Hardware-Based Efficiency Advances in the EXA-DUNE Project

Peter Bastian, Christian Engwer, Jorrit Fahlke, Markus Geveler, Dominik Göddeke, Oleg Iliev, Olaf Ippisch, René Milk, Jan Mohring, Steffen Müthing, Mario Ohlberger, Dirk Ribbrock, and Stefan Turek

Abstract We present advances concerning efficient finite element assembly and linear solvers on current and upcoming HPC architectures obtained in the frame of the EXA-DUNE project, part of the DFG priority program 1648 *Software for Exascale Computing* (SPPEXA). In this project, we aim at the development of both flexible and efficient hardware-aware software components for the solution of PDEs based on the DUNE platform and the FEAST library. In this contribution, we focus on node-level performance and accelerator integration, which will complement the proven MPI-level scalability of the framework. The higher-level aspects of the EXA-DUNE project, in particular multiscale methods and uncertainty quantification, are detailed in the companion paper (Bastian et al., Advances concerning multiscale methods and uncertainty quantification in EXA-DUNE. In: Proceedings of the SPPEXA Symposium, 2016).

P. Bastian • S. Müthing (✉)
Interdisciplinary Center for Scientific Computing, Heidelberg University, Heidelberg, Germany
e-mail: peter.bastian@iwr.uni-heidelberg.de; steffen.muething@iwr.uni-heidelberg.de

C. Engwer • J. Fahlke • R. Milk • M. Ohlberger
Institute for Computational and Applied Mathematics, University of Münster, Münster, Germany
e-mail: christian.engwer@wwu.de; rene.milk@wwu.de; mario.ohlberger@wwu.de

D. Göddeke
Institute of Applied Analysis and Numerical Simulation, University of Stuttgart, Stuttgart, Germany
e-mail: dominik.goeddeke@mathematik.uni-stuttgart.de

M. Geveler • D. Ribbrock • S. Turek
Institute of Applied Mathematics, TU Dortmund, Dortmund, Germany
e-mail: markus.geveler@math.tu-dortmund.de; dirk.ribbrock@math.tu-dortmund.de; stefan.turek@math.tu-dortmund.de

O. Iliev • J. Mohring
Fraunhofer Institute for Industrial Mathematics ITWM, Kaiserslautern, Germany
e-mail: oleg.iliev@itwm.fraunhofer.de; jan.mohring@itwm.fraunhofer.de

O. Ippisch
Institut für Mathematik, TU Clausthal-Zellerfeld, Clausthal-Zellerfeld, Germany
e-mail: olaf.ippisch@tu-clausthal.de

© Springer International Publishing Switzerland 2016

H.-J. Bungartz et al. (eds.), *Software for Exascale Computing – SPPEXA 2013-2015*, Lecture Notes in Computational Science and Engineering 113,
DOI 10.1007/978-3-319-40528-5_1

1 The EXA-DUNE Project

Partial differential equations (PDEs) – often parameterized or stochastic – lie at the heart of many models for processes from science and engineering. Despite ever-increasing computational capacities, many of these problems are still only solvable with severe simplifications, in particular when additional requirements like uncertainty quantification, parameter estimation or optimization in engineering applications come into play.

Within the EXA-DUNE[1] project we pursue three different routes to make progress towards exascale: (i) we develop new computational algorithms and implementations for solving PDEs that are highly suitable to better exploit the performance offered by prospective exascale hardware, (ii) we provide domain-specific abstractions that allow mathematicians and application scientists to exploit (exascale) hardware with reasonable effort in terms of programmers' time (a metric that we consider highly important) and (iii) we showcase our methodology to solve complex application problems of flow in porous media.

Software development, in the scope of our work for the numerical solution of a wide range of PDE problems, faces contradictory challenges. On the one hand, users and developers prefer flexibility and generality, on the other hand, the continuously changing hardware landscape requires algorithmic adaptation and specialization to be able to exploit a large fraction of peak performance.

A framework approach for entire application domains rather than distinct problem instances facilitates code reuse and thus substantially reduces development time. In contrast to the more conventional approach of developing in a 'bottom-up' fashion starting with only a limited set of problems and solution methods (likely a single problem/method), frameworks are designed from the beginning with flexibility and general applicability in mind so that new physics and new mathematical methods can be incorporated more easily. In a software framework the generic code of the framework is extended by the user to provide application specific code instead of just calling functions from a library. Template meta-programming in C++ supports this extension step in a very efficient way, performing the fusion of framework and user code at compile time which reduces granularity effects and enables a much wider range of optimizations by the compiler. In this project we strive to redesign components of the DUNE framework [1, 2] in such a way that hardware-specific adaptations based on the experience acquired within the FEAST project [18] can be exploited in a transparent way without affecting user code.

Future exascale systems are characterized by a massive increase in node-level parallelism, heterogeneity and non-uniform access to memory. Current examples include nodes with multiple conventional CPU cores arranged in different sockets. GPUs require much more fine-grained parallelism, and Intel's Xeon Phi design shares similarities with both these extremes. One important common feature of all

[1] http://www.sppexa.de/general-information/projects.html#EXADUNE

these architectures is that reasonable performance can only be achieved by explicitly using their (wide-) SIMD capabilities. The situation becomes more complicated as different programming models, APIs and language extensions are needed, which lack performance portability. Instead, different data structures and memory layouts are often required for different architectures. In addition, it is no longer possible to view the available off-chip DRAM memory within one node as globally shared in terms of performance. Accelerators are typically equipped with dedicated memory, which improves accelerator-local latency and bandwidth substantially, but at the same time suffers from a (relatively) slow connection to the host. Due to NUMA (non-uniform memory access) effects, a similar (albeit less dramatic in absolute numbers) imbalance can already be observed on multi-socket multi-core CPU systems. There is common agreement in the community that the existing MPI-only programming model has reached its limits. The most prominent successor will likely be 'MPI+X', so that MPI can still be used for coarse-grained communication, while some kind of shared memory abstraction is used within MPI processes at the UMA level. The upcoming second generation of Xeon Phi (Knight's Landing) will be available both as a traditional accelerator and as a standalone, bootable CPU, enabling new HPC architecture designs where a whole node with accelerator-like properties can function as a standalone component within a cluster. Combined with the ISA convergence between standard CPUs and the new Xeon Phi, this will allow for a common code base that only has to be parameterized for the different performance characteristics (powerful versus simplistic cores and the much higher level of intra-node parallelism of the Xeon Phi) with a high potential of vastly improved developer productivity. At the same time, Xeon Phi processors will contain special on-package RAM, bringing the highly segmented memory architecture of accelerator cards one step closer to general purpose CPUs. Similarly, NVIDIA has announced the inclusion of general purpose ARM cores in upcoming generations of their GPGPUs.

