Supported by SPPEXA (Software for exascale computing, DFG), project ExaDG
Supported by Bayerisches Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR)

Fast Matrix-Free High-Order Discontinuous Galerkin Kernels: Performance Optimization and Modeling

Martin Kronbichler

in collaboration with

Niklas Fehn Katharina Kormann Wolfgang A. Wall

Institute for Computational Mechanics
Technical University of Munich

March 8, 2018

Outline

Application background

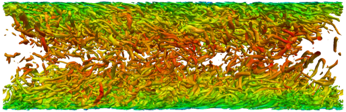
Matrix-free algorithm

Performance modeling

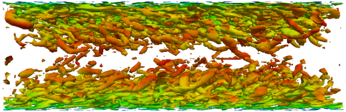
Summary

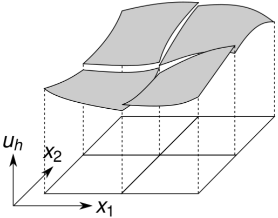
- Complex fluid dynamics simulations
- Code development based on `deal.II` library, `dealii.org`
- Main interest in incompressible flow at high Reynolds numbers $\rightarrow$ solutions smooth, but fine features $\rightarrow$ high resolution
- High-order polynomials within elements
- Fluxes at the element boundaries to weakly enforce continuity, allow for upwind fluxes similar to finite volumes
- Built-in numerical dissipation, in particular in underresolved scenarios
  - fewer degrees of freedom
  - particularly attractive for convection-dominated flows due to low dispersion errors

Visualization: turbulent flow in channel at $\mathrm{Re}_\tau = 590$, vortices visualized by lambda2-criterion DNS, 64^3 mesh with $k = 4$



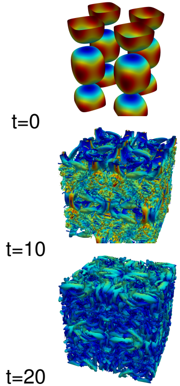
Implicit LES, 16^3 mesh with $k = 4$



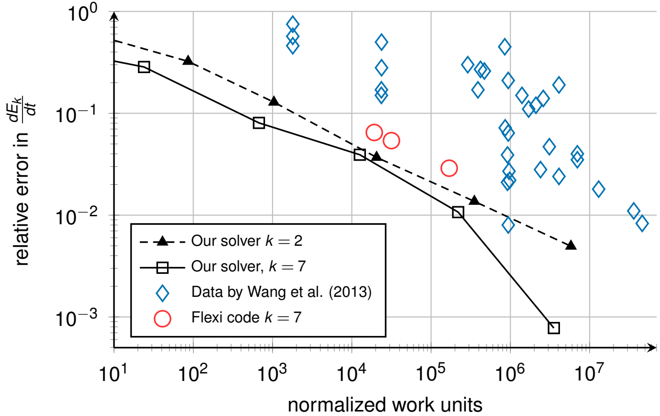


3D Taylor–Green vortex problem

Taylor–Green vortex at Re = 1600: iso-contours of q-criterion (value 0.1) colored by velocity magnitude



Opportunities with tuned implementation: We are **one order of magnitude faster** than all results from Wang et al. (2013), normalized run time



Wang, Fidkowski et al., High-order CFD methods: current status and perspective, *Int. J. Numer. Meth. Fluids* 72(8), 2013

Fehn, Wall, Kronbichler, Efficiency of high-performance discontinuous Galerkin spectral element methods for under-resolved turbulent incompressible flows, arXiv:1802.01439, 2018

