Automatic Generation of Adjoint Operators for the Lattice Boltzmann Method

Stephan Seitz, Andreas Maier
Pattern Recognition Lab
Friedrich-Alexander-University Erlangen-Nürnberg
Erlangen, Germany
email: {stephan.seitz, andreas.maier}@fau.de

Martin Bauer, Negar Mirshahzadeh, Harald Köstler
Chair for System Simulation
Friedrich-Alexander-University Erlangen-Nürnberg
Erlangen, Germany
email: {martin.bauer, negar.mirshahzadeh, harald.koestler}@fau.de

***Abstract*—Gradient-based optimization techniques for Computational Fluid Dynamics have been an emerging field of research in the past years. With important applications in industrial product design, science and medicine, there has been an increasing interest to use the growing computational resources in order to improve realism of simulations by maximizing their coherence with measurement data or to refine simulation setups to fulfill imposed design goals. However, the derivation of the gradients with respect to certain simulation parameters can be complex and requires manual changes to the used algorithms. In the case of the popular Lattice Boltzmann Method, various models exists that regard the effects of different physical quantities and control parameters. In this paper, we propose a generalized framework that enables the automatic generation of efficient code for the optimization on general purpose computers and graphics processing units using symbolic descriptions of arbitrary Lattice Boltzmann Methods. The required derivation of corresponding adjoint models and necessary boundary conditions are handled transparently for the user. We greatly simplify the process of fluid-simulation-based optimization for a broader audience by providing Lattice Boltzmann simulations as automatic differentiable building blocks for the widely used machine learning frameworks Tensorflow and Torch.**

Keywords–Lattice Boltzmann method; Computational Fluid Dynamics; Adjoint Methods; Gradient-based Optimization; Tensorflow.

I. INTRODUCTION

Applications for optimization using Computational Fluid Dynamics (CFD) range from numerical weather prediction [1], medicine [2], computer graphics [3], scientific exploration in physics to mechanical [4] and chemical engineering [5]. The Lattice Boltzmann Method (LBM) is a promising alternative to established finite element or finite volume flow solvers due to its suitability for modern, parallel computing hardware and its simple treatment of complex and changing geometries [6]. These properties make it also well suited for gradient-based optimization schemes. The gradient calculation for LBMs is based on a backward-in-time sensitivity analysis called Adjoint Lattice Boltzmann Method (ALBM) [7][8]. Manually deriving the adjoint method is a tedious and time-consuming process. It has to be done for each concrete setup because the adjoint method is highly dependent on the chosen LBM, its boundary conditions, the set of free parameters, and the objective function. This effortful workflow currently impedes the usage of LBM-based optimization by a greater audience with no experience in the implementation of this CFD algorithm, despite the wide range of possible applications. Also for experts, it might be tedious to efficiently implement ALBM for a specific Lattice Boltzmann (LB) model and instance of a problem, especially if MPI-parallel (message passing interface) execution or Graphics Processing Unit (GPU) acceleration is desired.

A great simplification for the efficient implementation of optimization algorithms by non-experts has been achieved recently in the field of machine learning. Array-based frameworks like *Tensorflow* [9] and *Torch* [10] have dramatically accelerated the pace of discovery and led to a democratization of research in this discipline. Commonly needed building blocks are exposed to scripting languages like Python or Lua while preserving a relatively high single-node performance through their implementation in C++ or CUDA. Most of the available operators provide an implementation for the calculation of their gradient which enables the automatic differentiation and optimization of user-defined objective functions. We are convinced that providing automatically generated implementations for arbitrary LBMs as building blocks for optimization frameworks could greatly simplify their usage and increase the efficiency of their implementation. In this paper, we thus present a code generation framework for a wide range of LBMs with automatic derivation of forward and adjoint methods. Our tool generates highly optimized, MPI-parallel implementations for Central Processing Units (CPUs) and GPUs, including boundary treatment. We automatically account for optimizable, constant and time-constant variables. The integration of our tool in the automatic differentiation frameworks *Tensorflow* and *Torch* allows for rapid prototyping of optimization schemes while preserving the flexibility of switching between various LB models without making trade-offs in terms of execution performance.

While many stencils code generation tools have been proposed before, ours is the first to our knowledge which facilitates automatic optimization using ALBM, given only a specification of the LB model, the geometry and boundary conditions. Another strength of our approach is that no new domain-specific language is introduced. Instead, we rely on an extension of the widespread computer algebra system *SymPy* [11] which allows code generation in the familiar programming environment of Python. The novel generation of custom operations from symbolic mathematical representation for both major machine learning frameworks might also be useful for other applications.