Our work within the EXA-DUNE project currently targets pilot applications in the field of porous media flow. These problems are characterized by coupled elliptic/parabolic-hyperbolic PDEs with strongly varying coefficients and highly anisotropic meshes. The elliptic part mandates robust solvers and thus does not lend itself to the current trend in HPC towards matrix-free methods with their beneficial properties in terms of memory bandwidth and/or Flops/degree of freedom (DOF) ratio; typical matrix-free techniques like stencil-based geometric multigrid are not suited to those types of problems. For that reason, we aim at algebraic multigrid (AMG) preconditioners known to work well in this context, and work towards further improving their scalability and (hardware) performance. Discontinuous Galerkin (DG) methods are employed to increase data locality and arithmetic intensity. Matrix-free techniques are investigated for the hyperbolic/parabolic parts.

In this paper we report on the current state of the lower-level components of the EXA-DUNE project, while the more application-oriented parts will be presented in a separate article in these proceedings [3]. Message passing parallelism is well established in DUNE (as documented by the inclusion of DUNE's solver library in

the High-Q-Club[2]), and we thus concentrate on core/node level performance. After a short introduction to our UMA node concept in Sect. 2.1, the general structure comprises two major parts: Sect. 3 focuses on problem assembly and matrix-free solvers, in particular thread-parallel assembly (Sect. 3.1) and medium to high order DG schemes (Sect. 3.2), while Sect. 4 is concerned with linear algebra, where we show the integration of a modern, cross-platform matrix format (Sect. 4.1) as well as hardware-oriented preconditioners for the CUDA architecture (Sect. 4.2).

2 Hybrid Parallelism in DUNE

In the following, we introduce the 'virtual UMA node' concept at the heart of our hybrid parallelization strategy, and ongoing current steps to incorporate this concept into the assembly and solver stages of our framework.

2.1 UMA Concept

Current and upcoming HPC systems are characterized by two trends which greatly increase the complexity of efficient node-level programming: (i) a massive increase in the degree of parallelism restricts the amount of memory and bandwidth available to each compute unit, and (ii) the node architecture becomes increasingly heterogeneous. Consequently, on modern multi-socket nodes the memory performance depends on the location of the memory in relation to the compute core (NUMA). The problem becomes even more pronounced in the presence of accelerators like Xeon Phi or GPUs, for which memory accesses might have to traverse the PCIe bus, severely limiting bandwidth and latency. In order to demonstrate the performance implications of this design, we consider the relative runtime of an iterative linear solver (Krylov-DG), as shown in Table 1: an identical problem is solved with different mappings to MPI processes and threads, on a representative 4-socket server with AMD Opteron 6172 12-core processors and 128 GB RAM. On this architecture, a UMA domain comprises half a socket (6 cores), and thus, (explicit or implicit) multi-threading beyond 6 cores actually yields slowdowns. Note that all different configurations in this benchmark use all available cores; they only differ in how many MPI ranks (UMA domains) they allocate. This experiment validates our design decision to regard heterogeneous nodes as a collection of 'virtual UMA nodes' on the MPI level: internal uniform memory access characteristics are exploited by shared memory parallelism, while communication between UMA domains is handled via (classical/existing) message passing.

[2] http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/_node.html

Table 1 Poisson problem on the unit cube, discretized by the DG-SIPG method, timings for 100 Krylov iterations. Comparison of different MPI/shared memory mappings for varying polynomial degree p of the DG discretization and mesh width h. Timings $t_{M/T}$ and speedups for varying numbers of MPI processes M and threads per process T

p	h^{-1}	$t_{48/1}[s]$	$t_{8/6}[s]$	$\frac{t_{48/1}}{t_{8/6}}$	$t_{4/12}[s]$	$\frac{t_{48/1}}{t_{4/12}}$	$t_{1/48}[s]$	$\frac{t_{48/1}}{t_{1/48}}$
1	256	645.1	600.2	1.07	1483.3	0.43	2491.7	0.26
2	128	999.5	785.7	1.27	1320.7	0.76	2619.0	0.38
3	64	709.6	502.9	1.41	1237.2	0.57	1958.2	0.36

3 Assembly

As discussed before, we distinguish between three different layers of concurrency. Below the classical MPI level, we introduce thread parallelization as a new layer of parallelism on top of the existing DUNE grid interface. The number of threads is governed by the level of concurrency within the current UMA node, as explained above.

The grid loop is distributed among the threads, which allows for parallel assembly of a finite element operator or a residual vector. In order to achieve good performance within an individual thread, two major problems need to be solved: (i) as the majority of operations during finite element assembly are memory bandwidth bound, a naive approach to multithreading will not perform very well and achieve only minor speedups. (ii) Vectorization (SIMD, ILP) is required to fully exploit the hardware of modern processors or many core systems. Even in memory bound contexts, this is important as it reduces the number of load and store instructions. Due to the much more complicated and problem-specific kernels that occur as part of problem assembly, a user-friendly integration of vectorization into assembly frameworks poses a much more difficult problem compared to linear algebra, where the number of kernels is much smaller.

Finally, these building blocks need to be optimized with regard to additional constraints like memory alignment and optimal cache utilization.

We believe that locally structured data is the key to achieve both of these goals. We follow two approaches to introduce local structure to a globally unstructured discretization:

1. Low order methods: we employ a structured refinement on top of an unstructured parallel mesh. The structured refinement is computed on the fly and leads to a well defined sparse local structure, i.e., band matrices.
2. Higher order methods: higher order DG methods lead to block structured data structures and access patterns, which allow for high computational efficiency. We pair these types of methods with a sum factorized assembly algorithm to achieve a competitive effort per DOF.

While both of these methods yield computational benefits, they are still very different, and while the higher amount of structure in a medium or high order DG

method might yield a higher computational efficiency, these methods also require a solution with more regularity than a low order method to achieve good convergence, making the right choice of method strongly dependent on the problem at hand.

3.1 Thread Parallel Assembly

Thread parallel assembly mainly consists of thread parallel grid sweeps and concurrent writes to the residual vector or stiffness matrix. We identify two design decisions with a significant performance impact:

(A) *Partitioning:* an existing set of grid cells – with unknown ordering – has to be split into subsets for the different threads.
(B) *Access Conflicts:* each thread works on its own set of cells, but shares some DOFs with other threads, requiring a strategy to avoid those write conflicts.

In order to describe subsets of cells in an implementation agnostic manner, we introduce the concept of `EntitySets`. They encapsulate an iterator range, describing a set of grid objects, in this case a set of grid cells and provide a map from grid entities into a (globally) consecutive index range for data storage.

For a given mesh $\mathscr{T}(\Omega)$ we consider three different strategies, where the first two are directly based on the induced linear ordering of all mesh cells $e \in \mathscr{T}(\Omega)$. In [7] we presented performance tests to evaluate the different partitioning strategies.

strided: for P threads, each thread p iterates over the whole set of cells $e \in \mathscr{T}(\Omega)$, but stops only at cells where $e \bmod P = p$ holds.
ranged: $\mathscr{T}$ is split into consecutive iterator ranges of the size $|\mathscr{T}|/P$, using iterators over $\mathscr{T}$ to define the entry points.
general: general partitions, like those obtained from graph partitioning libraries like METIS or SCOTCH need to store copies of all cells in the `EntitySet`. This is the most flexible approach, but typically creates non-contiguous per-thread subsets of the `EntitySet`, which in turn leads to less cache-efficient memory access patterns.

