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 |

# D5.3

# Efficient architecture-specific compute kernels

| | |
|---|---|
| **Work Package** | WP 5: Improving LBM solver versatility and advanced Extreme Scale SW technologies |
| **Lead Author** | Philipp Suffa (FAU) |
| **Contributing Authors** | Philipp Suffa, Markus Holzer, Prof. Ulrich Rüde, Prof. Harald Köstler |
| **Reviewed By** | |
| **Due Date** | March 31, 2023 |
| **Date** | |
| **Version** | 0.1 |

**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)

Copyright © 2021 – 2023, The Scalable Consortium

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** | D5.3 |
| **Deliverable Type** | Report |
| **Deliverable Title** | Efficient architecture-specific compute kernels |
| **Keywords** | |
| **Dissemination Level** | Public |
| **Work Package** | WP 5: Improving LBM solver versatility and advanced Extreme Scale SW technologies |
| **Lead Partner** | FAU |
| **Lead Author** | Philipp Suffa |
| **Contributing Authors** | Philipp Suffa, Markus Holzer, Prof. Ulrich Rüde, Prof. Harald Köstler |
| **Reviewed By** | |
| **Due Date** | March 31, 2023 |
| **Planned Date** | |
| **Version** | 0.1 |
| **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.*

# Efficient architecture-specific compute kernels

Philipp Suffa, Markus Holzer, Prof. Ulrich Rüde, Prof. Harald Köstler

April 3, 2023

# Contents




*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

The goal of tasks 5.1 [6] and 5.2 [9] was to extend the code generation pipeline of *lbmpy* to support a full CFD simulation, which can be run with sparse data kernels. That means that we integrated sparse LBM kernels as well as sparse boundary kernels and communication kernels into the generation pipeline.

The goal of task 5.3 is to optimize these sparse kernels to achieve better performance results on CPUs as well as on GPUs. Therefore, a description of the automatic code generation for architecture-specific sparse data kernels will be given in chapter 2.

The first optimization for the sparse data kernels is an in-place streaming pattern. The implementation of an in-place streaming pattern, here the AA-pattern, reduces the amount of memory needed for the simulation and, more importantly, reduces the number of memory accesses and thus increases the performance for the LBM on CPUs, as it is shown in chapter 3.

Furthermore, in chapter 4, we utilized communication hiding on GPUs to achieve greater scaling efficiency on this hardware.

Lastly, we demonstrate the increase in performance of the optimized generated sparse kernels. Therefore, we integrate the generated sparse kernels in our massive parallel multi-physics framework waLBerla to enable parallel execution with great scalability [5]. Then we compare the sparse kernels with and without optimizations by running a turbulent channel scenario, which is one of the target applications of the SCALABLE project. These scaling runs are made on the CPU cluster Juwels-Cluster [1] as well as on the GPU cluster Juwels-Booster [1].

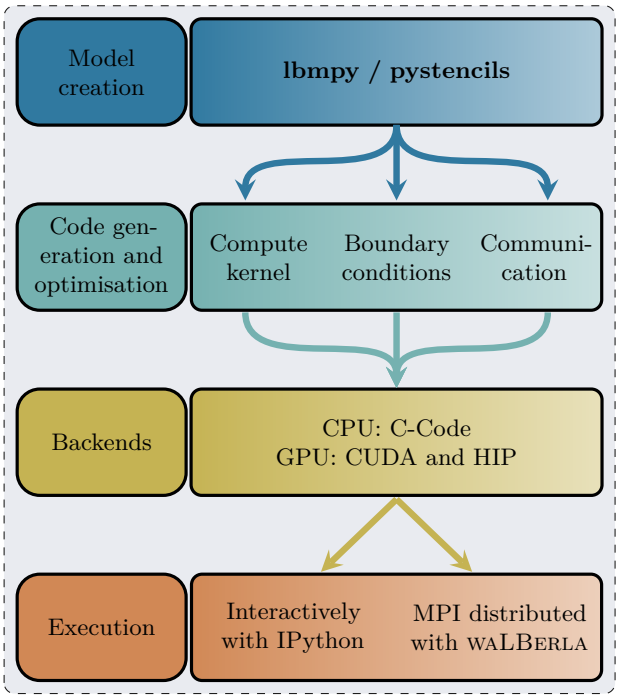


Figure 1.1: Complete workflow of combining *lbmpy* and waLBerla for MPI parallel execution.

*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 architecture-specific sparse data kernels

In this section, we generally describe how architecture-specific sparse data kernels are generated with our code generation pipeline, which is separated into different layers to guarantee strong modularity. An overview is shown in Figure 2.1 and will be described in more detail in the following.

