Compas'2022 : Parallélisme / Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

Parallelization of the Lattice-Boltzmann schemes using the task-based method

Clément Flint$^{\alpha}$ [*], Berenger Bramas$^{\alpha}$, Stephane Genaud$^{\alpha}$, Philippe Helluy$^{\beta}$

$^{\alpha}$ CAMUS Inria Nancy – Grand Est, Strasbourg University, ICPS ICube, Illkirch, France
$^{\beta}$ TONUS Inria Nancy – Grand Est, IRMA, Strasbourg, France

Abstract
The popularization of graphic processing units (GPUs) has led to their extensive use in high-performance numerical simulations. The Lattice Boltzmann Methodology (LBM) is a general framework for constructing efficient numerical fluid simulations. In this scheme, the fluid quantities are approximated on a structured grid. At each time step, a shift-relaxation process is applied, where each kinetic value is shifted to the corresponding direction in the lattice. Thanks to its simplicity, the LBM is subject to many software optimizations. State-of-the-art techniques aim at adapting the LBM scheme to improve the computational throughput on modern processors. Currently, most effort is put into optimizing this process on GPUs, as their architecture is highly suited for this type of computation. A bottleneck of GPU implementations is that the data size of the simulation is limited by the GPU memory. This restricts the number of volume elements and, therefore, the degree of precision one can obtain. In this work, we divide the lattice structure into multiple subsets that can be executed individually. This allows the work to be distributed among different processing units at the cost of increased complexity and memory transfers. But the constraint on GPU memory is relaxed, as the subsets can be made as small as needed. Additionally, we use the task-based approach for parallelizing the application, which allows the computation to be efficiently distributed among multiple processing units.

1. Introduction

The classical Lattice-Boltzmann Method (LBM) is a computational fluid dynamics (CFD) approach for simulating the physics of fluid flows that rely on a scalar kinetic function. The literature on this subject is consequent and we hence only refer to the work of Succi [17]. The classical method is not the most general. For instance, it does not handle compressible flow properly. In the present work, we thus use a more general version of the LBM, based on a vectorial kinetic representation proposed by Bouchut [6] and Aregba-Natalini et al. [1]. In this approach, the simulation consists of a transport (streaming) step and a relaxation (collision) step. The transport step represents the movement of fictive particles, while the relaxation step handles the collisions between them. We also impose that the grid is regular. This allows solving the transport step of the Lattice-Boltzmann Method conveniently, as the kinetic vectors propagate directly to their neighbors. It is also well-suited for massively parallel architectures, as the regular arrangement of the points allows for fully coalesced GPU memory accesses throughout the execution. The relaxation and the transport steps are both fully parallelizable and can be grouped together.
However, this efficient representation faces challenges when implemented on modern hardware. Indeed, GPUs, whose massively parallel architecture is well suited for these types of

[*]. Corresponding author, clement.flint@inria.fr.

problems, bring two substantial problems. The first is that their memory is limited in size as compared to typical main memories. The second is that their data transfer rates are typically low and can, therefore, significantly impact the execution time. The GPU memory limit becomes problematic in cases where the lattice structure does not fit in it. In this case, the GPU can not perform a step on the whole lattice and the step must be split. On the other hand, data transfer rates become more and more problematic, as the number of different processors grows, especially in the case of multi-GPU simulations.
In this paper, we provide a task-based implementation that aims at solving all these issues. This solution relies on the StarPU framework, whose ability to handle large-scale and heterogeneous hardware makes it well-suited for using modern simulation platforms at their full potential. We evaluate our implementation by comparing it against an already-existing single GPU application known as *Patapon*.
In section 2, we present an overview of previous studies that relate to this work. In section 3, we define the framework of our method and provide relevant technical details. In section 4, we show various measurements that demonstrate the efficiency of our approach, in particular its resilience to a change in the hardware configuration. We use the classical Orszag-Tang test since it offers multiple ways of verifying the accuracy of the simulation. Finally, in section 5 we summarize the essence of this work and present our perspectives regarding future works.