To avoid *write conflicts* we consider three different strategies:

entity-wise locks are expected to give very good performance, at the cost of additional memory requirements.
batched write uses a global lock, but the frequency of locking attempts is reduced. Updates are collected in a temporary buffer and the lock is acquired when the buffer is full.
coloring avoids write conflicts totally, but requires a particular partitioning scheme.

The experiments performed in [7] indicate that ranged partitioning with entity-wise locking and coloring can be implemented with a low overhead in the thread parallelization layer and show good performance on classic multi-core CPUs and

on modern many-core systems alike. In our test the performance gain from coloring was negligible (cf. Sect. 3.3), but the code complexity increased considerably, leading us to settle on the ranged partitioning strategy for the rest of this paper.

3.2 Higher Order DG Methods

As explained in the introduction, we focus on porous media applications to demonstrate the overall viability of our approach to extreme scale computing within our project and we initially consider the prototypical problem of density driven flow in a three-dimensional domain $\Omega = (0,1)^3$ given by an elliptic equation for pressure $p(x,y,z,t)$ coupled to a parabolic equation for concentration $c(x,y,z,t)$:

$$-\nabla \cdot (\nabla p - c\mathbf{1}_z) = 0 \,, \tag{1}$$

$$\partial_t c - \nabla \cdot \left((\nabla p - c\mathbf{1}_z)c + \frac{1}{Ra}\nabla c \right) = 0 \,. \tag{2}$$

This system serves as a model for the dissolution of a CO_2 phase in brine, where the unstable flow behavior leads to enhanced dissolution. The system is formulated in non-dimensional form with the Raleigh number Ra as the only governing parameter. For further details, we refer to [4], where we introduce the problem in a more detailed fashion and describe our decoupled solution approach based on an operator splitting for the pressure and the concentration parts. In the following, we focus on the performance characteristics of the DG scheme used for the discretization of the transport equation and (optionally, instead of a finite volume scheme) the pressure equation.

DG methods are popular in the porous media flow community due to their local mass conservation properties, the ability to handle full diffusion tensors and unstructured, nonconforming meshes as well as the simple implementation of upwinding for convection dominated flows.

Due to the large computational effort per DOF, an efficient implementation of DG methods of intermediate and high order is crucial. In many situations it is possible to exploit the tensor product structure of the polynomial basis functions and the quadrature rules on cuboid elements by using sum factorization techniques. At each element the following three steps are performed: (i) evaluate the finite element function and gradient at quadrature points, (ii) evaluate PDE coefficients and geometric transformation at quadrature points, and (iii) evaluate the bilinear form for all test functions. The computational complexity of steps (i) and (iii) is reduced from $O(k^{2d})$, $k-1$ being the polynomial degree and d the space dimension, to $O(dk^{d+1})$ with the sum factorization technique, see [14, 15]. This can be implemented with matrix–matrix products, albeit with small matrix dimensions. For the face terms, the complexity is reduced from $O(k^{2d-1})$ to $O(3dk^d)$. For practical polynomial degrees, $k \leq 10$, the face terms dominate the overall computation time, resulting in the time

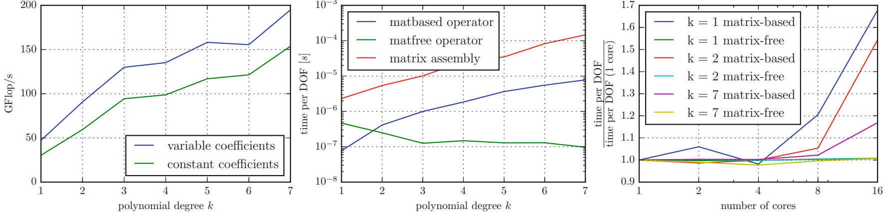

Fig. 1 Performance of the sum factorized DG assembly: GFlop/s rates for a matrix-free operator application *(left)*, time per DOF for matrix-based and matrix-free operator application as well as matrix assembly *(middle)*, relative overhead per DOF when weakly scaling from 1 to 16 cores *(right)*

per degree of freedom (DOF) to be independent of the polynomial degree. This can be seen in the second plot in Fig. 1, where the time per DOF for the matrix-free operator is almost constant starting from $k = 3$.

Our implementation of the DG scheme is based on exploiting the matrix–matrix product structure of the sum factorization kernels. We initially relied on compiler auto-vectorizers for the vectorization, but as can be seen in the results published in [4], this did not yield acceptable performance. We have thus reimplemented the kernels with explicit vectorization; for this purpose, we rely on the small external header library VCL [9] which provides thin wrappers around x86-64 intrinsics. In order to further improve and stabilize the performance of these kernels across different discretization orders k, we exploit the fact that our equation requires both the solution itself as well as its gradient, yielding a total of 4 scalar values per quadrature point, which fits perfectly with the 4-wide double precision SIMD registers of current CPUs, eliminating the need for complicated and costly data padding and or setup/tail loops. This scheme can be extended to wider architectures like AVX512 and GPUs by blocking multiple quadrature points together.

In the following, we present some results obtained with our new CPU implementation of this sum factorized DG scheme. For these benchmarks, we evaluate a stationary general convection diffusion reaction equation on the 3D unit cube. As many models assume the equation parameters to be constant within a single grid cell, our code has a special fast path that avoids parameter evaluation at each quadrature point, reducing the number of evaluations per cell from $O(k^d)$ to 1.

We performed our measurements on one CPU of a server with dual Xeon E5-2698v3 (Haswell-EP at 2.3 GHz, 16 cores, 32 hyper-threads, AVX2/FMA3, AVX clock 1.9 GHz, configured without TurboBoost and Cluster on Die, theoretical peak 486.4 GFlop/s) and 128 GB DDR4 DRAM at 2.13 GHz. All benchmarks were performed using thread pinning by first distributing hardware threads across the available cores before employing hyper-threading (if applicable). The same platform was used for all subsequent CPU benchmarks described in this paper except for Sect. 4.2. We investigated the scalability of our code within this UMA node according to our parallelization concept laid out in the introduction by means of a weak scalability study.

Figure 1 shows the overall GFlop/s rate achieved on all 16 cores during a complete matrix-free operator application for different polynomial degrees as well as the time required for these applications in relation to a matrix multiplication. As can be seen, the code already outperforms the matrix-based version for $k = 2$, without taking into account the large overhead of initial matrix assembly, which also becomes a severely limiting factor for the possible per-core problem sizes at larger core counts and higher discretization orders (for $k = 7$, we were only able to allocate 110,592 DOFs per core in order to stay within the available 4 GB RAM, which is already above average when considering current HPC systems). As can be seen, the larger amount of work per quadrature point in case of non-constant parameters directly translates into higher GFlop/s rates.

Finally, the third plot of Fig. 1 demonstrates the better scalability of the matrix-free scheme: it shows the relative overhead per DOF after weakly scaling to different numbers of active cores compared to the time per DOF required when running on a single core. While the computationally bound matrix-free scheme achieves almost perfect scalability, the matrix multiplication starts to saturate the 4 memory controllers of the CPU at between 4 and 8 active cores, causing a performance breakdown.

