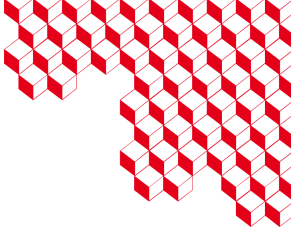
irfm



PTC - Discretization Oriented Software at Exascale

Performance portable Fourier & Spline transforms for Vlasov-Poisson numerical methods

Coordinator:
- CEA/DRF/MdlS PADIOLEAU Thomas
Partners:
- CEA/DRF/IRFM OBREJAN Kevin
- CEA/DRF/IRFM GRANDGIRARD Virginie
- CEA/DRF/MdlS BIGOT Julien
- CEA/DRF/IRFM LEGOUIX Baptiste

PTC IRFM + Maison de la simulation

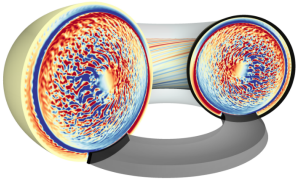
27/11/2023

PTC - Discretization Oriented Software at Exascale

1. **Overview**
   - The GyselaX++ code
   - The DDC library
   - The Vlasov-Poisson problem
2. **Change of basis function**
   - Change of basis function as DDC kernels
   - Fourier transform
   - Spline transform
   - Memory layout management
3. **Performance**
   - Evaluation of Ginkgo as a backend to Splines kernel
   - Parallelizability
   - Sequential slicing
   - Parallel slicing
   - Preconditionning
   - GyselaX++

- GYSELA is a massively parallel Fortran code developed since 2001 at IRFM in strong collaboration between physicists, mathematicians and computer scientists. *(IPP Garching, INRIA, Maison de la Simulation, EPFL, CINES...)*
- Aims to simulate turbulent transport in future magnetic confinement fusion devices.
- Provides understanding on underlying physics and permits exploration of design choices to improve the performance of future machines.

- GYSELAX++ , a complete state-of-the-art rewrite of the Fortran code in C++:
  - Adapt to exascale architectures: Support GPUs, be scalable, avoid idling of computation ressources to get closer to the performance required to simulate ITER-like plasma.
  - Proper software architecture: Benefit from modern C++ features to minimize execution-time overhead and produce modular code.
  - Handle realistic geometries & optimized meshes: Non-circular magnetic configurations, X-point, appropriate meshing in regions of interest...



- DDC is a C++ scientific computing library developed at Maison de la Simulation, aiming to support zero-overhead dimension labelling for multi-dimensional arrays and performance portable multi-dimensional algorithms.

- Describes in the same object a discrete function and the mesh on which its defined, with labeled dimensions.
- Mostly based on Kokkos (performance portability). Support major Kokkos backends (OpenMP, CUDA and HIP).
- Supports high-dimensionnal spaces and abstractize dimensions through C++ templates (managed at compile-time, zero-overhead abstractions).
- Provides convenient and optimized classes to describe basis functions (uniform or non-uniform) and change of basis kernels. ← **Object of my contribution**

Overview — The Vlasov-Poisson problem

- Minimal academic problem modeling plasma in a kinetic context:
  - Vlasov (Mechanics): $\frac{\partial f_s}{\partial t} + \frac{\vec{p}}{m_s}\frac{\partial f_s}{\partial \vec{x}} = q_s \vec{\nabla}\Phi \cdot \frac{\partial f_s}{\partial \vec{p}}$ → Semi-Lagrangian Method
  - Poisson (Electrostatics): $\Delta\Phi = -\frac{\rho}{\epsilon}$ → Spectral method or Finite Elements Method
  - Charge: $\rho = \sum_s q_s \int f_s d\vec{v}$ → Reduction

  With $f_s(\vec{x}, \vec{p}, t)$ the distribution function for species s (electrons or deuterium ions most of the time), Φ the electric potential and ρ the electric charge. Note that f_s is 6D function evolving in time in general case ! → Main reason of HPC requirement.
- **Important remark: all three numerical schemes involve change of basis function!**
  - Vlasov (advection) is solved using Semi-Lagrangian method which involves interpolation, which is itself a combination of basis changes ($\delta_1 \to \mathcal{B} \to \delta_2$).
  - Poisson can be solved using:
    - Spectral method: using Fourier basis function (see next slide).
    - Finite Elements Method over a function space.
  - Charge computation is numerical integration over velocity space, which is no more than a sum of basis function coefficients.