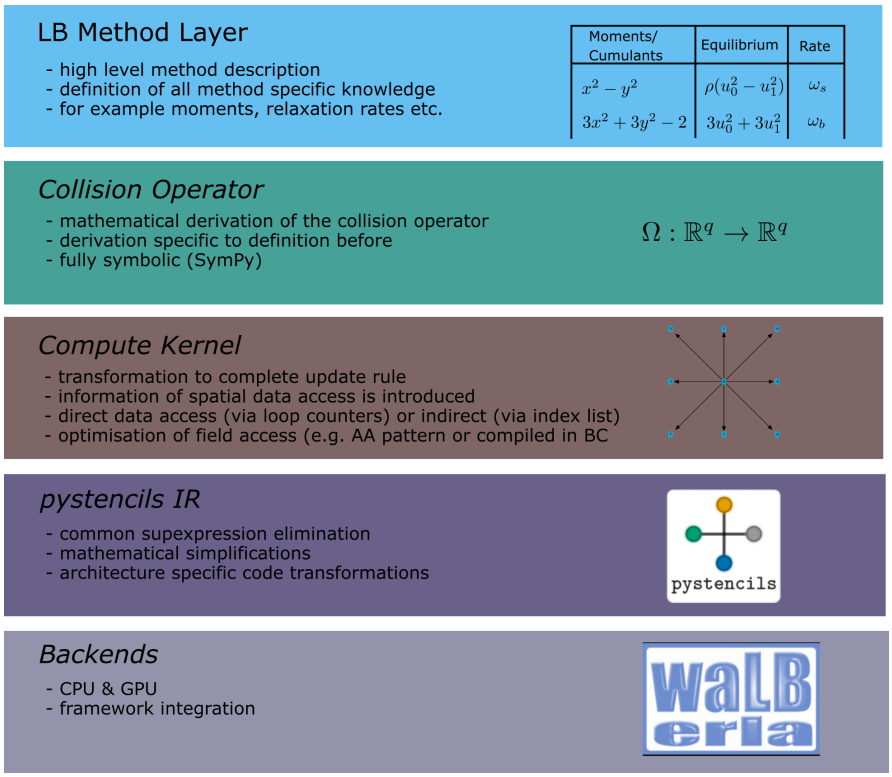


Figure 2.1: Overview of different software layers to start from the LBM modeling layer and end with architecture-specific code that can be combined in existing HPC software. Note here that in the compute kernel creation layer, the necessary information for direct or indirect addressing is introduced [3].

## 2.1 LB method layer

The top layer of our code metaprogramming pipeline is the description of the LB method that is later used to obtain the solution of the simulation. This layer is consequently called the LB method layer. At this stage, all LBM-specific parts are described. This includes the lattice stencil (e.g. D2Q9, D3Q19 ...), the collision space (e.g. moment space, central moment space, cumulant space), stabilization techniques like entropic conditions

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

or recursive regularisation, forcing schemes and additional advanced options like the usage of turbulence or Non-Newtonian models.

With this description, an LB collision operator can be fully defined. Since the specification is given in symbolic form, we can derive all information on the mathematical level from the continuous Maxwell-Boltzmann equilibrium. Also, all transformations to the representative collision spaces ca be performed automatically. This together creates a flexible powerful system for the user of waLBerla. Note that a typical user of waLBerla will only have to work on this level. The transformations on the following layers are typically performed hidden from theuser in the backend of the software.

## 2.2 Collision Operator

As a next step, the symbolic collision operator is created. The collision operator is stored as a *pystencils* `AssignmentCollection`. The `AssignmentCollection` is essentially a list of assignments needed for the compute kernel generated later. Each of these assignments contains a right-hand side, which is a symbolic expression that needs to be evaluated to obtain a value on the left hand side. In the context of LB methods, the `AssignmentCollection` contains a set of $q$ pre-collision particle distribution functions (PDFs) that occur on the right-hand sides of the assignments and a set of $q$ post-collision PDFs that will be on the left-hand sides of our assignments. In order to complete the compute kernel, several transformation steps (to get to the representative collision space and back), as well as the relaxation towards the equilibrium (the actual collision), is needed. We describe this as additional assignments in our `AssignmentCollection`. Additionally, terms are added e.g., to apply forcing terms in the collision process.

Up to this stage, we have a purely symbolically defined `AssignmentCollection` that has no neighbor information yet. Thus it lacks essential information how to obtain an actual compute kernel in a lower-level language. It is important to note that on this level already many optimizations for FLOP reductions can be performed. Mostly this concerns mathematical rewriting that happens on single symbolic expressions. However, since all expressions are combined in a list structure, it is also possible to perform mathematical optimizations on the whole list, like the elimination of common terms or the insertion of constants (like the relaxation parameter or other model-specific values).

## 2.3 Compute Kernel

As a next step, we create the compute kernel. Similar to the collision operator, we still have a list of assignments to form the *pystencils* `AssignmentCollection`. However, in the collision operator layer, no spatial information was encoded yet. This changes now by replacing individual symbols with *pystencils* `Fields`. The *pystencils* `Field` essentially is still a symbolic variable but with spatial information in the form of index notation. Thus different access patterns, like the pull-streaming pattern or the AA-pattern, will be introduced at this stage. This is done with simple substitutions mapping *sympy* `symbols` to *pystencils* `Fields`.

Compute kernels for a sparse data structure differ from kernels for a dense data structure just by their access pattern. Unlike obtaining the field access from the loop counters as direct addressed compute kernels would do, indirectly addressed kernels get their spatial access from a list of indices, giving it their name list-based compute kernels. Thus this is the only layer in our metaprogramming pipeline where changes need to be done according to the later-used data structure.

## 2.4 Pystencils

Now that the compute kernel is fully defined in an abstract symbolic manner, we create an abstract syntax tree (AST) in the *pystencils* intermitted representation (IR). In this tree representation, we introduce architecture-specific AST nodes. For example, in the case of compute kernels created for CPUs, this would be a loop nest around the `AssignmentCollection`. This loop nest will be automatically created based on the spatial access information given in the *pystencils* `Fields` that are contained in the `AssignmentCollection`. Once the loop nest is created, loop based optimizations like spatial blocking or OpenMP parallelization can be employed. Furthermore, constant evaluations can be moved out of the loop nest to reduce the computational cost.

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

For kernels that are meant to run on accelerators like NVIDIA GPUs, guard statements need to be introduced around the `AssignmentCollection`. Essentially this ensures the correctness of the array access in the later execution and defines the loop counters for the GPU grid.

Creating the AST also involves the introduction of data types based on the C-programming standard. Thus in this level also bandwidth-reducing optimizations can be introduced by introducing casts between the stored information and the information computations are executed on (mixed precision computations) [7].

## 2.5 Backends

Finally, the intermediate representation of the compute- and boundary kernels are transformed either by the C or the CUDA backend. The task of the backend is to print the AST as a C-function with a clearly defined interface. Each function takes raw pointers for array accesses together with their representative shape and stride information as well as all remaining free parameters. Examples of these free parameters would be values for the relaxation rates or forces etc. This simple and consistent interface makes it possible that the kernels can be called from a large variety of languages because it is usually possible to call C-functions from most languages. In the case of the Python language, it is possible to call C-functions using the Python C-API. Furthermore, many optimized Python packages like NumPy are already written in C to guarantee performance. Thus integrating these with our generated kernels is rather easy to do. Thus, we can provide a powerful interactive development environment by utilizing *lbmpy*/*pystencils* as stand-alone packages where the generated kernels are provided as Python functions via Pythons C-API.

