Semi-Lagrangian 4d, 5d, and 6d kinetic plasma simulation on large-scale GPU-equipped supercomputers

Lukas Einkemmer and Alexander Moriggl

Abstract
Running kinetic plasma physics simulations using grid-based solvers is very demanding both in terms of memory as well as computational cost. This is primarily due to the up to six-dimensional phase space and the associated unfavorable scaling of the computational cost as a function of grid spacing (often termed the curse of dimensionality). In this article, we present 4d, 5d, and 6d simulations of the Vlasov–Poisson equation with a split-step semi-Lagrangian discontinuous Galerkin scheme on graphic processing units (GPUs). The local communication pattern of this method allows an efficient implementation on large-scale GPU-based systems and emphasizes the importance of considering algorithmic and high-performance computing aspects in unison. We demonstrate a single node performance above 2 TB/s effective memory bandwidth (on a node with four A100 GPUs) and show excellent scaling (parallel efficiency between 30% and 67%) for up to 1536 A100 GPUs on JUWELS Booster.

Keywords
Fully kinetic simulation, general purpose computing on graphic processing units (GPGPUs), Vlasov–Poisson equation, semi-Lagrangian discontinuous Galerkin method

Introduction
Computational fluid dynamics has become an extremely successful framework to understand a range of natural phenomena ranging from astrophysics to everyday technological applications. Despite these successes, it is increasingly realized that many phenomena in plasma physics require a kinetic description (Howes et al., (2008); Zweibel and Yamada (2009)). The most challenging aspect of kinetic models, from a computational point of view, is certainly the up to six-dimensional phase space. The traditional approach to overcome this difficulty has been to employ the so-called particle methods (Verboncoeur (2005); Hariri et al. (2016)). While these simulations are comparably cheap and have been successfully employed in some applications, it is well known that they miss or do not resolve accurately certain physical phenomena, such as Landau damping. In addition, the inherent numerical noise introduced makes it extremely difficult to obtain accurate results for many physical phenomena. More recently, sparse grids (Kormann and Sonnendrücker (2016); Guo and Cheng (2016)) and low-rank methods (Einkemmer and Lubich (2018); Einkemmer and Joseph (2021)) have been used to reduce memory requirements and computational complexity. Nevertheless, the need to run fully kinetic simulations in high dimension persists for a range of applications.

To run such simulations, large supercomputers are needed, and codes to facilitate this for CPU-based systems are available. See, for example, Bigot et al. (2013) for a scaling study of the five-dimensional GYSELA code. More recently, also simulations for the full six-dimensional problem have been considered (Kormann et al., 2019). In the latter work, scaling to up to 65,536 processes has been reported on the SuperMUC HPC system of the Leibnitz Supercomputing Center. We also note that the amount of communication required usually increases as we increase the dimensionality of the problem. The reason for this is that the ratio of boundary points (that need to be transferred) to interior grid points increases significantly.

To numerically discretize these models, semi-Lagrangian schemes have proved popular. By using information on the
characteristics of the underlying partial differential equations (PDEs), these methods do not suffer from the usual step size restriction of explicit methods. A particular emphasis in collisionless models, as we will consider here, is to introduce as little numerical diffusion as possible into the approximation. It is well known that semi-Lagrangian schemes based on spline interpolation (see, e.g., Sonnendrücker et al. (1999) and Fillot and Sonnendrücker (2003) for a comparison of different methods) and spectral methods (see, e.g., Klimas and Farrell (1994); Klimas and Viñas et al. (1999) and Filbet and Sonnendrücker (2003) for a non-dimensionalized Vlasov equation). To describe the kinetic dynamics of a plasma, we use the non-dimensionalized system here, the time is expressed in multiples of the inverse of the plasma frequency.

The particle density is up to six-dimensional. Depending on the physical model, numerical simulations can be run using different numbers of dimensions in space (denoted by \(d_x\)) and in velocity (denoted by \(d_v\)). In this article, we consider problems with 2x2v (\(d_x = 2\) and \(d_v = 2\), four dimensional), 2x3v (\(d_x = 2\) and \(d_v = 3\), five dimensional), and 3x3v (\(d_x = 3\) and \(d_v = 3\), six dimensional).

While we restrict ourselves to the Vlasov–Poisson model in this work, we emphasize that the performance results obtained provide a good indication for other kinetic models as well. This is true for both problems from plasma physics (such as in case of the Vlasov–Maxwell equations) and kinetic problems in radiative transport.

Timed splitting

The first step in our algorithm for solving the Vlasov–Poisson equation is to perform a time splitting. The approach we use here has been introduced in the seminal paper (Cheng and Knorr, 1976) and has subsequently been generalized to, for example, the Vlasov–Poisson equation (Crouseilles et al., 2015).

The idea of the splitting approach is to decompose the nonlinear Vlasov–Poisson equation into a sequence of simpler and linear steps. The strategy to compute the numerical solution at time \(t^{n+1} = t^n + \Delta t\), where we use \(f^n = f(t^n, x, v)\), is as follows:

1. Solve \(\partial_t f(t, x, v) + v \cdot \nabla_x f(t, x, v) = 0\) with initial value \(f(0, x, v) = f^n(x, v)\) to obtain \(f^*(x, v) = f(\Delta t/2, x, v)\).
2. Compute \(E\) using \(f^*\) by solving the Poisson problem (2).
3. Solve \(\partial_t f(t, x, v) - E(t) \cdot \nabla_x f(t, x, v) = 0\) with initial value \(f(0, x, v) = f^*(x, v)\) to obtain \(f^{**}(x, v) = f(\Delta t, x, v)\).
4. Solve \(\partial_t f(t, x, v) + v \cdot \nabla_x f(t, x, v) = 0\) with initial value \(f(0, x, v) = f^{**}(x, v)\) to obtain \(f^{n+1}(x, v) = f(\Delta t/2, x, v)\).

This results in a second-order scheme (i.e., Strang splitting). Since the advection speed never depends on the direction of the transport, the characteristics can be determined analytically. For example, for step 1, we have

\[ f^*(x, v) = f^n(x - v \Delta t/2, v). \]

This advection in \(d_v\) dimensions can be further reduced to one-dimensional advections by the following splitting procedure

1a. \(f^{**}(x_1, x_2, x_3, v) = f^n(x_1 - v_3 \Delta t/2, x_2, x_3, v)\).
1b. \(f^{**}(x_1, x_2, x_3, v) = f^{**}(x_1, x_2 - v_2 \Delta t/2, x_3, v)\).
1c. \( f^\ast(x_1, x_2, x_3, v) = f^\ast_b(x_1, x_2, x_3 - v_3 \Delta t/2, v) \).