In order to gain further insight into the relative performance of the different assembly components, we instrumented our code to record separate timings and operation counts for the three parts of the sum factorized algorithm: calculation of the solutions at the quadrature points, per-quadrature point operations and the folding of the per-point integral arguments into the test function integrals. As can be seen in Fig. 2, the sum factorized kernels achieve very good GFlop/s rates due to their highly structured nature, especially for the 3D volume terms. In comparison, the face terms are slightly slower, which is due to both the lower dimension (less work per datum) and the additional work related to isolating the normal direction (the normal direction needs to be treated as the first/last direction in the sum factorization kernels, requiring an additional permutation step in most cases). While this step can be folded into the matrix multiplication, it creates a more complicated memory access pattern, reducing the available memory bandwidth due to less efficient prefetching, which is difficult to overcome as it involves scalar accesses spread over multiple cache lines. The lower amount of work also makes the face

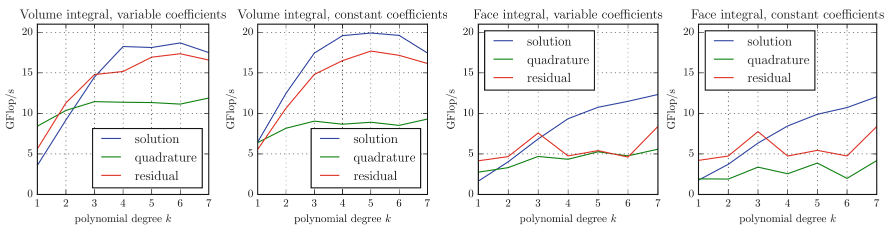

Fig. 2 GFlop/s rates for different parts of the sum factorized assembly. Rates are shown for a single core, benchmark was run with all 16 cores active

integrals more sensitive to the problem size, the residual calculation in particular hitting local performance peaks for $k = 3$ and $k = 7$, which translates into either 4 or 8 quadrature points per direction, exactly filling the 4-wide SIMD registers of the processor.

The calculations at the quadrature points do not achieve the same efficiency as the sum factorization, which is to be expected as they possess a more complex structure with e.g. parameter evaluations and (in the case of the face terms) branching due to the upwinding in the DG scheme. In order to improve performance in this area, we are currently investigating vectorization across multiple quadrature points.

3.3 Low Order Lagrange Methods

In contrast to the spectral DG methods, low order conforming methods have several major drawbacks regarding the possible performance:

(i) The memory layout is much less favorable – assembly is performed cell wise, but the DOFs are attached to vertices (and edges etc. for polynomial degrees >1), leading to scattered memory accesses. Moreover, vertex DOFs are shared between multiple cells, increasing the size of access halos and the probability of write conflicts compared to DG.

(ii) The algorithmic intensity is very low and performance thus memory bandwidth bound rather than compute bound. While structured meshes allow to calculate a lot of information on-the-fly, reducing the amount of expensive memory transfers and increasing computational intensity, many real world applications do require unstructured meshes to correctly model complex geometries or for adaptive computations. We limit the costs of these unstructured meshes by combining globally unstructured coarse meshes with several levels of locally structured refinement on each cell to recover a minimum amount of local structure.

In the DUNE-PDELAB interface, users of the library must implement a local operator that contains the cell- and face-based integration kernels for the global operator. Vectorization has to be added at the innermost level to these kernels, i.e., at the level of cell operations, which is user code that has to be rewritten for every new problem. In order to lessen this implementation burden on the user, our framework vectorizes the kernels over several mesh cells and replaces the data type of the local residual vector with a special data type representing a whole SIMD vector. In C++ this can be done generically by using vectorization libraries, e. g. Vc[12] or VCL[9], and generic programming techniques. With this approach, the scalar code written by the user is automatically vectorized, evaluating the kernel for multiple elements simultaneously. The approach is not completely transparent, as the user will have to e.g. adapt code containing conditional branches, but the majority of user code can stay unchanged and will afterwards work for the scalar and the vectorized case alike.

Table 2 Matrix-based assembly performance: Poisson problem, Q_1 elements, assembly of Jacobian. *Left:* Xeon E5-2698v3 (cf. Sect. 3.2). *Right:* Xeon PHI 5110P (Knights Corner, 60 cores, 240 hyper-threads, peak 1011 GFlop/s)

SIMD	Lanes	Threads	Runtime	GFlop/s	%peak
None	1	1	38.626 s	3.01	0.6
None	1	16	2.455 s	47.28	9.7
None	1	32	3.426 s	33.88	7.0
AVX	4	1	16.570 s	4.95	1.0
AVX	4	16	1.126 s	72.85	15.0
AVX	4	32	2.271 s	36.12	7.4

SIMD	Lanes	Threads	Runtime	GFlop/s	%peak
None	1	1	43.641 s	0.17	0.02
None	1	60	2.974 s	2.44	0.24
None	1	120	1.376 s	5.27	0.52
Vect.	8	1	12.403 s	0.58	0.06
Vect.	8	60	1.474 s	4.92	0.49
Vect.	8	120	1.104 s	6.57	0.65

Table 3 Matrix-free assembly performance: Poisson problem, Q_1 elements, 10 iterations of a matrix-free CG. *Left:* Xeon E5-2698v3 (cf. Sect. 3.2). *Right:* Xeon PHI 5110P

SIMD	Lanes	Thread	Runtime	GFlop/s	%peak
None	1	1	56.19 s	0.10	0.02
None	1	16	6.84 s	0.82	0.17
None	1	32	6.13 s	0.91	0.19
AVX	4	1	44.55 s	0.09	0.02
AVX	4	16	6.12 s	0.64	0.13
AVX	4	32	5.50 s	0.72	0.15

SIMD	Lanes	Threads	Runtime	GFlop/s	%peak
None	1	1	139.61 s	0.12	0.01
None	1	60	14.74 s	1.09	0.11
None	1	120	10.50 s	1.53	0.15
Vect.	8	1	61.23 s	0.26	0.03
Vect.	8	60	12.47 s	1.29	0.13
Vect.	8	120	9.22 s	1.75	0.17

To evaluate the potential of vectorized assembly on structured (sub-)meshes, we present initial tests results in Table 2. The first test problem uses a conforming FEM Q_1 discretization. We measure the time to assemble a global stiffness matrix using numerical differentiation. Three levels of sub-refinement are applied and vectorization is employed across neighboring subelements. For the Xeon Phi, we include timings for 1, 60 and 120 threads. Here, the most real-world configuration involves 120 threads, as each of the 60 cores requires at least two threads to achieve full utilization. We do not include measurements for 180/240 threads, as our kernels saturate the cores at two threads per core and additional threads fail to provide further speedups. On the CPU we obtain a significant portion of the peak performance, in particular for low numbers of threads with less memory bandwidth pressure.

