Fast matrix-free discontinuous Galerkin kernels on modern computer architectures

Martin Kronbichler[1] Katharina Kormann[2,3]
Igor Pasichnyk[4] Momme Allalen[5]

[1]Institute for Computational Mechanics, Technical University of Munich, Boltzmannstr. 15, 85747 Garching, Germany
kronbichler@lnm.mw.tum.de
[2]Max–Planck–Institute for Plasma Physics, Boltzmannstr. 2, 85748 Garching, Germany
[3]Zentrum Mathematik, Technical University of Munich, Boltzmannstr. 3, 85747 Garching, Germany
[4]IBM Deutschland, Boltzmannstr. 1, 85748 Garching, Germany
[5]Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften, Boltzmannstr. 1, 85748 Garching, Germany

Abstract

This study compares the performance of high-order discontinuous Galerkin finite elements on modern hardware. The main computational kernel is the matrix-free evaluation of differential operators by sum factorization, exemplified on the symmetric interior penalty discretization of the Laplacian as a metric for a complex application code in fluid dynamics. State-of-the-art implementations of these kernels stress both arithmetics and memory transfer. The implementations of SIMD vectorization and shared-memory parallelization are detailed. Computational results are presented for dual-socket Intel Haswell CPUs at 28 cores, a 64-core Intel Knights Landing, and a 16-core IBM Power8 processor. Up to polynomial degree six, Knights Landing is approximately twice as fast as Haswell. Power8 performs similarly to Haswell, trading a higher frequency for narrower SIMD units. The performance comparison shows that simple ways to express parallelism through `for` loops perform better on medium and high core counts than a more elaborate task-based parallelization with dynamic scheduling according to dependency graphs, despite less memory transfer in the latter algorithm.

1 Introduction

The increasing accuracy requirements when simulating partial differential equations in engineering applications can often not be satisfied by simply scaling up existing codes. A limitation in the solver design of many codes is their heavy

use of sparse linear algebra routines, with matrices coming from some low-order discretization on unstructured meshes. Sparse matrix algebra is heavily memory bandwidth bound and has only seen moderate performance gains from the advances in computer architecture during the last decade. On systems with a limited amount of high-bandwidth memory, such as the 16 GB of MCDRAM on the Intel Knights Landing architecture, the sheer memory consumption of sparse matrices can further limit the applicability of legacy implementations.

In iterative linear solvers which are dominated by matrix-vector products, an alternative to matrix-based schemes is an evaluation on the fly without actually constructing the matrix. Stencil-based realizations such as finite differences often impose too strong restrictions on the computational mesh. On the other hand, the integrals underlying a matrix-vector product in a finite element discretization are amenable to fast matrix-free implementation by sum factorization [6] for meshes consisting of quadrilaterals or hexahedra. A generic sum factorization finite element kernel was introduced to the deal.II finite element library [1] in [11]. For polynomial degrees two and higher, it has been shown to be several times faster than matrix-based schemes.

For discretization of complex transport phenomena, higher-order discontinuous Galerkin (DG) methods are very attractive: As opposed to continuous finite elements, they do not strongly impose the continuity of the solution over element interfaces but rather link the elements by integrals involving numerical fluxes as a combination of the solution on both sides. This flexibility allows for taking directionality into account, such as upwinding [2], and makes the method robust also in complex flow scenarios. Furthermore, the independent definition of the solution on each element in DG avoids the indirect addressing inherent to the access of degrees of freedom in continuous spectral element methods [6] in favor of "packed" data access. Its combination of highly desirable characteristics makes DG an essential building block in next-generation solvers and motivates the development of efficient and tuned implementations. The present work has its background in a high-order discontinuous Galerkin solver for simulating incompressible turbulent flow described by the Navier–Stokes equations, where one of the central algorithmic components is a solver for the pressure Poisson equation which is implemented via the multigrid infrastructure described in [12].

For this purpose, we have extended the sum-factorization finite element framework presented in [11] to discontinuous elements with face integrals. In this work, we consider single node code optimizations. Two central aspects of high-performance DG implementations on modern compute architecture are addressed, namely efficient SIMD vectorization and shared-memory parallelization. These two components form the basis for hybrid codes that additionally use MPI to span over several nodes. Our shared memory parallelization is realized with the Intel Threading Building Blocks (TBB) library (tightly integrated into `deal.II`) with support for both parallel `for` loops as well as task-based parallelism that schedules according to dependencies [13].

A major contribution of the present study is the identification of code patterns that provide best performance in shared memory, given several options. Most previous HPC implementations of DG use patterns similar to what we

Table 1: Specification of hardware systems used for evaluation. Memory bandwidth on KNL according to the STREAM benchmark.

	Xeon Phi KNL	Haswell	Power8
Cores	64	14	16
Threads	4 Threads/core	2 Threads/core	8 Threads/core
Frequency	1.3 GHz/core	2.6 GHz/core	3.8 GHz/core
L1 cache	32 kB/core	32 kB/core	64 kB/core
L2 cache	1 MB/(2 cores)	256 kB/core	512 kB/core
Memory	16 GB MCDRAM @ 430 GB/s	L3 Cache: 2*17.5 MB	L3 Cache: 8 MB
	384 GB DDR4 @ 90 GB/s	2.3 GB/core	8 GB/core
SIMD	512 bit	256 bit	128 bit

identify as the loop variant in the following, which also we find to perform best when parallelized. However, the pure performance metrics in terms of memory transfer are better for the alternative task implementation adapted from [9] as seen in Sect 5.1, thus motivating our comparative analysis.

The second major contribution is the documentation of the absolute performance of our kernels on contemporary hardware, namely an Intel Xeon Phi system based on the Knights Landing architecture, a dual-socked Intel Haswell system, and an IBM Power8 system. Table 1 lists the key characteristics of these systems. While the Haswell and Power8 systems are conventional (latency-optimized) CPUs with a moderate number of cores, the KNL system is throughput-oriented with more parallelism but slower two-wide cores derived from the Intel Atom processor [5]. The Knights Landing architecture also comes with a new memory technology, a high bandwidth on-package memory called Multi-Channel DRAM (MCDRAM) in addition to the traditional DDR4 memory. MCDRAM provides up to 5× the bandwidth of DDR4 but is of lower capacity (16GB), accessible through the "cache", "flat" and "hybrid" modes, respectively. The optimal usage of MCDRAM is an open issue, and addressed in our work by memory-lean kernels that can fit into this fast memory.