We emphasize that no error is introduced by this splitting as the corresponding operators commute. We apply a similar procedure to step 3, which results in the following:

3a. \( f^\ast_a(x, v_1, v_2, v_3) = f^\ast_a(x_1, v_1 + E_1(x) \Delta t, v_2, v_3) \).
3b. \( f^\ast_b(x, v_1, v_2, v_3) = f^\ast_b(x_1, v_1, v_2 + E_2(x) \Delta t, v_3) \).
3c. \( f^\ast(x, v_1, v_2, v_3) = f^\ast_b(x_1, v_1, v_2, v_3 + E_3(x) \Delta t) \).

Since the advection in the \( v \)-direction (step 3a-3c) does not change the electric field, we have reduced the up to six-dimensional Vlasov–Poisson equation to a sequence of one-dimensional linear advectons and a single Poisson solve. The scheme outlined here is second-order accurate (Einkemmer and Ostermann, 2014c). High-order methods have been constructed as well (Casas et al., 2017), and generalization of this basic approach to more complicated problems also exists (see, e.g., Crouseilles et al. (2015) for the Vlasov–Maxwell equations).

The advectons are treated using a so-called semi-Lagrangian method, which evaluates the function approximation at the foot of the characteristic curves. Since these end points of the characteristics do not lie on the grid points, an interpolation technique has to be applied. We will describe this procedure in detail in the next section.

**Semi-Lagrangian discontinuous Galerkin method**

Semi-Lagrangian methods have been widely used to solve the Vlasov equation. In particular, spline interpolation (Sonnendrücker et al. (1999); Filbet et al. (2001)) has emerged as a popular method as it does not introduce too much numerical diffusion, is conservative, and gives good accuracy at a reasonable computational cost.

However, in the construction of a spline, each degree of freedom couples with each other degree of freedom. Due to this global data dependency, spline interpolation is not suited to distributed memory parallelism. To remedy this in Crouseilles et al. (2009), a method that performs spline interpolation on patches has been proposed. However, on today’s massively parallel systems with relatively few degrees of freedom per direction on each MPI process (especially in the six-dimensional case), it is unclear how much advantage this approach provides. In this context, it should be noted that the massively parallel implementation found in Kormann et al. (2019) forgoes using spline interpolation and focuses exclusively on Lagrange interpolation. The same is true for the GPU-based implementation in Mehrenberger et al. (2013). While it is true that Lagrange interpolation scales well, it is also known to be very diffusive and usually large stencils are required in order to obtain accurate results.

In the present work, we will use a semi-Lagrangian discontinuous Galerkin scheme (sldg). This method, see, for example, Crouseilles et al. (2011); Rossmanith and Seal (2011), is conservative by construction (Einkemmer, 2017), has similar or better properties with respect to limiting diffusion than spline interpolation (see Einkemmer (2019) for a comparison), and is completely local (at most two neighboring cells are accessed to update a cell). Thus, it provides a method that in addition to being ideally suited for today’s massively parallel CPU and GPU systems is competitive from a numerical point of view. In addition, a well-developed convergence analysis is available for the Vlasov–Poisson equation (Einkemmer and Ostermann, 2014c,b).

The sldg scheme computes in an efficient way \( u^{n+1}(y) = u^n(y - at) \). The one-dimensional advectons in the previous section are written precisely in this form (all the remaining variables are considered as parameters here). This scheme first divides the domain into cells. The restriction of the solution to each such cell is a polynomial of degree \( k \). Note that there is no continuity constraint across cell interfaces and the approximant is thus discontinuous across this interface (see the left picture in Figure 1). The basis to form these polynomials are Lagrange polynomials which interpolate at the Gauss–Legendre points scaled to each cell. Thus, the degrees of freedom are function values \( u_i^n \) where the index \( i \) is the cell index and the index \( j \) refers to the \( j \)-th Gauss–Legendre point of cell \( i \). Since the translated function \( u^{n+1} \) in general does not lie in this approximation space, an \( L^2 \) projection is applied, see Figure 1. Only two adjacent cells are required in order to compute the values at a given cell, we denote the index of them with \( j^a \) and \( r^a \) in general. The resulting numerical scheme has order \( k + 1 \) and can be written as follows:

\[
\begin{align*}
   u_{ij}^{n+1} = \sum_l A_{ij} u_{il}^n + \sum_l B_{ij} u_{i+1,l}^n
\end{align*}
\]

(3)

The matrices \( A \) and \( B \) are of size \((k + 1) \times (k + 1)\) and only depend on the advection speed, which can be precomputed. More details about the specific form of these two matrices can be found in Einkemmer (2019).

**Poisson solver**

To determine the electric field, the Poisson equation (2) has to be solved once in each time step. However, since this is only an up to three-dimensional problem, the computational cost of this part of the algorithm is small. Using FFT-based methods is a common approach that is also supported in our SLDG code.
When the spatial variables $x$ are partitioned over multiple compute nodes, however, the FFT method requires global communication. Thus, we propose an alternative method which requires only local communication to solve this problem. Since the advection is already treated using a discontinuous Galerkin scheme, we will use the same approximation space for the Poisson solver. This has the added benefit that no interpolation to and from the equidistant grid on which FFT-based methods operate is necessary. The details of the discontinuous Galerkin Poisson solver are outlined in the Appendix.

Implementation and parallelization

Our code is written in C++/CUDA and uses templates to treat problems with different numbers of dimensions in physical and velocity space. This allows us to only maintain one codebase, while still providing excellent performance in all these configurations. Our code SLDG is available at https://bitbucket.org/leinkemmer/sldg under the terms of the MIT license.

The main computational effort of the code is due to the following three parts:

Advection: In the advection step, equation (3) is applied to each cell. This is done in a CUDA kernel, and the operation is memory bound on modern GPUs. This step is required whether we run the code on a single GPU or on a large supercomputer, and its cost per degree of freedom is expected to be independent of the number of MPI processes taking part in the simulation.

In 6d, to perform, for example, the advection in direction $x_3$, algorithm 1 has to be implemented. $Nv_{1,2,3}$ and $N_{x_{1,2}}$ are the amounts of degrees of freedom in the corresponding direction, $N_3$ is the number of cells in direction $x_3$, and $k_3$ is the degree of the approximating polynomial. Each thread in the CUDA kernel performs the innermost loop, while the number of threads is given by the second innermost loop. For many problem sizes, this number of threads is too small. Therefore, additional threads work on indices of the third innermost loop. The remaining indices are finally covered by thread blocks.

Algorithm 1. Computation of step 1c.