Our framework [12] is available as open-source software without the LBM abstraction layer [13]. The results of this paper providing automatically derived operators and integration for PyTorch and Tensorflow are about to be published in the same place.

 ISBN: 978-1-61208-732-0

The remainder of this paper is organized as follows: Section II presents related work regarding LBM, ALBM and frameworks for automatic differentiation. Section III explains the architecture of our framework, while Section IV evaluates and Section V discusses its application on an example problem.

II. Related Work

A. Lattice Boltzmann Method

The Lattice Boltzmann Method is a mesoscopic method that has been established as an alternative to classical Navier-Stokes solvers for Computational Fluid Dynamics (CFD) simulations [6]. It is a so-called kinetic method, using particle distribution functions (PDFs) $f_i(\boldsymbol{x}, t)$ to discretize phase space. The PDFs thus represent the probability density of particles at location $\boldsymbol{x}$ at time t traveling with the discrete velocity $\boldsymbol{c}_i$. Macroscopic quantities like density and velocity are obtained as moments of the distribution function. Space is usually discretized into a uniform Lattice of cells which can be addressed using Cartesian coordinates. An explicit time stepping scheme allows for extensive parallelization even on extreme scales due to its high locality. The Lattice Boltzmann equation is derived by discretizing the Boltzmann equation in physical space, velocity space and time:

$$f_i(\boldsymbol{x} + \boldsymbol{c}_i \Delta t, t + \Delta t) = f_i(\boldsymbol{x}, t) + \Omega_i(f). \tag{1}$$

Ω_i denotes the collision operator that models particle collisions by relaxing the PDFs towards an equilibrium distribution. Depending on the concrete collision operator, various different LB models exist [6]: From single-relaxation-time (SRT) models using the Bhatnagar–Gross–Krook approximation over two-relaxation-time (TRT) models proposed by Ginzburg et al. [14] up to generic multi-relaxation-time schemes [15][16]. Recent contributions aim to increase the accuracy and stability of the method by relaxing in cumulant-space instead of moment space [17] or by choosing relaxation rates subject to an entropy condition [18]. Several LBM frameworks exist that aid the user with setting up a full LB simulation, e.g., Palabos [19][20], OpenLB [21] and waLBerla [22][23]. However, none of these frameworks support both LB code generation as well as automatic differentiation for adjoint problems.

B. Adjoint Methods

Adjoint methods provide efficient means to evaluate gradients of potentially complex time-dependent problems. Given a mathematical model to simulate forward in time, the method of Lagrangian multipliers can be used to derive a complementary backward model [24]. The adjoint can either be determined from a mathematical description of the direct problem (analytical derivation) or by an algorithmic analysis of the code used for forward calculation (automatic differentiation). Examples for automatic derivation of forward and backward code from a high-level symbolic description of partial differential equations (PDEs) are the *Devito* [25][26] and the *Dolfin Adjoint* framework [27][28]. *Devito* generates code for finite-differences whereas *Dolfin* uses the finite element method. Frameworks implementing the adjoint method using automatic differentiation are usually not bound to a specific method.

For LBM, manual derivations of the adjoint dominate, either by an analytical gradient of the discrete time stepping step [7] (discretize-then-optimize approach) or by re-discretizing the gradient of a continuous formulation of the Lattice-Boltzmann equation [8] (optimize-then-discretize). Despite most authors recommend the usage of a computer algebra system for the derivation and Laniewski et al. [4] even use a source-to-source auto-differentiation tool on a forward LBM implementation to obtain backward code, a fully-automated derivation of an ALBM scheme is still missing. ALBMs have been applied to a wide range of problems, e.g., parameter identification [7], data assimilation to measurement data [8], or topology optimization [4][29]–[31]. Also, comparisons with finite-element-based optimization have been made, giving comparable results [30].

C. Automatic Differentiation

As alternative to analytical derivation, adjoint models can also be obtained using automatic differentiation (AD). Sequential chaining of elementary differentiable operations often helps to avoid typical problems of numerical and analytical differentiation arising in highly complex expressions, like numerical instabilities [24]

AD is based on the multi-dimensional chain rule where the gradient ∇f of $f = f_1 \circ f_2 \circ \cdots \circ f_N$ with respect to the inputs of f_1 can be calculated as

$$\nabla f = J_N \cdot J_{N-1} \cdot \cdots \cdot J_2 \cdot J_1 \tag{2}$$

where J_1 to J_N are the respective Jacobians of f_1 to f_N.

The evaluation of this expression can be performed in different ways [32]: **forward-mode** automatic differentiation corresponds to a direct evaluation of Equation 2. This corresponds to a direct tracing of the computation paths of the input variables, such that the computation order and memory access pattern is conserved.

