h-to-*p* efficiently: the use of collections with accelerators within Nektar++

D. Moxey
College of Engineering, Maths & Physical Sciences, University of Exeter

C. Cantwell, S. J. Sherwin
Department of Aeronautics, Imperial College London

R. M. Kirby
Scientific Computing and Imaging Institute, University of Utah

SIAM Conference on Computational Science & Engineering,
Atlanta, GA, USA

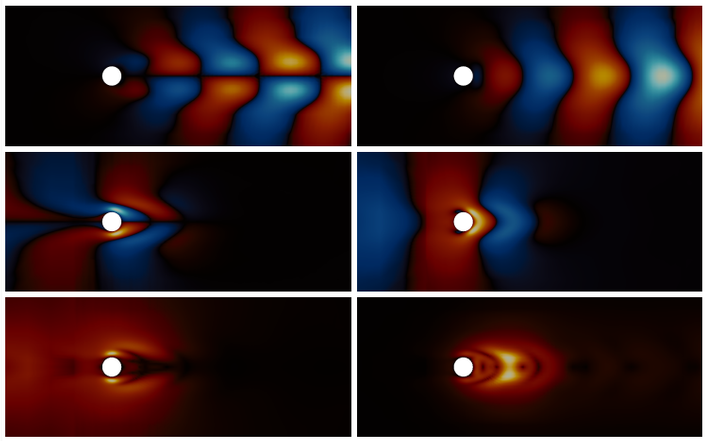
26th February 2017

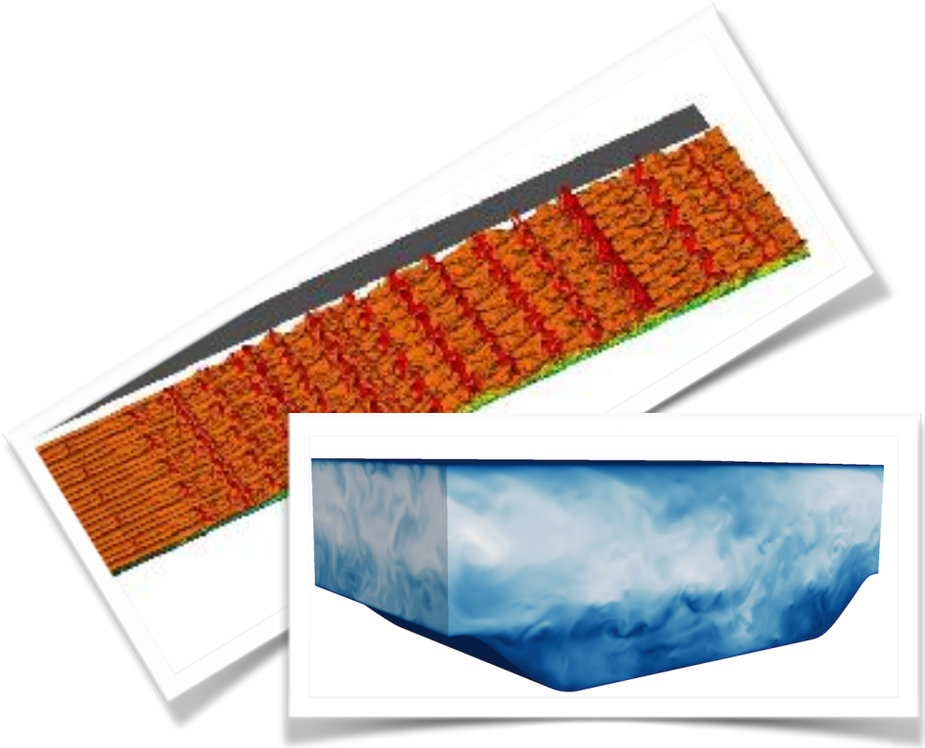
Outline

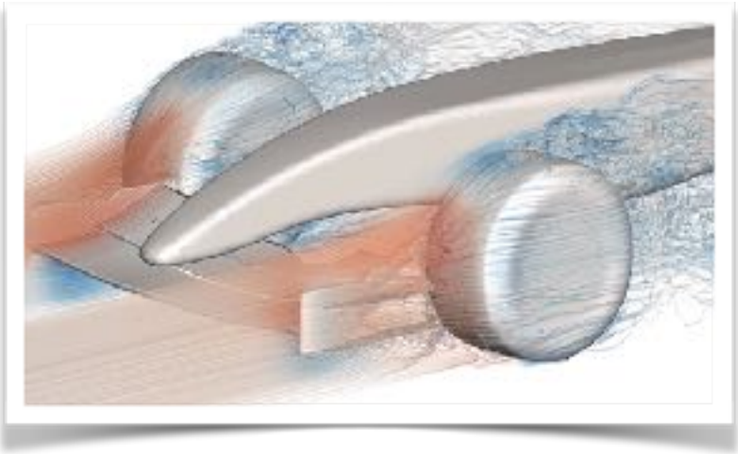
- Nektar++ & spectral/*hp* method
- Challenges for KNL
- Collections & `libxsmm`
- Summary

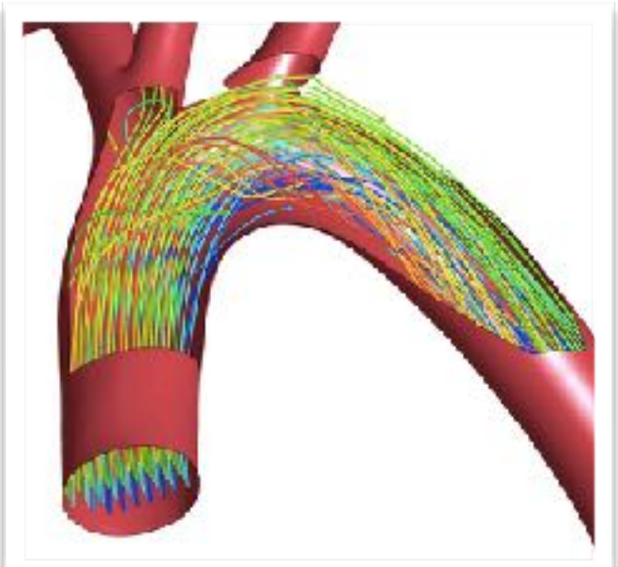
Nektar++ goals

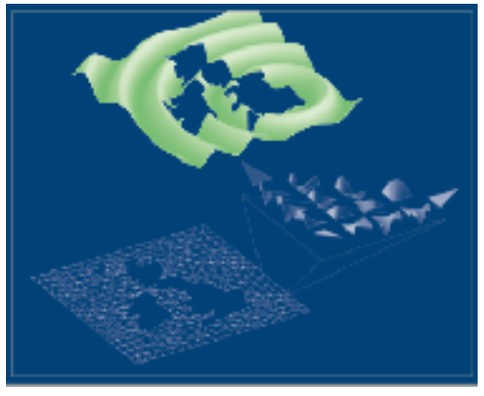
- Make it simpler/quicker to create high-order solvers for a range of fields and applications (including CG/DG/HDG)
- Support 1/2/3D and **unstructured hybrid meshes** for complex geometries: tets, prisms, etc.
- Scale to large numbers of processors
- Be efficient across a range of polynomial orders and core counts
- Bridge current and future hardware diversity