1: for $n_3 = 1$ to $N_3$ do
2: for $n_2 = 1$ to $N_2$ do
3: for $n_1 = 1$ to $N_1$ do
4: for $n_3 = 1$ to $N_3$ do
5: for $i = 1$ to $N_2$ do
6: for $j = 1$ to $k_3 + 1$ do
7: perform (3)
8: end for
9: end for
10: end for
11: end for

Communication: In order to run the code on multiple GPUs that possibly span multiple nodes, a domain decomposition approach is used. Depending on the problem, either only the velocity direction or both the space and velocity directions are divided along $dx$ or $dx + dv$ directions. Each of these blocks is assigned to a single GPU. Performing an advection step necessitates the exchange of boundary data from GPU to GPU. On a single node, this can be done using a call to cudaMemcpy, which makes use of NVLink if available. Communication between nodes is done via MPI. Our implementation supports the use of CUDA-Aware MPI which can make use of GPU memory in MPI calls. Thus, in this configuration, MPI routines can directly read from and place data in GPU memory.

Especially for high-dimensional problems, this type of boundary exchange can be relatively expensive. The reason for this is that the number of degrees of freedom in each spatial and velocity direction that can be stored on a single GPU is relatively small (compared to a three-dimensional problem). Thus, boundary cells constitute a larger fraction of the total problem size.
of the overall degrees of freedom, which due to the discrepancy between GPU memory and InfiniBand bandwidth means that communication can take up a significant fraction of the total run time. The data transfer is illustrated in Figure 2. Let us emphasize, however, that not the entire boundary of each data block associated to a GPU has to be transferred to each neighboring GPU. Since advection is directional, depending on the sign of the velocity field only either a send or a receive is required. This is illustrated in Figure 3. By taking this fact into account, only half the boundary data has to be transferred.

**E field:** The computation of the electric field proceeds in the following steps. First, the charge density $\rho(x) = \int f(x, v) \, dv$ has to be determined. This is first done on each GPU (using the CUB library NVIDIA (2018)), and then a reduction operation in the velocity direction is performed by calling MPI_Reduce on processes which span the entire velocity domain but represent the same part of the spatial domain (as illustrated in Figure 4).

At the completion of this step, $\rho$ is distributed over certain MPI root processes which span the entire space domain. These processes then participate in computing the electric field $E$. This can be done either by an FFT-based method, which requires global communication among the processes, or an iterative conjugate gradient discontinuous Galerkin (CGdG) solver, which only requires nearest neighbor exchange in each iteration (as illustrated in Figure 5).

Finally, the electric field has to be made available to all processes that span the velocity domain. To do this, MPI_Broadcast is called (see Figure 6).

Although from a computational perspective the computation of the electric field is relatively cheap (as it is only an up to three-dimensional problem), the associated data
transfer can become an impediment to scaling, especially on large systems.

In a multi-node context, each MPI process is assigned to one or multiple blocks. Each block manages a single GPU. The configuration we employ on most clusters with multiple GPUs on each node is to have one MPI process per node (i.e., one MPI process per shared memory domain) and consequently as many blocks per MPI process as GPUs are on a node. However, on JUWELS Booster, this significantly degrades the bandwidth that is available to conduct MPI communication between different nodes. Thus, the multi-node simulations in this article are run using a single MPI process per GPU. On JUWELS Booster therefore four MPI processes are launched on each node, and CUDA-aware MPI is used both for inter-node as well as intra-node communication.

Validation

To validate our implementation, two commonly used test scenarios are considered. The linear Landau damping problem given by the initial value

$$f(0, x, v) = \exp\left(-\frac{\|v\|^2}{2}\right) \left(1 + \alpha \sum_{i=1}^{d} \cos(k x_i)\right)$$

on $[0, 4\pi]^d \times [-6, 6]^d$, $\alpha = 10^{-2}$, and $k = 0.5$ and a bump-on-tail instability:

$$f(0, x, v) = \frac{1}{(2\pi)^{d/2}} \left(0.9\exp\left(\frac{v_1^2}{2}\right) + 0.2\exp(-2(v_1 - 4.5)^2)\right)$$

$$\exp\left(-\frac{\sum_{i=2}^{d} v_i^2}{2}\right) \left(1 + 0.03 \sum_{i=1}^{d} \cos(0.3x_i)\right).$$

on $[0, 20\pi/3]^d \times [-9, 9]^d$.

In the simulations presented here, we use the 4th order semi-Lagrangian discontinuous Galerkin scheme (as it represents a good compromise between accuracy and computational cost Einkemmer (2017; 2020)) and a time step size of 0.1. In Figures 7 and 8, the time evolution of the electric energy for these initial conditions is shown for five- and six-dimensional configurations, respectively. The results agree well with what has been reported in the literature. In addition, the Landau damping results show excellent agreement with the analytically obtained decay rate. There is no significant difference between the results obtained with the FFT and discontinuous Galerkin Poisson solver. The SLDG code also includes a set of unit tests that check a range of results against known analytic and numerical solutions and compares the results obtained on the GPU to the results on the CPU.

We further remark that the results obtained using single and double precision overlap in the plots. Thus, if one is not interested in too stringent tolerances running simulations with the semi-Lagrangian discontinuous Galerkin scheme in single precision is a viable option. This reduces memory consumption by a factor of two. In particular, for GPUs this can be attractive as memory, in general, is more limited on GPU-based systems. Thus, larger problem sizes can be
treated per GPU. In addition, for some problems consumer cards can potentially be used, which do not have strong double-precision performance. We will present and discuss both double- and single-precision results in the remainder of this article.

Single-node performance

The results in this section are obtained simulating the linear Landau test case. If not otherwise stated, a fixed time step size of 0.1 and the 4th order semi-Lagrangian discontinuous Galerkin method is used. To measure the performance of our code, the achieved memory bandwidth (BW) is used as a metric. This allows us, for our primarily memory bound problem, to easily compare the performance of our implementation with the peak performance of the hardware we run the simulations on. In addition, the run time can be easily obtained from this metric as well.

For the 3x3v (i.e., the six-dimensional) problem solved using the second-order Strang splitting scheme, nine advections have to be performed in each time step. This implies that the entire data have to be accessed 18 times (one read and one write per advection). For the 2x2v and 2x3v problem, six advections have to be performed and the data are consequently accessed 12 times. Moreover, for calculating the electric field once more, the entire data have to be accessed (to compute the density $\rho$). This is done twice (once at the end of the time step to compute diagnostic values such as the electric energy and once at the middle of the time step). Therefore, we have the following:

$$\text{achieved bandwidth} = \frac{F_{d,v} \cdot \text{sizeof}(fp) \cdot \text{dof} \cdot 10^{-9}}{\text{time per step in seconds}} \text{GB/s}$$

where $fp$ is single or double and $F_{d,v}$ is 20 for the 3x3v problem and 14 otherwise. For comparison, we give in Table 1 the theoretical peak bandwidth for the different GPUs that we will use in this work. These are the values specified by the vendors and the actual bandwidth obtained, even in a simple benchmark such as copy, is somewhat less.