2. Related work

Lattice-Boltzmann methods originate from the idea of using Boltzmann equations for simulating lattice-gas automata [11]. In the LBM, we use infinitesimal volume elements for approximating solutions to equations of fluid dynamics [4, 18]. François Bouchut presents a framework for building kinetic BGK schemes given a set of conservation laws [5]. In terms of computing, these schemes work with 2 additional dimensions (in addition to the 2 or 3 spatial dimensions) : the conservative variable (that represents the different values a cell stores) and the kinetic value (that represents the flow direction). In this work, we only consider this vectorial kinetic representation which makes our approach differ from the standard LBM schemes.
This framework leads to schemes that can be seen as a specific kind of stencil algorithm because the new value of a cell will be computed depending on the values of a set of neighbors. In particular, for a specific neighbor, the accessed direction will always be the same, which allows for fully coalesced GPU memory accesses. Numerous studies focus on improving kernels for the GPUs [15, 14, 7]. Other studies provide distributed implementations across different nodes [19, 13, 9]. It is generally accepted that the limiting factor of these types of simulations is the memory bandwidth. Memory transfers typically take 80% of the simulation time on the GPU. This is why much effort is put into optimizing these transfers, especially as the simulations scale up in size. Different techniques such as temporal blocking and region sharing have been introduced throughout the literature and aim at improving the execution flow in these kinds of memory-bound applications [16, 8, 10, 12]. In this work, we aim to investigate the ability of the task-based parallelization method to efficiently run large-scale LBM simulations. In particular, we focus on adapting the synchronization methodology to the task-based paradigm.

3. Contribution

3.1. Formalization

In this section, we present the formalization we use for the Lattice-Boltzmann schemes. We use the relevant terminology for a 3D space, but the terms can be replaced by their corresponding equivalents for other dimensions.

The data consist of an N-dimensional orthotope (i.e. hyperrectangle), where one dimension represents the conservative variable, another represents the direction, and the others represent the position in the space. The spatial dimensions are discretized, such that the volume elements are identical. In the 3D case, we have the inequality :

$$\Delta x = \Delta y = \Delta z$$

Where Δx (respectively Δy, Δz) is the size of the volume element on the x-axis (respectively y, z). Also, as the volume elements are distributed in a grid-like structure, we will refer to them as *cells*. The simulation process consists of a succession of time steps. To ensure the stability of the scheme, we require that $\Delta t = \frac{\Delta x}{\lambda}$, where Δt is the time of the step and λ is the maximum wave velocity of the fluid in this problem.
The principle of the method is to compute fluxes for all cells, depending on the value of their respective vectors of conservative variables W. These fluxes are then transported to the d neighbors of each cell, depending on the vector of directions D. D represents the possible directions where the flux can flow at the scale of the cell. The size d of D is not constrained, but the scheme typically requires that there are at least two directions per axis. This phase of transport, known as the *shift* phase, results in the W vectors being split and shifted. A second phase, known as the *relax* phase, combines the resulting fluxes into a new W vector and re-splits them for the next *shift* phase. The relax phase is designed in such a way that it ensures second-order time accuracy of the whole scheme.
The implemented scheme must specify the behavior of these two phases. This lets a high degree of freedom and numerous schemes fit our formalization. The most noticeable downside of this formalization is that the volume elements must have a particular regular structure. This would not allow the use of adaptive mesh refinement (AMR) optimizations. We, however, postulate that efficiently using the parallelization potential of the GPU renders AMR obsolete in numerous cases.

3.2. Implementation

We create a C++ engine that implements our method. The use of template metaprogramming from the C++ language allows the user to define parameters for the simulation.

Structure of an execution

To allow the algorithm to be distributed on multiple nodes, the grid is divided into multiple subgrids. Each subgrid contains extra data at its border that duplicate the values of its neighbors. To ensure the coherency of these data, a synchronization phase is needed. This phase is referred to as the *exchange* phase and is schematized in Figure 1. This area is often referred to as the *halo* of a subgrid. This halo can contain more than the immediate neighbors. We, therefore, introduce a customizable *depth* parameter that controls how many additional values will be added on each side. This *depth* parameter directly controls (with the direction vectors) the number of consecutive *step* phases that can be performed after one synchronization. Figure 4 gives a visual example that shows how different choices of directions and *depth* parameters impact successive steps. Figure 3 schematizes the general execution flow of our method.

StarPU

Our engine uses StarPU [2], a task-based multicore runtime system, for handling the task-based parallelization. It supports multiple technologies such as OpenCL, CUDA, and MPI.