In optimization applications, the last Jacobian J_N usually correspond to a scalar objective function. In this case, the **backward-mode** evaluation order becomes favorable, that is obtained by taking the transpose or adjoint of above expression.

$$\nabla f = \left(J_1^T \cdot J_2^T \cdot \cdots \cdot J_{N-1}^T \cdot J_N^T\right)^T. \tag{3}$$

Now, with J_N^T being a vector, only matrix-vector multiplications have to be performed. This approach is more memory-efficient and also facilitates the incremental calculation of partial gradients. However, this comes with a disadvantage: the paths in the calculation graph are inverted and the evaluation can only begin after the forward pass has completed. All the intermediate results of the forward pass need to be stored and all memory accesses have to be transformed by turning write operations into read operations and vice versa. Changing the memory access pattern can critically harm execution performance, since efficient "pull" kernels that read multiple neighbors and write only locally are transformed into "push" kernels that read local values and write to neighboring cells. This change also affects the parallelization strategy i.e. the halo layer exchange for MPI execution.

For this reason, Hückelheim et al. proposed a scheme called transposed forward-mode automatic differentiation (**TF-MAD**) for stencil codes that mimics memory access patterns of the forward pass [32]. In this scheme, the summations that evaluate the gradients are re-ordered inside a single time step in similar ways as in the forward differentiation. This preserves the memory access structure while keeping the desired backward-in-time evaluation order.

III. Methods

In this section we first present a code generation tool for forward LBMs that automates the derivation, performance optimization, and implementation of LBMs for CPUs and GPUs.

Then an extension of this tool for adjoint LBMs is described. Finally, we show how the integration into popular automatic differentiation frameworks enables the user to flexibly describe a wide range of optimization problems.

A. Automatic LBM Code Generation

The main contribution of this work is a code generation tool for Adjoint Lattice Boltzmann Methods that can be fully integrated into popular automatic differentiation frameworks. Our code generation approach aims to overcome the following flexibility-performance trade-off: On the one hand, the application scientist needs a flexible toolkit to set up the physical model, boundary conditions and objective function. On the other hand, a careful performance engineering process is required, to get optimal runtime performance. This process includes a reformulation of the model to save floating point operations, loop transformations for optimal cache usage and manual vectorization with single-instruction-multiple-data (SIMD) intrinsics. While previously, this had to be done manually, our tool fully automates this process.

On the highest abstraction layer, the Lattice Boltzmann collision operator is symbolically formulated using the formalism of multi relaxation time methods. A set of independent moments or cumulants has to be provided, together with their equilibrium values and relaxation times. This formalism includes the widely used single relaxation time (SRT) and two relaxation time (TRT) methods as a special case. Relaxation times can be chosen either constant or subject to an entropy condition [18]. This layer allows for flexible modification of the model, e.g., by introducing custom force terms or by locally adapting relaxation times for turbulence modeling or simulation of non-Newtonian fluids. One important aspect that increases the complexity of LB implementations significantly is the handling of boundary conditions. Our code generation framework can create special boundary kernels for all standard LB boundary conditions including no-slip, velocity and pressure boundaries.

This high-level LBM description is then automatically transformed into a stencil formulation. The stencil formulation consists of an ordered assignment list, containing accesses to abstract arrays using relative indexing. These assignments have to be independent of each other, such that they can all be executed in parallel.

Both representations are implemented using the *SymPy* computer algebra system [11]. The advantages of using a Python package for symbolic mathematics are evident: a high number of users are familiar with this framework and problems can be formulated intuitively in terms of abstract formulas. Mathematical transformations like discretization, simplifications and solving for certain variables can be concisely formulated in a computer algebra system. To reduce the number of operations in the stencil description, we first simplify the stencil description by custom transformations that make use of LBM domain knowledge, then use *SymPy*'s generic common subexpression elimination to further reduce the number of operations.

Next, the stencil description is transformed into an algorithmic description, explicitly encoding the loop structure. Loop transformations like cache-blocking, loop fusion and loop splitting are conducted at this stage. The algorithmic description is then finally passed to the C or CUDA backend. For CPUs, an OpenMP parallel implementation is generated that is also explicitly vectorized using SIMD intrinsics. We support the