Note that the achieved bandwidth is a good measure of performance because the problem is memory bound. To see this, we determine the number of arithmetic operations from equation (3). In total, we require $(k + 1)(4(k + 1) - 1)$ operations per cell. Thus, the flop/byte ratio is as follows:
Table 1. Different GPUs used in this work and their specifications.

| GPU     | Memory bandwidth | Memory size |
|---------|------------------|-------------|
| GTX1080 | 320 GB/s         | 8 GB        |
| Titan V | 653 GB/s         | 12 GB       |
| V100    | 900 GB/s         | 16 GB       |
| A100    | 1555 GB/s        | 40 GB       |

Table 2. Performance using different Poisson solvers on our local GPU cluster. The computations are done in double precision on an A100 GPU.

|                  | FFT         | CGdG        |
|------------------|-------------|-------------|
|                  | dof         | Time/step   | Time/step   | Average iter. |
| 2x2v             | 220^4       | 0.372 s     | 0.477 s     | 173           |
| 2x3v             | 72^4        | 0.299 s     | 0.306 s     | 55            |
| 3x3v             | 36^4        | 0.619 s     | 0.646 s     | 38            |

\[
\text{flop} = \frac{\text{dof} \cdot (4(k + 1) - 1)}{2 \cdot \text{dof} \cdot \text{sizeof}(\text{f}p)} = \frac{4(k + 1) - 1}{2 \cdot \text{sizeof}(\text{f}p)}
\]

For double precision and the 4th-order method, that is, \(k = 3\), this turns out to be approximately equal to 0.94, which is significantly below the hardware flop-byte ratio on modern GPUs (e.g., on A100 the double-precision flop-to-byte ratio is approximately 6.2).

Single GPU performance

First, we compare the performance of the FFT and discontinuous Galerkin Poisson solver. The results are presented in Table 2. The run time for the FFT and the discontinuous Galerkin method is very similar, particular for the 2x3v and 3x3v problems. For the 2x2v problem, the FFT-based scheme is faster by approximately 30%. This can be explained by the fact that the degrees of freedom per direction are reduced for higher dimensional problems (due to computational constraints) and thus not as many iterations are needed to obtain convergence. Since the FFT Poisson solver is faster and there is no disadvantage of using it for single-node computations (we will discuss its role in scaling to larger systems later in this work), all of the computations performed in this section use the FFT-based Poisson solver.

In Tables 3 and 4, we show the achieved bandwidth in the 2x3v and the 3x3v cases using four different types of GPUs. We achieve approximately half the bandwidth of what is theoretically possible on each platform. We consider this an excellent result as the memory access pattern is significantly less uniform than in a stencil code and there is a significant amount of computation that is not taken into account by our estimate of the achieved bandwidth (e.g., computing the electric field from the density \(\rho\); see Einkemmer (2020) for a more detailed discussion).

Since single GPU results for the 2x2v and 2x3v cases were already presented in Einkemmer (2020), we will focus on the six-dimensional 3x3v case in the following. In Figure 9, a comparison of the achieved bandwidth between different orders of the method in the 3x3v case is shown. The 4th and 5th order methods achieve better performance than the 2nd and 3rd order methods, which is similar to the behavior observed in Einkemmer (2020) for the 2x2v and 2x3v cases.

In Figure 10, we show the timings for the different parts that constitute the algorithm in the 2x3v and the 3x3v cases using different GPUs. The chosen number of degrees of freedom is the maximum amount possible on the specific GPU. In the 2x3v case, it can be seen once more that the choice of the Poisson solver has no impact on the overall performance. The main difference between the consumer GPU (the GTX1080) and the enterprise grade GPUs is that the GTX1080 needs more time for the computation of the charge density \(\rho\). This comes from the fact that the GTX1080 is able to perform significantly less floating-point operations per second than the other GPUs. Those are important to compute the diagnostic quantities (such as electric energy) in the reduction step. Similar results can be observed for the 3x3v case.

To conclude this section, we will briefly compare the performance between the GPU implementation (which is the focus of the present work) and the CPU implementation. A comparison for the 3x3v problem between the A100 GPU and an Intel Xeon Gold 6226R dual socket system (having 2 × 16 CPUs) is given in Figure 11. It can clearly be observed that the GPU drastically outperforms the CPU-based system. In fact, every part of the code is computed by the GPU in significantly less time. This, in addition to the four and five dimensional results presented in Einkemmer (2020), underscores the advantage of running semi-Lagrangian Vlasov simulations on GPU-based systems.

Table 3. Achieved bandwidth for the 2x3v case on a single GPU using single and double precision. Missing values imply that there is not enough memory on the specific GPU for the specified resolution.

| dof | GTX | TitanV | V100 | A100 |
|-----|-----|--------|------|------|
| 56^4 | 104 | 356    | 412  | 583  | 649  |
| 60^4 | 100 | 377    | 430  | 607  | 670  |
| 64^4 | 401 | 482    | 666  | 717  |      |
| 68^4 | 454 | 634    | 696  |      |      |
| 72^4 | 680 | 727    |      |      |      |
| 76^4 | 660 |        |      |      |      |
| 80^4 | 708 |        |      |      |      |
| 84^4 | 681 |        |      |      |      |
We also note that this is enabled by the semi-Lagrangian discontinuous Galerkin method which, in contrast to spline- or FFT-based interpolation, maps extremely well to the GPU architecture.

### Multiple GPU performance

In this section, we analyze the performance of the code using four A100 GPUs on a single node. We increase the number of grid points in two of the velocity dimensions proportionally to the number of GPUs used. That is, we use weak scaling, which is the only metric of interest for these high-dimensional problems. For the computation, we use the JUWELS Booster supercomputer. The four A100 GPUs on a single node are connected via NVLink3 to each other. The communication process is performed using CUDA-Aware MPI, which automatically uses NVLink for intra-node communication. Since CUDA-Aware MPI is not available on every GPU-equipped system, we also provide an option that uses classic MPI communication between nodes and explicit communication via the host on each node. The corresponding performance results presented in Table 5 have been obtained on JUWELS Booster, where for the without NVLink option the data are explicitly transferred via the host. As we can see, using NVLink results in approximately a factor of 2 improvement, yielding a total single node achieved bandwidth of between 2.05 and 2.55 TB/s.

