Optimising the performance of the spectral/hp element method with collective linear algebra operations

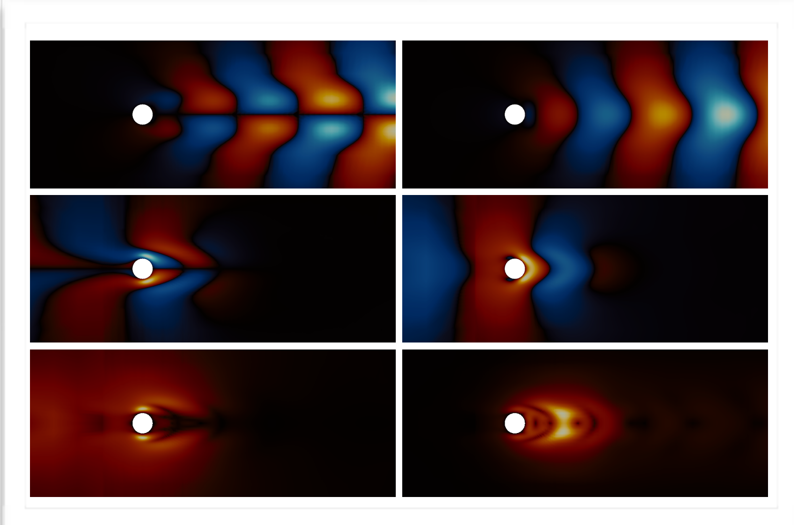
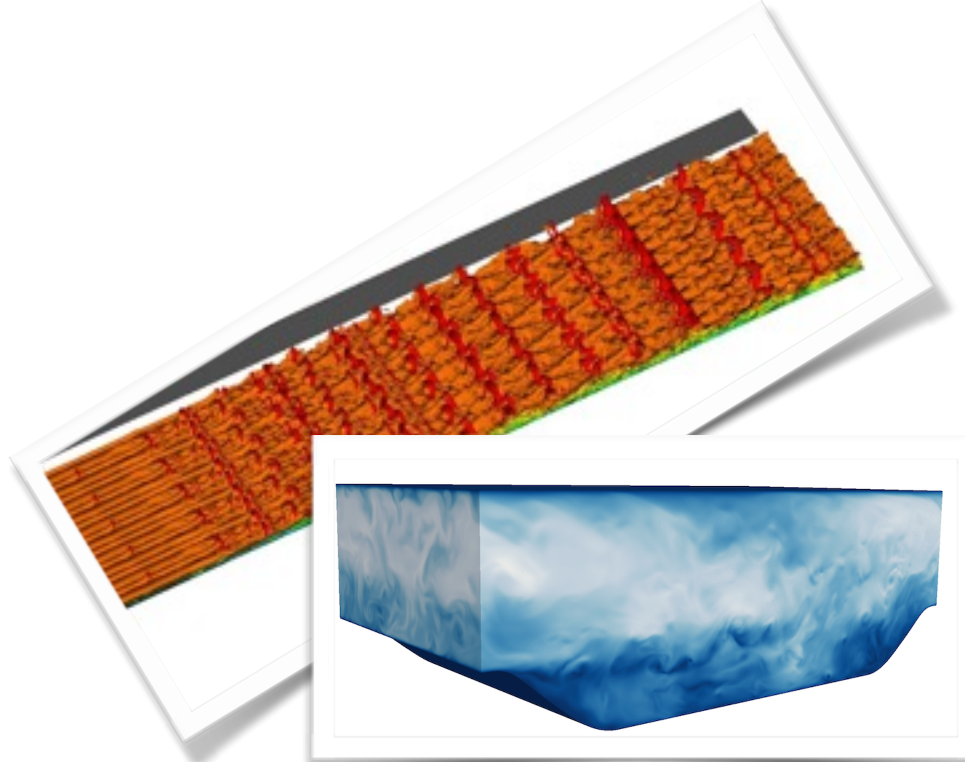
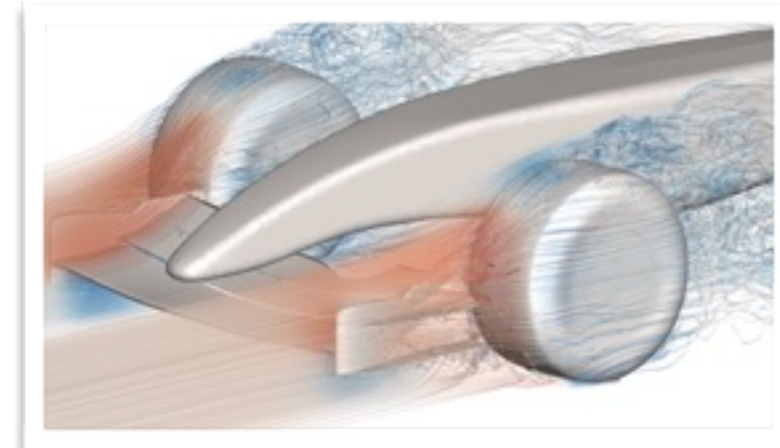
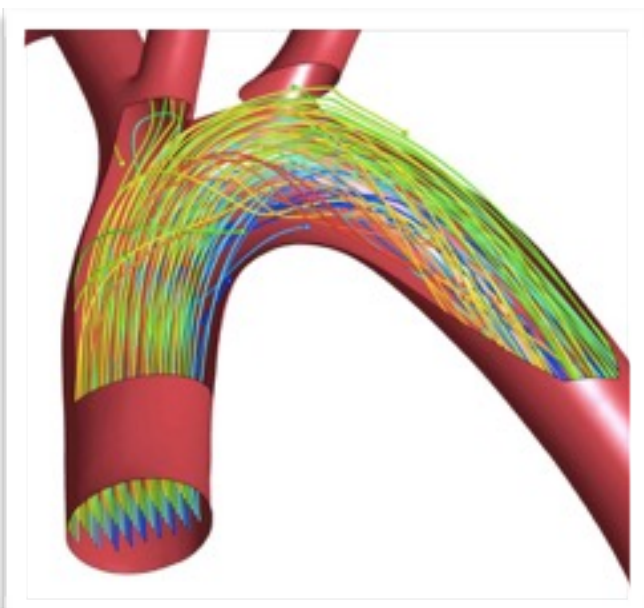
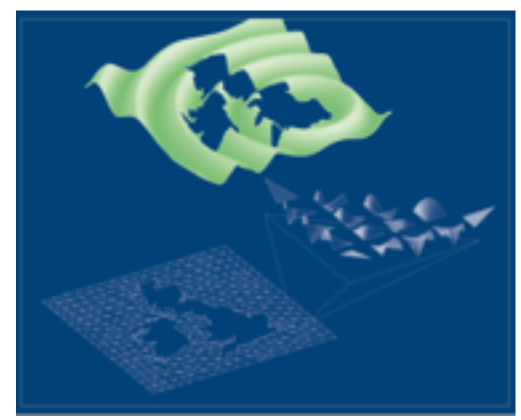
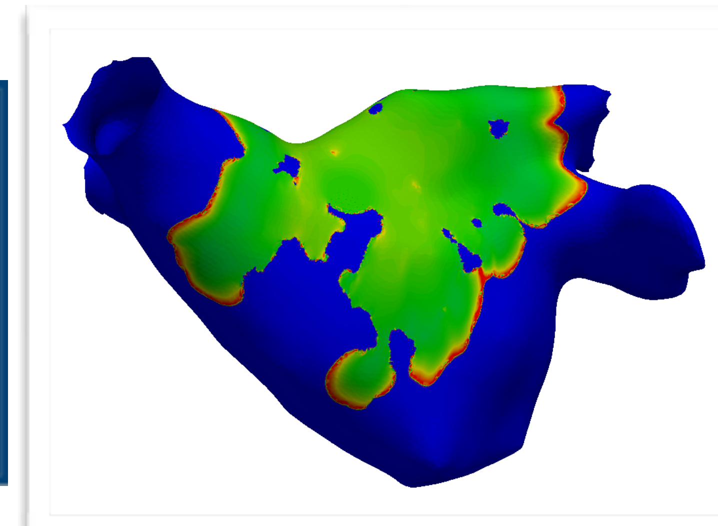
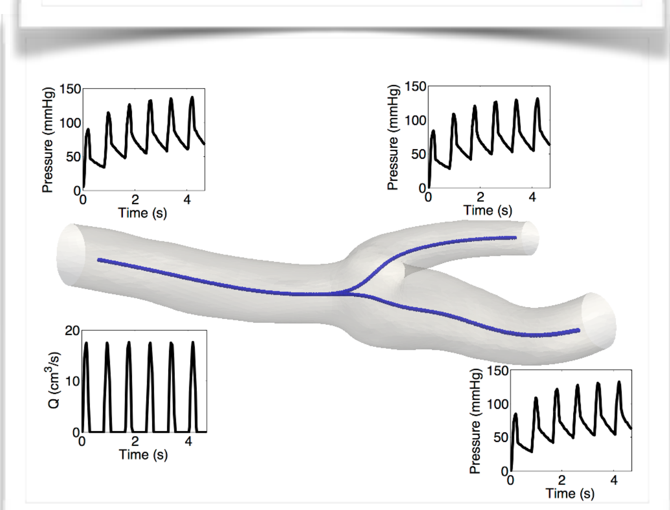
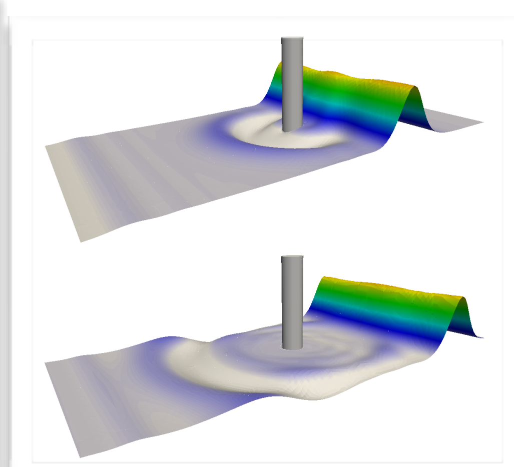
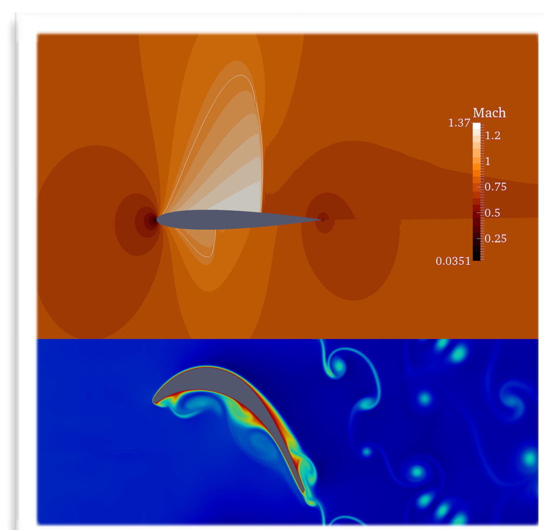
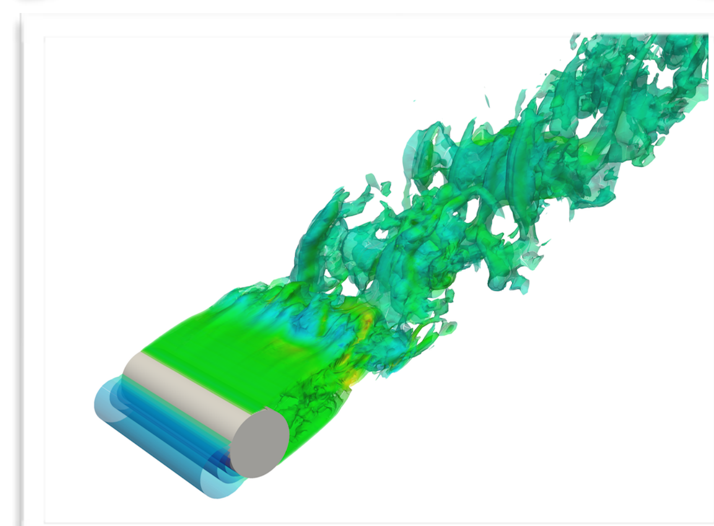
D. Moxey, C. Cantwell, S. J. Sherwin
Department of Aeronautics, Imperial College London
R. M. Kirby
Scientific Computing and Imaging Institute, University of Utah