⇒ **Changes of basis functions are centrals in Gysela and optimizing the numerical tool which performs them is strategic!**

Interfacing numerical tools with DDC requires to *think* the mathematical objects & operations abstractly:

- DDC provides types to describe dimensions (and build spaces), discrete spaces and discrete functions. **Basis functions are themselves spaces** (ie. Fourier space) and are thus described as such.
- Usual discrete functions representation (values of the function at each point of the mesh) is the same as any other: set of coefficients in the particular basis function of Dirac impulses δ. **In some sense we generalized the concept of mesh**.
- Underlying data structures are managed by DDC and accessed through its API.
- Change of basis functions are implemented as DDC kernels and are build on well-recognized computation libraries (ie. cuFFT or Ginkgo), with transparent portability.

Computation of the discrete Fourier transform with:

$$c_k =< f|e^{ikx} >= \sum_j f(x_j)e^{ikx_j}\ dx_j$$

Is a $O(n^2)$ problem. It can be reduced to $O(n\ log(n))$ with the fast Fourier transform algorithm. Optimized implementations up to $3D$ are available in **FFTW**, **cuFFT** and **hipFFT**.

- Convenient bridges to those libraries taking DDC objects as arguments have been implemented in DDC, in a similar fashion to Kokkos spirit.
- It also computes the spectral mesh (discrete Fourier basis function) in the Fourier space from the spatial mesh.
- It is used in a simple version of the GyselaX++'s Poisson solver (spectral method):

$$\Delta\Phi = -\frac{\rho}{\epsilon} \quad \Rightarrow \quad \tilde{\Phi} = \frac{1}{\epsilon||\vec{k}||^2}\tilde{\rho}$$

Change of basis function — Spline transform

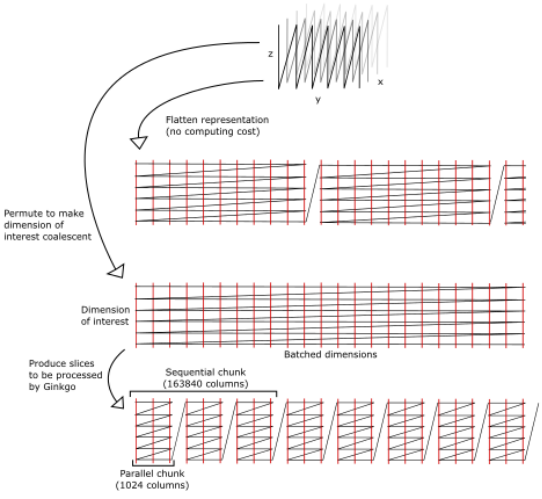
Piecewise polynomial basis function with very good derivability properties (Spline of degree d is $\mathcal{C}^{d-1}$).
1D Splines computation is a multiple-rhs quasi-band linear problem $AX = B$:

$$\begin{pmatrix} * & * & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & * \\ * & * & * & 0 & 0 & \cdots & 0 & 0 & 0 & * \\ 0 & * & * & * & 0 & \cdots & 0 & 0 & 0 & * \\ 0 & 0 & * & * & * & \cdots & 0 & 0 & 0 & * \\ 0 & 0 & 0 & * & * & \cdots & 0 & 0 & 0 & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 & \cdots & * & * & * & * \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & * & * & * \\ * & * & * & * & * & \cdots & * & * & * & * \end{pmatrix} \begin{pmatrix} c_{11} & \cdots & c_{1n_b} \\ c_{21} & \cdots & c_{2n_b} \\ c_{31} & \cdots & c_{3n_b} \\ c_{41} & \cdots & c_{4n_b} \\ c_{51} & \cdots & c_{5n_b} \\ \vdots & & \vdots \\ c_{(n-3)1} & \cdots & c_{(n-3)n_b} \\ c_{(n-2)1} & \cdots & c_{(n-2)n_b} \\ c_{(n-1)1} & \cdots & c_{(n-1)n_b} \\ c_{n1} & \cdots & c_{n\ n_b} \end{pmatrix} = \begin{pmatrix} f_{11} & \cdots & f_{1n_b} \\ f_{21} & \cdots & f_{2n_b} \\ f_{31} & \cdots & f_{3n_b} \\ f_{41} & \cdots & f_{4n_b} \\ f_{51} & \cdots & f_{5n_b} \\ \vdots & & \vdots \\ f_{(n-3)1} & \cdots & f_{(n-3)n_b} \\ f_{(n-2)1} & \cdots & f_{(n-2)n_b} \\ f_{(n-1)1} & \cdots & f_{(n-1)n_b} \\ f_{n1} & \cdots & f_{n\ n_b} \end{pmatrix}$$