These results get worse if we switch to operations with lower algorithmic intensity or to many-core systems like the Xeon Phi 5110P. This is illustrated in the second example, where we consider the same problem but use a matrix free operator within an unpreconditioned CG solver, see Table 3. For such low order methods we expect this operation to be totally memory bound. In this case our benchmarks only show a very small speedup. This is in part due to bad SIMD utilization (cf. the single core results), but also due to the unstructured memory accesses, which are even more problematic on Xeon Phi due to its in-order architecture that precludes efficient latency hiding apart from its round-robin scheduling to multiple hardware

threads per core; as a result, we are currently not able to leverage the performance potential of its wider SIMD architecture.

The scattered data issues can be reduced by employing special data structures for the mesh representation and the local stiffness matrix. Regarding the algorithmic intensity we expect a combination with strong smoothers, see Sect. 4.2, to improve the overall performance.

We will now discuss our modifications to the original data layout and data structures within PDELab aimed at making the data more streaming friendly. As general unstructured meshes are not suited for streaming and vectorization, we introduce a new layer, which we refer to as a *patch*. A patch represents a subset of an unstructured mesh with a local structured refinement, which is constructed on the fly and only used during assembly, which allows for a data layout which is well suited to accelerator units.

Each patch consists of a set of macro elements, made up of a number of elements extracted from the initial, unstructured mesh. We restrict ourselves to one type of macro element (i.e. either simplex or hypercube) per patch. In mixed type meshes a higher-level abstraction layer is expected to group the elements accordingly. This enables us to vectorize assembly across multiple macro elements of the patch. The macro elements are sub-refined on the fly in a structured way to a given level. For vectorized assembly, all lanes deal with corresponding subelements of different macro elements at the same time.

In the host mesh, a DOF may be associated with mesh entities of codimension > 0, which might form part of multiple patches. Thus, care must be taken to avoid data races when writing to global data structures. We circumvent this problem on the level of patches by provisioning memory for shared DOFs per patch macro element, enabling us to optimize the per-patch memory layout for vectorized access. Figure 3 illustrates the mapping of DOFs between global and per-patch storage. When preparing the assembly of a residual on a patch, the DOFs in the global coefficients vector are copied to a dedicated per-patch vector, and after the per-patch assembly we then need to accumulate the assembled data back into the layout imposed by the host mesh. While doing so we need to accumulate partial data for shared DOFs, taking care not to introduce races. The same issues and solution apply to Jacobian assembly.

This design trades increased data size for better access patterns, which creates its own set of trade-offs. In the case of the vertex coordinates used to describe the patch this should not have a big impact, because we apply virtual sub-refinement and the amount of storage for vertex coordinates should be much less than the amount of storage used for coefficient vectors and Jacobian matrices. In the case of coefficient vectors and Jacobian matrices we benefit not only from the improved access patterns, but also from the isolation of the per-patch data from the global state, reducing the need for locking or similar schemes to the patch setup/teardown layer.

The underlying DUNE interfaces, in particular the unstructured mesh, do not know about the virtual refinement. To allow reuse of existing components, we further provide a particular shape function implementation, which describes a

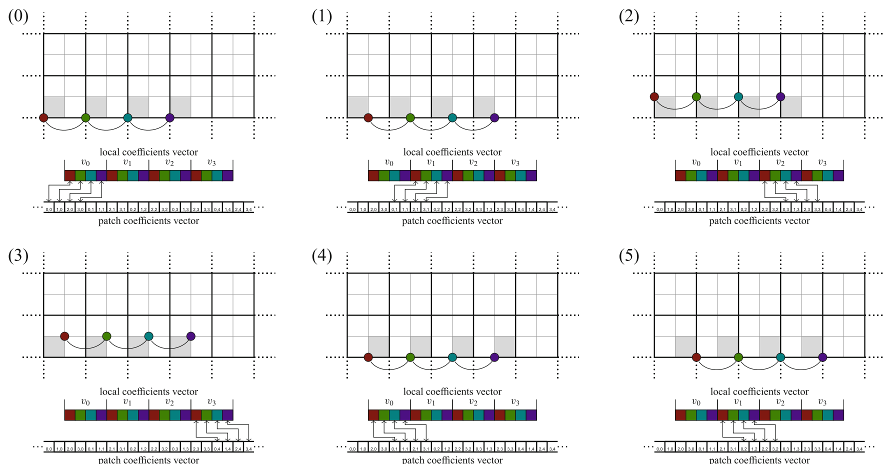

Fig. 3 Vectorized assembly for low order Lagrange discretization. Looping through consecutive elements in parallel and computing the local contributions for each local DOF. Each macro element is refined into 4 sub-cells and has 9 entries in the patch coefficients vector. Consecutive macro-cells are stored interleaved as vectors of doubles. This allows for fully automatic vectorization

refined Q_1 basis. This basis is used to encapsulate the additional intricacies of the virtual refinement layer and allows for re-use of existing DUNE components for visualization etc.

4 Linear Algebra

As laid out in the beginning, we are convinced that the problems from our domain of interest (porous media) will require a mix of matrix-free and matrix-based methods to be solved efficiently. The linear algebra part of these calculations will typically be memory bound, which makes it attractive to support moving these parts to accelerators and exploit their high memory bandwidth. In the following, we present some of our efforts in this direction.

4.1 Efficient Matrix Format for Higher Order DG

Designing efficient implementations and realizations of solvers effectively boils down to (i) a suitable choice of data structures for sparse matrix–vector multiply, and (ii) numerical components of the solver, i.e., preconditioners.

DUNE's initial matrix format, (block) compressed row storage, is ill-suited for modern hardware and SIMD, as there is no way to efficiently and generally expose

a block structure that fits the size of the SIMD units. We have thus extended the SELL-C-σ matrix format introduced in [13] which is a tuned variant of the sorted ELL format known from GPUs, to be able to efficiently handle block structures [16].

As we mostly focus on solvers for DG discretizations, which lend themselves to block-structured matrices, this is a valid and generalizable decision. The standard approach of requiring matrix block sizes that are multiples of the SIMD size is not applicable in our case because the matrix block size is a direct consequence of the chosen discretization. In order to support arbitrary block sizes, we interleave the data from N matrix blocks given a SIMD unit of size N, an approach introduced in [6]. This allows us to easily vectorize existing scalar algorithms by having them operate on multiple blocks in parallel, an approach that works as long as there are no data-dependent branches in the original algorithm. Sparse linear algebra is typically memory bandwidth bound, and thus, the main advantage of the block format is the reduced number of column block indices that need to be stored (as only a single index is required per block). With growing block size, this bandwidth advantage quickly approaches 50 % of the overall required bandwidth.

So far, we have implemented the SELL-C-σ building blocks (vectors, matrices), and a (block) Jacobi preconditioner which fully inverts the corresponding subsystem; for all target architectures (CPU, MIC, CUDA). Moreover, there is an implementation of the blocked version for multi-threaded CPUs and MICs. While the GPU version is implemented as a set of CUDA kernels, we have not used any intrinsics for the standard CPU and the MIC – instead we rely on the auto-vectorization features of modern compilers without performance penalty [16]. Due to the abstract interfaces in our solver packages, all other components like the iterative solvers can work with the new data format without any changes. Finally, a new backend for our high-level PDE discretization package enables a direct assembly into the new containers, avoiding the overhead of a separate conversion step. Consequently, users can transparently benefit from our improvements through a simple C++ typedef.

