Fast matrix-free evaluation of discontinuous Galerkin finite element operators*

Martin Kronbichler[†] Katharina Kormann[‡]

October 2, 2019

Abstract

We present an algorithmic framework for matrix-free evaluation of discontinuous Galerkin finite element operators. It relies on fast quadrature with sum factorization on quadrilateral and hexahedral meshes, targeting general weak forms of linear and nonlinear partial differential equations. Different algorithms and data structures are compared in an in-depth performance analysis. The implementations of the local integrals are optimized by vectorization over several cells and faces and an even-odd decomposition of the one-dimensional interpolations. Up to 60% of the arithmetic peak on Intel Haswell, Broadwell, and Knights Landing processors are reached when running from caches and up to 40% of peak when also considering the access to vectors from main memory. On 2×14 Broadwell cores, the throughput is up to 2.2 billion unknowns per second for the 3D Laplacian and up to 4 billion unknowns per second for the 3D advection on affine geometries, close to a simple copy operation at 4.7 billion unknowns per second. Our experiments show that MPI ghost exchange has a considerable impact on performance and we present strategies to mitigate this effect. Finally, various options for evaluating geometry terms and their performance are discussed. Our implementations are publicly available through the deal.II finite element library.

Key words. Matrix free method, Finite element method, Discontinuous Galerkin method, Sum factorization, Vectorization, Parallelization.

1 Introduction

The discontinuous Galerkin (DG) method has gained a lot of momentum in a wide range of applications in the last two decades. The method combines the favorable features of the numerical fluxes in finite volume methods, also called Riemann solvers, with the high-order capabilities of polynomial spaces in finite elements. This construction allows for both high convergence rates on complicated computational domains as well as robustness in transport-dominated problems. DG methods are promising ingredients for next-generation solvers in fluid dynamics and wave propagation problems (see e.g. [60] and references therein) and are also applied to a large number of other problems with mixed first and second order derivatives.

There is a large body of literature on implementing DG schemes and performance tuning for particular equations, especially for GPUs, see e.g. [31, 48, 1] and references therein. Implementations for explicit time integration and various optimizations for triangles and tetrahedra have

*This work was supported by the German Research Foundation (DFG) under the project "High-order discontinuous Galerkin for the exa-scale" (ExaDG) within the priority program "Software for Exascale Computing" (SPPEXA), grant agreement no. KO5206/1-1 and KR4661/2-1. The authors acknowledge the support given by the Bayerische Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR) in the framework of the project *Matrix-free GPU kernels for complex applications in fluid dynamics*. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (LRZ, `www.lrz.de`) through project id pr83te.

[†]Institute for Computational Mechanics, Technical University of Munich, Boltzmannstr. 15, 85748 Garching b. München, Germany (`kronbichler@lnm.mw.tum.de`).

[‡]Max Planck Institute for Plasma Physics, Boltzmannstr. 2, 85748 Garching, Germany, and Zentrum Mathematik, Technical University of Munich, Boltzmannstr. 3, 85748 Garching, Germany (`katharina.kormann@ipp.mpg.de`).

reached a high level of maturity [21]. The present work targets DG methods on quadrilaterals and hexahedra with moderate polynomial degrees between 2 and 10 in the context of general meshes and possibly variable coefficients. In this case, the final stencil cannot be separated into tensor products of 1D matrices (as opposed to the constant coefficient Cartesian mesh case [25]) and the fastest implementation option is usually the evaluation of integrals on the fly by fast sum-factorization techniques [12, 28, 35] that have their origin in spectral elements [50, 51]. Sum factorization gives rise to an operator evaluation cost of $\mathcal{O}(dk)$ per degree of freedom in d spatial dimensions for k polynomials per direction, i.e., polynomial degree $p = k - 1$, in contrast to costs of $\mathcal{O}(k^d)$ in (non-separable) matrix-based variants. While originally used for higher degrees, recent work [9, 41, 44] has shown that these techniques are up to an order of magnitude faster than sparse matrix-vector products already for medium polynomial degrees of $p = 3$ or $p = 4$, with increasing gaps at higher orders.

Tensor product evaluation is a very active research area with implementations available in the generic finite element software packages deal.II [3], DUNE [8, 7], Firedrake [52, 47, 45], Loopy [30], mfem [33], Nek5000 [16], Nektar++ [10], NGSolve [55], or via OCCA [54] as well as in application codes such as the compressible flow solver framework Flexi [22], SPECFEM3D [34] or pTatin3D [46]. Despite the wide availability of software, including code generators and domain-specific languages in Firedrake and Loopy, the analysis of high performance computing aspects and the expected performance envelopes of operator evaluation—independent of the user interfaces—have not yet been established. This work fills this gap by an extensive analysis of the evaluation of high-order finite element and discontinuous Galerkin operators with focus on the choice of data layouts and loop structures with their respective impact on performance for architectures with a deep memory hierarchy, supported by careful numerical experiments and results from hardware performance counters that quantify the effect of optimizations. Arithmetic intensities are in the range of 1 to 10 Flop/Byte, which is similar to the machine balance of modern hardware. Thus, both arithmetic optimizations and memory access optimizations are essential. We propose to separately analyze these two effects in order to select the best possible algorithm and to gain understanding into the behavior of the proposed algorithms.

The polynomial degree p is mostly treated as a parameter throughout this work and analyzed over a wide range $1 \leq p \leq 25$, even though there are vastly different requirements on meshing and data structures at the two ends. Since intermediate polynomial degrees $3 \leq p \leq 8$ are particularly interesting in many challenging engineering applications [12, 14], the results for degrees $p > 15$ are mainly meant to show the limits of the present techniques, and do not exploit all possible optimizations. The proposed algorithms aim for reaching high single-node performance, but they also directly apply to the massively parallel context of petascale machines [6, 38, 44], given that the communication in operator evaluation is only between nearest neighbors and naturally scales to large processor counts. Our analysis is based on algorithms that are available in the open-source finite element library deal.II [3] and accessible to generic applications within a toolbox providing other advanced features such as multigrid solvers and mesh adaptivity with hanging nodes. Applications of these algorithms in fluid dynamics can be found in [38, 40, 14] and for wave propagation in [36, 43, 56]. The developments in this work interoperate with the continuous finite element implementations from [41] using the same optimized code paths in the relevant algorithms. Operator evaluation is traditionally the dominating algorithmic part both for explicit time integration and for some iterative solvers such as multigrid with selected smoothers [11, 44].

This article is structured as follows. Sec. 2 introduces the test problems, the DG discretization, and the matrix-free implementation based on fast integration. Sec. 3 concentrates on the compute part of the algorithm. The access to the source and destination vectors for cell and face integrals are analyzed in Sec. 4 alongside with the question of efficient ghost data exchange with MPI. Sec. 5 discusses the options for how to apply the geometry and Sec. 6 concludes this work.

2 DG algorithm

We assume a decomposition of the computational domain Ω into a set of quadrilateral or hexahedral elements $\mathcal{T}_h = \{\Omega_e\}$. The faces $\mathcal{F}_h$ are the set of all intersections $\bar{\Omega}_{e^-} \cap \bar{\Omega}_{e^+}$, i.e., the edges of the cells in 2D and the surfaces of the cells in 3D, with the subset $\mathcal{F}_h^{\mathrm{i}}$ denoting the interior faces between two cells Ω_{e^-} and Ω_{e^+} with solution u_h^- and u_h^+, respectively, and the set of boundary

faces $\mathcal{F}_h^{\mathrm{b}}$ where only the solution field u_h^- is present. The vectors $\mathbf{n}^- = -\mathbf{n}^+$ denote the outer unit normal vectors on either side of a face. We also write $\mathbf{n}$ instead of $\mathbf{n}^-$ for the normal vector associated to the cell Ω_e under consideration.

The quantity $\{\{u_h\}\} = \frac{1}{2}(u_h^- + u_h^+)$ denotes the average of the values on the two sides of a face and the jump is written as $[\![u_h]\!] = u_h^-\mathbf{n}^- + u_h^+\mathbf{n}^+ = (u_h^- - u_h^+)\mathbf{n}^-$. At domain boundaries, suitable definitions for the exterior solution u_h^+ in terms of the boundary conditions and the inner solution u_h^- are used, e.g. the mirror principle $u_h^+ = -u_h^- + 2g$ in case of Dirichlet conditions [21]. Inhomogeneous Dirichlet data add contributions to the right hand side vectors in linear systems.

To exemplify the algorithms, we consider two prototype discontinuous Galerkin discretizations to stationary problems: The DG discretization of the stationary advection equation with local Lax–Friedrichs (upwind) flux [21], characteristic for first-order hyperbolic PDEs, reads

$$-(\nabla v_h, \mathbf{c}u_h)_{\Omega_e} + \left\langle v_h\mathbf{n}, \{\{\mathbf{c}u_h\}\} + \frac{|\mathbf{c}\cdot\mathbf{n}|}{2}[\![u_h]\!]\right\rangle_{\partial\Omega_e} = (v_h, f)_{\Omega_e}, \tag{1}$$

where $\mathbf{c} = \mathbf{c}(\mathbf{x})$ denotes the direction of transport and f is some forcing. The bilinear form $(a,b)_{\Omega_e} = \int_{\Omega_e} ab\,\mathrm{d}\mathbf{x}$ denotes volume integrals and $\langle a,b\rangle_{\partial\Omega_e} = \int_{\partial\Omega_e} ab\,\mathrm{d}\mathbf{s}$ boundary integrals.

The symmetric interior penalty discretization of the Laplacian [4] is

$$(\nabla v_h, \nabla u_h)_{\Omega_e} - \langle v_h\mathbf{n}, \{\{\nabla u_h\}\}\rangle_{\partial\Omega_e} - \left\langle \nabla v_h, \frac{1}{2}[\![u_h]\!]\right\rangle_{\partial\Omega_e} + \langle v_h\mathbf{n}, \tau[\![u_h]\!]\rangle_{\partial\Omega_e} = (v_h, f)_{\Omega_e}, \tag{2}$$

on element Ω_e, consisting of the cell integral, the primal consistency term, the adjoint consistency term, and an interior penalty term with factor τ sufficiently large to render the discretization coercive.

On each element Ω_e, we assume the solution to be given by an expansion $u_h^{(e)} = \sum_{i=1}^N \varphi_i^{(e)}(\mathbf{x})u_i^{(e)}$, where $u_i^{(e)}$ are the coefficient values determined through the variational principle and $\varphi_i^{(e)}(\mathbf{x})$ are basis functions. The basis functions are defined as polynomials $\varphi_i(\boldsymbol{\xi})$ on the reference element Ω_{unit} with coordinates $\boldsymbol{\xi}$ and transformed to the coordinates $\mathbf{x}$ by a transformation $\hat{\mathbf{x}}^{(e)}$ as $\mathbf{x} = \hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi})$, i.e., $\varphi_i^{(e)}(\mathbf{x}) = \varphi_i\left(\boldsymbol{\xi}^{(e)}(\mathbf{x})\right)$. The Jacobian of the transformation is denoted as $\mathcal{J}_{(e)}(\boldsymbol{\xi}) = \frac{\mathrm{d}\hat{\mathbf{x}}^{(e)}}{\mathrm{d}\boldsymbol{\xi}} = \nabla_{\boldsymbol{\xi}}\hat{\mathbf{x}}^{(e)}$ with partial derivatives arranged in columns and coordinates in rows, which allows to express the derivative as $\nabla_{\mathbf{x}}\varphi_i(\boldsymbol{\xi}^{(e)}(\mathbf{x})) = \mathcal{J}_{(e)}^{-\mathrm{T}}\nabla_{\boldsymbol{\xi}}\varphi_i(\boldsymbol{\xi})$. In this work, we assume quadrilateral elements in two dimensions or hexahedral elements in three dimensions with reference-cell shape functions defined through the tensor product of one-dimensional functions $\varphi_i(\xi_1,\xi_2,\xi_3) = \varphi_{i_1}^{\mathrm{1D}}(\xi_1)\varphi_{i_2}^{\mathrm{1D}}(\xi_2)\varphi_{i_3}^{\mathrm{1D}}(\xi_3)$ with the respective multi-index (i_1,i_2,i_3) associated to the index i. The integrals in equations (1) and (2) are computed numerically in the reference space by summation of the integrands evaluated on a set of quadrature points $\boldsymbol{\xi}_q = (\xi_{q_1},\xi_{q_2},\xi_{q_3})$ with associated weights w_q defined as the tensor product of 1D quadrature formulas. For example, the cell term for advection in (1) is approximated by

$$\begin{aligned}(\nabla\varphi_i, \mathbf{c}u_h)_{\Omega_e} &= \int_{\Omega_{\text{unit}}} \left(\mathcal{J}_{(e)}(\boldsymbol{\xi})^{-\mathrm{T}}\nabla_{\boldsymbol{\xi}}\varphi(\boldsymbol{\xi})\right)\cdot\left(\mathbf{c}\left(\hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi})\right)u_h^{(e)}(\boldsymbol{\xi})\right)\det(\mathcal{J}_{(e)}(\boldsymbol{\xi}))\,\mathrm{d}\boldsymbol{\xi}\\ &\approx \sum_{q=1}^{n_q}\left(\mathcal{J}_{(e)}(\boldsymbol{\xi}_q)^{-\mathrm{T}}\nabla\varphi_i(\boldsymbol{\xi}_q)\right)\cdot\left(\mathbf{c}\left(\hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi}_q)\right)u_h^{(e)}(\boldsymbol{\xi}_q)\right)\det(\mathcal{J}_{(e)}(\boldsymbol{\xi}_q))w_q.\end{aligned} \tag{3}$$

Since the operators on the left hand sides of equations (1) and (2) are linear, integrating an equation against all test functions $v_h = \varphi_i$ corresponds to a matrix-vector product taking a vector of coefficient values $\mathbf{u} = [u_i]$ associated with the solution field u_h and returning the integrals $\mathbf{y} = [y_i]$,

$$\mathbf{y} = A\mathbf{u}. \tag{4}$$

Since our methods are based on numerical integration rather than the final cell matrices, the techniques extend straight-forwardly to the residual evaluation of nonlinear equations.

2.1 Two implementation options for face integrals

The formulas (1) and (2), respectively, define the face integrals directly associated with the cell integrals of Ω_e. The operator evaluation implemented with this form of face integrals and a single

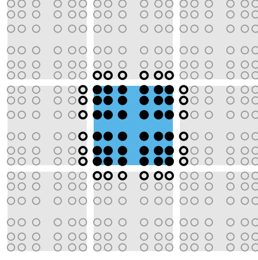

Figure 1: Data access pattern for element-wise face integrals of advection on a nodal basis with $p = 5$. Black disks indicate read/write access by the marked cell, black circles read only. Gray circles are not touched.

outer sum over all elements Ω_e is labeled "element-wise face integrals" in the following. In this variant, the same contribution to the numerical flux, e.g. $\{\{\mathbf{c}u_h\}\}+\frac{|\mathbf{c}\cdot\mathbf{n}|}{2}[\![u_h]\!]$ in the case of advection, is computed twice with different interpretations of u_h^- and u_h^+ and $\mathbf{n}$ on Ω_{e^-} and Ω_{e^+}, respectively. An alternative is to rearrange the loops to collect all contributions to an individual face $F \in \mathcal{F}_h^{\mathrm{i}}$. For advection, the left-hand side integrals can be equivalently stated as

$$\sum_{\Omega_e\in\mathcal{T}_h} (\nabla v_h, -\mathbf{c}u_h)_{\Omega_e} + \sum_{F\in\mathcal{F}_h^{\mathrm{i}}} \left\langle [\![v_h]\!], \{\{\mathbf{c}u_h\}\} + \frac{|\mathbf{c}\cdot\mathbf{n}|}{2}[\![u_h]\!] \right\rangle_F + \sum_{F\in\mathcal{F}_h^{\mathrm{b}}} \langle v_h\mathbf{n}, \mathbf{c}u_h\rangle_F\,, \tag{5}$$

representing the homogeneous part of a Dirichlet problem with $u_h^+ = -u_h^-$ at the boundary.

The formulation (5), labeled "compact face integrals" in the following, has the advantage of computing the flux term for both sides Ω_{e^-} and Ω_{e^+} simultaneously. This reduces the number of arithmetic operations and data access at quadrature points. Conversely, the element-wise evaluation of face integrals computes all contributions to a cell at once (at the price of keeping results from all $2d$ faces in flight), leading to a more structured access into vectors arranged cell-wise. For example, this allows for a single write of the result originating from Ω_e into the vector $\mathbf{y}$ of the operator evaluation (4). Fig. 1 shows the access pattern in the element-wise face integrals, which resembles finite differences. In the compact form of face integrals, the face's data access is somewhat delayed compared to the access of cell integrals, especially on unstructured meshes, reducing the efficiency of caching. Since it is not a priori clear which formulation to prefer, both are analyzed in this work.

2.2 Algorithm outline for discontinuous Galerkin finite element operator evaluation

The matrix-free evaluation of the integrals representing the product (4) is implemented by a loop over all the cells and faces appearing in the operators (1) or (2). Algorithm 1 describes the procedure for the advection operator with compact face integrals. The evaluation is split into three phases, addressing the cell integrals, integrals for interior faces, and integrals for boundary faces in separate steps, following the sums in (5). Each of these loops logically consists of five components, which are (a) the extraction of the solution values pertaining to the current cell(s) by a gather operation, (b) the evaluation of values or gradients of the local solution at the quadrature points, (c) the operation at quadrature points including the application of the geometry, (d) the multiplication by the test functions and the summation in the numerical quadrature, and finally (e) accumulating the local values into the respective entry of the global vector by a scatter-add operation. Splitting the evaluation (b), operations at quadrature points (c), and integration (d) into separate phases is a common interface that allows for combining generic programming interfaces with optimal-complexity implementations [32]. Optimizations across these three steps is less common due to code complexity, but it is possible by code generation frameworks such as Firedrake [24]. The performance data available for Firedrake as of writing this text [58] is not yet conclusive as to whether this would enable efficiency gains. Since each cell has independent degrees of freedom in

ALGORITHM 1: Evaluation of advection (5) with **compact face integrals** and Dirichlet b.c.

(i) `update_ghost_values`: Import vector values of $\mathbf{u}$ from other MPI processes that are adjacent to locally owned cells and for which computations of the respective face integrals are scheduled on the current MPI rank.

