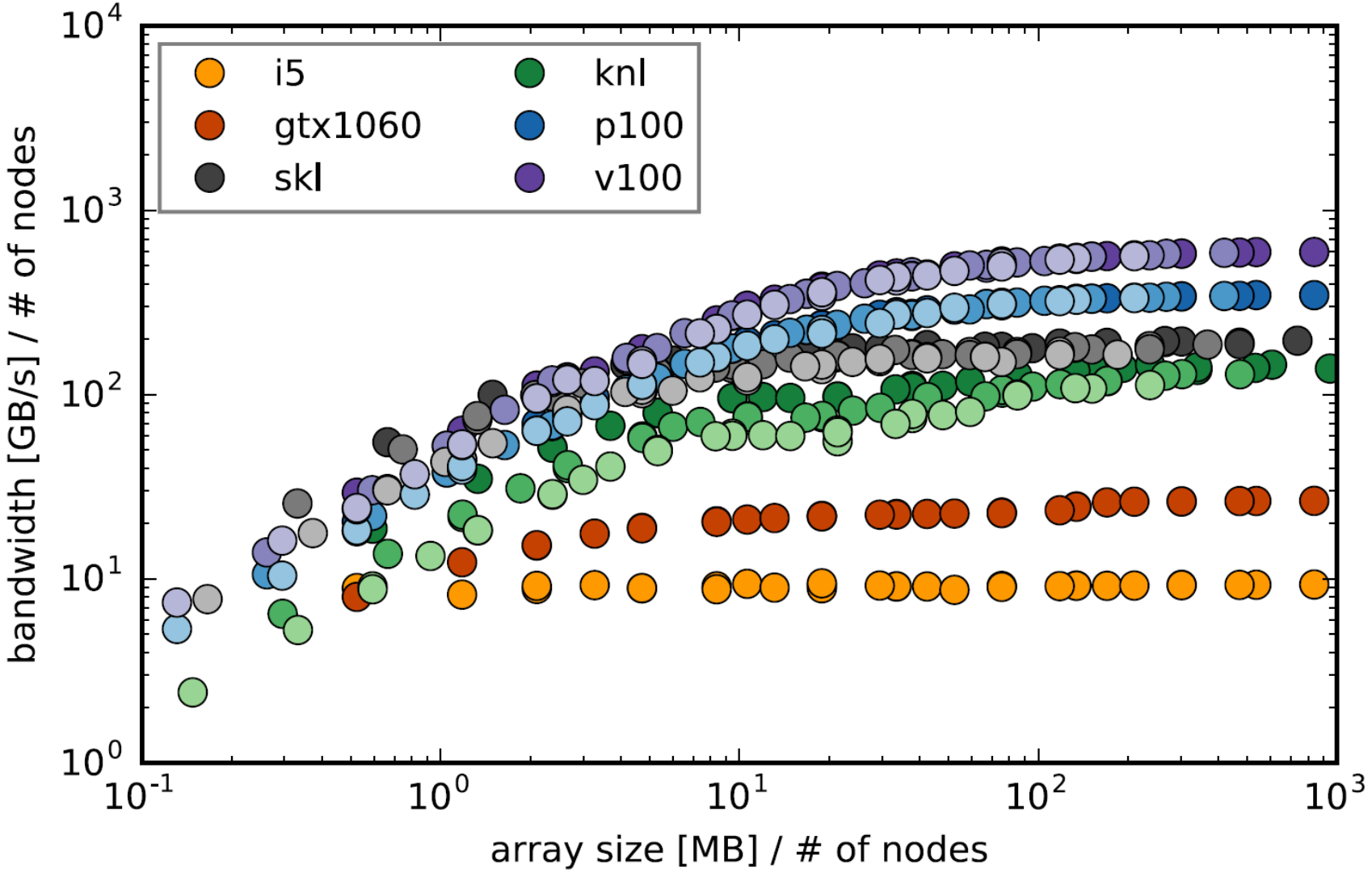
1. Identify the basic functions you need (don't make it too fine-grained, prefer functions with larger workload)
2. Implement for various architectures (i.e. write a CUDA Kernel and at least a serial version for testing)
3. Call the correct version based on the type of the input variables using a **Tag 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