The remainder of this text is structured as follows. Section 2 presents the fluid dynamics application underlying the tuning. Section 3 gives an overview of the implementation used for benchmarking. In Sect. 4, tuning steps of the code are described. Section 5 compares the performance on three systems using relevant test cases.

2 Application background and discretization

Incompressible fluid flow is governed by the Navier–Stokes equations,

$$\begin{aligned} \frac{\partial \vec{u}}{\partial t} + \nabla \cdot \left(\vec{u} \otimes \vec{u} + p\vec{I} - \frac{1}{\mathrm{Re}} (\nabla \vec{u} + \nabla \vec{u}^{\mathrm{T}}) \right) &= \vec{f}, \\ \nabla \cdot \vec{u} &= 0, \end{aligned} \tag{1}$$

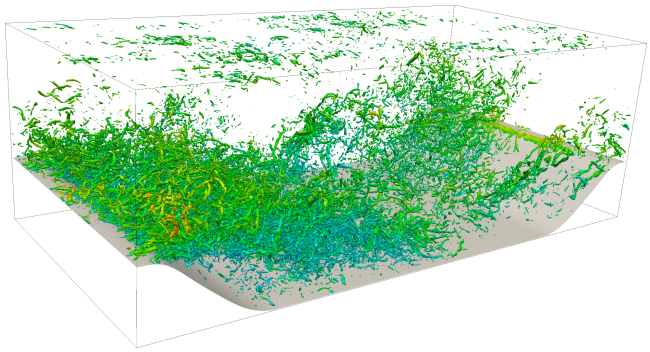

Figure 1: Turbulent flow along a periodic hill, visualized through the Q-criterion, on a computation on a $128 \times 64 \times 64$ boundary-fitted mesh with fourth-degree elements involving 260 million degrees of freedom.

where $\vec{u}$ denotes the (non-dimensional) fluid velocity, p is the pressure, and $\vec{f}$ represents body forces. For large Reynolds numbers Re, the flow becomes turbulent and develops instationary and small-scale features that needs high resolution and efficient solvers. The physically relevant scale range in space and time behaves as Re^3. For moderate to large Reynolds numbers $\mathrm{Re} = 10^4 \ldots 10^8$ whose resolution requirements exceed even the power of large supercomputers, modeling approaches such as large or detached eddy simulation complement direct numerical simulation.

For time discretization of Eq. (1), splitting schemes are most common at larger Reynolds numbers, such as the dual-splitting approach by Karniadakis [7], where each time step involves an explicit convection step, a pressure Poisson equation, a projection to make the velocity field divergence-free, and an implicit viscous step. In Fig. 1, the result of a direct numerical simulation of the turbulent flow around a periodic hill is shown, i.e., a highly resolved computation that covers all length scales relevant to the flow physics. The simulation results have been obtained for a polynomial degree $k = 4$ on a mesh of $128 \times 64 \times 64$ elements, producing 260 million spatial degrees of freedom that are followed over several million time steps.

Table 2 details the distribution of run time in the four phases of a time step in this application, the fraction of time spent in integration kernels similar to the matrix-vector product analyzed in the sequel of this work, and the active access of memory of each step. The active memory can be compared to the global resident memory of 332 GB as measured by accumulation over all 128 nodes involved in the computation. The most demanding part is to solve the pressure Poisson equation with 65 million equations. To ensure optimal complexity and thus efficient use of computational resources, we use an iterative conjugate gradient solver preconditioned by a geometric multigrid V-cycle [12]. The smoothing on each level is done by the Chebyshev iteration which only needs access to the inverse of the entries on the matrix diagonal besides the

Table 2: Run times of the sub-steps involved in one time step of the incompressible flow solver with 260 million degrees of freedom when running on 2048 Sandy Bridge cores. The projection step includes a stabilization according to [10], which invokes fast local conjugate gradient solvers independently for each element.

	run time	# iterations	share mat-vec	memory accessed
explicit convective step	0.012 s	—	100%	59 GB of 332 GB
pressure Poisson equation	0.29 s	11	77%	41 GB of 332 GB
projection step	0.045 s	$20-50$	100%	26 GB of 332 GB
viscous step	0.066 s	3	73%	36 GB of 332 GB

matrix-vector product.

The numbers in Table 2 highlight that code optimizations need to concentrate on the matrix-vector product which accounts for approximately 80% of the total run time. Note that the implementation according to [11] uses a generic interface to integration and improvements made for one kernel typically translate to similar profits in the other variants of the integration loops and thus the whole complex application code. Besides the times for a time step listed in the table, the code also consists an initial setup phase and small data analysis parts take less than 1% of overall run time.

A discretized partial differential operator corresponds to a matrix-vector product in our model. We assume a triangulation of the computational domain into elements $K \in \mathcal{T}_h$. The set of interior faces is denoted by $\mathcal{F}_h^{\mathrm{i}}$ with p^- and p^+ the pressure solution on the respective side of the face, and the set of boundary faces by $\mathcal{F}_h^{\mathrm{b}}$. The bilinear forms $(a,b)_K = \int_K a \odot b \,\mathrm{d}\vec{x}$ and $\langle a,b\rangle_F = \int_F a \odot b \,\mathrm{d}\vec{x}$ denote the inner product of the two quantities and subsequent integration over the element K or the face F, respectively. Using this notation, the discretization of the pressure Poisson equation finds p_h such that the equation

$$\begin{aligned}
&\sum_{K\in\mathcal{T}_h} (\nabla q_h, \nabla p_h)_K + \sum_{F\in\mathcal{F}_h^{\mathrm{i}}} \Big[\big\langle q_h^- - q_h^+, \sigma(p_h^- - p_h^+)\big\rangle_F \\
&\qquad - \left\langle (q_h^- - q_h^+)\vec{n}^-, \frac{\nabla p_h^- + \nabla p_h^+}{2} \right\rangle_F - \left\langle \frac{\nabla q_h^- + \nabla q_h^+}{2}, \vec{n}^-(p_h^- - p_h^+) \right\rangle_F \Big] \\
&+ \sum_{F\in\mathcal{F}_h^{\mathrm{b}}} \langle q_h, 2\sigma p_h\rangle_F - \langle q_h\vec{n}, \nabla p_h\rangle_F - \langle \nabla q_h, \vec{n} p_h\rangle_F = \sum_{K\in\mathcal{T}_h} \Big(q_h, -\frac{\gamma_0}{\Delta t}\nabla\cdot\hat{\vec{u}}\Big)_K
\end{aligned} \tag{2}$$

