Cross-platform programming model for many-core lattice Boltzmann simulations

Jonas Latt*, **Christophe Coreixas**, **Joël Beny**

Computer Science Department, University of Geneva, Carouge, Switzerland

* jonas.latt@unige.ch

Abstract

We present a novel, hardware-agnostic implementation strategy for lattice Boltzmann (LB) simulations, which yields massive performance on homogeneous and heterogeneous many-core platforms. Based solely on C++17 Parallel Algorithms, our approach does not rely on any language extensions, external libraries, vendor-specific code annotations, or pre-compilation steps. Thanks in particular to a recently proposed GPU back-end to C++17 Parallel Algorithms, it is shown that a single code can compile and reach state-of-the-art performance on both many-core CPU and GPU environments for the solution of a given non trivial fluid dynamics problem. The proposed strategy is tested with six different, commonly used implementation schemes to test the performance impact of memory access patterns on different platforms. Nine different LB collision models are included in the tests and exhibit good performance, demonstrating the versatility of our parallel approach. This work shows that it is less than ever necessary to draw a distinction between research and production software, as a concise and generic LB implementation yields performances comparable to those achievable in a hardware specific programming language. The results also highlight the gains of performance achieved by modern many-core CPUs and their apparent capability to narrow the gap with the traditionally massively faster GPU platforms. All code is made available to the community in form of the open-source project `stlbm`, which serves both as a stand-alone simulation software and as a collection of reusable patterns for the acceleration of pre-existing LB codes.

1 Introduction

1.1 Overview

A highly challenging aspect of High Performance Computing (HPC) is the need to reformulate and restructure scientific algorithms to perform well on different types of parallel architectures. In the current hardware landscape, a special focus is devoted to many-core platforms, which include homogeneous systems like the AMD Zen processors investigated in this article, or heterogeneous systems which use a many-core device as an accelerator, including GPUs or Intel's now discontinued Xeon Phi platform. Optimized simulation software can be developed for given platforms using device specific programming languages or programming paradigms,

including the use of multi-threading on many-core CPUs, or the use of dedicated languages like OpenCL or Cuda on GPUs. In these cases, the obtained code can be very different from a desirable code for scientific research, as a strong emphasis is put on meeting hardware constraints rather than express the actual numerical model or simulation algorithm. Furthermore, this approach requires different code bases to be developed for different architectures, which strongly limits its portability and long-term maintainability.

For this reason, multiple language extensions (such as OpenMP and OpenACC) or completely new programming languages (such as Futhark) have been introduced, which propose to achieve notable performance improvements on parallel hardware in a platform-independent manner, through the application of targeted directives or through the adoption of a specific implementation model. In the recent literature, special attention is devoted to functional paradigms, the principles of which are summarized by the so-called map-reduce formalism. In this case, an algorithm to be executed on a collection of data elements is expressed in terms of two element access functions, the "map" operation that applies a transformation to a data element, and the "reduce" operation that expresses a global reduction over the transformed elements. The actual, repeated execution of these operations is expressed by the framework instead of the user, allowing for automatic adaptation and optimization of a code on different devices.

This philosophy of automatic acceleration of functional code is embraced by the C++17 standard. It introduces execution policies for standard algorithms, which we will refer to as Parallel Algorithms, that allow parallel and SIMD optimizations. Examples of algorithms that are accounted for by execution policies are `transform-reduce` (the C++ equivalent of map-reduce), `for_each`, which only addresses the "map" part of map-reduce, or problem-specific functions like the sorting algorithm `sort`. Parallel Algorithms do not necessarily exhibit better performance than traditional language extensions, as a given implementation of the C++ standard library may for example decide to execute Parallel Algorithms with the help of OpenMP directives. They do however offer an elegant formalism, as all parallel constructs are expressed as an inherent part of the C++ language. They furthermore offer a strong guarantee of cross-platform compatibility and long-term maintainability. Indeed, the C++ language is standardized by the International Organization for Standardization ISO and widely embraced by the scientific community. In addition, the maintainability of a code written in C++ is guaranteed, because the language is known for its success in maintaining backward compatibility over several decades.

While the formalism of Parallel Algorithms is quite recent, it has been tested quite thoroughly on many-core CPU systems thanks to the available implementation through Intel's Threading Building Blocks, or its implementation in Visual C++. Only recently though, less than three months before the initial submission of this manuscript, an implementation has been made available that ports Parallel Algorithms to GPUs in form of the NVIDIA HPC SDK C++ compiler, allowing to take profit of a truly cross-platform experience of this formalism.