With X and B matrices of typical sizes $n_{row} \sim 1000$ and $n_{cols} \sim 10^7$.

- Requires performant appropriate solver.
- Columns of X and B correspond to independant linear systems → **very suitable for parallelization**.
- Memory layouts must be managed with great attention.

- Memory layout permutation & slicing algorithm to format the data.
- Multiple linear solvers are called sequentially and in parallel to process small chunks of data.
- Aims to comply to Ginkgo constraints (*right* layout, maximum size limit) and improve memory access.



Performance — Evaluation of Ginkgo as a backend to Splines kernel

We compare the new GPU implementation based on Ginkgo library with the legacy CPU-only Lapack-based implementation:

Lapack *gbsv:

- Direct method (computes solution using triangular factorization method).
- Optimized for our problem structure (quasi-band).
- CPU-only.

Ginkgo BiCGStab:

- Iterative method (converges toward the solution).
- Appropriate for all sparse problems but cannot benefit from band structure.
- Optimized on GPU.

Benchmark test is pure advection based on characteristics method (interpolations).

100x100 (100 iterations):

Lapack Serial	5s
Ginkgo Serial	7s
Ginkgo OpenMP 160ths	2.4s
Ginkgo CUDA A100	1.2s

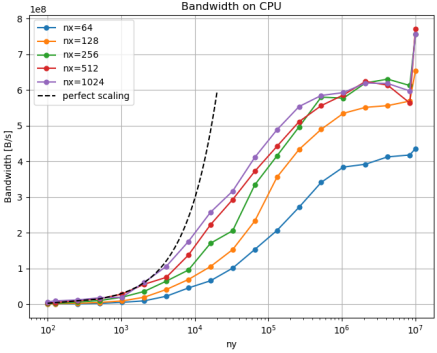
1000x10000 (100 iterations):

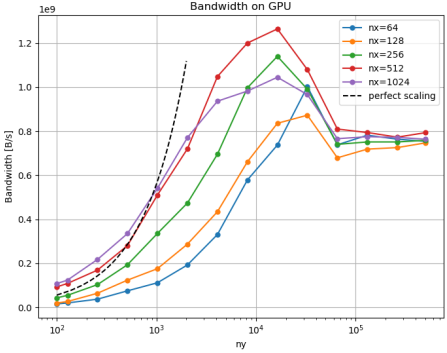
Lapack Serial	4872s
Ginkgo Serial	6941s
Ginkgo OpenMP 160ths	217s
Ginkgo CUDA A100	83s

⇒ **Lapack algorithm seems a bit more suitable than Ginkgo's for our problem ($\times 1.4$) but benefit from parallelism is obviously advantageous!**

Performance — Parallelizability

Splines on CPU and GPU with Ginkgo backend are benchmarked on pure advection problem based on characteristics method (interpolation).