PRISM Workshop on Embracing Accelerators
Imperial College London

18th April 2016

Outline

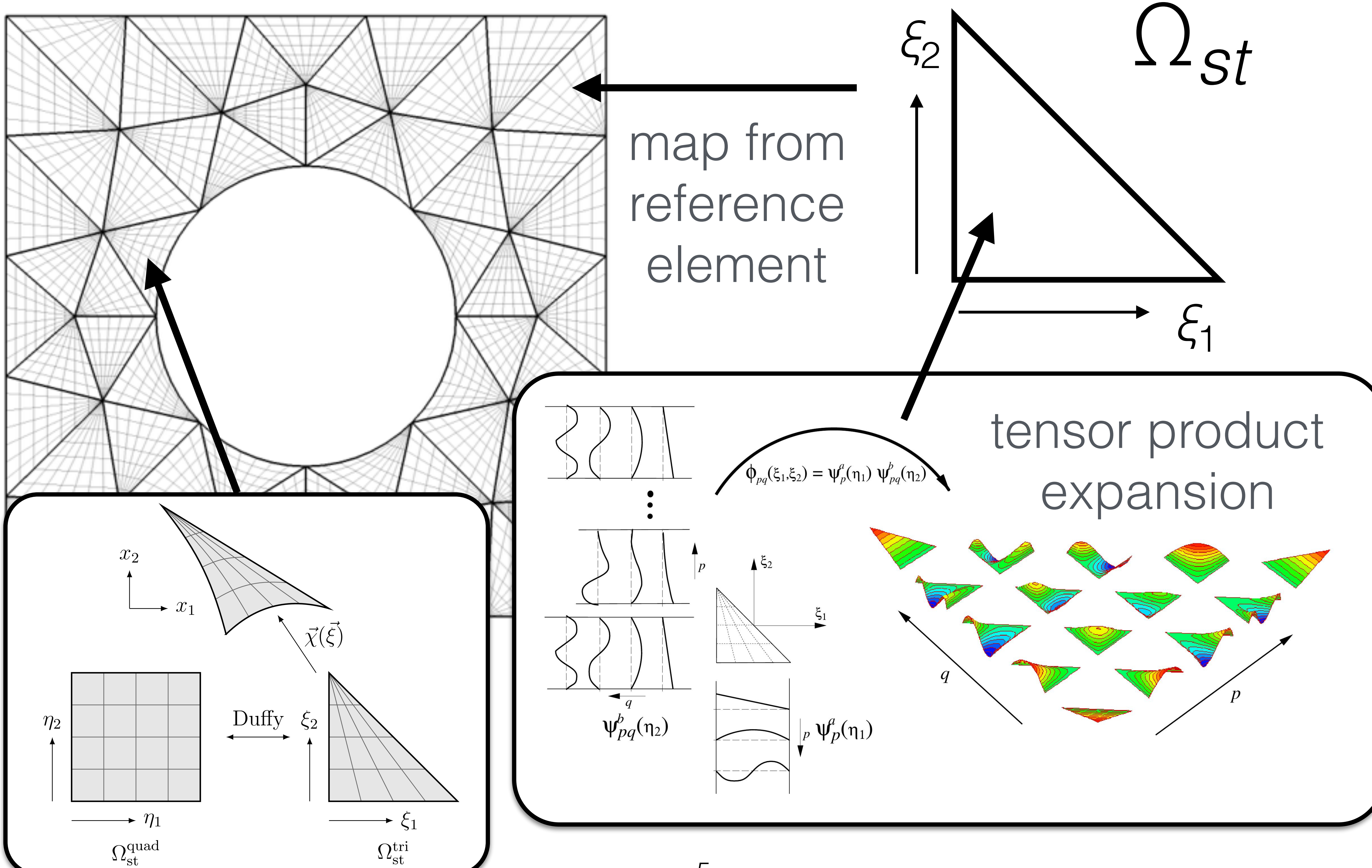
- Nektar++: brief overview and motivation
- Goals and structure
- Examples
- Conclusions








Pressure (mmHg)
150
100
50
0
0
2
4
Time (s)
Pressure (mmHg)
150
100
50
0
0
2
4
Time (s)
Q (cm^3/s)
20
10
0
0
2
4
Time (s)
Pressure (mmHg)
150
100
50
0
0
2
4
Time (s)