Additionally, this simple low-level interface provided by the *pystencils* backends makes it possible to combine the highly optimized compute kernels with existing HPC frameworks like the multiphysics framework waLBerla. In this case, waLBerla provides the necessary boilerplate code to integrate the kernels nicely in the framework as well as all additional functionality needed to run complex simulations at large scales. This includes a domain decomposition mechanism via a forest of octrees that is decomposed with the message passing interface (MPI), functionality for complex meshes in the form of STL files, and parallel I/O to enable post-processing for large-scale simulations. More details about the whole tool-chain can be found in [4, 3, 6, 9].

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

*Chapter*

# 3 In-place streaming pattern for sparse data kernels: AA-pattern

The most common streaming patterns for LBM are the two-grid algorithms, where either all PDFs of a cell are pushed into the neighbor cells (push scheme) or all PDFs are pulled from the neighbor cells (pull scheme). These algorithms have in common that a temporary PDF field is needed to get correct results. This is because PDFs are stored in a different position than where they are read from. As it is shown in Figure 3.1, these two-grid streaming patterns read from PDF field A, then they propagate (push/pull) the PDFs and, lastly, they store the propagated results at a different location in the temporary PDF field B. After the propagation, a field swap of fields A and B is needed.

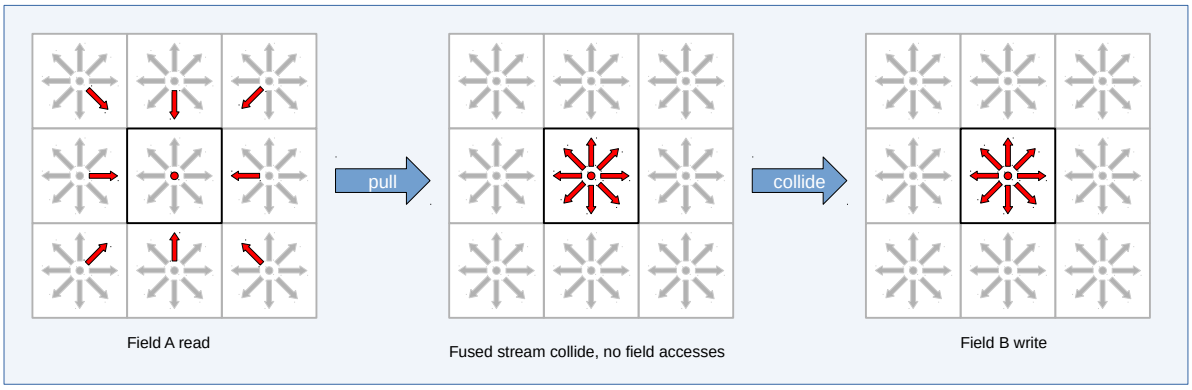


Figure 3.1: Pull scheme: PDFs are read from field A and, after the fused stream-collide step, they are written to field B.

The in-place streaming AA-pattern [2], on the other hand, enables writing and storing PDF values in the same positions of the PDF field, so it can read from field A and write to field A without risking data dependencies. This means that no temporary PDF field B is needed anymore, which saves memory.

This is achieved by introducing two alternating streaming time steps, as it is shown in Figure 3.2. In the ”odd” time step, PDFs are read from field A and pulled to the current cell. Then, the collision is performed, and the resulting PDFs are pushed back to the neighbor cells to the positions where they were read from on field A. This means that the odd time step consists of two propagations and one collision, and the reads and writes of the PDFs take place at the same positions on the same field A. But this also means that after the second propagation, the PDFs are somehow in the wrong position of the stencil, as $cell_{x,y}$ pushes its PDF in direction $East$ to $cell_{x+1,y}$ at stencil direction $West$. So we have to take care that PDFs are always in the opposite stencil position after an odd time step. This matters if we want to calculate macroscopic values or if we want to communicate with neighbor MPI blocks after an odd time step.

The second time step, the ”even” time step, only consists of one collision, but the PDFs needed for the collision have to be read from the opposite stencil positions to get correct macroscopic value calculations and similar. As there is no propagation in the even time step, no neighboring information is needed there, and so there is also no need to access the pull index list, which saves memory access, as it will be discussed in the following.

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

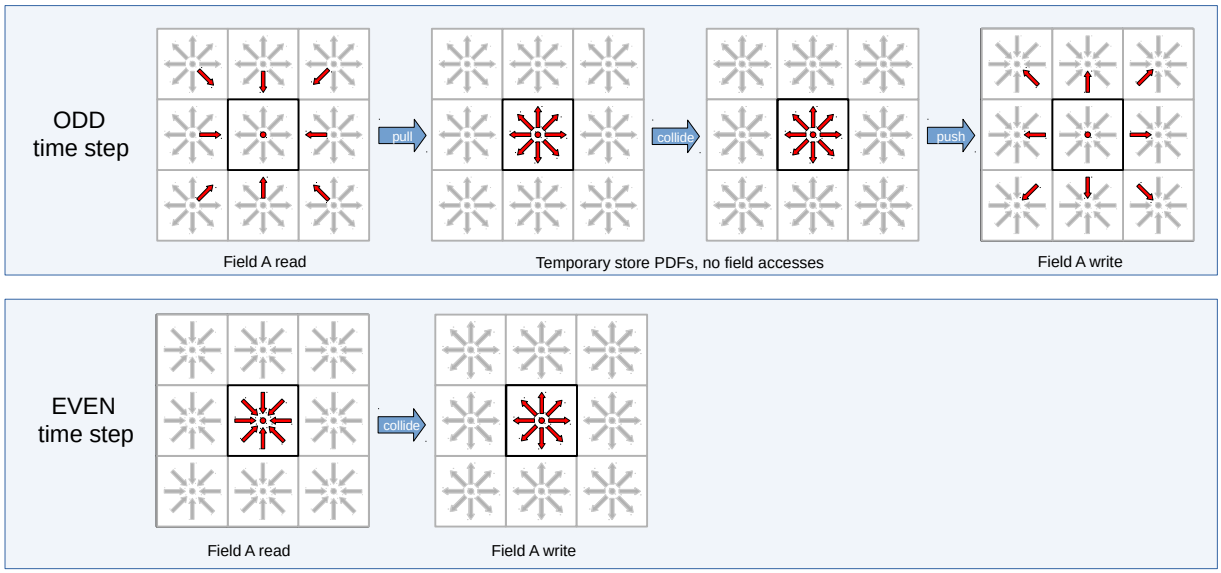


Figure 3.2: AA-pattern: In the "odd" time step, a pull streaming is done, followed by a collision and a push streaming step. As the PDFs are pushed to the same positions as they are read from, the PDFs are stored in the opposite stencil directions. In the "even" time step, only one collision is done, where the PDFs have to be written from the opposite stencil direction to achieve correct macroscopic values.