holds for all test functions q_h. In this equation, $\frac{\gamma_0}{\Delta t}\nabla\cdot\hat{\vec{u}}$ is the forcing given by the divergence of some intermediate-step velocity $\hat{\vec{u}}$ that is usually used in a slightly modified form with integration by parts including central fluxes for the velocity for stability reasons according to [10]. The interior penalty parameter

is denoted by σ and penalizes jumps of the solution over faces, see e.g. [2].

3 Implementation

The operator evaluation is realized by fast integration, using an extension of the framework presented in [11] to discontinuous Galerkin. For the linear operator $\mathcal{L}$ implementing the left hand side of Eq. (2), an input vector $\vec{P}$ is interpreted by its solution function p_h and tested by all basis functions q_h, giving rise to an output vector $\vec{Q}$,

$$\vec{Q} = \mathcal{L}\vec{P}. \tag{3}$$

The integrals are performed according to Eq. (2) on both cells K and faces F. For the cell integrals, the degrees of freedom related to the cell from the global vector are extracted, the local operator is evaluated by integration and tested by all local basis functions and, finally, the local integrals are written into the global result vector. In the integrals, the gradient operators ∇ with respect to the spatial variable $\vec{x}$ in Eq. (2) are replaced by gradients in the reference coordinate $\vec{\xi} \in (0,1)^3$ and multiplied by the Jacobian of the transformation in the usual finite element fashion [11]. The unit-cell operation is the same on all elements and implemented by sum factorization kernels for hexahedra [6, 8]. The Jacobian transformation on Cartesian meshes is the same throughout an element (and possibly over many different elements), whereas a separate $d \times d$ matrix for each quadrature point is necessary for curved meshes. Our realization makes use of these optimizations if the mesh allows for that, significantly reducing memory transfer in the Cartesian mesh case.

The face integrals involve interpolated solutions from the two neighboring cells, tested by basis functions and integrated by a quadrature formula on the faces. In order to avoid double computations when evaluating the integrals to all faces of a cell, we use a separate loop indexing for the faces. Integrals on inner faces combine the information from both adjacent cells in a single step.

The evaluation complexity per degree of freedom with sum factorization is $\mathcal{O}(k+1)$ in the polynomial degree k for cell integrals [11] and $\mathcal{O}(1)$ for face integrals. The proportionality constants are such that the number of arithmetic operations for on-the-fly integration is lower than with the final matrix "stencils" starting at polynomial degrees three to four for continuous elements [11] and at degree two to three of discontinuous ansatz spaces. Since a matrix-based scheme is usually heavily memory-bandwidth bound, matrix-free evaluation is the fastest available evaluation option already for quadratic shape functions. The specific characteristics of the method allow for an almost constant run time per degree of freedom for a wide range of polynomial degrees $2 \leq k \leq 8$ [11, 12], making the polynomial degree essentially a parameter that can be adapted to the complexity of the geometry to be meshed: A more complex geometry will use more elements of somewhat lower polynomial degree. On more regular domains, the higher solution quality of high-order shape functions can be leveraged. Our kernels are integrated into the `deal.II` finite element library [1],

which provides the infrastructure of the mesh, definition of degrees of freedom and parallelization for our application code. This allows for implementing weak forms such as the Laplacian (2) in compact form with only up to a few dozens of lines of code. Despite their generality, the matrix-free kernels outperform the benchmark code from the HPGMG project[1] by 1.5 to 2.5 times on continuous elements [12], which is due to the careful selection of stored data structures vs. on-the-fly computation. The arithmetic intensities of the resulting algorithms are between 1 and 6 FLOP/byte, depending on the geometry, i.e., close to the ridge of memory-bound and computation-bound algorithms [12]. This characteristic makes it necessary to consider both memory efficiency as well as optimizations addressing arithmetics and instruction scheduling.

Since several faces compute integral contributions to the same cell in this layout, the face computations cannot be simply split into subranges within a parallel `for` loop over the faces, and they cannot be arbitrarily mixed with cell integrals in order to avoid race conditions when accessing the global result vector $\vec{Q}$ in Eq. (3). In the following, we discuss two shared-memory parallelizations that avoid these race conditions in different ways. We focus on implementation with the Intel TBB library which is the main thread parallelism paradigm in `deal.II`. An alternative OpenMP-based implementation of our loops has shown very similar performance. We do not consider atomic operations or locks in this work because they have been found to be less efficient on preliminary tests. For the former, no vectorized versions are implemented in CPU hardware yet, reducing efficiency.

3.1 Parallel evaluation through tasks

The task-based parallel scheme adapts the partition-partition scheme described in [9] for finite element operator application. In the discontinuous Galerkin setting, each task includes operations on a range of cells, a range of inner faces, and a range of boundary faces. The latter two ranges are associated with the cell range in order to leverage solution data already in caches from the cell integrals. A race condition can appear if one task operates on an inner face accessing cells K_1 and K_2 and another task simultaneously operates on another inner face involving either K_1 or K_2 or a cell integral on $K_{1/2}$. We note that at least one of the two cells—the one the face is associated with—will be part of the same task and not conflict. However, this does not necessarily hold true for the other cell.

The idea of the partitioning strategy is to compute the connectivity between tasks based on the access pattern of face integrals. The connectivity graph is then split in such a way that layer i is only connected to layers $i-1$ and $i+1$. With this layout, the cells in all even partitions can be parallelized without race conditions. Afterwards, all odd partitions can be run in parallel. Starting with a group of cells for the first partition, the size of the partitions can increase and the total number of partitions might be rather small. In order to create enough

[1] https://hpgmg.org

parallelism, we therefore create another layer of partitions inside each partition. Each partition on the second level is a multiple of a certain block size in order to make sure that vectorization is possible. The idea is related to graph coloring but adapted such that only local synchronization points between the adjacent tasks are necessary, as opposed to global synchronization when running parallel loops on one color at a time, see [9].