Mach
1.37
1.2
1
0.75
0.5
0.25
0.0351


Nektar++ goals

- Make it simpler/quicker to develop solvers for a range of fields and applications
- Support 1/2/3D and unstructured hybrid meshes for complex geometries
- Scale to large numbers of processors
- Be efficient across a range of polynomial orders and core counts
- Bridge current and future hardware diversity

Spectral/*hp* element method



Motivation

Consider the Helmholtz equation:

$$\Delta u + \lambda u = f$$

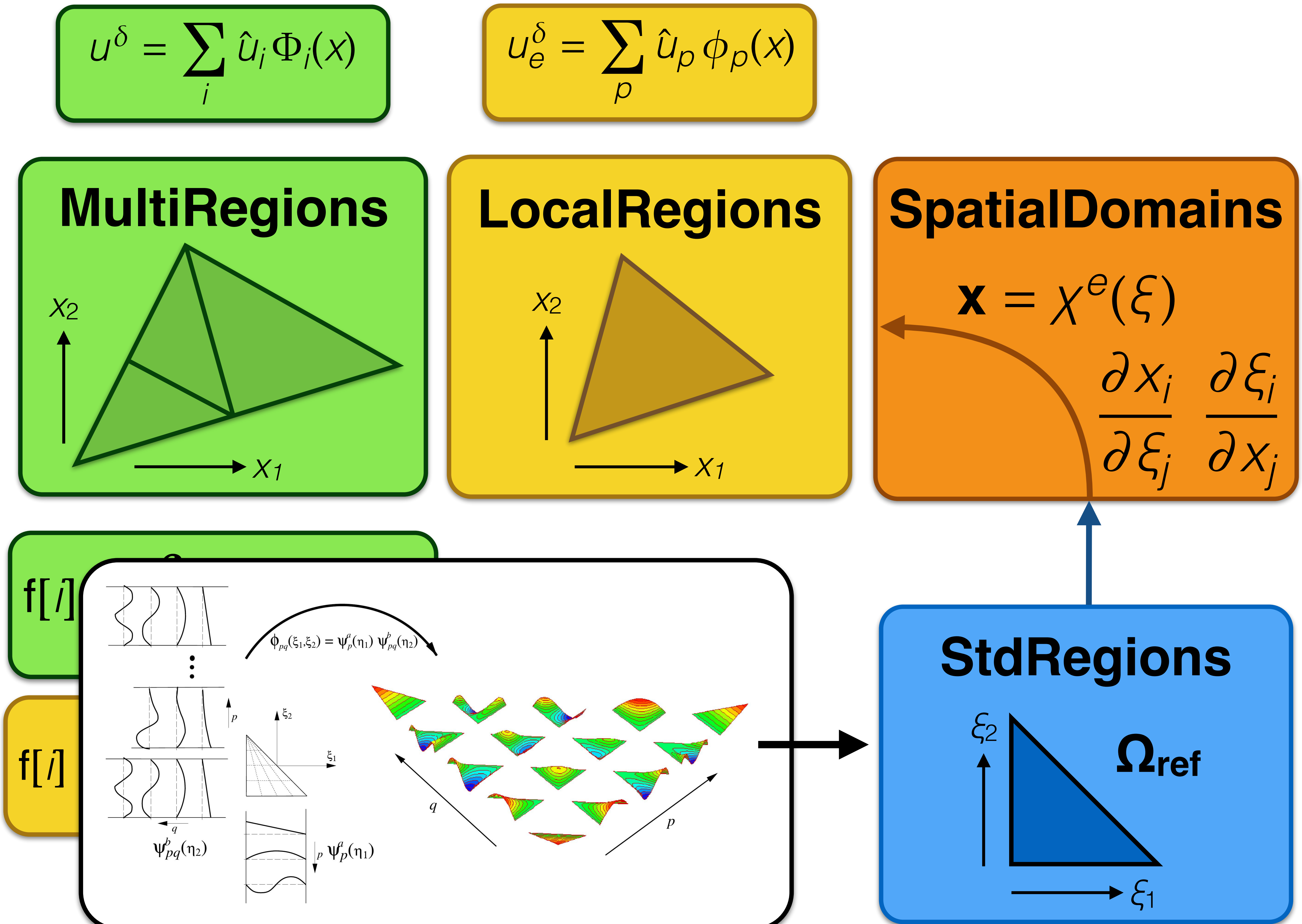
Put it into weak form:

$$-(\nabla u, \nabla v) + \lambda(u, v) + (\nabla u, v)|_{\partial\Omega} = (f, v)$$

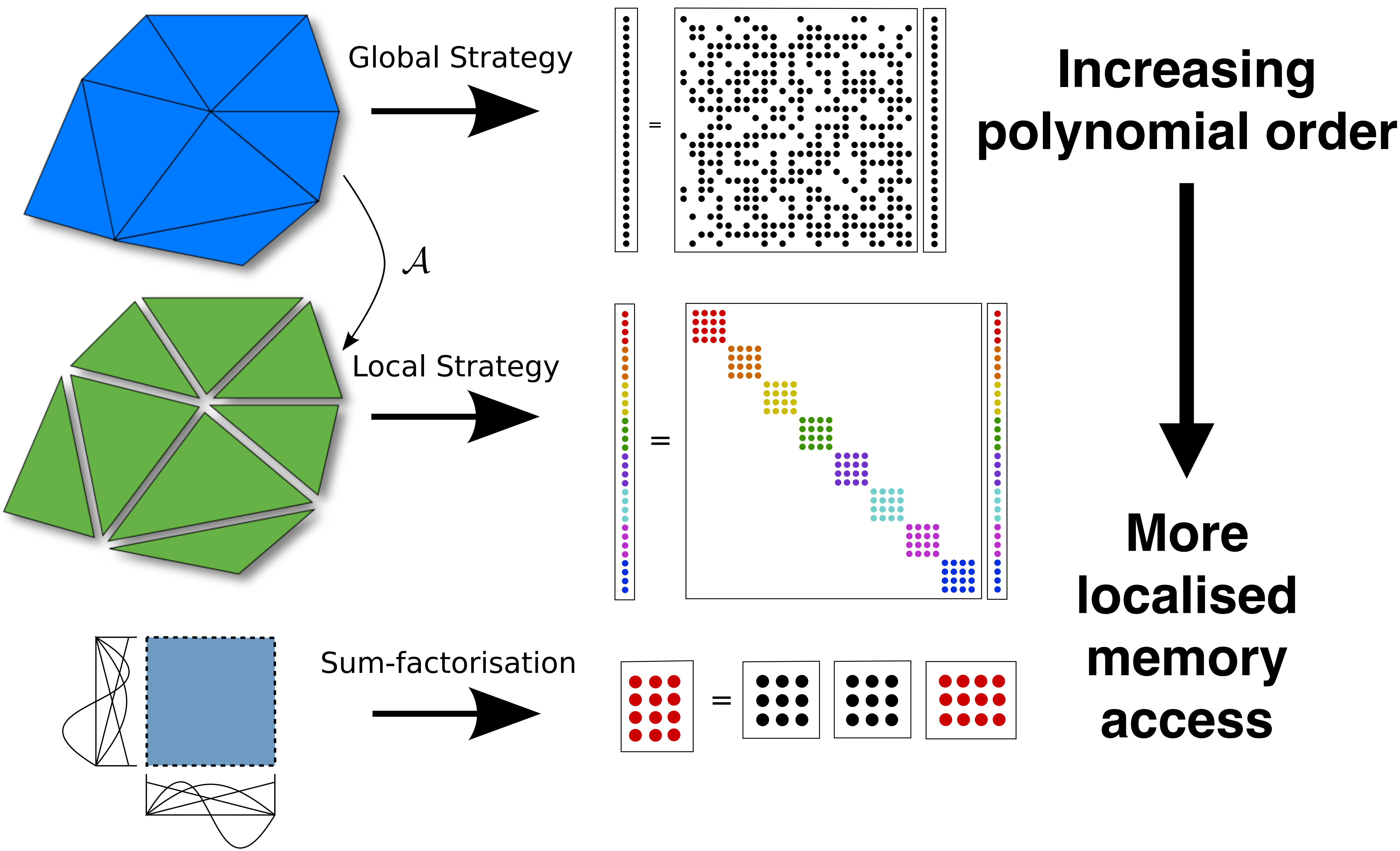
Expand u and v in terms of **local** modes (on each element) or **global** modes (on whole mesh):

$$u_e^\delta = \sum_p \hat{u}_p \phi_p(x) \qquad u^\delta = \sum_i \hat{u}_i \Phi_i(x)$$