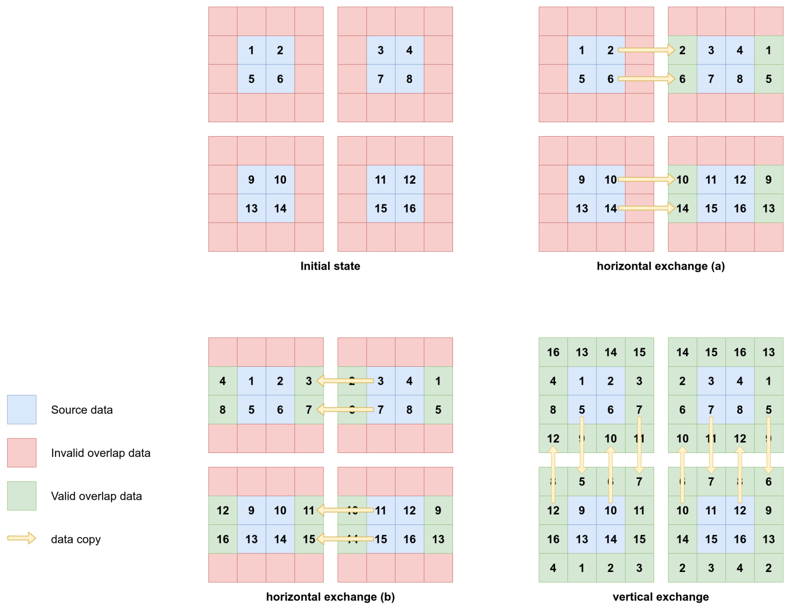

FIGURE 1 – This schema shows how the faces are exchanged between the sub-lattices in the 2D case, with an overlapping depth of 1. The horizontal copy phase is divided into two parts to better display the process. For the sake of clarity, not all necessary data copies (yellow arrows) are shown. In a real case scenario, the data consist of vectors of floating-point values, rather than integers.

With StarPU, an application is described by a set of tasks and their respective data accesses. In principle, the end-user does not have to handle the scheduling or the memory transfers. Hence, when the dependency structure of a process is complex, StarPU typically produces efficient executions. In the framework of this work, the use of StarPU is justified by different needs. The use of the task-based parallelization method allows the engine to have a clear view of the scheduled work and the available resources at runtime. It aims at automatically making the best use of the available resources and in particular overlapping data transfer technologies. Algorithm 1 describes how the tasks are submitted to StarPU in our method.

4. Experimental results

Configuration

— Hardware : the experiments were performed on a node with two Intel(R) Xeon(R) CPU E5-2683 v4 at 2.10GHz (32 cores in total). The node also has two NVidia P100 GPUs, each with 16GB of memory.
— Software : we use the 11 February 2022 commit of the master branch of StarPU, GCC 9.3, and CUDA 11.2. Our application has been compiled with the following flags *-03* with GCC and *-O3 -arch=sm_60* with nvcc. We use the StarPU scheduler DMDA (HEFT).

Test case

We run the Orszag-Tang vortex, a common test for Magnetohydrodynamics. We use the same parameters for the simulation as Baty et al. [3]. In particular, we ensure that we observe the same second-order convergence as them, which suggests that we successfully replicated their scheme.

We compare our TB-LBM (Task-Based Lattice-Boltzmann) that implements the Orszag-Tang vortex with the StarPU task-based approach and the system of subgrids against *Patapon*, a state-of-the-art python framework that uses OpenCL for performing LBM computations on one GPU. *Patapon* can only use 1 GPU, whereas TB-LBM can distribute the work on multiple processors. We, therefore, present its results for 3 configurations : 1 GPU, 2 GPUs, and 2 GPUs + 1 CPU. For the 1-GPU configuration, we set a single subgrid of the same size as the global grid and an overlapping depth of 1. For the other 2-GPU configurations, we divide the space into $4 \times 4 = 16$ subgrids and use an overlap of 8.

We use different grid dimensions : *small* (1024), *medium* (2048), *large* (4096) and *huge* (8192). The data of the *huge* test case does not fit into the memory of one of the tested GPU and cannot be simulated by *Patapon*. TB-LBM with 1 GPU could execute this test case if the grid was split into smaller subgrids, but we impose that there is only one subgrid in this benchmark. We adapt the number of performed time steps depending on the grid size. The *huge* case performs 64 time steps, the *large* one performs 256, the *medium* 1024, and the *small* 4096 (we multiply by 4 between each case). This allows us to expect the number of computations to be the same between the different grid sizes.

Analysis

We provide the results in Figure 2. We observe that the performance between the different configurations varies greatly depending on the grid size. For the *medium* and the *large* grid sizes, going from 1 GPU to 2 GPUs speedups the execution by about 2. It is an argument in favor of our strategy because it means that we do not lose excessive performance with our decomposition. For a *small* grid size, however, the TB-LBM with 1 GPU is the fastest configuration of the four. This is a surprising result that is likely due to the small size of the data that makes the face exchanges very efficient.