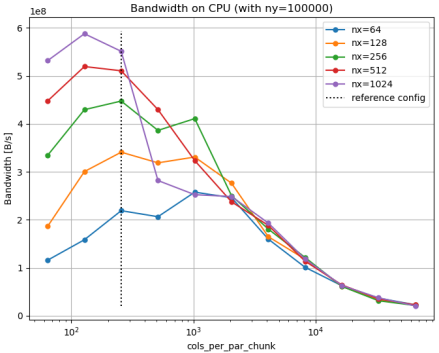


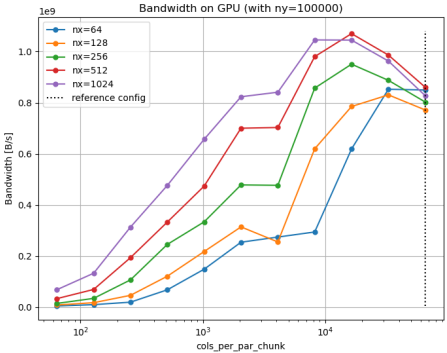
- *Bandwidth* is $n_x \times n_y \times \text{sizeof(double)}/t$.
- *Perfect scaling* is the curve for which execution time is constant while we increase the size of the problem.

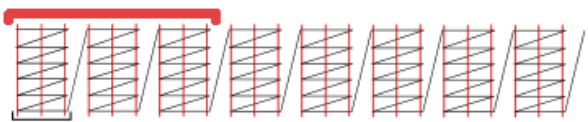
$\Rightarrow$ **GPU scales better than CPU but saturates before.**

Performance — Sequential slicing

Problem is sliced in subproblems and impact of subproblems sizes on performance has been benchmarked.

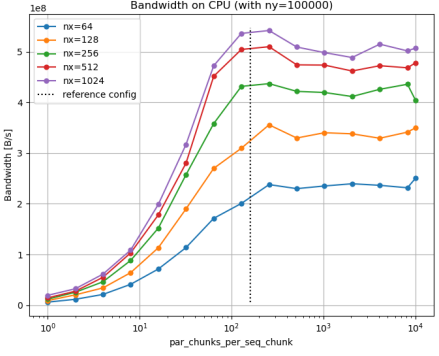


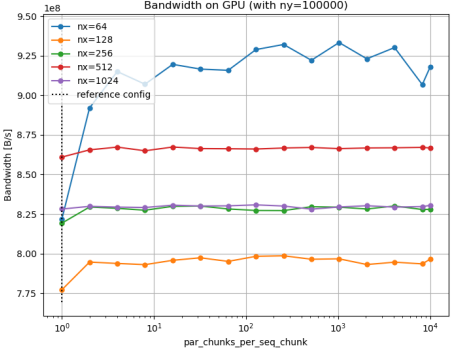


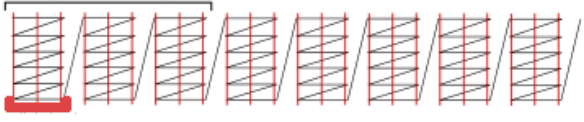


Performance — Parallel slicing

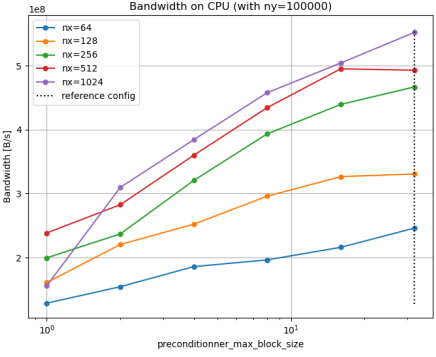
Subproblems are themselves sliced in subsubproblems called in parallel, and impact on performance has been benchmarked.

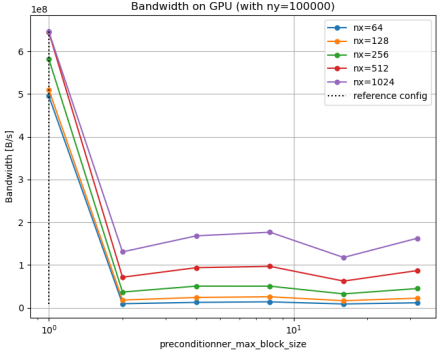






Preconditionning is beneficial for iterative methods (but quasi-band is already well-conditionned). However there is an issue with Ginkgo Block-Jacobi preconditionner on GPU which is prohibitive for performance.





⇒ **An optimal solver configuration is identified on both CPU and GPU.**

GyselaX++ contains three main modules: Vlasov and Poisson solvers, and charge computation. In the current state of the project, charge computation for LandauXYVxVy does not yet benefit from new Splines kernel and parallelism. This work will be made in coming months.