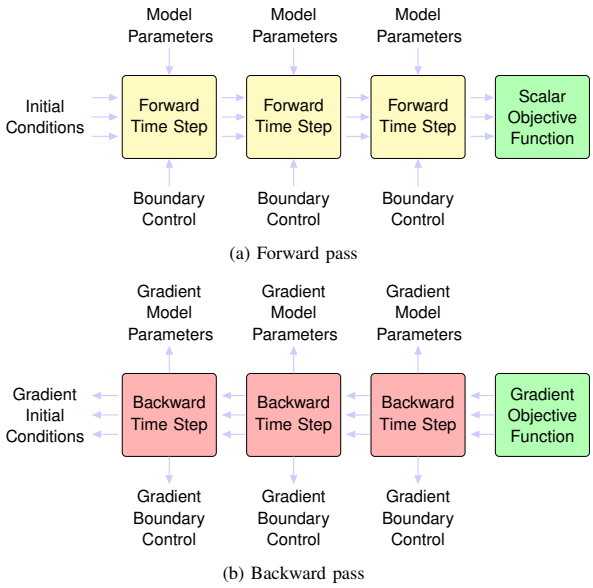


Figure 1. Auto-differentiation-based optimization with automatically generated kernels for forward and backward pass.

Intel SSE, AVX and AVX512 instruction sets. Vectorization can be done without any additional analysis steps since our pipeline guarantees that loop iterations are independent. Conditionals are mapped to efficient blend instructions. To run large scale, distributed-memory simulations, the generated compute kernels can be integrated in the *waLBerla* [22][23] framework that provides a block-structured domain decomposition and MPI-based synchronization functions for halo layers.

Feasibility and performance results for this framework have been shown already in the context of large-scale phase-field simulations using the finite element method [12].

B. Automatic Stencil Code Differentation

In the following, we take advantage of the symbolic representation of the forward stencils as *SymPy* assignments and use the capabilities of this computer algebra system to implement a set of rules to automatically generate the assignments defining the corresponding adjoint. This distinguishes our method from automatic derivation of adjoint finite difference in the *Devito* framework [25] which uses *SymPy* to derive the adjoint directly from the symbolic representation of the partial differential equation and its discretization for finite differences. Our approach is not bound to a specific discretization scheme and can therefore also be applied for LBM.

1) Backward Automatic Differentiation: As aforementioned, we can automatically generate CPU or GPU code for one time step of the targeted method if we are able to express the necessary calculations as a set of symbolic assignments operating on relative read and write accesses. We consider therefore one time step as an elementary operation for our automatic backward differentiation procedure (see Figure 1). This means we determine the gradients of an arbitrary objective function by successively applying the chain rule for each time step and propagate the partial gradients backward in time.

We will represent the forward kernel mathematically as a vector-valued function $f(.) = (f_1(.), f_2(.), \ldots, f_M(.))$ which depends on argument symbols $r_0, r_1, \ldots, r_N$ for N read accesses. The result of each component will be assigned to one of the symbols $w_0, w_1, \ldots, w_M$ representing M write

accesses. Both w_i and r_j operate on two sets of fields that may not be disjunctive.

$$w_i \longleftarrow f_i(r_0, r_1, \ldots, r_N) \qquad i \in \{1, \ldots, M\} \tag{4}$$

Symbolic differentiation as provided by *SymPy* is used to determine and simplify corresponding assignments for the calculation of the discrete adjoint and also for common subexpression elimination. For automatic backward differentiation, an additional buffer for each intermediate result in the calculation graph is required. We denominate relative write and read accesses to adjoint variables corresponding to w_i and r_j with $\hat{w}_i$ and $\hat{r}_j$. In other words, our adjoint kernel calculates the Jacobian of the outputs w_i of one time step of the forward pass with respect to its inputs r_j in order to apply the multidimensional chain rule and to backpropagate the accumulated gradients $\hat{w}$ to $\hat{r}$.

$$\hat{r}_j \longleftarrow \sum_{i=1}^{M} \frac{\partial f_i}{\partial r_j}(r_0, r_1, \ldots, r_N) \cdot \hat{w}_i \qquad j \in \{1, \ldots, N\} \tag{5}$$

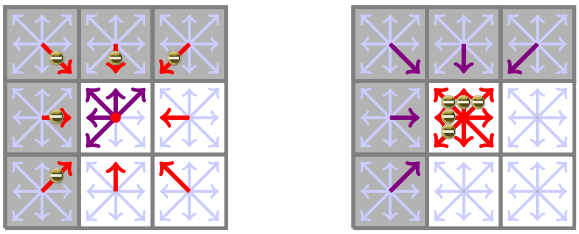
Equation 5 is represented, symbolically evaluated, and simplified directly in *SymPy*. Note that store accesses are transformed into loads and load accesses into stores, while the relative offsets remain unchanged. As long as the stencils defined by r_j do not overlap, stores to $\hat{r}_j$ can be performed in parallel. Otherwise, the stores have to be realized by atomic additions, accumulating the contributions of each grid cell. This can critically harm execution performance.