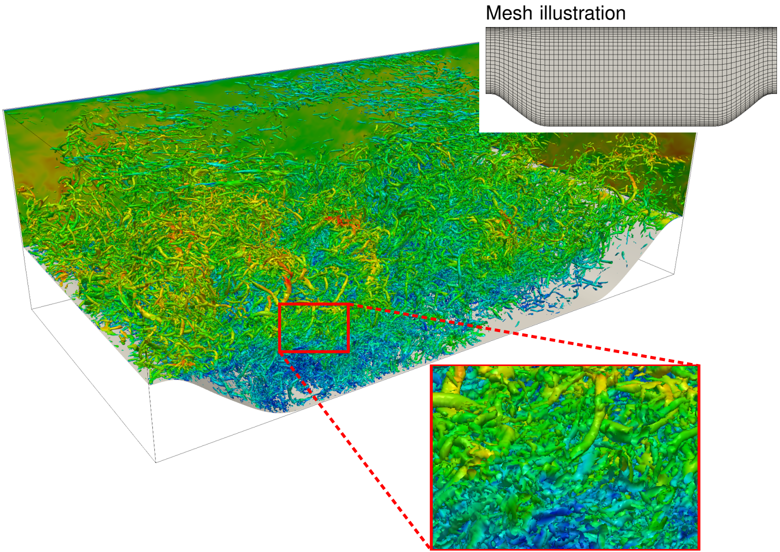
Compressible DG code Flexi: https://github.com/flexi-framework/flexi

High-order geometry example: Flow over periodic hills

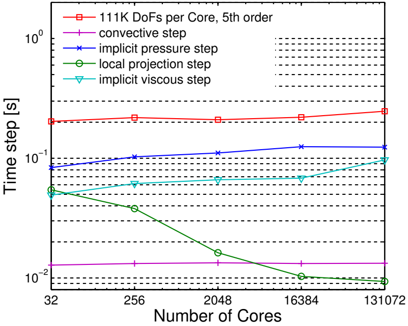
- High order usually includes high-order geometry: manifold descriptions
- Support for unstructured and adaptive meshes through `deal.II` and `p4est` libraries

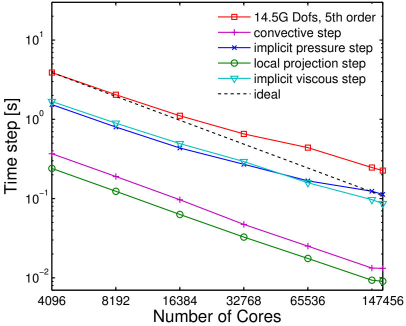
- Computations with up to 800m DoFs and 9m time steps

Krank, Kronbichler, Wall, Direct numerical simulation of flow over periodic hills up to $\mathrm{Re}_\tau = 10,595$, submitted, 2017



Implementation: Scales to 147k cores





Weak and strong scaling on SuperMUC (9 216 nodes with 16 Sandy Bridge cores each)

- Very good scaling to largest size. Algorithmic components:
  - Explicit convective step (typical explicit time stepping)
  - Solution of pressure Poisson equation (**geometric multigrid**)
  - Projection step + Helmholtz equation: CG solver preconditioned by inverse mass matrix

Krank, Fehn, Wall, Kronbichler, A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *J. Comput. Phys.*, 348, 2017

Symmetric interior penalty discretization of the Laplacian $\nabla^2 u$:

$$\sum_{K \in \text{cells}} (\nabla \varphi_i, \nabla u)_K - \left\langle \varphi_i, \frac{\mathbf{n} \cdot (\nabla u^- + \nabla u^+)}{2} \right\rangle_{\partial K} - \left\langle \mathbf{n} \nabla \varphi_i, \frac{u^- + u^+}{2} \right\rangle_{\partial K} + \langle \varphi_i, \tau(u^- - u^+) \rangle_{\partial K}$$

Notation in face integrals $\langle \cdot, \cdot \rangle_{\partial K}$

- u^- is solution inside element K
- u^+ is solution on neighbor over the face
- At boundary, u^+ from b.c., e.g. $u^+ = -u^-$

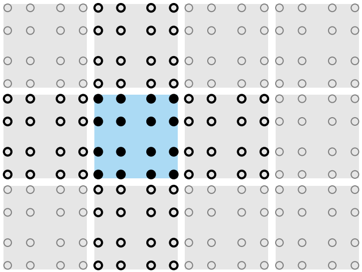
Matrix-vector product $v = Au$
Entry v_i generated by testing with φ_i

Matrix-free algorithm: loop over cells K

- Extract local vector values on cell: $u_K = P_K u$
- Compute cell integral $(\nabla \varphi_i, \nabla u)$ from local interpolation of u_K, $i = 1, \ldots, (k+1)^d$
- Loop over all faces, $f = 1, \ldots, 2d$:
  - Load neighboring values $u^+_{K,f}$
  - Compute face integral contributions and add to cell integral
- Write contribution back into vector

