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

# Development of an Exascale-Ready Fluid Dynamics Solver with the deal.II Library

Martin Kronbichler

in collaboration with

Niklas Fehn Katharina Kormann Benjamin Krank Peter Münch Dmytro Sashko Wolfgang A. Wall
and the deal.II authors, https://www.dealii.org

Institute for Computational Mechanics
Technical University of Munich, Germany

November 29, 2019

44th ORAP forum, Paris

# Outline

CFD application requirements

Mathematical requirements

Software design

Fast matrix-free operator evaluation

Geometric multigrid

Iterative solver performance

# Outline

CFD application requirements

Mathematical requirements

Software design

Fast matrix-free operator evaluation

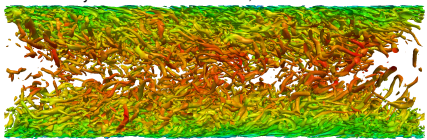
Geometric multigrid

Iterative solver performance

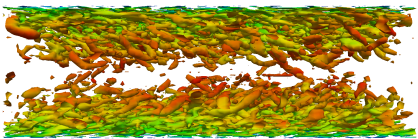
# Target application: CFD

- Complex fluid dynamics simulations
- Main interest in incompressible flow at high Reynolds numbers
  - solutions smooth, but fine features
  - **high resolution**
  - under-resolved setting
- Geometric flexibility
- Large-scale parallel computing capabilities
- High-order finite element and discontinuous Galerkin methods
- **Write a new code**, but use existing mathematical frameworks
  - Code development based on deal.II library, `dealii.org`

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



Under-resolved simulation, $16^3$ mesh with $k = 4$



# Incompressible Navier–Stokes equations

Consider the incompressible Navier–Stokes equations in a domain $\Omega \subset \mathbb{R}^d$

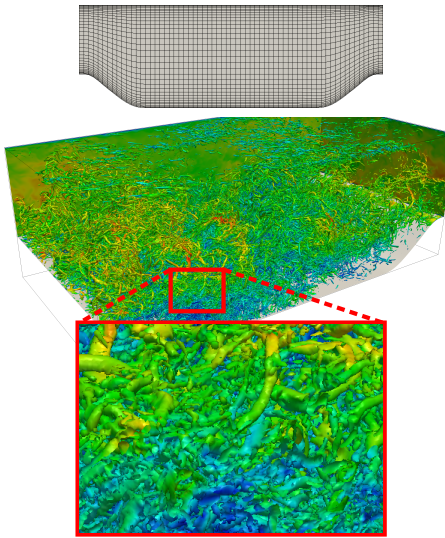
$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) - \nu \nabla \cdot \nabla \mathbf{u} + \nabla p = \mathbf{f} \text{ in } \Omega \times [0, T],$$

$$\nabla \cdot \mathbf{u} = 0 \text{ in } \Omega \times [0, T],$$

where

- $\mathbf{u} = (u_1, ..., u_d)^T$ fluid velocity
- $p$ kinematic pressure
- $\mathbf{f}$ given body force
- $\mathbf{u} \otimes \mathbf{u}$ convective flux in conservative formulation
- $\nu \nabla \mathbf{u}$ viscous flux in Laplace formulation

Mesh illustration



# Outline

CFD application requirements

Mathematical requirements

Software design

Fast matrix-free operator evaluation

Geometric multigrid

Iterative solver performance

# Mathematical requirements

- Flexibility with respect to temporal discretization
  - Coupled/monolithic solution approach
  - Explicit vs. implicit discretization of terms
  - Splitting methods: velocity and pressure solved by separate (simpler) linear systems
- Robust spatial discretization
  - Divergence-free condition $\nabla \cdot \mathbf{u} = 0$ crucial for robustness in under-resolved scenarios
  - Benefit of discontinuous Galerkin methods
    - built-in numerical dissipation due to weak coupling between elements
    - low dispersion errors
- Compressible or incompressible formulation
- Coupling with other equations
  - Transport of scalar fields
  - Transport of particles (work in progress)

- **Coupled/monolithic solution approach** with explicit formulation of convective term

$$\frac{\gamma_0 \mathbf{u}^{n+1} - \sum_{i=0}^{J-1} \left(\alpha_i \mathbf{u}^{n-i}\right)}{\Delta t} - \nabla \cdot \left(\nu \nabla \mathbf{u}^{n+1}\right) + \nabla p^{n+1} = -\sum_{i=0}^{J-1} \left(\beta_i \nabla \cdot \left(\mathbf{u}^{n-i} \otimes \mathbf{u}^{n-i}\right)\right) + \mathbf{f}(t_{n+1})$$

$$\nabla \cdot \mathbf{u}^{n+1} = 0$$

- **High-order dual splitting scheme**[1]

$$\frac{\gamma_0 \hat{\mathbf{u}} - \sum_{i=0}^{J-1} \left(\alpha_i \mathbf{u}^{n-i}\right)}{\Delta t} = -\sum_{i=0}^{J-1} \left(\beta_i \nabla \cdot \left(\mathbf{u}^{n-i} \otimes \mathbf{u}^{n-i}\right)\right) + \mathbf{f}(t_{n+1})$$

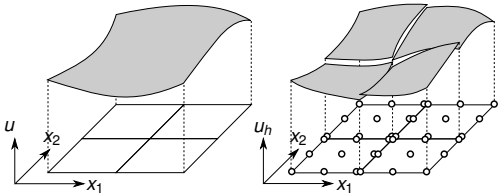
$$-\nabla^2 p^{n+1} = -\frac{\gamma_0}{\Delta t} \nabla \cdot \hat{\mathbf{u}}$$

$$\hat{\hat{\mathbf{u}}} = \hat{\mathbf{u}} - \frac{\Delta t}{\gamma_0} \nabla p^{n+1}$$

$$\frac{\gamma_0}{\Delta t} \mathbf{u}^{n+1} - \nabla \cdot \left(\nu \nabla \mathbf{u}^{n+1}\right) = \frac{\gamma_0}{\Delta t} \hat{\hat{\mathbf{u}}}$$

[1] Karniadakis, Israeli, Orszag, High-order splitting methods for the incompressible Navier–Stokes equations. *J Comput Phys* 97(2), 1991

# Spatial DG discretization – function spaces




- Fully discontinuous, i.e., $L^2$-conforming formulation

$$\mathscr{V}_h^u = \left\{ \mathbf{u}_h \in [L_2(\Omega_h)]^d \,:\, \mathbf{u}_h(\mathbf{x}(\xi))|_{\Omega_e} = \tilde{\mathbf{u}}_h^e(\xi)|_{\tilde{\Omega}_e} \in \mathscr{V}_{h,e}^u = [\mathscr{P}_{k_u}(\tilde{\Omega}_e)]^d \,,\ \forall e = 1,\ldots,N_{\text{el}} \right\}$$

$$\mathscr{V}_h^p = \left\{ p_h \in L_2(\Omega_h) \,:\, p_h(\mathbf{x}(\xi))|_{\Omega_e} = \tilde{p}_h^e(\xi)|_{\tilde{\Omega}_e} \in \mathscr{V}_{h,e}^p = \mathscr{P}_{k_p}(\tilde{\Omega}_e) \,,\ \forall e = 1,\ldots,N_{\text{el}} \right\}$$

- Mixed-order polynomials $k_p = k_u - 1$ (inf–sup)
- Tensor product shape functions for hexahedral element shapes
- Nodal Lagrange polynomials based on the Legendre–Gauss–Lobatto (LGL) points
- Gaussian quadrature ensuring exact integration on affine elements, e.g., 3/2-overintegration for nonlinear convective term

Find $\mathbf{u}_h \in \mathscr{V}_h^u$, $p_h \in \mathscr{V}_h^p$ such that for all $(\mathbf{v}_h, q_h) \in \mathscr{V}_{h,e}^u \times \mathscr{V}_{h,e}^p$ and for all elements $e$

$$m_{h,u}^e\left(\mathbf{v}_h, \frac{\partial \mathbf{u}_h}{\partial t}\right) + c_h^e(\mathbf{v}_h, \mathbf{u}_h) + v_h^e(\mathbf{v}_h, \mathbf{u}_h) + g_h^e(\mathbf{v}_h, p_h) = b_h^e(\mathbf{v}_h, \mathbf{f}(t)) \,,$$

$$-d_h^e(q_h, \mathbf{u}_h) = 0 \,.$$

DG discretization of basic operators:[2]

- Convective operator $c_h^e(\mathbf{v}_h, \mathbf{u}_h)$: Local Lax–Friedrichs flux
- Viscous operator $v_h^e(\mathbf{v}_h, \mathbf{u}_h)$: Symmetric interior penalty method
- Gradient operator $g_h^e(\mathbf{v}_h, p_h)$: Central fluxes
- Divergence operator $d_h^e(q_h, \mathbf{u}_h)$: Central fluxes

[2] Fehn, Wall, Kronbichler, On the stability of projection methods for the incompressible Navier–Stokes equations based on high-order discontinuous Galerkin discretizations, *J Comput Phys 348:634–659*, 2017.

- Robust method in under-resolved regime $\mathrm{Re}_h = \frac{h\mathbf{u}}{k\nu} > 1$ requires stabilized approach, realized e.g. by **consistent penalty terms**:[3]

$$m^e_{h,u}\left(\mathbf{v}_h, \frac{\partial \mathbf{u}_h}{\partial t}\right) + c^e_h(\mathbf{v}_h, \mathbf{u}_h) + v^e_h(\mathbf{v}_h, \mathbf{u}_h) + g^e_h(\mathbf{v}_h, p_h) + a^e_{\mathrm{D}}(\mathbf{v}_h, \mathbf{u}_h) + a^e_{\mathrm{C}}(\mathbf{v}_h, \mathbf{u}_h) = b^e_h(\mathbf{v}_h, \mathbf{f}(t)) \,,$$
$$-d^e_h(q_h, \mathbf{u}_h) = 0 \,,$$

divergence and continuity penalty terms:

$$a^e_{\mathrm{D}}(\mathbf{v}_h, \mathbf{u}_h) = (\nabla \cdot \mathbf{v}_h, \tau_{\mathrm{D}} \nabla \cdot \mathbf{u}_h)_{\Omega_e} \,, \quad a^e_{\mathrm{C}}(\mathbf{v}_h, \mathbf{u}_h) = \left(\mathbf{v}_h \cdot \mathbf{n}, \tau_{\mathrm{C},f}\,[\mathbf{u}_h] \cdot \mathbf{n}\right)_{\partial\Omega_e \setminus \Gamma_h} \,.$$

- Penalty term mimic properties of $H(\mathrm{div})$ conforming spaces[4] with exact fulfillment of $\nabla \cdot \mathbf{u} = 0$ condition, e.g. Raviart–Thomas elements on hexahedra
- Goal: switch between $L_2$ conforming + stabilization and RT elements in implementation

[3] Fehn, Wall, Kronbichler, Robust and efficient discontinuous Galerkin methods for under-resolved turbulent incompressible flows, *J Comput Phys 372:667–693*, 2018.

[4] Fehn, Kronbichler, Lehrenfeld, Lube, Schroeder, High-order DG solvers for underresolved turbulent incompressible flows: A comparison of $L_2$ and $H(\mathrm{div})$ methods, *Int J Numer Meth Fluids 91:533–556*, 2019.

# Outline

CFD application requirements

Mathematical requirements

Software design

Fast matrix-free operator evaluation

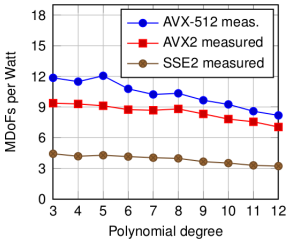
Geometric multigrid

Iterative solver performance

# Solver and hardware requirements towards exascale

- Application & mathematical flexibility
- Selection of most efficient discretization parameters
  - Meshing (high-order curved, adaptive, . . . )
  - Polynomial degree
  - Time stepping
- Selection of most efficient iterative solver
  - $> 10^9$ spatial unknowns, millions of time steps $\rightarrow$ **optimal-complexity solvers** essential
  - Example: Multigrid solver for pressure Poisson equation
- Selection of most efficient implementation
  - Matrix-free vs matrix-based $\rightarrow$ **avoid memory wall**
  - Usage of most efficient instruction set – e.g. vectorization with AVX/AVX-512
  - Novel architectures: Intel Xeon Phi, NVIDIA GPUs, ARM SVE, . . .
  - Scalability to 10,000+ nodes

SuperMUC-NG supercomputer (top) (source: www.lrz.de) and efficiency of various instruction sets on Intel Skylake-X for DG operator evaluation (bottom)

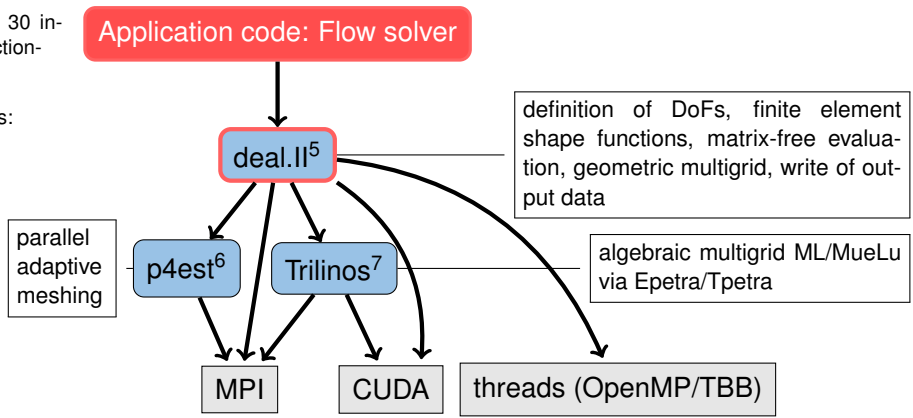


# Design philosophy: Composition of libraries

C++ Source: 70k lines
8 compressible setups, 30 incompressible, 7 convection-diffusion: 32k lines

C++ headers + sources:
660k lines of code
11,800 test cases
we use $\sim$ 200k lines

we use $\sim$ 200k lines




[5] www.dealii.org/
[6] www.p4est.org/: C. Burstedde, L.C. Wilcox, O. Ghattas
[7] www.trilinos.org: M. Heroux et al.

# Code activities in my group – library approach

**CFD application code development**

- Implementation of CFD equations
  - Based on appropriate data structures from deal.II library
- Simulation control based on library components
  - Iterative solvers, multigrid suited for equations at hand
  - Embeds equation evaluator into existing interfaces

**Code development within deal.II finite element library**

- Building blocks targeting particular hardware (SIMD, CUDA)
- MPI parallelization of mesh
- Interface to linear algebra libraries like Trilinos/PETSc
- Move common functionality to deal.II library
  - Separation of concerns
  - Forces split of application-specific vs generic FEM toolbox
  - Reuse among many applications with more resources

# Outline

CFD application requirements

Mathematical requirements

Software design

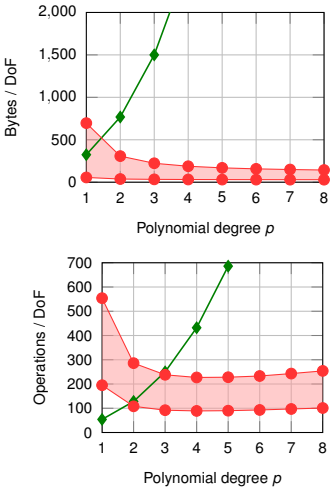
Fast matrix-free operator evaluation

Geometric multigrid

Iterative solver performance

# Motivation for matrix-free algorithms

- Iterative solvers (or explicit time stepping) typically spend 60–95% of time in matrix-vector product
- Classical approach: **sparse matrix-vector product**
- Performance limit is **memory bandwidth**
- SpMV: 0.16–0.25 Flop/Byte
  - Bad already on 2005's hardware
  - Terrible in 2019 and in foreseeable future
  - Modern CPUs/GPUs provide 3–20 Flop/Byte
- Matrix-free algorithm: **less memory transfer**
  - Stencil-based (low order, meshes with structure)
  - Fast computation of FEM integrals $\rightarrow$ my choice
- Sum factorization enables $\mathcal{O}(p)$ complexity per DoF
- Matrix-free: 1–8 Flop/Byte
- Interesting: Both memory access and arithmetics matter
- Closely related to US CEED project



# Matrix-free operator evaluation by fast integration

**Matrix-vector product**

matrix-based:

$$\begin{cases} A = \sum_{e=1}^{N_{\text{el}}} P_e^{\mathsf{T}} A_e P_e & \text{(assembly)} \\ v = Au & \text{(matrix-vector product within iterative solver)} \end{cases}$$

matrix-free:

$$v = \sum_{e=1}^{N_{\text{el}}} P_e^{\mathsf{T}} A_e (P_e u)$$

**implication**: assembly facilities within iterative solvers

Matrix-free evaluation of FEM Laplacian

- $v = 0$
- loop over elements $e = 1, \ldots, N_{\text{el}}$
  - (i) Extract local vector values: $u_e = P_e u$
  - (ii) Apply operation locally by integration: $v_K = A_K u_K$ (**without** forming $A_K$)
  - (iii) Sum results from (ii) into the global solution vector: $v = v + P_e^{\mathsf{T}} v_e$

**Design goals**:

- Data locality for higher arithmetic intensity: single sweep through data
- Absolute performance in **unknowns per second** (DoFs/s), not maximal GFlop/s or GB/s

M. Kronbichler, K. Kormann, A generic interface for parallel finite element operator application. *Comput. Fluids* 63:135–147, 2012
M. Kronbichler, K. Kormann, Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM TOMS* 45(3), 29, 2019

# Matrix-vector product on cell by integration

Contribution of cell $K$ to matrix-vector product for finite element Laplacian

$$(A_K u_K)_j = \int_K \nabla_{\mathbf{x}} \phi_j \cdot \nabla_{\mathbf{x}} u^h d\mathbf{x} \approx \sum_q w_q \det J_q \nabla_{\mathbf{x}} \phi_j \cdot \nabla_{\mathbf{x}} u^h \Big|_{\mathbf{x}=\mathbf{x}_q}$$

$$= \sum_q \nabla_\xi \phi_j J_q^{-1} (w_q \det J_q) J_q^{-\mathsf{T}} \sum_i \nabla_\xi \phi_i u_{K,i} \Big|_{\mathbf{x}=\mathbf{x}_q}, \quad j = 1, \ldots, \text{cell_dofs}$$

(a) Compute unit cell gradients $\nabla_\xi u^h = \sum (\nabla_\xi \phi_i) u_{K,i}$ at all quadrature points

(b) At each quadrature point, apply geometry $J_q^{-\mathsf{T}}$, multiply by quadrature weight and Jacobian determinant, apply geometry for test function $J_q^{-1}$

(c) Test by unit cell gradients of all basis functions and sum over quadrature points

**Matrix notation:**

$$v_K = A_K u_K$$
$$= S^{\mathsf{T}} W S u_K$$

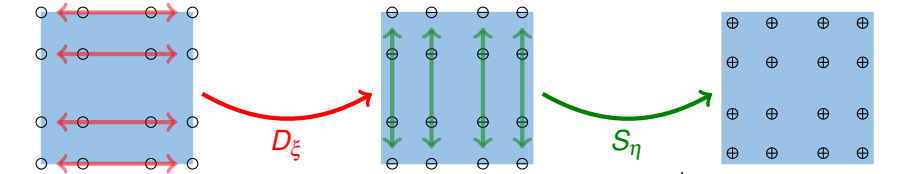
with

$S_{qi} = \nabla_\xi \phi_i \big|_{\xi_q}$

$W_{qq} = J_q^{-1} (w_q \det J_q) J_q^{-\mathsf{T}}$

# Fast interpolation and integration: sum factorization

- Efficient evaluation of $S$ and $S^\mathsf{T}$ matrices with structure $S_{\text{3D}} = \begin{bmatrix} S_\zeta \otimes S_\eta \otimes D_\xi \\ S_\zeta \otimes D_\eta \otimes S_\xi \\ D_\zeta \otimes S_\eta \otimes S_\xi \end{bmatrix}$
- Ideas from spectral elements (1980s) for tensor product shape functions and tensor product quadrature
- Visualization of interpolation of $\frac{\partial u}{\partial \xi}$ with $\mathscr{Q}_3$ element (Lagrange basis) $S_\eta \otimes D_\xi$: successively apply 1D kernels

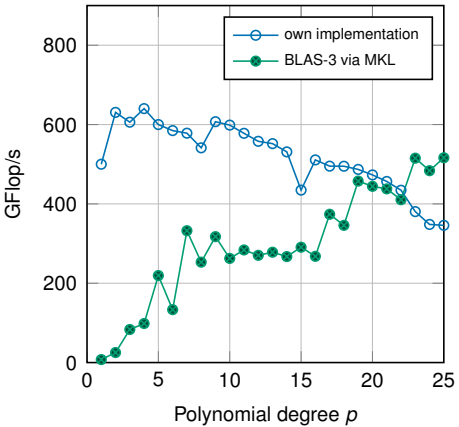



Tensor-based evaluation reduces evaluation cost from $4^4$ to $2 \times 4^3$

In general for degree $p$ and dimension $d$: $\mathscr{O}((p+1)^{2d})$ to $\mathscr{O}(d(p+1)^{d+1})$

# Sum factorization – implement ourselves or use performance libraries?

- Sum factorization consists of series of matrix-matrix multiplication of size $(p+1) \times (p+1)$ and $(p+1) \times (p+1)^2$ for polynomial degree $p$
- Standard BLAS optimized for large $N$
  - bad here due to function call overhead & rearrangement cost
- Own implementation with SIMD and unrolled loops (compiler via templated C++ code)

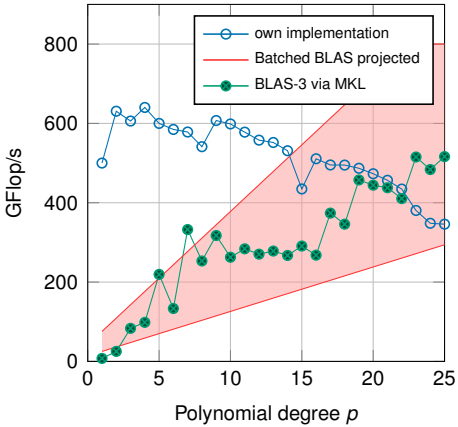



**System**: $2 \times 14$ Intel Broadwell 2690 v4, peak: 1.3 TFlop/s, 115 GB/s

Own implementation 2–20 times faster in **interesting regime** $2 \leq p \leq 10$

# Sum factorization – implement ourselves or use performance libraries?

- Sum factorization consists of series of matrix-matrix multiplication of size $(p+1)\times(p+1)$ and $(p+1)\times(p+1)^2$ for polynomial degree $p$
- Series of mat-mat! Batched BLAS possible?
  - Combine small mat-mat of all elements
  - **Limit**: Data locality between mat-mat lost
  - Unclear how to best utilize data in caches (CPU) or registers(GPU)
  - Projected performance
    - Full main memory bandwidth
    - Lower bound: No caching between mat-mat
    - Upper bound: Ideal caching of 3 mat-mat
- Own implementation with SIMD and unrolled loops (compiler via templated C++ code)

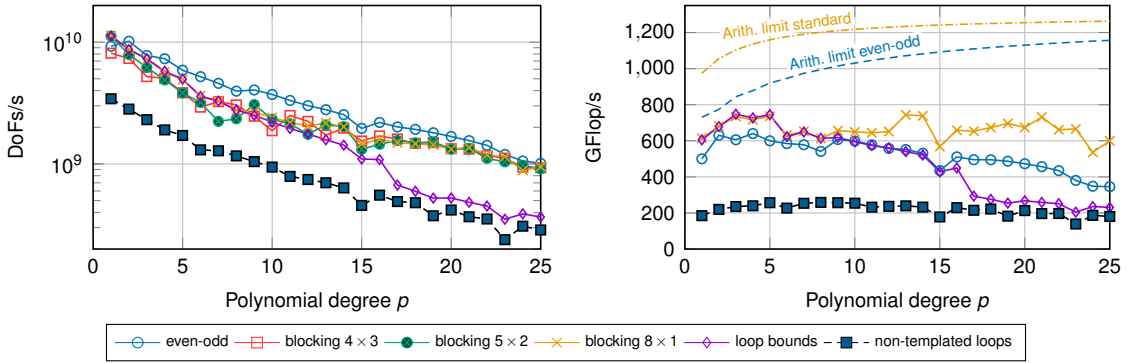



**System**: $2\times14$ Intel Broadwell 2690 v4, peak: 1.3 TFlop/s, 115 GB/s

Own implementation 2–5 times faster in **interesting regime** $2 \leq p \leq 10$

# Optimization of local computations

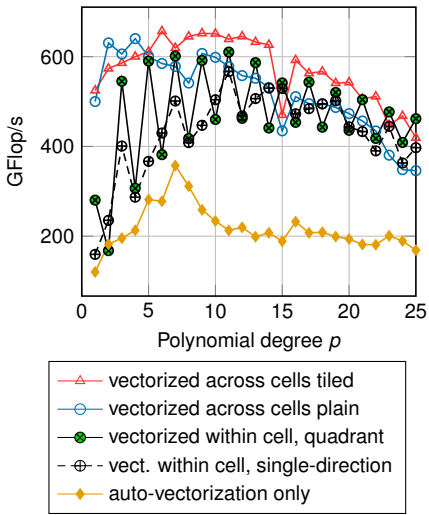
- Optimize for throughput (unknowns processed per second, DoFs/s) rather than GFlop/s
- Local tensor contractions in sum factorization with even-odd decomposition
  $\rightarrow$ symmetry allows us to do less flops
- Example: cell integrals of 3D Laplacian on 28 cores of Intel E5-2690 v4 (Broadwell), peak 1,299 GFlop/s; fastest option (even-odd) not with highest GFlop/s



# Example optimization: SIMD vectorization

- Operations within cell:
  - Mat-mat with stride $1, p+1, (p+1)^2$ among $(p+1)^3$ points
  - Latency effects
  - Auto-vectorization gives poor efficiency
- Manual vectorization necessary for best performance
- Abstraction: Introduce a SIMD array class `VectorizedArray` that overloads `operator +,-,*,/`
  - Concentrates ISA-specific code in single file
  - All other files ISA-agnostic
  - Currently SSE(2), AVX, AVX-512, AltiVec
  - Planned SVE
- Best performance: Vectorize across cells
  - Array-of-struct-of-array data type
  - No chance to do this auto-vectorized?

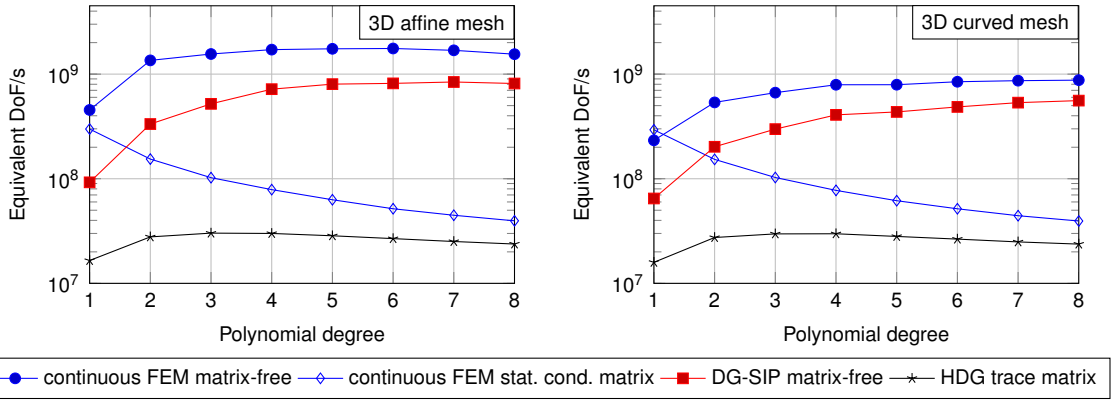
Example on $2 \times 14$ Intel Broadwell cores



# Comparison versus matrix-based: matrix-free wins

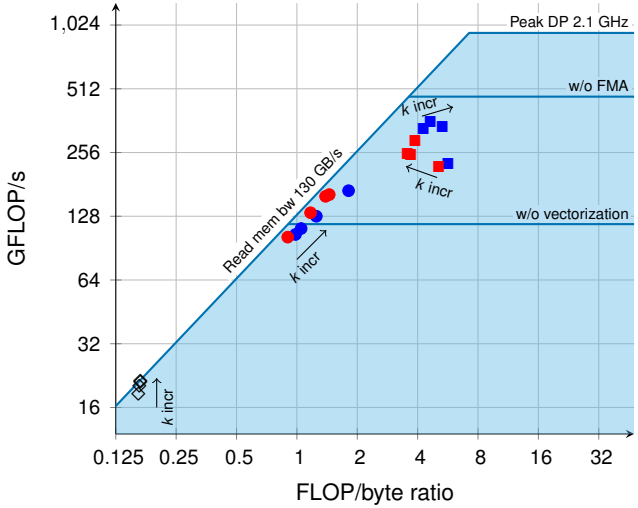
Continuous finite elements, DG-SIP, hybridizable discontinuous Galerkin (HDG) representing efficient sparse matrix-based scheme

Throughput of matrix-vector product measured using run time against $N_{el} k^3$ "equivalent" DoFs to make different discretizations comparable[8]




[8] Kronbichler, Wall, A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, *SIAM J Sci Comput* 40:A3423–48, 2018

# Performance modeling by roofline




- ■ matrix-free continuous FEM, affine
- ● matrix-free continuous FEM, curved
- ■ matrix-free DG-SIP, affine
- ● matrix-free DG-SIP, curved
- ◇ Sparse matrix-vector

**Explicit vectorization over several elements**

**Actual memory transfer**

Measured with likwid tool; transfers mostly unavoidable global data: Fetch source and result vectors, fetch Jacobians

**System**:
2-socket Intel Xeon E5-2690 v4 (Broadwell, 2.6 GHz, $2 \times 14$ cores)

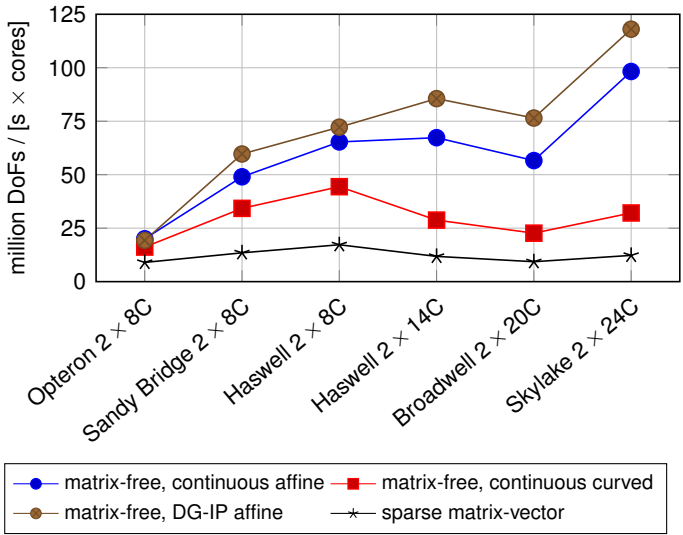
Arithmetic balance for polynomial degrees $p = 1, 3, 5, 8$

# Node-level performance over time

Track performance of operator evaluation **per core** over 10 years of hardware

- Baseline: sparse matrix-vector product, linear elements
- Continuous elements, affine (arithmetic intensive) and high-order deformed (memory intensive), $p = 4$
- Discontinuous elements, affine, degree $p = 4$

Matrix-free and SpMV within a factor 2 in 2010, **more than** $10\times$ **apart today**!



# Performance potential of matrix-free methods

- Matrix-vector product with matrix-free methods at $p = 4, 5, 6$ faster than best known matrix-based method (hybridization/static condensation) by **10**$\times$ and linear FEM by **3**$\times$
- Performance improvement because memory-intensive sparse matrix-vector is traded for computations through integrals
- Despite several years of work, performance is farther away from theoretical potential than for matrix-based methods
- Cannot afford tuning for a single equation/application
- Our effort is integrated into the general-purpose finite element library deal.II at `wwww.dealii.org`
  - Can be shared between multiple applications
  - Independent testing of mathematical ingredients such as interpolation, derivative, integration, etc. from Navier–Stokes equations
  - Needed to implement our own parallel vectors rather than using PETSc or Trilinos
- Separation of concerns keeps application code lean

# CPU code example from deal.II: Cell term for Laplacian

Evaluation of weak form $(\nabla\phi_j, \nabla u)_{\Omega_h}$ representing $v = Au$

```
void cell(MatrixFree<dim> &data,
          Vector &v,
          const Vector &u,
          const std::pair<unsigned int,unsigned int> &range)
{
  FEEvaluation<dim,degree> eval (data);
  for (unsigned int cell=range.first; cell<range.second; ++cell)
    {
      eval.reinit (cell);                                  // set pointers to data
      eval.gather_evaluate(u,                              // read from source
           /*interpolate_values=*/ false,                 // sum factorization
           /*interpolate_gradients=*/ true);
      for (unsigned int q=0; q<phi.n_q_points; ++q)
        eval.submit_gradient (phi.get_gradient(q), q); // equation
      eval.integrate_scatter (false, true,                 // integrate with sum fact
                              dst);                        // sum into result
    }
}
```

# CPU versus GPU code

- CPU and GPU use different codes and class names but similar interfaces → can replace one by the other via `typedef` statements
- Most flexible code on CPU includes some generic part to control what to evaluate → can be hidden by base class
- Can reduce CPU code to operation at quadrature point:

```
      // test expression gradient(u) by nabla phi,
      // i.e., (nabla phi_i, nabla u)_K
      eval->submit_gradient(fe_eval->get_gradient(q), q);
```

- On GPU, quadrature point functor is passed to loop: uses CUDA threads to parallelize over quadrature points (deal.II tutorial step-64):

```
template <int dim, int fe_degree>
__device__ void LaplaceOperatorQuad<dim, fe_degree>::
  operator()(CUDAWrappers::FEEvaluation<dim, fe_degree> *eval,
             const unsigned int                          q) const
{
  eval->submit_gradient(eval->get_gradient(q), q);
}
```

# Outline

CFD application requirements

Mathematical requirements

Software design

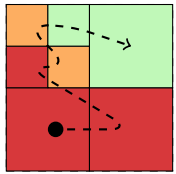
Fast matrix-free operator evaluation

Geometric multigrid

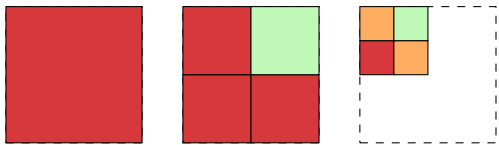
Iterative solver performance

# A scalable multigrid framework

- Implicit solution of large linear systems
- Poisson-type equations: multigrid is optimal
- Applies simple scheme on hierarchy of coarser problems
- Level smoothers crucial ingredient
- Select methods where fast operator evaluation (matrix-free) is core component
  - Point-diagonal or block-diagonal with tensor product inversion
  - Chebyshev iteration around these methods
- In deal.II: flexible geometric multigrid infrastructure like *h*-MG, *p*-MG, adaptive meshes



Sample view of active mesh on 3 processors



level $\ell = 0$ level $\ell = 1$ level $\ell = 2$

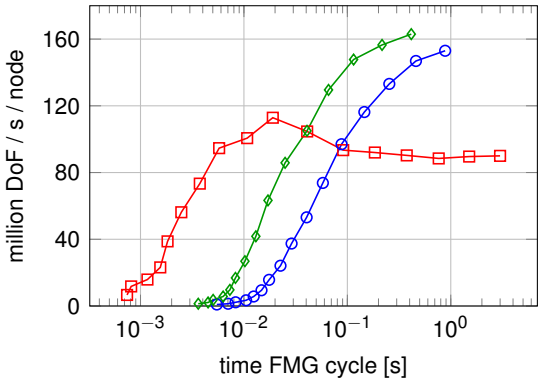

Clevenger, Heister, Kanschat, Kronbichler, A Flexible, Parallel, Adaptive Geometric Multigrid method for FEM, *arXiv*:1904.03317, 2019

Fehn, Munch, Wall, Kronbichler, Hybrid multigrid methods for high-order discontinuous Galerkin discretizations, *arXiv*:1910.01900, 2019

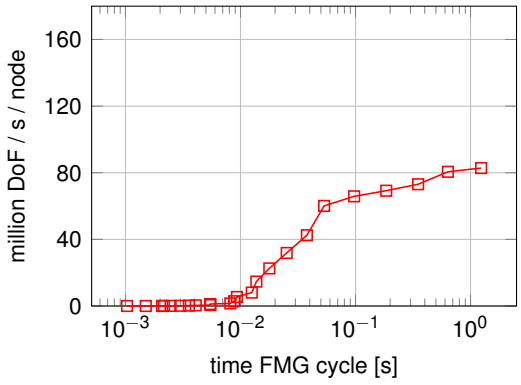
# Multigrid node-level comparison: CPU vs Xeon Phi vs GPU

GMG with full multigrid cycle, Chebyshev (5,6) smoother, continuous $\mathcal{Q}_4$ elements

Single-node




256 nodes, 7168 cores

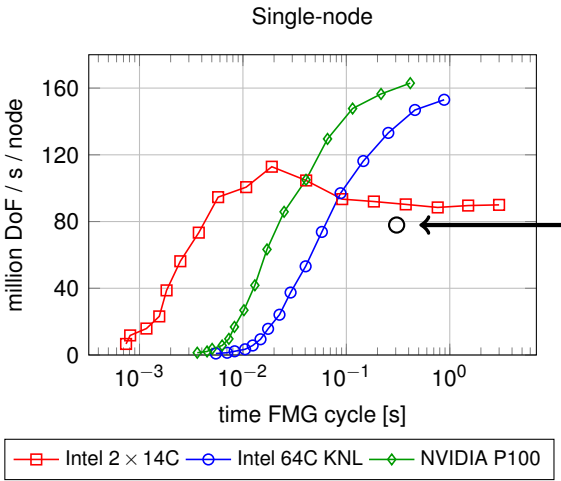



2 × 14 core Intel Xeon E5-2697 v3 (Haswell)


Kronbichler, Wall, A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, *SISC* 40:A3423–48, 2018
Kronbichler, Ljungkvist, Multigrid for matrix-free high-order finite element computations on graphics processors, *ACM TOPC*, 6(1):2/1–2/29, 2019

# Multigrid node-level comparison: CPU vs Xeon Phi vs GPU

GMG with full multigrid cycle, Chebyshev (5,6) smoother, continuous $\mathcal{Q}_4$ elements



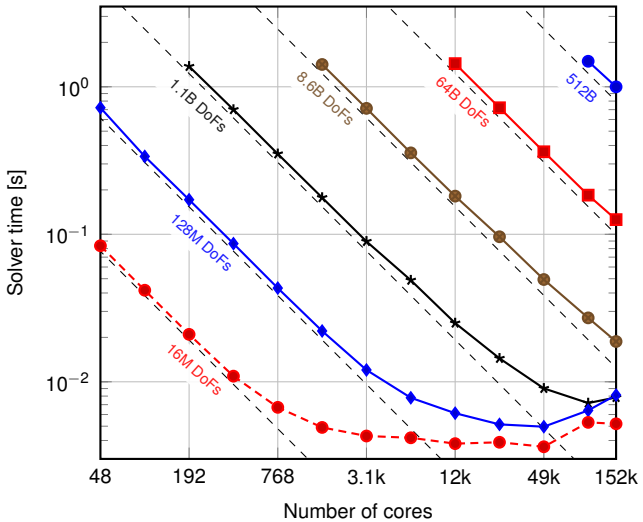

One matrix-vector product with SpMV for statically condensed finite elements with $\mathcal{Q}_4$ elements, 17m DoF, 79 million DoF/s on Broadwell

Matrix-free **solves a linear system faster than one SpMV**!


Kronbichler, Wall, A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, *SISC* 40:A3423–48, 2018
Kronbichler, Ljungkvist, Multigrid for matrix-free high-order finite element computations on graphics processors, *ACM TOPC*, 6(1):2/1–2/29, 2019

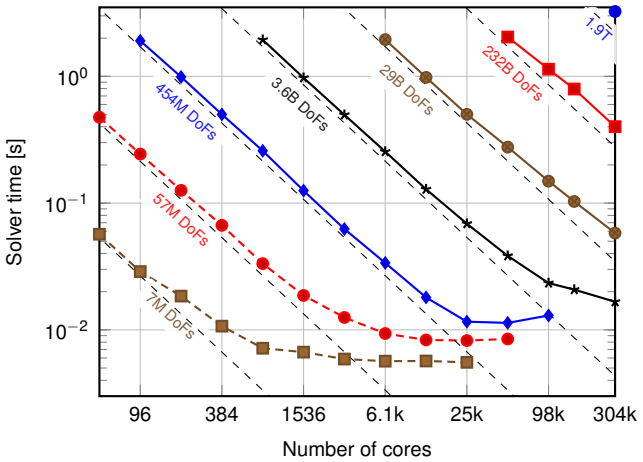
# Scalability of multigrid: continuous elements

- Poisson equation in 3D
- Quartic $\mathcal{Q}_4$ continuous FEM
- Conjugate gradient solver preconditioned by *h* multigrid
  - Chebyshev (5,5) smoother
  - Only mat-vec and matrix diagonal
  - Multigrid V-cycle in single precision (1.8× performance)
- 2 CG iterations (tolerance $10^{-3}$)
- Intel Xeon Platinum 8174, 48 cores per node, up to 3168 nodes (half SuperMUC-NG)



# Scalability of multigrid: discontinuous elements

- $\mathcal{Q}_5$ discontinuous elements, Hermite-like basis
- Conjugate gradient solver preconditioned by *ch* multigrid
  - DG: Chebyshev (6,6) iteration around block-Jacobi (fast diagonalization)
  - FEM: Chebyshev (6,6) around point Jacobi
  - Multigrid V-cycle in single precision
- 2 CG iterations (tolerance $10^{-3}$)
- Intel Xeon Platinum 8174, 48 cores per node, up to 6336 nodes (full SuperMUC-NG)




Arithmetic performance 1.9 trillion DoFs:
**5.2 PFlop/s** (4.9 PFlop/s in SP, 0.27 PFlop/s in DP)
180 GB/s per node (STREAM: 210 GB/s)

# Outline

CFD application requirements

Mathematical requirements

Software design

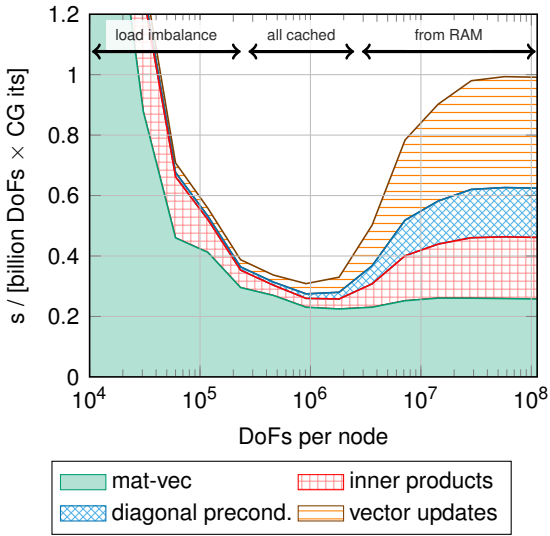
Fast matrix-free operator evaluation

Geometric multigrid

Iterative solver performance

# Going beyond the matrix-vector product: CEED benchmarks

- Benchmark problem BP5 of US exascale discretization initiative CEED
- 3D Poisson, deformed geometry, continuous elements
- conjugate gradient + diagonal preconditioner, GLL quadrature
- 1 node of dual-socket Intel Skylake
- Matrix-vector product might no longer be dominant
- Vector operations take significant time when operated from RAM

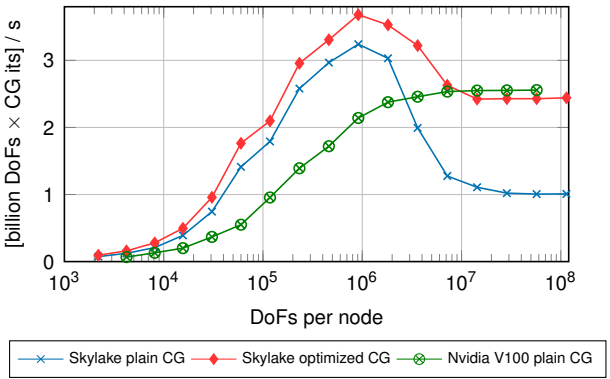



https://ceed.exascaleproject.org/bps/

Fischer, Min, Rathnayake, Dutta, Kolev, Dobrev, Camier, Kronbichler, Warburton, Świrydowicz, Brown, Running faster in HPC applications, *submitted to IJHPCA*, 2019

# Solution: Merge vector operations with mat-vec

- Solution: Data locality
- Best approach: Run all vector operations of CG before/after touching DoFs in mat-vec
- Overlap arithmetic intensive phase of mat-vec with memory-intensive vector operation




Achieved throughput 34M DoF:

- Volta: 2560 MDoFs/s
- Intel Skylake $2 \times 24$C baseline: 1010 MDoFs/s
- Skylake optimized: 2420 MDoF/s

**Arithmetic:**

- Volta: 586 GFlop/s
- Skylake optimized: 702 GFlop/s

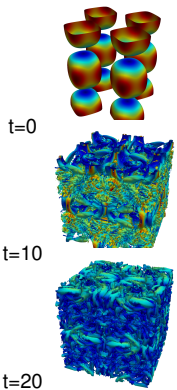
**Memory:**

- Volta: 699 GB/s
- Skylake optimized: 191 GB/s

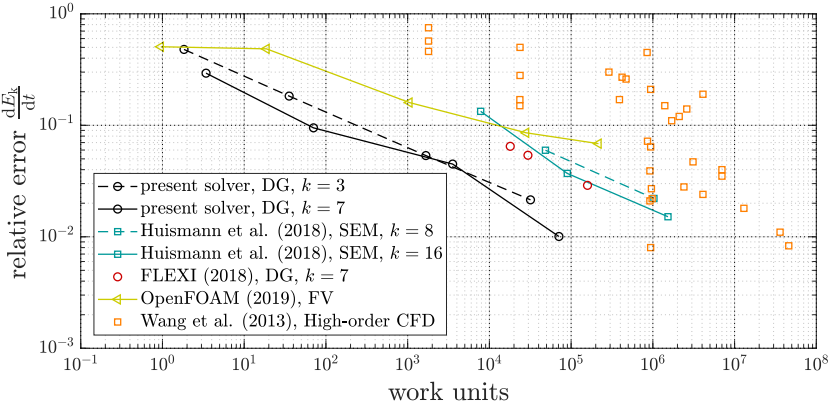
Note: Skylake (opt) and Volta run vastly different implementations! CPU code considerably more complex!

# Evaluation of primary design target – CFD performance

3D 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 in normalized run time** than all results from Wang et al. (2013)





- Fehn, Wall, Kronbichler, Efficiency of high-performance discontinuous Galerkin spectral element methods for under-resolved turbulent incompressible flows, *Int J Numer Meth Fluids 88:32–54*, 2018 + recent updates
- Wang et al., High-order CFD methods: current status and perspective, *Int J Numer Meth Fluids 72:811–845*, 2013
- www.flexi-project.org
- Huismann, Stiller, Fröhlich, Scaling to the stars – a linearly scaling elliptic solver for *p*-multigrid, *J Comput Phys*, 2019

# Summary

- Efficient CFD solver: incompressible, compressible, multiphysics
- Target modern and evolving exascale hardware efficiently
  - Parallel programming models (SIMD, CUDA, ...)
  - Mixed precision: DP/SP in use, going to bfloat16?
- Programming models with CPUs/GPUs
  - In `deal.II` we have different back-ends but similarly looking application code
  - CPUs and GPUs are within a factor of 2 (at same power) for our algorithms
    - GPUs $3-4\times$ more memory bandwidth
    - CPUs allow for clever caching strategies/more merging
- Memory bandwidth bottlenecks $\rightarrow$ compute more, store (even) less
- Matrix-vector product no longer dominant
- Separation of concerns to combine HPC with large set of equations & powerful math
  - Finite element infrastructure from deal.II library
  - High-performance matrix-free components co-developed with deal.II library (primary developer)
  - Mesh infrastructure from p4est library, METIS, etc.
  - Algebraic multigrid via Trilinos (ML/MueLu)