Patapon is relatively consistent in terms of execution time throughout the possible grid sizes. TB-LBM is more fluctuant, which can be explained by the fact that we use a system of subgrids and slice exchanges that induces memory transfers and scheduling choices that we do not control, whereas *Patapon* has a linear transferless execution. The difference between *Patapon* and TB-LBM also depends on the grid size :

— For the *small* size, TB-LBM is always faster than *Patapon*.
— For the *medium* size, *Patapon* is always faster.
— For the *large* size, *Patapon* is faster than the 1-GPU configuration but slower than the two 2-GPU configurations.

In theory, we would expect that the *step* kernel of *Patapon* is slower or comparable to that of 1-GPU TB-LBM because OpenCL kernels are generally slower than CUDA ones. In this experiment, the *medium* and *large* test cases do not behave as expected. There are 3 main differences between *Patapon* and TB-LBM that can explain this. The first one is the *step* kernel that is slightly different. The second difference is the use of StarPU itself which can lead to inefficient scheduling choices. In our case, we observed that only the DMDA scheduler provides satisfactory results. Finally, the last and more likely difference that could explain that TB-LBM is sometimes slower is the regular use of a synchronization mechanism to keep the halo of each subgrid coherent. In TB-LBM 1-GPU, there is only one subgrid but the halo of this subgrid still needs to be synchronized between the opposite sides which induces a substantial amount of additional read/writes. In *Patapon*, there is no such synchronization as the *step* kernel accesses the neighbors with a modulo operation. For the 2-GPU and the 2-GPU + CPU configurations,

the difference with *Patapon* is less noticeable. On the other hand, adding the CPU processor does not appear to help the execution.
Finally, the result of interest is the *huge* test case. Since the data do not fit into a single GPU, we measure the ability of TB-LBM to distribute the algorithm. The test cases are designed in such a way that the amount of operation stays the same. We can, therefore, extrapolate that the computational throughput of *Patapon* would theoretically allow an execution time of $\approx$ 2.4s in the *Huge* test case which corresponds to a speedup of approximately 31%. This demonstrates the relevance of using StarPU and, by extension, the task-based method for performing LBM simulations on a large scale.

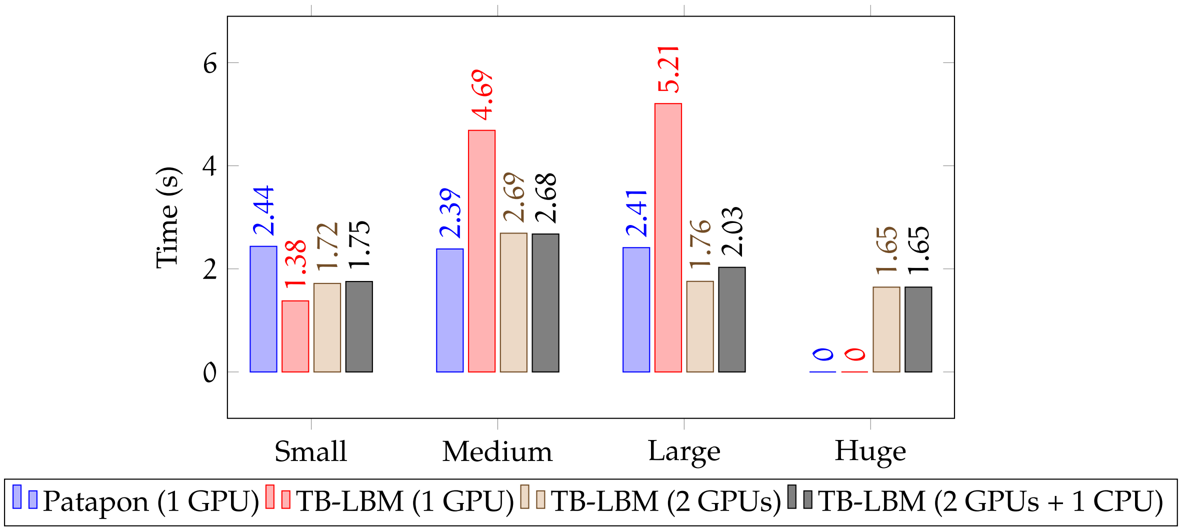

FIGURE 2 – Execution times for the different grid sizes (*small*, *medium*, *large* and *huge*) and three hardware setup alternatives. *Patapon* is a state-of-the-art software that is only able to run on 1 GPU. TB-LBM implements the subgrid mechanism and is able to run on multicore hybrid configurations.