Kronbichler, Kormann, A generic interface for parallel finite element operator application. *Comput. Fluids* 63 (2012)
Kronbichler, Kormann, Fast matrix-free evaluation of discontinuous Galerkin finite element operators. arXiv:1711.03590 (2017)

Illustration for $k = 3$ (4th order) in 2D



- ○ Degrees of freedom in mesh
- **o** Read access element i, j
- ● Read + write access element i, j

Basic matrix version:

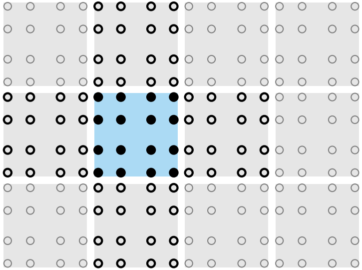
$$v_{i,j} = A^{0,0} u_{i,j} + A^{+,0} u_{i+1,j} + A^{-,0} u_{i-1,j} + A^{0,+} u_{i,j+1} + A^{0,-} u_{i-1,j-1}$$

Full matrix complexity $(k+1)^{2d}$: Very inefficient at high degree k in 3D

Tensor product of 1D matrices

$$A^{0,0} = A^{0}_{1\mathrm{D}} \otimes M_{1\mathrm{D}} + M_{1\mathrm{D}} \otimes A^{0}_{1\mathrm{D}}$$

Illustration for $k = 3$ (4th order) in 2D



- ○ Degrees of freedom in mesh
- **o** Read access element i,j
- ● Read + write access element i,j

Basic matrix version:

$$v_{i,j} = A^{0,0}u_{i,j} + A^{+,0}u_{i+1,j} + A^{-,0}u_{i-1,j} + A^{0,+}u_{i,j+1} + A^{0,-}u_{i-1,j-1}$$

Full matrix complexity $(k+1)^{2d}$: Very inefficient at high degree k in 3D

Tensor product of 1D matrices

$$A^{0,0} = A^0_{1\text{D}} \otimes M_{1\text{D}} + M_{1\text{D}} \otimes A^0_{1\text{D}}$$

More efficient evaluation with sum factorization

$$v_{i,j} = \left(A^0_{1\text{D}} U_{i,j} + A^+_{1\text{D}} U_{i+1,j} + A^-_{1\text{D}} U_{i-1,j}\right) M^{\mathrm{T}}_{1\text{D}} + M_{1\text{D}} \left(U_{i,j} A^{0,\mathrm{T}}_{1\text{D}} + U_{i,j+1} A^{+,\mathrm{T}}_{1\text{D}} + U_{i,j-1} A^{-,\mathrm{T}}_{1\text{D}}\right)$$

$U_{i,j}$ column-major matrification of $u_{i,j}$

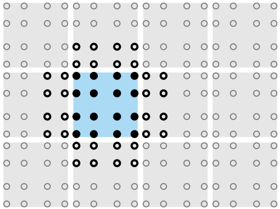
Complexity: $d(k+1)^{d+1}$

Initial optimization: Choice of basis

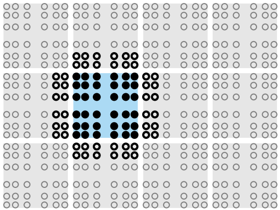
Evaluation of $v_{i,j} = \left(A^0_{1D} U_{i,j} + A^+_{1D} U_{i+1,j} + A^-_{1D} U_{i-1,j}\right) M^T_{1D} + M_{1D} \left(U_{i,j} A^{0,T}_{1D} + U_{i,j+1} A^{+,T}_{1D} + U_{i,j-1} A^{-,T}_{1D}\right)$

- Choice 1 (spectral elements): Use basis where M is diagonal
  - Full neighbor pulled in (possible indirect addressing)
  - Interpolation matrix from neighbor with $(k+1)^d$ points
  - Total number of tensor product interpolations: 6 in 2D, 9 in 3D
- Choice 2: Basis with two 1D shape functions have $\phi_i(0) \neq 0$ and $\phi_i'(0) \neq 0$ (Hermite)
  - Coupling matrices cheaper
  - Cheaper despite additional mass matrices
  - Total number of tensor product interpolations: 4 in 2D, 8 in 3D