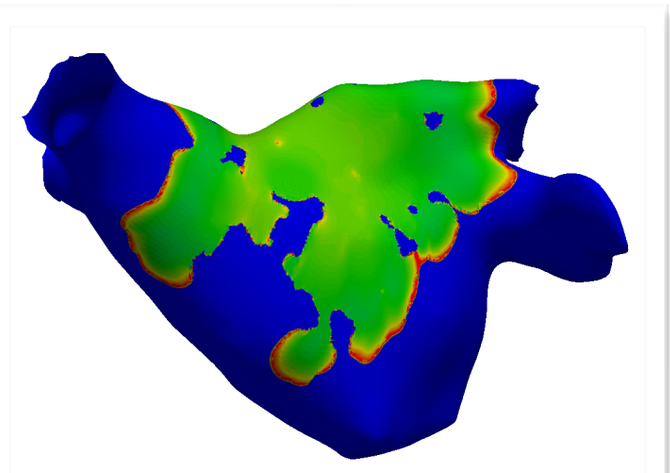


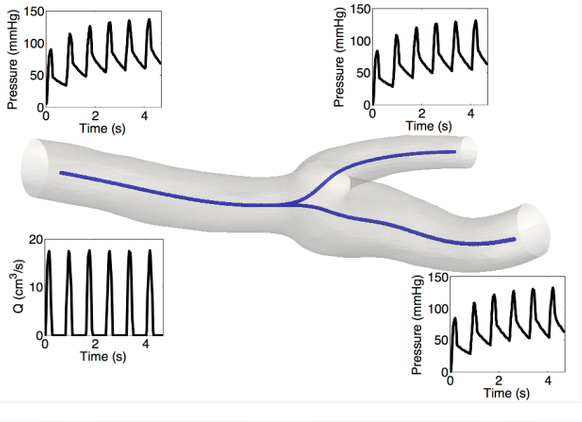


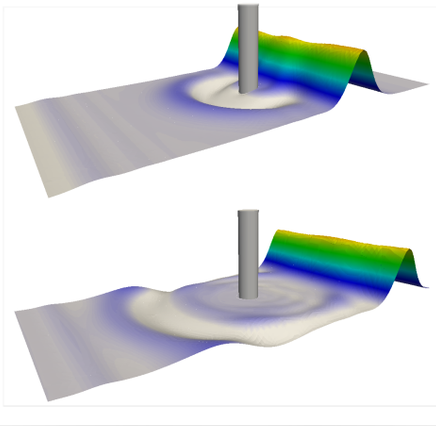


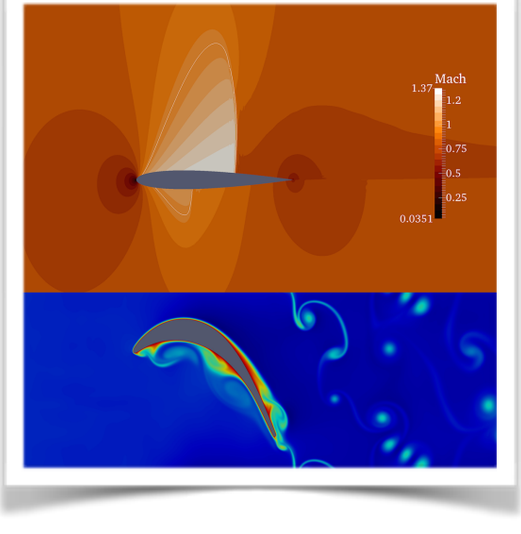


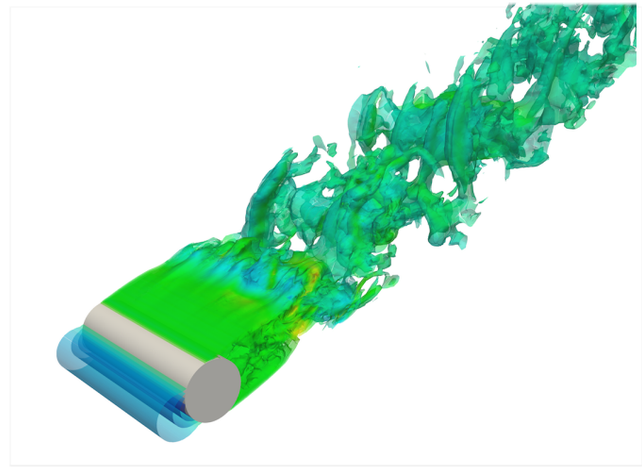





Pressure (mmHg)
150
100
50
0
0
2
4
Time (s)
Pressure (mmHg)
150
100
50
0
0
2
4
Time (s)
Q (cm3/s)
20
10
0
0
2
4
Time (s)
Pressure (mmHg)
150
100
50
0
0
2
4
Time (s)

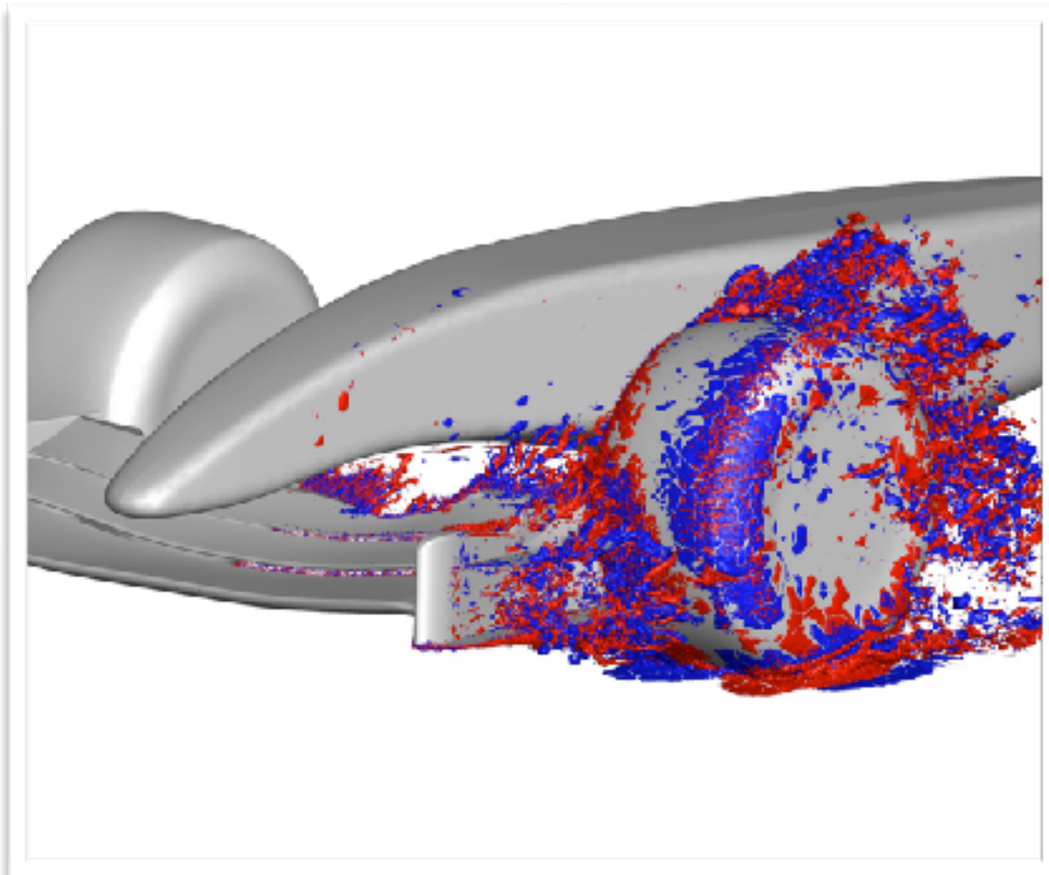


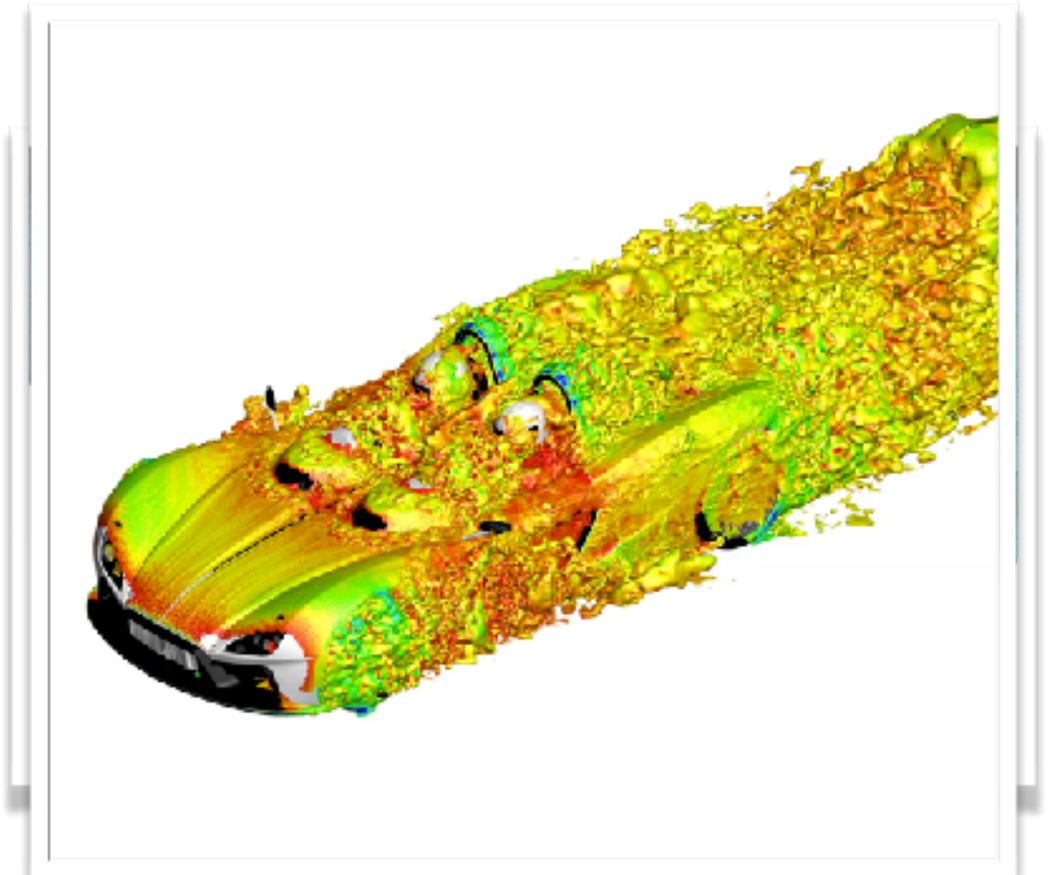

Mach
1.37
1.2
1
0.75
0.5
0.25
0.0351



Unstructured simulations

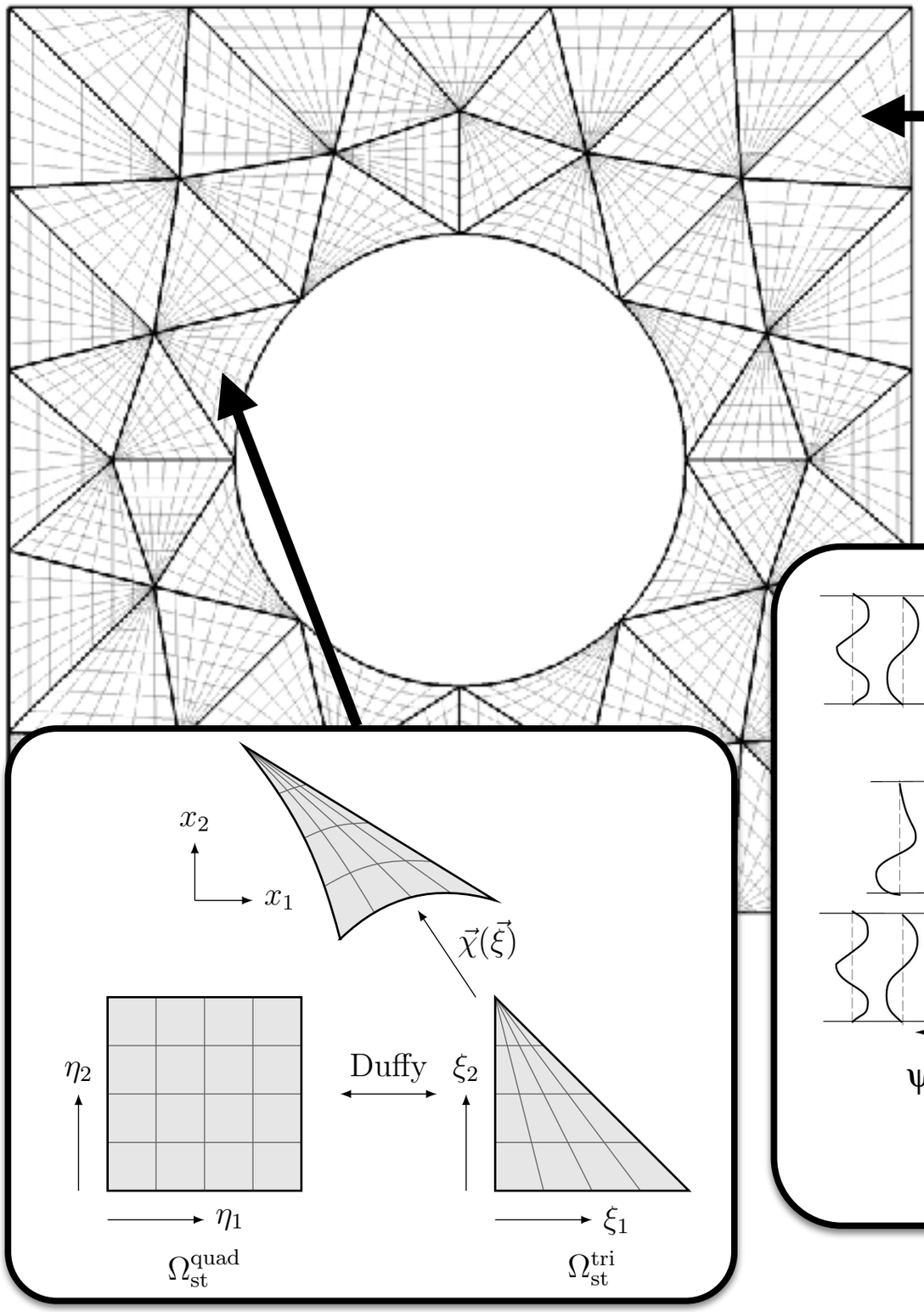
Common knowledge that hexes yield best performance