5. Conclusion

In this paper, we have provided the first results of our task-based LBM solver, which is competitive compared to a state-of-the-art implementation and that can be executed on heterogeneous architectures. This allows us to solve problems that cannot fit into a single GPU memory. However, several important issues should be addressed before obtaining an efficient application.
Therefore, in future work, we will (1) evaluate the raw performance of our computational kernels both on CPUs and GPUs, and find out the best CUDA grid configuration for a given size of the sub-grid. Then (2) we plan to parallelize the tasks internally on the CPUs. Indeed, to avoid a huge performance difference between CPUs and GPUs, we would like to test if manually implementing the OpenMP parallelization in a single CPU task improves the CPU kernels. Finally, (3) we will ensure that the tasks are scheduled such that the memory transfers are minimized. For this, we will try to guide the scheduler so that the subgrids tend to be grouped in the GPUs based on their proximity in the simulation space. This will be the key point to expect obtaining performance with more than 2 computing nodes.

Appendix

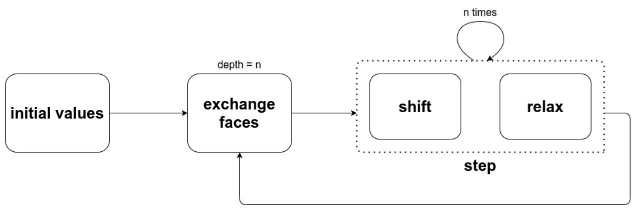

FIGURE 3 – The execution flow of the solver consists of a succession of *exchange* and *step* phases.

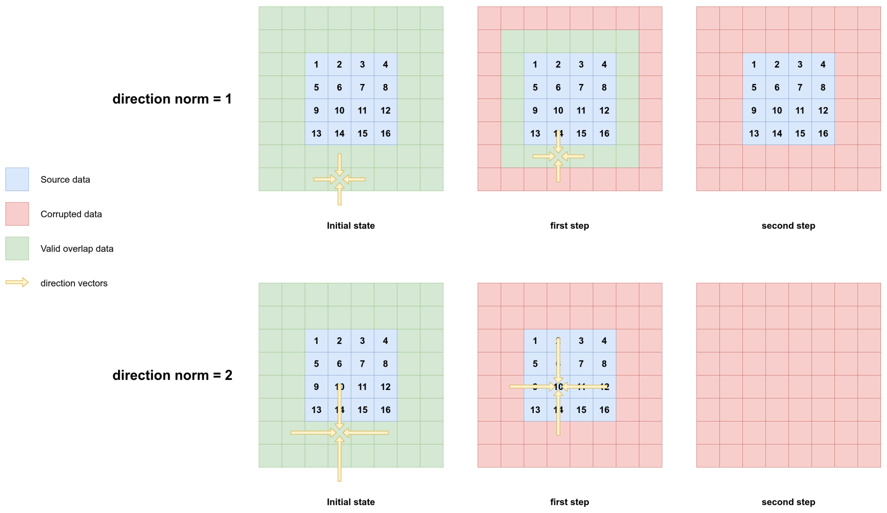

FIGURE 4 – This example shows how bordering data are corrupted, depending on the direction vectors. The sub-lattice has an overlapping depth of 2 and a data size of 4×4. If the data are fetched (yellow arrows) from outside the subgrid or from a corrupted cell, the target cell becomes corrupted. When the norm of the direction vectors is 1, 2 steps can be performed without corrupting the main data, while a norm of 2 only allows for 1 step.

References

1. Aregba-Driollet (D.) et Natalini (R.). – Discrete kinetic schemes for multidimensional systems of conservation laws. *SIAM Journal on Numerical Analysis*, vol. 37, n6, 2000, pp. 1973–2004.
2. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE - Concurrency and*

Algorithm 1: Insertion of the tasks for a 2D grid G.

```
1  for t from 0 to NB_STEPS do
       // Insert the operation step on each sub-grid
2      for i from 0 to G.subGridsNumber-1 do
3          insert_tasks(step, READ_WRITE, G.sub_grid[i]);
       // Insert copy of the interface
4      for i from 0 to G.subGridsNumber-1 do
           // List of directions; first the horizontal ones, then the
              vertical ones
5          directions = [(1,0), (-1,0), (0,1), (0,-1)];
6          for d in directions do
               // Get the neighboring cell in direction d
7              neighbor = get_neighbor(G.sub_grid[i], d);
               // Get the id of the buffer that we are going to use
8              bufferIndex = (i*numberOfDirections+d)%numberOfBuffers;
               // Copy the bordering cells of the neighbor to a temporary
                  buffer (slice)
9              insert_tasks(subGrid_to_slice, READ, neighbor, WRITE, buffer[bufferIndex]);
               // Write the buffered values to the target subgrid
10             insert_tasks(slice_to_subGrid, READ, buffer[bufferIndex], WRITE,
                 G.sub_grid[i]);
```