## Scaling to multiple nodes

As can be observed in the previous sections, the resulting grid is still relatively coarse. This is especially true in the five- and six-dimensional cases. In order to run the simulations on finer grids, more nodes are required. In this section, we will report weak scaling results for our code on JUWELS Booster. This is the most important metric for practical applications, where a finer grid is often required in order to obtain physically relevant results. The reported...
measured wall clock time is the average over the computed
time steps. We analyze the total computation time and the
time for the pure computation of the advections (denoted
as $A_{\text{Advection}}$ in the figures), the time which is required for
the data transfer before the advection takes place (denoted
as $C_{\text{Communication}}$), and the time for computing the
electric field (denoted as $E_{\text{field}}$), which includes the
computation of the charge density, the time for solving the
Poisson problem (including all the required communication),
and the broadcast of the electric field. The time step size is chosen such that the CFL-number is close to one.
Moreover, the largest possible local grid which
fits into

\[ \text{GPU memory} \]

is chosen.

### $2x2v$

Two test cases are considered in the four-dimensional case.
First, the full domain is partitioned and distributed over
multiple GPUs, and second, only the velocity domain is
refined. The latter is quite common as some problems re-
quire a higher resolution in velocity space, for example, to
shift back in time the (purely numerical) recurrence effect in
linear Landau damping simulations (Einkemmer and Ostermann, 2014a). In Table 6, the considered grids are
listed.

Let us first consider the case when the full domain is distributed over multiple GPUs. The corresponding results are shown in Figure 12. A time step of 0.015 is chosen in order to keep the CFL number below one, and the CGdG method is used to solve the Poisson problem. The largest grid here has $880^21760^2$ degrees of freedom. It can be observed that the time to perform the advection remains constant. This holds for each configuration in this chapter. The communication time in this setting is relatively small as at each time step only 2 GB of data has to be transferred
to neighboring MPI processes. The time for computing the
electric field increases from 4 to 16 GPUs since a larger problem has to be solved, that is, going from a grid size of
$220^2$ to $440^2$ and thus more iterations are required to solve
the linear system. The system size stays then constant until
64 GPUs are used. Subsequently, the problem size in
physical space increases again. That is, a grid size in the
space direction of $880^2$ is used when 256 or more GPUs are
used. This again requires more iterations, and the wall
clock time increases again. The total time per step rises from about 0.5 seconds to around 1.0 seconds. Thus,
running the simulation on 1024 GPUs a parallel efficiency of approximately 50% is achieved.

Since the impact of the Poisson solver on the overall
performance due to the large number of required iterations in the $2x2v$ setting is quite high, we also investigate the use of an FFT-based method to solve the Poisson problem. The results are shown in Figure 13. The FFT-based solver is faster on a single node, but scaling is a challenge. Thus, the overall parallel efficiency decreases a little, while overall the run time on 1024 GPUs is still reduced by approximately 10% compared to the CGdG method.

Next, we analyze the behavior when the degrees of
freedom are increased in the velocity directions only.
Here, since in physical space no parallel numerical
method for the computations is required, the FFT-based
Poisson solver is used since it is faster. In Figure 14, the
weak scaling results are given. It can be observed that the
required time for computing the electric field increases when more than 144 GPUs are used. Here, in contrast to the results shown in Figures 12 and 13, not the time for
solving the Poisson problem increases, but the required
time to compute the charge density increases, that is, the
call to $\text{MPI}_\text{Reduce}$. The reason for this is that now a
very fine resolution of $7040^2$ grid points in the velocity
directions is used. The total run time increases from about
0.36 to 0.6; thus, a parallel efficiency of 60% is achieved
on 1024 GPUs.

The simulations presented previously were done with
a relatively small time step size of 0.015 in order to keep
the CFL number in the spatial directions below one.
However, the semi-Lagrangian discontinuous Galerkin
scheme is also stable for larger time steps. We will thus
investigate the parallel efficiency for a CFL number larger

### Table 5. Comparison of the achieved bandwidth using four A100
GPUs on one computation node with and without NVLink.

\begin{array}{|c|c|c|c|}
\hline
\text{Dim} & \text{dof} & \text{Without NVLink} & \text{With NVLink} \\
\hline
2x2v & $220^2440^2$ & 1821 GB/s & 2437 GB/s \\
2x3v & $72^2144^272$ & 1202 GB/s & 2567 GB/s \\
3x3v & $36^272^2$ & 859 GB/s & 2052 GB/s \\
\hline
\end{array}
than one. The main additional issue here is that the amount of boundary data that needs to be communicated to neighboring processes increases linearly with the CFL number. We use a local grid with a resolution of 2164 points and in total 256 GPUs, thus distributing the domain in each direction over four GPUs. The local grid here is slightly smaller than in the previous simulations since additional memory is required to store the boundary data. The CFL number increases from one to five by increasing the time step size by 0.02 until we reach a step size of 0.095 (where the CFL number becomes 5). The results are reported in Figure 15. It can be observed that the time for communication increases continuously, which is expected since more boundary cells have to be sent to the neighboring processes. Furthermore, the time for computing the advection remains almost constant, and the time for computing the electric field increases. This comes from the fact that when solving the Poisson problem iteratively, the starting point is the electric potential obtained at the previous time step. Thus, when a larger time step is used, the initial estimate used in the iterative Poisson solver is farther away from the solution and thus more iterations are required for convergence. Nevertheless, we note that the run for a CFL number smaller than one to a CFL number of five increases by approximately 50%. Thus, running the numerical simulation at a CFL number of five still results in a speedup of approximately a factor 3.3 (despite the increased cost of communication per time step).

Table 6. For the 2x2v problem, we use a local grid of 220^4 degrees of freedom on each GPU. In the multi-node setting, the GPUs are either equally divided among the different coordinate axes (middle table) or only the velocity direction is refined (bottom table).

| Dim dof | \(x_1\) 220 | \(x_2\) 220 | \(v_1\) 220 | \(v_2\) 220 |
|---------|-------------|-------------|-------------|-------------|
| #GPUs   | Decom. in \(xv\) | Decom. in \(xv\) |
| 1       | 1           | 1           | 1           | 1           |
| 4       | 1           | 1           | 2           | 2           |
| 8       | 1           | 2           | 2           | 2           |
| 16      | 2           | 2           | 2           | 2           |
| 32      | 2           | 2           | 2           | 4           |
| 64      | 2           | 2           | 4           | 4           |
| 128     | 2           | 4           | 4           | 4           |
| 256     | 4           | 4           | 4           | 4           |
| 512     | 4           | 4           | 4           | 8           |
| 1024    | 4           | 4           | 8           | 8           |

| #GPUs   | Decom. in \(v\) | Decom. in \(v\) |
|---------|-----------------|-----------------|
| 1       | 1               | 1               |
| 4       | 1               | 2               |
| 16      | 1               | 4               |
| 64      | 1               | 8               |
| 144     | 1               | 12              |
| 256     | 1               | 16              |
| 484     | 1               | 22              |
| 1024    | 1               | 32              |

Figure 12. Weak scaling in the 2x2v dimensional setting using multiple GPUs for each dimension. A time step size of 0.015 and the CGdG method to solve the Poisson problem are used.