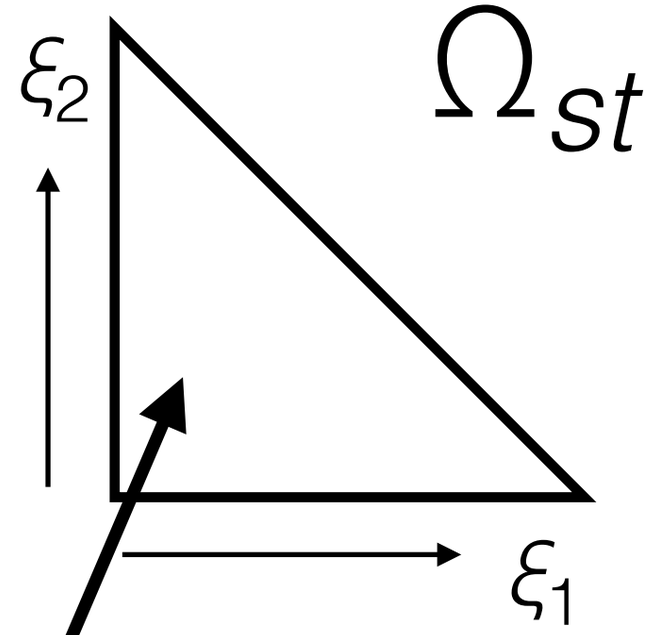


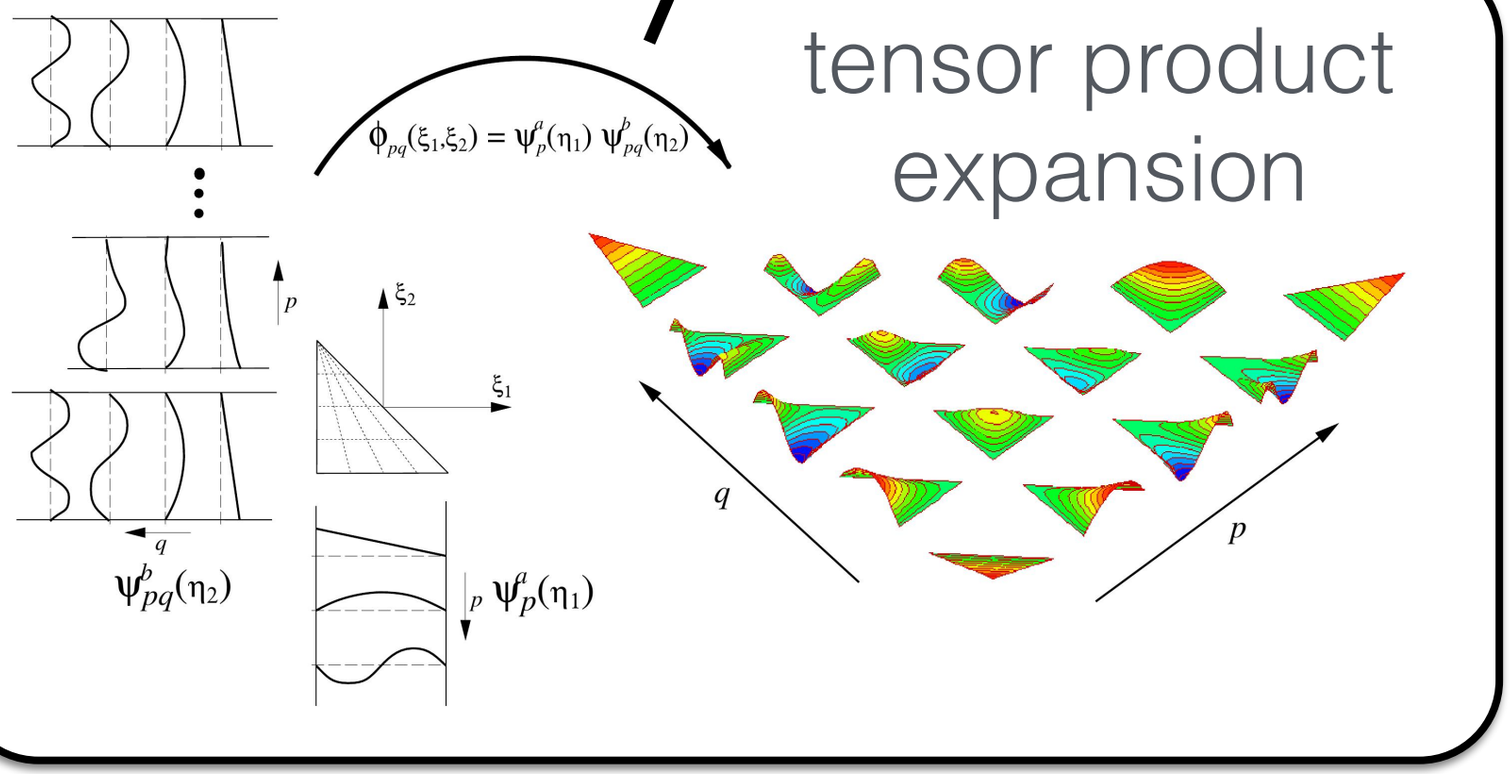
Complex geometries presently **require** unstructured meshes - how to improve performance?

Spectral/*hp* element method



map from reference element



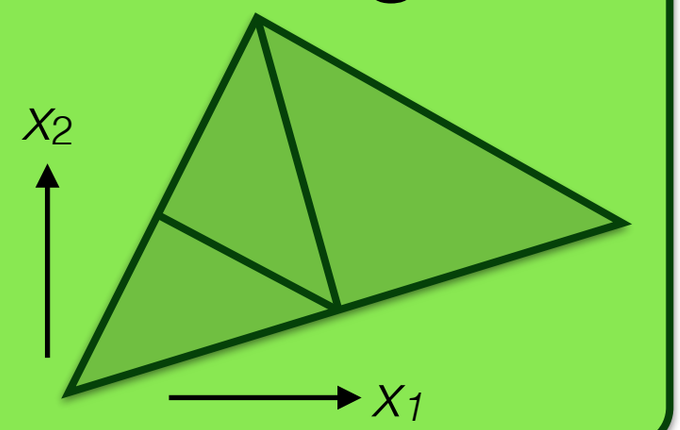


Framework design

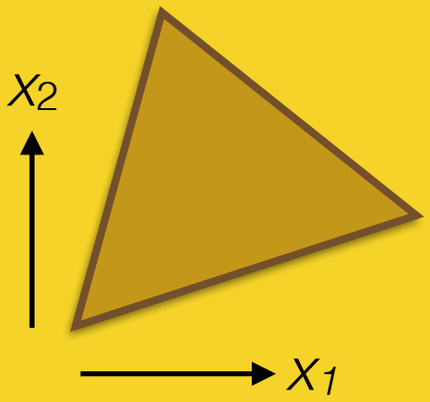
$$u^{\delta} = \sum_i \hat{u}_i \Phi_i(x)$$

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

MultiRegions



LocalRegions



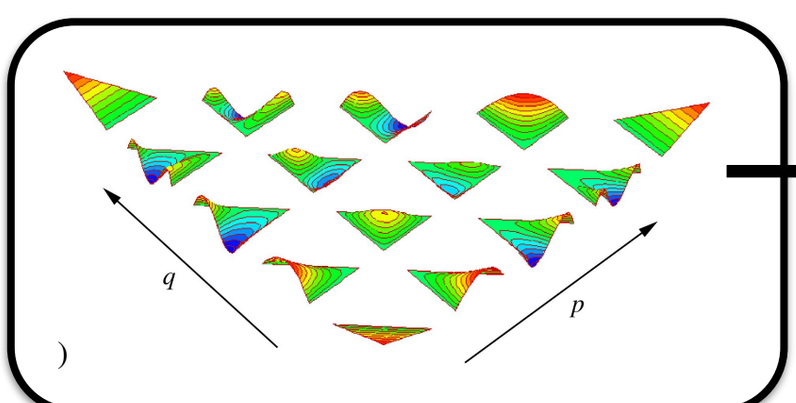
SpatialDomains

$$\mathbf{x} = \chi^e(\xi)$$

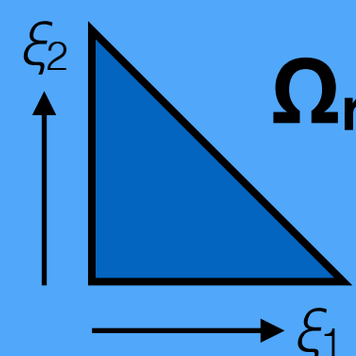
$$\frac{\partial x_i}{\partial \xi_j} \quad \frac{\partial \xi_i}{\partial x_j}$$

$$\mathrm{f}[i] = \int_{\Omega} f(x)\Phi_i(x)\,dx$$

$$\mathrm{f}[i] = \sum_{e=1}^{N_{\mathrm{el}}} \int_{\Omega^e} f(x)\psi_i^e(x)\,dx$$