Framework design



Implementation choices



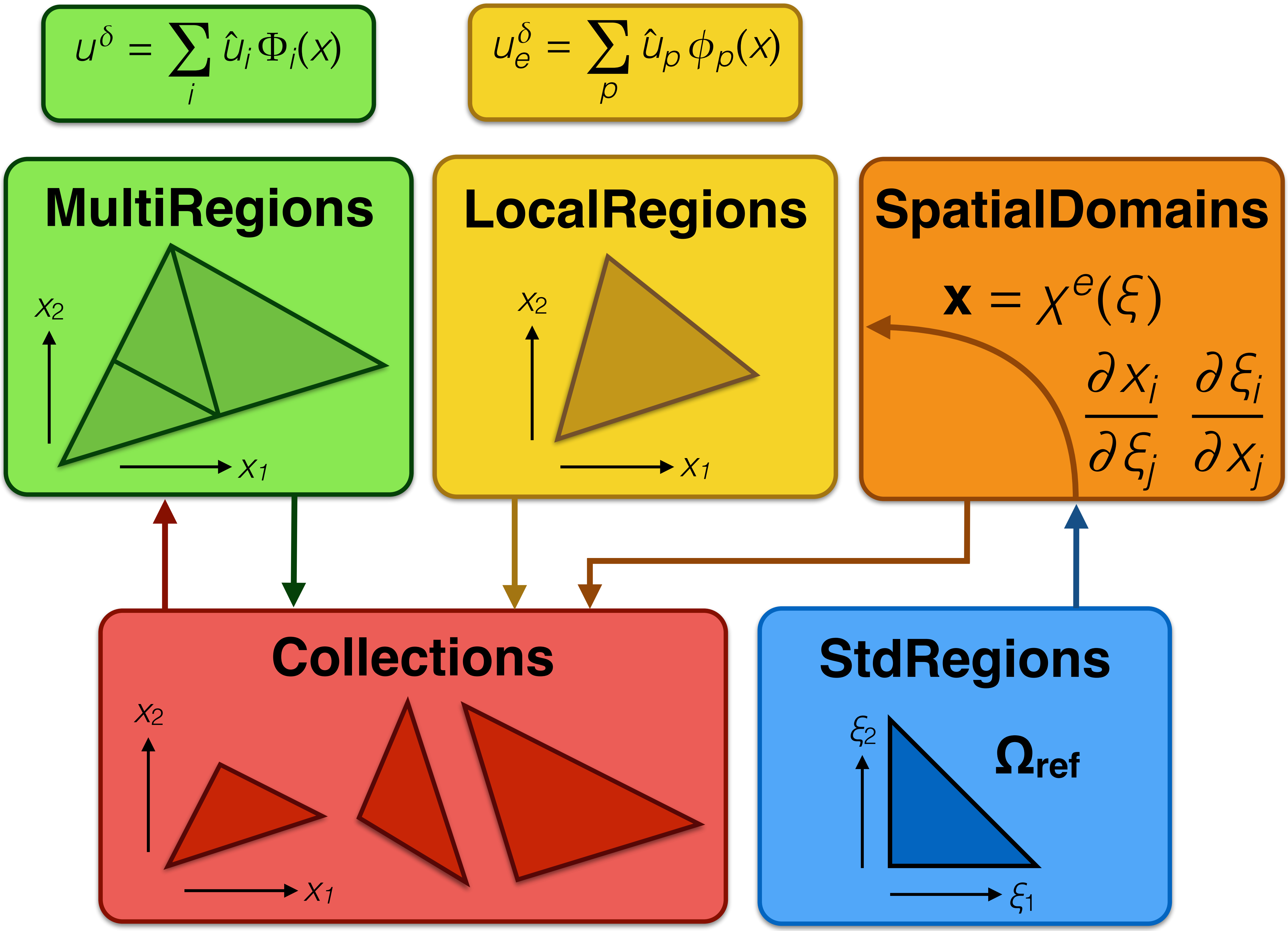
Local approaches

- These approaches give different performance results depending on variety of factors (element type, polynomial order, machine specifications...)
- Also very flexible in terms of development and allowing more advanced features (e.g. adaptivity)
- But performance is not optimal (and implementation not easy) when looking to accelerator hardware
- Needs better control over memory management

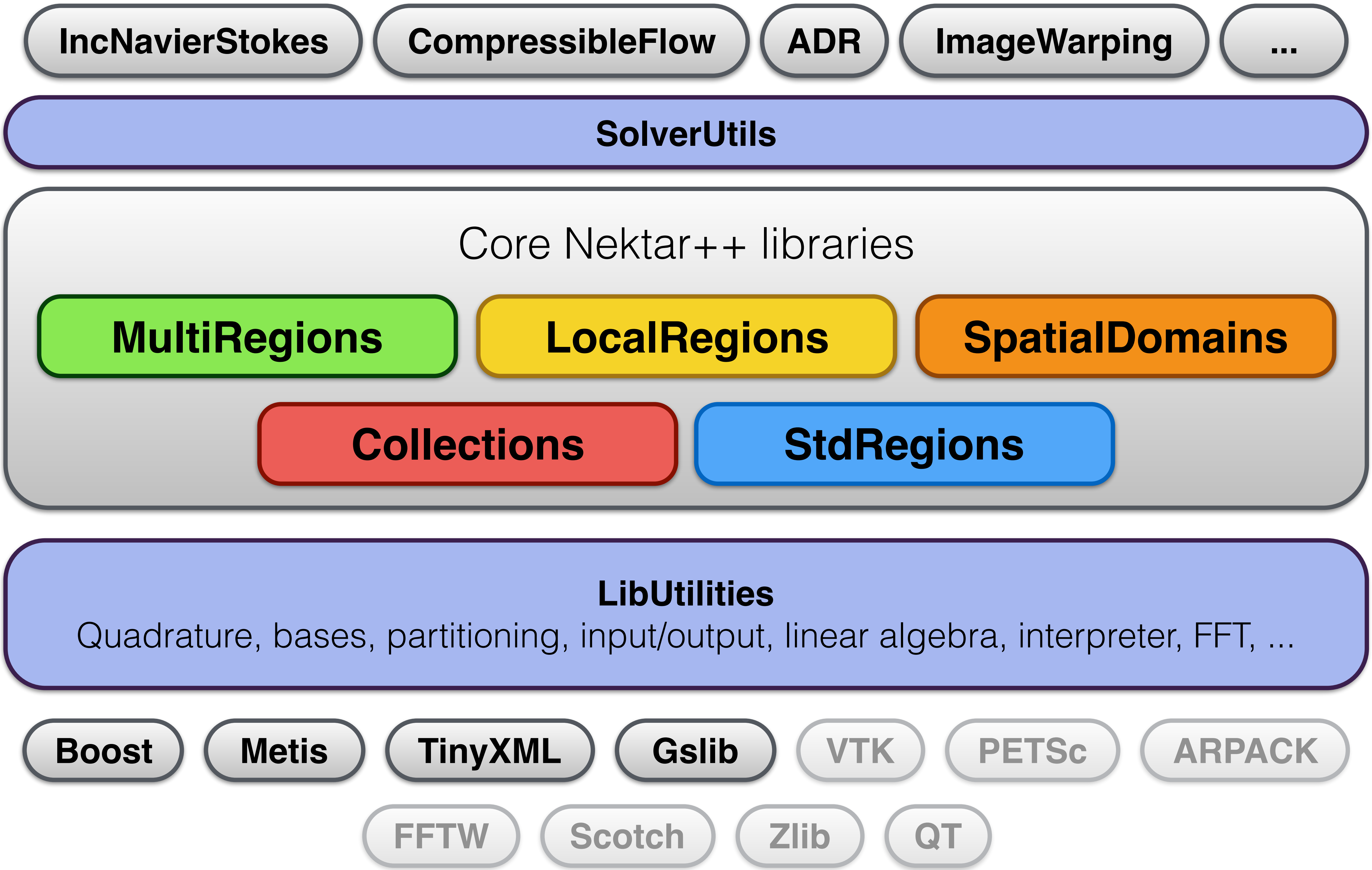
Collections

- **Main idea:** Reformulate implementation choices in terms of groups of elements
- Group geometric terms $\dfrac{\partial x_i}{\partial \xi_j}$ and apply to entire mesh
- Focus around key operators of different complexities:
  - ➡ Backward transformation: $u_e^\delta = \sum_p \hat{u}_p \phi_p(x)$
  - ➡ Inner product: (Φ_i, Φ_j)
  - ➡ Derivatives: $\partial u / \partial x_i$
  - ➡ Inner product w.r.t. derivative: $(\Phi_i, \nabla \Phi_j)$