(ii) loop over cells

  (a) gather local vector values $u_i^{(e)}$ on cell from global input vector $\mathbf{u}$

  (b) interpolate local vector values $\mathbf{u}^{(e)}$ in quadrature points, $u_h^{(e)}(\boldsymbol{\xi}_q) = \sum_i \varphi_i u_i^{(e)}$

  (c) for each quadrature index q, prepare integrand at each quadrature point by computing $\mathbf{t}_q = \mathcal{J}_{(e)}(\boldsymbol{\xi}_q)^{-1}\mathbf{c}\left(\hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi}_q)\right) u_h^{(e)}(\boldsymbol{\xi}_q)\det(\mathcal{J}_{(e)}(\boldsymbol{\xi}_q))w_q$

  (d) evaluate local integrals by quadrature $y_i^{(e)} = \left(\nabla\varphi_i, \mathbf{c}u_h^{(e)}\right)_{\Omega_e} \approx \sum_q \nabla\varphi_i(\boldsymbol{\xi}_q)\cdot\mathbf{t}_q$ for all test functions i

  (e) set the local contributions $y_i^{(e)}$ into the global result vector $\mathbf{y}$

(iii) loop over interior faces $\mathcal{F}_h^{\mathrm{i}}$

  (a_-) gather local vector values u_i^- from global input vector $\mathbf{u}$ associated with interior cell e^-

  (b_-) interpolate u^- in face quadrature points $u_h^-(\boldsymbol{\xi}_q)$

  (a_+) gather local vector values u_i^+ from global input vector $\mathbf{u}$ associated with exterior cell e^+

  (b_+) interpolate u^+ in face quadrature points $u_h^+(\boldsymbol{\xi}_q)$

  (c) for each quadrature index q, compute the numerical flux contribution $(\mathbf{f}^*\cdot\mathbf{n}^-)_q = \frac{1}{2}\mathbf{c}\left(\hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi}_q)\right)\cdot\mathbf{n}^-\left(u_h^-(\boldsymbol{\xi}_q)+u_h^+(\boldsymbol{\xi}_q)\right) + \frac{1}{2}\left|\mathbf{c}(\hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi}_q))\cdot\mathbf{n}^-\right|\left(u_h^-(\boldsymbol{\xi}_q)-u_h^+(\boldsymbol{\xi}_q)\right)$ and multiply it by integration weight, $t_q = (\mathbf{f}^*\cdot\mathbf{n}^-)_q h(\boldsymbol{\xi}_q)w_q$ with area element h of face

  (d_-) evaluate local integrals by quadrature $y_i^- = (\varphi_i^-, \mathbf{f}^*\cdot\mathbf{n}^-) \approx \sum_q \varphi_i(\boldsymbol{\xi}_q)t_q$

  (e_-) add local contribution y_i^- into the global result vector $\mathbf{y}$ associated with e^-

  (d_+) evaluate local integrals by quadrature $y_i^+ = (\varphi_i^+, -\mathbf{f}^*\cdot\mathbf{n}^-) \approx \sum_q -\varphi_i(\boldsymbol{\xi}_q)t_q$ due to $\boldsymbol{n}^+ = -\boldsymbol{n}^-$ and the conservativity of the numerical flux function

  (e_+) add local contribution y_i^+ into the global result vector $\mathbf{y}$ associated with e^+

(iv) loop over boundary faces $\mathcal{F}_h^{\mathrm{b}}$

  (a) gather local vector values u_i^- from cell e^- from global input vector $\mathbf{u}$

  (b) interpolate u^- in face quadrature points $u_h^-(\boldsymbol{\xi}_q)$

  (c) for each quadrature index q, compute the numerical flux contribution and multiply by integration factor

  (d) evaluate local integrals y_i^- by quadrature similar to inner faces

  (e) add local contribution y_i^- into the global result vector $\mathbf{y}$ associated with e^-

(v) `compress`: Export parts of the residuals that have been generated on the current MPI process to the owning process.

DG, the result vector needs not be zeroed explicitly as typical in continuous finite elements [41] and the integral values in step (ii)(e) overwrite the previous vector content.

Algorithm 2 highlights the changes in the loop layout compared to Algorithm 1 if the element-wise formulation of face integrals of formula (1) is used instead. The numerical flux is evaluated twice for each interior face and an additional interpolation step (b_+) is added. The other interpolation and integration steps, (b_-) and (d_-), exactly represent the operations done from both sides in Algorithm 1. At the same time, no explicit vector access (a_-) and (e_-) to the elements' vector data is necessary as the local data can be shared with cell integrals. Furthermore, Algorithm 2 only needs one MPI ghost exchange step, involving twice the data, though.

ALGORITHM 2: DG integration loop for the advection operator (1) with **element-wise face integrals** and Dirichlet b.c.

(i) `update_ghost_values`: Import vector values of $\mathbf{u}$ from other MPI processes on all cells that are adjacent to locally owned cells.

(ii) loop over cells

- (a) read local vector values $u_i^{(e)}$
- (b) interpolate local vector values $\mathbf{u}^{(e)}$ in quadrature points of cell, $u_h^{(e)}(\boldsymbol{\xi}_q) = \sum_i \varphi_i u_i^{(e)}$
- (c) for each quadrature index q, prepare integrand at each quadrature point of cell by computing $\mathbf{t}_q = \mathcal{J}_{(e)}(\boldsymbol{\xi}_q)^{-1} \mathbf{c}\left(\hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi}_q)\right) u_h^{(e)}(\boldsymbol{\xi}_q) \det(\mathcal{J}_{(e)}(\boldsymbol{\xi}_q)) w_q$
- (d) evaluate local cell integrals by quadrature $y_i^{(e)} = \left(\nabla\varphi_i, \mathbf{c} u_h^{(e)}\right)_{\Omega_e} \approx \sum_q \nabla\varphi_i(\boldsymbol{\xi}_q) \cdot \mathbf{t}_q$ for all test functions i

(iii) loop over all $2d$ faces of cell Ω_e

- (b_-) interpolate values from cell array $\mathbf{u}^{(e)}$ to quadrature points of face $u_h^-(\boldsymbol{\xi}_q)$
- (a_+) if not on boundary, gather values from neighbor Ω_{e^+} of current face or use $-u^-$ if on boundary
- (b_+) interpolate u^+ in face quadrature points $u_h^+(\boldsymbol{\xi}_q)$
- (c) for each quadrature index q, compute the numerical flux contribution $(\mathbf{f}^* \cdot \mathbf{n}^-)_q = \frac{1}{2}\mathbf{c}\left(\hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi}_q)\right) \cdot \mathbf{n}^- \left(u_h^-(\boldsymbol{\xi}_q) + u_h^+(\boldsymbol{\xi}_q)\right) + \frac{1}{2}\left|\mathbf{c}(\hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi}_q)) \cdot \mathbf{n}^-\right| \left(u_h^-(\boldsymbol{\xi}_q) - u_h^+(\boldsymbol{\xi}_q)\right)$ and multiply it by integration weight, $t_q = (\mathbf{f}^* \cdot \mathbf{n}^-)_q h(\boldsymbol{\xi}_q) w_q$ with area element h of face
- (d_-) evaluate local face integrals by quadrature and add into cell contribution, $y_i^{(e)} = y_i^{(e)} + \sum_q \varphi_i(\boldsymbol{\xi}_q) t_q$

- (e) set all contributions of cell, $\mathbf{y}^{(e)}$, into global result vector $\mathbf{y}$

2.3 Sum factorization

In Algorithm 1, the steps (ii–iv)(b) and (ii–iv)(d) interpolating the solution from the coefficient values to quadrature points and the summation for quadrature are the crucial components for higher polynomial degrees because all vector entries inside a cell can contribute to the values at each quadrature point. We specialize the evaluation for tensor product shape functions on tensor product quadrature formulas.

Let us denote by S_i the $k \times k$ matrix of values of all k one-dimensional shape functions $\varphi^{1\mathrm{D}}$ of degree $p = k-1$ evaluated at k quadrature points and by D_i the matrix of their derivatives along direction i, respectively. Quadrature points are arranged in rows and shape functions in columns. Note that all algorithms equally apply to the case where the number of quadrature points is larger than k at similar arithmetic performance, see e.g. [15]. Further, denote by $\mathbf{u}^{(e)}$ the DG coefficients of the input vector. The interpolation step (ii)(b) of Algorithms 1 and 2 has the Kronecker product form

$$\left[S_d \otimes \ldots \otimes S_2 \otimes S_1\right] \mathbf{u}^{(e)}. \tag{6}$$

In order to avoid the naive $2k^{2d}$ arithmetic cost (additions and multiplications counted individually), the Kronecker matrix is not applied in expanded $k^d \times k^d$ form, but rather for each of the d

factors separately in a rearranged way by sum factorization: The multiplication with e.g. the matrix $\left[I_3 \otimes I_2 \otimes S_1\right]$ in three dimensions can be implemented by the multiplication of the $k \times k$ matrix S_1 with the $k \times k^2$ matrix obtained from reshaping $\mathbf{u}^{(e)}$ in column-major form. The matrix-matrix multiplication corresponds to a 1D interpolation with matrix S_i along the coordinate direction i for k^{d-1} lines. In total, the interpolation (6) involves d of these matrix-matrix multiplications. We call each matrix multiplication a **sum-factorization sweep** in the rest of this work. With sum factorization, the overall cost for the interpolation is $2dk^{d+1}$ arithmetic operations. Sum-factorization algorithms have been established by the spectral element community [50, 57, 12, 28, 35] and have also been derived for other element shapes than quadrilaterals and hexahedra with suitable bases using truncated tensor products [57].

The integration step (ii)(d) multiplies the d partial derivatives of the test function $\nabla\varphi_i$ with the d components of the integrand contribution, labeled $\mathbf{t}_{:,1}, \ldots, \mathbf{t}_{:,d}$ in Algorithm 1 and sums over quadrature points. This step is given by

$$\mathbf{y}^{(e)} = \begin{bmatrix} S_d \otimes \ldots \otimes S_2 \otimes D_1 \\ S_d \otimes \ldots \otimes D_2 \otimes S_1 \\ \vdots \\ D_d \otimes \ldots \otimes S_1 \otimes S_1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \mathbf{t}_{:,1} \\ \mathbf{t}_{:,2} \\ \vdots \\ \mathbf{t}_{:,d} \end{bmatrix}, \tag{7}$$

and can again be treated by a series of d^2 sum-factorization sweeps in total.

For the case where quadrature points coincide with the node positions of Lagrange polynomials, the interpolation matrix is the $k \times k$ identity matrix, $S_i = I_i$. As a consequence, the number of sum-factorization sweeps in (7) reduces to d evaluations of D_i^{T}. It is a classical optimization in spectral element codes to choose a suitable nodal basis and quadrature pairing, for example Lagrange polynomials on Gauss–Lobatto points with Gauss–Lobatto quadrature on the same points [12, 35]. For a general polynomial basis, the cost of (7) can be reduced by combining such a simplified derivative with a basis change: if we define a 1D gradient matrix $(D_i^{\mathrm{co}})_{q,j}$ as the gradient of Lagrange polynomials $\varphi_j^{\mathrm{1D,co}}(\xi_q)$ with nodes at the quadrature points, i.e., $D_i = D_i^{\mathrm{co}} S_i$, expression (7) can be rewritten as

$$\mathbf{y}^{(e)} = \underbrace{\left[S_d^{\mathrm{T}} \otimes \ldots \otimes S_2^{\mathrm{T}} \otimes S_1^{\mathrm{T}}\right]}_{\text{basis change}} \underbrace{\begin{bmatrix} I_d \otimes \ldots \otimes I_2 \otimes D_1^{\mathrm{co}} \\ I_d \otimes \ldots \otimes D_2^{\mathrm{co}} \otimes I_1 \\ D_d^{\mathrm{co}} \otimes \ldots \otimes I_2 \otimes I_1 \end{bmatrix}^{\mathrm{T}}}_{\text{collocation derivative}} \begin{bmatrix} \mathbf{t}_{1,:} \\ \mathbf{t}_{2,:} \\ \vdots \\ \mathbf{t}_{d,:} \end{bmatrix}. \tag{8}$$

The first multiplication to the right corresponds to the summations of derivative contributions in the basis associated to the quadrature points (e.g. Lagrange polynomials at Gauss quadrature points), whereas the multiplication by the matrix $\left[S_d^{\mathrm{T}} \otimes \ldots \otimes S_2^{\mathrm{T}} \otimes S_1^{\mathrm{T}}\right]$ transforms the integral contributions to the actual basis (e.g. Lagrange polynomials in Gauss–Lobatto points). The approach via the basis change reduces the number of sum-factorization sweeps for the gradient from d^2 in (7) to $2d$. The basis change concept also applies similarly to the case with more integration points than polynomials (sometimes called over-integration). In case the gradient of the solution $\nabla_{\boldsymbol{\xi}} u_h^{(e)}(\boldsymbol{\xi}_q)$ is needed, like for the Laplacian (2), the transpose of (8) is applied, using $2d$ sum-factorization sweeps for a general basis or d sum-factorization sweeps for collocated nodal and integration points.

The interpolation and integration operations for face integrals are of similar form. As an example, let us consider the evaluation of u_h and $\nabla_{\boldsymbol{\xi}} u_h$ at all quadrature points of a face in 3D with normal in ξ_2 direction. The interpolation matrix consists of four blocks, the first block for the values at the quadrature points, the two subsequent blocks for the derivatives in the local coordinate direction of the face (ξ_1 and ξ_3 in this case), and the last block for the derivative in face-normal direction,

$$\begin{bmatrix} \mathbf{u}_h \\ \partial_{\xi_1}\mathbf{u}_h \\ \partial_{\xi_3}\mathbf{u}_h \\ \partial_{\xi_2}\mathbf{u}_h \end{bmatrix} = \underbrace{\begin{bmatrix} \begin{bmatrix} \begin{bmatrix} I_3 \otimes I_1 \\ I_3 \otimes D_1^{\mathrm{co}} \\ D_3^{\mathrm{co}} \otimes I_1 \end{bmatrix} \left[S_3 \otimes S_1\right] \end{bmatrix} & 0 \\ 0 & \left[S_3 \otimes S_1\right] \end{bmatrix}}_{\text{interpolation within face}} \underbrace{\begin{bmatrix} I_3 \otimes S_{\mathrm{f}} \otimes I_1 \\ I_3 \otimes D_{\mathrm{f}} \otimes I_1 \end{bmatrix}}_{\text{face-normal interpolation}} \mathbf{u}^{(e)}, \tag{9}$$

Table 1: Number of sum-factorization sweeps in d dimensions for the advection operator with a Lagrange basis on Gauss–Lobatto points and the Laplacian on a Hermite-type basis using Gaussian quadrature on k^d points, using compact face integrals according to Algorithm 1. The basis change and derivative columns specify how the total number of sum-factorization sweeps is derived.

	total no. of sweeps	basis change	derivative	face normal
advection, cell	$3d$ on k^d data	$2d$	d	—
advection, inner face	$4d$ on k^{d-1} data	$4(d-1)$	—	4 on k^{d-1} data
advection, boundary face	$2d$ on k^{d-1} data	$2(d-1)$	—	2 on k^{d-1} data
Laplacian, cell	$4d$ on k^d data	$2d$	$2d$	—
Laplacian, inner face	$12(d-1)$ on k^{d-1} data	$8(d-1)$	$4(d-1)$	4 on $2k^{d-1}$ data
Laplacian, boundary face	$6(d-1)$ on k^{d-1} data	$4(d-1)$	$2(d-1)$	2 on $2k^{d-1}$ data

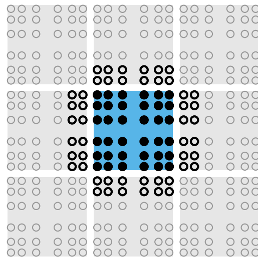

Figure 2: Data access pattern for element-wise face integrals for the Laplacian using a Hermite-like basis with $p = 5$. Black disks indicate read/write access by the marked cell, black circles read only. Gray circles are not touched.

where the $1 \times k$ matrices S_{f} and D_{f} evaluate the shape functions and their first derivative on the respective boundary of the 1D reference cell. For derivatives in ξ_1 or ξ_3 directions, the interpolation matrices are moved to the respective slots in the face-normal interpolation.

The interleaved cell and face evaluation for Algorithm 2 allows to re-use the basis change operation for an alternative evaluation strategy

$$\begin{bmatrix} \mathbf{u}_h \\ \partial_{\xi_1}\mathbf{u}_h \\ \partial_{\xi_3}\mathbf{u}_h \\ \partial_{\xi_2}\mathbf{u}_h \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} I_3 \otimes I_1 \\ I_3 \otimes D_1^{\mathrm{co}} \\ D_3^{\mathrm{co}} \otimes I_1 \end{bmatrix} & 0 \\ 0 & \begin{bmatrix} I_3 \otimes I_1 \end{bmatrix} \end{bmatrix} \begin{bmatrix} I_3 \otimes S_{\mathrm{f}}^{\mathrm{co}} \otimes I_1 \\ I_3 \otimes D_{\mathrm{f}}^{\mathrm{co}} \otimes I_1 \end{bmatrix} \underbrace{\left[S_3^{\mathrm{T}} \otimes S_2^{\mathrm{T}} \otimes S_1^{\mathrm{T}}\right] \mathbf{u}^{(e)}}_{\text{shared with cell integrals}}. \tag{10}$$

The face-normal interpolation matrices $S_{\mathrm{f}}^{\mathrm{co}}$ and $D_{\mathrm{f}}^{\mathrm{co}}$ in (10) refer to the Lagrange polynomials in quadrature points. Formula (10) is used for processing u_h^- as well as the test functions local to the element, whereas (9) is preferred for the exterior solution u_h^+ because a full basis change on the exterior would be more expensive than (9). The operation counts in Sec. 3.5 below show that working with the changed basis for interior face terms brings the cost of the element-wise face integrals close to the cost of the compact face integrals. For the latter, such re-arrangement is infeasible due to the unstructured access.

Table 1 specifies the number of sum-factorization sweeps for the interpolation steps (ii–iv)(b) and the integration steps (ii–iv)(d) when calling Algorithm 1 for the advection operator on the left hand side of (1) and the respective algorithm for the evaluation of the left hand side of the Laplacian (2), in terms of the number of shape functions and points involved in the respective sweeps (labeled "data"). A nodal basis on Gauss–Lobatto points with data access according to Fig. 1 and a Hermite-type basis[1] with data access according to Fig. 2, respectively, are chosen.

As compared to advection, face integrals for the Laplacian involve considerably more interpolation steps for extracting both the solution values and gradients according to (9). Furthermore,

[1] We define a Hermite-type basis as a basis where at most two out of the k one-dimensional shape functions have non-zero value and first derivative on a face as for the cubic Hermite polynomials. Hermite-type basis functions of higher degree are constructed by adding suitable "bubble" functions. As a result, only $2k^{d-1}$ out of the k^d solution values need to be accessed for evaluating u_h and ∇u_h on the faces, see Fig. 2. Note that this basis is non-nodal, despite the intuitive visualization in Fig. 2. The operator evaluation with the collocation basis executes more slowly on an actual implementation despite fewer sum-factorization sweeps because the face-normal interpolation involves all k^d points of a neighbor cell rather than only the $2k^{d-1}$ with non-zero value and first derivative on the face.

twice the number of vector entries in face-normal direction are involved, compare also Fig. 1 with Fig. 2. Other differential operators, like the scalar advection-diffusion-reaction equation, can be treated with the same number of sum-factorization sweeps and only some additional operations at quadrature points. Likewise, for systems of equations the cost in Table 1 is multiplied by the number of components.