StdRegions



"Defining" features

Generally *not* collocated

$$u(\xi_{1i}, \xi_{2j}) = \sum_{n=1}^{P^2} \hat{u}_n \phi_n(\vec{\xi}) = \sum_{p=1}^{P} \sum_{q=1}^{Q} \hat{u}_{pq} \phi_p(\xi_{1i}) \phi_q(\xi_{2j})$$

quadrature points

modal coefficients

Uses tensor products of 1D basis functions, *even for non-tensor product shapes*, but indexing harder

$$u(\xi_{1i}, \xi_{2j}, \xi_{3k}) = \sum_{p=1}^{P} \sum_{q=1}^{Q-p} \sum_{r=1}^{R-p-q} \hat{u}_{pqr} \phi^a_p(\xi_{1i}) \phi^b_{pq}(\xi_{2j}) \phi^c_{pqr}(\xi_{3k})$$

Sum-factorisation

Essential for performance at high polynomial orders

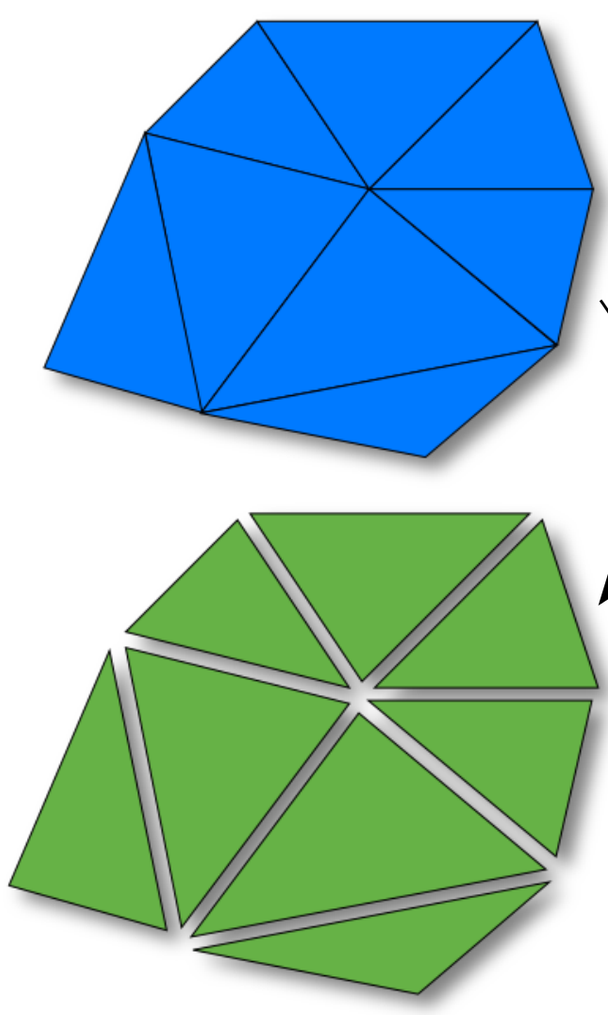
$$\sum_{p=1}^{P}\sum_{q=1}^{Q}\hat{u}_{pq}\phi_p(\xi_{1i})\phi_q(\xi_{2j}) = \sum_{p=1}^{P}\phi_p(\xi_{1i})\left[\sum_{q=1}^{Q}\hat{u}_{pq}\phi_q(\xi_{2j})\right]$$

store this

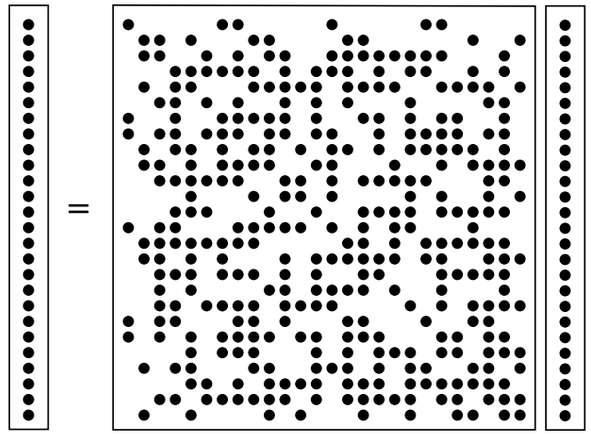
2D: $O(P^4) \rightarrow O(P^3)$ 3D: $O(P^6) \rightarrow O(P^4)$

We **can also do this** for tris, tets, prisms... but have to cope with harder indexing and smaller matrices

Implementation choices

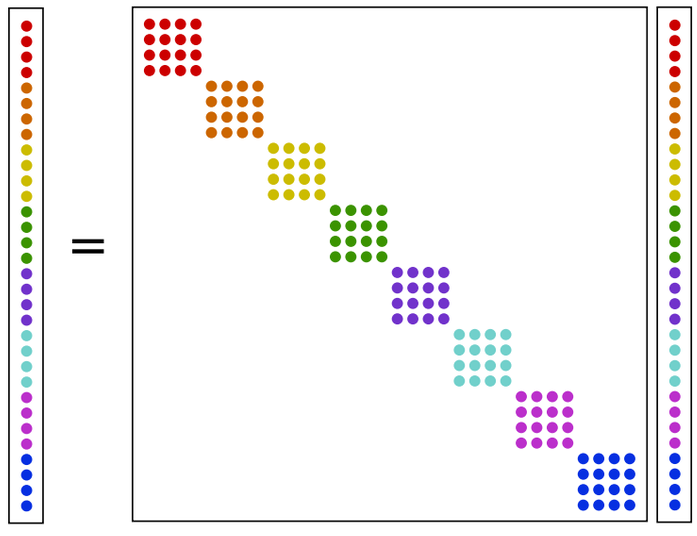


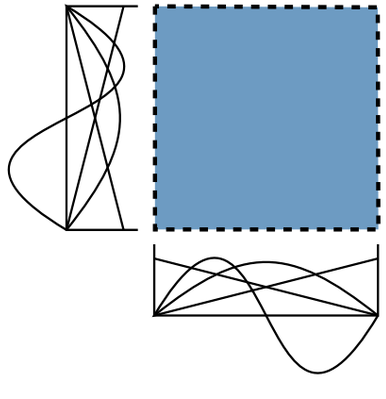
Global Strategy



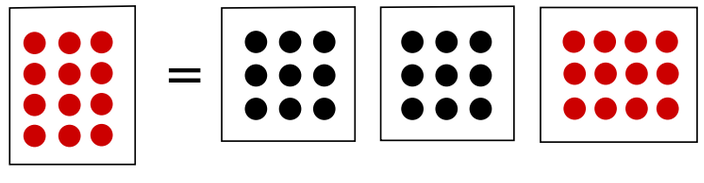
$\mathcal{A}$

Local Strategy





Sum-factorisation



Increasing polynomial order

More localised memory access

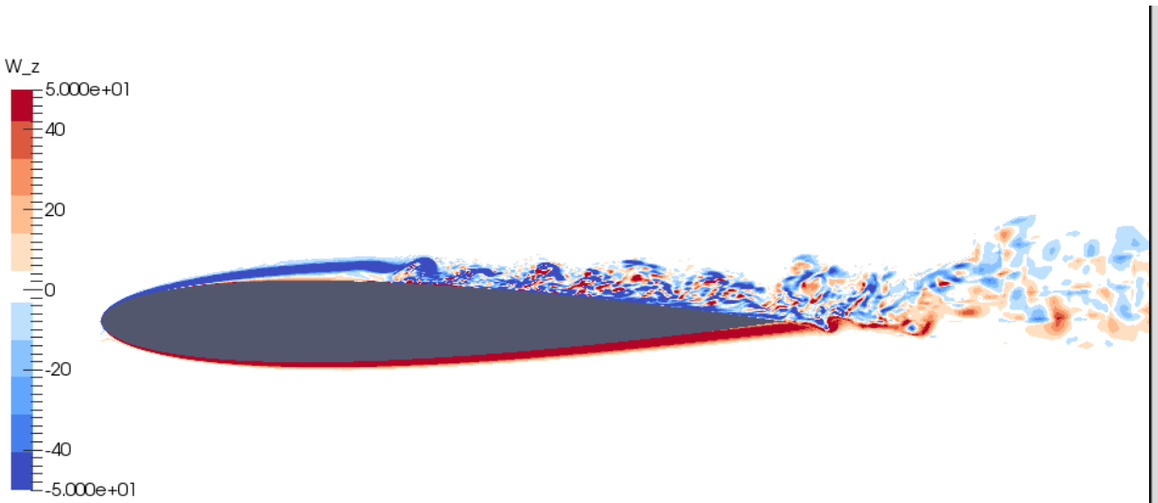
h-to-*p* efficiently

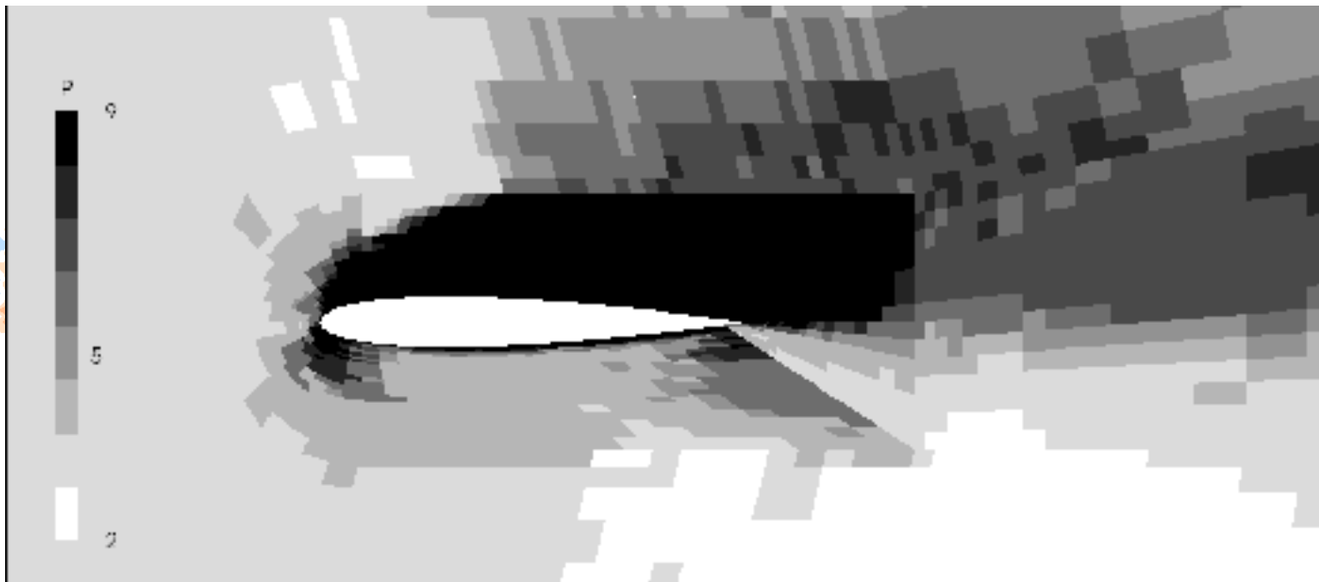
Why do we care about having *all* these approaches?