Figure 13. Weak scaling in the 2x2v dimensional setting using multiple GPUs for each dimension. A time step case size of 0.015 and an FFT-based solver for the Poisson problem are used.

Figure 14. Weak scaling in the 2x2v dimensional setting using multiple GPUs to increase the degrees of freedom in the velocity dimensions only.
In Figures 16 and 17, the scaling results for the 2x3v case are shown. Again, as in the 2x2v case, two configurations are considered, namely, the full grid refinement and increasing the degrees of freedom in the velocity domain only. The considered configurations are listed in Table 7.

When the full domain is distributed over multiple GPUs, a time step of 0.05 is chosen which implies a CFL number below one. The CGdG method is used to solve the Poisson problem. The largest problem has a grid of 2884432 and is run on 1536 GPUs. In Figure 16, it can be observed that the computation of the electric field has much less impact on the overall performance than in the 2x2v setting. This is mainly because solving the Poisson problem with the CGdG requires less iterations due to the smaller number of degrees of freedom per direction. Nevertheless, also here, as expected, an increase of required wall time can be observed if the problem size increases. The time for the communication which is required to perform the advection has more impact on the overall performance than in the 2x2v setting. The CGdG method allows us to limit the impact to performance and is the preferred choice for this five-dimensional problem. For 1536 GPUs, we achieve a parallel efficiency of 43%.

In problems where only additional degrees of freedom are added in the velocity directions, similar results as have been obtained in the 2x2v case can be observed. In this case, the FFT-based solver is the preferred choice. The corresponding weak scaling results are shown in Figure 17, where we observe almost ideal scaling for up to 1440 GPUs. The parallel efficiency in this case is approximately 67%.

### 3x3v

Although the best choice is to distribute the problem in each direction (if a distribution is done only in the velocity direction, the number of degrees of freedom in physical space is too small to be of practical interest). The used configurations are listed in Table 8. Using 1536 GPUs results in a grid of size 72^1108^144^4 in double precision, which is still relatively coarse, and 80^1120^160^4 in single precision.

In the six-dimensional setting, the only reasonable choice is to distribute the problem in each direction (if a distribution is done only in the velocity direction, the number of degrees of freedom in physical space is too small to be of practical interest). The used configurations are listed in Table 8. Using 1536 GPUs results in a grid of size 72^1108^144^4 in double precision, which is still relatively coarse, and 80^1120^160^4 in single precision.

In Figure 18, the scaling results for the 3x3v case are shown. Due to the coarse grid, a time step of 0.1 is chosen which is enough to keep the CFL number below one. The CGdG solver is the preferred choice to solve the Poisson problem. It can be observed that the communication time increases significantly until 64 GPUs are used and remains almost constant afterward. The reason for this behavior is that for a six-dimensional problem, 64 GPUs are required so that in each direction we use more than a single GPU. Due to the nearest neighbor exchange, any further parallelization
In principle, does not contribute to the overall communication time. The time per step increases from 0.6 seconds using one GPU to 1.9 seconds using 1536 GPUs; thus, a parallel efficiency of approximately 30% is achieved.

**Conclusion**

In this work, we have demonstrated that semi-Lagrangian discontinuous Galerkin schemes can be used to run five- and six-dimensional Vlasov simulations on large-scale GPU-equipped supercomputers. The local nature of this method gives excellent performance on a single node using four GPUs (between 2 and 2.5 TB/s depending on the configuration) and scales well up to 1500 GPUs on JUWELS Booster with a parallel efficiency of between 30% and 67% (depending on the configuration). Since GPU-based supercomputers will play an important role in exascale systems and, most likely, in future generations, these results are promising.

**Table 7.** For the 2x3v problem, we use a local grid of $72^5$ degrees of freedom on each GPU. In the multi-node setting, the GPUs are either equally divided among the different coordinate axes (middle table) or only the velocity direction is refined (bottom table).

| #GPUs | Decompl. in xv |
|-------|----------------|
| 1     | 1              |
| 4     | 2              |
| 8     | 2              |
| 16    | 2              |
| 32    | 2              |
| 64    | 2              |
| 128   | 2              |
| 256   | 2              |
| 512   | 4              |
| 1024  | 4              |
| 1536  | 6              |

**Figure 18.** Weak scaling in the 3x3v case using multiple GPUs. The spatial configuration is listed in Table 8. Computations are done in double precision.

**Figure 19.** Weak scaling in the 3x3v case using multiple GPUs. The configuration is listed in Table 8. Computations are performed in single precision.

**Table 8.** Configurations in the 3x3v case using multiple GPUs.

| #GPUs | Decompl. in xv |
|-------|----------------|
| 1     | 2              |
| 4     | 2              |
| 8     | 2              |
| 16    | 2              |
| 32    | 4              |
| 64    | 4              |
| 216   | 6              |
| 512   | 8              |
| 1000  | 10             |
| 1440  | 12             |

does, in principle, not contribute to the overall communication time. The time per step increases from 0.6 seconds using one GPU to 1.9 seconds using 1536 GPUs; thus, a parallel efficiency of approximately 30% is achieved.

In Figure 19, the results using single precision are shown. Since less memory is required, approximately 1.9 times more degrees of freedom can be considered than double precision. We observe a similar parallel efficiency of approximately 33%.
the future of high-performance computing, the results demonstrated in this work pave the way toward running large-scale Vlasov simulations on such systems.

Acknowledgments
We acknowledge PRACE for awarding us access to JUWELS at GCS@FZJ, Germany. The computational results presented have been achieved in part using the Vienna Scientific Cluster (VSC).