After one odd and one even time step, the PDFs are again stored in their right positions, and the outcome is the same as after two push or pull time steps.

Nevertheless, the AA-pattern not only saves memory, but the more significant benefit is the reduction of memory accesses in the LBM kernels. This benefit regarding memory accesses is shown in Figure 3.3 and Figure 3.4. For the Pull-pattern in direct addressing kernels, $3 \cdot q$ memory accesses are needed, where $q$ is the stencil size, usually D3Q19 or D3Q27. One memory access is needed for the read of field A, one is needed for the write on field B, and the third one is a "write allocate B", which occurs, if the data of the PDF of field B is not already stored in the CPU cache, and therefore has to be loaded into the cache to be written on. This memory access we want to avoid by utilizing an in-place streaming pattern. There, the data is already in the cache because we read and write on the same PDF positions on the same PDF field. By this we avoid the cache miss and end up with $2 \cdot q$ memory accesses, as it is shown in Figure 3.4.

| Direct addressing | | | Indirect addressing | | |
|---|---|---|---|---|---|
| # | Type | Comment | # | Type | Comment |
| $q$ | PDF | Loads $A$ | $q$ | PDF | Loads $A$ |
| $q$ | PDF | Write allocate $B$ | $q-1$ | IDX | Stores into $B$ |
| $q$ | PDF | Stores $B$ | $q$ | PDF | Write allocate $B$ |
| | | | $q$ | PDF | Stores $B$ |
| $\Sigma = 3q$ PDFs | | | $\Sigma = 3q$ PDFs $+ (q-1)$ IDXs | | |

Figure 3.3: Memory accesses for the Pull-pattern for direct and indirect addressing kernels

For indirect addressing kernels, the memory accesses of the AA-pattern has even more advantages. For the pull-pattern, in addition to the PDF list, also the index list has to be read to get the pull accesses for the propagation step, which adds a $(q-1)$ to our count of memory accesses. It is $q-1$ because pull indices have to be read for all stencil directions but the center direction. To sum up, we need $3 \cdot q + (q-1)$ memory accesses for sparse LBM kernels with the pull streaming pattern.

For the AA-pattern on the other side, we only need neighboring information in every second (odd) time step, because on even time steps we only compute cell-local, and therefore no neighboring information in needed. So the memory accesses for the index list can be halved to $\frac{(q-1)}{2}$. Therefore, we end up with $2 \cdot q + \frac{(q-1)}{2}$ memory accesses for sparse LBM kernels with the AA streaming pattern. Therefore, the AA-pattern reduces the memory accesses compared to pull-pattern by $1 - \frac{3 \cdot q + (q-1)}{2 \cdot q + (q-1)/2} \sim 37\%$ and, because most LBM codes are usually memory bound, increases the performance of the LBM in the optimal case kernels by the same amount.

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

| Direct addressing | | | Indirect addressing | | |
|---|---|---|---|---|---|
| # | Type | Comment | # | Type | Comment |
| | | *Even time-steps* | | | *Even time-steps* |
| $q$ | PDF | Loads | $q$ | PDF | Loads local |
| $q$ | PDF | Stores | $q$ | PDF | Stores local |
| | | *Odd time-steps* | | | *Odd time-steps* |
| $q$ | PDF | Loads | $q$ | PDF | Loads remote |
| $q$ | PDF | Stores | $q$ | PDF | Store remote |
| | | | $(q-1)$ | IDX | Access neighbors |
| $\Sigma = 2q$ PDFs | | | $\Sigma = 2q$ PDFs $+ (q-1)/2$ IDXs | | |

Figure 3.4: Memory accesses for the AA-pattern for direct and indirect addressing kernels

This performance boost can only be achieved on CPUs, as GPUs do not work with caches in that sense, so no ”write allocate” on the cache can be avoided and only half the memory accesses for the index list can be saved, with results in a theoretical performance increase of $1 - \frac{2 \cdot q + (q-1)}{2 \cdot q + (q-1)/2} \sim 16\%$.

## 3.1 Boundary and communication kernels

As was already mentioned above, special care must be taken for most kernels because of the alternating time steps of the AA pattern, which also change the store directions of the PDFs.

Therefore, we also need two alternating boundary kernels as well as packing kernels for the communication to match the behavior of the LBM kernels.

For boundary kernels, for example for velocity bounce back or pressure boundaries, the ”even” boundary step behaves similarly to a boundary kernel with a two-grid streaming pattern. So PDFs are read and macroscopic values for the equilibrium are calculated on the neighboring fluid cell next to the boundary cell, marked with blue arrows in Figure 3.5, and the result of the calculation is written to the PDFs on the boundary cell, in Figure 3.5 the red arrow in direction East. The ”odd” boundary kernel, on the other side, behaves quite differently. Here the PDFs for the boundary condition calculation are written from the pull indices of the neighboring fluid cell, and the result is stored on the neighboring fluid cell and not on the boundary cell. This is necessary because after the ”odd” boundary step an ”even” LBM step follows, which has no access to the boundary cell, as it only calculates its collision cell-local.