- Approach performance varies wildly depending on many factors that are not *a priori* determinable

- Allow us to explore the space of flops/byte ratio

- Also important for e.g. variable-*p* simulations





Challenges for KNL

- Nektar++ written in C++ (surprise!), uses abstraction, inheritance and OO heavily
  - ✔ Great for writing code & rapid prototyping
  - ✘ Hard to make it highly performant
  - ✘ Also hard to track/control memory usage

- Handing memory
  - ➡ DRAM vs. MCDRAM or host vs. device
  - ➡ Making this transparent to solver developers

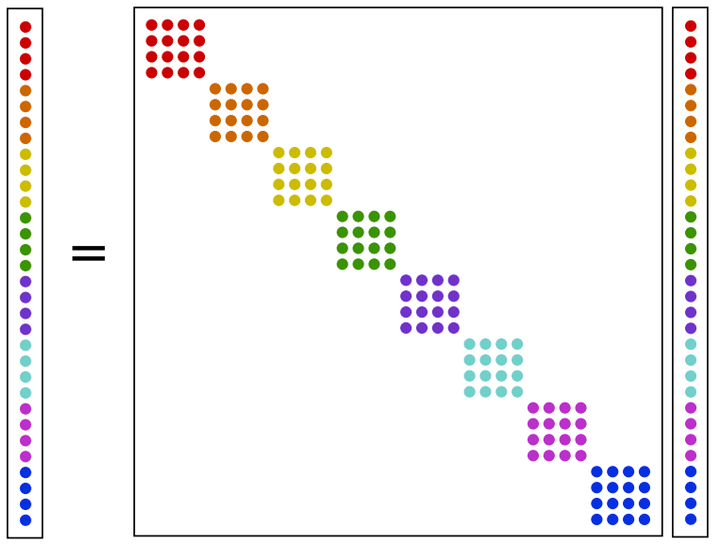
- Threading and SIMD vectorisation

Collections

- Reformulate implementation choices into kernel operations over multiple elements
- Group geometric terms $\frac{\partial x_i}{\partial \xi_j}$
- Focus around key components of Laplacian:
  - ➡ 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: $(\Phi_i, \nabla \Phi_j)$

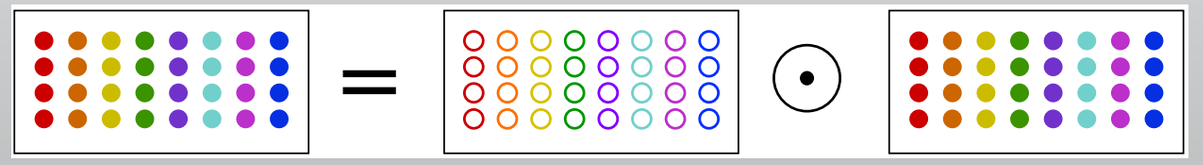
Schemes

Local Matrix

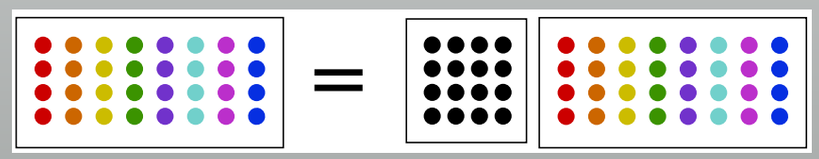


StdMat (standard matrix)

1. Apply Jacobian **(L1)**

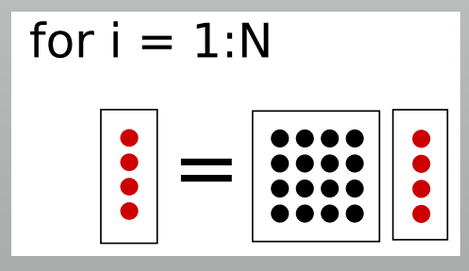


2. Multiply by ref. matrix **(L3)**



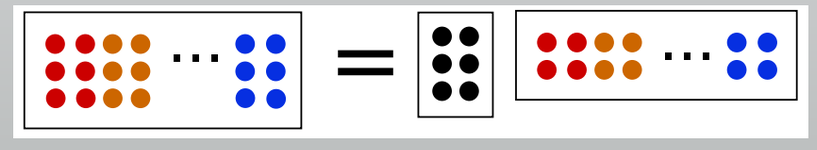
IterPerExp

1. Apply Jacobian **(L1)**
2. Apply local sum fact. **(N x L3)**

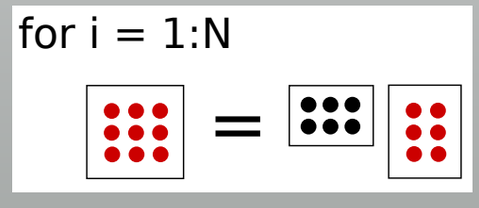


SumFac

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



3. Mult. second dimension **(N x L3)**

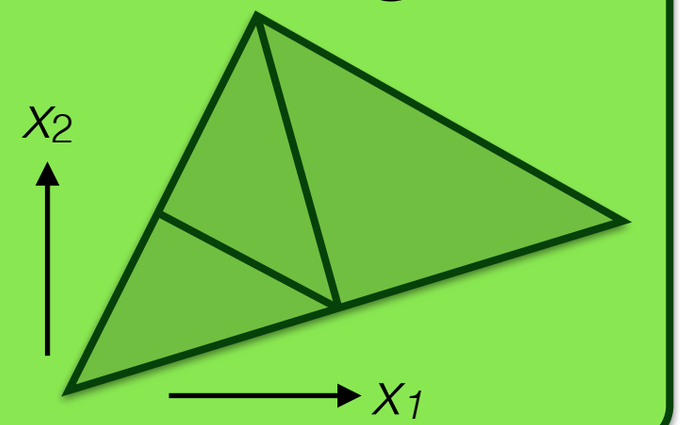


Collections

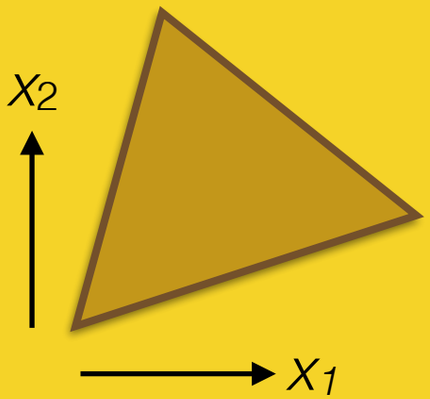
$$u^{\delta} = \sum_i \hat{u}_i \Phi_i(x)$$

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

MultiRegions



LocalRegions

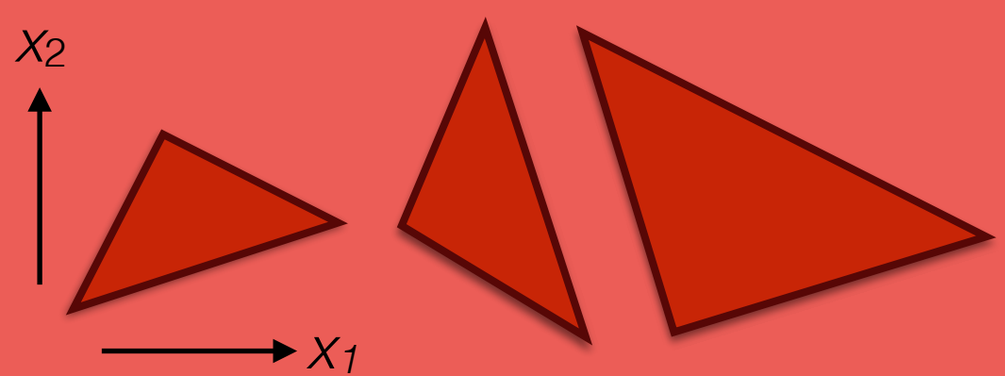


SpatialDomains

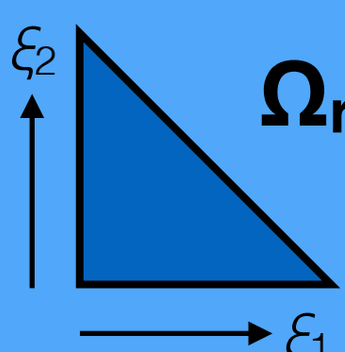
$$\mathbf{x} = X^e(\xi)$$

$$\frac{\partial x_i}{\partial \xi_j} \quad \frac{\partial \xi_i}{\partial x_j}$$