The algorithm can be summarized as follows

- Preprocessing: Find the connectivity structure where each cell is associated with a list of cells that share a face with it.
- Assign each block to one partition:
  - Assign the first cells (up to a user-specified grain size) to partition zero.
  - Assign all cells that are connected with cells in partition zero, but not already assigned a partition to partition one. If the number of cells is not a multiple of the grain size, add neighbors of the cells in partition one. This fill-up avoids empty lanes when vectorizing over several cells as described in Sect. 3.3 below.
  - Repeat this procedure until all cells are assigned to a partition.
- For each partition, create a second layer of partitions analogously.
- Create an integer cell indexing according to the double partitioning.

Note that the algorithm is analogous to one described in [9] with the only difference that the connectivity graph uses the dependency of the face integrals instead of the degrees of freedom shared at the element boundary in continuous finite elements.

3.2 Parallel evaluation through `for` loops

An alternative approach to avoid simultaneous writes into the same vector positions is to introduce temporary data structures that hold information for each face integral separately. A common approach in discontinuous Galerkin methods [2, 3] is to interpolate the function u^h to all the faces. In the context of integration where the differential operator is implemented by quadrature and tensor products are involved, the most efficient approach is to perform an interpolation step in face-normal direction and store this data in a global auxiliary variable. This gives the following algorithm for the SIP discretization (2):

1. Loop over all cells $K \in \mathcal{T}_h$:
   - Read values from input vector.
   - Interpolate elemental input values to all $2d$ faces for both values and the reference-cell normal derivative. This gives $2(k+1)^{d-1}$ data values per face that are stored in a global auxiliary vector.
   - Perform cell integration by sum factorization and write result into destination vector.
2. Loop over all inner faces $F \in \mathcal{F}_h^{\mathrm{i}}$:

   - Load the values and normal derivatives from the global auxiliary vector from the element storage on both elements K^- and K^+ and local face numbers f^- and f^+ involved in the face.
   - Sum factorization provides values and reference gradients in face quadrature points.
   - On each quadrature point, implement all face terms involved in (2).
   - Sum over quadrature points and multiplication by test function $v_h^\pm$ and unit cell gradient $\nabla v_h^\pm$ with sum factorization.
   - Write the resulting contribution to value and normal derivative to be tested back into the global auxiliary vector, indexed by the element numbers K^- and K^+ and local face numbers f^- and f^+ involved in the face.
3. Loop over all boundary faces $F \in \mathcal{F}_h^\mathrm{b}$: Similar steps to inner faces.
4. Loop over all cells $K \in \mathcal{T}_h$:
   - Read global auxiliary vector for values and normal derivatives on each face.
   - Finalize integration step by expanding the test functions into the elements.
   - Add resulting contribution into destination vector.

This algorithm has the advantage that all quantities accessed inside the loops are independent from one another. This includes the face loops where different faces access different sections in the auxiliary vector also when they refer to the same element. Thus, a simple parallel `for` loops can be used. The price to pay for this alternative is the global auxiliary vector which needs to be transferred from/to main memory five times, twice for the initial write operation (write, including read-for-ownership), twice during the face loops (read and write), and once for the final interpolation (read). This increased memory access possibly reduces performance in memory-constrained situations.

3.3 Vectorization

An essential ingredient to high performance of the matrix-free operator evaluation is to use SIMD instructions. Automatic vectorization or OpenMP-SIMD annotations apply vectorization to the innermost loops, which is not the optimal strategy for the complex data flow in local integration with sum factorization that runs through the local vectors in different orders when passing through one direction at a time. Also, the subsequent operations on quadrature points use yet another data access pattern. Thus, full vectorization would involve a series of cross-lane permutations on each element. In addition, some lanes might remain empty or non-vectorized peel and remainder loops arise in case the number of degrees of freedom per direction, $k+1$, is not a multiple of the SIMD width, reducing the effective throughput. Finally, we note that vectorization in the sum factorization kernel alone is not enough, as the operation on quadrature points can account for 10 to 50% of the instructions on non-vectorized code [11],

with higher values for more complex operators like the convective term in the incompressible Navier–Stokes equations.

Instead, our work follows the approach proposed in [11], evaluating the contributions from several cells or faces within a SIMD instruction. This is profitable since the operation on each element is the same, albeit with different data from the vector and different geometries and coefficients. This approach only involves a single cross-lane or gather operation when the global vector data is read and written in the task-based algorithm according to Sect. 3.1. The data in solution vectors is stored contiguously for each element and needs to be "transposed" in this step for putting the same nodal point on all elements involved in the SIMD array next to each other. This transposition transforms an array of structures into a structure-of-array data layout. For the loop-based approach from Sect. 3.2, there are two additional transpose operations involved, one when cells access the auxiliary vector in steps 1 and 4 and one when faces access the auxiliary vector in steps 2 and 3, the latter accessing different components as compared to the cell. Note that some transpositions could be avoided by storing the solution vector in an array-of-structure-of-array data layout. However, the access into at least half of the faces would be considerably more complicated, involving gather and scatter operations where each index points into different cache lines, an operation which typically serializes the data access on contemporary hardware implementations and is less efficient than the vectorized access proposed here.

On Haswell, SIMD instructions process four cells/faces (double precision) or eight cells/faces (single precision) at once. On Knights Landing, the numbers are eight for double precision and 16 for single precision, whereas the numbers are two and four on Power8. A disadvantage of the proposed vectorization scheme is that the size of the scratch data fields holding temporary results from sum factorization increases with increasing vector width. The size of the scratch data used for processing a cell integral scales as $6(k+1)^d$, which is multiplied by 32 or 64 bytes in case of Haswell and Knights Landing, respectively. Thus, the scratch data spills L1 caches of Haswell for $k = 5$ and for $k = 4$ on KNL, relying on fast next-level caches for higher degrees. However, the results below show that memory hierarchies are sufficiently capable on both systems, with the exception of KNL on high degrees $k \geq 9$ when the L2 cache capacity is exhausted.