Collections

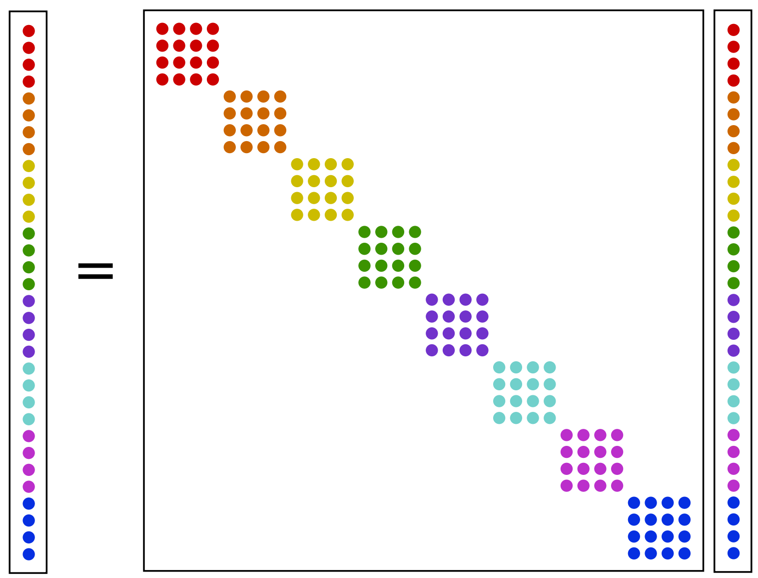


Framework design



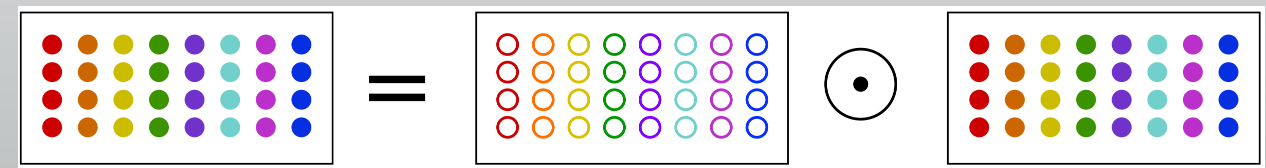
Schemes

Local Matrix

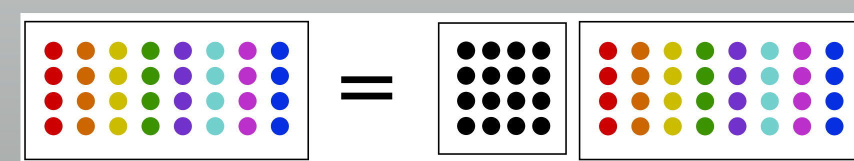


StdMat (standard matrix)

