Vectorization of High-Order DG in Ateles for the NEC SX-ACE

Harald Klimach, Jiaxing Qi, Stephan Walter, and Sabine Roller

Abstract In this chapter, we investigate the possibilities of deploying a high-order, modal, discontinuous Galerkin scheme on the SX-ACE. Our implementation Ateles is written in modern Fortran and requires the new sxf03 compiler from NEC. It is based on an unstructured mesh representation that necessitates indirect addressing, but allows for a large flexibility in the representation of geometries. However, the degrees of freedom within the elements are stored in a rigid, structured array. For sufficiently high-order approximations these data structures within the elements can be exploited for vectorization.

1 Introduction

Memory has become the limiting factor in most computing systems for most computations. Both, processing speeds and memory access has exponentially increased during the development of computing technology, albeit with different paces. This development led to a gap between the memory and processing capabilities in modern devices [1]. The important factor describing this relation for numerical applications based on floating point numbers is the Byte to FLOP (floating point operation) ratio. It can be used to judge the suitability of a system for a given algorithm. Because on the one hand, the computing system is capable of providing a fixed Byte to FLOP ratio, while on the other hand, the algorithm requires a certain amount of data to be moved for each operation.

Besides the speed of the memory and the number of required transactions, another important factor is the size of the memory. In comparison to the processing speed the amount of available main memory in high-performance computing systems did not increase much over the last decade. The largest amount of total memory provided by a *Top500* system is 1.5 PetaBytes on the *Sequoia* IBM

H. Klimach (✉) • J. Qi • S. Roller
University of Siegen, Adolf-Reichwein Str. 2, 57076 Siegen, Germany
e-mail: harald.klimach@uni-siegen.de; jiaxing.qi@uni-siegen.de; sabine.roller@uni-siegen.de

S. Walter
Höchstleistungsrechenzentrum Stuttgart, Nobelstr. 19, 70569 Stuttgart, Germany
e-mail: walter@hlrs.de

M.M. Resch et al. (eds.), *Sustained Simulation Performance 2017*,
DOI 10.1007/978-3-319-66896-3_5

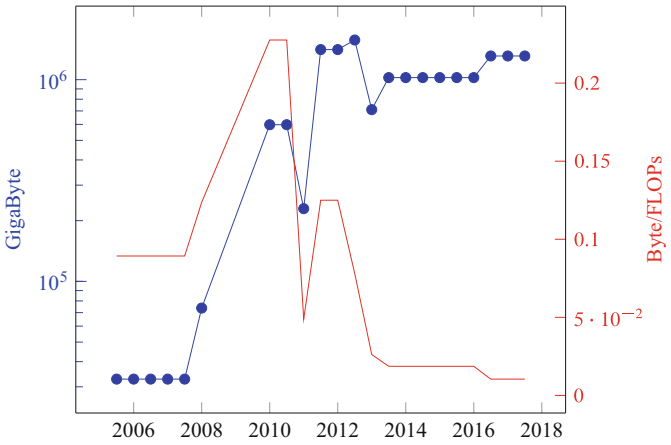

Fig. 1 Development of available memory in the top system of the *Top500* list over time. In *blue* the total available memory of the fastest system from the list at that point in time is shown. In *red* the ratio between that memory and the computing power of the system is in terms of floating point operations per second is indicated

BlueGene/Q installation at the Lawrence Livermore National Laboratory. This was the fastest system in terms of floating point operations per second in 2012. In June 2017 it was ranked fifth in the *Top500*.

Figure 1 illustrates the development of the fastest system in the *Top500* lists with respect to the available memory over time. The blue trend with the dot markers indicates the available overall memory in the fastest system on a logarithmic scale. To put this in context to the available computing power of the system, the red line indicates the ratio between available main memory and the number of floating point operations per second. We can see that the overall amount of memory, which allows us to solve larger or better resolved problems only grows slowly, and even as it grows it does not keep pace with the computing speed of the systems. Thus, we can observe that memory is a precious resource in modern computing systems, both in terms of speed and of size. Furthermore, we even expect increasing importance of the memory in the foreseeable future as with the current growth rates the gap between processing and memory speeds will continue to grow.