4 Performance tuning

The performance tuning is driven by identifying the most significant bottlenecks in our code and making appropriate changes that reduce or eliminate the effect of these bottlenecks. Node level tuning is performed using the Intel VTune Amplifier Tool [4]. We conduct our analysis and optimization on a dual-socket Intel Xeon E5-2697 v3 according to Table 1 using the double-precision matrix-vector product. To reduce the amount of collected information the hot spot analysis is performed using one thread. The vector load intrinsic operation `_mm256_loadu_pd`

Function / Call Stack	CPU Time: Effective Time by Utilization	CPI Rate
_mm256_loadu_pd	6.4%	3.089
dealii::vectorized_load_and_transpose<double>	6.4%	3.089
_mm256_storeu_pd	6.2%	3.017
dealii::vectorized_transpose_and_store<double>	6.2%	3.017
_mm256_loadu_pd	5.4%	2.980
dealii::vectorized_load_and_transpose<double>	5.4%	2.980
_mm256_loadu_pd	4.5%	2.903
dealii::vectorized_load_and_transpose<double>	4.5%	2.903

Figure 2: Top-Down view of the function call stack for the Cartesian grid case.

Function / Call Stack	CPU Time: Effective Time by Utilization	CPI Rate
dealii::VectorizedArray<double>::operator+=	11.8%	2.858
↖ dealii::FEEvaluationAccess<(int)3, (int)1, double, (bool)0>::get_gradient ⟵ l	11.8%	2.858
dealii::VectorizedArray<double>::operator+=	6.4%	1.837
↖ dealii::FEEvaluationBase<(int)3, (int)1, double, (bool)1>::get_normal_gradier	6.4%	1.837
_mm256_loadu_pd	6.3%	3.934
dealii::vectorized_load_and_transpose<double>	6.3%	3.934
_mm256_storeu_pd	5.1%	3.908
dealii::vectorized_transpose_and_store<double>	5.1%	3.908

Figure 3: Top-Down view of the function call stack for curvilinear grid case.

which is called several times from the function `vectorized_load_and_transpose` is the most time-consuming operation as can be seen from the top-down view of the function call stack for the case of the Cartesian grid configuration shown in Fig. 2, which appears inside the aforementioned transposition step. The loop in the function `vectorized_load_and_transpose_base` performs vector load operations based on offset values inside a loop. We could improve this code by moving the redundant calculation of the offset indices outside the loop by introducing double pointers that can be held in registers.

Furthemore the same function `vectorized_load_and_transpose` appears in the code vectorization analysis performed by Intel Advisor. The analysis shows an inefficient use of vector registers due to the complex structures of the array indices. Keeping this in mind, we modify the function `vectorized_load_and_transpose` correspondingly. To help the compiler performing vectorization of the loop we additionally inserted an OMP SIMD pragma.

Going to the case of a curvilinear grid configuration, Fig. 3 identifies the same function `vectorized_load_and_transpose` among the hot spots. However, the most time consuming function in this case is the overloaded `+=` operation called from the function `get_gradient`. The source code has multiple layers of abstractions for performing SIMD data additions. However, this function representing the actual arithmetic work maps to optimal assembler code, including fused multiply-add instruction identified by the compiler.

5 Performance comparison

All code was compiled with the `gcc` compiler, version 6.2, at optimization level `-O3 -march=native`. The performance with `gcc` is within 2% of the performance with the Intel compiler (v. 16.0 and 17.0) when compiled at `-O3 -xhost` on Haswell. The similar performance is due to the explicit vectorization and it depends on the particular instruction scheduling and loop unrolling which of these two compilers performs slightly better. On KNL, the code generated by `gcc` provides around 1.5 times higher throughput because the Intel compiler for KNL is not able to merge vectorized multiplications and additions arising in the high-level `C++` implementation in `deal.II` into fused multiply-add instructions for `__m512d` data types. For the Power8 system the code is compiled using the Advanced Toolchain for PowerLinux.9.0 with the IBM's Mathematical Acceleration Subsystem (MASS) libraries. It exploits the advanced capabilities for the POWER vector instruction set from the Vector Multimedia eXtension (VMX). Scalability tests on Power8 are performed by varying the number of hardware threads in each core via the SMT option.

All times are reported as the minimal run time of the matrix-vector product out of ten experiments. Since Intel TBB dynamically distributes the tasks to threads without direct pinning [13], we use the "affinity partitioner" in the `for` loop variant to ensure that repeated loops run on the same threads. This is beneficial for the non-uniform memory access on the Haswell system where a first-touch page assignment of the data stored on cells such as Jacobian transformations for the loop-based algorithm from Sect. 3.2 is used. No affinity can be used for the task-based scheme. For KNL, MCDRAM memory is configured in "flat" mode where it is mapped to physical address space and exposed as a NUMA node (allocatable memory) in all experiments except the detailed analysis in Sect. 5.4.

5.1 Comparison of task and loop parallelization

Fig. 4 compares the parallel scaling of a code with $\mathcal{Q}_4$ elements and 32.8 million degrees of freedom for the two algorithmic variants presented in Sect. 3 on the Haswell, Knights Landing, and Power8 architectures.

On a single Haswell core, the task-based scheme is faster than the loop-based scheme that splits computations into several parts, using 0.72 s rather than 1.17 s. This is due to the lower memory transfer and better cache utilization. An analysis of the matrix-vector product with the likwid tool[2] on a single core reports 0.63 GB of read transfer and 0.32 GB of write transfer to main memory for the layout from Sect. 3.1 (one solution vector: 0.26 GB). Conversely, the `for` loop from Sect. 3.2 involves 2.76 GB of reads and 1.82 GB of writes (size of auxiliary vector: 0.63 GB).