Data access Hermite basis $k = 3$



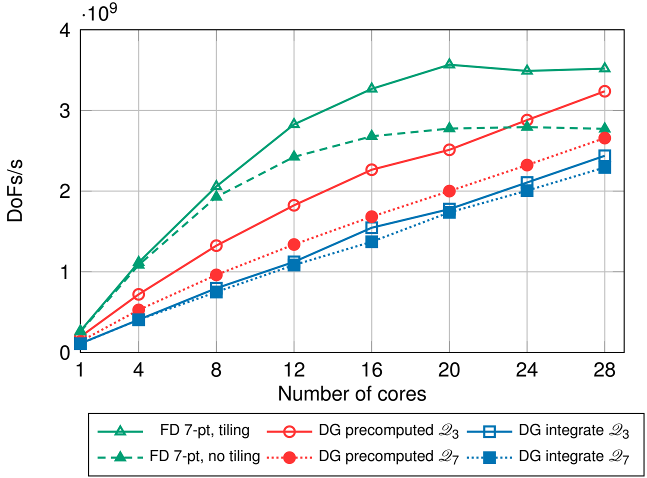
Data access Hermite-like $k = 5$



Choice 2 gives (much) better performance

Comparison to finite differences: throughput

Evaluation on Cartesian mesh on 2 $\times$ 14 core Intel Broadwell Xeon E5-2690 v4



Upper performance limit: 4.7 DoFs/s (read input vector from RAM at 112 GB/s, read + write output vector)

DG operator evaluation almost *for free* because **additional arithmetics on cacheable data**, mostly hidden behind transfer on otherwise memory-starved system

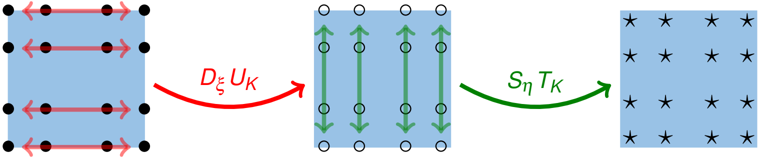
Tuned finite difference stencils with tiling together with K.-R. Wichmann, W. A. Wall

- Final stencil only separable into precomputed 1D matrices A^0_{1D}, A^+_{1D}, A^-_{1D}, M_{1D} for axis-aligned (Cartesian) meshes and constant coefficients
- Want to address numerical integration for general geometries and nonlinear operators
- Sum factorization not in final matrix but for transformation between vector entries and quadrature points (established by spectral element community)

Generic matrix-free implementation with integration

Efficient matrix-free implementation using deal.II[1] [2] `www.dealii.org`

General layout for interpolation into quadrature points:



Vector values u_K on nodes

$\frac{\partial}{\partial \xi} u^h$ on quadrature points

$$\frac{\partial}{\partial \xi} u^h(\xi_q, \eta_q)\Big|_{\text{q\_points}} = (D_\xi \otimes S_\eta)\mathbf{u}_K$$

Dense matrix-matrix multiply $D_\xi \; \mathbf{U}_K \; S_\eta^\mathrm{T}$

Evaluation cost: $\mathcal{O}((k+1)^4)$ per element (degree k in 3D)

Naive evaluation: $\mathcal{O}((k+1)^6)$

[1] Kronbichler, Kormann, A generic interface for parallel cell-based finite element operator application, *Comput. Fluids* 63 (2012)

[2] Kronbichler, Kormann, Fast matrix-free evaluation of discontinuous Galerkin finite element operators, arXiv:1711.03590 (2017)

Performance characterization with DG operator evaluation

Objective assessment much less mature than in finite difference or lattice Boltzmann communities

- What is the expected number of arithmetic operations?
  - Sum factorization most beneficial on quad/hex elements
  - Evaluation without sum factorization is too much work for $k \geq 3$ (except triangles)
  - Evaluate the geometry on the fly or load precomputed data?
  - Which basis and what evaluation techniques appropriate?
- What is the expected memory transfer?
  - Some projects separate face integrals into global data structure $\rightarrow$ several sweeps through data for a single operator evaluation
  - Some projects split loops in quadrature into global loops
- Must use implementation-independent **throughput metric: degrees of freedom per second (DoFs/s)**
- No DG cross-project benchmarking yet (?) corresponding recent CEED initiative for FEM `http://ceed.exascaleproject.org/bps`