Table 1 provides an overview to the memory properties of contemporary HPC architectures in relation to their floating point operation speed. The first column indicates the system, the second the memory bandwidth in Bytes per second divided by the floating point operations per second. In the last column the available amount of main memory, again divided by the number of floating point operations per second. This is the measure for which the development over time for the fastest system in the *Top500* is given in Fig. 1. The issues we face with the big amounts of data produced in large scale simulations only get worse when we actually want to store results. Storage devices are even slower than main memory and when considering time dependent data, we often need to store several snapshots of the overall main memory used by the simulation.

Table 1 Contemporary HPC-systems with respect to their memory size and speed compared to their floating point operations per second

System	Per FLOPS memory-	
	Bandwidth	Size
NEC SX-ACE	1.000	0.250
K Computer (SPARC64 VIIIfx)	0.500	0.032
Sequoia (IBM BlueGene Q)	0.208	0.078
Sunway TaihuLight (SW26010)	0.178	0.010
nVidia Tesla P100	0.138	0.003
Hazel Hen (Intel Xeon E5-2680 v3)	0.141	0.133

We see that the memory is slow and small when compared to the computing power in terms of performed operations. Therefore, an important criterion for numerical schemes to be deployed on such modern large-scale computing systems is their ability to provide good approximations with as little amount of memory as possible.

The discontinuous Galerkin (DG) scheme is a promising numerical method that enables us to move into the desired direction of reduced memory consumption for solutions of partial differential equations. It employs a discretization of the simulation domain by a mesh, where the solution within each element of the mesh is approximated by a local function. A typical choice for the functions to use in this approximation are polynomial series. The usage of functions to represent the solution allows for high-order representations, as the scheme works for arbitrary numbers of terms in the deployed functions. High-order approximations have the advantage that they can approximate smooth functions with few degrees of freedom, due to the exponential convergence with increasing number of modes. Thus, the scheme requires only a minimal amount of data to represent the solution in the elements. Interaction between elements is realized by fluxes like in finite volume schemes. The discontinuous Galerkin scheme, thereby, offers a combination of aspects from the finite volume method and spectral discretizations. It provides to some extend the efficiency of spectral methods and at the same time some of the flexibility offered by finite volume methods.

From the numerical side the discontinuous Galerkin scheme appears to provide suitable characteristics to address the growing imbalance between memory and processing power of modern computing systems. On the side of computing architectures, the NEC SX-ACE is a vector system that offers some nice capabilities for numerical schemes with a focus on good memory performance. It offers a high Byte per FLOP ratio of 1 (256 GFLOP and 256 GB per second) with access to 16 GB of main memory per core with this speed when using all 4 cores of the processor. If this Byte to FLOP ratio is insufficient for an algorithm, it can be increased up to 4 Bytes per FLOP by employing fewer cores of the processor in the computation. As can be seen in Table 1 this is at the high end of this ratio for contemporary HPC architectures.

Because of these properties, we believe the discontinuous Galerkin method and the *NEC SX-ACE* architecture are a good match for large-scale simulations. The one provides an option to reduce the memory usage and the other attempts to provide a high data rate to allow a wider set of applications to achieve a high sustained performance. However, an obstacle we face in typical applications is the need for a great deal of flexibility and dynamic behavior during the runtime of the simulation. This often does not fit too well with the more rigid requirements for efficient computations on vector systems. Here we want to lay out, how the discontinuous Galerkin scheme with a sufficiently high order may be used to combine the flexibility required by the application with the vectorized computation on the NEC SX-ACE. This possibility is opened by the two levels of computation present in the discontinuous Galerkin scheme, where we can find high flexibility on the level of the mesh, but a highly structured and rigid layout within the elements. We believe, that it is a feasible option to use vectorization within elements of high-order discontinuous Galerkin schemes, while maintaining the large flexibility, offered by the method on the mesh level. Such a strategy opens the possibility to combine dynamic and adaptive simulations with the requirements of vectorized computing, which is increasingly important also on other architectures than the NEC SX-ACE.