1. Apply Jacobian **(L1)**

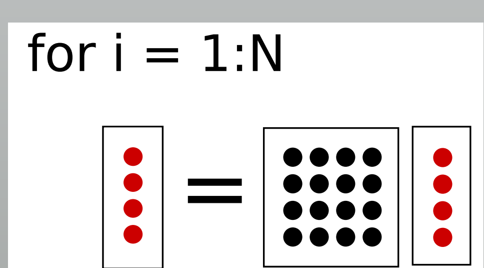


2. Multiply by ref. matrix **(L3)**



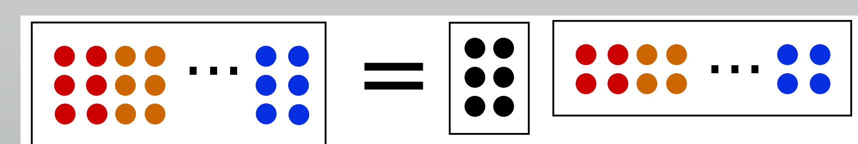
IterPerExp

1. Apply Jacobian **(L1)**
2. Multiply by ref. matrix **(N x L2)**

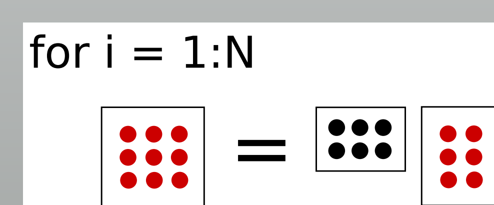


SumFac

1. Apply Jacobian **(L1)**
2. Mult. first dimension **(L3)**

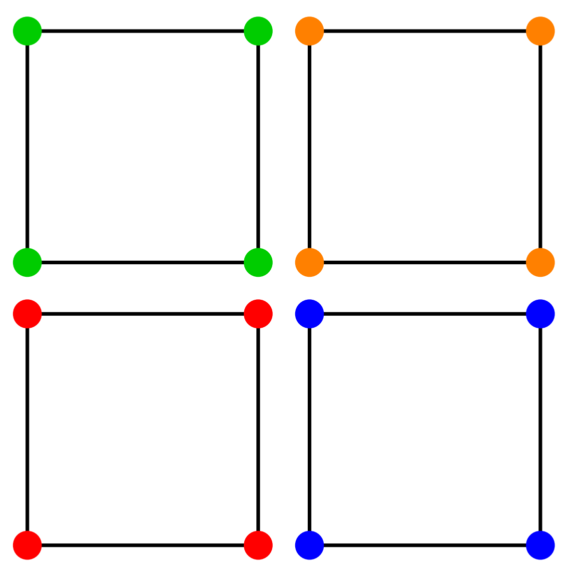


3. Mult. second dimension **(N x L2)**

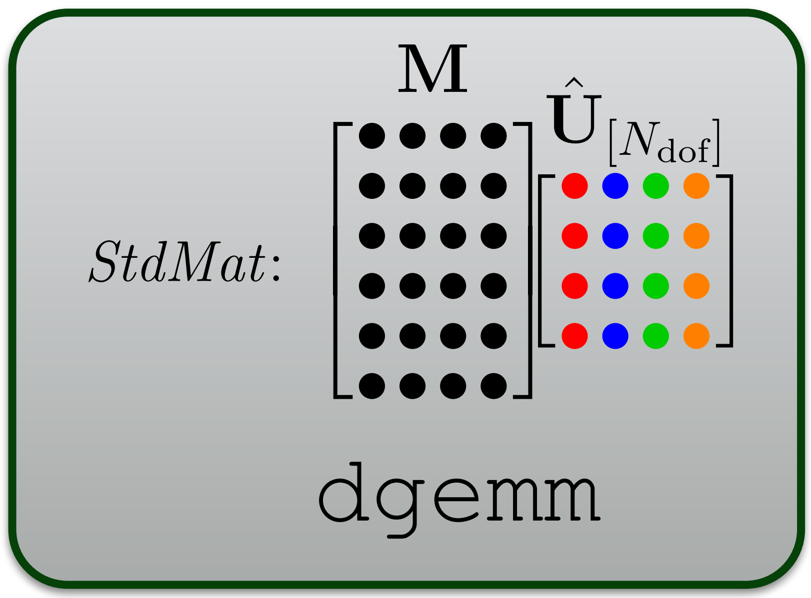


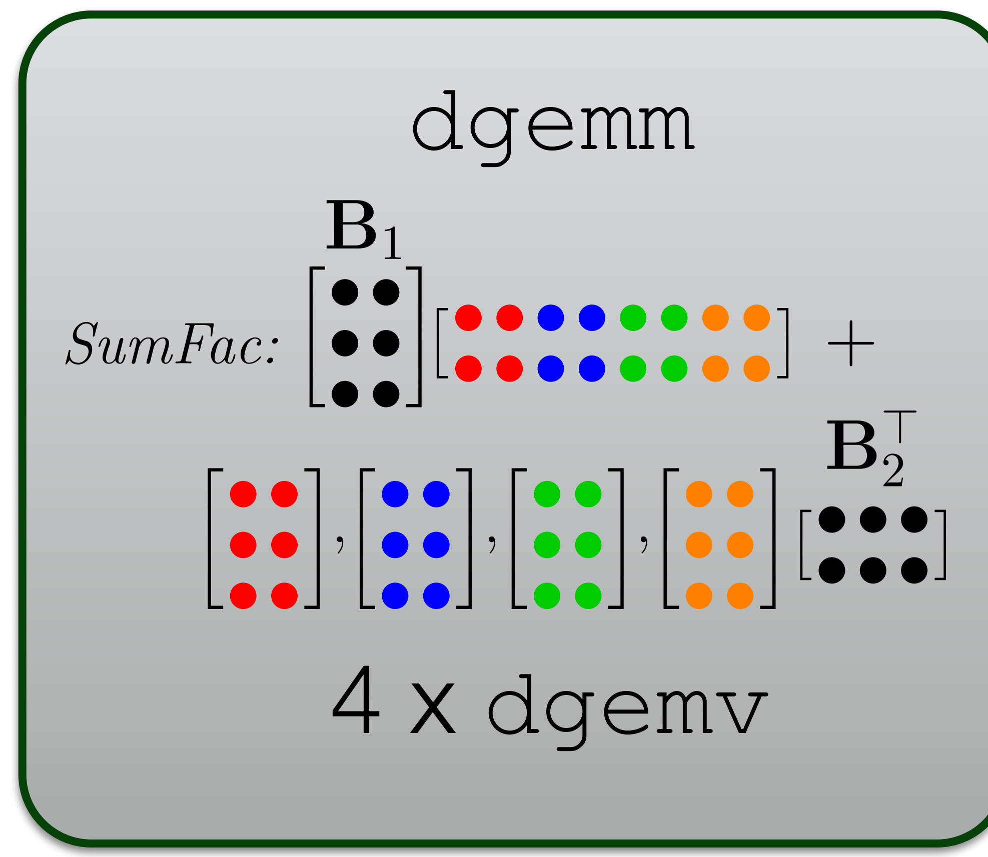
Collections

Use BLAS calls throughout