Despite the advantage in terms of memory transfer, the parallel task implementation scales considerably worse than the `for` loops. Therefore, the latter

[2] `https://github.com/RRZE-HPC/likwid`, retrieved on September 18, 2016

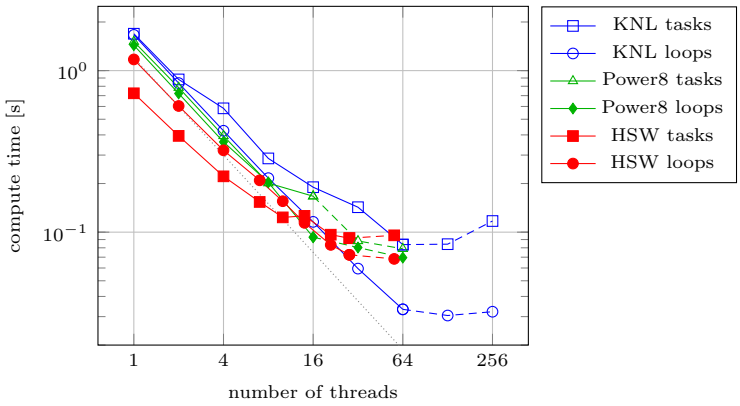

Figure 4: Parallel scaling study of various task parallelization schemes on Haswell and Knights Landing using a problem with $\mathcal{Q}_4$ elements on a 64^3 Cartesian mesh, using 32.7 million degrees of freedom. Dashed lines indicate the step to logical cores with simultaneous multithreading (SMT).

reaches 0.068 s on 28 Haswell cores, faster than the task-based scheme at 0.092 s. Power8 is slower with a single thread than Haswell due to the narrower SIMD width, but reaches similar performance to Haswell at higher thread counts and in particular with SMT thanks to more parallelism inside the cores and presumably a better memory controller.

On a single core, the Knights Landing system is slowest with a wall time of around 1.7 s according to Fig. 4. Given that a KNL core is weak with only half the clock frequency of Haswell and restricted capabilities, a slowdown of only about 30% is remarkable particularly for the loop kernel and shows the effect of wider vectorization. The loop-based algorithm shows excellent strong scaling when increasing the core count to 64, reaching a parallel speedup of more than 50 (and 55 when going to 128 cores). For comparison, the parallel speedup is 17 on the Haswell system.

The task-based parallelization from Sect. 3.1 shows worse behavior on both Intel systems, in particular the KNL system with more cores. Since the code analysis shows that CPUs are mostly busy in that case, we suspect that the dynamic scheduling of tasks with complex dependencies results in less optimal usage of the memory hierarchy such as prefetchers and caches.

5.2 Analysis of vectorization

In order to leverage the higher performance of single-precision arithmetics, our solvers use mixed precision: The outer residual computations and matrix-vector products according to Eq. (2) are done with double precision for algorithmic stability, whereas it is enough to use single precision for preconditioning the linear systems, which applies to the full multigrid V-cycle in the Poisson solver.

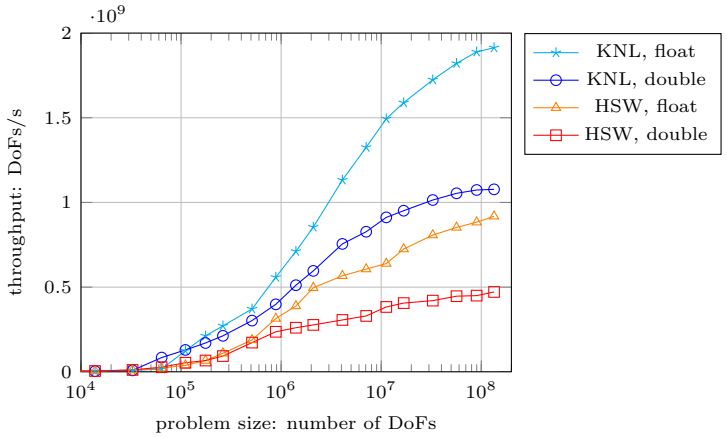

Figure 5: Performance of matrix-vector product on discontinuous $\mathcal{Q}_3$ elements. Loop parallelization from Sect. 3.2.

Thus, the throughput of operator evaluation is recorded for both single precision and double precision. Fig. 5 displays the number of degrees of freedom processed per second on the Haswell and Knights Landing architectures, respectively. For large problem sizes, evaluation in single precision is between 1.6 and 1.9 times as fast. The gap to the ideal factor 2 is mainly because the global loops are half as long when the number of element batches due to SIMD halves. This can be seen from the fact that the gap still widens as the problem size increases and more parallelism becomes available.

When turning to the absolute throughput numbers, our results show that the dual-socket Haswell system can evaluate the DG operator (2) for up to 480 million degrees of freedom per second in double precision, whereas Knights landing reaches 1.1 billion degrees of freedom per second. This speedup of a factor of 2.3 at a somewhat lower power consumption shows the capabilities of the KNL system for throughput-oriented tasks such as the massively parallel integration tasks in DG. On the other hand, initialization routines including many indirections and a mix of integer and floating point code run half as fast on KNL as on Haswell when both systems are fully populated.

Fig. 6 evaluates the effectiveness of the explicit vectorization over elements as described in Sect. 3.3 by comparison with auto-vectorization explored by the compiler. In order to increase the possibilities for automatic vectorization, the pointers inside the sum-factorization kernels are annotated with the C/C++ `__restrict` keyword and OMP SIMD pragmas to exclude pointer aliasing. In all configurations, the explicit vectorization holds a clear performance advantage with up to a factor of 4.3. For Knights Landing with wider vector units, the gain with explicit vectorization is larger than on Haswell. Likewise, single precision shows larger gains than double precision. We note that the explicit vectorization path performs essentially all arithmetic operations in packed form.

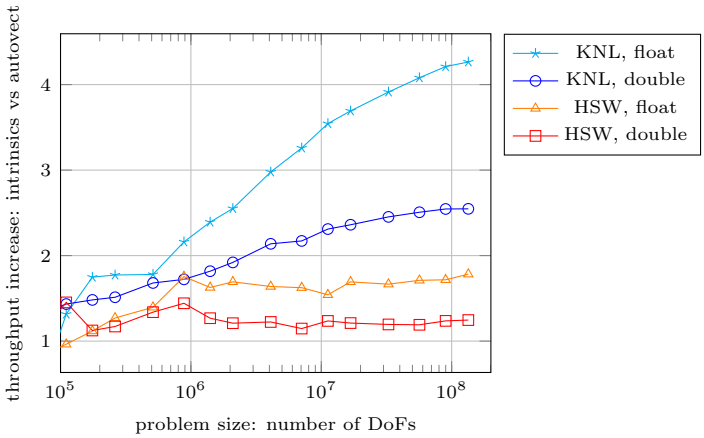

Figure 6: Effectiveness of vectorization measured as the ratio in throughput of auto-vectorized code and the explicit vectorization over several elements with intrinsics according to Sect. 3.3 for $\mathcal{Q}_3$ elements. Numbers larger than 1 point to an advantage of the explicit vectorization.