Note that this work does not consider specific optimizations for the case where the final cell matrix (steps (b)–(d) in Algorithm 1) can be represented as a sum of Kronecker products, such as the Laplacian on a Cartesian geometry $L = L_2 \otimes M_1 + M_2 \otimes L_1$ in 2D. This involves fewer sum-factorization sweeps than numerical integration and can sometimes be further reduced in complexity [25]. These separable matrices only appear for the case of constant coefficients and axis-aligned meshes and are not the primary interest of this work, even though the presented computational kernels can also be applied to that setting.

2.4 Overview of algorithm design

In the literature, Algorithms 1 or 2 have often been implemented by specializations for the particular equations at hand or selected parallelization schemes. This work attempts to systematically identify patterns ensuring high performance for generic C++ implementations that are not tied to a particular equation and to define a set of algorithmic tests to analyze performance on modern hardware.

The vector access steps (ii–iv)(a) and (ii–iv)(e) in Algorithm 1 merely rearrange data or can even be merged with the other operations. The sum-factorization sweeps (ii–iv)(b) and (ii-iv)(d) of Algorithm 1 operate on some arrays of size k^d that are combined with the coefficients of the polynomial evaluation S_i and D_i in a matrix-vector product fashion with different strides and allow for caching. The operations performed at quadrature points, step (ii–iv)(c) in the algorithm, have most variability and depend on the differential operator, the geometry, and the user code. In order to assess the effects of code optimizations separately, the following three sections discuss these aspects in sequence, starting from the in-cache case of sum factorization with a simple geometry representation over the vector data access to the geometry representation.

3 Compute optimizations

In this section, we ignore the memory access and analyze the implementation of the compute phase with focus on the sum-factorization sweeps for cell integrals at a complexity of $\mathcal{O}(k)$ per degree of freedom. All experiments in this section use a microbenchmark that runs an outer loop around the cell integrals (ii)(b)–(ii)(d) in Algorithm 1 (and face integrals in Sec. 3.5) for the advection (1) and the Laplacian (2) with data $\mathbf{u}^{(e)}$ of the same cell. In order to represent realistic data dependencies, we include the loop over quadrature points (ii)(c) that applies the geometry and quadrature weight w_q with the same constant factor $\det(\mathcal{J}_{(e)})\mathcal{J}_{(e)}^{-1}\mathcal{J}_{(e)}^{-\mathrm{T}}$ at all quadrature points (representing an affine cell geometry). Apart from possibly different geometric tensors, the arithmetic operations performed in the next two sections are complete in the sense that they allow integration on an arbitrarily deformed mesh with precomputed coefficients at quadrature points. In terms of memory access, this represents a best-case scenario as explained in Sec. 5 below.

The outer loop length is set to $n_{\text{cells}} = \lceil 50000/k^4 \rceil$, which gives a run time of around 10^{-4} seconds. When using vectorization through single-instruction/multiple-data (SIMD) units, each lane of a SIMD vector runs identical operations but with different data, except for the specific vectorization tests in Section 3.3. The time to complete this computation is measured with the C++ high-resolution timer `std::chrono::system_clock`. To gather accurate statistics [23], the test is repeated 50,500 times and numbers are reported based on the arithmetic average t_{avg} of the last 50,000 repetitions, excluding cold caches and possible frequency changes of the processor during the first 500 repetitions. For all experiments reported in this section, the standard deviation over the 50,000 samples is below 5% of the measured time, and the minimal timings over all samples are within 2% of the mean. All cores perform the same work on separate data arrays by running a parallel for loop with OpenMP and threads pinned to the cores. From the recorded timings, a throughput number in terms of degrees of freedom processed per second is calculated as the size

Table 2: Specification of hardware systems used for evaluation. Memory bandwidth according to the STREAM triad benchmark (optimized variant without read for ownership transfer involving two reads and one write) and GFlop/s based on the theoretical maximum at the respective AVX frequency.

	Haswell Xeon E5-2630 v3	Broadwell Xeon E5-2690 v4	Knights Landing Xeon Phi 7210
cores	2×8	2×14	64
frequency base	2.4 GHz	2.6 GHz	1.3 GHz
max AVX frequency	2.6 GHz	2.9 GHz	1.1 GHz
SIMD width	256 bit	256 bit	512 bit
arithmetic peak @ AVX freq.	666 GFlop/s	1299 GFlop/s	2253 GFlop/s
last level cache	2.5 MB/core L3	2.5 MB/core L3	512 kB/core L2
memory interface	DDR4-1866, 8 chan.	DDR4-2400, 8 chan.	MCDRAM
STREAM memory bandwidth	95 GB/s	112 GB/s	450 GB/s
Linux operating system release	CentOS 7.1	CentOS 7.1	SLES 11 SP4
compiler	`g++`, version 6.3.0, flags `-O3 -funroll-loops -march=native`		

per core divided by the time per core,

$$\text{degrees of freedom per second (DoF/s)} = \sum_{i=1}^{n_{\text{cores}}} \frac{n_{\text{cells}} k^d n_{\text{SIMD lanes}}}{t_{i,\text{avg}}},$$

accumulating the throughput over all cores. Likewise, GFlop/s numbers are computed from the DoF/s numbers based on the operation counts per degree of freedom like the one reported for the cell terms of the 3D Laplacian in Table 3. We have verified with hardware performance counters through the tool likwid [59] that the expected number of arithmetic operations is actually executed.

Our experiments are performed on the three Intel HPC systems presented in Table 2, including two server processors and a throughput-oriented Xeon Phi (Knights Landing, KNL). We have verified that all benchmarks were run at the maximum AVX frequency specified in the table. The GNU compiler `g++` version 6 with flags described in Table 2 is used on all systems. For our code `g++` generates executables of slightly better performance (0–10%) than `clang` 3.9.0 and the Intel compiler versions 16, 17, 18.

The Haswell configuration with 2×8 cores features a relatively high memory bandwidth as compared to arithmetic throughput. The Broadwell system with 2×14 cores theoretically offers twice the arithmetic performance of the Haswell system, but with the same number of 8 DDR4 memory channels and only slightly higher STREAM memory throughput due to a higher memory frequency. The Knights Landing processor has 64 cores that run with wider vector units but at a low frequency of 1.1 GHz and with reduced features, the most important of which are the only 2-wide decode throughput and the absence of a level-3 cache [27]. On the other hand, KNL provides a high-bandwidth memory interface, which is addressed in flat mode with `numactl` [27] in all our experiments.

3.1 Code choices for sum-factorization sweeps

For simple operators such as the Laplacian or advection, the sum-factorization sweeps for interpolation and integration contribute with around 70% of arithmetic costs at $k = 1$ and more than 85% for $k \geq 6$, see Table 3 and the algorithms from Section 3.2 below. Thus, efficient operator evaluation critically depends on these operations. At the same time, the work at quadrature points is not negligible, and it becomes even more important for more complicated operators, for instance the compressible Navier–Stokes equations [15]. Hence, a close integration between sum-factorization sweeps and quadrature-point operations is desirable. For example, SIMD vectorization needs to be applied to both.

Due to the small dimensions of the coefficient arrays S and D^{co} with k rows and columns and matrices of dimension $k \times k^2$ for the 3D solution values $\mathbf{u}^{(e)}$, generic BLAS multiplication kernels `dgemm` that are specialized for the LINPACK context of medium and large sizes are not suitable due to large function call, dispatch, and data rearrangement overheads. As an alternative, optimized small matrix multiplication kernels have been suggested by the batched BLAS initiative [13] and

Table 3: Number of arithmetic operations per degree of freedom for evaluating the cell integrals of the 3D Laplacian $(\nabla\varphi_i, \nabla u_h)_K$ with 12 sum-factorization sweeps for basic matrix-matrix multiplication implementations as well as using the even-odd decomposition for the one-dimensional operations described in Section 3.2.

polynomial degree $p = k - 1$	1	2	3	4	5	6	7	8	9	10
	number of arithmetic operations, Flop/DoF									
sum fact., basic matrix multiply	36	60	84	108	132	156	180	204	228	252
sum fact., even-odd matrix multiply	36	44	60	70	84	94	108	119	132	143
operations at quadrature points	18	18	18	18	18	18	18	18	18	18
	of which fused multiply-add (FMA) instructions									
sum fact., basic matrix multiply	12	24	36	48	60	72	84	96	108	120
sum fact., even-odd matrix multiply	0	4	12	17	24	29	36	41	48	53
operations at quadrature points	6	6	6	6	6	6	6	6	6	6

by the `libxsmm` project [18]. While these interfaces could provide a path towards standardization, their suitability for moderate polynomial degrees $p \leq 10$ is not clear a priori: A black-box matrix multiplication interface will typically lose optimization opportunities of repeated operations along different directions and quadrature points, which can be a substantial portion of a sum-factorization sweep given out-of-order execution windows of 200 instructions or more. Furthermore, the 1D interpolations over ξ_1, ξ_2 and ξ_3 directions with strides of 1, k and k^2 leads to additional "reshape" operations when addressed by a single matrix multiplication interface, i.e., load/store access between registers and the L1 cache or permute operations in case SIMD is used. Also, to enable batched BLAS calls [13] with low overhead, tens of thousands of arithmetic operations must be combined by batching together the work of many elements, which necessarily exceeds at least the L1 cache. In other words, aggregating enough matrix-matrix multiplications separates the individual sum-factorization sweeps far away from each other and from the quadrature-point operations, reducing data locality and thus possibly performance in the low and medium polynomial degree case.

In this work, we analyze possible code layouts enabling an optimizing compiler to generate highly efficient machine code. As we will show below, very good performance is obtained by letting the compiler perform the loop unrolling and register allocation for $p \leq 10$. However, it is essential that the loop bounds are specified by templates and the use of full SIMD vectors is forced by intrinsics. For our implementation, vectorization is mostly transparent to the user code by vectorized data types and operator overloading. In an implementation in C++ like the one in deal.II, the full set of sum-factorization sweeps can be handled by three functions in transpose and non-transpose flavors, namely a function to change the basis with the matrices $\left[S_d \otimes \ldots \otimes S_2 \otimes S_1\right]$ according to (8), by a second one to compute the derivative in the collocation space, and face-normal interpolation from (9).

3.2 Implementation of matrix-matrix multiplications

We start the analysis by considering implementation options for the small matrix-matrix multiplications in sum factorization according to Eqs. (6) and (8) with operation counts from Table 3. Several steps beyond a basic implementation with run time loop bounds are tested:

- **Template parameter on loop bounds.** This optimization makes loop bounds a compile-time constant and is essential for the short loops at small polynomial degrees, as it allows the compiler to completely unroll the loops and to re-arrange operations to improve instruction flow. In the results denoted "templated loop bounds", no further loop optimizations are specified in the C++ code, with one accumulator in the innermost loop. The compiler typically places the data input along a one-dimensional line of the data array in registers for $p < 10$ and loads the entries in the 1D interpolation matrix from memory, i.e., L1 cache.
- For higher degrees, the compiler's heuristics do not generate optimal matrix multiplication code from the templated loops alone. Throughput is considerably improved by **loop un-**

rolling and register blocking as classically used in state-of-the-art matrix-matrix multiplication `gemm` kernels and appearing in `libxsmm` [18]. For the reported results, we manually apply 4×3, 5×2, and 8×1 unrolling with 12, 10 and 8 independent accumulators, with the first number referring to the block size along the rows of the coefficient matrices S or D^{co}, respectively, and the second number the block size aggregating over several lines in the array $\mathbf{u}^{(e)}$. At low degrees where not enough data streams are available for blocking, appropriate remainder code is generated. For a practical implementation of a given size k, the unrolling strategy with the lowest amount of remainder code is most beneficial.

- **Even-odd decomposition of local interpolation.** For the case that integration points are symmetric, shape functions are symmetric, and derivatives skew-symmetric with respect to the center of the 1D reference cell, i.e., $\varphi_i^{\text{1D}}(\xi_q) = \varphi_{p+2-i}^{\text{1D}}(\xi_{n_q+1-q})$, there are only $k^2/2$ unique entries of the k^2 total entries in the 1D interpolation or derivative matrices S and D^{co}. Thus, working separately on the even and odd components of the vector [35, Sec. 3.5.3], e.g. the four components of $k = 4$,

$$u_{1,\text{e}} = u_1 + u_4, \quad u_{2,\text{e}} = u_2 + u_3 \quad u_{1,\text{o}} = u_1 - u_4 \quad u_{2,\text{o}} = u_2 - u_3, \tag{11}$$

reduces the operation count for a one-dimensional kernel from $k(2k-1)$ arithmetic operations (k multiplications, $k(k-1)$ fused multiply-add operations, FMAs) to

$$2k \text{ additions/subtractions, } k \text{ multiplications, and } \lfloor k(k-2)/2 \rfloor \text{ FMAs.} \tag{12}$$

The separate additions and subtractions are due to extracting the even and odd contributions via (11) and similarly to transform the even and odd result of the 1D interpolation back to the values at the physical quadrature points, wheres multiplications and FMAs are spent in two matrix-vector products of size $\lfloor k/2 \rfloor \times \lceil k/2 \rceil$ each. Note that it is not feasible to perform all computations in the even and odd contributions directly because quadrature-point operations cannot be assumed to exhibit the symmetry of the shape functions, and thus an in-place decomposition is needed for every access. Table 3 shows the number of arithmetic operations per DoF for the 3D cell Laplacian with even-odd decomposition as well as for a basic matrix multiplication that ignores these symmetries. The numbers in Table 3 multiply the result of formula (12) by 12 (number of sum-factorization sweeps from Table 1) and normalize to one DoF by division by k.

Fig. 3 displays the performance of these variants in terms of DoF/s and the arithmetic throughput in terms of GFlop/s, using explicit vectorization by doing the same operation on each SIMD lane (see "vectorized over cells plain" in Sec. 3.3 below). Alongside the results, the right panel of Fig. 3 also plots the attainable GFlop/s rate for both the standard matrix-matrix multiplications and the even-odd decomposition given the SIMD execution ports of Broadwell, which is lower than the arithmetic peak due to the instruction mix with FMAs and separate additions and multiplications, especially for the even-odd case according to (12). The throughput of non-templated loops is more than three times lower than that of all other options. The 8×1 blocked variant of the basic matrix-matrix multiplications runs at up to 750 GFlop/s in 3D, i.e., 58% of arithmetic peak or 61% of the attainable peak. In 2D, the 5×2 blocked variant reaches up to 900 GFlop/s or 75% of the attainable peak (red dotted line in right panel of Fig. 3). Especially for $p > 8$, the additional register blocking clearly helps performance. For comparison, the highly tuned `dgemm` implementation of the Intel MKL library (version 18.2) runs at up to 1035 GFlop/s on the same system with matrices of 12,000 rows and columns. Note that the actual performance limit is often a combination of the L1 cache access and instruction latencies rather than the sole throughput of floating point units.

Despite the lower GFlop/s number for the even-odd decomposition, its DoF/s throughput is considerably higher than all variants of the standard matrix-matrix multiplications for $p > 1$. This is because around half the arithmetic operations according to Table 3 are run with an arithmetic throughput only 10–20% lower, i.e., up to 620 GFlop/s. Even assuming `dgemm` level throughput of the basic matrix-matrix multiplication at 1,000 GFlop/s, the even-odd decomposition provides higher throughput for degrees $p > 3$. These experiments highlight that the GFlop/s metric is only a secondary quantity as long as the algorithm is not fixed, subordinated to optimizing the

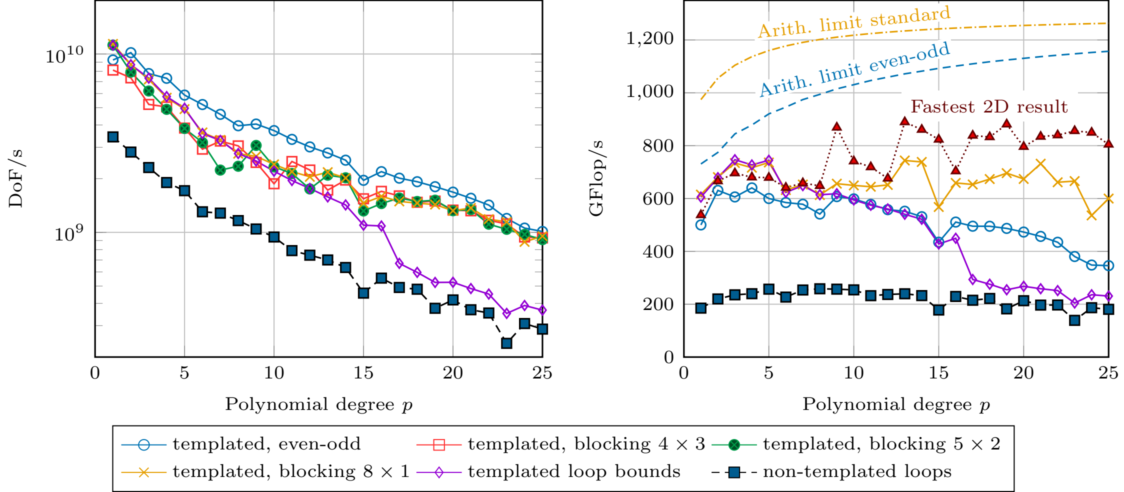

Figure 3: Arithmetic performance of implementation variants for matrix multiplications in sum-factorization sweeps of cell terms for the 3D Laplacian on fully populated 2 × 14 cores Intel Broadwell (Xeon E5-2690 v4 @ 2.9 GHz). The standard matrix-matrix multiplications are provided in five variants, three of which use register blocking with $r \times s$ accumulators, running through r local shape value rows in S and D^{co} and s layers of coefficients in $\mathbf{u}^{(e)}$.

DoF/s metric. At very high degrees, $p > 15$, the even-odd implementation could be enhanced by further loop unrolling/register blocking besides the natural 2 × 1 blocking for the even and odd contributions. In the remainder of this work, the even-odd decomposition is used for all sum-factorization sweeps, without additional register blocking. We note that the strategies below similarly apply to the other loop variants considered in this section.

Design Choice 1 *The results of this subsection have established the conclusion that it is important to implement the sum-factorization sweeps with compile-time loop bounds based on an algorithm that minimizes arithmetic operations by employing the collocation idea of equation* (8) *and an even-odd decomposition.*

3.3 Vectorization strategy

Modern high-performance CPU architectures increasingly rely on SIMD primitives as a means to improve performance per watt. An arithmetic or load/store operation is issued by a single instruction for n_{lanes} data elements in parallel. Cross-lane permutations require separate instructions that may incur a performance penalty, depending on the superscalar execution capabilities of the microarchitecture. Furthermore, loads and stores to a contiguous range of memory (packed operation) are faster than indirect addressing with `gather` or `scatter` instructions with multiple address generation steps.