In the following we briefly introduce the high-order discontinuous Galerkin scheme implemented in our solver *Ateles*. Then we go on with the presentation of the vectorization approach of the scheme on the NEC SX-ACE in Sect. 3 and conclude this chapter with some measurements and observations in Sect. 4.

2 High-Order Discontinuous Galerkin in Ateles

The discontinuous Galerkin method is especially well suited for conservation laws of the form:

$$\frac{\partial u}{\partial t} + \nabla f(u) = g \tag{1}$$

To find a solution to (1), the overall domain to be investigated is split into finite elements Ω_i and the solution is approximated by a function u_h within each of these elements. The equation is then multiplied with test functions ϕ to create a system that can be solved and after integration by parts we obtain:

$$\frac{\partial}{\partial t}\int_{\Omega_i} u_h \phi \mathrm{d}V - \int_{\Omega_i} f(u_h)\nabla\phi \mathrm{d}V + \int_{\partial\Omega_i} f^* \phi \mathrm{d}S = \int_{\Omega_i} g\phi \mathrm{d}V \tag{2}$$

From the integration by parts we get the surface integral where the new term f^* is introduced. This is a numerical flux that ties together adjacent elements as it requires the state from both sides of the surface. In a numerical discretization the employed

function spaces for the solution and the test functions need to be finite and Eq. (2) then provides an algebraic system in space, where the products of the functions can be written in matrices. Especially we get the mass matrix:

$$M = \int \psi \phi \mathrm{d}V \tag{3}$$

and the stiffness matrix:

$$S = \int \psi \nabla \phi \mathrm{d}V \tag{4}$$

2.1 The Modal Basis

Ateles implements the discontinuous Galerkin scheme with Legendre polynomials as a basis to represent the solution u in cubical elements. The Legendre polynomials can be defined recursively by:

$$\begin{aligned} L_0(x) &= 1, \quad L_1(x) = x \\ L_k(x) &= \frac{2k-1}{k} \cdot x \cdot L_{k-1}(x) - \frac{k-1}{k} L_{k-2}(x) \end{aligned} \tag{5}$$

They are defined on the reference interval $[-1, 1]$ and have some favorable properties. Most importantly they build an orthogonal basis with respect to the inner product with a weight of 1 over this interval. Another nice property is that all Legendre polynomials except for $L_0(x)$ are integral mean-free. Our solution within the elements of the discontinuous Galerkin scheme are obtained in the form of a series of Legendre polynomials:

$$u(x) = \sum_{k=0}^{m} c_k L_k(x) \tag{6}$$

Here, the coefficients c_k are the (Legendre) modes that describe the actual shape of the solution. The maximal polynomial degree in this series is denoted by m. Its choice determines the spatial convergence order of the scheme and the degrees of freedom (modes) required to represent the solution in the element $(m + 1)$. To represent the solution in three-dimensional space, we build a tensor product of the one-dimensional polynomials. By introducing the multi-index $\alpha = (i, j, k)$, we can denote the three-dimensional solution by:

$$u(x, y, z) = \sum_{\alpha=(0,0,0)}^{(m,m,m)} c_\alpha L_i(x) L_j(y) L_k(z) \tag{7}$$

With this definition for the solution in d dimensions, we get $(m+1)^d$ degrees of freedom. The layout of this data is highly structured, as we need a simple array with $(m+1) \times (m+1) \times (m+1)$ entries to store the c_α in three dimensions for example.

The orthogonality of the Legendre polynomials enables a fast computation of the mass matrix and its inverse, and their recursive nature enables a fast application of the stiffness matrix.