Performance expectations

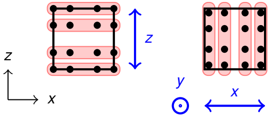
- **Ideal memory access:** A single load to source, a single store (or load+store with read-for-ownership)
- **Arithmetics:** Around 120–250 operations per DoF
- **Balance:** 5–10 FLOPs/Byte

Must consider both compute and memory access!

Only cell integrals, only compute phase of 3D Laplacian, 2×14 core Intel Broadwell E5-2690 v4, 2.9 GHz (incl AVX-2 turbo)

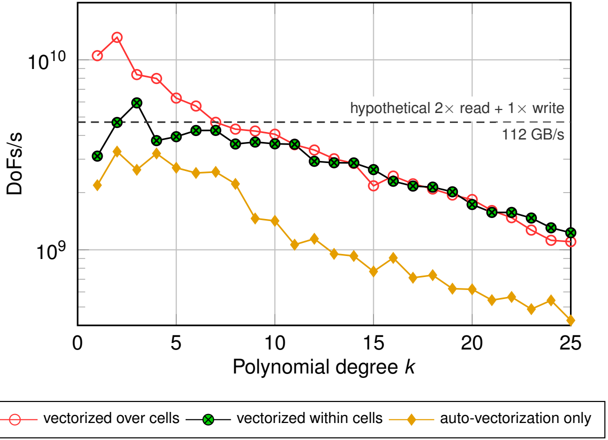
Forced vectorization within cells, 4-times blocking in z

Values in nodes   Values in q-points



Vectorize over several cells

50	54	58	62	51	55	59	63
34	38	42	46	35	39	43	47
18	22	26	30	19	23	27	31
2	6	10	14	3	7	11	15
48	52	56	60	49	53	57	61
32	36	40	44	33	37	41	45
16	20	24	28	17	21	25	29
0	4	8	12	1	5	9	13



Conclusion: Must explicitly vectorize

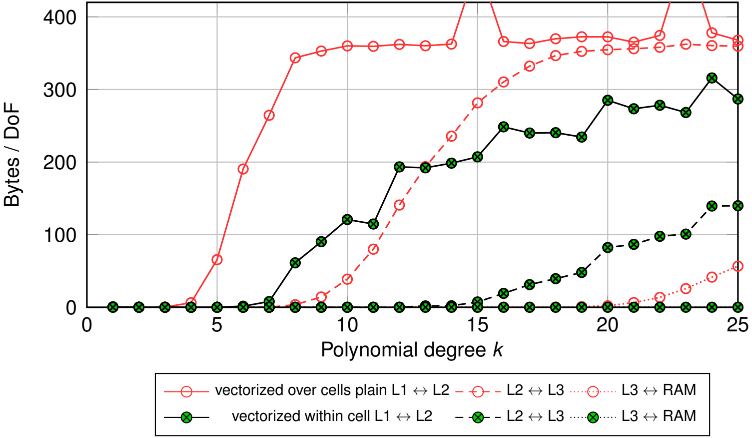
Doesn’t vectorization over cells drown caches?

Yes, but there are outer level caches...

Higher degree: more cache access → more arithmetics → **no big impact** for $k < 20$ on Broadwell (or $k < 12$ on KNL)

Cache transfer analysis with the Likwid tool `github.com/RRZE-HPC/likwid` (hardware performance counters)

Performance can be improved by tiling within tensor product, see Kronbichler & Kormann, arXiv:1711.03590