We demonstrate the benefits of our approach for a linear system generated by a 3D stationary diffusion problem on the unit cube with unit permeability, discretized using a weighted SIPG DG scheme [8]. Timings of 100 iterations of a CG solver using a (block) Jacobi preconditioner on a Xeon E5-2698v3 (cf. Sect. 3.2, no hyperthreading), on a NVIDIA Tesla C2070 for the GPU measurements and on an Intel Xeon Phi 7120P, are presented in Fig. 4, normalized per iteration and DOF.

As can be seen, switching from MPI to threading affords moderate improvements due to the better surface-to-volume ratio of the threading approach, but we cannot expect very large gains because the required memory bandwidth is essentially identical. Accordingly, switching to the blocked SELL-C-σ format consistently yields good improvements due to the lower number of column indices that need to be loaded, an effect that becomes more pronounced as the polynomial degree grows due to larger matrix block sizes. Finally, the GPU and the MIC provide a further speedup of 2.5–5 as is to be expected given the relative theoretical peak memory bandwidth figures of the respective architectures, demonstrating that our

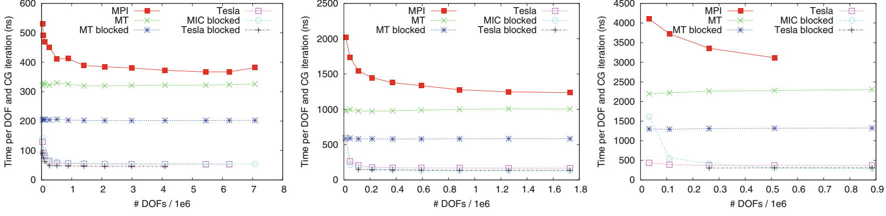

Fig. 4 Normalized execution time of the (block) Jacobi preconditioned CG solver for polynomial degrees $p = 1, 2, 3$ (*left* to *right*) of the DG discretization. The multithreaded (MT) and MIC versions use a SIMD block size of 8. The Tesla GPU versions use a SIMD block size of 32. Missing data points indicate insufficient memory

code manages to attain a constant fraction of the theoretically available memory bandwidth across all target architectures.

4.2 GPU Accelerated Preconditioners and Strong Smoothers

The promising results from enhancing the sparse matrix–vector multiply (SpMV) and therefore the whole DUNE-ISTL in DUNE by using the SELL-c-σ and BELL-c-σ storage formats lead to the idea of using this kernel in the linear solver more heavily by employing sparse approximate inverse preconditioners. Preconditioning with approximate inverses means direct application of a $M \approx A^{-1}$ that is, left-multiplying with a preassembled (sparse) preconditioner that approximates the inverse of the matrix A when solving $A\mathbf{x} = \mathbf{b}$. One potent representative of this family of preconditioners is the Sparse Approximate Inverse (SPAI) Algorithm initially proposed by Grote and Huckle [11] and recently applied very successfully in smoothers for Finite Element Multigrid solvers on the GPU within the FEAST software family [10]. The SPAI algorithm can briefly be described as follows:

$$\| I - M_{\text{SPAI}} A \|_F^2 \quad = \quad \sum_{k=1}^{n} \| \mathbf{e}_k^{\mathrm{T}} - \mathbf{m}_k^{\mathrm{T}} A \|_2^2 \quad = \quad \sum_{k=1}^{n} \| A^{\mathrm{T}} \mathbf{m}_k - \mathbf{e}_k \|_2^2$$

where $\mathbf{e}_k$ is the k-th unit-vector and $\mathbf{m}_k$ is the k-th column of M_{SPAI}. Therefore it follows that for n columns of M we solve n independent and small *least-squares* optimization problems to construct $M = [\mathbf{m}_1, \mathbf{m}_2, \ldots \mathbf{m}_n]$:

$$\min_{m_k} \| A^{\mathrm{T}} \mathbf{m}_k - \mathbf{e}_k \|_2, \quad k = 1, \ldots n \, .$$

The resulting M_{SPAI} can then typically be used to accelerate a Richardson Iteration,

$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \omega M_{\text{SPAI}} (\mathbf{b} - A\mathbf{x}^k)$$

which can be employed as a stronger smoother in a multigrid scheme, or alternatively be applied directly without the cost of the additional defect correction. Typical variants of the SPAI procedure restrict the fill-in within the assembly to the main diagonal (SPAI(0)) or to the non-zero pattern of the system matrix A (SPAI(1)). It has been reported that SPAI(0) has approximately the same smoothing properties as an *optimally* damped Jacobi, while SPAI(1) can be compared to Gauß-Seidel [5]. In this paper, we use SPAI(1) in our benchmarks. However, stronger preconditioning based on such techniques needs discussion, in particular with regard to performance relative to simple preconditioning such as Jacobi. Although both kernels ($\omega M_{\text{JAC}}\mathbf{d}$ and $M\mathbf{d}$ with the former representing a component-wise vector multiply and the latter an SpMV with the approximate inverse) are generally memory bound, the computational complexity of the SPAI preconditioner application depends on the sparsity pattern of M_{SPAI} and the memory access patterns imposed by the sparse matrix storage on the respective hardware. The Jacobi preconditioner comes at significantly lower cost and can be executed many times before reaching the computational cost of a single SPAI application. In addition, the performance gain through higher global convergence rates offered by SPAI must amortize the assembly of M_{SPAI}, which is still an open problem especially considering GPU acceleration (also being addressed within EXA-DUNE but not yet covered by this paper). On the other hand, with SPAI offering a numerical quality similar to Gauß-Seidel there is justified hope that in combination with well-optimized SpMV kernels based on the SELL-c-σ and BELL-c-σ storage formats a better overall solver performance can be achieved (also compared to even harder to parallelize alternatives such as ILU). In addition, the effectiveness for the Jacobi preconditioning depends on a good choice of ω, while SPAI is more robust in this regard.

In order to show that the SPAI preconditioner can be beneficial, we compare the overall performance of a Conjugate Gradient solver, preconditioned with SPAI(1) and Jacobi (with different values for ω) respectively. Here, we adapt an existing example program from DUNE-PDELAB that solves a stationary diffusion problem:

$$\nabla \cdot (K\nabla u) = f \text{ in } \Omega \subset \mathbb{R}^3$$
$$u = g \text{ on } \Gamma = \partial\overline{\Omega}$$

with $f = (6 - 4|x|^2)\exp(-|x|^2)$ and $g = \exp(-|x|^2)$, discretized with the same SIPG DG scheme [8] already used in Sect. 4.1. We restrict our experiments to the unit cube $\Omega = (0, 1)^3$ and unit permeability $K = 1$.