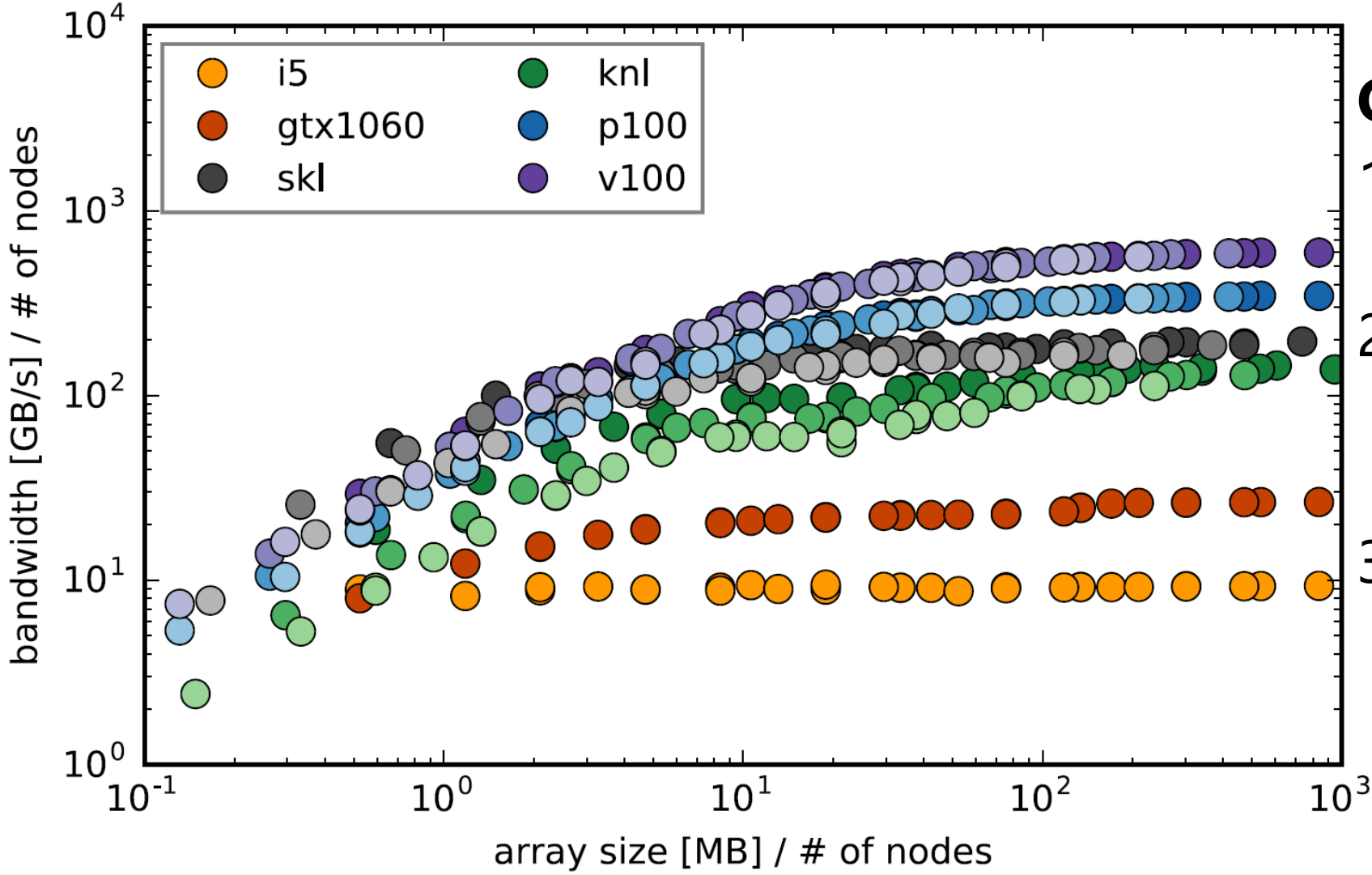
Parallelization Tag



Array size / runtime – dg::blas1::dot