We propose to vectorize over several cells which is an option that is simple to program. Fig. 4 shows a numbering of degrees of freedom on a $\mathcal{Q}_3$ basis in 2D with 4-wide vectorization which allows for direct packed access. The lower left node values of four cells are placed adjacent in memory. The next storage location is the second node for all four cells, and so on. In this format, no cross-lane data exchange is needed for cell integrals and the sum-factorization sweeps can be directly applied to the data stored in the global input vector without a separate gather step. This scheme can straight-forwardly be extended to operations at quadrature points and also select the most beneficial width of vectorization for a given hardware by using overloaded SIMD data types according to [41, 42]. Partially filled SIMD lanes occur at most on a single cell batch per operator evaluation for meshes whose number of cells is not divisible by the vectorization width. We also apply this approach to face integrals, i.e., we process the integrals of several faces at once, rather than SIMD-parallelizing within a face or over the two cells adjacent to a face. Besides changing the loop over the mesh, possible disadvantages of this scheme are

- a somewhat larger spread in the indices of gather/scatter steps of face integrals,

e_3				e_4				e_7			
50	54	58	62	51	55	59	63	114	118	122	126
34	38	42	46	35	39	43	47	98	102	106	110
18	22	26	30	19	23	27	31	82	86	90	94
2	6	10	14	3	7	11	15	66	70	74	78
48	52	56	60	49	53	57	61	112	116	120	124
32	36	40	44	33	37	41	45	96	100	104	108
16	20	24	28	17	21	25	29	80	84	88	92
0	4	8	12	1	5	9	13	64	68	72	76
e_1				e_2				e_5			

Figure 4: Visualization of layout of degrees of freedom for vectorization over elements, using an array-of-structure-of-array data layout.

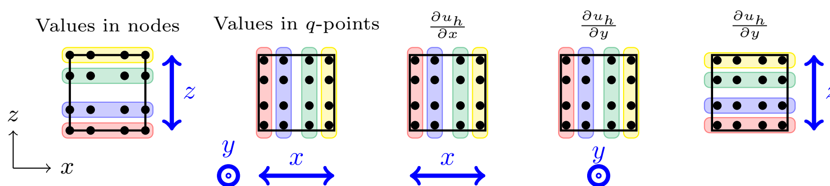

Figure 5: Visualization of a single-dimension vectorization scheme within a single element with degree $p = 3$ shown in $x - z$ plane with y-direction orthogonal to the sketch. Shaded areas collect the entries within a single SIMD array, involving transpose (cross-lane) operations to switch from one order to the other. Blue arrows show the action of 1D interpolations, one arrow per sum-factorization sweep to compute $\nabla_{\boldsymbol{\xi}} u_h$.

- cases where the number of locally processed elements is less than the width of vectorization do not see speedups, which is usually only limiting over the communication cost for more than 500 degrees of freedom per cell [44], and
- a large working set size of the temporary arrays in sum factorization, which might exceed the capacity of caches and thus slow down execution.

These disadvantages can be avoided when vectorizing the work done on a single cell instead. On the other hand, such strategies often must use different implementations for different sizes of the SIMD vectors [29]. Several strategies are conceivable, in particluar the reference [49] proposes to vectorize over gradient components. That strategy is, however, intimately tied to the representation (7). Here, we evaluate two flavors compatible with Design Choice 1 for comparison for the special case of 4-wide SIMD.

A first variant is to combine the values along a single direction in a SIMD array. We exemplify a variant in Fig. 5 where SIMD lanes collect data subject to the same 1D interpolation, i.e., no lanes are crossed during interpolation. Instead, three transpose operations of length k^d in the spirit of array-of-struct into struct-of-array conversions are needed to arrange the data properly for a particular stride in 1D interpolations along the various directions. In case the number of 1D degrees of freedom is not a multiple of the SIMD width, the last SIMD array is filled up with

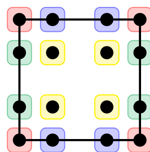

Figure 6: Visualization of vectorization scheme over quadrants within a single element for polynomial degree $p = 3$. Shaded areas of the same color collect the entries within a single SIMD array.

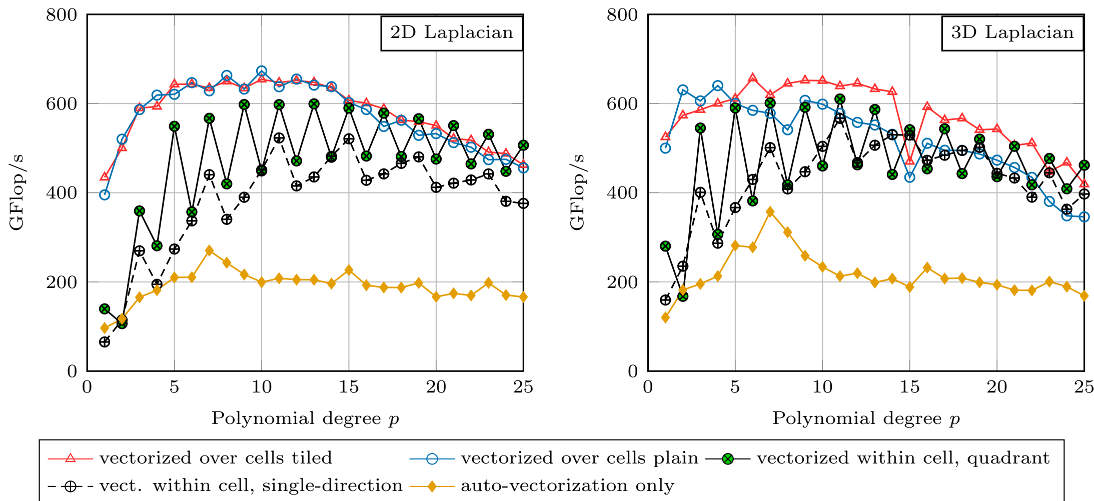

Figure 7: Comparison of throughput of local cell kernels on fully populated 2×14 cores of Intel Broadwell (Xeon E5-2690 v4) for cell integrals with respect to vectorization for Laplacian in 2D and 3D.

dummy values, which leads to distinct drops in performance.

A second variant for vectorization within a cell is to embed the lane-crossing within the 1D interpolations. A quadrant-based layout as displayed in Fig. 6 combines well with the even-odd decomposition. Here, a 4-wide SIMD lane contains the entries starting from the four corner degrees of freedom. Each interpolation then computes the even $u_{i,\mathrm{e}}$ and odd parts $u_{i,\mathrm{o}}$, e.g. in x-direction, within a lane by

$$[u_{1,\mathrm{e}}\ u_{1,\mathrm{o}}\ u_{3,\mathrm{e}}\ u_{3,\mathrm{o}}] = [u_2\ u_1\ u_4\ u_3] + \begin{bmatrix}1 & -1 & 1 & -1\end{bmatrix} .\!*\, [u_1\ u_2\ u_3\ u_4] \,.$$

Here, the first array on the right hand side is obtained by a permutation of the input order $[u_1\ u_2\ u_3\ u_4]$ representing e.g. the values in the corners according to Fig. 6 in the SIMD array (using the `vpermilpd` assembly command in this example), and the addition and subtraction in different lanes is realized by an FMA with multiplication of the respective sign. For the multiplication by S and D^{co}, a $\lceil k/2 \rceil \times \lceil k/2 \rceil$ matrix containing coefficients with odd and even parts in the appropriate lanes is used. Finally, the end result is obtained by an additional permutation and FMA. This option exactly fills all 4-wide SIMD lanes for all odd degrees p (i.e., $k = p + 1$ even), making it more flexible than the single-line vectorization. For the full interpolation for the 3D Laplacian, the number of instructions on the permute/shuffle port of Broadwell is similar for the single-direction vectorization and the quadrant vectorization.

The performance of the three vectorization variants, implemented through wrappers around intrinsics, is shown in Fig. 7. The floating point performance is related to operation counts of the even-odd decomposition according to Table 3 for all variants, without counting the excess operations due to partially filled SIMD lanes for vectorization within a cell. Vectorization over several cells shows the best performance and an approximately constant arithmetic throughput for all p, whereas vectorization within cells shows a distinct difference in GFflop/s throughput depending on the occupation of the SIMD lanes, with lower performance for even degrees and especially for $p = 4, 8, 12, 16, 20, 24$ for the single-direction vectorization. The quadrant-based vectorization performs better than the single-direction variant and is able to almost reach the performance of vectorization over cells for odd degrees p.

The figure also contains data points with automatic vectorization as exploited by the compiler with `__restrict` annotations to arrays to help in the aliasing analysis. Automatic vectorization is not competitive at less than half the throughput (similar numbers have been obtained with the Intel compiler). This is because only 5% to at most 25% of arithmetic instructions are done in packed form, with the best result for $p = 7$ (i.e., $k = 8$).

In order to reduce the cache pressure of the sum-factorization sweeps with vectorization over cells, Algorithm 3 proposes to merge loops over x and y within a single loop over the last direction z

ALGORITHM 3: Loop tiling for sum factorization, exemplified for the 3D cell Laplacian

- for $i_z = 0, \ldots, k-1$
  - Apply S_1 along x for k^2 points in xy plane with offset $i_z k^2$ and stride 1
  - Apply S_2 along y for k^2 points in xy plane with offset $i_z k^2$ and stride k
- for $i = 0, \ldots, k^2-1$
  - Apply S_3 along z for k points with offset i and stride k^2
  - Apply D_3^{co} along z for k points with offset i and stride k^2
- for $i_z = 0, \ldots, k-1$
  - Apply D_2^{co} along y for k^2 points with offset $i_z k^2$
  - for $i_y = 0, \ldots, k-1$
    - Apply D_1^{co} along x for k points with offset $i_z k^2 + i_y k$
    - Laplacian at k quadrature points along x with offset $i_z k^2 + i_y k$ according to Alg. 1, (ii)(c)
    - Apply $(D_1^{\text{co}})^{\text{T}}$ (integration) along x for k points with offset $i_z k^2 + i_y k$
  - Apply $(D_2^{\text{co}})^{\text{T}}$ along y for k^2 points with offset $i_z k^2$; sum into result from x direction
- for $i = 0, \ldots, k^2-1$
  - Apply $(D_3^{\text{co}})^{\text{T}}$ along z for k points with offset i; sum into results from x, y directions
  - Apply S_3^{T} along z for k points with offset i and stride k^2
- for $i_z = 0, \ldots, k-1$
  - Apply S_2^{T} along y for k^2 points in xy plane with offset $i_z k^2$ and stride k
  - Apply S_1^{T} along x for k^2 points in xy plane with offset $i_z k^2$ and stride 1

(applied to step (ii)(b) and (ii)(d) in Algorithms 1 and 2). The corresponding data point is marked as "vectorized over cells tiled" in Fig. 7. As opposed to the inner register blocking explored in Sec. 3.2, this tiling is across different sum-factorization sweeps. The options for register blocking (which is not used here) are limited somewhat due to the lower dimensionality in the inner loop dimensions. In 2D, performance is similar apart from slight code generation differences by the compiler. For low degrees in 3D, performance is slightly reduced due to latency effects,[2] but it is higher for higher degrees.

Further details on the vectorization variants are provided by a cache access analysis. Fig. 8 shows measurements of the actual data transfer (read + evict) between the various cache levels of Intel Broadwell Xeon with 4-wide vectorization extracted from hardware performance counters with the `likwid` tool [59] with `likwid-perfctr -C 0-27 -m -g CACHES`. Raising the polynomial degree increases the size of the temporary arrays holding intermediate results of sum-factorization sweeps. For vectorization over cells, degrees larger than 5 spill to the L2 cache and degrees larger than 10 spill to the L3 cache. The tiled algorithm approximately halves the cache access because data is aggregated along 2D planes and only 5 instead of 13 outer loops are run. Assuming perfect caching of inner layers and the coefficients, the tiled algorithm performs 7 read and 7 write accesses per DoF rather than 17 reads and 15 writes for the untiled one. An apparent outlier in Fig. 8 is degree 15 with $k = 16$ on 16^3 degrees of freedom per cell which is affected by cache associativity limitations due to access to 64 entries at a distance of exactly $8\,\text{kiB} = 256 \times 32$[Bytes]. For vectorization within cells, the active set is smaller by a factor of four approximately. Spilling to L2 cache only becomes significant at degree 8 and to L3 cache for degrees larger than 15.

The reduced cache pressure when vectorizing within cells does not materialize in better performance, though. This is because more transfer from outer level caches comes along with more arithmetic due to the linear complexity $\mathcal{O}(k)$ per DoF, offsetting the reduced throughput and increased latency of outer level caches. For example, the L2 cache of Broadwell can sustain around

[2] The tiled algorithm is up to 8% faster with 2-way hyperthreading enabled at 640 GFlop/s on $p = 2$ or 710 GFlop/s for $p = 8$. For the plain sum-factorization sweeps, performance is not significantly increased with hyperthreading.

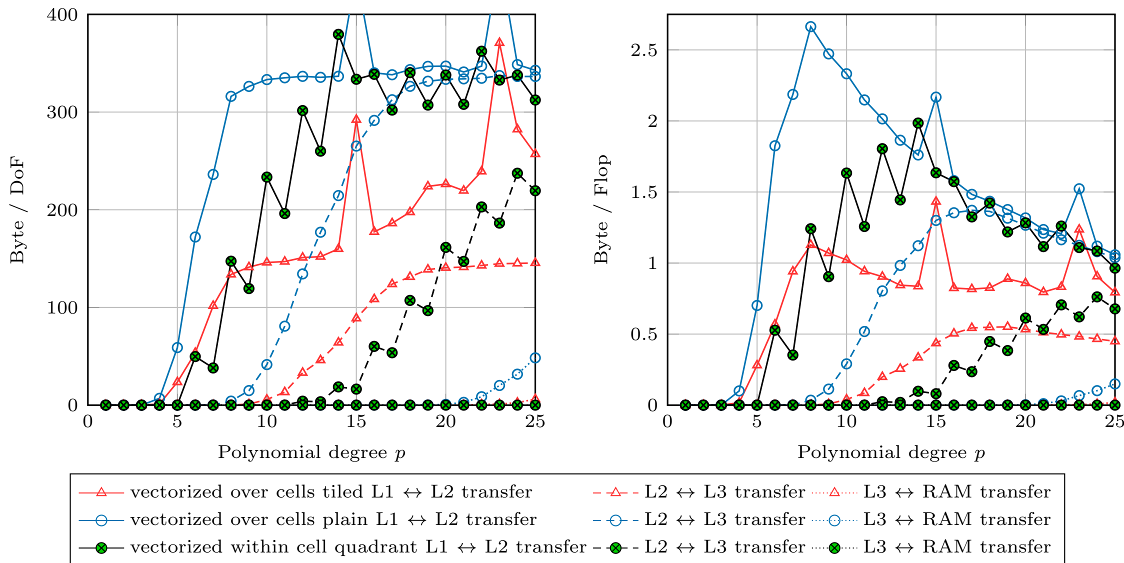

Figure 8: Memory access on various levels of the memory hierarchy for the compute part of the cell integral of the 3D Laplacian, listed as Bytes/DoF (left panel) and Bytes/Flop (right panel).

30 Bytes/cycle compared to 16 peak Flop/cycle [26]. According to the right panel of Fig. 8, the "vectorized over cells plain" data point at $k = 8$ with 2.67 Byte/Flop runs at a flop rate around half of the arithmetic peak, so L2 bandwidth is not a hard limit, even though a relaxation of the cache access as provided by the tiling indeed leads to slightly better performance. Likewise, throughput of the L3 cache is around half that of L2 cache in recent Intel architectures, which again fits well with the access patterns in sum factorization according to Fig. 8 (dashed lines).

Note that the cache behavior is intimately linked with the architecture. For example, the KNL or Skylake-SP microarchitectures with 8-wide vectorization and faster L1 caches do indeed push the L2 cache interface to its limit, and the throughput is considerably higher for the tiled algorithm than for the plain sum-factorization sweeps. Likewise, the overall capacity of caches on KNL (1MB L2 cache for two cores) is not enough to hold all temporary arrays for degrees beyond 11 in 3D, spilling to memory.

Design Choice 2 *From the results of this subsection, we conclude that vectorization over cells yields best performance. Cache pressure can be reduced by using tiling over the sum factorization sweeps.*

We note that the performance reported by [49, 29] indeed suffers from a performance deficit by a factor of three to four due to the higher operation count of their algorithm tied to a particular vectorization choice, reflected in the final throughput of operator evaluation compared to our results in Sec. 4 below.

3.4 Compute performance on CPUs and Xeon Phi

We now analyze the throughput of the cell integrals for the Laplacian, implemented with even-odd decomposition, loop tiling and templated loop bounds, on the three Intel processors listed in Table 2. The result is shown in Fig. 9. On Haswell, 320–340 GFlop/s are recorded in both 2D and 3D for degrees p around 10. This represents 40–50% of the arithmetic peak or up to 70% of the possible arithmetic performance for the given instruction mix, similar to Broadwell. Similar GFlop/s rates are also recorded for mass matrices (at 2× the DoF/s throughput) and advection (at 1.5× the DoF/s throughput).