In the case of a two-buffer LBM scheme, there are no overlapping read and write accesses. Hence, automatic backward differentiation can safely be applied for parallel execution.

2) Backward Automatic Differentation with Forward Memory Access Patterns: A change in the memory access patterns by transforming forward kernels into backward kernels may not be an issue if the parallel execution can still be guaranteed. However, this is not the case for most stencil operations and one must be aware of the fact that a change in the memory access pattern also affects the implementation of boundary handling. As previously proven by Hückelheim et al. [32], the derivations in the sum of equation 5 can be reordered to mimic the access patterns of the forward pass if certain conditions are met. This is based on the observation that in most cases the forward assignments are written in a form that the write operations can be performed collision-free and in parallel on each output cell. If the perspective is changed and indexing based on the read buffers, Equation 5 can be reformulated in order to yield equivalent operations with collision-free writes in the backward pass: For the sake of simplicity of notation, we assume that all read accesses $r_0, r_1, \ldots, r_N$ operate on the same scalar field r. In this case, we can state that each cell of r will have exactly once the role of each $r_0, r_1, \ldots, r_N$. Therefore, we can simply sum over all the read accesses in the forward kernel in order to obtain the gradient $\hat{r}$ for each cell of the read field r:

$$\hat{r} \longleftarrow \sum_{j=1}^{N} \sum_{i=1}^{M} \frac{\partial f_i}{\partial r_j}(r_0, r_1, \ldots, r_N) \cdot \overline{w}_j \tag{6}$$

Since in TF-MAD, the indexing is based on the read accesses in the forward pass, the sign of the relative indexes of $\hat{w}$ has to be switched, which is indicated by $\overline{w}$, e.g., the south-east neighbor of the forward write is the north-west neighbor of the forward read.



(a) Forward boundary handling (b) Backward boundary handling

Figure 2. Adjoint bounce-back boundary: invalid red memory locations and have to be swapped with facing violet locations.

C. Automatic Generation of Adjoint Lattice Boltzmann Methods

To calculate the correct adjoint, not only the method itself but also the boundary handling has to be adapted. Using backward-mode automatic differentiation, a LB scheme with pull-reads from neighboring cells is transformed into a push-kernel that writes to neighboring cells. This transformation also changes the memory access pattern of all boundaries as shown for a bounce-back boundary in Figure 2. Similar to compute kernels, the boundary treatment is also generated from a symbolic representation. This allows us to re-use the same auto-differentiation techniques that we applied earlier to compute kernels and generate backward boundary kernels from their respective forward implementations. The only difference of boundary kernels is their iteration pattern. While compute kernels are executed for each cell, a boundary kernel is executed only in previously marked boundary cells. Boundary values like density or velocity can either be set to a constant or marked for optimization. In the latter case, the gradient with respect to the boundary parameters is automatically derived and calculated as well. All boundary options are accessible to the user through a simple, high-level user interface.

D. Interface to Machine Learning Frameworks

Next, the automatically generated, efficient CPU/GPU implementations of a LB time step with boundary treatment are integrated into the *Tensorflow* and *Torch* packages. We provide building blocks for the LB time step itself and for initialization and evaluation steps that compute LB distribution functions from macroscopic quantities and vice versa. For performance reasons, it is beneficial to combine multiple LB time steps into a single *Tensorflow*/*Torch* block.

This allows the user to easily specify an optimization problem by constructing a computation graph in *Tensorflow* or *Torch*. The user can choose which quantities should be optimizable, constant or constant over time. The system then automatically derives the necessary gradient calculations. Both machine learning frameworks offer test routines that check whether gradients are calculated correctly by comparing them to results obtained via a time-intensive but generic numerical differentiation method. We use these routines to ensure the correctness of our generated implementations. For simplicity, we do not use advanced checkpointing schemes, like *revolve* [33], and instead save all the intermediate results of the forward pass. Alternatively, the user can opt to checkpoint only each n-th time step and use interpolated values for determination of the Jacobian of the missing forward time steps.

IV. Evaluation

For evaluation of our framework, we chose a simple geometry optimization problem after verifying the correctness

of our gradient calculations for standard LBM methods by numerical differentiation. Many LBM-based approaches were proposed to make cell-wise optimization stable and feasible, mainly differing in the way of defining the objective functions and their porousity models. The values of various parameters are crucial for the existence of non-trivial solutions and the stability of an optimization scheme.