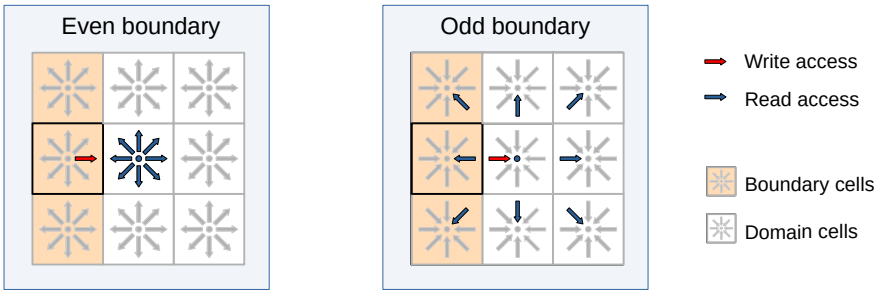


Figure 3.5: An example boundary condition, which runs on the mid left boundary cell and calculates the PDF in direction East. ”Even” boundary steps read PDF values for macroscopic value calculations from the cell of the PDF field next to the boundary cell and write the value on the PDF of the boundary cell, as it is also done in two-grid streaming pattern. ”Odd” boundary steps read PDFs from the index list of the domain cell next to the boundary cell, and write on the PDF in the direction ”West” of the fluid cell on the domain.

As shown in Figure 3.6, also for communication two different kernels have to be generated. For the ”even” communication kernel, as usual, PDFs from cells next to the MPI interface are packed into an MPI buffer and unpacked from the buffer into a ghost-layer, as it is explained in [9]. The ”odd” communication kernel, on the other hand, packs PDFs from the ghost layers into the MPI buffer and unpacks these data from the buffer to the PDF field, which results in the opposite behavior of the even communication kernel. This is necessary

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

because the odd LBM step pushes data to the ghost layers, and this information has to be communicated with block neighbors.

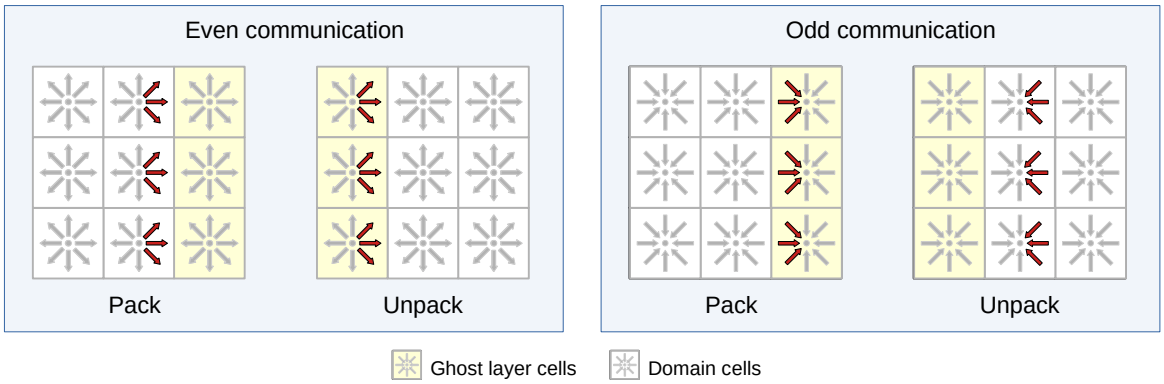


Figure 3.6: Even communication packs PDFs of the domain into an MPI buffer, and unpacks them into ghost layers, as usual. Odd communication packs PDFs from ghost layer cells into MPI buffer and unpacks them into domain cells.

## 3.2 Scaling benchmark results

In Figure 3.7, a scaling graph of the Pull-pattern vs. the AA-pattern on Juwels CPU cluster is shown, where one node consists of 48 CPU cores. The tested scenario is a turbulent channel with velocity bounce-back on west boundary, pressure outflow conditions on east boundary and no-slip boundaries in all other directions. It is shown that we nearly achieved the expected increase in performance for the AA-pattern on CPU. On average, the performance increase is about 33.77% MLUPS per core, close to the expected increase of $\sim 37\%$.

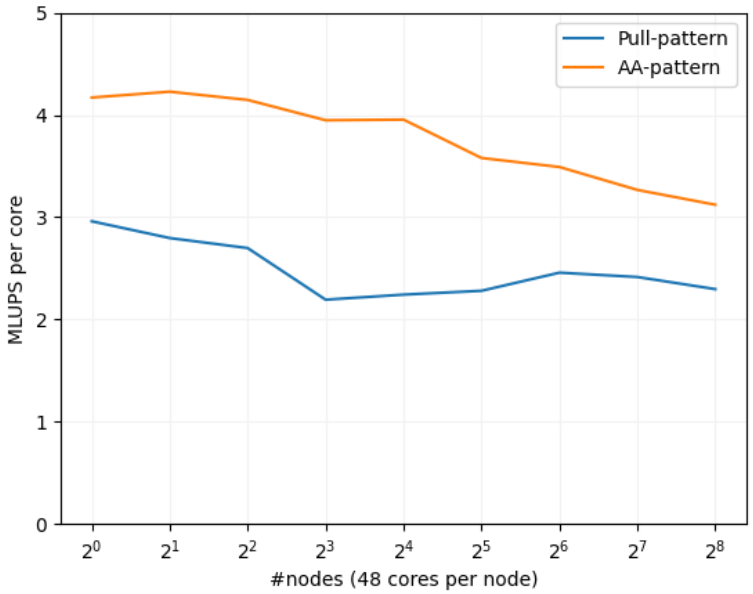


Figure 3.7: Pull-pattern vs AA-pattern on CPU cluster Juwels-Cluster for turbulent channel with D3Q27, cumulant method and $64^3$ cells per core.

The same scenario is evaluated on the GPU cluster Juwels-Booster. Figure 3.8 indicates that there is no real performance gain for the AA-pattern on GPUs. The average performance gain over all numbers of GPUs is $-2.67\%$, so the AA-pattern on average performs slightly worse than the Pull-pattern. We obtain from Figure 3.8, that for up to 8 GPUs the use of the AA-pattern results in slightly better performance than the

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

Pull-pattern, while for runs with a higher number of GPUs the MLUPS per core gets a bit worse. This is, because communication becomes the bottleneck of the simulation for a higher number of GPUs, and therefore savings in the form of memory accesses have a lower effect on the overall performance. Nevertheless, as discussed before, no temporary PDF field has to be stored for the AA-pattern, and therefore the AA-pattern can still be a good idea to save memory for runs on GPUs.