The performance on the KNL many-core processor is less regular. In 3D, a distinct decrease in arithmetic performance is observed for $p \approx 11$ where the local data arrays exhaust the 512 kiB of L2 cache per core and an increasingly larger part of the temporary arrays for sum-factorization sweeps needs to be fetched from the MCDRAM memory. Note that performance for $p > 10$ is only

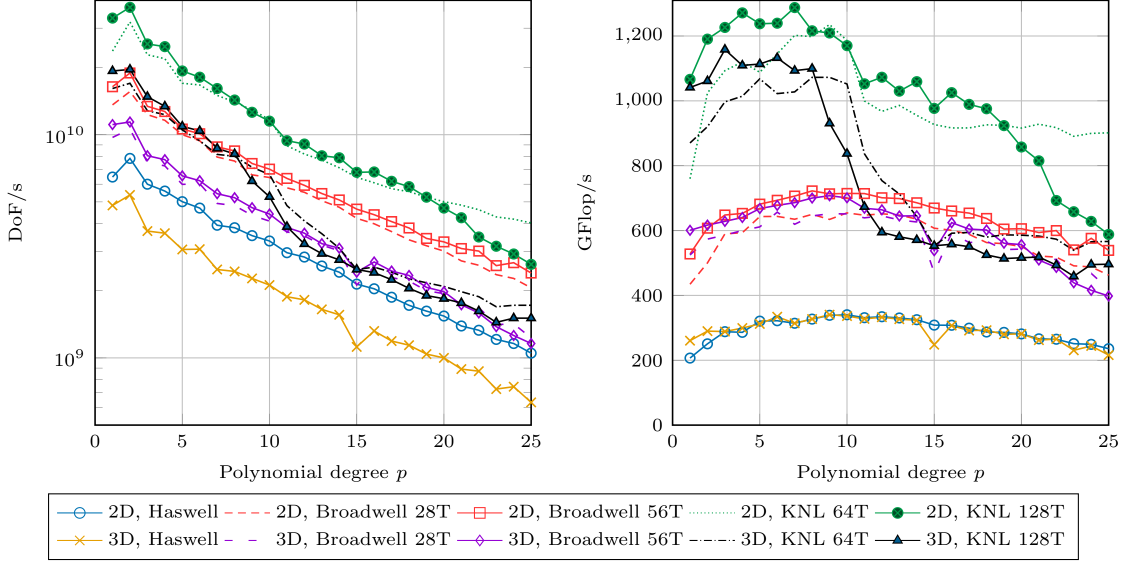

Figure 9: Throughput of local cell integrals of Laplacian in 2D and 3D on 2×8 core Intel Xeon E5-2630 v3 (Haswell), 2×14 core Intel Xeon E5-2690 v4 (Broadwell), and 64 core Intel Xeon Phi 7210 (KNL). Broadwell and KNL are run with one thread per core (28T/64T) as well as two-way hyperthreading (56T/128T).

around 80 GFlop/s when binding the kernel to the slow DDR4 RAM with `numactl --membind=0` rather than the fast 16 GB of MCDRAM [27]. The tensor product tiling of Algorithm 3 is more important on KNL than on the CPUs. We reach up to 1.3 TFlop/s for low to moderate polynomial degrees or 57% of the arithmetic peak, which is extremely good given the simplified hardware of the KNL microarchitecture. This number can be compared to 56% of peak performance (720 GFlop/s) recorded on Broadwell with 2-way simultaneous multithreading (hyperthreading) for degree $p = 8$ in 2D and 3D. For low degrees $p \leq 8$ in 3D and for $p \leq 19$ in 2D, 2-way hyperthreading (128 threads) increases throughput due to latency hiding, but the picture is reversed for higher degrees when too much data is in flight; a similar behavior regarding hyperthreading is also observed on Broadwell. For the remaining experiments, KNL and Broadwell are run with 2-way hyperthreading unless noted otherwise, whereas Haswell has hyperthreading disabled in all experiments.

3.5 Cell and face integrals on CPUs and Xeon Phi

Finally, we look at cell and face integrals together. For efficient evaluation of face integrals, we use a nodal basis in the nodes of the k-point Gauss–Lobatto–Legendre quadrature formula for advection, i.e., one node placed at each of the 1D interval end points. This allows to directly pick the k^{d-1} node values on the face according to Fig. 1 without interpolation normal to the face over all k^d points. Likewise, a Hermite-type basis is chosen for the Laplacian (2) according to Sec. 2.3. Gaussian quadrature on k^d and k^{d-1} points for cells and faces, respectively, is used. In analogy to Table 3 for cell operations, Table 4 lists the number of arithmetic operations per degree of freedom, both in the variant with compact face integrals and with element-wise face integrals using the optimization of the basis change for the interior portion of the face interpolations according to Equation (10). The final operation counts are similar for the two variants, with the compact face integral variant involving somewhat more work in face sum-factorization sweeps, and the element-wise face integrals involving more work at the face quadrature points.

Fig. 10 presents the throughput for the "compact face integrals" variant on the three hardware systems of Table 2 for a benchmark representing the artificial case of periodic boundary conditions within the same cell that can directly use the vectorized data storage according to Fig. 4 in order to ignore the cost of indirect addressing. The data in Fig. 10 confirm the performance results from the cell integrals in the previous figures, with up to 650 GFlop/s on Broadwell and 1030 GFlop/s on KNL. As can be seen by comparing Table 4 with the sum of the second and third data rows of

Table 4: Number of arithmetic operations per degree of freedom (Flop/DoF) for evaluating the cell and face integrals of advection and the Laplacian with even-odd decomposition in sum-factorization sweeps. Both the variant of compact face integrals according to Algorithm 1 as well as the element-wise face integrals of Algorithm 2 with face interpolations of the interior shared with cell integrals according to (10) are listed.

polynomial degree $p = k - 1$	1	2	3	4	5	6	7	8	9	10
2D advection, compact face integral	46	38	50	49	59	61	69	72	80	84
2D advection, element-wise face int.	57	49	59	59	68	70	78	81	89	93
3D advection, compact face integral	90	70	90	88	102	105	118	122	134	139
3D advection, element-wise face int.	86	72	88	88	101	104	117	121	133	139
2D Laplacian, compact face integral	109	83	98	94	106	106	117	120	131	135
2D Laplacian, element-wise face int.	114	95	111	108	121	122	134	137	148	152
3D Laplacian, compact face integral	238	177	206	191	212	209	226	228	244	249
3D Laplacian, element-wise face int.	236	183	211	199	219	217	234	238	254	259

Table 3, face integrals contribute with more than two thirds of the arithmetic work of the Laplacian up to cubic polynomials. Since the cost of face integrals scales as $\mathcal{O}(1)$ (sum-factorization sweeps) and $\mathcal{O}(1/k)$ (operations at quadrature points), an approximately constant DoF/s throughput at low degrees $2 \leq p \leq 6$ is observed. For the 3D Laplacian, a throughput of more than $4 \cdot 10^9$ degrees of freedom per second on KNL for $1 \leq p \leq 7$ and more than $2.6 \cdot 10^9$ degrees of freedom on Broadwell for $3 \leq p \leq 9$ can be reached. For the same in-cache test of the "element-wise face integrals" of Algorithm 2, GFlop/s numbers are very close to the results from Fig. 10 and not reported separately. According to Table 4, the DoF/s throughput between Algorithms 1 and 2 is also very similar.

4 Data access patterns and parallelization

In this section, we analyze the performance of the operator evaluation including the actual data access patterns of DG cell and face integrals into the input and output vectors $\mathbf{x}$ and $\mathbf{y}$ of (4), as well as parallelization. As in Sec. 3, we assume constant factors $\det(\mathcal{J}_{(e)})\mathcal{J}_{(e)}^{-1}\mathbf{c}$ (advection) $\det(\mathcal{J}_{(e)})\mathcal{J}_{(e)}^{-1}\mathcal{J}_{(e)}^{-\mathrm{T}}$ (Laplacian) at all quadrature points, i.e., only d or $d(d+1)/2$ data points are needed for the full mesh, which corresponds to evaluating constant-coefficient operators on an affine domain.

To exploit the parallelism of multi-core processors that are connected by high-speed networks in modern petascale machines, two parallelization concepts are commonly used, the shared-memory paradigm of OpenMP/pthreads and the distributed memory paradigm implemented by the message passing interface (MPI). Increasingly, a mixture of both is applied, separating intranode and internode parallelism, respectively. MPI parallelism for finite elements usually relies on domain decomposition to partition the cells in the mesh among the processors. For the exchange of information between subdomains, the locally owned subdomain is augmented by ghost elements on each MPI rank. In our implementation, we assume one layer of ghost elements around the owned cells, supported by the massively parallel algorithm from [6]. The particular form of the mesh partitioning is immaterial, as long as the information provided by the mesh infrastructure allows for a unique identification of the degrees of freedom in the locally owned and ghosted cells.

For shared-memory parallelism, loops over the mesh entities are split across the participating threads. Algorithm 2 computing face integrals element-wise allows for a straight-forward use of shared memory, since all of a cell's integrals are computed without interference from other threads. We consider both OpenMP and MPI parallelization for this variant. For the compact face integral variant in Algorithm 1, however, some coordination is necessary to avoid race conditions when integrals from several faces go to the same vector entries. This is because face integrals are scheduled freely and all faces of an element could be scheduled to different threads (and thus conflict) in the worst case. As these synchronization models are not yet well-developed in the finite element community, only MPI is considered as a shared memory model for Algorithm 1.

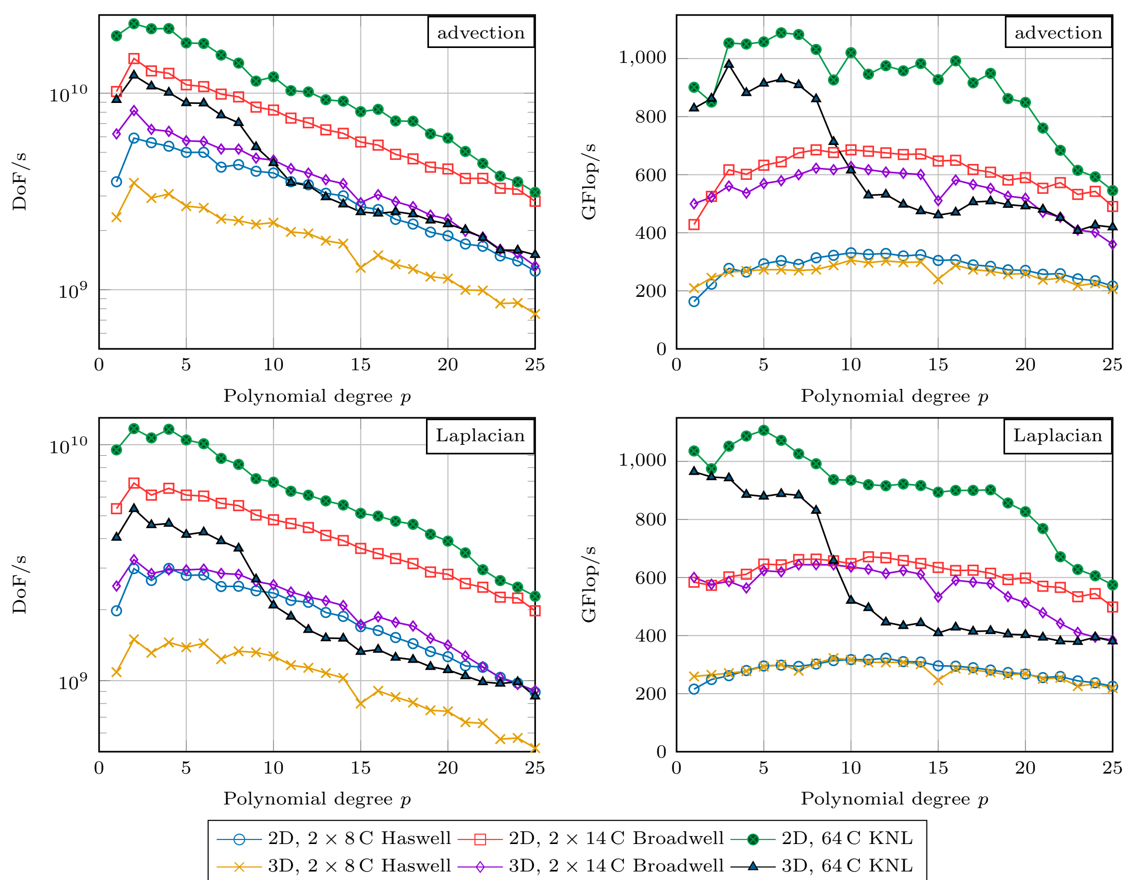

Figure 10: Throughput of cell and face integrals together, variant "compact face integrals", for advection and Laplacian in 2D and 3D without vector access on 2×8 core Intel Xeon E5-2630 v3 (Haswell), 2×14 core Intel Xeon E5-2690 v4 (Broadwell) and 64 core Intel Xeon Phi 7210 (KNL). Broadwell and KNL are run with 2-way hyperthreading.

Table 5: Arithmetic intensity in Flop/Byte for cell integral of 3D Laplacian with vector access of 24 Bytes per DoF, assuming a global affine geometry.

polynomial degree $p = k - 1$	1	2	3	4	5	6	7	8	9	10
Flop/Byte for 3D cell Laplacian	2.2	2.6	3.2	3.7	4.2	5.7	5.2	5.7	6.2	6.7

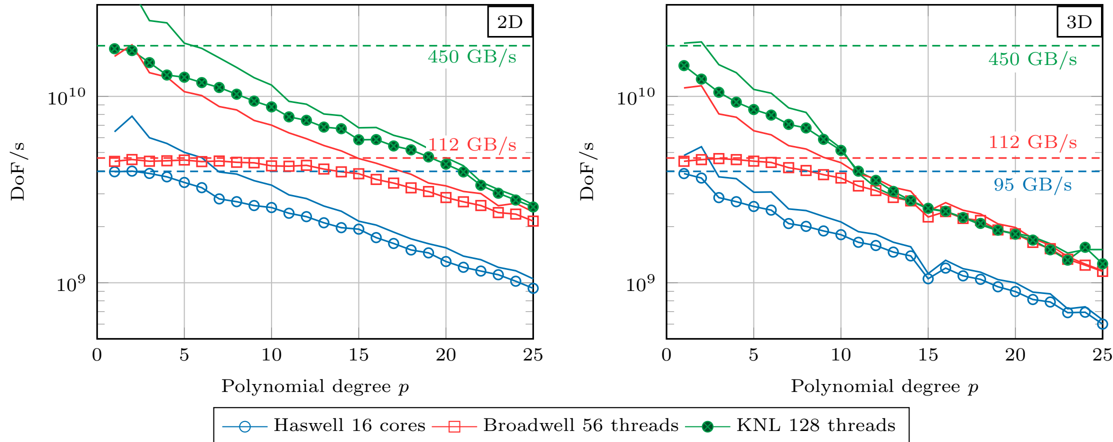

Figure 11: Performance of cell integrals of Laplacian in 2D and 3D including the global vector access on 2×8 core Intel Xeon E5-2630 v3 (Haswell), 2×14 core Intel Xeon E5-2690 v4 (Broadwell) and 64 core Intel Xeon Phi 7210 (KNL). Broadwell and KNL use two-way hyperthreading. The solid lines indicate the computational throughput according to Fig. 9 and the dashed lines the memory bandwidth with stream triad at two reads, one write, matching the data access in this benchmark.

4.1 Vector access analysis for cell integrals only

We first consider the access to global input and output vectors, running the full phase (ii) of Algorithm 1 on $\lfloor 5 \cdot 10^7/k^d \rfloor$ cells, representing vectors of about 50 million unknowns. The compute part corresponds to the experiment in Fig. 9, i.e., using even-odd decomposition and vectorization over cells with sum factorization tiling according to Algorithm 3. Due to vectorization over several cells, the cell operations are performed for a batch of `SIMD_WIDTH` cells in one loop iteration of the algorithm. Indices are laid out in interleaved form according to Fig. 4, i.e., sum factorization can directly operate on the respective sections of the vectors. The test is repeated 500 times and arithmetic averages are reported according to the procedure in Sec. 3, with standard deviations below 2%. The vector size eliminates cache effects and results in one vector read for the input vector, one vector write for the output vector, and one vector read operation on the output vector due to the read-for-ownership memory access pattern [17]. Since no interactions with neighbors appear for this test case, the memory access is of streaming character, alternating with large blocks of computations on cached data in steps (ii)(b)–(ii)(d) of Algorithm 1. The performance of this test case is relevant in case cell and face integrals are done in separate loops, for applying DG mass matrices or inverse mass matrices [56, 15], or for continuous finite element algorithms with DG data storage, as e.g. used by Nek5000 [16].

Table 5 shows the arithmetic intensity for $1 \leq p \leq 10$, computed as the ratio of the arithmetic operations in Table 3 and the vector access of 24 Bytes/DoF. By comparing the machine balances of Haswell, Broadwell, and KNL at 7.0, 11.6 and 5.0 Flop/Byte, respectively, with the arithmetic intensities of the 3D Laplacian from Table 5 and further taking into account that an arithmetic efficiency of around 50% of peak can be achieved due to local instruction mix and core limitations as elaborated in Sec. 3, the roofline model [61] predicts that degrees $p \leq 4$ are memory-limited on Haswell, $p \leq 9$ are memory-limited on Broadwell, and $p = 1$ on KNL, whereas higher degrees are core-limited. This matches well with the computational results in Fig. 11. The results are presented together with the two performance limits, namely the memory bandwidth of the vector access as measured by a STREAM triad test (95 GB/s, 112 GB/s, and 450 GB/s for Haswell, Broadwell, and KNL, respectively) and the compute throughput from Fig. 9. A strong impact of the memory bandwidth on throughput especially in 2D and for low and medium polynomial

degrees $k \leq 10$ can be observed. The gap at intermediate degrees is due to effects not captured by the simple distinction between main memory bandwidth limit and arithmetic performance in a roofline fashion, but could be explained by a more detailed representation of the cache hierarchy such as the execution cache memory model [17]. For example, KNL reaches around 900 GFlop/s for polynomial degrees between 5 and 9 in 3D, around 15% less than what is reported for the in-cache compute of Sec. 3.4. Apart from these effects, the envelope established by in-core performance and memory bandwidth closely describes the achieved performance.

Since the arithmetic intensity of 3D cell integrals at $1 \leq p \leq 10$ is between 2 to 7 Flop/Byte, close to the machine balance of today's hardware, it can be concluded that optimal execution of arithmetic work must be combined with memory access patterns that do not waste bandwidth. As has been proposed in Algorithm 1(ii), all operations belonging to the cell integrals must be done within a single loop through the vector data such that data loaded to caches can be reused. This includes a possible copy of the vector portions $\mathbf{u}^{(e)}$ into formats more amenable to vectorization or basis change operations. A separate global loop over the mesh would increase the memory traffic by several times and lower throughput significantly unless all vector data fits into the last-level caches. Even though there are phases with several tens of thousands of instructions without memory access, close to full memory utilization can be achieved thanks to hardware prefetching that eagerly pre-loads the vector entries before the actual read operation is executed on all three architectures. Note that no software prefetching is used in our experiment. This result shows that the memory-intensive operations in the vector access can indeed be mixed with the compute-dominated interpolation and integration steps. Hardware performance counter verify that the data that flows between the phases (a)–(e) of the algorithm remains in cache memory according to the results of Fig. 8.

The present experiment and the roofline model also predict what will happen once face integrals are added to the picture: The combined arithmetic cost of cell and face integrals in Table 4 with a single access to the source and destination vectors gives rise to arithmetic intensities around 7 to 10 Flop/Byte for the 3D Laplacian. For example on Broadwell, the compute throughput of at most 3 GDoF/s for the 3D Laplacian and 7 GDoF/s for 3D advection according to Fig. 10 is at a comparable level as the memory throughput of two loads and one store at around 4.7 GDoF/s. Thus, vector entries loaded to caches for cell integrals that are re-used for face integrals promise significantly better performance as two sweeps through data, which would at best achieve 2.3 GDoF/s. This can be achieved by interleaving the cell loop (ii) with the inner face loop (iii) and the boundary face loop (iv) in Algorithm 1. Algorithm 2 enforces this locality by design. For curved geometries where separate geometric tensors are loaded at each quadrature point, the Flop/Byte ratios are lower and a reduction in memory transfer for the vector access as given by the interleaved loop execution can improve performance also on machines with high-bandwidth memory such as KNL, albeit not by integer factors.