2.2 The Mesh Structure

The local discretization by polynomial series as described above is done locally in elements that are then combined in a mesh to cover the complete computational domain. *Ateles* employs an octree topology to construct this mesh of cubical elements with an unstructured layout. The unstructured organization requires an explicit description of elements to be considered but allows for a greater flexibility in describing arbitrary geometrical setups. By relying on an octree structure, large parts of the topological information is implicitly known and does not have to be explicitly stored or referred to. This is especially of an advantage for distributed parallel computations, as most neighbor information can be computed locally with a minimal amount of data exchange. With the choice of cubical elements, we can employ an efficient dimension by dimension approach and avoid the need for complex transformations. Boundaries are then implemented by penalizing terms inside the elements, very similar to approaches found in spectral discretizations. These allow for the approximation of the geometry with the same order as the one used for the representation of the scheme.

3 Vectorization on the NEC SX-ACE

Vector instructions mean that we perform the same instruction to many data concurrently. This single instruction, multiple data (SIMD) concept is becoming more and more important also on traditional scalar systems, as can be seen in the increasing register lengths of the AVX instructions in Intels x86 architecture. The NEC SX-ACE as a traditional vector computing system offers long vector data registers that hold 256 double precision real numbers and can perform one instruction on all of them simultaneously. From the algorithmic point we need long loops with independent iterations to utilize this mechanism.

In simulations that involve meshes, we usually need to perform the same operations for each mesh element, and we have many mesh elements for detailed simulations. Thus, an obvious choice for vectorization is here the loop over elements of the mesh. However, for high-order schemes this is not so straight forward anymore. For one, there are fewer elements used in the discretization, and maybe even more important, the computation for each element gets more involved. The

greatest problem for an efficient vectorization over the elements, however, is the desire for flexibility on the level of the mesh. As described above, we use an unstructured mesh description to enable an efficient approximation of arbitrary geometries. This introduces an indirection, which is in turn hurting the performance, as the vector data needs to be gathered and scattered when moved between memory and registers. Even more flexibility is required on the mesh level, when we allow hp-adaptivity, that is dynamic mesh adaptation to the solution and a variation in the polynomial degree from element to element. These features are desirable, because they minimize the computational effort in terms of memory and operations.

With this large degree of flexibility and unstructured data access across the elements of the mesh, a vectorized computation appears hard to achieve. Instead we look here into the vectorization within elements. As described in Sect. 2 the data within elements is highly structured and the operations we need to perform on it also nicely fits into SIMD schemes for a large part. One of the main computational tasks is the application of the stiffness- and mass-matrices. Such matrix-vector multiplications can be perfectly performed in vector operations. Other numerical tasks within the elements often follow a similar scheme and require the application of one operation to all degrees of freedom. The main limitation we face with an approach of vectorization within elements is the limited vector length. However, the vector length grows with the polynomial degree, opening the possibility to fully exploit even long vector registers, if the polynomial degree is only sufficiently high.

Most operations in *Ateles* need to be done on the polynomials in one direction, leaving the other dimensions open for concurrent execution. Thus, when the solution in a three-dimensional element is approximated by a maximal polynomial of degree m, there are $(m+1)^3$ degrees of freedom in total, and in most operations $(m+1)^2$ independent computations with the same instruction need to be performed. With this quadratic growth over the polynomial degree, $m = 15$ is already sufficient to fill the vector data registers with a length of 256 for the most important parts of the implementation. For the high-order discretization in *Ateles* we aim for polynomial degrees greater than 10, and for linear equations even for polynomial degrees in the range of 100. With this range of scheme orders, a vectorization within elements appears suitable and meaningful, even for such long vectors as found in the NEC SX-ACE.

The use of polynomials of high degree to represent the solution, thereby enables us to combine the flexibility of mesh adaptivity and unstructured meshes with efficient vector computations.