Declaration of conflicting interests
The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding
The author(s) disclosed receipt of following financial support for the research, authorship, and/or publication of this article: This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no 847476. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

ORCID iD
Alexander Moriggl https://orcid.org/0000-0003-1739-212X

References
Bigot J, Grandgirard V, Latu G, et al. (2013) Scaling gysela code beyond 32K-cores on blue gene/Q. ESAIM: Proceedings EDP Sciences, 43: 117–135.
Camporeale E, Delzanno G, Bergen B, et al. (2016) On the velocity space discretization for the vlasov–poisson system: comparison between implicit hermite spectral and particle-in-cell methods. Computer Physics Communications 198: 47–58.
Casas F, Crouseilles N, Faou E, et al. (2017) High-order hamiltonian splitting for the vlasov–poisson equations. Numerische Mathematik 135(3): 769–801.
Cheng C and Knorr G (1976) The integration of the vlasov equation in configuration space. Journal of Computational Physics 22(3): 330–351.
Crouseilles N, Einkemmer L and Faou E (2015) Hamiltonian splitting for the vlasov–maxwell equations. Journal of Computational Physics 283: 224–240.
Crouseilles N, Latu G and Sonnendrücker E (2009) A parallel vlasov solver based on local cubic spline interpolation on patches. Journal of Computational Physics 228(5): 1429–1446.
Crouseilles N, Mehrenberger M and Vecil F (2011) Discontinuous galerkin semi-lagrangian method for vlasov-poisson. ESAIM: Proceedings EDP Sciences, 32: 211–230.
Einkemmer L (2017) A study on conserving invariants of the Vlasov equation in semi-Lagrangian computer simulations. Journal of Plasma Physics 83(2).
Einkemmer L (2019) A performance comparison of semi-Lagrangian discontinuous Galerkin and spline based Vlasov solvers in four dimensions. Journal of Computational Physics 376: 937–951.
Einkemmer L (2020) Semi-Lagrangian Vlasov simulation on GPUs. Computer Physics Communications 254: 107351.
Einkemmer L and Joseph I (2021) A mass, momentum, and energy conservative dynamical low-rank scheme for the Vlasov equation. Journal of Computational Physics 110495.
Einkemmer L and Lubich C (2018) A low-rank projector-splitting integrator for the Vlasov–Poisson equation. SIAM Journal on Scientific Computing 40(5): B1330–B1360.
Einkemmer L and Ostermann A (2014a) A strategy to suppress recurrence in grid-based Vlasov solvers. The European Physical Journal D 68: 197.
Einkemmer L and Ostermann A (2014b) Convergence analysis of a discontinuous Galerkin/Strang splitting approximation for the Vlasov–Poisson equations. SIAM Journal on Numerical Analysis 52(2): 757–778.
Einkemmer L and Ostermann A (2014c) Convergence analysis of Strang splitting for Vlasov-type equations. SIAM Journal on Numerical Analysis 52(1): 140–155.
Epshteyn Y and Rivière B (2007) Estimation of penalty parameters for symmetric interior penalty Galerkin methods. Journal of Computational and Applied Mathematics 206(2): 843–872.
Filbet F, Sonnendrücker E and Bertrand P (2001) Conservative numerical schemes for the Vlasov equation. Journal of Computational Physics 172(1): 166–187.
Filbet F and Sonnendrücker E (2003) Comparison of Euelerian Vlasov solvers. Computer Physics Communications 150(3): 247–266.
Guo W and Cheng Y (2016) A sparse grid discontinuous Galerkin method for high-dimensional transport equations and its application to kinetic simulations. SIAM Journal on Scientific Computing 38(6): A3381–A3409.
Hariri F, Tran T, Jocksch A, et al (2016) A portable platform for accelerated PIC codes and its application to GPUs using OpenACC. Computer Physics Communications 207: 69–82.
Howes G, Dorland W, Cowley S, et al (2008) Kinetic simulations of magnetized turbulence in astrophysical plasmas. Physical Review Letters 100: 065004.
Klimas A and Farrell W (1994) A splitting algorithm for Vlasov simulation with filamentation filtration. Journal of Computational Physics 110(1): 150–163.
Klimas A and Viñas A (2018) Absence of recurrence in Fourier–Fourier transformed Vlasov–Poisson simulations. Journal of Plasma Physics 84(4).
Kormann K, Reuter K and Rampp M (2019) A massively parallel semi-Lagrangian solver for the six-dimensional Vlasov–Poisson equation. The International Journal of High Performance Computing Applications 33(5): 924–947.
Kormann K and Sonnendrücker E (2016) Sparse Grids for the Vlasov–Poisson Equation In: Sparse Grids and Applications - Stuttgart 2014. Springer International Publishing, pp. 163–190.
Author biographies

Alexander Moriggl received his masters degree in mathematics at the University of Verona. Currently, he is enrolled as a PhD student at the University of Innsbruck. He is working on semi-Lagrangian discontinuous Galerkin methods for GPUs.

Lukas Einkemmer is a professor at the University of Innsbruck. His main research interests include efficiently solving high-dimensional problems in plasma physics and other hyperbolic systems. For example, dynamical low-rank approximation, splitting methods, semi-Lagrangian schemes, and discontinuous Galerkin methods. A particular focus is the development of methods that can be used efficiently on modern computer systems.

Appendix I

Discontinuous Galerkin Poisson solver

First, we describe the 1d case which can then be easily extended to arbitrary dimensions, as we will show. The same polynomial space

\[ V_h^k = \left\{ \varphi \in L^2(\Omega) : \varphi \in \mathbb{P}^k(I_i), \quad I_i = [x_i, x_{i+1}], \quad i = 0, \ldots, N - 1 \right\} \]

that is used to perform the advections is chosen. As basis for \( V_h^k \) the Lagrange polynomials interpolating on the \( k + 1 \) Gauss–Legendre points \( x_{il} \) restricted to the cell \( I_i \) are chosen:

\[
\varphi_j(x) = \left\{ \begin{array}{ll}
\frac{x - x_{il}}{x_{ij} - x_{il}} & x \in I_i \\
0 & x \in \Omega \setminus I_i
\end{array} \right.
\]

The first step in deriving the discontinuous Galerkin Poisson solver consists in multiplying the Poisson equation (2) by a test function \( v \in V_h^k \), integrating over each cell \( I_i = [x_i, x_{i+1}] \) and summing up all terms. For more details, see Rivière (2008). This gives, in 1d, the following equation

\[
\sum_{i=0}^{N-1} \left( \int_{I_i} \phi'(x) v'(x) \, dx - \left[ \phi'(x_i) \right] \{v(x_i)\} - \left\{ \phi'(x_i) \right\} \{v(x_i)\} + \frac{\sigma \hbar}{2}[v(x_i)][\phi(x_i)] \right) = \int_{\Omega} (1 - \rho(x)) v(x) \, dx,
\]

where \( \{v(x_i)\} = \frac{1}{2}(v(x_i^-) + v(x_i^+)) \) is the average of the function \( v \) at the point \( x_i \) and \( [v(x_i)] = v(x_i^-) - v(x_i^+) \) is the jump height. From the left hand side of (4) a symmetric bilinear form \( a : V_h^k \times V_h^k \to \mathbb{R} \) can be derived. Therefore, we are looking for \( \phi^{\text{DG}} \in V_h^k \) such that

\[
a(\phi^{\text{DG}}, v) = I(v) \forall v \in V_h^k
\]

where the linear form \( I \) is defined by the right-hand-side of (4). Plugging \( \phi^{\text{DG}}(x) = \sum l \phi_l(x) \varphi_i(x) \) and \( v = \varphi_{lm} \) into \( a \) results in a linear system of the form \( Au = b \) where we are looking for the unknown \( u \) specified by the coefficients \( \phi_l(x_i) \). This method is called the symmetric interior penalty Galerkin (SIPG) method. It can be easily verified that this matrix \( A \) is block tridiagonal, and it can be constructed from two small matrices \( M \) and \( B \) of size \( k + 1 \),

\[
A = \frac{1}{\hbar} \begin{pmatrix}
M & B & \vdots \\
B^T & M & B \\
\vdots & B^T & M
\end{pmatrix}
\]

To ensure coercivity of the bilinear form \( a \) and positive definiteness of the matrix \( A \), the condition \( \sigma \geq k^2 \) is required, see Epshteyn and Rivière (2007). The matrices \( M \) and \( B \) are defined as follows
\( (M)_{i,j} = + \frac{1}{2} \phi'(0) \phi_j(0) + \frac{1}{2} \phi'(0) \phi_j'(0) + \sigma \phi_j(0) \phi_j(0) \\
- \frac{1}{2} \phi'(1) \phi_j(1) - \frac{1}{2} \phi'(1) \phi_j'(1) + \sigma \phi_j(0) \phi_j(1) \\
+ \frac{1}{2} \sum_{i=0}^{k} \phi'(x_i) \phi'(x_i) w_i, \\
(\mathcal{B})_{i,j} = - \frac{1}{2} \phi'(0) \phi_j(1) + \frac{1}{2} \phi'(0) \phi_j'(1) - \sigma \phi_j(0) \phi_j(1) \)

where we denote with \( \phi_i(x) \) the Legendre polynomials in the interval \([0,1]\) that interpolate the Gauss–Legendre quadrature points \( x_i \). The \( w_i \) are the Gaussian quadrature weights. In the higher dimensional setting, the bilinear form is given as follows

\[
a(w, v) = \sum_{\epsilon} \iint_{\mathcal{E}} \nabla w \cdot \nabla v - \sum_{\mathcal{E}} \iint_{\epsilon} \{ \nabla w \cdot n \} \lfloor w \rfloor - \sum_{\epsilon} \iint_{\epsilon} \{ \nabla w \cdot n \} \lfloor v \rfloor + \sum_{\epsilon} \frac{\sigma}{|\mathcal{E}|} \iint_{\epsilon} |v|^2 \\
\]

where \( E \) in 2d are rectangles and in 3d are squares, and \( e \) are the edges of the rectangles or the faces of the squares. Usually, the denominator in the last term is defined as \(|\epsilon|^{\beta} \), but setting \( \beta = 1 \) in 2d and \( \beta = \frac{1}{2} \) in 3d is almost equivalent if the length of the edges do not vary much. Since in higher dimensions the basis is the tensor product of the 1d basis, the resulting matrix can be written in the following form in 2d

\[
(h, W_j \otimes A_x + h_c A_y \otimes W_z) u = b
\]

and in 3D

\[
(h, h_c W_z \otimes W_y \otimes A_x + h_c h_x W_z \otimes W_y \otimes W_z + h_c h_x A_z \otimes W_y \otimes W_z) u = b,
\]

where \( W_x, W_y \) and \( W_z \) are diagonal matrices with the Gaussian quadrature weights on the diagonal.

To solve this linear system at every time step, we use the conjugate gradient method. The reason why we prefer using this method over a direct method is, that as starting vector of this iterative method the electric potential \( \phi \) computed at the previous time step can be used. Therefore, the number of iterations is small, since, if short time steps are used \( \phi \) does not change dramatically. Moreover, the whole matrix is never constructed, a function is implemented which performs the action of this matrix applied on a vector. Thus, we use a completely matrix-free implementation.

If multiple compute nodes are used, only the boundary cells have to be transferred to the neighboring processes at each iteration. Therefore the method can be parallelized easily. Unfortunately, the matrix is singular since the system does not contain a constraint on the mean value of \( \phi \). In principle this is not a problem since the electric field is the gradient of the potential function \( \phi \) and therefore this constant shift vanishes. Nevertheless, it is difficult to construct a computationally cheap termination criteria in the CG method when the mean is growing continuously. Therefore, we embedded this condition in the CG method itself with almost no additional computational cost. The resulting CG method, which is mean preserving, is described in Algorithm 2.

**Algorithm 2. Mean preserving conjugate gradient method**

1. \( r_0 = b - A u_0, \text{mean}(u_0) = 0 \)
2. \( r_0 = r_0 - \text{mean}(r_0), p_0 = r_0 \)
3. \( \eta = (b + r_0)^T u_0 \)

while \( i \leq \text{MAXITER} \) do

    \[w_i, \text{mean}(w_i) = Ap_i,\]

    \[a_i = r_i^T r_i / p_i^T w_i,\]

    \[u_{i+1} = u_i + a_i p_i,\]

    \[r_{i+1} = r_i - a_i (w_i - \text{mean}(w_i))\]

    \[\eta = \eta + a_i r_i^T r_i\]

    if \( ||r_{i+1}||_2 < \text{tol} ||h||_2 \) then

        break

end while

To ensure that the dG Poisson solver is working properly, we test it first on a simple example with periodic boundary conditions,

\[-\Delta \phi(x) = \cos(0.5x)\]

In figure 20 we observe that the error is decreasing with order \( k + 1 \) (note that the dG Poisson solver converges with the same order as the dG advection solver).
Figure 20. Relative $L^\infty$ error as a function of the number of cells for the discontinuous Galerkin Poisson solver in 2d and 3d. In the Vlasov-Poisson simulations, we set the relative tolerance in the CG method below $10^{-3}$ which is sufficient as can be seen in Figures 7 and 8. Moreover, $\sigma$ is set to its minimum allowed value.

We observed, that when $\sigma$ is increasing, also the condition number of the matrix $A$ is increasing and more iterations are required to reach convergence. Additionally, we observe that convergence takes (almost) place within $\left( \prod_{i=1}^{d} N_i (k_i + 1) \right)^{1/d}$ iterations. Therefore, we set the maximum number of allowed iterations to this threshold times a safety factor.