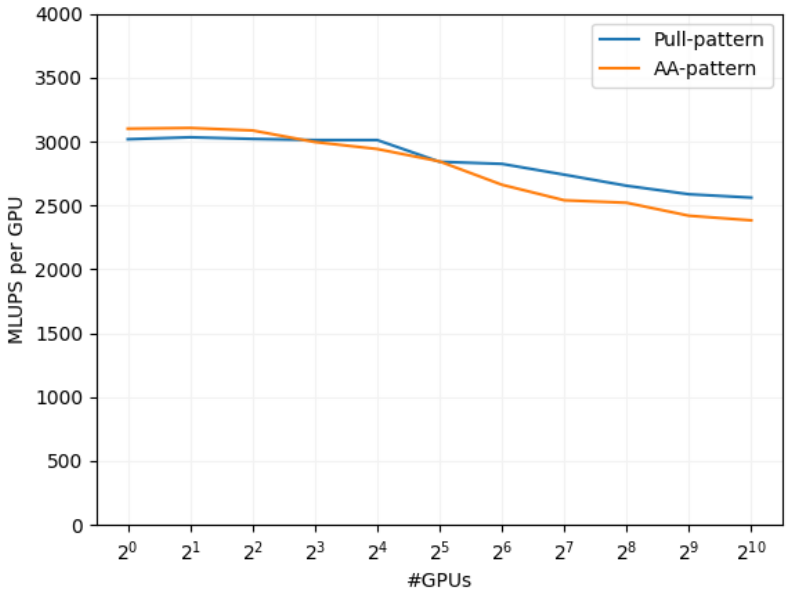


Figure 3.8: Pull-pattern vs AA-pattern on GPU cluster Juwels-Booster for turbulent channel with D3Q19, SRT method, $320^3$ cells per GPU and a frame width for communication hiding of 1 in every dimension. A discussion about the scaling efficiency of the benchmarks is following in chapter 4

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

*Chapter*

# 4 Communication hiding for sparse data kernels

## 4.1 Idea

In the following, we present how to implement communication hiding for a sparse data structure. Communication hiding is used to hide the exchange of data of MPI processes on CPUs or GPUs. For this, the PDF field on every block has to be divided into the ”block interior” and a frame, as it is shown in Figure 4.1. The code for hiding communication is shown in Algorithm 1, where at first, the start of the communication is provoked. Every block packs its PDFs close to the MPI interfaces in an MPI buffer and performs a non-blocking MPI Send to its neighbors. Next, as the information in the ghost layers is not needed for the cells in the block interior, the update in these cells can now be calculated. So the LBM kernel, as well as boundary kernels, can be run on the cells in the block interior. After this step, we have to wait for the communication to finish and write the PDFs of the MPI buffer to the ghost layer. Lastly, with the updated information on the ghost layer, we can now run LBM and boundary kernels on the cells of the frame.

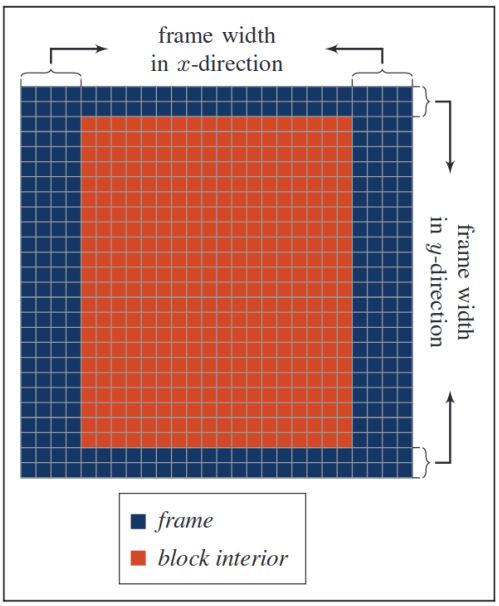


Figure 4.1: Subdivision of the PDF field in a frame and the block interior to enable communication hiding

With this algorithm, the communication of the simulation can be overlapped with the kernel runs on the interior, which leads, in the optimal case, to higher performance because of better scalability on an increasing number of MPI processes. The width of the frame in all three dimensions has to be chosen wisely to achieve maximum performance. Of course, a smaller frame width would increase the number of cells of the interior and, therefore, could provide more time to overlap the communication. On the other hand, a small frame width results in very small kernel calls and prevents consecutive memory access, which could reduce the performance of the simulation.

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

**Algorithm 1** Communication Hiding

```
1:  for each time step do
2:      Start communication
3:
4:      // run kernels on block interior
5:
6:      Run boundary kernels on block interior
7:      Run LBM kernel on block interior
8:
9:      Wait communication
10:
11:     // run kernels on frame
12:
13:     Run boundary kernels on frame
14:     Run LBM kernel on frame
```

## 4.2 Communication hiding for sparse data structure

The implementation of communication hiding for sparse data LBM is not straightforward because we miss neighboring information of the cells. This means that a cell has no information about whether it is inside the block interior or on the frame. Therefore, we have to store two additional index lists, one for the interior and one for the frame PDFs. These index lists also get filled by the flag field at the start of the simulation, where we still have the information of the frame width. Afterward, this information is not needed anymore. In addition to this, two values are stored with the number of cells in the block interior and them on the frame.