XYVxVy $64 \times 64 \times 127 \times 127$:

Lapack Serial	2846s
Ginkgo Serial	4400s
Ginkgo OpenMP 160ths	228s
Ginkgo CUDA A100	240s

	OMP	**CUDA**
Vlasov	129s	74s
Poisson	0.2s	0.08s
Charge (**CPU Serial**)	82s	146s
Others	17s	20s

Investigation is ongoing to understand why charge computation is slower when compiled with *nvcc* (CUDA compiler) whereas this is a purely CPU Serial section.

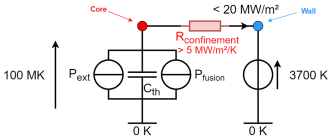
⇒ **GyselaX++ is already partially parallelized and ported to GPU! However, gain on Vlasov solver with GPU compared to CPU is not so good.**

Conclusion —

- **DDC is getting a new feature** that is very usefull for scientific computing: **change of basis functions**.
- A **performance portable FFT** has been integrated in DDC.
- **Parallellization of Gysela's spline transform** for CPU and GPU is ongoing, with good solutions **already implemented and successfully tested** in simplest cases.
- PTC roadmap:
  - Support non-periodic boundary conditions.
  - Support multidimensional splines.
  - Interface with Gysela xyv_xv_y case and validate performance.
  - Support polar splines.
  - Interface with Gysela $r\theta v_{||}v_{\perp}$ case and validate performance.
- Optimization is a long-term task that will be decisive for Gysela's capability to **make the most of exascale machines** and produce new scientific results from simulations that has never been done before.

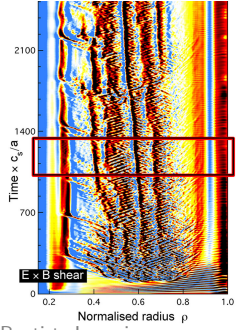
References —

- DDC: https://ddc.mdls.fr/.
- Gysela: https://gyselax.github.io/
- Virginie Grandgirard's HDR on Gysela: The GYSELA project: A semi-Lagrangian code addressing gyrokinetic full-f global simulations of flux driven tokamak plasmas.
- Emily Bourne's PhD on Splines: Non-Uniform Numerical Schemes for the Modelling of Turbulence in the 5D GYSELA Code.



- A $DT-$fuelled tokamak can be a fusion reactor only if $R_{confinement} \geq 5\ MW.m^{-2}.K^{-1}$. This heat resistance is a $0-$order model involving:
  - Geometry (but impacts on C_{th} and P_{fusion} too).
  - Magnetic field (constraint by magnets technos.).
  - Radiative transfert.
  - Collisionnal diffusion.
  - Turbulent transport $\Leftarrow$ **main channel!**

- Remark: plasma physicists prefer confinement time $\tau_c = R_{confinement} \times C_{th}$.



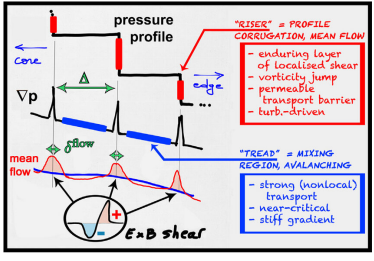


Figure: The $\mathbf{E} \wedge \mathbf{B}$ staircase of magnetised plasmas, G. Dif-Pradalier, G. Hornung, X. Garbet, Ph. Ghendrih

- Develop a **performant 2D+2D circular case** is complex enough to keep programmers busy for a while, but little interest for modern physics (no turbulence, no magnetic field gradient). This milestone should be reached in 2024.
- Next step: introduce Gyrokinetics to describe a full tokamak geometry with 5D distribution function (one dimension is removed based on the physical argument $\omega_{\mathrm{turb}} \ll \omega_{\mathrm{larmor}}$ at the cost of a very complex coordinates transformation).

Remark: all three numerical schemes involve changes of basis function !