Our code generation framework allows us to specify objective functions and porous media models in an abstract way that is very close to their mathematical description. Thus one can implement setups from literature without much development effort. As an example we present here a setup investigated by Nørgaard et al. [31]. They use the partial bounceback model of Zhu and Ma [34] and achieve better optimization stability by separating the design domain from the physical permeability. Physical permeability is obtained by applying a filter followed by a soft-thresholding step. The "hardness" of the thresholding is increased during the optimization procedure to enforce a zero/one solution.

We set up a similar problem as described in their work, by requiring a generated geometry in the design domain to have as close area as possible to a certain fraction θ of the total available domain space and enforcing minimal pressure drop in steady-state. We operate in a low Reynolds number regime with fixed velocity (0.01 lattice units per time step, inlets on left side) and fixed pressure conditions (1 in lattice units, outlets on right side) as shown in Figure 3. With the same objective function [31], that penalizes pressure drop, deviation from a given volume fraction and instationarity of the LB simulation, we obtain the results as shown in Figure 3.

V. Conclusion

Our work shows that adjoint LBMs can be automatically derived from a high-level description of the forward method. This makes optimization based on this method also tractable for non-CFD-experts in a wide range of problem domains, by hiding the inherent complexity caused by adjoint derivation, boundary handling, and model implementation.

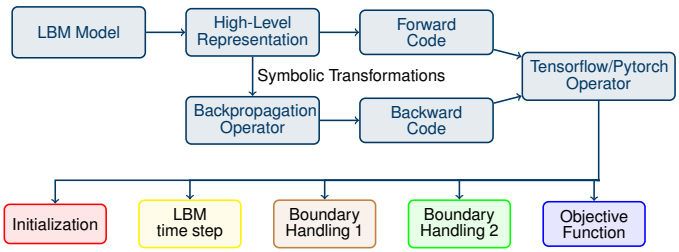


Figure 4. Proposed work flow for optimization with ALBM.

In our exemplary geometry optimization problem, we could successfully apply our suggested automated work flow (Figure 4): a high-level LB model layer is used to derive a porous media LB scheme, define the simulated geometry and generate corresponding *SymPy* expressions for all necessary operations including initialization, time stepping and boundary handling. Forward expressions are transformed into their respective adjoint and finally combined into *Tensorflow* and *Torch* operations. Note that we decided for performance reasons to fuse time steps and boundary operations to a single operation. Derived gradients are automatically assessed for correctness in concrete problem instances by numerical differentiation.

Code generation is well suited for adjoint-based optimization, especially ALBM. Our tool can be used to experiment with different LB models, boundary treatments, optimizers and objective functions without facing the burden of a manual implementation. The symbolic intermediate representation facilitates a simple automatic derivation of gradients for 2D and 3D models. This helps to maintain clean and re-usable code bases. Our code generation approach still guarantees good performance by delegating hardware-specific optimizations to different backends (CPU, GPU), which has been proven for large-scale phase-field simulations [12], though a performance evaluation for LBM and ALBM is still future work. Portability is provided since integration for new frameworks only requires minimal wrapper code to call our dependency-free C code. We hope that this approach could help to save time and money to bring future code from prototypical experiments to large-scale production.

Our initial work leaves room for a lot of open questions, which need to be evaluated in the future under more realistic conditions. Scientific computing applications need to proof optimum execution performance also on large scale MPI systems. Efficient CFD-based optimization needs to use dedicated optimization frameworks with support for MPI simulations and efficient step size control while machine learning frameworks might be sufficient only for prototyping. Also a more advanced checkpointing strategy, like *revolve* [33] is needed to reduce memory usage. Furthermore, more research is required to test which generated ALBM models are suited in practice by analyzing their performance in real world examples.

Acknowledgments: Stephan Seitz thanks the International Max Planck Research School Physics of Light for supporting his doctorate.

References

[1] I. M. Navon, "Data assimilation for numerical weather prediction: A review," Data Assimilation for Atmospheric, Oceanic and Hydrologic Applications, 2009, pp. 21–65.

[2] F. Klemens, S. Schuhmann, G. Guthausen, G. Thäter, and M. J. Krause, "CFD-MRI: A coupled measurement and simulation approach for accurate fluid flow characterisation and domain identification," Computers & Fluids, vol. 166, 2018, pp. 218–224.

[3] T. Kim, N. Thürey, D. James, and M. Gross, "Wavelet turbulence for fluid simulation," ACM Trans. Graph., vol. 27, no. 3, Aug. 2008, pp. 50:1–50:6.

[4] Ł. Łaniewski-Wołłk and J. Rokicki, "Adjoint lattice boltzmann for topology optimization on multi-gpu architecture," Computers & Mathematics with Applications, vol. 71, no. 3, 2016, pp. 833–848.