Note that the importance of memory transfer is the result of optimizing the compute phase first with the aim to keep the arithmetic work to a minimum.

4.2 Vector access for face integrals with vectorization over cells

Since all DoF are local to a cell in DG without continuity constraints that link to neighboring cells, a single index per cell suffices to represent the local–global index translation even on unstructured meshes. Furthermore, the index layout can be set up as exemplified in Fig. 4 to ensure direct SIMD array access on cells. However, this setup inevitably leads to some indirect access to vector data in face integrals. For the element-based face integrals of Algorithm 2, access to u^+ requires a gather operation. For compact face integrals, the role of u^- and u^+ is arbitrary, and thus, indirect access can appear on either side. The decision between a fast contiguous read path and a slower gather/scatter path depends on the mesh and is made at run time by conditional jumps. Furthermore, run-time decisions are involved for different local face numbers within the cell from both sides u^- and u^+ (e.g. for face-normal interpolation) or for faces that are not in the standard orientation with respect to the cell's local coordinate system, an unavoidable case on general 3D meshes as opposed to the 2D case [2]. This dispatch overhead can be kept small by organizing the vectorization such that (most) decisions are the same for all SIMD lanes. For compact face integrals, the two aims of keeping decisions uniform across SIMD lanes (e.g. the imposition of boundary conditions) and of performing face integrals after cell integrals to ensure "set" semantics

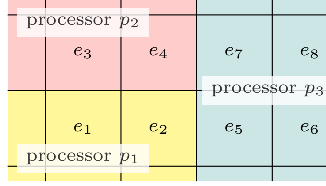

Figure 12: When three processors meet at a corner, it is possible that one processor cannot have the same stride on all SIMD lanes in its ghosted cells.

in cell integrals according to Algorithm 1 imply that some faces involve indirect access on both sides. The optimization of this process is beyond the scope of this work, and a greedy approach is used in the code available through the deal.II library and the experiments below. As a result, faces with indirect addressing on both sides are about as many as there are faces on subdomain interfaces with MPI.

Besides one 32-bit integer for the vector index for each SIMD lane, one 8-bit integer is needed for identifying the local face number, one 8-bit integer for identifying a possible face on a hanging node (for 2:1 mesh refinement ratio), and one 8-bit integer for the face orientation. Compared to the access of data, e.g. 4 values for each SIMD lane already for $p = 1$, the memory consumption for this metadata is well below 10% of all data access on the face, often below 2%. Following the mathematical formulas (9), we keep the number of rearrangements with different coordinate systems in different SIMD lanes to a minimum by ensuring that the normal derivative component to a face is always the last component in the local coordinate system, and arrange geometry terms such as the Jacobian $\mathcal{J}_{(e)}$ appropriately.

For the indirect gather/scatter access on faces, e.g. the faces on the exterior right of the cells $\{e_1, e_2, e_3, e_4\}$ in Fig. 4, extraction of data along the vertical axis accesses $\{1, 64, 3, 66, 17, 80, 19, 82, \ldots\}$. With a gather operation and index offsets $\{1, 64, 3, 66\}$, the data values on the cells can be accessed with the same offset, e.g. `SIMD_WIDTH * local_offset = 4 * 4`.

In an MPI parallelization where a single layer of ghost cells is added around the locally owned cells, however, a third case besides contiguous access and gather with the same stride must be considered. As an example, consider the case of `SIMD_WIDTH=2` for a cell at the corner with two additional processors according to Fig. 12. Further, assume that the face $e_2 \cap e_4$ is processed by processor p_2 and face $e_2 \cap e_5$ by p_3, respectively. If the indices on p_1 are interleaved between e_1 and e_2, this interleaved access is also possible on p_2. However, p_3 has no knowledge of e_1, so the ghost indices transformed to the MPI-local index space on p_3 appear to have stride 1 for e_2. This case is handled by scalar data access to each SIMD lane in our implementation.

4.3 Partitioning of faces for compact face integrals with MPI parallelization

For element-wise face integrals, ghost data from all cells around the locally owned cells must be imported in the MPI setting. For compact face integrals, all integrals associated to a face are computed at the same time. It must then be decided how to divide the integrals on faces between different MPI processes. The import of ghost data from the source vector in Algorithm 1 only involves half as much data as Algorithm 2, which is however compensated by an equivalent send operation of the resulting face integrals on faces to ghosted cells back to the owner. Assuming a balanced domain decomposition of cells, we propose to split face integrals *pairwise* on each interface $p_i \cap p_j$ of two processes p_i and p_j to reach balanced work on faces. We set the following two restrictions:

- If the set of shared faces F_{ij} along the interface between p_i and p_j contains two faces f_a and f_b referring to a single cell e from processor p_i, we schedule both integrals on p_j. This approach reduces the amount of data sent, since the data from e sent from p_i to p_j is used for two (or more) face integrals.
- If a face $f \in F_{ij}$ is on an interface of different mesh levels (hanging node) where p_i is on the coarser side and p_j on the refined side, we schedule the integral on p_j. If p_j holds all children,

Table 6: Description of operator evaluation test case on an affine geometry in 2D and 3D.

	two space dimensions	three space dimensions
mesh	$p = 1, 2$: 2048^2; $p = 3 \dots 5$: 1024^2	$p = 1, 2$: 128^3; $p = 3 \dots 5$: 64^3
	$p = 6 \dots 11$: 512^2; $p = 12 \dots 23$: 256^2	$p = 6 \dots 11$: 32^3; $p = 12 \dots 23$: 16^3
number of DoF	11.1M. . . 37.7M	9.0M. . . 56.6M
property	globally affine, cube topology, Morton ordering of cells, including MPI partitioning	

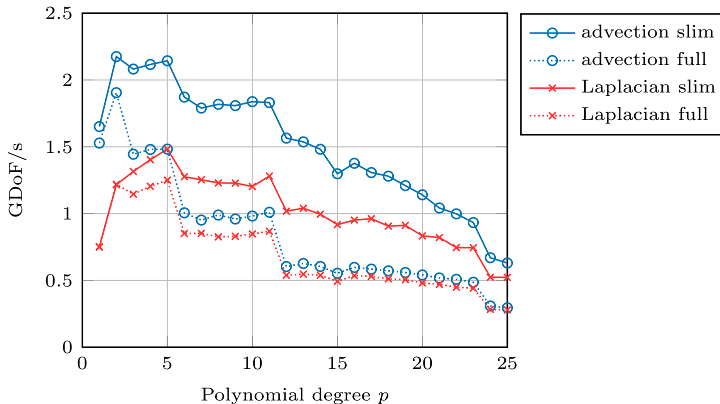

Figure 13: Compact face integrals: Influence of the MPI communication pattern (full cells vs. slim) on throughput of cell and face integrals as a function of the polynomial degree on 28 Broadwell cores.

which is the most common case, we again reduce the data sent.

Since the number of cells is balanced and each processor gets half the faces with each of its neighbors, the faces are also globally split evenly. This algorithm involves up to twice the number of messages than what would be theoretically necessary. However, it is easier and cheaper to set up than balancing faces more globally. We observed experimentally that in all our setups, the number of locally computed faces does not differ by more than up to a few dozens of faces among all MPI ranks, more or less independent of the number of cells in the mesh (which is typically a few thousands or more).

4.4 Minimal ghost data exchange

Following the layout described in [41], the ghost exchange is done by dedicated vector types that provide additional space beyond the locally owned range to fit the ghost data. Access to ghosted vector entries occurs with the same index space as the owned data but a larger index value. Outside the integration loops, such as in time steppers or linear solvers, only the vector data of the locally owned range (one-to-one map over processors) is exposed and ghost data is ignored. Avoiding a deep copy of the full vector into a ghosted one is beneficial because we aim for performance close to the limits of vector access, rendering a copy of the whole vector together with ghosts inefficient.

Fig. 13 lists the throughput of the matrix-vector product including the MPI exchange with the compact face algorithm from Algorithm 1. To ensure comparable vector sizes, we vary the mesh with the polynomial degree according to Table 6. Fig. 13 reveals sudden drops in performance at those polynomial degrees where the mesh size is reduced. This decreases the volume-to-surface ratio for the next smaller mesh level and a larger proportion of cells must be exchanged. For $p \leq 2$, around 4% of cells have remote neighbors, whereas the number is more than 80% at $p \geq 24$. The default MPI ghost exchange is labeled "full" in the figure and exchanges all degrees of freedom on each remote cell. For $6 \leq p \leq 11$, the ghost exchange consumes more than 40% of the operator evaluation time for the Laplacian in 3D. We have verified that this part indeed runs at full memory speed of 110 GB/s on Broadwell, so the issue is a memory bandwidth issue due to an excessive number of ghost entries.

The data exchange for face integrals can be reduced by observing that the face-normal interpolation will only touch some of the degrees of freedom of a cell for certain bases and derivatives,

namely those where S_f and possibly D_f of Eq. (9) are non-zero. For the example of advection on the configuration of Fig. 1, only 6 out of 36 DoF are needed for evaluating integrals with function values on faces. For a Hermite-like basis, two layers of $2k^{d-1}$ points are necessary as illustrated in Fig. 2. We propose to pack/unpack and communicate only those vector entries where the neighbor's S_f or D_f matrix entries of the respective cell are nonzero in Algorithm 4. The algorithm keeps the integration and vector access routines agnostic of this fact: we provide storage for all the degrees of freedom of a cell but only populate some of the entries with data. Copying the slim indices adds another indirection to extract the DoF on the cell's surface in the "unpack" stage of `update_ghost_values` and in the "pack" stage of `compress`, respectively. The code pattern is analogous to the selection of some entries among the locally owned DoF, so existing implementations can be readily extended. Note that Algorithm 4 cannot be directly implemented via general-purpose vector classes with MPI facilities such as PETSc [5] or Trilinos' Epetra and Tpetra [19, 20] without deep vector copies, which is why we use our own specialized implementation inside deal.II that is interlinked with the needs of the integrators.

ALGORITHM 4: Slim MPI communication

- Required input:
  - Type of basis: nodal at boundary, Hermite-type, generic.
  - Which terms are required for inner face integrals (only cell terms, values on face, values and first derivatives on face)?
- `update_ghost_values` fills the ghosted data, `compress` accumulates integral contributions to the respective entries at the owner. The communication is established according to the following three options:
  - If no face integrals, do nothing in DG (or cell-only ghost exchange for continuous elements).
  - If only face values and basis nodal at boundary, send only one single layer of k^{d-1} vector entries per cell at the interface.
  - If only up to first derivatives on face and Hermite-type basis, send only the two layers representing values and normal derivatives with $2k^{d-1}$ entries per cell at the interface.

Fig. 13 shows that the slim MPI communication of Algorithm 4 improves throughput by more than 30% for the Laplacian with $p = 6$ and by almost 80% for the advection operator with $p = 6$, both involving 8.9 million unknowns. The drops in throughput when going to smaller meshes with larger surfaces at processor boundaries are still visible for the slim implementation but less pronounced.

The cost of MPI communication is explicitly listed in a breakdown of times of the various algorithmic components in the DG operator evaluation in Fig. 14. The small experiment uses one mesh level less than what is reported in Table 6 (i.e., one eighth the number of cells), whereas the large experiments uses 16 times as many cells. Timings are based on the RDTSC timer register of Intel processors placed in inner loops. To show uncertainties and influence of timers on out-of-order execution that are more than 10% at $p < 4$, we also include reference runs with disabled timers, displayed by diamonds. The results show that MPI communication (`memcpy` and pack/unpack) is responsible for around a third of run time for $3 \leq p \leq 11$ for problem sizes between 1 and 7.1 million DoF. Sum-factorization sweeps on cells of complexity $\mathcal{O}(k)$ per DoF get more expensive for higher degrees according to Fig. 10.

Since the above experiments are run in shared memory, communication is not overlapped with computation, as opposed to communication over an Infiniband-like fabric that indeed is overlapped with our implementation.

4.5 Performance on Broadwell and Knights Landing

Fig. 15 shows our main result, the performance with compact face integrals and element-wise face integrals for the DG discretization of the advection equation (1) and the DG discretization of the Laplacian (2), including the full code with MPI data exchange. Results are recorded with the algorithmic setup analyzed in Sec. 3.5 for an affine-mesh case according to Table 6. The test is

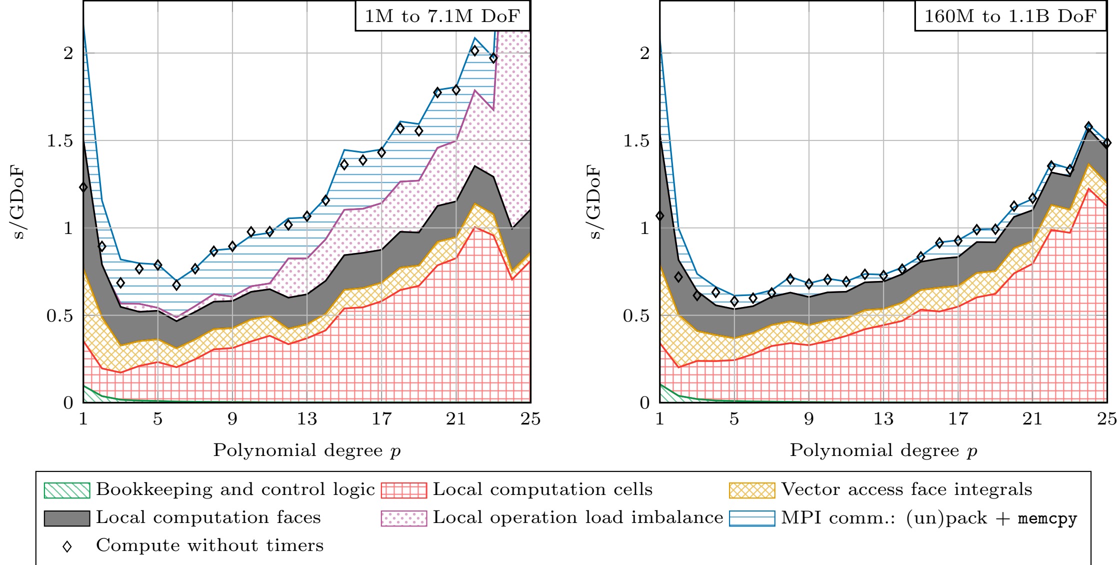

Figure 14: Breakdown of computation times into different phases for the 3D Laplacian for a small sized experiment at 1–7.1 million DoF and a large experiment at 128 million to 900 million DoF. The timings have been recorded for compact face integrals and with slim MPI communication on 28 Broadwell cores and rescaled to the time for a fixed fictive size of 1 billion (10^9) degrees of freedom.

repeated 500 times and the average is reported. As in the other tests, standard deviations between different runs are below 5%. We compare the actually achieved performance (lines with marks) to the in-cache performance limit established in Fig. 10 and the memory bandwidth limit. The 3D MPI test cases on Broadwell are run with 28 ranks, i.e., without hyperthreading, whereas all 2D cases as well as the OpenMP cases are run with hyperthreading, which represents the better performing option in either case for all $p \leq 15$. On KNL, 2-way hyperthreading is fastest in both cases for $p \leq 10$ and used throughout.

The results in Fig. 15 show that the element-wise face integrals of Algorithm 2 deliver consistently higher performance than the compact face integrals of Algorithm 1. This result suggests that the explicit collection of integral values within an element with a single write to the global data structure is superior to synchronizing over the result vector (where caches need to make sure that data is properly handled). In an analysis using the likwid tool for hardware performance counters [59], we observed that the compact face integrals access more vector data not only in the ghost transfer (where one additional write/read-for-ownership transfer is necessary for the final addition in compress) but also during operator evaluation. This is because caching is needed for both the input and the output vector, and face integrals can be delayed compared to cells due to the requirement that cell integrals set the vector value and face integrals add into it, whereas only the input vector data is accessed several times by the element-wise face integrals. The performance counter data for 28 MPI ranks shows that cache misses result in reading data worth 3.0 vectors for $p = 5$ in 3D for element-wise face integrals (including read-for-ownership) but reading worth 3.2 vectors and writing worth 1.5 vectors for compact face integrals. The imperfect caching of neighboring data is well-studied in finite differences [17]. It is related to the order the cells are passed through, which is the Morton order for the present experiments according to Table 6.

The gap between the in-cache performance and the actual result is due to three main factors. The first factor is the general-purpose gather access pattern of face integrals. In element-wise face integrals, indirect addressing occurs for reads to exterior values u^+ only and reduces throughput by about 5–10%, e.g. from 650 GFlop/s to 580 GFlop/s on an in-cache case with $p = 8$ on Broadwell. Furthermore, not all vector access of face integrals hits caches, as detailed below, which lowers performance by another 5–10%. For compact face integrals, both reads and writes are subject to indirect addressing and cache misses, such that the performance drop is more significant, losing up to 25% of the throughput recorded in Fig. 10. Secondly, the MPI data exchange consumes between

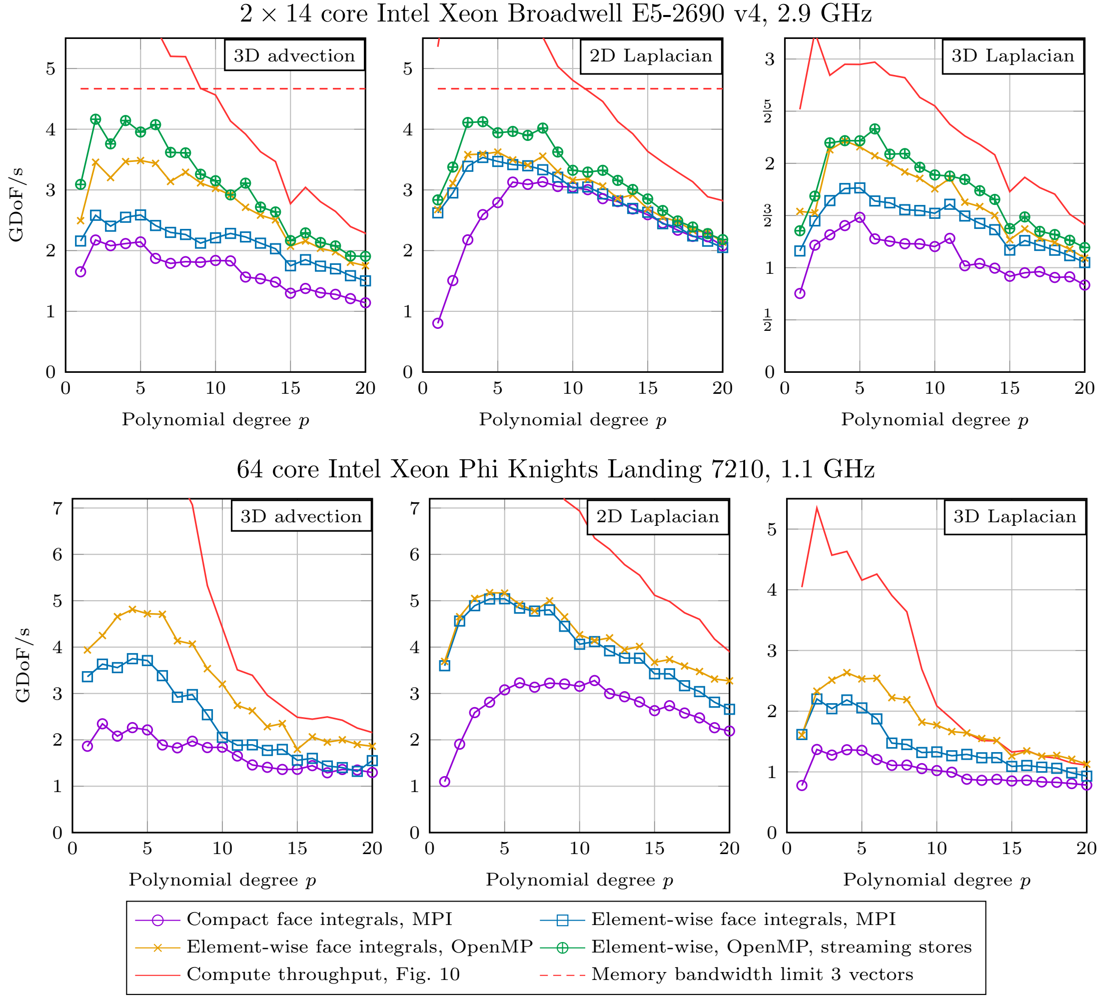

Figure 15: Comparison of the throughput on the 2D and 3D advection operator and Laplacian on Broadwell (top row) and KNL (bottom row). The measurements are compared to the theoretical throughput of the kernels from Fig. 10 (solid lines) and the memory bandwidth of 112 GB/s on Broadwell (dashed lines) and 450 GB/s on KNL (outside shown range) for the idealized setting of two vector reads and one vector write.