3.1 Porting of Ateles

Ateles is implemented in modern *Fortran* and utilizes some features from the *Fortran 2003* standard. Unfortunately, the existing Fortran compiler from older SX systems did not provide all the required features and was unable to compile *Ateles*. But NEC has implemented a new compiler for the SX-ACE, which supports the

complete *Fortran 2003* standard. This new compiler *sxf03* was able to compile *Ateles* and create a working executable for the SX-ACE, with surprisingly little effort. Yet, as this is a new compiler, not all optimizations from the old compilers where initially available and after the first porting, we ran into a vectorization issue with one of the loops, that was nicely vectorized by the old compiler, but not by the new *sxf03*. Because compiled files from the old and the new compiler could not be combined, the work on further optimization stalled at that point. A little more on these first porting issues can be found in [2] from last year, where also some more explanations on the porting of the APES suite in general are provided. After this issue was fixed in the compiler by NEC, we were now able to further look into the vectorization of *Ateles* and how the vectorization strategy within elements works out. In the following we report on the progress of this effort.

4 Measurements and Observations

To compile *Ateles* for the NEC SX-ACE in this report, we make use of the *sxf03* compiler in version "*Rev.050 2017/01/06*". As explained in Sect. 3, we are concerned with the operations within elements, and most of those resemble matrix-vector operations or are quite similar to them. One major distinction can be drawn depending on the kind of equation system that we need to solve. For linear equations we can perform all numerics in modal space, directly using the terms from the polynomial series, as introduced in Eq. (6). When dealing with nonlinear equations this is not so easily possible anymore. Instead, we transform the representation into physical space to obtain values at specific points, perform the nonlinear operation in each point and then transform the new values back into modal representation again. These transformations need to be done additionally and are quite expensive. The performance characteristics of the two cases are accordingly largely different in these two cases.

4.1 Linear Equations

Let us first look at linear equations, as their building blocks are also relevant for the nonlinear equations. As a representative for linear equations we look into the Maxwell equations for electrodynamics. We use a simple case without boundary conditions and polynomials of degree 11. All computations are done on a single core of the SX-ACE. In our first setup we used 64 elements, and found a really poor performance of only 26 MFLOPS in the most expensive routine according to the *ftrace* analysis. The crucial loop of that routine is shown in Listing 1 and we would expect this to nicely vectorize the inner, collapsed loop. Indeed, the extremely poor performance was due to the number of elements, as this is the first index here, and we end up with a strided access, according to the number of elements. When

Listing 1 Main loop of the volume to face projection

```
do iAnsZ=1,m+1,2
  !collapsed loop
  do iVEF=1,6*nElems*(m+1)**2
    ! indices actually computed from iVEF
    facestate(iElem,facepos,iVar,side) &
      & = facestate(iElem,facepos,iVar,side) &
      & + volstate(ielem,pos,iVar)
  end do
end do
```

Table 2 Excerpt from the tracing of *Ateles* for Maxwell equations and a discretization with polynomials of degree 11

Procedure	%	MFLOPS	V.OP %	V.LEN	Bank Conf.		ADB %
					CPU	Net	
VolToFace	28.7	929.5	99.12	204.6	0.137	0.531	86.07
PrjFlux2	10.3	1475.0	99.53	83.6	0.802	1.742	79.04
PrjFlux1	10.3	1479.4	99.53	83.2	0.685	1.294	71.65
PrjFlux3	10.2	1490.8	99.57	83.4	0.381	1.752	76.73
MaxFlux	9.9	513.3	79.88	38.7	0.057	0.321	63.90
MassMat	9.3	1906.7	87.69	63.0	0.503	18.491	45.46
PhysFlux	5.5	0.2	94.54	241.3	0.941	7.973	0.00

The first column states the measured routine, the second the running time percentage of the routine, the third the observed MFLOPS, the fourth is the vector operation ratio as a percentage (time spent on vector instructions to the time spent in total on that routine) and the fifth column provides the average vector length used in the vector instructions. The next two columns (6 and 7) provide the time spent on conflicts when accessing memory banks. In the eighth and last column, the ADB hit rate is given. Shown are the main procedures contributing to the overall compute time

changing to the element count to 63, the performance indeed increases from 26 to more than 900 MFLOPS. It appears that strides at multiples of 64 result in extremely bad performance, due to conflicts in the memory bank accesses.