Collections

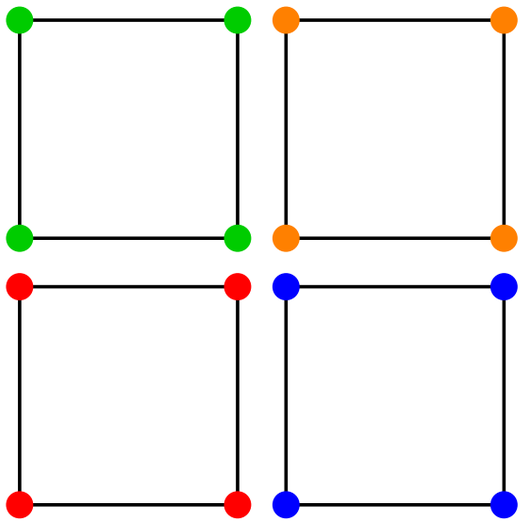


StdRegions

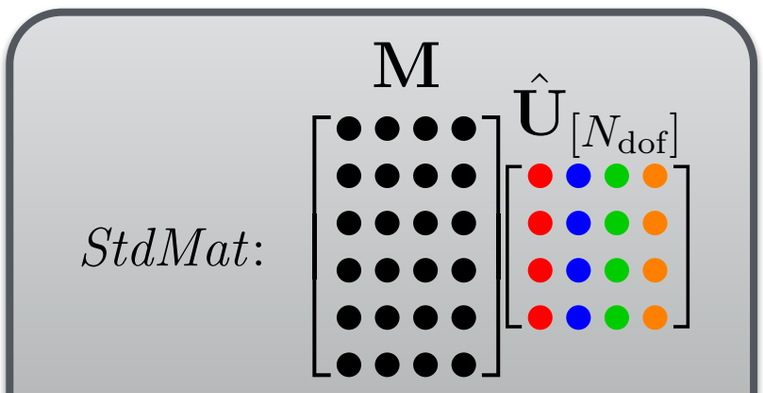


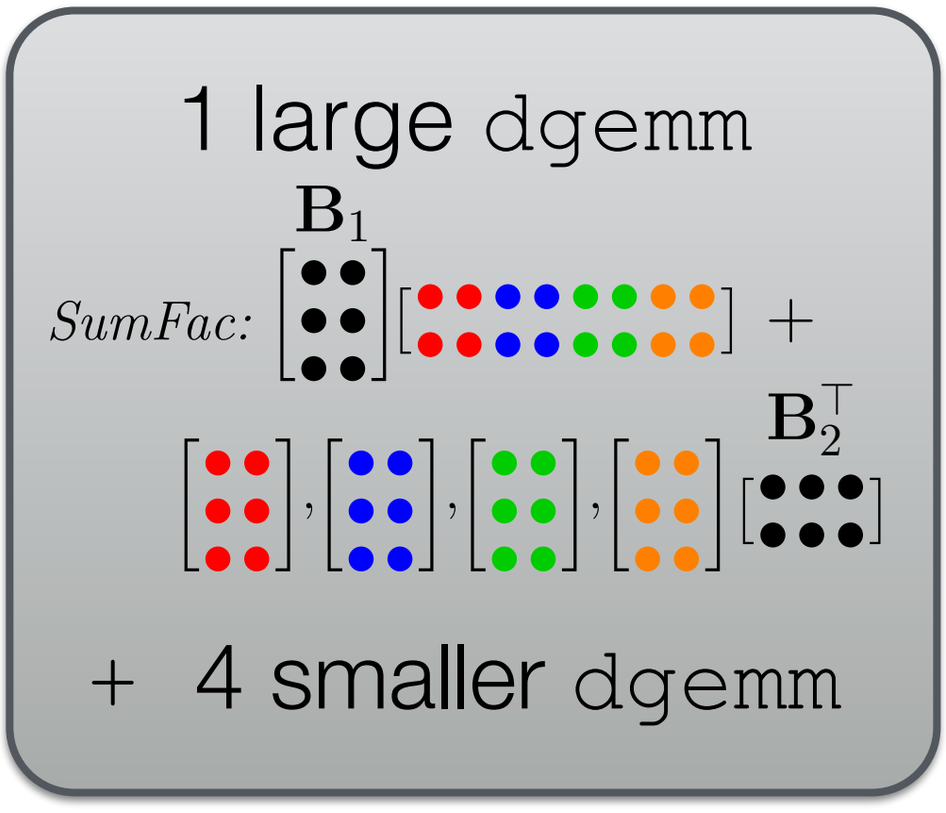
Collections

Use BLAS calls throughout



4 quad mesh





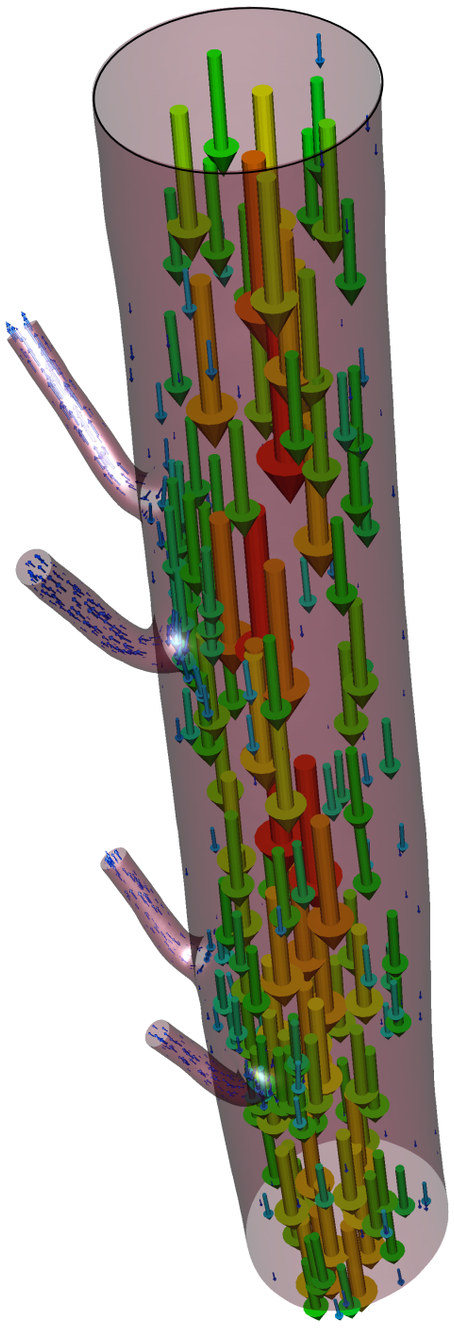
Can also do this for non-TP elements:
data ordering harder, matrices smaller (bad for BLAS)