Measurements of the Haswell code with the likwid tool shows that more than 99% of floating point instructions in the relevant sections are on 256-bit packed data, whereas only up to 15% of arithmetic operations are vectorized by auto-vectorization according to the likwid analysis, both measured for $\mathcal{Q}_3$ elements. Despite a potential fourfold improvement for AVX vectorization on Haswell, the actual speedup is between 20% and 80% because of memory bottlenecks in the wider code. This is explained by a substantially higher instruction throughput for the non-vectorized code at 2.38 instructions per clock cycle versus 1.47 instructions per cycle for the explicitly vectorized code.

The comparison of vectorization efficiency of Haswell and KNL also allows for projections about the run time on future systems: As soon as Intel Xeon CPUs move to AVX-512 instructions with the Skylake Server architecture, we expect them to surpass KNL in efficiency on computation bound kernels on Cartesian meshes with twice the theoretical throughput of Haswell. However, we expect KNL to remain faster due to the higher memory throughput of MCDRAM on the curved mesh which is memory bound.

5.3 Performance metrics of loop kernel

Table 3 details the run time of the individual components in the loop parallelization according to Sect. 3.2 together with an analysis of memory transfer and arithmetic operations measured with the likwid tool. Steps 2+3 as well as the part that computes the cell integral in step 1 involve more arithmetics as compared to the other operations. For the cell integration, Haswell reaches an arithmetic throughput of almost 400 GFLOP/s when counting FMA instruc-

Table 3: Run time analysis of loop-parallel code in terms of memory transfer and GFLOP/s as measured with the likwid tool on 32.7 million degrees of freedom with $\mathcal{Q}_4$ elements.

	Haswell 56 threads			KNL 128 threads		
	time [s]	GB/s	GFLOP/s	time [s]	GB/s	GFLOP/s
step 1, all	0.026	80	230	0.013	160	460
step 1, cell part	0.012	67	386	0.0092	87	510
steps 2+3	0.024	55	210	0.012	110	420
step 4	0.020	60	72	0.0063	190	220

tions as two operations.[3] About 55% of the arithmetic instructions are FMAs and the others separate additions and multiplications. This relatively low proportion of FMAs is due to our implementation that targets minimal execution time rather than maximal FLOP rates by using the so-called even-odd decomposition [8]. With these numbers, the cell integration part from step 1 reaches more than 60% of arithmetic peak on Haswell. In the memory-dominated parts, the complicated access patterns and possible issues in the memory pipeline prevent the implementation to reach the full memory performance which is measured as 95 GB/s for the STREAM add kernel on the Haswell system and 430 GB/s on KNL. Due to different memory intensities in these four steps, the complete matrix-vector product is relatively far away from the performance limits of the architectures, reaching 67 GB/s and 182 GFLOP/s on Haswell and 150 GB/s and 406 GFLOP/s on KNL. KNL is generally a bit farther from theoretical performance limits, showing the impact of the weaker core with instruction-scheduling bottlenecks.

As a metric for the performance in different application scenarios, Fig. 7 shows the throughput of the matrix-vector product as a function of the polynomial degree. Results have been generated for problem sizes between 16 and 134 million degrees of freedom on the Cartesian mesh and 4.5 to 25 million degrees of freedom on the curved mesh, both chosen such that the local kernels fit into the 16 GB MCDRAM memory of KNL that is operated in flat mode. As the polynomial degree increases, there are jumps in the number of elements which have a slight effect on the throughput, namely between $k = 3$ and $k = 4$ and between $k = 7$ and $k = 8$, respectively. The results show more than a two-fold advantage of KNL over Haswell on moderate polynomial degrees $k \leq 6$. Moreover, the advantage is more pronounced on the curved mesh case which has a lower arithmetic intensity (2 FLOP/byte versus 5 FLOP/byte). This result highlights the importance of fast MCDRAM memory as compared to the

[3] As a complement to the numbers given by likwid that count FMAs as one FLOP, we recorded FMAs and additions and multiplication separately with the Intel software development emulator.

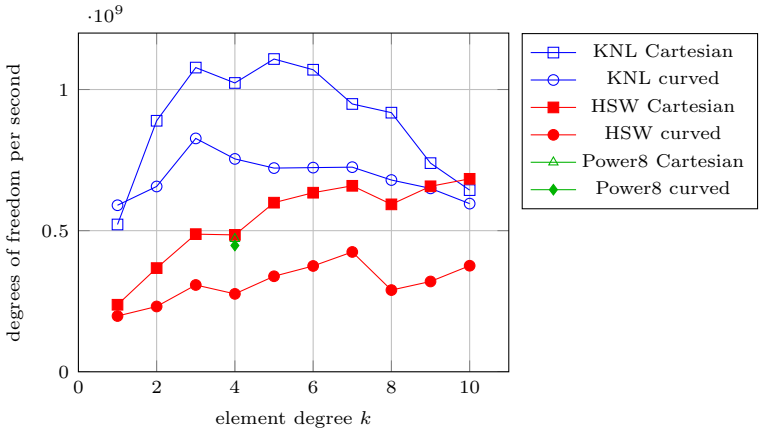

Figure 7: Performance of double-precision matrix-vector product on discontinuous elements as a function of the polynomial degree.

Haswell system, see also the results in Table 4 below.

5.4 Memory mode of KNL

For the KNL system, the fast MCDRAM is an essentialy ingredient to reach high performance, in particular due to the high dependence on memory throughput documented by the kernel analysis in Sect. 5.3. In Table 4 the performance of "flat" mode is compared to "cache" mode. For the "flat" mode, we use `numactl` to ensure that all memory allocations go to the MCDRAM or the DDR4 RAM, respectively. In flat mode, the code runs two to three times faster from MCDRAM than from DDR4 memory. When comparing the results of the flat mode with the cache mode, we see that the cache mode reaches similar or even slightly better performance than the flat mode for algorithms which repeatedly access the same memory. According to the numbers shown in Table 2, the fluid dynamics application is an ideal target for this mode: The application code does not need to specify at compile time where the memory should be allocated (like when using `hbwmalloc` through `memkind` [5] going to MCDRAM): In case the whole application fits into MCDRAM, full performance is reached in the steady state of many time steps. In case the overall program exceeds the MCDRAM cache, all steps except for the explicit convection step (with only one sweep through data) involve iterations with repeated access to at most one tenth of global resident memory.