Table 2 shows the most important routines for a run with polynomials of degree 11 and 63 elements. The main routines that contribute more than 84% to the overall compute time are the projection of the polynomials in the volume to the faces of the elements (*VolToFace*), the projection of the fluxes onto the testfunctions (*PrjFlux1*, *PrjFlux2* and *PrjFlux3*), the actual computation of the Maxwell flux (*MaxFlux*), multiplication with the inverse of the mass matrix (*MassMat*) and computation of the physical flux for the Maxwell equations (*PhysFlux*). Here, the projection of the flux onto testfunctions is actually the same operation that needs to be performed, albeit in three different directions and there is an individual implementation for each direction. Their only distinction is a different striding in the access to the three dimensional data.

As can be seen in Table 2, the volume to face projection (*VolToFace*) and the projection of the physical flux on the testfunctions are the main consumers of the computing time in this run with polynomials of degree 11. Both contribute about 30% to the overall running time. It also can be seen that already this run without tuning, provides relatively good vectorization properties with vectorization rates above 99%. Nevertheless, the computational efficiency is not quite high and we see for the *VolToFace* routine less than one GFLOPS. But this may also be due to the relatively low computational density in this operation. What needs to be done is just the summation of the degrees of freedom in one space direction. This single addition for each real number does not allow us to fully exploit the functional units of the processor.

One improvement that can be done in this routine is the simultaneous computation of the left and right faces in the given direction of the element. This improves the computational density as the volume data only needs to be loaded once for both sides, and we obtain between 1262 and 1462 MFLOPS (depending on the striding for the different directions). After this change, the projection of the physical fluxes becomes the most time consuming part. When we double the degrees of freedom and use polynomials of degree 23, this changes again and the multiplication with the inverse of the mass-matrix becomes the most important routine. Computing the multiplication with the inverse of the mass-matrix makes use of a short recursion, as with the recursive definition of the polynomial basis, already computed values can be reused. While this is computational efficient in terms of saving operations, it makes it harder to achieve good vectorization and a high sustained performance. Yet, for high orders and when avoiding bad striding we are capable to achieve already reasonable performance and before looking into this common part in more detail we now looked into other equations.

Unsurprisingly the acoustic and linearized Euler equations showed a very similar behavior. However, we found an excessive use of flux functions there to be an issue. This is already a little bit visible in Table 2 for the *PhysFlux* routine. The problem with that routine is that it is very small and used to compute the flux for just a single mode. Similarly this was found for the other linear equations, but there it the flux computation consumed a larger fraction of the overall compute time, and the effect was more pronounced. The remedy is fairly simple, though, as the loop over the modes can be pulled into this routine quite easily.

Another linear equation we have implemented in *Ateles* are locally linearized Euler equations. These use a linearization within elements, but nonlinear fluxes on the element faces for the exchange between elements. With those we ran again into the striding issue with the number of elements. Further investigation revealed that this striding indeed is the most important factor inhibiting better sustained performance on the SX-ACE. Our findings for the linear equations reinforced the idea that we need to do the vectorization within the elements, and we now need to change the data structure to reflect this, as the element index is often the fastest running index in our arrays. This will be a larger effort and instead we now turn to the nonlinear equations and have a brief look at the performance of the inviscid Euler equations for compressible flows.

4.2 Nonlinear Equations

To investigate nonlinear terms, we look here into the Euler equations for compressible fluid flows. As mentioned, we need to perform polynomial transformations between modal and nodal space in this case. There are several methods for this task implemented in *Ateles*, see [3] for more details and a comparison of the methods. For now we will only consider the L_2 projection of the Legendre modes to their nodes (L2P transformation in *Ateles*). This method is the most straight-forward one and can simply be written as a matrix-vector multiplication.