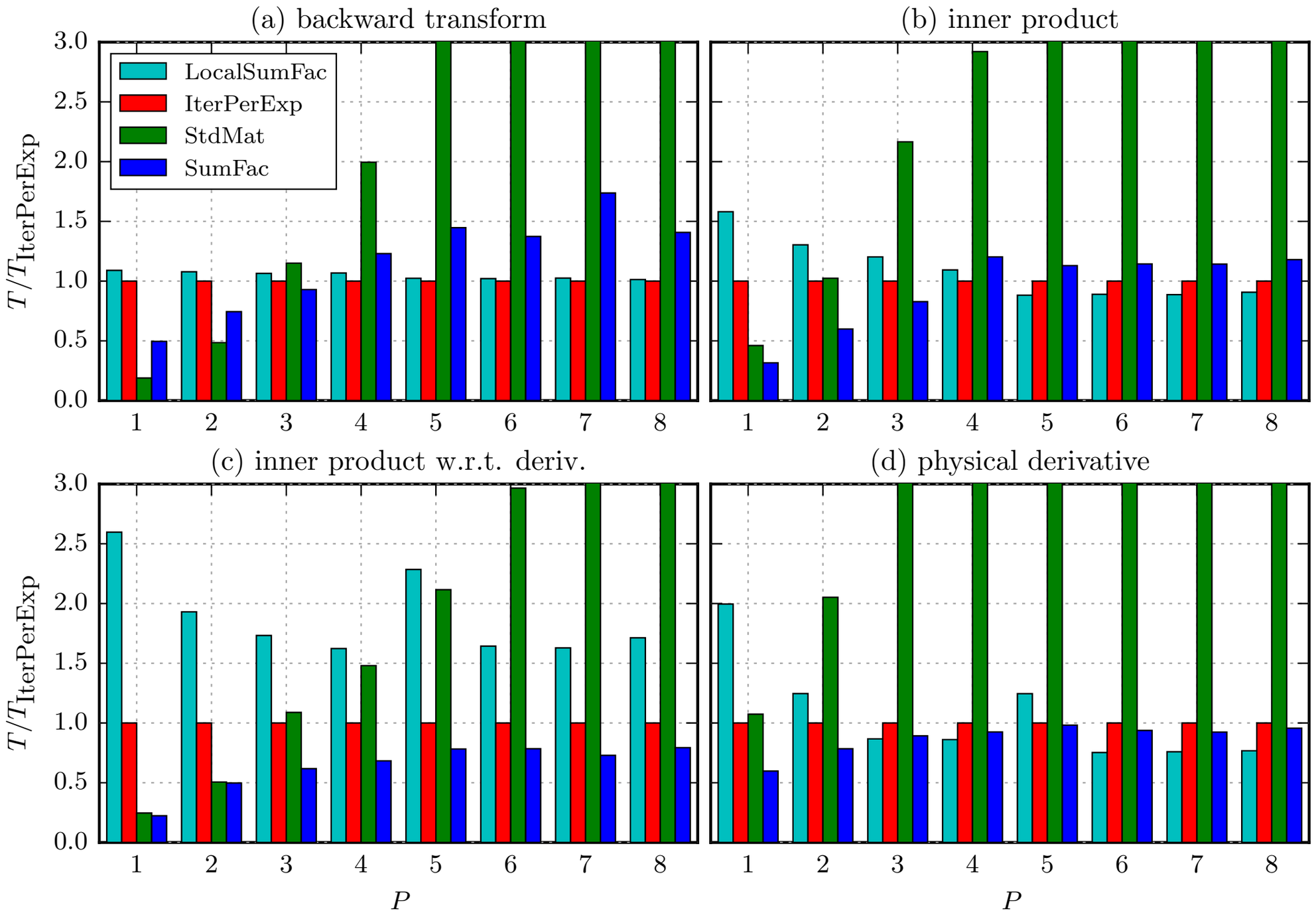
Test case

Intercostal pair

21k prisms

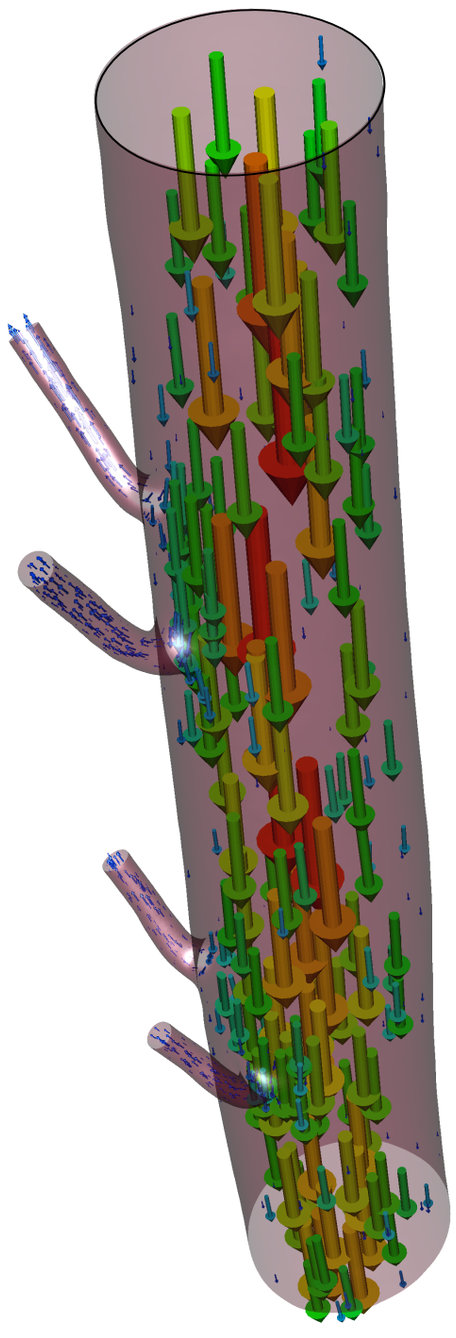
41k tets

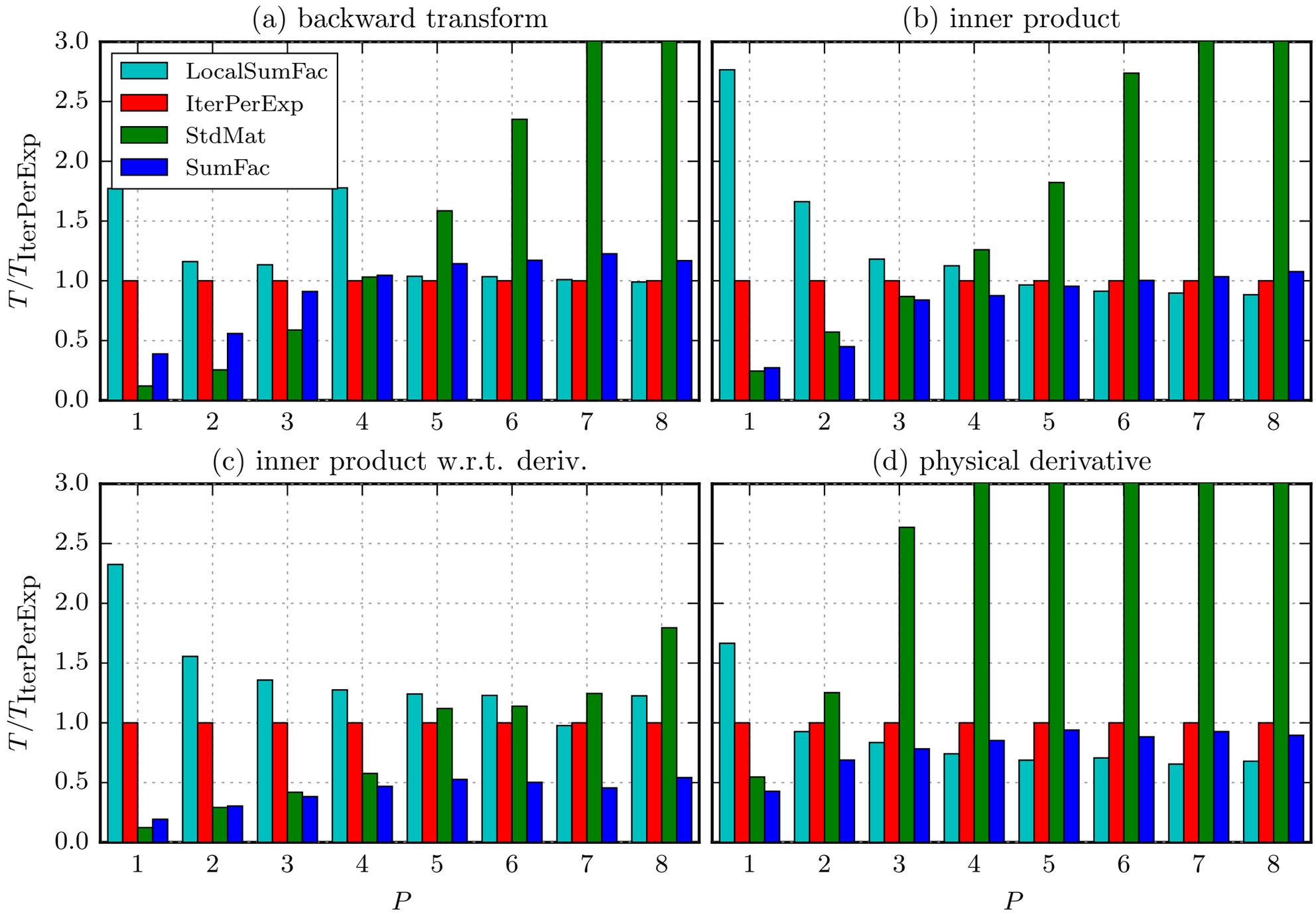




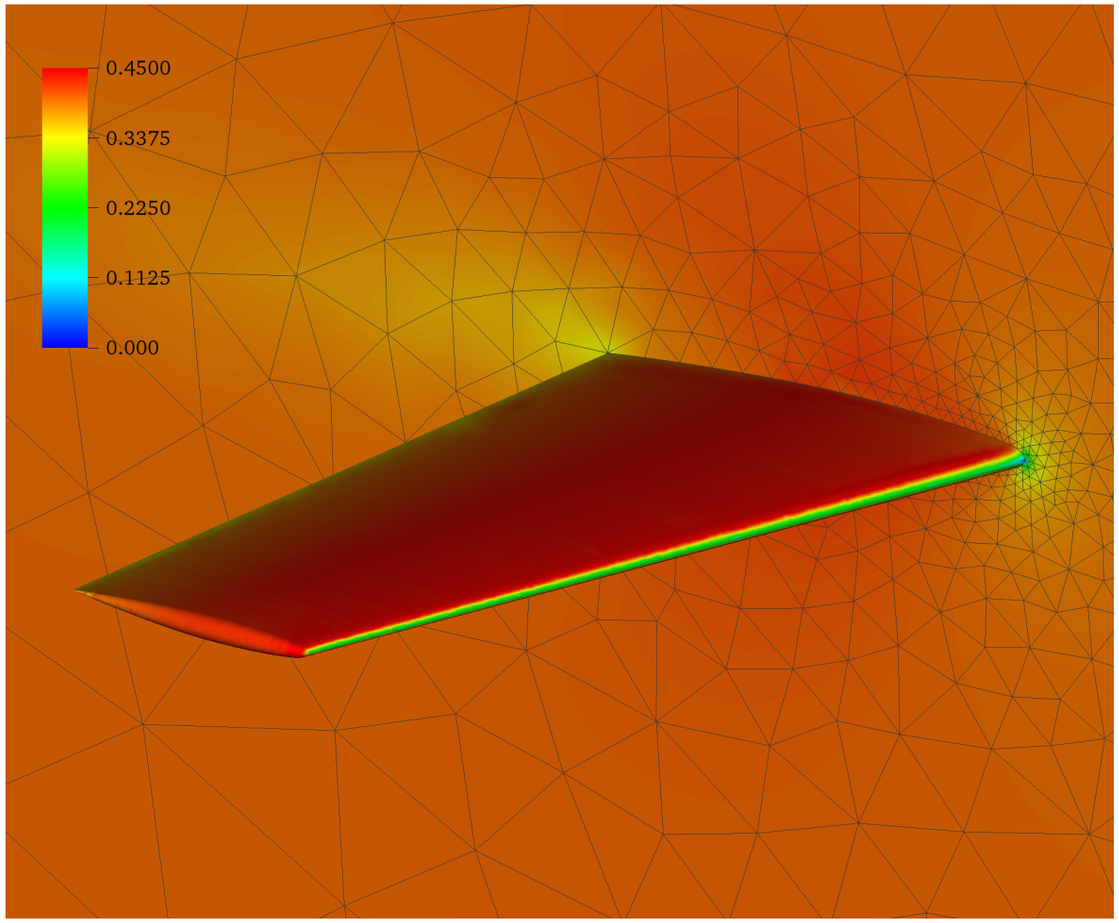
Test case

Intercostal pair
21k prisms
41k tets





Example: ONERA M6 wing



		Scheme timings [s]			
Machine	Operator	*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