In this manuscript, we propose a model for the implementation of lattice Boltzmann (LB) codes within the framework of C++ standard algorithms, and show the potential for accelerating such codes with the help of execution policies. One of the challenges consists in the implementation of the streaming step, which is non-local and does therefore not fit naturally into the framework of a repeated application of the "map" element access function to single elements. Instead, we take profit from the liberties allowed by the C++ standard to deduce the global index within a data collection for certain types of iterators (including raw pointers), and manipulate these indices further to achieve non-local data accesses. While this strategy incurs the cost of added arithmetic operations, the numerical tests show that these do not impact the

overall performance in a way that would be prohibitive for the use of this method in a HPC environment.

The proposed modeling environment relies on the usual LB collision-streaming cycles but expresses the local collision through a cell-independent scheme. This approach opposes the frequently adopted strategy of adjusting the local collision scheme to implement boundary conditions (see *e.g.* the Palabos software [1]). Instead, the streaming model is locally adjusted to react to encounters with a wall and implement general Dirichlet boundary conditions, following the common, link-based bounce-back approach [2, 3]. This approach allows to build a framework with the ability to treat both bulk and boundaries coherently across a large variety of collision models, in a manner that is compatible with efficient many-core parallelism. The presented validation case of a lid-driven 3D cavity shows that the selected boundary treatment does not need to hide behind more sophisticated approaches, as the present framework shows to produce stable and accurate results even with the notoriously unstable BGK collision model. It is pointed out that, although this path is not further explored in this work, the link-based bounce-back collision model can be extended to the treatment of general curved boundary conditions while remaining entirely local (see *e.g.* [4]).

The result of our efforts is a framework that allows to freely combine six different memory access schemes with nine different LB collision models and bounce-back based Dirichlet walls. In all cases, efficient, platform agnostic many-core parallelism is available. The framework is disseminated in form of the open-source C++ library `stlbm`, which offers an extensive collection of reusable codes and patterns for the acceleration of LB software. To enhance the educational value of the `stlbm` library, selected self-contained single-file codes are also provided, which reduce the full software framework to a single memory access scheme and LB collision model. They illustrate that it is possible to write a cross-platform, efficiently parallelized LB application in a readable manner and with less than 500 lines of code, including all necessary data analysis and post-processing functionalities.

The remainder of this introductory section provides a short overview of programming models used to implement parallel software on many-core platforms, and specifically on strategies used to accelerate LB programs on such platforms. Section 2 summarizes the method used in this article to implement the LB algorithm and explains in some detail the six schemes (with different data structures and memory access patterns) and nine collision models available in `stlbm`. It then explains the programming model of C++ Parallel Algorithms, and explains the methodology used to fit the LB algorithm in this model. It finally lists the hardware platforms used to test the performance of the `stlbm` library. Section 3 presents a validation case for the `stlbm` library (a flow in a lid-driven cavity) and performance measurements for the various implementation schemes, collision models, and hardware platforms considered in this article. Finally, conclusions are drawn in Section 4. Section 5 provides a link to the repository containing the open-source `stlbm` library and to the detailed performance measurements obtained with `stlbm`.

1.2 Programming models for many-core platforms

The present article investigates two types of many-core platforms, namely homogeneous systems, in the form of many-core CPUs, and heterogeneous systems, in the form of systems accelerated with GPUs. In the latter case, the term "heterogenous" refers to the fact that the program is initially executed on a host (the CPU) which handles the program setup and data initialization, after which the computationally intensive tasks are typically offloaded to a large extent to a device (the GPU). Other types of devices, which are not explicitly reviewed here, include FPGAs and Neural Processing Units (NPUs).

Applications on many-core platforms can be programmed using low-level programming models, which include threading mechanisms such as POSIX Threads [5] for homogeneous systems. On heterogeneous systems, a strong standard is available in the form of OpenCL, which offers a vendor-independent programming model. In practice however, the vendor specific Cuda programming language [6] has established itself as a *de facto* standard for heterogeneous GPU platforms, a fact which is sometimes attributed to the difficulty of achieving optimal cross-vendor performances in view of the diversity of many-core architectures [7].

Higher-level programming models, which are the focus of this article, relieve the programmers from hardware specific details and allow an increased focus on the implemented algorithm, and potentially a greater independence from the selected hardware platform. The two reviewed approaches are

- **Directive based programming models**. This approach, based on extra-language directive annotations, includes industry standards like OpenMP [8], which is mostly targeted at homogeneous many-core systems, and OpenACC [9] for heterogeneous platforms. They typically allow a code to run both on conventional platforms (on which the directives are simply ignored) and on accelerator-supplied systems.