We look at the Euler equations for inviscid, compressible flows here. The original code showed no performance, due to the fluxes being called for each integration point and ending up to be the most time-consuming parts with only little to none FLOPS achieved. By pulling the loops over the points into the flux computation, this can be avoided and the contribution of these routines to the total computational time becomes negligible for now. Instead the *VolToFace* and the L2P are the most important contributors.

As the L_2 projection basically is a matrix-vector product, we also see a relatively high performance for this routine of more than 13 GFLOPS. However, we actually need to perform many of these matrix-vector products and it can be interpreted as a matrix-matrix product. If we rewrite our code such that the compiler recognizes this construct, it replaces it with a highly optimized implementation for the architecture and we gain close to 48 GFLOPS or 75% sustained performance for this operation when using polynomials of degree 29. To allow the compiler to recognize the construct, a two-dimensional array has to be used, which was previously not the case, as a collapsed index was used.

Table 3 shows the most relevant routines for Euler equations with a 30th order spatial discretization. The most efficient routine, the L_2 projection with the recognized matrix-matrix operation is also the most time consuming one, leading to a relatively good overall sustained performance. Further we recognize the volume to face projection, that was also relevant for the linear equations, but in its optimization has been split into three routines, one for each direction. Also the projection of the fluxes to the testfunctions are again contributing visibly to the overall compute time. The only other routine with more than 5% in the overall compute time is the *FromOver* routine, which implements the copying of the modal state to an oversampled space.

Thus, we see that further optimizations of the general routines that are important in the linear equations also will be beneficial for the nonlinear equations. We expect to achieve the next larger performance increment with the change of the index ordering for our state arrays.

Finally, we want to compare the serial performance of the current implementation status for varying orders to the observed performance on a scalar system. The scalar system we compare against is the Cray XC 40 *Hornet* at *HLRS* that is equipped with Intel Xeon E5-2680 v3 processors. To allow the comparison of runs on different machines and between different orders we use thousand degree of freedom updates

Table 3 Excerpt from the tracing of *Ateles* for Euler equations and a discretization with polynomials of degree 29

Procedure	%	MFLOPS	V.OP %	V.LEN	Bank Conf.		ADB %
					CPU	Net	
L2project	34.3	47916.4	99.21	174.5	1.680	6.543	75.38
VolToFaceY	6.9	1459.4	99.89	256.0	0.001	0.000	4.02
FromOver	6.5	2841.2	99.86	254.7	0.013	0.096	3.80
PrjFlux3	6.5	2544.4	99.90	186.9	0.781	1.237	68.88
VolToFaceX	6.1	1312.0	99.87	256.0	0.001	0.000	1.51
VolToFaceZ	6.1	1316.1	99.88	256.0	0.012	0.000	2.44
PrjFlux2	5.6	2943.7	99.89	187.0	0.511	0.504	70.55
PrjFlux1	5.5	2992.5	99.89	186.8	0.470	0.366	26.54

The first column states the measured routine, the second the running time percentage of the routine, the third the observed MFLOPS, the fourth is the vector operation ratio as a percentage (time spent on vector instructions to the time spent in total on that routine) and the fifth column provides the average vector length used in the vector instructions. The next two columns (6 and 7) provide the time spent on conflicts when accessing memory banks. In the eighth and last column, the ADB hit rate is given. Shown are the main procedures contributing to the overall compute time