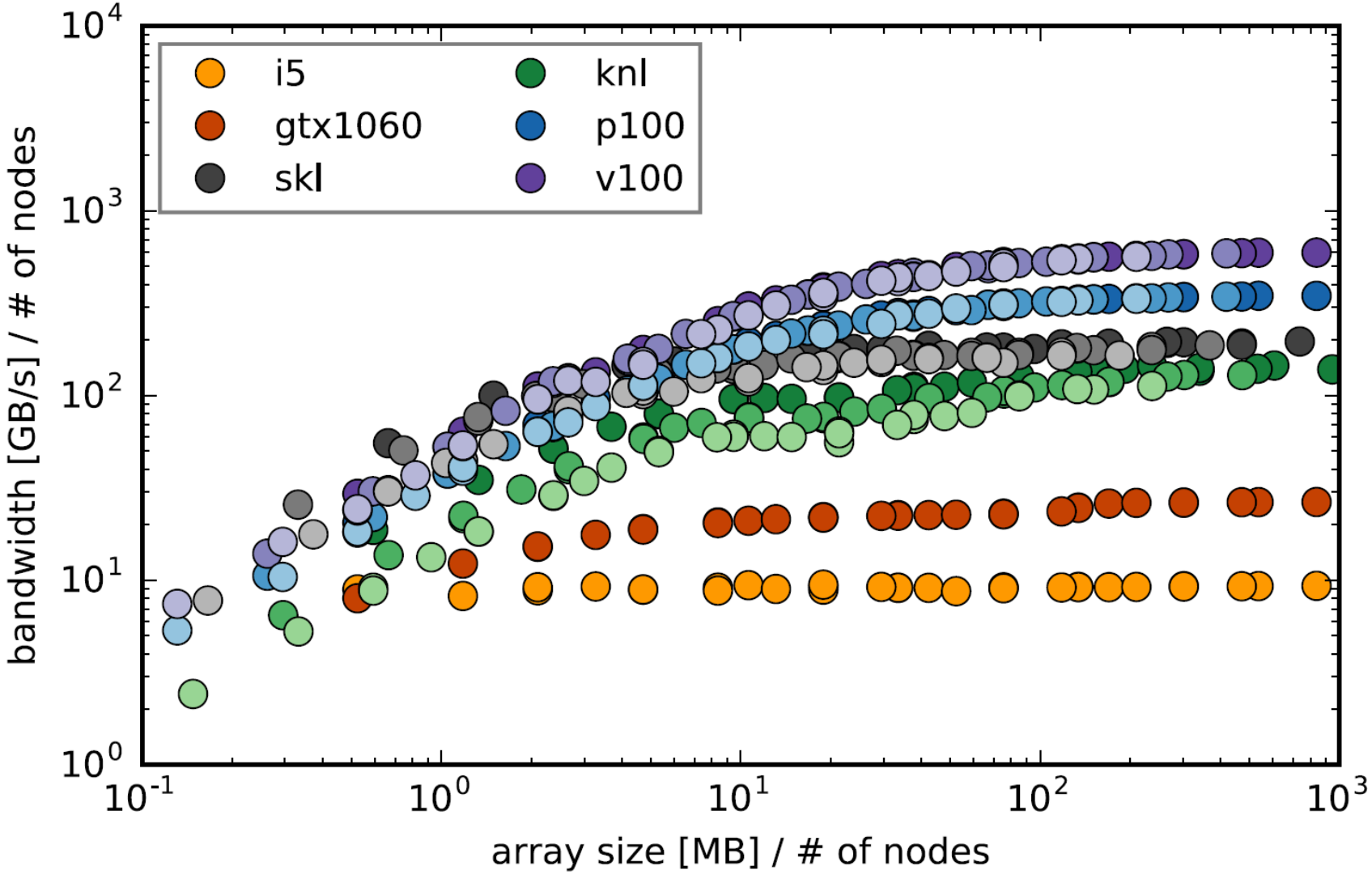
Array size / runtime – dg::blas1::dot



Questions:

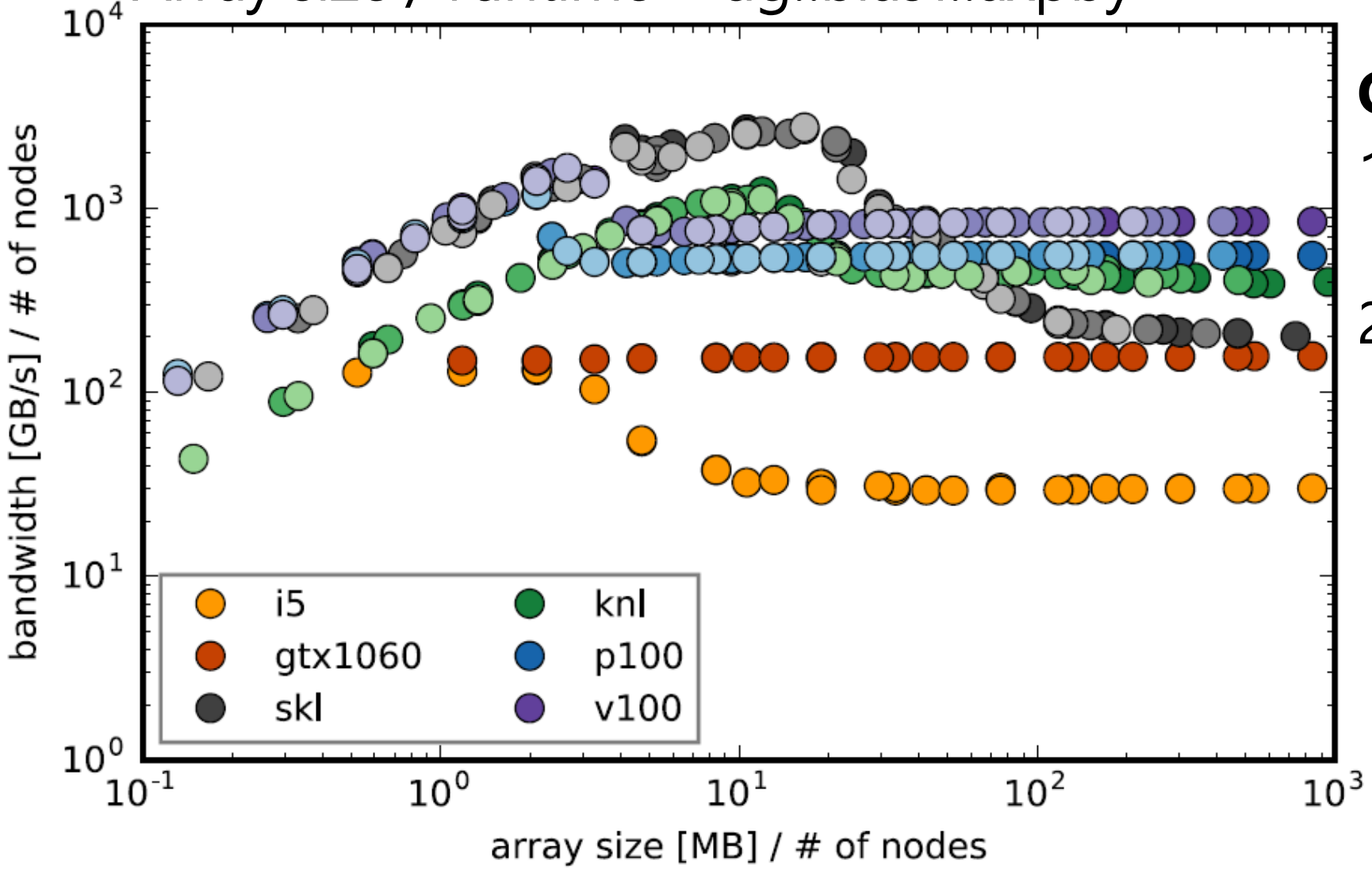
1. Which architecture is fastest?
2. What does a linear increase in throughput signify?
3. What limits the performance for large array sizes?

Array size / runtime – dg::blas1::dot



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

Array size / runtime – dg::blas1::axpby



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

	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 much faster due to their faster memory

Conclusion

- GPUs are good at 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