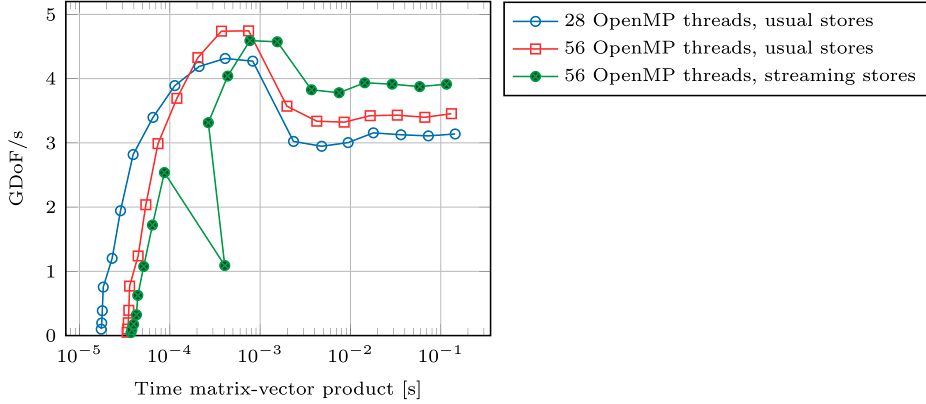

Figure 16: Throughput of matrix-vector product over its duration for $p = 5$ with OpenMP parallelization on 28 core Broadwell for 3D advection.

2–5% of operator evaluation time in 2D and more than 30% on KNL in 3D. The cost of the MPI exchange is clearly visible by comparing the results of the element-wise face integrals with MPI parallelization versus the OpenMP version, see also the cost of MPI in Fig. 14. The third factor are main memory reads and writes that can slow down the cores below the pure roofline limits for cases close to the memory bandwidth and arithmetic limits as has been observed in Sec. 4.1. For element-wise face integrals with OpenMP parallelization on Broadwell, Fig. 15 also displays the throughput for a variant where the usual SIMD store to the result vector is replaced by a streaming store operation, which avoids the read-for-ownership transfer and reduces the ideal data transfer to 8 Bytes read and 8 Bytes written. For the Laplacian, throughput increases by around 10% even though the memory bus runs at no more than 70 GB/s, and we record up to 480 GFlop/s for advection and 520 GFlop/s for the 3D Laplacian. On KNL with up to 550 GFlop/s, no significant performance gain is observed by streaming store, indicating that memory bandwidth is sufficient. Nonetheless, the performance loss is on KNL more severe, which we identified to be more stalls due to missing prefetching.

Fig. 16 plots the throughput of the matrix-vector product over its wall time for the 3D advection case with $p = 5$, contrasting the throughput on the y-axis over latency barriers on the x-axis. The problem size varies from 8 cells with 1,700 unknowns to 453 million unknowns. Up to 55,000 unkowns (256 cells or 64 cell batches with 4-wide SIMD), there is not enough parallelism to saturate the system. Once hyperthreading is enabled, twice the parallelism is needed and the overall latency is 1.8 times higher. The throughput reaches a peak at slightly less than 10^{-3} seconds (corresponding to 480 GFlop/s) for a size of 3.5 million unknowns where both vectors fit into the 70 MB of L3 cache in the case of usual stores or up to 7.1 million unknowns for streaming stores (input vector cached, output written to main memory).

For large problem sizes where all vector data must be read from main memory, streaming stores permit a higher performance, but the effect is reversed if data would fit into caches. Regarding throughput, hyperthreading improves performance by around 10% (and similarly for the Laplacian). Together with performance counters which report memory stalls due to wait for L2/L3 caches or memory, the performance increases by the added memory-level parallelism of hyperthreading suggests that prefetching is not perfect and latency issues in the processor's execution pipeline can be hidden. A more accurate modeling and prediction of these effects is beyond the scope of the present work.

The algorithms are equally beneficial for large-scale parallel simulations as shown in Table 7 for a pure MPI parallelization. We record almost ideal strong scaling until around 0.5 ms. The main reason for the loss in efficiency is the larger proportion of pack/unpack and `memcpy` routines in the communication reported in Fig. 14.

Table 7: Strong scaling experiment of Laplacian on 3D affine mesh with 262,144 cells and $p = 5$ (56.6 million degrees of freedom) on Xeon E5-2697 v3 (2×14 cores operating at 2.1 GHz) based cluster on up to 512 nodes with 14,336 cores. Numbers are reported as the absolute run time in milliseconds [ms] of a matrix-vector product including communication and in terms of throughput measured as billion degrees of freedom per second and node (GDoF/s/node) reporting the parallel efficiency.

# nodes	1	2	4	8	16	32	64	128	256	512
double precision, compact face integrals										
time [ms]	53.9	28.0	14.9	7.88	3.97	2.01	1.10	0.646	0.418	0.261
GDoF/s/node	1.05	1.01	0.947	0.898	0.892	0.880	0.806	0.685	0.529	0.424
double precision, element-wise face integrals										
time [ms]	49.7	25.3	12.9	6.80	3.42	1.72	0.892	0.477	0.273	0.182
GDoF/s/node	1.14	1.12	1.10	1.04	1.04	1.03	0.992	0.928	0.808	0.609
single precision, element-wise face integrals										
time [ms]	26.3	13.3	6.86	3.43	1.73	0.909	0.492	0.421	0.213	0.110
GDoF/s/node	2.16	2.13	2.06	2.06	2.04	1.95	1.80	1.05	1.04	1.01

4.6 Comparison to global trace storage

A common implementation strategy in DG codes with minimal data exchange, not linked to a particular finite element basis as the one above, is to use compact face integrals in conjunction with performing face-normal interpolation D_f and S_f (or numerical traces) into a separate global storage, see also the algorithm layout described extensively in [21] and related to the flux memory layout of [31]. This involves an initial loop over the cells where the face data is collected (and cell integrals are computed), a loop over the faces which reference only to the separate trace storage for computing integrals within the faces, and a final loop over the cells that collects the face integrals and associates them with shape functions on the cells. Such a strategy is also easily run in multithreaded mode without race conditions [42]. A disadvantage of this scheme is an around $4\times$ higher data transfer on affine meshes since the result vector is accessed twice and the separate global trace storage is involved. An example for the transfer is given in Table 1 of [39]. Even though it would be conceivable to keep the data storage lower with dynamic dependency-based task scheduling, the authors' experience from [37, 42] suggests that available implementations such as Intel Threading Building Blocks [53] or OpenMP tasks do typically lead to significant memory access from remote NUMA domains and other cache or prefetcher inefficiencies that lower application performance once using 10 or more cores.

Table 8 compares the throughput of the proposed algorithms for the 3D Laplacian on a Hermite-like basis in an MPI-only experiment with a global trace storage as used e.g. in [22, 42]. For global trace storage a basis with collocation of nodal points and quadrature points is used because it can skip the `basis_change` algorithm in cell integrals. The results highlight that the proposed method with a single loop for cell and face integrals is almost twice as fast for degree $p = 5$ and still 25% faster for $p = 11$. The global trace storage only becomes superior at $p > 15$ where the advantages of collocated node and quadrature points in cell integrals get significant. Note that the global trace storage scheme runs at full memory throughput with $> 90\,\text{GB/s}$ for all degrees $p \leq 11$, whereas the proposed scheme does not utilize the full RAM bandwidth, despite being substantially faster.

5 Representation of geometry

In Algorithm 1, we assumed the final geometric factor involving the inverse Jacobian $\mathcal{J}_{(e)}$ of the transformation from unit to real cell to be given. In a generic computation of integrals, the inverse Jacobian as well as the determinant of the Jacobian and normal vectors $\mathbf{n}$ derived from the Jacobian need to be specifically supplied. The Jacobian is often defined as the derivative of a piecewise m-th degree polynomial description of the geometry through some mapping nodes $\mathbf{x}_\text{msp}^{(i)}$, $i = 1, \ldots, n_\text{points}$, related to evaluation of the geometry on the Gauss–Lobatto nodes of degree m, but it can also be defined by analytical tangent vectors on the geometry. Extending over a short discussion in [41], a high-performance implementation for arbitrary geometries can select between

Table 8: Throughput and measured memory throughput for evaluation of the 3D Laplacian on 2×14 Broadwell cores on a Hermite-like basis with compact and element-wise face integrals against scheme with global trace storage with collocated nodal and quadrature points minimizing arithmetic operations.

polynomial degree p	2	3	5	8	11	15	20
	billion degrees of freedom per second, GDoF/s						
element-wise face integrals	1.45	1.65	1.76	1.56	1.61	1.17	1.05
compact face integrals	1.22	1.32	1.48	1.23	1.28	0.918	0.833
global trace storage	0.551	0.646	0.897	1.01	1.07	0.901	0.902
	measured memory throughput, GB/s						
element-wise face integrals	51.5	54.4	56.7	58.3	60.8	55.3	53.7
compact face integrals	58.4	61.3	63.8	62.5	58.6	53.1	52.9
global trace storage	106	106	107	102	90.7	75.1	72.6

at least four main variants:

(G1) Storage of mesh nodes $\mathbf{x}_{\text{msp}}^{(i)}$ for all indices i in the mesh with usual indirect addressing of continuous finite elements and subsequent evaluation with sum factorization. This involves transfer of $\frac{k^d}{(k+1)^d} d$ *double precision values* per quadrature point for isoparametric mappings.

(G2) Storage of all quadrature points $\mathbf{x}_{\text{qp}}^{(q)}$ in physical space, from which the Jacobian can be computed by a collocation derivative. This needs d *double precision values* per quadrature point in 3D. For face integrals, separate data is needed except for Gauss–Lobatto like integration rules.

(G3) Pre-computation of $\mathcal{J}_{(e)}$ in all quadrature points of the mesh; for face integrals, separate data is needed. Storing the inverse Jacobian and the determinant of the Jacobian involves transfer of *10 double precision values* per quadrature point in 3D.

(G4) Pre-computation of the effective coefficient in the particular equation at hand, e.g. a symmetric $d \times d$ tensor $\mathcal{J}_{(e)}^{-1} \mathcal{J}_{(e)}^{-\mathrm{T}} \det(\mathcal{J}_{(e)})$ for the cell term of the Laplacian (*6 double precision values* in 3D) or the vector $\mathcal{J}_{(e)}^{-1} \mathbf{c}(\mathbf{x}) \det(\mathcal{J}_{(e)})$ for the advection equation (*3 double precision values* in 3D), a technique used e.g. by Nek5000 [16].

Furthermore, coefficients such as $\mathbf{c}\left(\hat{\mathbf{x}}^{(e)}(\boldsymbol{\xi})\right)$ in Algorithm (1) can be pre-computed and loaded during the operator evaluation or computed on the fly based on the location of the quadrature point $\mathbf{x}_{\text{qp}}^{(q)}$. The pre-computed variants can also be combined with simple memory compressions, such as the constant-Jacobian case on affine meshes [41] or constant-in-one-direction case on extruded meshes. The results in the previous section, using variant (G4) with constant data in all points, have shown that the operator evaluation is mostly core-limited on affine meshes, in particular on the Haswell and KNL systems. Since the first two options (G1) and (G2) involve additional computations, it is not clear a priori whether the variable-coefficient case runs faster with tabulation involving higher memory transfer or more computations. Note that [54] concluded that pre-computed variants on a GPU are superior over computation on the fly, which is related to the more limited cache sizes and a GPU's memory hierarchy.

The variants (G3) and (G4) result in different code layouts, respectively: (G3) allows for the definition of arbitrary weak forms and integration of nonlinear terms by separate control over the operators on trial functions and test functions. The variant (G4) hardcodes the differential operator in the coefficient, which is more efficient for certain operators like the Laplacian or advection, and involves reference-cell quantities otherwise. When a highly tuned operator evaluation according to the concepts presented above is used in a generic software package with a user-defined quadrature operation, our experience from the deal.II package is that core-limited patterns including table lookups, branches, or computations are more common. Thus, variant (G3) with pre-computed geometric quantities such as normal vectors, Jacobians, and Jacobian is often preferable over (G2) and particularly (G1). An optimizing library can provide implementations of the various variants (G1)–(G4) and let the user code switch to the desired option via run-time parameters or even

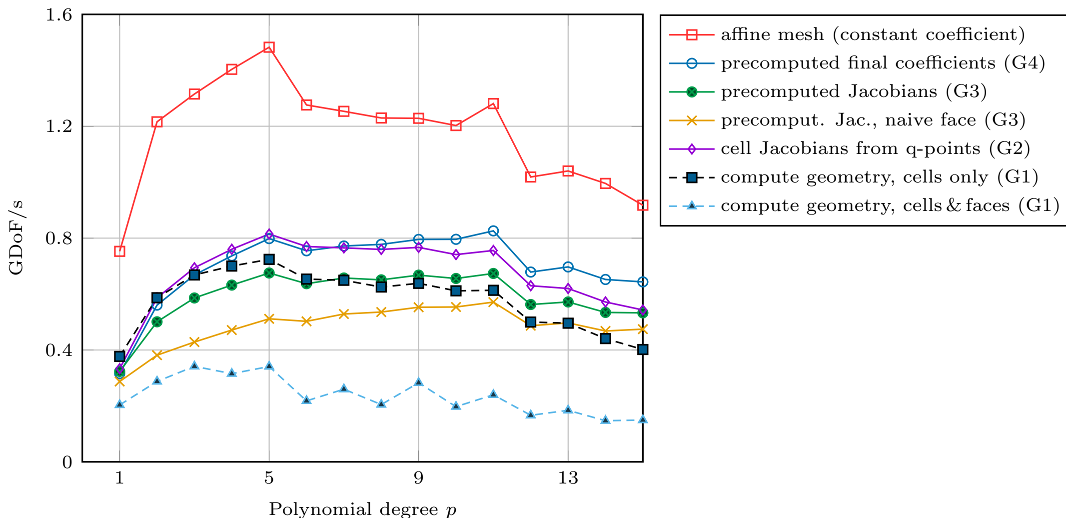

Figure 17: Throughput of evaluation of 3D Laplacian with compact face integrals using MPI for various ways to represent a curved geometry on 28 Broadwell cores.

determine the most suitable one by auto-tuning. Repetitive implementations of variants (G3) and (G4) could be avoided by a domain-specific compiler like is used in Firedrake [24], for instance. Note that access to large geometry arrays is often prefetcher-friendly, in particular on sophisticated CPUs such as Haswell and Broadwell.

Fig. 17 gives an example of the tradeoffs in the geometry evaluation, using the 3D Laplacian with compact face integrals on a deformed geometry using an iso-parametric polynomial description of degree p. The mesh topology and problem sizes are the same as in Table 6 with 8 million to 57 million unknowns. The pre-computed geometry options show the most consistent performance, despite the highest memory transfer as an inverse Jacobian and the Jacobian determinant (G3) or final coefficients of (G4) must be loaded for each quadrature point. Compared to the affine mesh, throughput is significantly lower, as the additional memory access reduces the arithmetic intensity from around 8 Flop/Byte in the affine mesh case to between 1 and 2 Flop/Byte, see Fig. 18. A distinctive trend is that the memory bandwidth limit renders throughput measured as the number of degrees of freedom processed per second almost constant for a wide range of polynomial degrees, $3 \leq p \leq 11$, since the data accessed per degree of freedom is $\mathcal{O}(1)$.

When computing the geometric factor $\mathcal{J}_{(e)}$ of both cells and faces from a p-degree continuous finite element representation including indirect addressing, marked "compute geometry, cells & faces (G1)" in Fig. 17, performance does not exceed 350 million degrees of freedom per second. Since face integrals only access k^{d-1} out of the k^d elemental values for the geometry, it is preferable to at least precompute the Jacobian data on the faces as in "compute geometry, cells only (G1)". For the test "precomputed Jacobian, naive face (G3)", we first compute the full gradient $\mathcal{J}_{(e)}^{-\mathrm{T}} \nabla_{\boldsymbol{\xi}} u$ and then multiply by $\mathbf{n}$. However, access can be simplified by directly computing the normal derivative $\mathbf{n} \cdot \nabla_{\mathbf{x}} u$ through a geometric quantity $\mathbf{j_n} = \mathbf{n} \cdot \mathcal{J}_{(e)}^{-\mathrm{T}}$. This simplification, proposed in Design Choice 3, is used for all the cases "precomputed final coefficients (G4)", "precomputed Jacobians (G3)", "cell Jacobians from q-points (G2)" as well as "compute geometry, cells only (G1)".