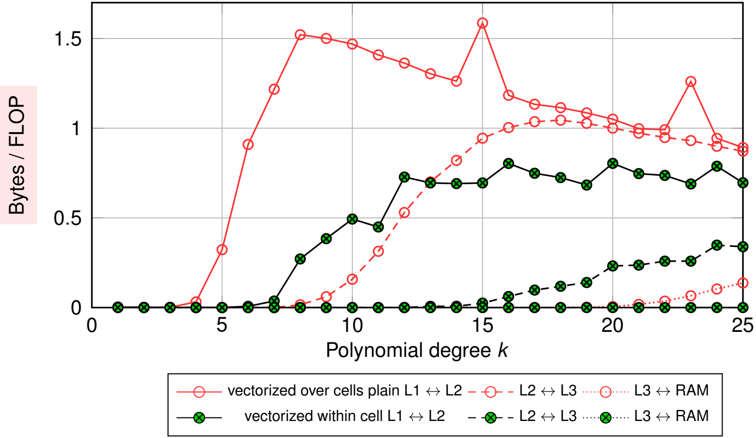
Doesn't vectorization over cells drown caches?

Yes, but there are outer level caches...

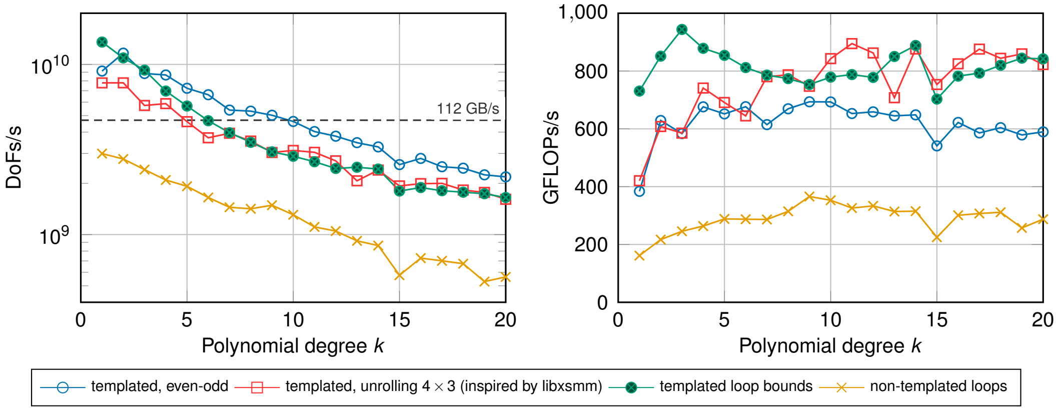
Higher degree: more cache access $\rightarrow$ more arithmetics $\rightarrow$ **no big impact** for $k < 20$ on Broadwell (or $k < 12$ on KNL)

Cache transfer analysis with the Likwid tool `github.com/RRZE-HPC/likwid` (hardware performance counters)

Performance can be improved by tiling within tensor product, see Kronbichler & Kormann, arXiv:1711.03590



Test of throughput on 2 × 14 core Intel Broadwell E5-2690 v4, 2.9 GHz (incl AVX-2 turbo), 1,299 GFLOPs/s arithmetic peak

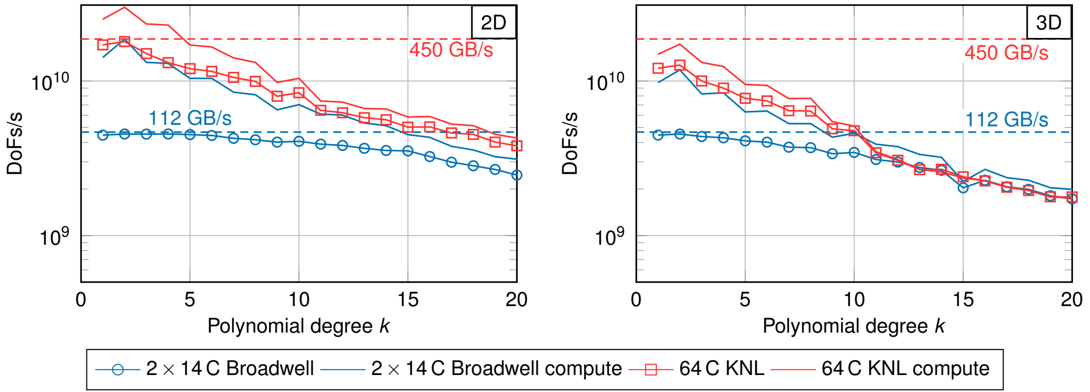


Favorite method: even-odd decomposition[3], compile-time loop bounds

[3] Kopriva, *Implementing spectral methods for partial differential equations*, Springer, 2009