[5] S. Yang, S. Kiang, P. Farzan, and M. Ierapetritou, "Optimization of reaction selectivity using CFD-based compartmental modeling and surrogate-based optimization," Processes, vol. 7, no. 1, Dec 2018, p. 9.

[6] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen, The Lattice Boltzmann Method. Springer International Publishing, 2017.

[7] M. M. Tekitek, M. Bouzidi, F. Dubois, and P. Lallemand, "Adjoint lattice boltzmann equation for parameter identification," Computers & fluids, vol. 35, no. 8-9, 2006, pp. 805–813.

[8] M. J. Krause, G. Thäter, and V. Heuveline, "Adjoint-based fluid flow control and optimisation with lattice boltzmann methods," Computers & Mathematics with Applications, vol. 65, no. 6, Mar 2013, pp. 945–960.

[9] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, http://arxiv.org/abs/1603.04467, Software available from tensorflow.org.

[10] R. Collobert, S. Bengio, and J. Marithoz, "Torch: A modular machine learning software library," 2002, technical report, infoscience.epfl.ch/record/82802/files/rr02-46.pdf.

 ISBN: 978-1-61208-732-0

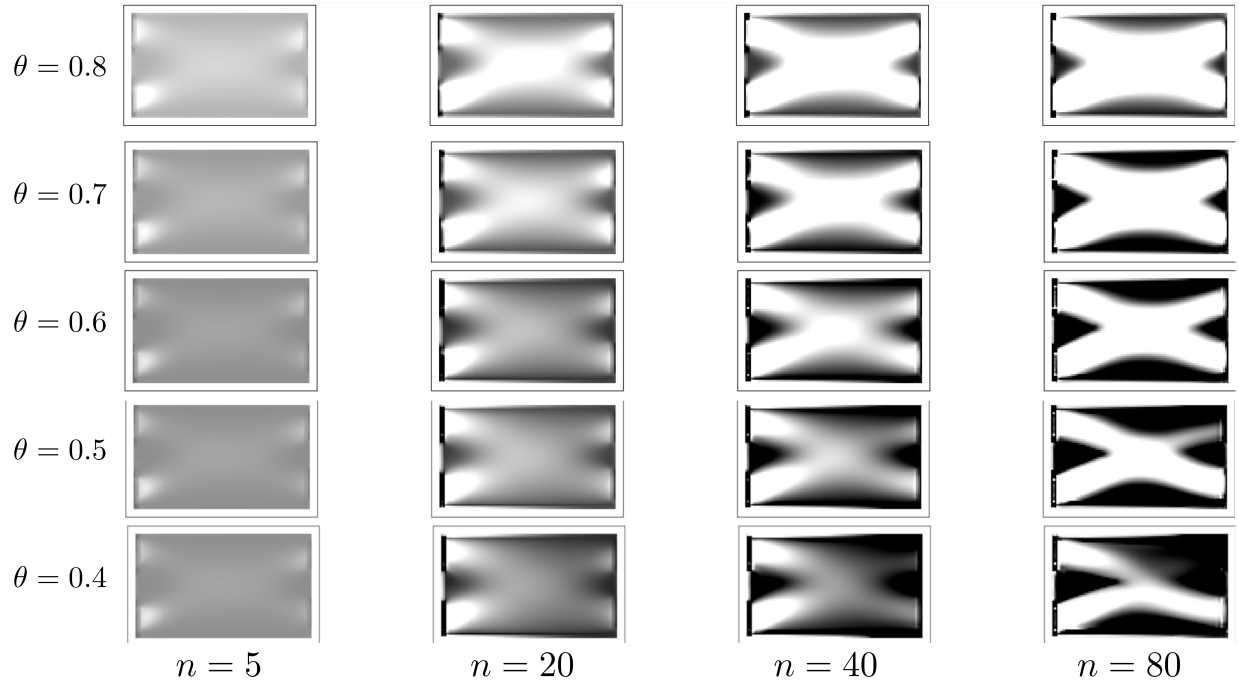


Figure 3. Optimization solution for a simple minimal-pressure-drop design after n optimization steps for given fraction θ of the design area targeted. Gray value intensity indicates permeability.

[11] A. Meurer *et al.*, "SymPy: symbolic computing in Python," PeerJ Computer Science, vol. 3, Jan. 2017, p. e103.

[12] M. Bauer *et al.*, "Code generation for massively parallel phase-field simulations," in Accepted at: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2019, in press.

[13] M. Bauer *et al.*, "pystencils," software available at www.github.com/mabau/pystencils.