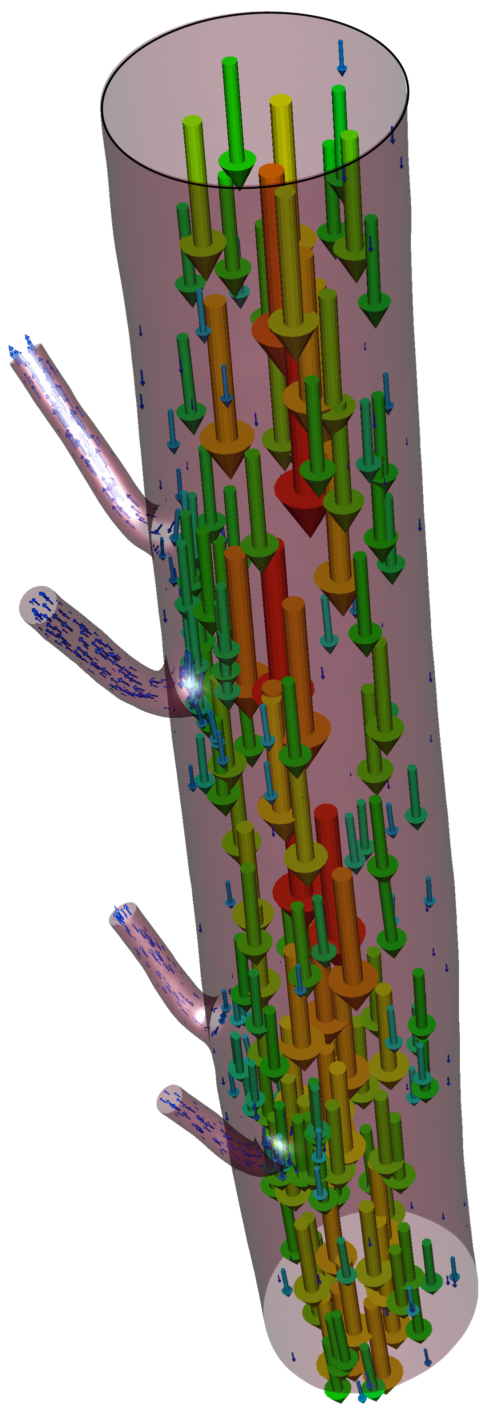
4 quad mesh

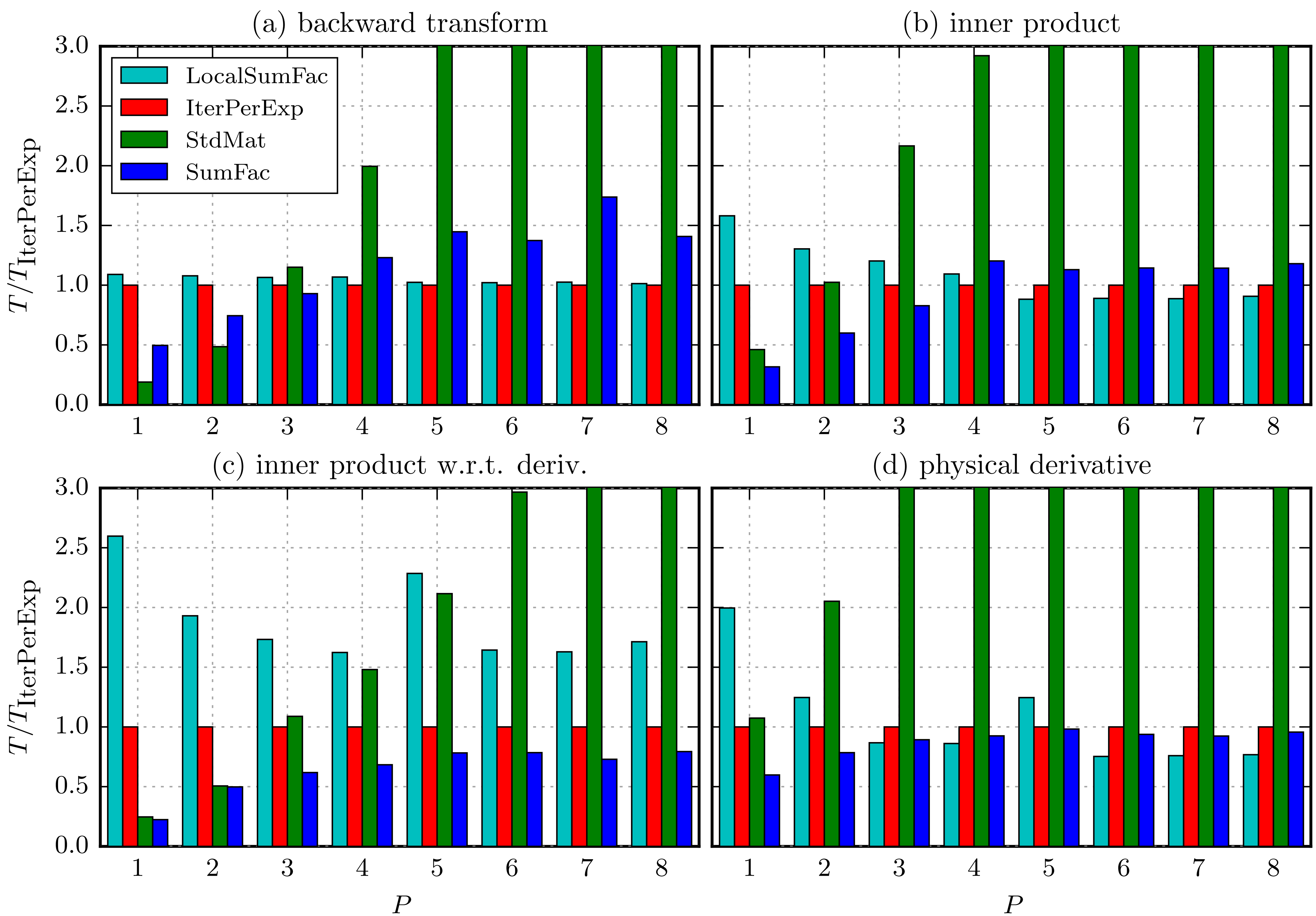




Test case

Intercostal pair
21k prisms
41k tets

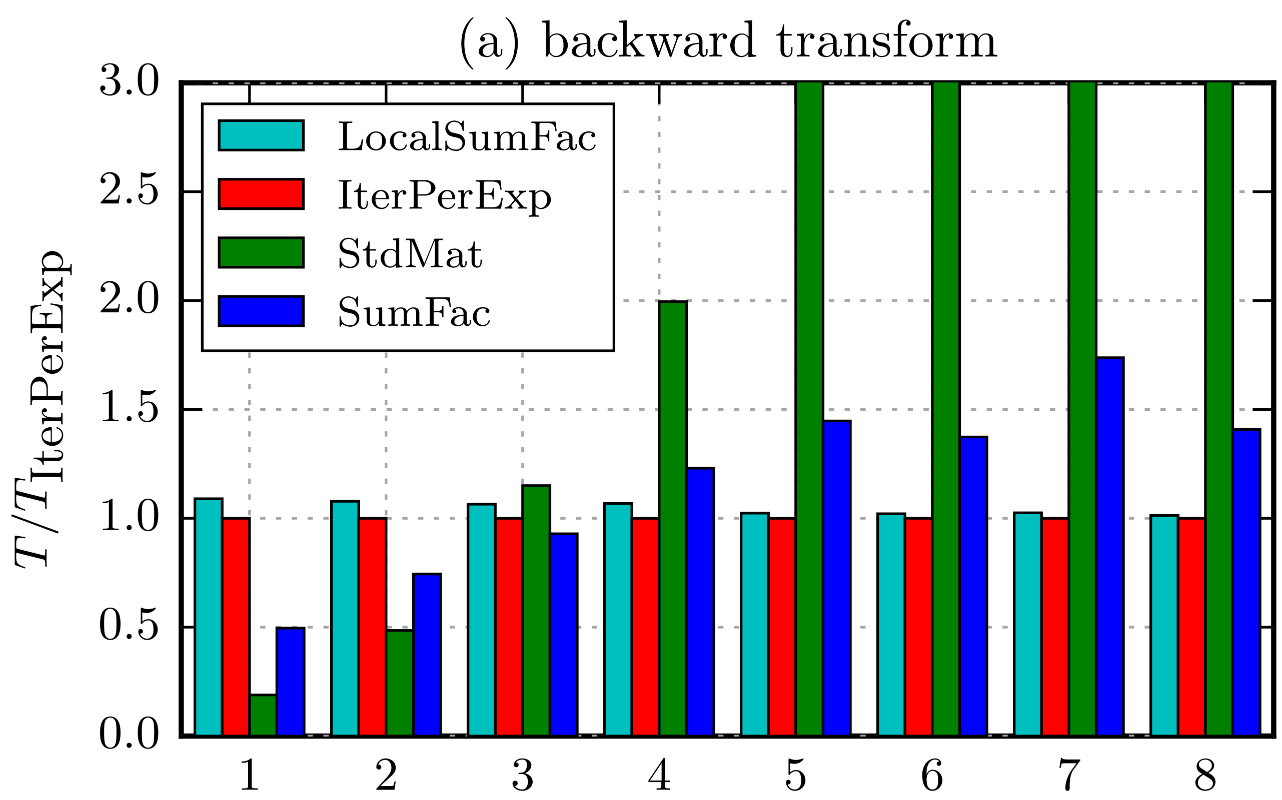




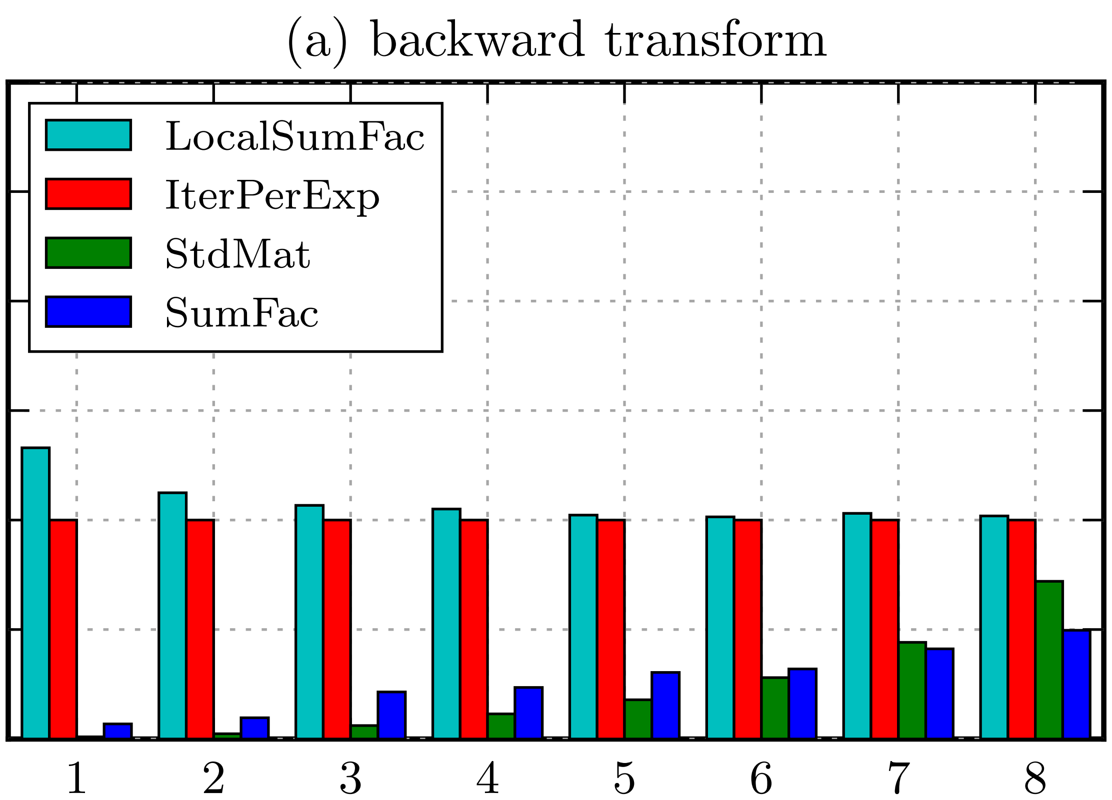
Performance overview

- *StdMat* tends to be most effective at lower orders
- Collections are less effective at high-order
    - Expected behaviour: matrices are very large for 3D elements - different story in 2D
- PhysDeriv benefits from *SumFac* even at very low polynomial orders
- Similar trends for tetrahedra, but cross-over points are different