	Wall-time per timestep [s]		
Machine	*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

libxsmm

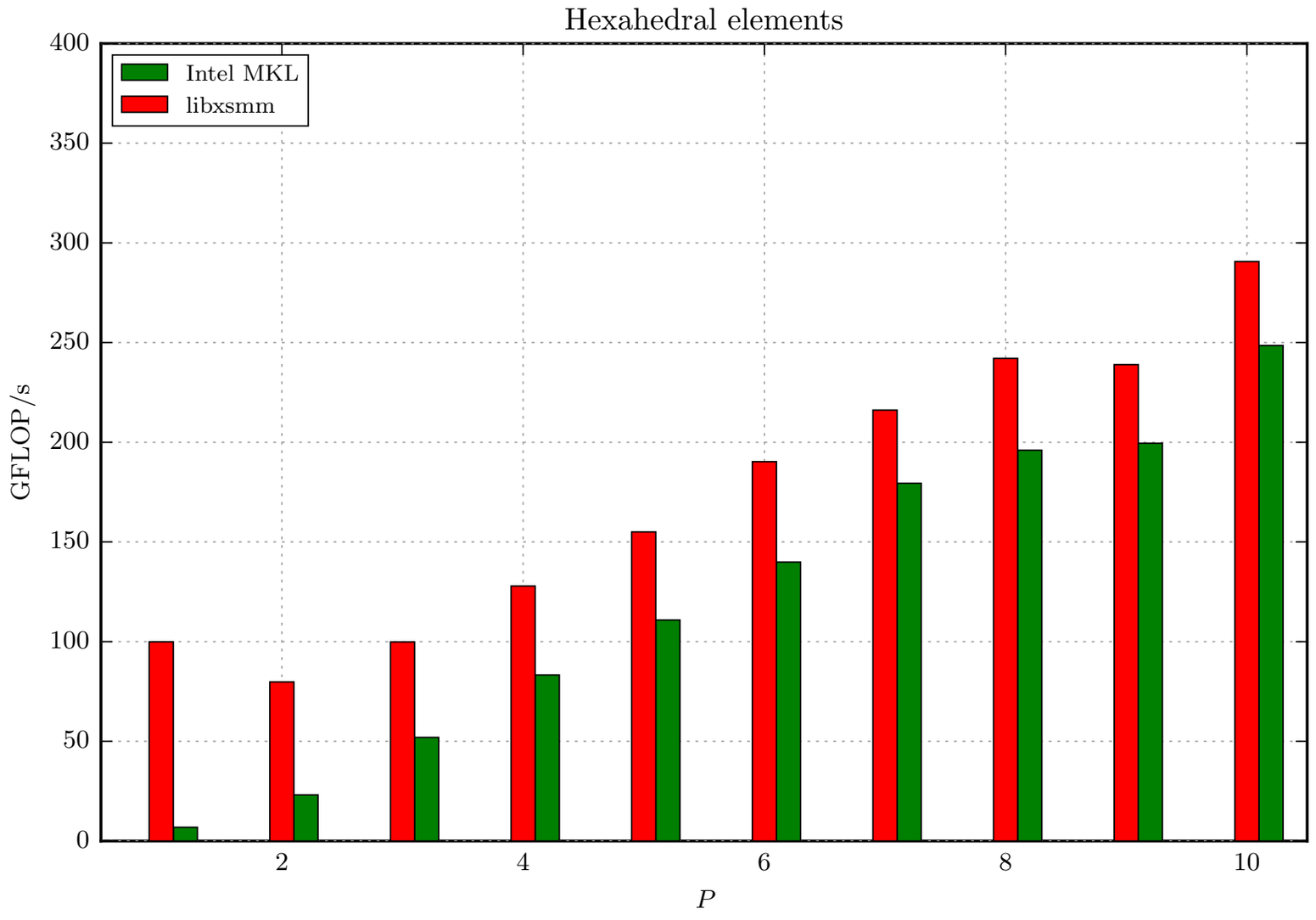
- Most of the matrix-matrix multiplies done in collections are small, at least in one rank

- libxsmm yields encouraging performance gains over standard MKL/BLAS, particularly for non-TP elements

- **Challenge:** our existing calls frequently use transposes - need to reorder/pretranspose

  ➡ This is very challenging for non-tensor product elements (tris/tets/etc)

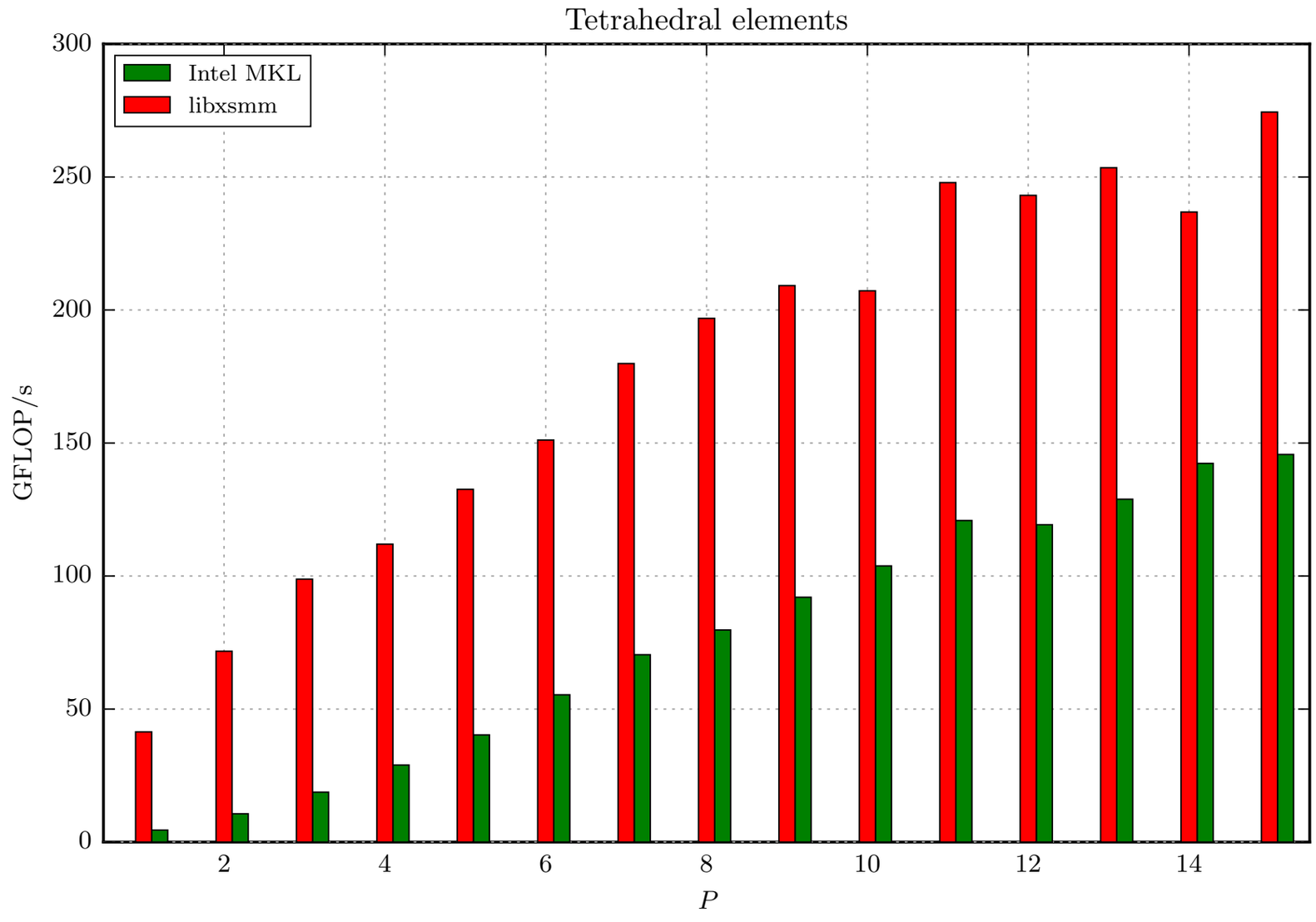
GFLOP/s performance



2 x Intel E5-2670v3
theoretical peak ~1TFLOP/s

~20-40% improvements
in both flops & runtime
over MKL

GFLOP/s performance



Higher performance gains
Recovers hex performance

Requires us to
reorder local coefficients

Summary

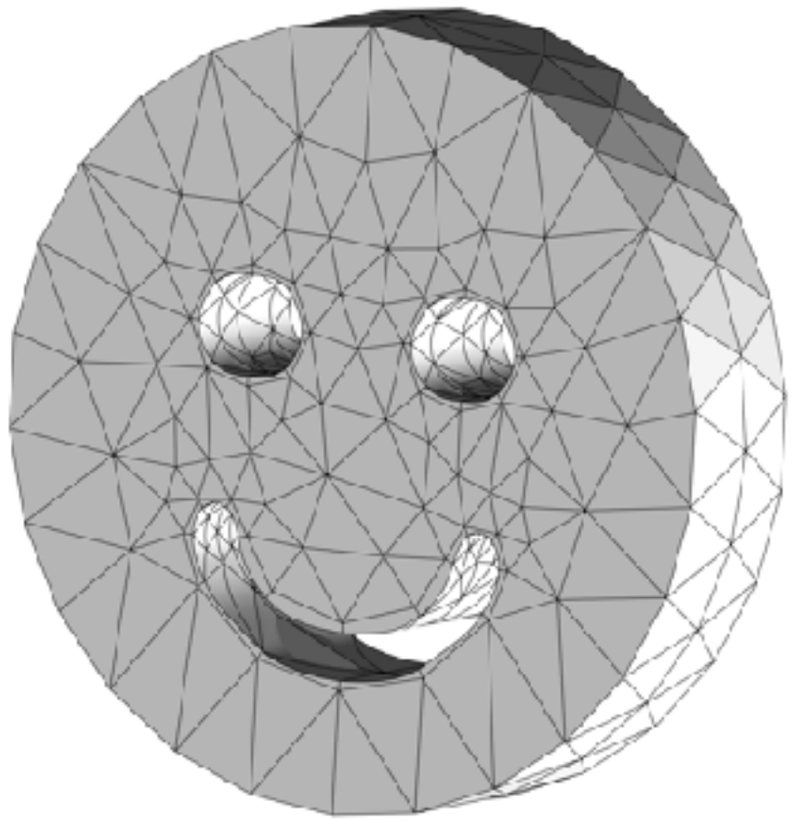
- Collections have sped up our code and made us think much harder about memory & hardware, particularly for KNL architecture

- Transition to kernels easier to consider threading, vectorisation & tuning, but keeping transparency

- `libxsmm` looks encouraging for maximising hardware potential, particularly for non-TP elements

- Still need to tackle memory management, particularly for KNL & non-CPU architectures

Thanks for listening!



https://davidmoxey.uk/ @davidmoxey

d.moxey@imperial.ac.uk

www.nektar.info