From the construction kit that comes with a fast SpMV on the GPU and a kernel to preassemble the global SPAI matrix in DUNE-ISTL, three types of preconditioners are directly made possible: a standard scalar Jacobi preconditioner,

$$S^{\omega}_{\text{JAC}} : \mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \omega M_{\text{JAC}}(\mathbf{b} - A\mathbf{x}^k), k = 1, \ldots, K \tag{3}$$

with M_{JAC} as defined above and a fixed number of iterations K (note, that this is in order to describe how the preconditioner is applied to a vector x in the PCG solver and that this iteration solves a defect correction already and thus here, $\mathbf{b}$ is the global defect). In addition, we can also precompute the exact inverse of each logical DG block in the system matrix, making good use of the BELL storage by switching to a block Jacobi preconditioner:

$$S_{\text{BJAC}} : \mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + M_{\text{BJAC}}(\mathbf{b} - A\mathbf{x}^k), k = 1, \ldots, K \tag{4}$$

with $M_{\text{BJAC}} = \sum_i R_i^T A_i^{-1} R_i$ being the exact DG-block-inverse, precomputed by a LU decomposition (using cuBLAS on the GPU). Third, a direct application of the SPAI(1) matrix to the defect can be employed:

$$S_{\text{SPAI}} : \mathbf{x} \leftarrow M_{\text{SPAI}}\mathbf{x} \tag{5}$$

with M_{SPAI} as defined above. We use both the SELL-c-σ and BELL-c-σ storage formats in this case.

We perform all benchmarks on the GPU and the CPU: here, we make use of a Maxwell GPU in a NVIDIA GTX 980 Ti consumer card with roughly 340 GB/s theoretical memory bandwidth. The Maxwell architecture of the 980 Ti is the same as in the (at the time of writing this paper) most recent iteration of the Tesla compute cards, the Tesla M40. For comparison, we use a 4-core Haswell CPU (Core i5 4690K) with 3.5 GHz (turbo: 3.9 GHz) and roughly 26 GB/s theoretical memory bandwidth.

First, we demonstrate the sensitivity of Jacobi preconditioning to damping in order to identify fair comparison configurations for the damping-free competitors Block-Jacobi and SPAI. Figure 5 shows the variation of the solver iterations depending on the damping parameter ω. We sample the parameter space in steps of

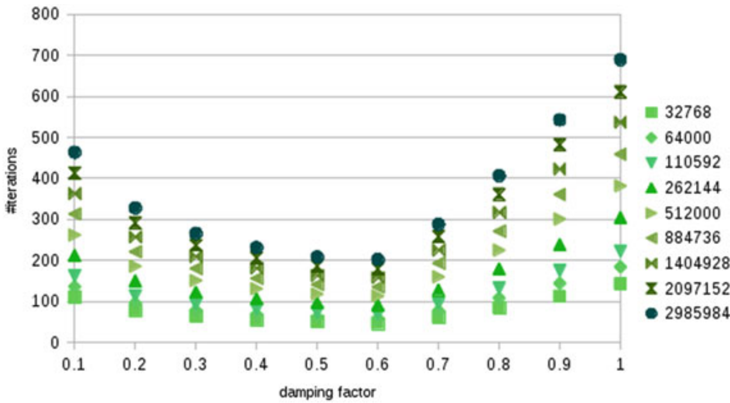

Fig. 5 Dependence of the Jacobi preconditioning on damping parameter

0.1 between 0 and 1. The measurements clearly show a ‘sweet spot’ around 0.6 and a worst case (expectedly) in the undamped case. Therefore, we employ $\omega = 0.6$ as an example of good damping and $\omega = 1.0$ as a bad damping coefficient. In addition, we consider a ‘median’ case of $\omega = 0.8$ in all following benchmarks. Note that in reality, ω is unknown prior to the solver run and has a huge impact on the overall performance which can be seen in the factor of more than 3 between the number of solver iterations for ‘good’ and ‘bad’ choices. In contrast to ω having an impact on the numerical quality of the preconditioner $S^{\omega}_{\mathrm{JAC}}$ only and not on its computational cost, the parameter K is somewhat more complicated to take into account in the performance modelling process: here, a larger value for K produces a numerical benefit, but also increases computational cost due to the additional defect correction with each additional iteration. In many cases, the numerical benefit of increasing K does not amortize the additional cost beyond a certain value K_{opt}, as can be seen in Fig. 6 for the Jacobi preconditioner and $\omega = 0.5$. For the benchmark problem at hand, it is always beneficial to perform only 2 iterations. Note that this behavior also depends on the damping parameter and more importantly, that K is also unknown a priori. This makes both ω and K subject to *autotuning* in preconditioners that try to solve an unknown correction equation, which is also a research topic of the upcoming EXA-DUNE phase two.

Figures 7 and 8 show the timing results and numbers of iterations for first and second order DG discretizations and the preconditioners defined by Eqs. 3 (with different parameters for ω) through 5, where for the latter we employed both the SELL and BELL (labeled BSPAI) matrix storage techniques.

The first thing to notice here is that for the $p = 1$ case, the SPAI and BSPAI variants cannot beat the inexact Block-Jacobi solves, due to a better overall convergence behavior of the latter, although they come close (within 5 %). However, the assembly must be considered more expensive for both SPAI versions of M (see below). For higher order Finite Elements, the SPAI and especially the BSPAI preconditioning can beat the best (Block-) Jacobi ones concerning overall solver wall clock time by generating a speedup of 1.5, which leaves up to 50 % of the solution time to amortize a pre-assembly of the sparse approximate inverse. Comparing the SELL-c-σ performance and the improved BELL variant thereof it becomes clear that the latter’s block awareness makes SPAI successful: using SELL,

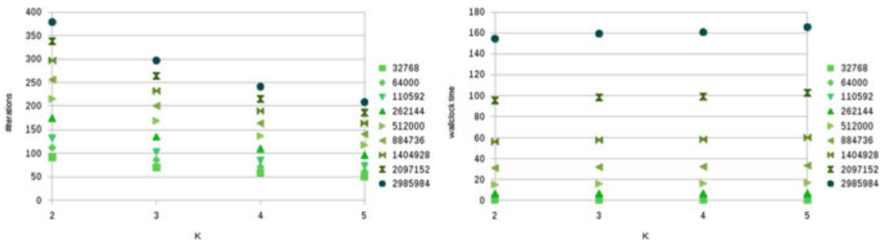

Fig. 6 K-Dependence of the Jacobi preconditioning. *Left*: Number of iterations. *Right*: Solver wall clock time

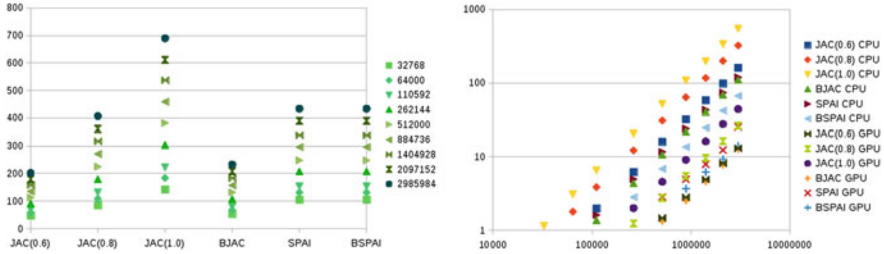

Fig. 7 Iteration count and wall clock times (logscale/logscale) of the PCG solver with different preconditioners for the benchmark problem using a first order DG discretization