[14] I. Ginzburg, F. Verhaeghe, and D. d'Humieres, "Two-relaxation-time lattice boltzmann scheme: About parametrization, velocity, pressure and mixed boundary conditions," Communications in computational physics, vol. 3, no. 2, 2008, pp. 427–478.

[15] P. Lallemand and L. Luo, "Theory of the lattice boltzmann method: Dispersion, dissipation, isotropy,galilean invariance, and stability," Physical Review E, vol. 61, no. 6, 2000, pp. 6546–6562.

[16] D. D'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.-S. Luo, "Multiple-relaxation-time lattice Boltzmann models in three dimensions." Philosophical transactions. Series A, Mathematical, physical, and engineering sciences, vol. 360, no. 1792, 2002, pp. 437–451.

[17] M. Geier, M. Schönherr, A. Pasquali, and M. Krafczyk, "The cumulant lattice Boltzmann equation in three dimensions: Theory and validation," Computers and Mathematics with Applications, vol. 70.4, 2015, pp. 507–547.

[18] F. Bösch, S. S. Chikatamarla, and I. V. Karlin, "Entropic multirelaxation lattice boltzmann models for turbulent flows," Physical Review E - Statistical, Nonlinear, and Soft Matter Physics, 2015, p. 043309.

[19] J. Latt and *et al.*, "Palabos, parallel lattice Boltzmann solver," 2009, software available at palabos.org.

[20] D. Lagrava, O. Malaspinas, J. Latt, and B. Chopard, "Advances in multi-domain lattice boltzmann grid refinement," Journal of Computational Physics, vol. 231, no. 14, 2012, pp. 4808–4822.

[21] M. Krause, A. Mink, R. Trunk, F. Klemens, M.-L. Maier, M. Mohrhard, A. Claro Barreto, H. M., M. Gaedtke, and J. Ross-Jones, "Openlb release 1.2: Open source lattice boltzmann code," 2019, online openlb.net, Accessed Jan. 2019.

[22] C. Godenschwager, F. Schornbaum, M. Bauer, H. Köstler, and U. Rüde, "A framework for hybrid parallel flow simulations with a trillion cells in complex geometries," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 35:1–35:12.

[23] F. Schornbaum and U. Rüde, "Massively parallel algorithms for the lattice boltzmann method on nonuniform grids," SIAM Journal on Scientific Computing, vol. 38, no. 2, 2016, pp. C96–C126.

[24] M. Asch, Data assimilation. Methods, algorithms, and applications, ser. Fundamentals of algorithms. Philadelphia, PA: SIAM, Society for Industrial and Applied Mathematics, 2016, vol. 11.

[25] F. Luporini *et al.*, "Architecture and performance of devito, a system for automated stencil computation," Geosci. Model Dev., July 2018, preprint available at https://arxiv.org/abs/1807.03032.

[26] M. Louboutin *et al.*, "Devito: an embedded domain-specific language for finite differences," Geoscientific Model Development, vol. 12, Aug 2018, pp. 1165–1187.

[27] P. Farrell, D. Ham, S. Funke, and M. Rognes, "Automated derivation of the adjoint of high-level transient finite element programs," SIAM Journal on Scientific Computing, vol. 35, no. 4, 2013, pp. C369–C393.

[28] S. W. Funke and P. E. Farrell, "A framework for automated PDE-constrained optimisation," CoRR, vol. abs/1302.3894, 2013, preprint available at https://arxiv.org/pdf/1302.3894.pdf.

[29] G. Pingen, A. Evgrafov, and K. Maute, "Topology optimization of flow domains using the lattice boltzmann method," Structural and Multidisciplinary Optimization, vol. 34, no. 6, Dec 2007, pp. 507–524.

[30] K. Yaji, T. Yamada, M. Yoshino, T. Matsumoto, K. Izui, and S. Nishiwaki, "Topology optimization using the lattice boltzmann method incorporating level set boundary expressions," Journal of Computational Physics, vol. 274, 2014, pp. 158–181.

[31] S. Nørgaard, O. Sigmund, and B. Lazarov, "Topology optimization of unsteady flow problems using the lattice boltzmann method," Journal of Computational Physics, vol. 307, 2016, pp. 291–307.

[32] J. Hückelheim, P. Hovland, M. Strout, and J.-D. Müller, "Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation," Optimization Methods and Software, vol. 33, no. 4-6, 2018, pp. 672–693.

[33] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation," ACM Trans. Math. Softw., vol. 26, 03 2000, pp. 19–45.

[34] J. Zhu and J. Ma, "An improved gray lattice Boltzmann model for simulating fluid flow in multi-scale porous media," Advances in Water Resources, vol. 56, 2013, pp. 61–76.

 ISBN: 978-1-61208-732-0