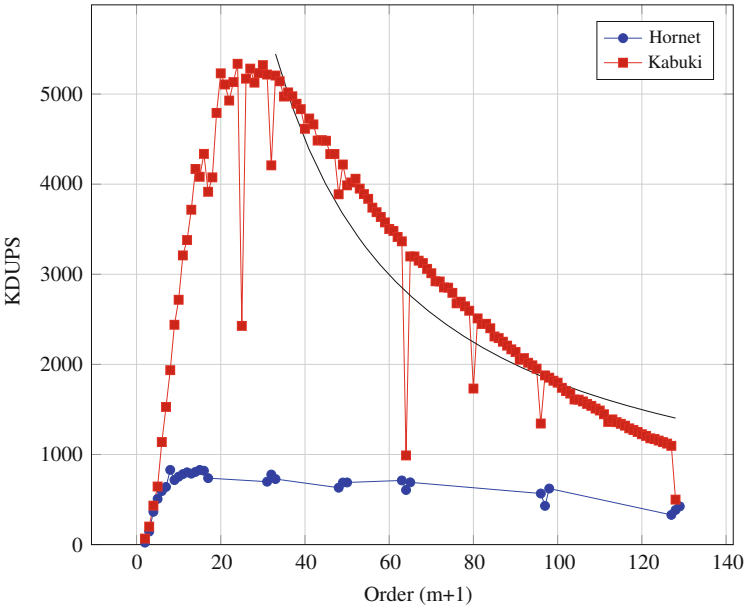

Fig. 2 Performance for the Euler 3D equation, 1 process, 20 million degrees of freedom in total

per second (KDUPS). The performance for the Euler equation and a total of 20 million degrees of freedom is shown in Fig. 2 over varying polynomial degrees for a fixed overall problem size of a total of 20 million degrees of freedom. Note that the computational effort per degree of freedom grows with order of the scheme

for nonlinear equations, and we expect a degradation of the degree of freedom update rate. This expected degradation is indicated by the black continuous line without marks. It does not imply a decrease in computational efficiency but rather the opposite, as we need to perform ever more operations without an increase in the data rate. As can be seen, a good performance is achieved when polynomials of degree 20 or higher are used for the nonlinear equations. We also notice several breakdowns of performance at some even order schemes. Most prominently for schemes of order 64, where we find the bad striding access described above, this time due to strided access over the modes of the polynomials within elements.

Of course there is not much of an benefit from the vectorization for scheme orders below 10 with a vectorization over within the elements, but already for 10 order schemes the vector computing capabilities can be used quite visibly. For very high orders beyond 100, the advantage seems to diminish again somewhat, but this seems to be more a point of the scalar system gaining efficiency due to the reduced bandwidth requirements in this range.

5 Summary and Outlook

High-order schemes are an attractive tool from the computational point of view, due to there reduced memory requirements. We presented a concept for the vectorization of the high-order discontinuous Galerkin scheme in *Ateles* with a focus on the structured data within elements to represent the three-dimensional polynomials. This approach enables a flexible computation on the mesh side with adaptivity and unstructured meshes, while at the same time allows for vectorized computations on highly structured data within the elements. Even with the long vectors of the NEC SX-ACE this concept works quite nicely already for relatively low order schemes with polynomials of degree 10 and higher.

Our current implementation is not yet tuned a lot for the vector system, and there are some legacy parts that need to changed, like the ordering of indices in the state representation. Nevertheless, the performance achieved on the SX-ACE already provides a quite good basis for further improvements. A particular positive surprise was the compiler optimization with the detected matrix-matrix multiplication construct and its optimized replacement by the compiler. Though, the nonlinear and linear equations have different routines that contribute to the overall computational effort, there is still a large overlap, and most further improvements are expected to affect all supported equations.

Acknowledgements We would like to thank Holger Berger from NEC for his kind support, the Tohoku University and HLRS for the opportunity to use their NEC SX-ACE installation.

References

1. Hennessy, J., Patterson D.: Computer Architecture, A Quantitative Approach. 5th edn. Morgan Kaufmann, Burlington (2012)
2. Klimach, H., Qi, J., Roller, S.: APES on SX-ACE. In: Resch, M., Bez, W., Focht, E., Patel, N., Kobayashi, H. (eds.) Sustained Simulation Performance 2016. Springer, Heidelberg (2016)
3. Anand, N., Klimach, H., Roller, S.: Dealing with non-linear terms in the modal high-order discontinuous Galerkin method. In: Resch, M., Bez, W., Focht, E., Patel, N., Kobayashi, H. (eds.) Sustained Simulation Performance 2016. Springer, Heidelberg (2016)