In the LBM kernel call, we distinguish between the run on the whole block, the run over the interior, and the run over the frame. In the run on the whole block, the kernel iterates over all cells of the block, and it can just use the iterator `iter` of the loop to access the proper PDFs and pull indices of the PDFs. This is not the case for the kernel calls for the interior and frame cells. They use the same LBM kernel as the full block run, but it has to be called with different parameters. For the interior run, the kernel is called with the number of interior cells and also with the interior index list, but with the same PDF list. In the kernel, we iterate only over the number of interior cells, which leads to a problem. So, for example, for the Pull-pattern, PDFs are read from neighboring cells with the help of the index list by `pdfList[indexList[iter]]` and stored in PDFs of the current cell accessed by the loop iterator (`pdfList[iter]`). But as we only loop over interior cells now, access to the PDF list by the loop iterator is not possible anymore. The solution is to store the pull index of the center direction in the interior index list, which was not needed before, as the pull index of the center PDF is the center PDF, which is not worth storing in the first place. But now, the write access of the PDF field can be handled by accessing the PDF list with the center index of the interior index list (`pdfList[indexList[iter]]`). This is possible because the interior index list is built so that the loop iterator over all inner cells can be used to access the right pull indices for the interior cells. The read access is still the same because it uses the interior index list anyways. Of course, the resulting access index for the PDF list has to be modified to store PDFs in other directions of the stencil than the center by adding the offset of the specific direction to the access index.

To run the LBM kernel on the frame, again, only the kernel call has to be modified. This time, the loop size is the number of frame cells, and the input index list is the frame index list. With this solution, the actual kernel stays the same for full, interior of frame runs, only the parameters of the loop size and the index list vary with which the kernel is called.

## 4.3 Results

The benchmarking results for communication hiding runs on the GPU cluster Juwels-Booster are shown in Figure 4.2. The benchmark scenario is a turbulent channel, with velocity bounce-back on the west boundary, pressure outflow on the east boundary, and no-slip boundaries in all other directions. Also, the scenario utilizes a D3Q19 stencil and the SRT collision model, and we run our sparse kernels on $320^3 = 32,768,000$ cells per GPU. We plotted the MLUPS (mega lattice updates per second) per GPU for an increasing number of GPUs, up to 1024 NVIDIA A100 GPUs, which results in $33.6 \cdot 10^9$ cells. The upper bound of the performance, the

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

roofline, is shown as black dashed line, which is measured by a stream-only benchmark. We compare three different configurations for the run. One run without communication hiding (blue), one run with the smallest possible frame width of 1 in all dimensions (orange), and one run with a frame width of 32 in x direction and 1 in y and z direction (green). This frame width is used to find a good trade-off between the greatest possible size of the interior kernel and still good performance for the kernel on the frame cells. Increasing the frame width in x direction should increase the performance of the outer kernel by allowing consecutive memory access for at least 32 cells and avoiding too small kernel starts.

In Figure 4.2 we can see that the run without communication hiding reaches $\sim 90\%$ of the maximum MLUPS per core for up to 4 GPUs, but for a number of GPUs beyond that, the performance drops down to 2082 MLUPS per core for 1024 GPUs, which results in a scaling efficiency of 63.48%. The configuration with a frame width of 1 in every dimension starts with a lower single GPU performance, but then scales nearly perfectly for up to 32 GPUs. For more than 32 GPUs the scaling is not perfect anymore, but we still reach 2556 MLUPs per GPU, which results in a good scaling efficiency of 84.73%. The configuration with a greater frame width of 32 cells in x direction behaves quite similarly to the smaller frame width for up to 64 GPUs. Afterward, it performs a bit worse than the configuration with the smaller frame width, with a scaling efficiency of 83.56%.

The scaling behavior of the sparse kernels with the same three configurations can also be seen in Figure 4.3, where the total MLUPS of all GPUs over an increasing number of GPUs are plotted. The perfect scaling is shown as black dashed line, which is calculated with the single GPU performance multiplied with the number of GPUs. Here, we demonstrate that the overall performance is close to the perfect scalability for all 3 runs.

Nevertheless, waLBerla is known to achieve very high scaling efficiency as demonstrated in [8] and [5], therefore, it seems that the perfect configuration in terms of cells per block, block size for the GPU call or frame width for the communication hiding still has to be found for the sparse data kernels, to achieve higher scaling efficiency than the 84.73% for the run with the small frame width. In [8] it is shown, that the right choice for the cells per block is a crucial decision to achieve good scalability with waLBerla. Also, the topology of the GPUs on the cluster could play a role in the scaling results. Therefore, further investigation is planned to find this perfect configuration for these sparse kernels on GPUs. The best idea is usually to increase the number of cells per block, but as we go higher than the $320^3$ cells per GPU shown in Figure 4.2 and Figure 4.3, the preprocessing time heavily increases because of the creation of the list structures needed for the sparse kernels.

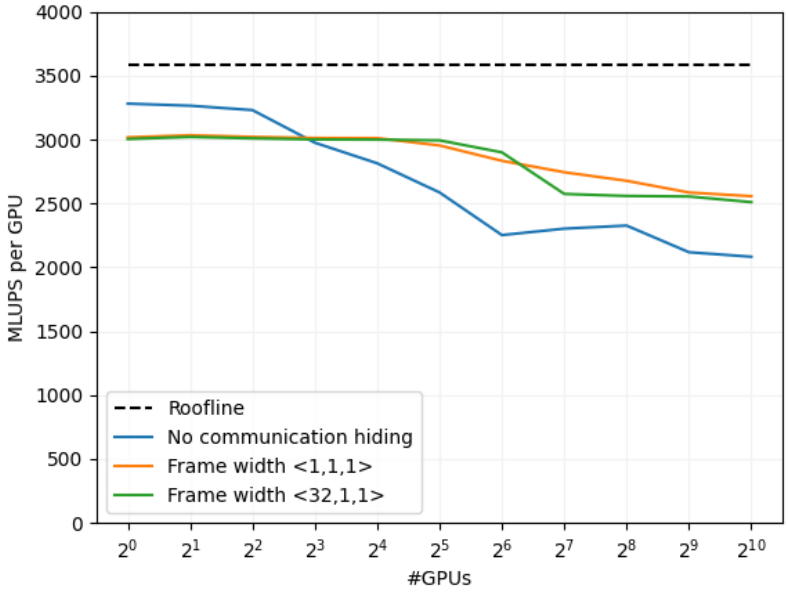


Figure 4.2: Scaling benchmark on GPU cluster Juwels-Booster with different configurations for the communication hiding. The roofline is obtained by a stream only benchmark. The figure indicates, that the perfect configuration for the scaling run still has to be found, as scaling efficiency is not optimal yet. The runs are executed with $320^3$ cells per GPU, a D3Q19 stencil and SRT collision model.