Cell integrals with vector transfer

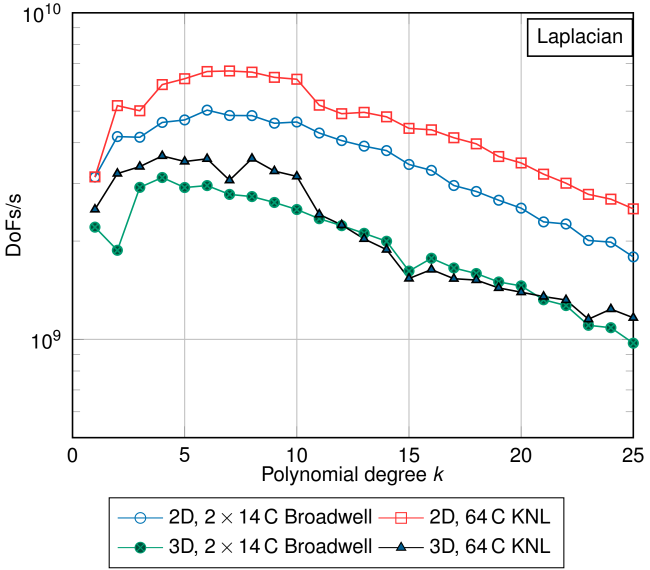
Compare 2×14 core Intel Broadwell E5-2690 v4 (at 2.9 GHz) and 64 core Intel Knights Landing 7210 (at 1.1 GHz), problem size 8M to 56M



Both KNL and Broadwell reach $> 50\%$ of arithmetic peak for instruction mix with 35% FMA, 30% add, 35% multiply

Arithmetics including face integrals

- Face integrals constitute significant portion of arithmetics
  - 80% at $k = 1$
  - 52% at $k = 5$
  - 35% at $k = 11$
- Involve additional gather access into neighbors for vectorization over cells not displayed here

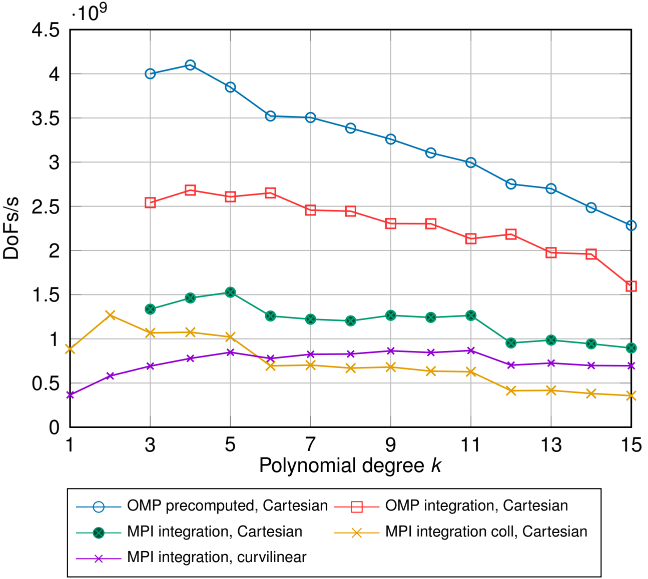


Throughput of operator evaluation: 3D DG-SIP Laplacian

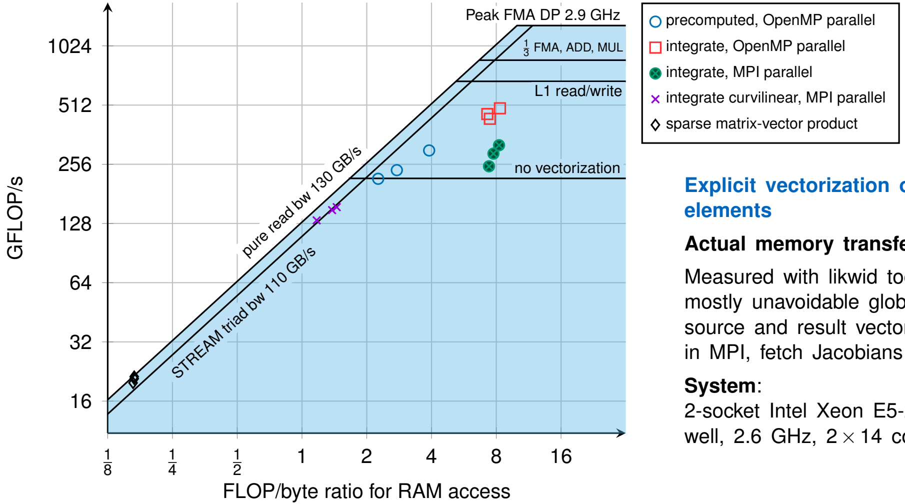
Two parallelization options:

- OpenMP: direct access to neighbors in shared memory
- MPI: Must explicitly send data, pack & unpack and MPI routines take around 35% of time at $k = 6$
- Shared memory model clearly better

System: 2×14 core Intel Broadwell E5-2690 v4, utilize streaming stores



Performance characterization by roofline



Explicit vectorization over several elements

Actual memory transfer

Measured with likwid tool; transfers mostly unavoidable global data: Fetch source and result vectors, pack/unpack in MPI, fetch Jacobians (curvilinear)

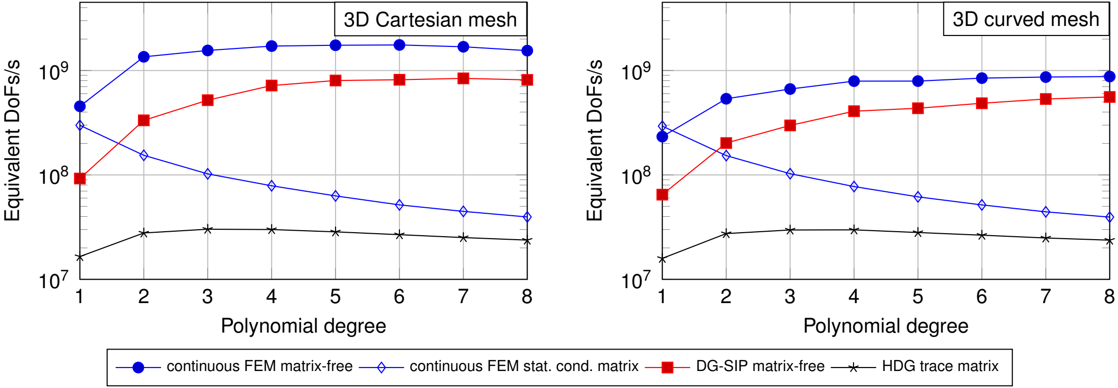
System:
2-socket Intel Xeon E5-2690 v4 (Broadwell, 2.6 GHz, 2×14 cores)

Arithmetic balance for polynomial degrees $k = 3, 5, 9$

Comparison to other schemes

Continuous finite elements, DG-SIP, hybridizable discontinuous Galerkin (HDG) representing efficient sparse matrix-based scheme

Throughput measured using run time against $n_{\text{cells}}(k+1)$ "equivalent" DoFs to make different discretizations comparable[4]



[4] Kronbichler, Wall, A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, *arXiv:1611.03029* (2016)

Summary

- Comprehensive approach to tuning – not just FLOPs or GB/s but
  - choose mathematical formulation considering both data access and computations
  - apply arithmetic optimizations such as even-odd decomposition guided by throughput, not GFLOPs/s
  - vectorization over cells very efficient on Intel machines (but **not** on GPUs, see Ljungkvist & Kronbichler, 2017)
- Realization by matrix-free approach with sum factorization on hexahedra is highly efficient! Reaches 3 billions DoFs/s = finite different performance
- Challenge 1: Memory transfer of geometry
- Challenge 2: BLAS-1 vector operations take large share of time
  - More than 60% in conjugate gradient solver with simple precond.
  - More than 30% in Chebyshev smoother with multigrid

Need programming model that interleaves global operations as much as possible

Ljungkvist, Kronbichler, Multigrid for matrix-free finite element computations on graphics processors. Technical Report IT-2017-006, Uppsala University, 2017