Every (discrete) function can be described in a (discrete) Hilbert Space as a linear combination on a basis function ϕ_i:

$$f(x) = \sum_i c_i \phi_i(x)$$

f is then entirely defined by the values of c_i. We distinguish two cases:

- **Orthogonal** basis function. Defining the inner product:

$$< a|b >= \int a(x)b(x)\ dx$$

  We have $< \phi_i|\phi_j >= \delta_{ij}$. Thus, $c_i =< f|\phi_i >$ is verified (but do not necessarily provide the optimal computation method, ie. Fourier transform).
- **Non-orthogonal** basis function: requires to solve linear problems specific to chosen basis function. ie. Lagrange elements or B-splines.

Piecewise polynomial basis function with very good derivability properties (B-spline of degree d is $\mathcal{C}^{d-1}$). Leads to a **global "quasi-band"** linear problem:

$$\begin{pmatrix} * & * & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & * \\ * & * & * & 0 & 0 & \cdots & 0 & 0 & 0 & * \\ 0 & * & * & * & 0 & \cdots & 0 & 0 & 0 & * \\ 0 & 0 & * & * & * & \cdots & 0 & 0 & 0 & * \\ 0 & 0 & 0 & * & * & \cdots & 0 & 0 & 0 & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 & \cdots & * & * & * & * \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & * & * & * \\ * & * & * & * & * & \cdots & * & * & * & * \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ \vdots \\ c_{n-3} \\ c_{n-2} \\ c_{n-1} \\ c_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ \vdots \\ f_{n-3} \\ f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix}$$

With band width $w = d - 1$. Lasts rows and columns correspond to boundary conditions. It can be shown that this problem is factorizable in a "big" band linear problem followed by a "small" dense linear problem.

$\Rightarrow$ **Using a performant band linear solver is very beneficial!**

Two very different approches are considered:

Block Tridiagonal Factorization:

- Direct method (computes solution using arithmetic operations).
- Made for tridiagonal block but very appropriate for band structure.
- Hard to parallelize (but wait next slide).

⇒ **Long-term solution, probably the most performant but also the most complicated.**

BiCGStab:

- Iterative method (converges toward the solution).
- Appropriate for all sparse problems but cannot benefit from band structure.
- Optimized on GPU in Ginkgo.

⇒ **Short-term solution, already developed.**

It can be shown that if multidimensional B-splines can be written as $\phi(x, y...) = \phi_x(x)\phi_y(y)...$ then the nD change of basis is a succession of $1D$ batched change of basis, each of them corresponding to linear problems of the form:

$$\begin{pmatrix} * & * & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & * \\ * & * & * & 0 & 0 & \cdots & 0 & 0 & 0 & * \\ 0 & * & * & * & 0 & \cdots & 0 & 0 & 0 & * \\ 0 & 0 & * & * & * & \cdots & 0 & 0 & 0 & * \\ 0 & 0 & 0 & * & * & \cdots & 0 & 0 & 0 & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 & \cdots & * & * & * & * \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & * & * & * \\ * & * & * & * & * & \cdots & * & * & * & * \end{pmatrix} \begin{pmatrix} c_{11} \cdots c_{1n_b} \\ c_{21} \cdots c_{2n_b} \\ c_{31} \cdots c_{3n_b} \\ c_{41} \cdots c_{4n_b} \\ c_{51} \cdots c_{5n_b} \\ \vdots \quad \cdots \quad \vdots \\ c_{(n-3)1} \cdots c_{(n-3)n_b} \\ c_{(n-2)1} \cdots c_{(n-2)n_b} \\ c_{(n-1)1} \cdots c_{(n-1)n_b} \\ c_{n1} \cdots c_{n\ n_b} \end{pmatrix} = \begin{pmatrix} f_{11} \cdots f_{1n_b} \\ f_{21} \cdots f_{2n_b} \\ f_{31} \cdots f_{3n_b} \\ f_{41} \cdots f_{4n_b} \\ f_{51} \cdots f_{5n_b} \\ \vdots \quad \cdots \quad \vdots \\ f_{(n-3)1} \cdots f_{(n-3)n_b} \\ f_{(n-2)1} \cdots f_{(n-2)n_b} \\ f_{(n-1)1} \cdots f_{(n-1)n_b} \\ f_{n1} \cdots f_{n\ n_b} \end{pmatrix}$$

With X and B matrices of size $n_x \times n_y n_z...$ for the first system, $n_y \times n_x n_z...$ for the second etc. Typically, $n = 1000$ and $n_b \sim 10^7$.