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

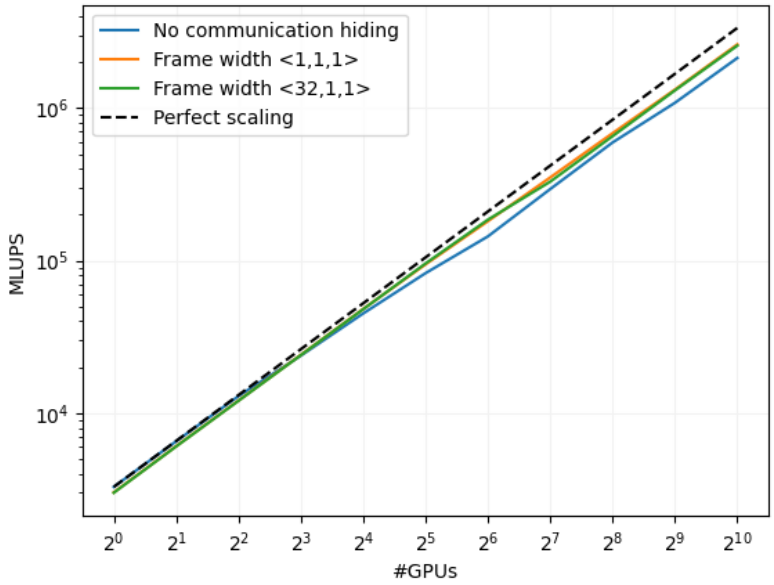


Figure 4.3: Scaling benchmark on GPU cluster Juwels-Booster with different configurations for the communication hiding. Here the total MPLUS are plotted. It shows, that all configurations show good, but not perfect scalability, because the perfect benchmark configuration for sparse kernels still has to be found. The runs are executed with $320^3$ cells per GPU, a D3Q19 stencil and SRT collision model.

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

*Chapter*

# 5 Conclusion

We demonstrated, how to implement an in-place streaming pattern, the AA-pattern, for architecture-specific sparse data kernels. We also showed, that we can achieve an average performance increase of about 33.77% MLUPS per core on CPUs, which is close to the theoretical increase of $\sim 37\%$. On GPUs on the other hand, the AA-pattern did not provide a performance increase for high numbers of GPUs, which could be caused by the communication, which becomes the bottleneck for runs with multiple GPUs, and therefore savings in form of memory accesses play a lower rule on the overall performance. Nevertheless, memory can be saved with he AA-pattern, as no temporal PDF field has to be stored on the GPU memory.

Furthermore, we implemented communication hiding for sparse data kernels, to increase the scaling efficiency of the sparse data kernels on GPUs. We showed, that most of the communication could be hidden. Thus, we were able to increase the scaling efficiency for 1024 NVIDIA A100 GPUs from 63.48% for runs without communication hiding to 84.73% for the configuration with the smallest possible frame size of <1,1,1>. Nevertheless, we aim for perfect scalability, and therefore, a better configuration for the sparse data kernels in terms of cells-per-GPU, frame width or GPU-kernel-call parameters still has to be found. Overall, however, we note that on 1024 GPUs we reach an aggregate performance of more than $2 \times 10^{12}$ LUPS, i.e., 2 TLUPs.

Al currently tested scenarios have a high portion of fluid cells in the domain. Even if the sparse kernels perform well for the tested scenarios, further work is planned to find a more suitable test case in order to demonstrate the particular advantages of the sparse kernels as compared to direct addressing kernels. A suitable test case could for example be taken from porous media flow.

Finally, we remark on the additional benefits that sparse data kernels may have. One option could be to combine sparse and dense kernels. Both types of kernels could be generated for CPUs and GPUs, and, depending on the type of the cells of the block, one could call the sparse or the dense kernel. So for a block with a high number of boundary cells, the sparse kernel would be the best choice, and for a block with mostly fluid cells, the dense kernels could be used. By this, the particular strength of each data structure would be used in a combined form to achieve the best possible performance on CPUs and GPUs.

*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.*

# Bibliography


[1] Damian Alvarez. "JUWELS Cluster and Booster: Exascale Pathfinder with Modular Supercomputing Architecture at Juelich Supercomputing Centre." In: *Journal of large-scale research facilities JLSRF* 7 (Oct. 2021). DOI: 10.17815/jlsrf-7-183.

[2] Peter Bailey et al. "Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors." In: *2009 International Conference on Parallel Processing*. Sept. 2009, pp. 550–557. DOI: 10.1109/ICPP.2009.38.

[3] 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.

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

[5] Martin Bauer et al. "waLBerla: A block-structured high-performance framework for multiphysics simulations." In: 2020. DOI: 10.1016/j.camwa.2020.01.007.

[6] Markus Holzer et al. *Deliverable 5.1: Code generation for sparse data storage LBM kernels*. https://scalable-hpc.eu/wp-content/uploads/2022/08/SCALABLE_D5.1_Basic-generated-sparse-data-storage-LBM-kernels.pdf. 2021 (accessed December 7, 2014).

[7] Moritz Lehmann et al. "Accuracy and performance of the lattice Boltzmann method with 64-bit, 32-bit, and customized 16-bit number formats." In: *Phys. Rev. E* 106.1 (July 2022). DOI: 10.1103/physreve.106.015308.

[8] Lubomir Riha and Gabriel Staffelbach. *SCALABLE Deliverable 2.3: Application performance, accuracy and energy efficiency*. 2021.

[9] Philipp Suffa et al. *Deliverable 5.2:Generated kernels for boundary handling and communication routines*. https://scalable-hpc.eu/wp-content/uploads/2023/01/SCALABLE_D5.2_Generated-kernels-for-boundary-handling-and-communication-routines1.pdf. 2023.



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