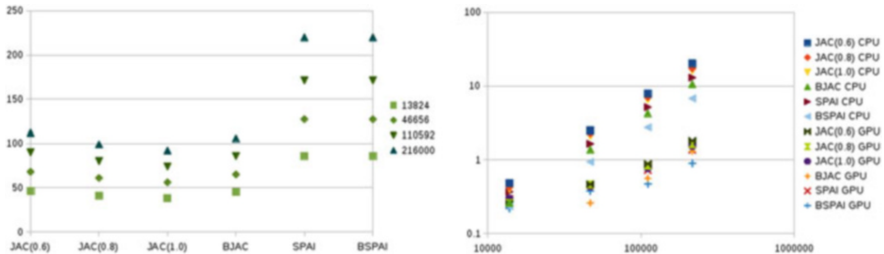

Fig. 8 Iteration count and wall clock times (logscale/logscale) of the PCG solver with different preconditioners for the benchmark problem using a second order DG discretization

SPAI is again only as good as Block-Jacobi. On the GPU, the solver performs 9.5 times better concerning wall clock time with the scalar SELL storage and 7.5 times with the BELL variant where in the former, the Haswell CPU can play out its sophisticated caches due to blocking. Here, the GPU cannot exploit the complete block structure due to the mapping of each row to one thread. Thus each thread can only exploit one row of each DG block. Both speedups are within good accordance of the factor between the theoretical memory bandwith of the respective architectures and a memory bound kernel.

Altogether, this shows that even for simple problems, the SPAI technique can be used to accelerate Krylov subspace solvers within DUNE especially for higher order Finite Elements. However, it must be stated that the overall feasibility of such Approximate Inverse techniques relies on being able to amortize the assembly time by means of faster application times of the preconditioner. In light of this, we are currently developing a GPU-based SPAI assembly based on fast QR decompositions with householder transforms on each column, which can be batched for execution similar to [17]. Also, SPAI(ϵ) (with more complex sparsity patterns for M) is being examined. Exploring the smoothing capabilities of SPAI-preconditioned iterations within DUNE's AMG schemes on the GPU is also currently under examination and expected to be finished within the remaining first phase of the EXA-DUNE project.

5 Outlook

The results presented in this contribution highlight some of the efforts of the first 2.5 years of the EXA-DUNE project. While these tools were developed mostly independently during that time, we intend to use the remaining 6 months of the project to integrate these tools into an initial demonstrator based on a porous media application. This demonstrator will combine the improved assembly performance and the faster linear algebra with a two-level preconditioner based on a matrix-free smoother for the DG level and an AMG-based subspace correction on a low order subspace, which we intend to combine with the multilevel methods and uncertainty quantification developed in parallel and detailed in [3].

Acknowledgements This research was funded by the DFG SPP 1648 *Software for Exascale Computing*.

References

1. Bastian, P., Blatt, M., Dedner, A., Engwer, C., Klöfkorn, R., Kornhuber, R., Ohlberger, M., Sander, O.: A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE. Computing **82**(2–3), 121–138 (2008)
2. Bastian, P., Blatt, M., Dedner, A., Engwer, C., Klöfkorn, R., Ohlberger, M., Sander, O.: A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework. Computing **82**(2–3), 103–119 (2008)
3. Bastian, P., Engwer, C., Fahlke, J., Geveler, M., Göddeke, D., Iliev, O., Ippisch, O., Milk, R., Mohring, J., Müthing, S., Ohlberger, M., Ribbrock, D., Turek, S.: Advances concerning multiscale methods and uncertainty quantification in EXA-DUNE. In: Proceedings of the SPPEXA Symposium 2016. Lecture Notes in Computational Science and Engineering. Springer (2016)
4. Bastian, P., Engwer, C., Göddeke, D., Iliev, O., Ippisch, O., Ohlberger, M., Turek, S., Fahlke, J., Kaulmann, S., Müthing, S., Ribbrock, D.: EXA-DUNE: flexible PDE solvers, numerical methods and applications. In: Lopes, L., et al. (eds.) Euro-Par 2014: Parallel Processing Workshops. Euro-Par 2014 International Workshops, Porto, 25–26 Aug 2014, Revised Selected Papers, Part II. Lecture Notes in Computer Science, vol. 8806, pp. 530–541. Springer (2014)
5. Bröker, O., Grote, M.J.: Sparse approximate inverse smoothers for geometric and algebraic multigrid. Appl. Numer. Math. **41**(1), 61–80 (2002)
6. Choi, J., Singh, A., Vuduc, R.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Principles and Practice of Parallel Programming, pp. 115–126. ACM, New York (2010)
7. Engwer, C., Fahlke, J.: Scalable hybrid parallelization strategies for the DUNE grid interface. In: Numerical Mathematics and Advanced Applications: Proceedings of ENUMATH 2013. Lecture Notes in Computational Science and Engineering, vol. 103, pp. 583–590. Springer (2014)
8. Ern, A., Stephansen, A., Zunino, P.: A discontinuous Galerkin method with weighted averages for advection-diffusion equations with locally small and anisotropic diffusivity. IMA J. Numer. Anal. **29**(2), 235–256 (2009)
9. Fog, A.: VCL vector class library, http://www.agner.org/optimize

10. Geveler, M., Ribbrock, D., Göddeke, D., Zajac, P., Turek, S.: Towards a complete FEM-based simulation toolkit on GPUs: unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses. Comput. Fluids **80**, 327–332 (2013)
11. Grote, M.J., Huckle, T.: Parallel preconditioning with sparse approximate inverses. SIAM J. Sci. Comput. **18**, 838–853 (1996)
12. Kretz, M., Lindenstruth, V.: Vc: A C++ library for explicit vectorization. Softw. Pract. Exp. **42**(11), 1409–1430 (2012)
13. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Bishop, A.R.: A unified sparse matrix data format for modern processors with wide SIMD units. SIAM J. Sci. Comput. **36**(5), C401–C423 (2014)
14. Kronbichler, M., Kormann, K.: A generic interface for parallel cell-based finite element operator application. Comput. Fluids **63**, 135–147 (2012)
15. Melenk, J.M., Gerdes, K., Schwab, C.: Fully discrete hp-finite elements: fast quadrature. Comput. Methods Appl. Mech. Eng. **190**(32–33), 4339–4364 (2001)
16. Müthing, S., Ribbrock, D., Göddeke, D.: Integrating multi-threading and accelerators into DUNE-ISTL. In: Numerical Mathematics and Advanced Applications: Proceedings of ENUMATH 2013. Lecture Notes in Computational Science and Engineering, vol. 103, pp. 601–609. Springer (2014)
17. Sawyer, W., Vanini, C., Fourestey, G., Popescu, R.: SPAI preconditioners for HPC applications. PAMM **12**(1), 651–652 (2012)
18. Turek, S., Göddeke, D., Becker, C., Buijssen, S., Wobker, S.: FEAST – realisation of hardware-oriented numerics for HPC simulations with finite elements. Concurr. Comput.: Pract. Exp. **22**(6), 2247–2265 (2010)