- Columns of X and B correspond to independant linear systems → **very suitable for parallelization**.
- Memory layouts must be managed with great attention.

To solve this linear problem $AX = B$ we represent A as a block matrix which can be block-LU factorized:

$$\begin{pmatrix} Q & \gamma \\ \lambda & \delta \end{pmatrix} = \begin{pmatrix} Q & 0 \\ \lambda & \delta - \lambda Q^{-1}\gamma \end{pmatrix} \begin{pmatrix} I & Q^{-1}\gamma \\ 0 & I \end{pmatrix}$$

Leading to two distinct linear systems:

$$\begin{cases} \begin{pmatrix} Q & 0 \\ \lambda & \delta - \lambda Q^{-1}\gamma \end{pmatrix} X' = B \\ \begin{pmatrix} I & Q^{-1}\gamma \\ 0 & I \end{pmatrix} X = X' \end{cases}$$

Substitutions give:

$$\begin{cases} \begin{pmatrix} Q & 0 \\ 0 & \delta - \lambda Q^{-1}\gamma \end{pmatrix} X' = \begin{pmatrix} B_1 \\ B_2 - \lambda X_1' \end{pmatrix} \\ X = \begin{pmatrix} X_1' - Q^{-1}\gamma X_2' \\ X_2' \end{pmatrix} \end{cases}$$

Constructor

$\beta \leftarrow$ solve $Q\beta = \gamma$;

Operator()

$X_1' \leftarrow$ solve $QX_1' = B_1$;
$X_2 \leftarrow$ solve $(\delta - \lambda\beta)X_2 = B_2 - \lambda X_1'$;
$X_1 \leftarrow X_1' - \beta X_2$;

Note that solving $QX_1' = B_1$ is by far the most costly operation because of dimensions of Q.

Backup slides — Cartesian product of $1D$ B-splines (1/2)

We assume that $1D$ problems could be independently written in the forms:

$$A_1 X_1 = B_1$$
$$A_2 X_2 = B_2$$

Thus the $2D$ problem can be written:

$$(A_1 \otimes A_2)\mathrm{vec}(X) = \mathrm{vec}(B)$$

We will use four mathematical theorems:

- For two tensors T_1 and T_2:

$$T_1 \otimes T_2 = (T_1 \otimes I)(I \otimes T_2)$$

- Also:

$$(T_1 \otimes T_2)\mathrm{vec}(T_3) = \mathrm{vec}(T_2 T_3 T_1^\top)$$

- With K the commutation matrice:

$$K(T_1 \otimes T_2)K^\top = T_2 \otimes T_1$$

- We also have:

$$K^\top \mathrm{vec}(T) = \mathrm{vec}(T^\top)$$

We show that it leads to two matrix-matrix linear systems:

$$(A_1 \otimes A_2)\mathrm{vec}(X) = \mathrm{vec}(B)$$

$$(A_1 \otimes I)(I \otimes A_2)\mathrm{vec}(X) = \mathrm{vec}(B)$$

$$K(I \otimes A_1)K^\top(I \otimes A_2)\mathrm{vec}(X) = \mathrm{vec}(B)$$

$$K(I \otimes A_1)K^\top \mathrm{vec}(A_2 X I^\top) = \mathrm{vec}(B)$$

$$K(I \otimes A_1)\mathrm{vec}(X^\top A_2^\top) = \mathrm{vec}(B)$$

$$K\mathrm{vec}(A_1 X^\top A_2^\top I^\top) = \mathrm{vec}(B)$$

$$\mathrm{vec}(A_2 X A_1^\top) = \mathrm{vec}(B)$$

$$A_2 X A_1^\top = B$$

$$A_1 (A_2 X)^\top = B^\top$$

Or in indicial notations:

$$A_{1ij} A_{2lk} X_{kj} = B_{li}$$

Thus we have to solve:

$$\begin{cases} A_1 Y = B^\top \\ A_2 X = Y^\top \end{cases}$$

The slicing algorithm requires buffers which leads to GPU memory occupancy overhead: