SCALABLE

Project Title	SCAlable LAttice Boltzmann Leaps to Exascale
Project Acronym	SCALABLE
Grant Agreement No.	956000
Start Date of Project	01.01.2021
Duration of Project	36 Months
Project Website	www.scalable-hpc.eu

D5.1

Code generation for sparse data storage LBM kernels

Work Package	WP 5: Improving LBM solver versatility and advanced Extreme Scale SW technologies
Lead Author	Markus Holzer (Cerfacs)
Contributing Authors	Markus Holzer, Prof. Ulrich Rüde, Prof. Harald Köstler, Gabriel Staffelbach
Reviewed By	
Due Date	December 31, 2021
Date	
Version	1.0

Dissemination Level

- ☒ PU: Public
- ☐ PP: Restricted to other programme participants (including the Commission)
- ☐ RE: Restricted to a group specified by the consortium (including the Commission)
- ☐ CO: Confidential, only for members of the consortium (including the Commission)

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Deliverable Information

Deliverable	D5.1
Deliverable Type	Report
Deliverable Title	Code generation for sparse data storage LBM kernels
Keywords	
Dissemination Level	Public
Work Package	WP 5: Improving LBM solver versatility and advanced Extreme Scale SW technologies
Lead Partner	Cerfacs
Lead Author	Markus Holzer
Contributing Authors	Markus Holzer, Prof. Ulrich Rüde, Prof. Harald Köstler, Gabriel Staffelbach
Reviewed By	
Due Date	December 31, 2021
Planned Date	
Version	1.0
Final Version Date	

Disclaimer:

The opinions of the authors expressed in this document do not necessarily reflect the official opinion of the SCALABLE partners nor of the European Commission.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Code generation for sparse data storage LBM kernels

Markus Holzer, Prof. Ulrich Rüde, Prof. Harald Köstler, Gabriel Staffelbach

January 11, 2022

Contents

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

List of Figures

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

List of Tables

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Chapter

1 Introduction

With the rapid development of today's technologies and increasing access to high-performant supercomputers and clusters, the number of modelling and hardware features increases accordingly for numerical schemes. The lattice Boltzmann method (LBM) is no exception. Figure 1.1 [1] provides a small overview of features that are supported by the *lbmpy* framework.

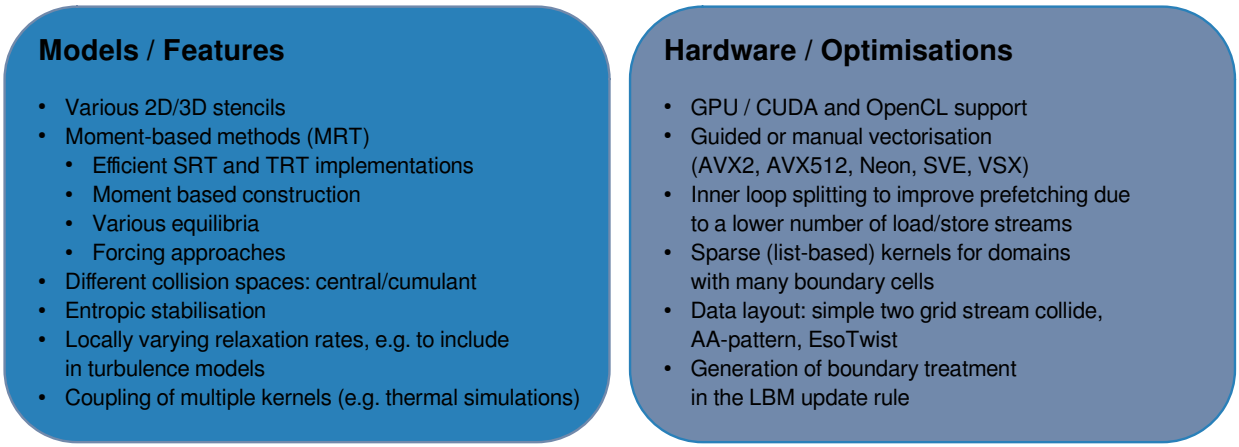


Figure 1.1.: Features for the LBM provided by *lbmpy*. The left hand side of the figure governs features concerning the modelling part or the mathematical description, while the right hand side aims for features in terms of different supported hardware or specific hardware optimisations.

On the modelling side, different LB stencils like D2Q9, D3Q19 or D3Q27 are commonly used (see appendix A). Already the mentioned stencils indicate simulations in two or three dimensions. On top of that, various approaches have been made in the literature to model the collision process. Among these, there are, e.g., moment-based methods like the SRT, TRT, MRT, or CLBM. But even more complex methods, like the cumulant or regularised LBM, are used in practical applications [3, 5]. Just from the features mentioned above, it is cumbersome to support all of them in a typical framework, written in a hardware-close language like C++ or Fortran. The reason for that is that these features need to be combinable with each other, and they need to be compatible with the MPI communication and boundary conditions. Only supporting two different stencils, e.g., D3Q19 and D3Q27, and a simple SRT scheme would already require two specialised LBM kernels or the introduction of a higher abstraction level. The latter, however, can diminish the computational performance. Additionally, one must provide an optimised scheme for the MPI communication of both stencils to ensure a minimum data transfer, and specific boundary conditions need to be written. To make matters worse, the LBM community uses different streaming schemes that have their respective advantages and disadvantages [8]. The most important here would be the pull and push patterns, the A-A pattern, and the Esoteric Twist. The last two are so-called in-place streaming patterns that have the advantage that no second temporary field of particle distribution functions (PDFs) is needed. However, this comes at the cost of more complexity. Only relying on in-place patterns is not possible. As these schemes come with a separate even and odd timestep, it is by no means trivial to use them in more complex settings like multi-physics, e.g., phase-field flows. Usually, a finite difference scheme is needed for these or truly compressible LBM schemes. Thus it would be favourable to support such optimisations when they can be applied while falling back to a simple scheme in more complicated scenarios. Nevertheless, MPI communication, boundary handling, and the LBM kernel would need to be implemented again for each combination.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

On the other side, all implemented compute kernels need to run on some hardware and thus need to be optimised for these. By looking at the CPU side, it is clear that many different architectures are using different instructions sets for vectorisation. The most important ones are SSE, AVX2 and AVX512 for Intel and AMD CPUs, but ARM-based CPUs using Neon or SVE instructions sets are gaining popularity. Although modern compilers can generate optimised code containing vectorisation, it is usually not possible to automatically enable specialised instructions like non-temporal stores that play an important role in the context of LBM simulations. Besides CPUs, accelerator hardware like NVIDIA or AMD GPUs is used and well suited for the LBM due to higher bandwidth. However, their kernels are distinct from their CPU counterparts, and need their own specialised implementation.

From the above features on the model and hardware side, the possibly most complicated was still not covered: the support of different data structures. Considering an LBM simulation, the optimal data structure highly depends on the problem description. The simplest way to store PDFs during the execution of LBM is in a linearized vector which represents the complete domain. With the direct addressing method, all cells are associated with their respective entries in this PDF vector, earning it the alternative name of the full-array method. The direct addressing makes it possible to locate neighbouring cells needed for streaming through simple index calculation. Compared to the direct addressing method, indirect addressing is exceptionally efficient for domains interspersed with many disjointed solid cells. This method only allocates memory space for fluid cells, resulting in a sparse PDF vector. During an initialization step, so-called index arrays are created where all necessary information regarding neighbourhood is stored: For q velocity directions, each fluid cell is assigned $q-1$ PDF indices that indicate where neighbouring PDFs are stored. The centre PDF value of a cell is the only one that is not streamed and can therefore be directly addressed.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Chapter

2 Code generation for indirect addressing LBM kernels

Chapter 2 covers the basic steps for the generation of LBM kernels using indirect addressing. The scope of deliverable 5.1 is to generate compute kernels for the LBM using an indirect addressing scheme, which is well suited for fluid simulations in sparse domains with many obstacle cells. In order to achieve the task, the code generation frameworks *lbmpy* [1] and *pystencils* [2] are used

An overview of the software stack is given in fig. 2.1 [4]. As a first stage, the equation set defining a certain LBM is derived using *lbmpy*. Inside *lbmpy* this is achieved by using *sympy* as a computer algebra toolbox. This means *lbmpy* exclusively acts on the modelling level, thus, on the level of the mathematical description. Just defining the mathematical model say defining the collison operator is insufficient to generate the complete kernel. They are only useful to generate a mathematical form using *sympy*. The final compute kernel loop needs additional information, namely, the data access pattern of the compute kernel. Until this stage, it was only necessary to use *sympy* for mathematical manipulation. However, additional information is needed for the generation of the compute kernels.

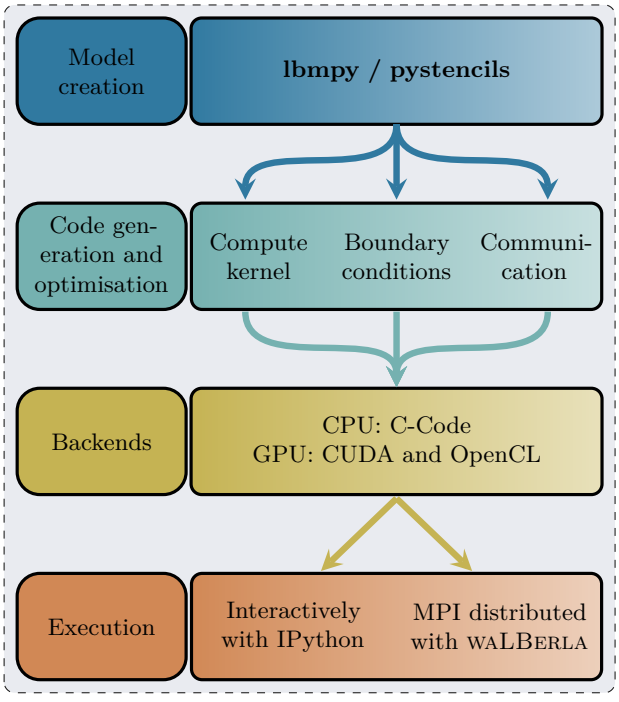


Figure 2.1.: Complete workflow of combining *lbmpy* and waLBerla for MPI parallel execution. Furthermore, *lbmpy* can be used as a stand-alone package for prototyping.

The missing information concerns the access of the data in the later compute kernel. To add this information to the derived equations *pystencils* is used. In *pystencils* the concept of a *Field* is implemented. This concept should be explained with a simple five-point stencil. The laplacian in an equidistant grid can be approximated with:

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

$$\nabla^2 f(x,y) \approx f(x-1,y) + f(x+1,y) + f(x,y-1) + f(x,y+1) - 4f(x,y) \tag{2.1}$$

Equation (2.1) could directly be found in the literature as a discretisation scheme. As the discretisation is derived for a specific grid, this information is also encoded in the equation. Here we can find the information in the arguments of the function f. Thus $f(x-1,y)$ expresses the access of the left neighbour point. This idea is encoded in the *pystencils Field*. The update scheme for eq. (2.1) would be defined in *pystencils* as:

$$\mathrm{dst}^0_{(0,0)} \leftarrow \mathrm{src}^0_{(-1,0)} + \mathrm{src}^0_{(1,0)} + \mathrm{src}^0_{(0,-1)} + \mathrm{src}^0_{(0,1)} - 4\mathrm{src}^0_{(0,0)} \tag{2.2}$$

Let us now look at one specific part of the above assignment. For example let us look at $\mathrm{src}^0_{(-1,0)}$. Firstly, it contains a special index $(-1,0)$, the so-called spatial index. It represents data access in a spatial direction, e.g., $(-1,0)$ represents the access of a two-dimensional data field at the left point. Hence, when looping over the shape of the two-dimensional Field, it can be represented as $pdf[i-1,j]$ where i,j are the loop counters. The superscript, in the example it is 0, refers to the index inside a cell. So, in this case, we refer to the zeroth value in the cell. Since eq. (2.1) is defined for a scalar field, only the zeroth entry in each cell can be obtained. If *src* were a Vector field, an individual cell would contain more than one data point. The idea of *pystencils* is now that each of the above field accesses acts as a *sympy* symbol. Thus as a mathematical variable. This means it is possible to apply all mathematical manipulations and optimisations provided by the *sympy* framework.

When the update scheme is defined in *pystencils* it is possible to generate the compute kernel. Furthermore, since the *pystencils Field* has the information of the field accesses, it is possible to directly determine which data points to send when dealing with a larger decomposed computation that is distributed among several compute nodes.

The generated compute kernel can be executed directly with *pystencils* in a Python environment. Additionally, it is also possible to print the kernels in a boilerplate class provided by waLBerla. Thus the cond generation can be seemingly used by waLBerla. waLBerla then brings the domain decomposition functionality so that large scale simulations would be enabled. However, the generation of the communication scheme is covered in deliverable 5.3 and thus should not be looked at here.

2.1. Code generation for direct addressing schemes

Firstly, the implementation of the direct scheme is revisited to understand which changes are necessary for the indirect addressing scheme. In *lbmpy*, kernels are expressed in terms of Assignments, e.g., for a simple SRT kernel, cf. eqs. (2.3) to (2.7). These include calculating the zeroth and first-order moments, i.e., macroscopic density and velocity. Since the switch from a direct to an indirect addressing scheme only affects how to access the data in the LBM kernel is not necessary to show the rest of the collision kernel here. Depending on the scheme, transformations from the PDF space to specific collision spaces, the collision, and back transformations would be necessary. However, once the data of the PDFs is loaded, the rest of the collision kernel is only numerical execution and, therefore, independent from the underlying data structure.

$$\mathrm{vel}_0 = \mathrm{pdfs}^8_{(-1,1)} + \mathrm{pdfs}^6_{(-1,-1)} + \mathrm{pdfs}^4_{(-1,0)} \tag{2.3}$$
$$\mathrm{vel}_1 = \mathrm{pdfs}^5_{(1,-1)} + \mathrm{pdfs}^1_{(0,-1)} \tag{2.4}$$
$$\rho = \mathrm{pdfs}^0_{(0,0)} + \mathrm{pdfs}^3_{(1,0)} + \mathrm{pdfs}^7_{(1,1)} + \mathrm{pdfs}^2_{(0,1)} + \mathrm{vel}_0 + \mathrm{vel}_1 \tag{2.5}$$
$$u_0 = -\mathrm{pdfs}^3_{(1,0)} - \mathrm{pdfs}^7_{(1,1)} - \mathrm{pdfs}^5_{(1,-1)} + \mathrm{vel}_0 \tag{2.6}$$
$$u_1 = -\mathrm{pdfs}^7_{(1,1)} - \mathrm{pdfs}^8_{(-1,1)} - \mathrm{pdfs}^2_{(0,1)} + \mathrm{pdfs}^6_{(-1,-1)} + \mathrm{vel}_1 \tag{2.7}$$

The code which is generated from eqs. (2.3) to (2.7) is shown in Listing 1. As expected for the two-dimensional case, a loop nest with two loops is generated. The range of both loops is from 1 to 64. In this example, the size of the data accessed at execution is prescribed at generation time. Thus the shape and strides of the PDF

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

field can be resolved directly at generation time. Note here that the data array enters the kernel as a simple C-pointer, which is why field accesses are resolved via pointer arithmetic. The generated compute kernel is passed further to a C-compiler which can apply further optimisations. However, it is also possible to directly generate optimisation like SIMD instructions or loop blocking into the kernel. A careful look at the benefits of this approach is not the subject of this deliverable but deliverable 5.3.

Listing 1: Generated C-code from eqs. (2.3) to (2.7). The PDF field is accessed directly via the loop counters. Thus the used access scheme is called direct addressing.

```
FUNC_PREFIX void kernel(double * RESTRICT const _data_pdfs)
{
   for (int64_t ctr_1 = 1; ctr_1 < 65; ctr_1 += 1)
   {
      double * RESTRICT _data_pdfs_11_28 = _data_pdfs + 66*ctr_1 + 34914;
      double * RESTRICT _data_pdfs_1m1_26 = _data_pdfs + 66*ctr_1 + 26070;
      double * RESTRICT _data_pdfs_10_24 = _data_pdfs + 66*ctr_1 + 17424;
      double * RESTRICT _data_pdfs_1m1_25 = _data_pdfs + 66*ctr_1 + 21714;
      double * RESTRICT _data_pdfs_1m1_21 = _data_pdfs + 66*ctr_1 + 4290;
      double * RESTRICT _data_pdfs_10_20 = _data_pdfs + 66*ctr_1;
      double * RESTRICT _data_pdfs_10_23 = _data_pdfs + 66*ctr_1 + 13068;
      double * RESTRICT _data_pdfs_11_27 = _data_pdfs + 66*ctr_1 + 30558;
      double * RESTRICT _data_pdfs_11_22 = _data_pdfs + 66*ctr_1 + 8778;
      for (int64_t ctr_0 = 1; ctr_0 < 65; ctr_0 += 1)
      {
         const double vel_0 = _data_pdfs_10_24[ctr_0 - 1] +
                              _data_pdfs_11_28[ctr_0 - 1] +
                              _data_pdfs_1m1_26[ctr_0 - 1];
         const double vel_1 = _data_pdfs_1m1_21[ctr_0] +
                              _data_pdfs_1m1_25[ctr_0 + 1];
         const double rho =   vel_0 + vel_1 +
                              _data_pdfs_10_20[ctr_0] +
                              _data_pdfs_10_23[ctr_0 + 1] +
                              _data_pdfs_11_22[ctr_0] +
                              _data_pdfs_11_27[ctr_0 + 1];
         const double u_0 =   vel_0 -
                              _data_pdfs_10_23[ctr_0 + 1] -
                              _data_pdfs_11_27[ctr_0 + 1] -
                              _data_pdfs_1m1_25[ctr_0 + 1];
         const double u_1 =   vel_1 -
                              _data_pdfs_11_22[ctr_0] -
                              _data_pdfs_11_27[ctr_0 + 1] -
                              _data_pdfs_11_28[ctr_0 - 1] +
                              _data_pdfs_1m1_26[ctr_0 - 1];
      }
   }
}
```

2.2. Code generation for indirect addressing schemes

Based on the direct addressing scheme, we will explain the indirect addressing scheme in the following. Equations (2.8) to (2.12) show its representation in equation form. The difference between the direct and the indirect addressing scheme is that the PDF field is not accessed directly via the loop counters anymore but through an index list that contains all accesses to the data. The index list only contains references to fluid nodes. Thus the PDF array only needs to contain the fluid nodes, saving the memory for non-fluid cells. It is not trivial to neglect the memory allocation for domain cells inside an obstacle in a direct addressing scheme while maintaining optimal performance. For example, the introduction of branches can lead to a compute kernel that is difficult to vectorise. Using an indirect addressing scheme, it is clear that a certain overhead

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

occurs for storing the index list. However, once the index list is created, the data can be accessed by a one-dimensional loop over the index list. Therefore, the data access is incredibly efficient, and no complicated logical constructions are necessary during the execution. Thus, as already shown in, e.g., [6], it is possible to achieve state-of-the-art performance results when using the indirect addressing scheme.

With this primary difference between the schemes, it becomes evident which changes in the abstract representation are necessary. In eqs. (2.3) to (2.7), a sub- and a superscript described the access of the PDF field. The index array, a one-dimensional symbolic *pystencils* Field that contains eight field accesses in each cell, now replaces the access via sub- and superscript. Further, notice that the subscript now contains all necessary information to resolve the field access at generation time. Thus the superscript can just be neglected by always setting it to zero. A cleaner solution would be to introduce a symbolic representation for sparse Fields, which no longer contain the superscript. However, this is subject to future work. The significant advantage of the symbolic representation of the complete collision operator is that it is possible to use substitution provided by *sympy*. The new symbols containing the indirect addressing information are just different *sympy* symbols. Hence, replacing all direct field accesses with indirect ones is easily possible before generating the compute kernel. Thus, after implementing the basic logic, it is applicable for all collision operators, targets, and other features provided by *lbmpy*.

$$\mathrm{vel}_0 = \mathrm{pdfs}^0_{idx_0^7} + \mathrm{pdfs}^0_{idx_0^5} + \mathrm{pdfs}^0_{idx_0^3} \tag{2.8}$$

$$\mathrm{vel}_1 = \mathrm{pdfs}^0_{idx_0^4} + \mathrm{pdfs}^0_{idx_0^0} \tag{2.9}$$

$$\rho = \mathrm{pdfs}^0_0 + \mathrm{pdfs}^0_{idx_0^2} + \mathrm{pdfs}^0_{idx_0^6} + \mathrm{pdfs}^0_{idx_0^1} + \mathrm{vel}_0 + \mathrm{vel}_1 \tag{2.10}$$

$$u_0 = -\mathrm{pdfs}^0_{idx_0^2} - \mathrm{pdfs}^0_{idx_0^6} - \mathrm{pdfs}^0_{idx_0^4} + \mathrm{vel}_0 \tag{2.11}$$

$$u_1 = -\mathrm{pdfs}^0_{idx_0^6} - \mathrm{pdfs}^0_{idx_0^7} - \mathrm{pdfs}^0_{idx_0^1} + \mathrm{pdfs}^0_{idx_0^5} + \mathrm{vel}_1 \tag{2.12}$$

As a next step, we take a look at the generated C-code, which results from eqs. (2.8) to (2.12). Once again, we only consider the calculation of the zeroth and first-order moment, which is usually the first step in the collision process of the LBM. As we can see in Listing 2, two pointers enter now the compute kernel. One points to the array containing the PDF values, while the other points to the array containing the field access which is represented by unsigned integers. As in the symbolic representation, both arrays are one-dimensional. We now encounter only a single loop over the size of the index list, whose entries access the data of the PDF field as expected.

2.3. Creation of the IndexList

Even though creating the index list is crucial, it is not directly covered here since deliverable 5.1 only targets the generation of the compute kernel. Thus, only a brief description follows, heavily based on [7].

lbmpy is not only a code generator for the waLBerla framework but also acts as a standalone package. Therefore, it must also provide the functionality to create the index array inside the package. Naïvely, this can be implemented directly in Python. However, NumPy cannot handle this creation as it is not describable as vector calculations. Since a pure Python implementation is by orders of magnitude too slow for larger domains, Cython was utilised for this purpose. It is important to note that *lbmpy* itself is intended only for simulations on a single node level. Therefore, the treated domain sizes in *lbmpy* remain small enough to employ a somewhat decent implementation. Transferring this logic to waLBerla, more thoughts need to be taken. On the other side, as described in [7], an essential first step for a parallel list creation would be the geometric decomposition of the domain. Since this is already in the backbone of waLBerla, it seems that an efficient parallel creation of the index array is compatible with the framework.

The implementation in *lbmpy* follows two steps. The first step is to loop over the flag array, which contains the information of boundary and fluid cells. The fluid cells get counted and ordered to form a one-dimensional index array in this step. The creation of this index array is then the second step. Here, depending on the streaming scheme, all required indices are appended to it. At this point, it is worth mentioning that no-slip and periodic boundary conditions can directly be encoded in the index array. No-slip conditions, which consist of a simple bounce-back scheme, can be resolved by storing the indices such that the direction flip of the PDF values happens directly inside the cell. The same holds for periodic boundary conditions. Thus both cases do not introduce any additional calculations and come for free from a computational point of view. This

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Listing 2: Generated C-code from eqs. (2.8) to (2.12). The PDF field is accessed indirectly via an index list. Thus the used access scheme is called indirect addressing.

```
FUNC_PREFIX void kernel(double * RESTRICT const _data_f, uint32_t * RESTRICT
    const _data_idx)
{
   double * RESTRICT _data_f_10 = _data_f;
   uint32_t * RESTRICT _data_idx_15 = _data_idx + 5*16384;
   uint32_t * RESTRICT _data_idx_13 = _data_idx + 3*16384;
   uint32_t * RESTRICT _data_idx_17 = _data_idx + 7*16384;
   uint32_t * RESTRICT _data_idx_14 = _data_idx + 4*16384;
   uint32_t * RESTRICT _data_idx_10 = _data_idx;
   uint32_t * RESTRICT _data_idx_11 = _data_idx + 16384;
   uint32_t * RESTRICT _data_idx_16 = _data_idx + 6*16384;
   uint32_t * RESTRICT _data_idx_12 = _data_idx + 2*16384;
   for (int64_t ctr_0 = 0; ctr_0 < _size_d_0; ctr_0 += 1)
   {
      const double vel0Term = _data_f_10[8*_data_idx_13[4*ctr_0]] +
                              _data_f_10[8*_data_idx_15[4*ctr_0]] +
                              _data_f_10[8*_data_idx_17[4*ctr_0]];
      const double vel1Term = _data_f_10[8*_data_idx_10[4*ctr_0]] +
                              _data_f_10[8*_data_idx_14[4*ctr_0]];
      const double rho = vel0Term + vel1Term +
                         _data_f_10[8*_data_idx_11[4*ctr_0]] +
                         _data_f_10[8*_data_idx_12[4*ctr_0]] +
                         _data_f_10[8*_data_idx_16[4*ctr_0]] +
                         _data_f_10[8*ctr_0];
      const double u_0 = vel0Term -
                         _data_f_10[8*_data_idx_12[4*ctr_0]] -
                         _data_f_10[8*_data_idx_14[4*ctr_0]] -
                         _data_f_10[8*_data_idx_16[4*ctr_0]];
      const double u_1 = vel1Term -
                         _data_f_10[8*_data_idx_11[4*ctr_0]] +
                         _data_f_10[8*_data_idx_15[4*ctr_0]] -
                         _data_f_10[8*_data_idx_16[4*ctr_0]] -
                         _data_f_10[8*_data_idx_17[4*ctr_0]];
      }
   }
}
```

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

optimisation is especially important for the no-slip boundary conditions since these are typically frequently encountered in practical simulations. It would also be possible to directly cover the boundary handling inside the compute kernel for a direct addressing scheme. However, it is not possible without introducing branches. A careful comparison of this technique and the list-based approach's performance follows in deliverable 5.3.

2.4. Numerical comparison

In the following, we compare the compute kernel generated with the direct and the indirect scheme on a numerical setup in a simple test case. Since the generation of boundary conditions and communication patterns are subject to deliverable 5.2, this test case is fully periodic and runs in a two-dimensional domain on a single node. For this purpose, we implemented a convected two-dimensional isentropic vortex (CoVo). As this example is already carefully described in deliverable 2.1, we only provide a brief description here. Further, it is vital to notice that an indirect addressing scheme cannot play its strength in this test case since it exclusively contains fluid cells. This means the direct addressing scheme is already the most optimal scheme here and thus the indirect addressing scheme would just produce an overhead because in addition to the PDF fields the index array needs to be allocated. In practice, however, introducing a second list, that stores the information of continuous fluid cells occurring in the domain, can reduce the size of the actual index list. After all, indices for parts of the domain where we find continuous fluid cells do not need to be stored because these can be accessed via direct addressing. Thus, it would act as a fallback case to the direct addressing. This approach is the so-called reduced indirect addressing approach [7]. Such optimisations are deferred to a later project stage.

The convected two-dimensional isentropic vortex can be generated using the following function:

$$\psi(x,y) = \Gamma \exp\left(-\frac{(x-x_c)^2 + (y-y_c)^2}{2R_c^2}\right) \tag{2.13}$$

with Γ and R_c the circulation and vortex radius, respectively. x_c and y_c correspond to the coordinates of the vortex centre. Following this, both pressure and velocity local components can be expressed as follows:

$$u = U_0 + \frac{\partial \psi}{\partial y}, v = V_0 - \frac{\partial \psi}{\partial x} \text{ and } P - P_0 = -\frac{\rho \Gamma^2}{2R_c^2} \exp\left(-\frac{(x-x_c)^2 + (y-y_c)^2}{R_c^2}\right) \tag{2.14}$$

Table 2.1 describes the exact setup to compare the direct and indirect addressing scheme.

For the parameter conversion between physical and lattice units, we choose a reference length of 0.1 and a reference velocity of 0.014099. The conversion factors are then defined as

$$C_l = \frac{l_{\text{ref}}}{l_{\text{lattice}}} \tag{2.15}$$

$$C_u = \frac{U_0}{u_{\text{lattice}}}. \tag{2.16}$$

Both simulations run for 1000 timesteps. Figure 2.2 and fig. 2.3 show the velocity magnitude after the initialisation as well as after 1000 timesteps for the direct and indirect addressing scheme, respectively.

From the visible results, it is clear that both simulations agree well. However, this is not enough to ensure that both simulations compute the same result. Therefore, the obtained velocity fields are compared in fig. 2.4 using their relative error

$$\epsilon = \frac{\|\mathbf{u}_{\text{indirect}} - \mathbf{u}_{\text{direct}}\|}{\|\mathbf{u}_{\text{indirect}}\|}. \tag{2.17}$$

The deviation between the velocity field obtained via the direct and the indirect addressing approach is calculated every 20 timesteps. At the beginning of the simulation, the deviation is zero, which has to be the case since both velocity fields are initialised in the same way. However, progressing further in the simulation,

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

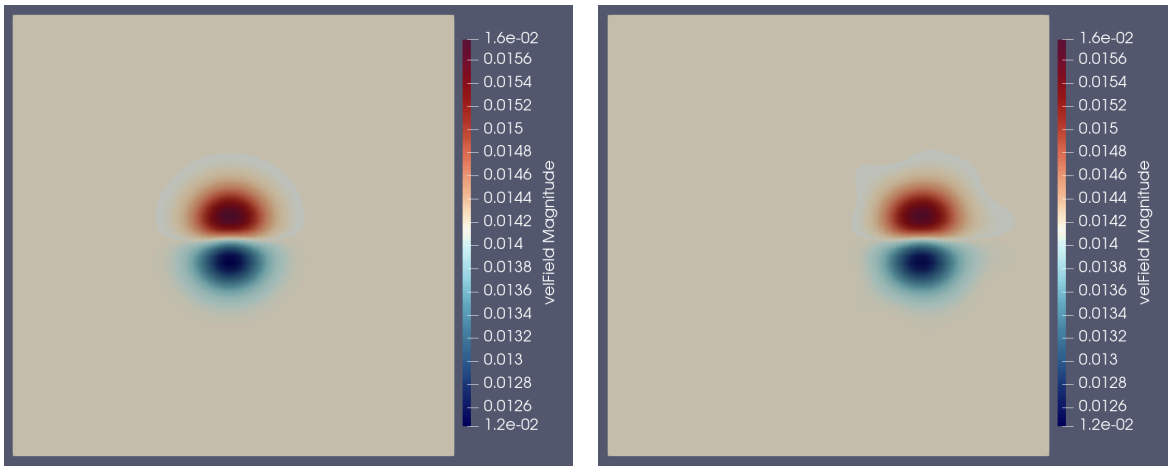
Table 2.1.: Characteristics of the CoVo use case

Convected Vortex test case	
Domain bounding box	-0.05 x 0.05 m
Resolution	64x64
reference length	0.1
Convection Speed	U_0= 170 $m.s^{-1}$;V_0= 0
Lattice velocity	0.014099
Circulation	Γ= 34.728 $m^2.s^{-1}$
Lattice circulation	0.0028802
Radius	R_c= 0.005 m
Lattice Radius	3.2
Reynolds number	1.082e6
Relaxation rate	1.99999
Collision operator	Cumulant
Lattice stencil	D2Q9

a small deviation is noticeable. Being in the order of 10^{-14}, it is not concerningly large. However, since both schemes execute the same setup with the same algorithm, the obtained results would be expected to be completely similar.

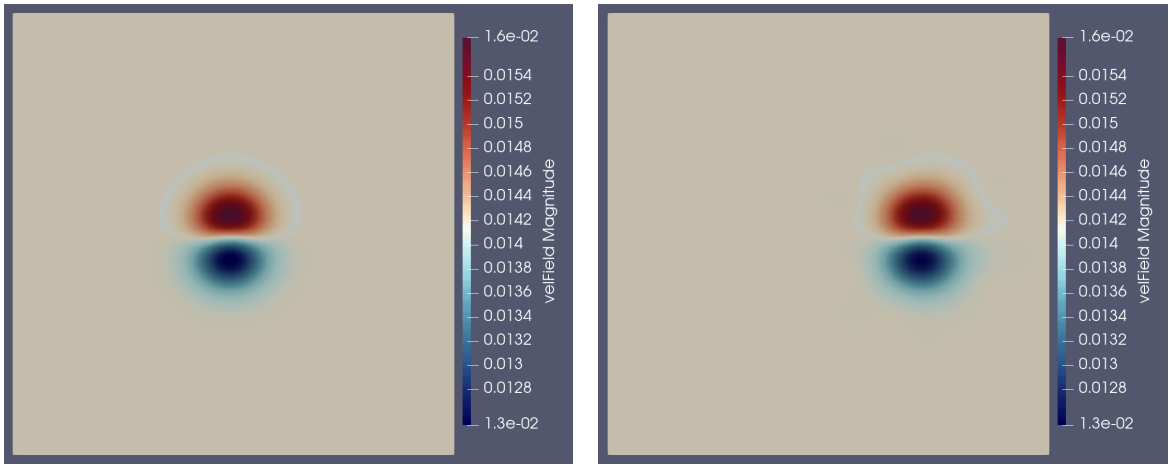
A possible explanation of the obtained deviation might be the execution order of the calculation. As stated in section 2.3, the first step to obtaining the index list is by enumerating the fluid cells. Thus the execution of the streaming and collision later follows this enumeration. Slight deviations can occur if this is not the same order as in the direct addressing scheme. A closer, careful look at the origin of this deviation is subject to future work.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.



(a) Initialisation (b) Timestep 1000

Figure 2.2.: Simulation of the CoVo test case with direct Addressing



(a) Initialisation (b) Timestep 1000

Figure 2.3.: Simulation of the CoVo test case with indirect Addressing

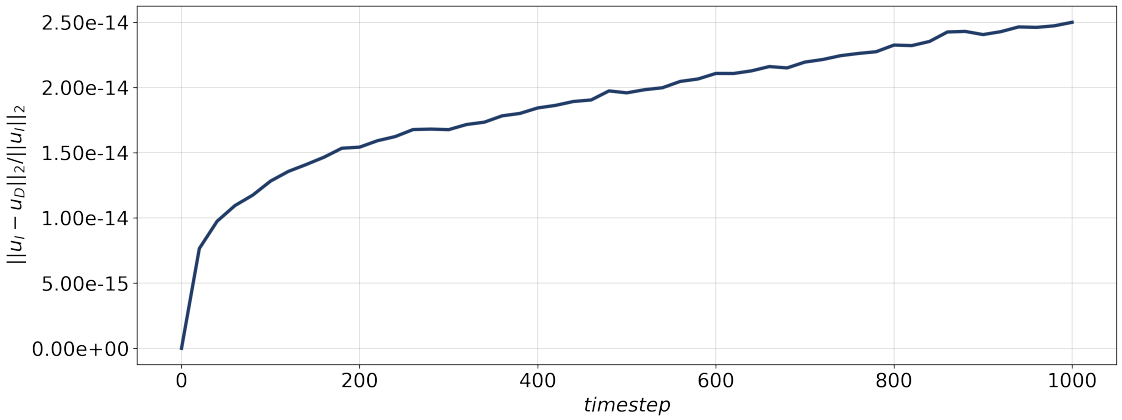


Figure 2.4.: Difference between the velocity field obtained from a simulation of the CoVo test case using a direct addressing scheme and using an indirect addressing scheme.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Chapter

3 Conclusion

In deliverable 5.1, we elucidated the successful generation of LBM kernels using an indirect addressing scheme. Due to the usage of the code generation framework *lbmpy*, it was possible to directly combine the indirect addressing scheme with all methods of *lbmpy*, which was demonstrated in section 2.4 by using the cumulant collision operator with the indirect addressing scheme. Furthermore, it is possible out of the box to obtain kernels for accelerator hardware like GPUs. While the results on such hardware already look promising compared to the results obtained by direct addressing, a careful analysis will be performed in deliverable 5.3. The same holds for the performance on CPU architectures. Here, necessary integer intrinsics need to be introduced to *pystencils* to ensure a SIMD vectorisation of the compute kernels. Without the specialised instructions, it is not possible to gather performance results with the LBM kernel using the indirect addressing that is compatible with a classical LBM kernel using direct addressing.

Furthermore, both schemes need to be compared in memory consumption since the indirect addressing scheme introduces an additional overhead with the index list. However, to gather a concise picture, the generation of the boundary kernels needs to be done first.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Chapter

A LB stencils

In appendix A the velocity sets for different three-dimensional LB stencils are shown.

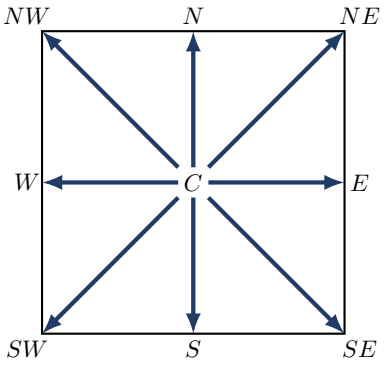


Figure A.1.: The discrete velocities for the D2Q9 stencil.

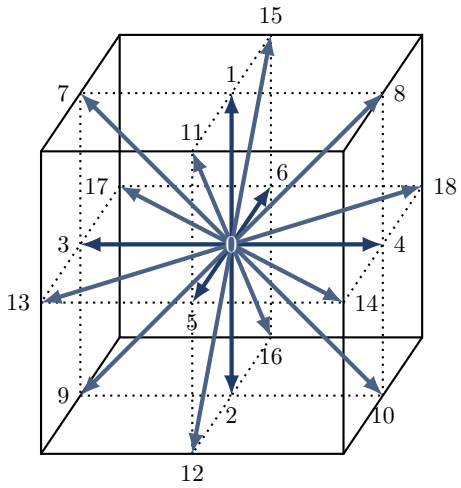


Figure A.2.: The discrete velocities for the D3Q19 stencil.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

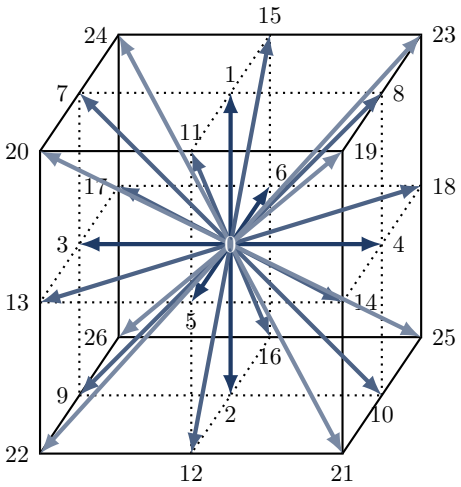


Figure A.3.: The discrete velocities for the D3Q27 stencil.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Bibliography

[1] Martin Bauer, Harald Köstler, and Ulrich Rüde. "lbmpy: Automatic code generation for efficient parallel lattice Boltzmann methods." In: *Journal of Computational Science* 49 (2021), p. 101269. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2020.101269.

[2] Martin Bauer et al. "Code Generation for Massively Parallel Phase-Field Simulations." In: Association for Computing Machinery, 2019. DOI: 10.1145/3295500.3356186.

[3] Martin Geier et al. "The cumulant lattice Boltzmann equation in three dimensions: Theory and validation." In: *Computers & Mathematics with Applications* (2015). DOI: 10.1016/j.camwa.2015.05.001.

[4] M. Holzer et al. "Highly efficient lattice Boltzmann multiphase simulations of immiscible fluids at high-density ratios on CPUs and GPUs through code generation." In: *The International Journal of High Performance Computing Applications* 35.4 (2021), pp. 413–427. DOI: 10.1177/10943420211016525.

[5] Jérôme Jacob, Orestis Malaspinas, and Pierre Sagaut. "A new hybrid recursive regularised Bhatnagar–Gross–Krook collision model for Lattice Boltzmann method-based large eddy simulation." In: *Journal of Turbulence* (2018). DOI: 10.1080/14685248.2018.1540879.

[6] M. Wittmann et al. "Lattice Boltzmann benchmark kernels as a testbed for performance analysis." In: *Computers & Fluids* (2018). DOI: 10.1016/j.compfluid.2018.03.030.

[7] Markus Wittmann. "Hardware-effiziente, hochparallele Implementierungen von Lattice-Boltzmann-Verfahren für komplexe Geometrien." doctoralthesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2016.

[8] Markus Wittmann et al. "Comparison of different propagation steps for lattice Boltzmann methods." In: *Computers & Mathematics with Applications* (2013). DOI: 10.1016/j.camwa.2012.05.002.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.