Computation : Practice and Experience, Special Issue : Euro-Par 2009, vol. 23, février 2011, pp. 187–198.

3. Baty (H.), Drui (F.), Franck (E.), Helluy (P.), Klingenberg (C.) et Thanhäuser (L.). – A robust and efficient solver based on kinetic schemes for Magnetohydrodynamics (MHD) equations. – avril 2021. working paper or preprint.
4. Boon (J.). – The lattice boltzmann equation for fluid dynamics and beyond. *European Journal of Mechanics - B/Fluids*, vol. 22, 01 2003, p. 101.
5. Bouchut (F.). – Construction of bgk models with a family of kinetic entropies for a given system of conservation laws. *Journal of Statistical Physics*, vol. 95, 04 1999, pp. 113–170.
6. Bouchut (F.), Jobic (Y.), Natalini (R.), Occelli (R.) et Pavan (V.). – Second-order entropy satisfying BGK-FVS schemes for incompressible Navier-Stokes equations. *SMAI Journal of Computational Mathematics*, vol. 4, mars 2018, pp. 1–56.
7. Datta (K.), Williams (S.), Volkov (V.), Carter (J.), Oliker (L.), Shalf (J.) et Yelick (K.). – Auto-tuning the 27-point stencil for multicore. – In *In Proc. iWAPT2009 : The Fourth International Workshop on Automatic Performance Tuning*volume 70, 2009.
8. Ino (F.), Miki (N.) et Hagihara (K.). – Pacc : a directive-based programming framework for out-of-core stencil computation on accelerators. *International Journal of High Performance Computing and Networking*, vol. 13, 01 2019, p. 19.
9. Jacquelin (M.), Araya-Polo (M.) et Meng (J.). – Massively scalable stencil algorithm. *arXiv preprint arXiv :2204.03775*, 2022.
10. Jin (G.), Lin (J.) et Endo (T.). – Efficient utilization of memory hierarchy to enable the computation on bigger domains for stencil computation in cpu-gpu based systems. – pp. 1–6,

12 2014.

11. McNamara (G.) et Zanetti (G.). – Use of the boltzmann equation to simulate lattice-gas automata. *Physical review letters*, vol. 61, 12 1988, pp. 2332–2335.
12. Muranushi (T.) et Makino (J.). – Optimal temporal blocking for stencil computation. *Procedia Computer Science*, vol. 51, 2015, pp. 1303–1312. – International Conference On Computational Science, ICCS 2015.
13. Pekkilä (J.), Väisälä (M. S.), Käpylä (M. J.), Rheinhardt (M.) et Lappi (O.). – Scalable communication for high-order stencil computations using cuda-aware mpi. *Parallel Computing*, vol. 111, 2022, p. 102904.
14. Rawat (P. S.), Rastello (F.), Sukumaran-Rajam (A.), Pouchet (L.-N.), Rountev (A.) et Sadayappan (P.). – Register optimizations for stencils on gpus. – In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 168–182, 2018.
15. Shen (J.), Deng (X.), Wu (Y.), Okita (M.) et Ino (F.). – Compression-based optimizations for out-of-core gpu stencil computation. *arXiv preprint arXiv :2204.11315*, 2022.
16. Shen (J.), Ino (F.), Farrés (A.) et Hanzich (M.). – A data-centric directive-based framework to accelerate out-of-core stencil computation on a gpu. *IEICE Transactions on Information and Systems*, vol. E103.D, 12 2020, pp. 2421–2434.
17. Succi (S.). – *The lattice Boltzmann equation : for fluid dynamics and beyond*. – Oxford university press, 2001.
18. Wolf-Gladrow (D.). – Lattice-gas cellular automata and lattice boltzmann models - an introduction. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*, vol. 1725, 01 2000.
19. Zabelok (S.), Arslanbekov (R.) et Kolobov (V.). – – Multi-gpu kinetic solvers using mpi and cudavolume 1628, 07 2014.