6 Conclusions and outlook

In this paper, we have discussed the portability of a matrix-free discontinuous Galerkin code to the new KNL and Power8 architectures. We have analyzed

Table 4: Comparison of memory modes on KNL, measured as degrees of freedom processed per second (DoFs/s).

	Cache mode		Flat mode	
	1st run	avg 100 runs	MCDRAM	DDR4
Cartesian mesh $\mathcal{Q}_2$	$0.46 \cdot 10^9$	$0.89 \cdot 10^9$	$0.89 \cdot 10^9$	$0.53 \cdot 10^9$
curved mesh $\mathcal{Q}_2$	$0.25 \cdot 10^9$	$0.68 \cdot 10^9$	$0.66 \cdot 10^9$	$0.20 \cdot 10^9$
Cartesian mesh $\mathcal{Q}_4$	$0.44 \cdot 10^9$	$1.22 \cdot 10^9$	$1.08 \cdot 10^9$	$0.59 \cdot 10^9$
curved mesh $\mathcal{Q}_4$	$0.24 \cdot 10^9$	$0.82 \cdot 10^9$	$0.75 \cdot 10^9$	$0.24 \cdot 10^9$
Cartesian mesh $\mathcal{Q}_8$	$0.41 \cdot 10^9$	$1.16 \cdot 10^9$	$0.92 \cdot 10^9$	$0.53 \cdot 10^9$
curved mesh $\mathcal{Q}_8$	$0.30 \cdot 10^9$	$0.75 \cdot 10^9$	$0.68 \cdot 10^9$	$0.22 \cdot 10^9$

and optimized node-level performance by vectorization and thread parallelism with two different algorithms. The first algorithm is based on tasks scheduled according to nearest-neighbor dependencies, while the second is based on simple `for` loops. While the latter parallelization scheme is simpler and allows for more regular data access, it comes with the overhead of additional transfer from RAM memory for temporary face data buffers. Due to the memory overhead this option performs generally worse for low order elements where the amount of face data as compared to cell data is larger and the arithmetic intensity is lower. Nonetheless, it performs considerably better when parallelized on the many-core KNL, highlighting that `for` loops with regular scheduling is beneficial.

Our code implements an explicit vectorization of the integrals which improves runtime significantly as compared to automatic vectorization. This effect is more pronounced for wider vector units, rendering this feature essential for new Intel architectures with 512-bit vector units. Explicit vectorization makes the double-precision kernel run around 2.5 times faster and the single-precision kernel more than four times faster on KNL. When going from Haswell to KNL, we obtain a speedup of a factor 1.5 to 2.5 for our matrix-vector products. Despite different hardware architectures, the IBM Power8 and the Intel Haswell system showed very similar performance, with a slight advantage of the former on memory-heavy operations and a slight advantage of the latter on more arithmetic-heavy parts.

In the future, we plan to combine shared-memory performance obtained on single KNL nodes with MPI in a hybrid parallelization scheme to obtain results in computational engineering with unprecedented accuracy on emerging KNL clusters.

Acknowledgements

The authors acknowledge the support given by the Bayerische Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR) in the framework of the project *High performance finite difference stencils for modern parallel processors*. 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). 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.

The authors acknowledge collaboration with Benjamin Krank, Niklas Fehn, and Matthias Brehm.

References

[1] Bangerth, W., Davydov, D., Heister, T., Heltai, L., Kanschat, G., Kronbichler, M., Maier, M., Turcksin, B., Wells, D.: The `deal.II` library, version 8.4. J. Numer. Math. 24(3), 135–141 (2016), `www.dealii.org`. doi:10.1515/jnma-2016-1045

[2] Hesthaven, J.S., Warburton, T.: Nodal discontinuous Galerkin methods: Algorithms, analysis, and applications, Texts in Applied Mathematics, vol. 54. Springer (2008). doi:10.1007/978-0-387-72067-8

[3] Hindenlang, F., Gassner, G., Altmann, C., Beck, A., Staudenmaier, M., Munz, C.D.: Explicit discontinuous Galerkin methods for unsteady problems. Comput. Fluids 61, 86–93 (2012). doi:10.1016/j.compfluid.2012.03.006

[4] Intel Corporation: Intel VTune Amplifier XE 2017, `https://software.intel.com/en-us/intel-vtune-amplifier-xe`

[5] Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming, Knights Landing edition. Morgan Kaufmann, Cambridge, MA (2016)

[6] Karniadakis, G.E., Sherwin, S.J.: Spectral/hp element methods for computational fluid dynamics. Oxford University Press, 2nd edn. (2005). doi:10.1093/acprof:oso/9780198528692.001.0001

[7] Karniadakis, G.E., Israeli, M., Orszag, S.A.: High-order splitting methods for the incompressible Navier–Stokes equations. J. Comput. Phys. 97(2), 414 – 443 (1991). doi:10.1016/0021-9991(91)90007-8

[8] Kopriva, D.: Implementing spectral methods for partial differential equations. Springer (2009). doi:10.1007/978-90-481-2261-5

[9] Kormann, K., Kronbichler, M.: Parallel finite element operator application: Graph partitioning and coloring. In: Proc. 7th IEEE Int. Conf. eScience. pp. 332–339 (2011). doi:10.1109/eScience.2011.53

[10] Krank, B., Fehn, N., Wall, W.A., Kronbichler, M.: A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. arXiv preprint arXiv:1607.01323 (2016)

[11] Kronbichler, M., Kormann, K.: A generic interface for parallel cell-based finite element operator application. Comput. Fluids 63, 135–147 (2012). doi:10.1016/j.compfluid.2012.04.012
[12] Kronbichler, M., Wall, W.A.: A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. arXiv preprint arXiv:1611.03029 (2016)
[13] Reinders, J.: Intel Threading Building Blocks. O'Reilly (2007)