Towards better performance



Reference BLAS



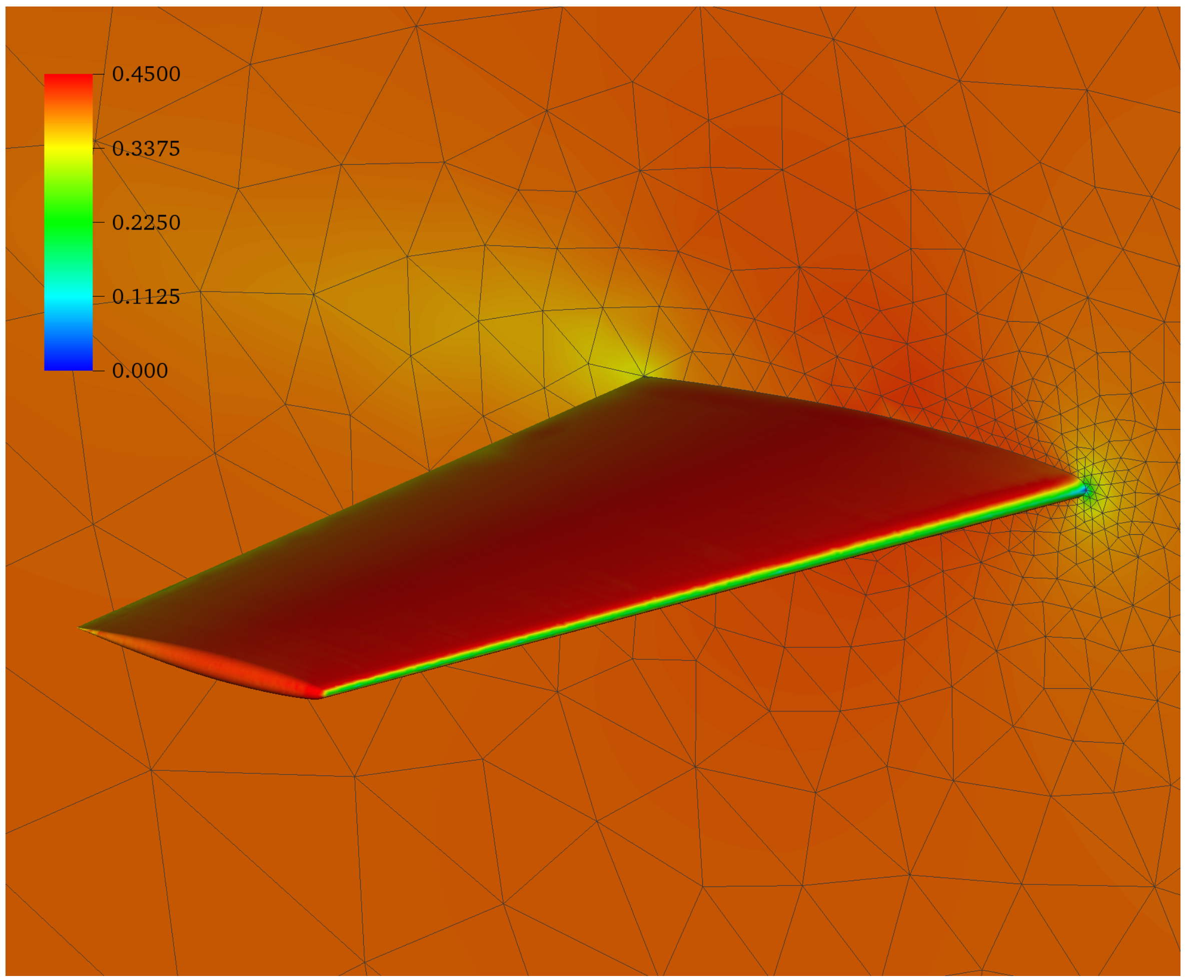
OpenBLAS

Clearly get a different picture!

Autotuning

- It's somewhat obvious that BLAS choice is very important, but lots of other factors:
  - ➡ machine-specific effects (processor frequency, cache, memory bandwidth/bus speed, ...)
  - ➡ different element types on each processor
- We therefore use a simple auto-tuning strategy at runtime
  - ➡ Every processor runs each implementation type for each operator at startup for 1 second each
  - ➡ Typically takes about 15-20 seconds
- Very simple but effective in selecting optimal scheme

Example: ONERA M6 wing



Machine	Operator	Scheme timings [s] *LocalSumFac*	*IterPerExp*	*StdMat*	*SumFac*
cx2	BwdTrans	0.00213393	0.00209944	**0.000202192**	0.000534608
	IProductWRTBase	0.00245141	0.00200234	**0.000233064**	0.000521411
	IProductWRTDerivBase	0.0266448	0.017248	**0.00201284**	0.00298702
	PhysDeriv	0.00485056	0.00492247	0.00389733	**0.00319892**
ARCHER	BwdTrans	0.000643393	0.000638955	**2.36882e-05**	4.74285e-05
	IProductWRTBase	0.000754697	0.000712303	**2.78743e-05**	0.000150587
	IProductWRTDerivBase	0.00827777	0.00530682	**0.00019947**	0.000643919
	PhysDeriv	0.00075556	0.000595179	**0.000287773**	0.000318533

Machine	Wall-time per timestep [s] *LocalSumFac*	Auto-tuned collections	Improvement
ARCHER	1.308	0.744	43%
cx2	0.356	0.135	62%

Runtime improvement: 40-60% ↑

Compressible Euler flow

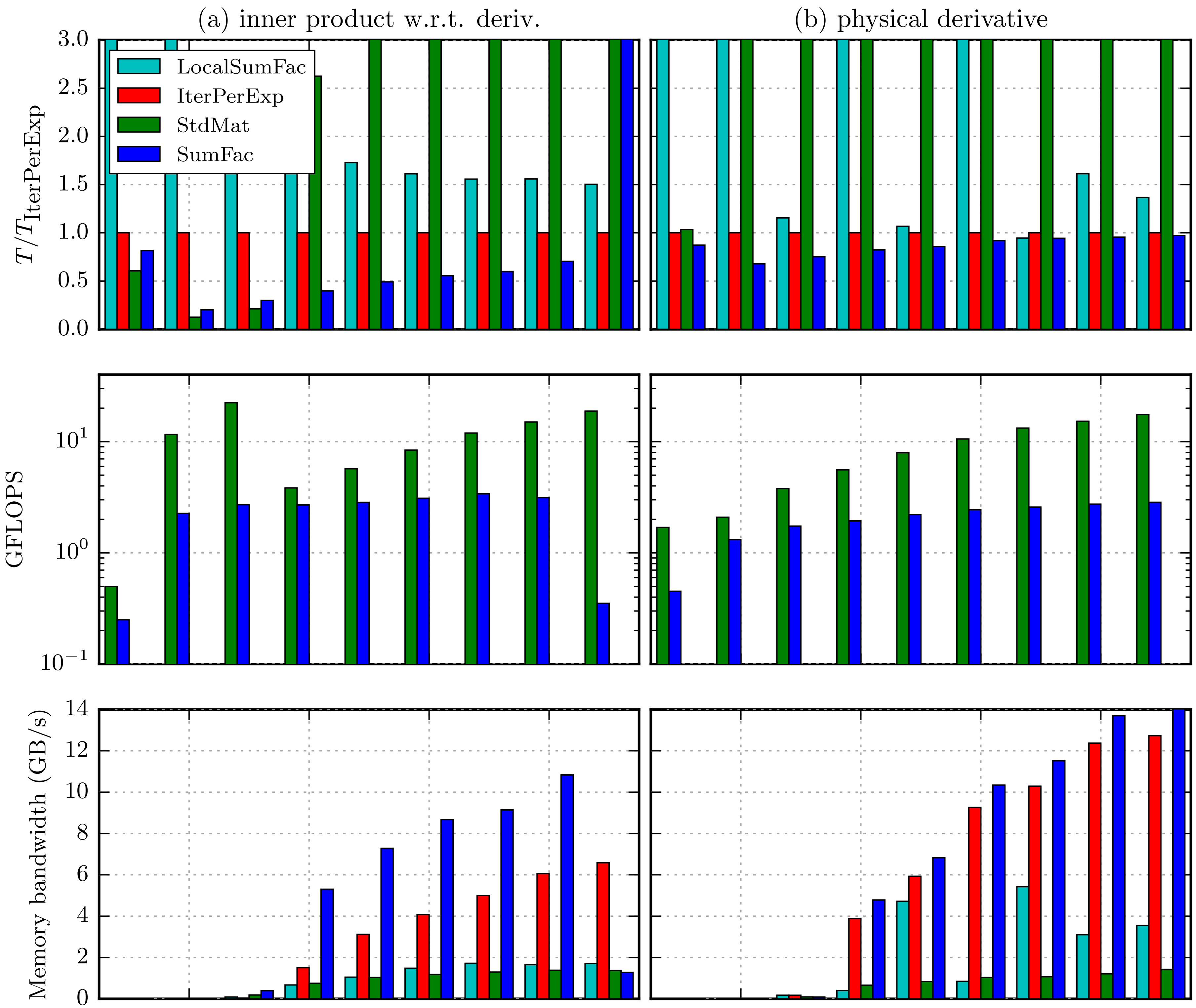
Fully explicit, $P = 2$, 960 cores, ~150k tets

Inner product w.r.t derivative very important

Insight into performance

- What determines performance?
- Examine hardware counters (core/uncore)
- Using Intel Performance Counter Monitor
- Intel i7-5960K system
- Still somewhat of a work in progress

Insight into performance



Low *P*: smaller matrices *StdMat* uses flops more effectively, operation count comparable to sum factorisation

High *P*: larger matrices, *SumFac* uses memory bandwidth more effectively in combination with lower operation count

Summary

- Collections speed up our code in fully explicit problems and explicit parts of implicit solvers
- Different schemes allow us to explore wider range of flop/byte space
- Auto-tuning important - maybe a little simplistic
- Inroad into using accelerators in a flexible manner
- Implicit solvers require different approach

Thanks for listening!

d.moxey@imperial.ac.uk

www.nektar.info