Design Choice 3 *For discretizations of operators that involve the inverse Jacobian* $\mathcal{J}_{(e)}^{-\mathrm{T}}$ *and normal vector* $\mathbf{n}$ *together, such as the symmetric interior penalty discretization of the Laplacian* (2)*, a pre-computed vector* $\mathbf{j_n} = \mathbf{n} \cdot \mathcal{J}_{(e)}^{-\mathrm{T}}$ *is preferable over the two separate factors. This way, only* $2d$ *values per quadrature point need to be accessed, rather than* $2d^2$ *for the inverse Jacobians on both sides and further* d *values for the normal vector.*

With respect to the geometry on cells, we see that the computation from a continuous finite element field (G1) that includes indirect access is only competitive for low degrees where the reduced memory access pays off. For $p > 5$, some pre-computation is preferable. The computation

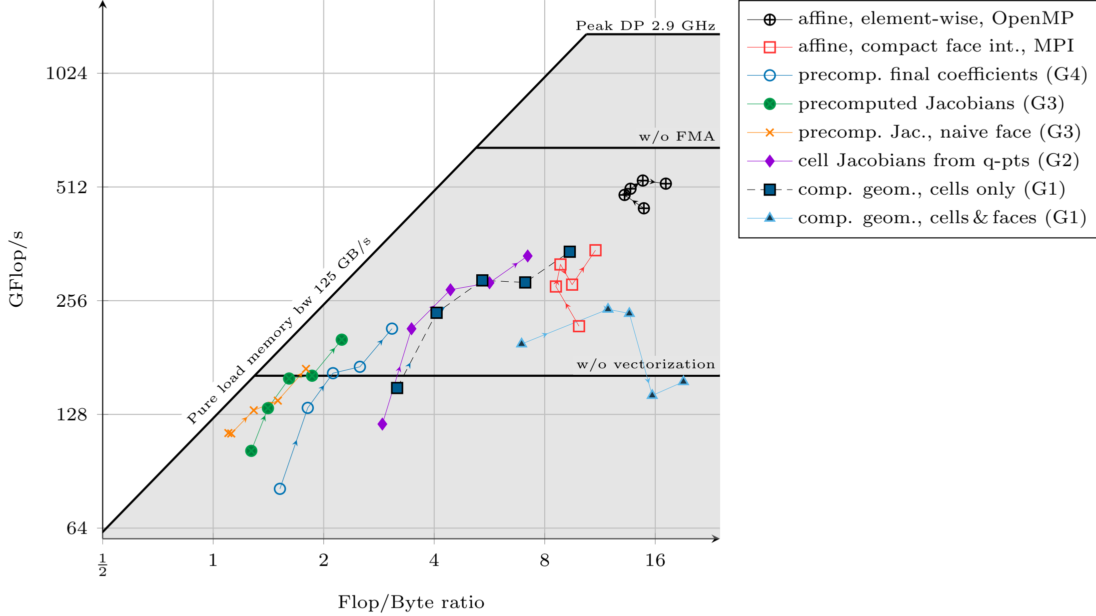

Figure 18: Roofline model for the evaluation of the 3D Laplacian with different geometry variants on 2×14 Broadwell cores, displaying the data for $p = 1, 3, 5, 8, 11$ for five cases from Fig. 17 regarding the geometry representation as well as the element-wise face integrals with streaming stores and OpenMP parallelization, the fastest option identified in Sec. 4. The arithmetic balance is based on theoretical (best-case) memory throughput computed from the vector and geometry access, assuming perfect caching, and GFlop/s rates according to the operation counts derived in this work and similar numbers for the geometry. Arrows indicate increasing polynomial degrees and higher degrees tend to be further to the right.

of cell Jacobians from the positions of the quadrature points that only involves a collocation derivative operation on all d components, i.e., d^2 sum-factorization sweeps, is a very attractive option and even outperforms the precomputed final coefficients option for low degrees. However, for large degrees $p > 10$ the arithmetic intensity of the cell work increases and the temporary arrays for the geometry's sum-factorization sweeps spill to outer level caches and eventually to main memory, making precomputed variants preferable.

The fact that the best performance is observed for the case where the geometry is precomputed, leading to higher memory transfer, is explained by the fact that the affine-mesh case is core-limited. As a consequence, doubling or tripling computations and the associate in-cache data access for the variants (G1) or (G2) directly increases pressure on the critical resource. To give a point of reference, evaluating the 3D Laplacian for $p = 5$ with precomputed coefficients (G4) involves 212 Flop/DoF according to Table 4, the variant "cell Jacobians from q-points (G2)" 337 Flop/DoF, and "compute geometry, cells only (G1)" 401 Flop/DoF. The memory access, on the other hand, can partly be absorbed behind computations. Even though the cost of memory access is high, the loss in performance is less than due to additional computations with (G1) or (G2). This choice obviously depends on the hardware and the Flop/Byte machine balance.

Fig. 18 displays the performance of operator evaluation in terms of the roofline performance model [61]. The results are based on best-case theoretical computations of the memory access, assuming perfect caching. As has been analyzed in Sec. 4, considerably more data is loaded from the solution vectors than this ideal value. For $p = 1$, the data transfer of metadata is also non-negligible. When looking at the actual memory access with the likwid tool, the pre-computed versions are essentially at the limit of the memory bandwidth. For computing the Jacobian from the quadrature point locations, a considerably higher arithmetic intensity with some components dominated by memory transfer and others being core-bound is observed. The slight deviation from the ideal memory transfer can be explained by the different performance boundaries in the high polynomial degree case: This variant has a lower arithmetic performance limit due to a

lower density of FMA instructions (inverting the Jacobian involves about three times as many multiplications as FMAs). Furthermore, it is heavier on the L2 and L3 cache access at $p = 8$ and $p = 11$. The evaluation of the geometry for cells and faces shows a very low performance despite a high arithmetic intensity. This is mainly because of the very intensive indirect addressing into the geometry arrays on faces with gather instructions and additionally relatively poor caching for high degrees $p = 8$ and $p = 11$.

6 Conclusions and future developments

We have presented a detailed performance analysis of matrix-free operator evaluation for discontinuous Galerkin methods with cell and face integrals. The methods are specialized for quadrilateral and hexahedral meshes and use sum-factorization techniques for computing the integrals by quadrature. This work has highlighted algorithmic choices to reach high performance and a set of tests to verify the performance of an implementation.

Firstly, we have considered the in-cache case of sum-factorization sweeps and quadrature point operations. We have presented an approach which minimizes the number of arithmetic operations by a basis change and collocation derivative together with an even-odd decomposition that utilizes the symmetry in shape functions and quadrature points. Our experiments have shown that up to 60% of arithmetic peak on Intel Haswell and Broadwell processors as well an Intel Knights Landing manycore processor can be reached.

In a second set of experiments, we have included the access to the input and output vectors. When cell integrals are run in isolation, the computations become memory bandwidth bound, especially for advection and two-dimensional problems with low and moderate polynomial degrees. Thus, our experiments suggest that cell and face integrals must be interleaved for reaching optimal performance. Guided by a roofline performance model, this setup has been shown to be limited by the in-core performance on simple geometries, but it is memory-bound in case a variable geometry with different factors must be loaded at each quadrature point. Two layouts for implementing face integrals have been considered, one computing all face integrals associated to a cell that visits each interior face twice (Algorithm 2), and a second one which computes all integrals to a face at once (Algorithm 1). Even though the latter would seem superior for the core-limited case due to less arithmetic operations, better performance has been recorded for the former due to simpler, finite-difference like data access where the full result of a cell is computed and written to memory in a single sweep and the neighbor's cells are only read.

On affine geometries and with 28 Intel Broadwell cores, we measured a throughput of the matrix-vector product of up to 4 billion unknowns per second for the 3D advection and up to 2.2 billion unknowns per second for the 3D Laplacian. The time to perform the matrix-vector product comes close to the time it takes to copy the source to the destination vector at 4.7–6.9 billion unknowns per second. This shows that the DG operator evaluation can almost be hidden behind the unavoidable memory transfer on processors like Broadwell which have a high machine balance in terms of Flop/Byte. The fast matrix-vector products are especially beneficial for iterative solvers and preconditioners that spend the bulk of their time in matrix-vector products or residual evaluations and apply equally well to explicit time integration as to linear and nonlinear systems. However, the high throughput has implications for future algorithm design: the matrix-vector product alone might no longer be the dominant part of algorithms and be outweighted by other seemingly cheaper operations, such as vector updates or inner products in iterative solvers, see e.g. [44, 14].

When it comes to the MPI parallelization, our experiments have identified the MPI data exchange operations to take up to a third of the operator evaluation time on a single node, even after optimizing the MPI data transfer for special polynomial bases according to Algorithm 4. This highlights the importance to consider shared-memory parallelization or MPI shared memory schemes according to the MPI-3 standard. Furthermore, these alternative parallelization concepts reduce the amount of duplicated data in general, promising better use of many-core architectures that have less memory per core available than today's multi-core processors.

References

[1] D. S. Abdi, L. C. Wilcox, T. C. Warburton, and F. X. Giraldo. A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model. *The International Journal of High Performance Computing Applications*, 33(1):81–109, 2019.

[2] R. Agelek, M. Anderson, W. Bangerth, and W. L. Barth. On orienting edges of unstructured two- and three-dimensional meshes. *ACM Transactions on Mathematical Software*, 44:5:1–5:22, 2017.

[3] G. Alzetta, D. Arndt, W. Bangerth, V. Boddu, B. Brands, D. Davydov, R. Gassmoeller, T. Heister, L. Heltai, K. Kormann, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells. The `deal.II` library, version 9.0. *J. Numer. Math.*, 26(4):173–184, 2018.

[4] D. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM Journal on Numerical Analysis*, 39:1749–1779, 2002.

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016. http://www.mcs.anl.gov/petsc.

[6] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic finite element codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28, 2011.

[7] P. Bastian, C. Engwer, J. Fahlke, M. Geveler, D. Göddeke, O. Iliev, O. Ippisch, R. Milk, J. Mohring, S. Müthing, M. Ohlberger, D. Ribbrock, and S. Turek. Hardware-based efficiency advances in the EXA-DUNE project. In H.-J. Bungartz, P. Neumann, and W. E. Nagel, editors, *Software for Exascale Computing – SPPEXA 2013-2015*, pages 3–23. Springer, Cham, 2016.

[8] P. Bastian, C. Engwer, D. Göddeke, O. Iliev, O. Ippisch, M. Ohlberger, S. Turek, J. Fahlke, S. Kaulmann, S. Müthing, and D. Ribbrock. EXA-DUNE: Flexible PDE solvers, numerical methods and applications. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 530–541. Springer, 2014.

[9] J. Brown. Efficient nonlinear solvers for nodal high-order finite elements in 3D. *J. Sci. Comput.*, 45(1-3):48–63, 2010.

[10] C. D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. M. Kirby, and S. J. Sherwin. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[11] L. E. Carr III, C. F. Borges, and F. X. Giraldo. Matrix-free polynomial-based nonlinear least squares optimizated preconditioning and its applications to discontinuous Galerkin discretizations of the Euler equations. *Journal of Scientific Computing*, 66:917–940, 2016.

[12] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-order methods for incompressible fluid flow*, volume 9. Cambridge University Press, 2002.

[13] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. Higham, J. Hogg, P. V. Lara, M. Zounon, S. Relton, and S. Tomov. A proposed API for batched basic linear algebra subprograms. Technical report, University of Tennessee, 2016. https://bit.ly/batched-blas.

[14] N. Fehn, W. A. Wall, and M. Kronbichler. Efficiency of high-performance discontinuous Galerkin spectral element methods for under-resolved turbulent incompressible flows. *International Journal for Numerical Methods in Fluids*, 88(1):32–54, 2018.

[15] N. Fehn, W. A. Wall, and M. Kronbichler. A matrix-free high-order discontinuous Galerkin compressible Navier–Stokes solver: A performance comparison of compressible and incompressible formulations for turbulent incompressible flows. *International Journal for Numerical Methods in Fluids*, 89(3):71–102, 2019.

[16] P. F. Fischer, S. Kerkemeier, et al. Nek5000 Web page, 2018. https://nek5000.mcs.anl.gov.

[17] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Boca Raton, 2011.

[18] A. Heinecke, G. Henry, and H. Pabst. LIBXSMM: A high performance library for small matrix multiplications, 2017. https://github.com/hfp/libxsmm.

[19] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, W. A., and K. S. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, 2005.

[20] M. A. Heroux et al. Trilinos Web page, 2018. http://www.trilinos.org.

[21] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Application*, volume 54 of *Texts in Applied Mathematics*. Springer, 2008.

[22] F. Hindenlang, G. Gassner, C. Altmann, A. Beck, M. Staudenmaier, and C.-D. Munz. Explicit discontinuous Galerkin methods for unsteady problems. *Computers & Fluids*, 61:86–93, 2012.

[23] T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems. In *SC15*, Nov. 2015.

[24] M. Homolya, R. C. Kirby, and D. A. Ham. Exposing and exploiting structure: optimal code generation for high-order finite element methods. *arXiv preprint*, 1711.02473:cs.MS, 2017.

[25] I. Huismann, J. Stiller, and J. Fröhlich. Factorizing the factorization – a spectral-element solver for elliptic equations with linear operation count. *Journal of Computational Physics*, 346:437–448, 2017.

[26] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, July 2017. Order no. 248966-037, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[27] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming, Knights Landing edition*. Morgan Kaufmann, Cambridge, MA, 2016.

[28] G. E. Karniadakis and S. J. Sherwin. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2nd edition, 2005.

[29] D. Kempf, R. Hess, S. Müthing, and P. Bastian. Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures. *arXiv preprint*, 1812.08075:math.NA, 2018.

[30] A. Klöckner. Loo.py: transformation-based code generation for GPUs and CPUs. In *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*, Edinburgh, Scotland., 2014. Association for Computing Machinery.

[31] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

[32] M. G. Knepley, J. Brown, K. Rupp, and B. F. Smith. Achieving high performance with unified residual evaluation. *arXiv preprint*, 1309.1204:cs.MS, 2013.

[33] T. Kolev et al. MFEM: Modular finite element methods, 2018. mfem.org.

[34] D. Komatitsch et al. SPECFEM 3D cartesian user manual. Technical report, Computational Infrastructure for Geodynamics, Princeton University, CNRS and University of Marseille, and ETH Zürich, 2015.

[35] D. Kopriva. *Implementing spectral methods for partial differential equations*. Springer, Berlin, 2009.

[36] K. Kormann. A time-space adaptive method for the Schrödinger equation. *Communications in Computational Physics*, 20(1):60–85, 2016.

[37] K. Kormann and M. Kronbichler. Parallel finite element operator application: Graph partitioning and coloring. In *Proceedings of the 7th IEEE International Conference on eScience*, pages 332–339, 2011.

[38] B. Krank, N. Fehn, W. A. Wall, and M. Kronbichler. A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *Journal of Computational Physics*, 348:634–659, 2017.

[39] M. Kronbichler and M. Allalen. Efficient high-order discontinuous Galerkin finite elements with matrix-free implementations. In H.-J. Bungartz, D. Kranzlmüller, V. Weinberg, J. Weismüller, and V. Wohlgemuth, editors, *Advances and Trends in Environmental Informatics*, pages 89–110, 2018.

[40] M. Kronbichler, A. Diagne, and H. Holmgren. A fast massively parallel two-phase flow solver for the simulation of microfluidic chips. *International Journal on High Performance Computing Applications*, 32(2):266–287, 2018.

[41] M. Kronbichler and K. Kormann. A generic interface for parallel cell-based finite element operator application. *Computers & Fluids*, 63:135–147, 2012.

[42] M. Kronbichler, K. Kormann, I. Pasichnyk, and M. Allalen. Fast matrix-free discontinuous Galerkin kernels on modern computer architectures. In J. M. Kunkel, R. Yokota, P. Balaji, and D. E. Keyes, editors, *ISC High Performance 2017, LNCS 10266*, pages 237–255, 2017.

[43] M. Kronbichler, S. Schoeder, C. Müller, and W. A. Wall. Comparison of implicit and explicit hybridizable discontinuous Galerkin methods for the acoustic wave equation. *International Journal for Numerical Methods in Engineering*, 106(9):712–739, 2016.

[44] M. Kronbichler and W. A. Wall. A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. *SIAM Journal on Scientific Computing*, 40(5):A3423–A3448, 2018.

[45] F. Luporini, D. A. Ham, and P. H. J. Kelly. An algorithm for the optimization of finite element integration loops. *ACM Transactions on Mathematical Software*, 44(1):3:1–3:26, 2017.

[46] D. A. May, J. Brown, and L. Le Pourhiet. pTatin3D: High-performance methods for long-term lithospheric dynamics. In J. M. Kunkel, T. Ludwig, and H. W. Meuer, editors, *Supercomputing (SC14)*, pages 1–11, New Orleans, 2014.

[47] A. T. T. Mcrae, G.-T. Bercea, L. Mitchell, D. A. Ham, and C. J. Cotter. Automated generation and symbolic manipulation of tensor product finite elements. *SIAM Journal on Scientific Computing*, 38(5):S25–S47, 2016.

[48] A. Modave, A. St-Cyr, and T. Warburton. GPU performance analysis of a nodal discontinuous Galerkin method for acoustic and elastic models. *Computers & Geosciences*, 91:64–76, 2016.

[49] S. Müthing, M. Piatkowski, and P. Bastian. High-performance implementation of matrix-free high-order discontinuous Galerkin methods. *arXiv preprint*, 1711.10885:math.NA, 2017.

[50] S. A. Orszag. Spectral methods for problems in complex geometries. *Journal of Computational Physics*, 37:70–92, 1980.

[51] A. T. Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468–488, 1984.

[52] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3):24:1–24:27, 2016.

[53] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[54] J.-F. Remacle, R. Gandham, and T. Warburton. GPU accelerated spectral finite elements on all-hex meshes. *Journal of Computational Physics*, 324:246–257, 2016.

[55] J. Schöberl. C++11 implementation of finite elements in NGSolve. Technical Report ASC Report No. 30/2014, Vienna University of Technology, 2014.

[56] S. Schoeder, K. Kormann, W. A. Wall, and M. Kronbichler. Efficient explicit time stepping of high order discontinuous Galerkin schemes for waves. *SIAM Journal on Scientific Computing*, 40(6):C803–C826, 2018.

[57] S. J. Sherwin and G. E. Karniadakis. Tetrahedral *hp* finite elements: Algorithms and flow simulations. *Journal of Computational Physics*, 124(1):14–45, 1996.

[58] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. J. Kelly. A study of vectorization for matrix-free finite element methods. *arXiv preprint*, 1903.08243:cs.MS, 2019.

[59] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010. `https://github.com/RRZE-HPC/likwid`, retrieved on October 15, 2018.

[60] Z. Wang, K. Fidkowski, R. Abgrall, F. Bassi, D. Caraeni, A. Cary, H. Deconinck, R. Hartmann, K. Hillewaert, H. Huynh, N. Kroll, G. May, P.-O. Persson, B. van Leer, and M. Visbal. High-order CFD methods: current status and perspective. *International Journal for Numerical Methods in Fluids*, 72(8):811–845, 2013.

[61] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.