- **Programming models based on C++ and on the C++ STL**. C++-based programming models include DPC++ that is part of Intel's OneAPI [10], C++ AMP which is based on DirectX11, AMD's GPU-oriented C++ API [11], and the unified PACXX programming model [12] that allows to write both host and device code within the C++14 standard. Recently, an increased interest is observed for programming models that are based on the C++ Standard Template Library (STL) or on extensions thereof. For example, Thrust [13] is a parallel template library in C++ inspired by the STL, which allows a hardware independent programming style for both many-core CPUs and Nvidia GPUs. However, hardware independent program development for many-core platforms is also possible within the strict limits of the C++17 standard, which introduces the concept of Parallel Algorithms. This concept is exploited by Intel's Threading Building Blocks (TBB) [14], as well as by the STL implementation of Visual C++, which provide a backend to C++ Parallel Algorithms for homogeneous many-core systems. Finally, NVidia introduced very recently a version of the NVidia HPC SDK which offers an NVidia GPU backend to C++ Parallel Algorithms. The TBB and the NVidia HPC SDK backends are used in the present work to test the performances of the proposed programming model for LB applications.

A more thorough review of programming models for homogeneous and heterogeneous many-core platforms is found in [7].

1.3 Acceleration of lattice Boltzmann codes on many-core systems

The LB method has always been considered an excellent candidate for efficient parallelism, most notably thanks to its convenient separation into a local collision step and a streaming step with non-local memory accesses of limited extent. Without investigating the numerous implementations and methodologies in detail, we point out the importance which has been devoted, across various systems and platforms, to the reduction of the memory footprint, the improvement of memory access patterns, and the definition of adequate and efficient data structures. Pohl *et al.* [15] propose an implementation strategy which requires the allocation of a single set of populations only (referred to as a *compressed grid* approach) and executes a collision and streaming step for all cells within a single memory traversal (referred to as a *fused collision-streaming step*). This approach, which is at the heart of the open-source WaLBerla library [16], stands at the beginning of a long list of subsequent, similar LB implementation

models. We mention in particular the swap algorithm [17], which is used in the open-source Palabos library [1] and the AA-pattern [18], which was described specifically in the context of GPU implementations of LB models, but has proved highly successful on CPU systems as well [19].

With the advent of general-purpose GPU (GPGPU) computing, GPUs have quickly become a target for LB implementations as well. An original proposition published in [20] proved promising in spite of its fairly poor performance, and was further improved in form of the 2D LB code proposed in [21]. Further performance improvements were exhibited over the years, often by fine-tuning memory access patterns, as published for example in [22–25].

To a lesser extent, LB implementations using the cross-platform formalism OpenCL are found in the literature, which document attempts to target GPUs of different vendors, but also the now discontinued Xeon Phi architecture of Intel (see for example [26, 27]). We also point out the attempts performed with the open-source Sailfish project [28] to use meta-programming capabilities of Python to target multiple platforms through both a Cuda and an OpenCL backend.

2 Materials and methods

2.1 LB algorithm

The Lattice Boltzmann method is a very specific (physical and numerical) discretization of the Boltzmann equation that splits explicit time iteration into a collision and a streaming step. Space is subdivided into N_{tot} cells that are in principle equally spaced with a distance δx along the three principal Euclidean directions. Inhomogeneous cell arrangements can be achieved through mesh refinement strategies [1, 29, 30], which are however not taken into account in this work. The state of the system is defined on every cell by a certain amount of scalar values $f_k(k = 0, \ldots, Q - 1)$, hereafter called populations. Their number Q depends on the stencil chosen to discretize the phase space (i.e., physical discretization of the Boltzmann equation). The present work is based on the commonly used 19-velocity stencil D3Q19 [3, 31, 32], a choice that limits the current scope of the implementation to isothermal flows. A time iteration takes the populations from their state at time t to the next state $t + \delta t$, where δt is a constant discrete time step:

$$Collision: \quad f_k^{\text{out}}(\boldsymbol{x}, t) = \Omega_k(f(\boldsymbol{x}, t)) \tag{1}$$

$$Streaming: \quad f_k(\boldsymbol{x} + \boldsymbol{c}_k \delta t, t + \delta t) = f_k^{\text{out}}(\boldsymbol{x}, t) \tag{2}$$

The 19 discrete velocities $\boldsymbol{c}_k$, which connect lattice nodes with near neighbors in lattice units (the components of the $\boldsymbol{c}_k$ have integer values), are defined for example in [3]. The temporary values f_i^{out}, sometimes called "outgoing populations", may be stored either in a separate cell array, or in the same array as the f_k, using an in-place value replacement scheme (see Section 2.3).

With the link-wise bounce-back scheme used presently to implement Dirichlet boundaries, populations that encounter a physical wall on the way from the original cell to the neighboring one, revert direction and are assigned to the original cell, in opposite direction [3, 33]. The adapted streaming step reads:

$$f_{\bar{k}}(\boldsymbol{x}, t + \delta t) = f_k^{\text{out}}(\boldsymbol{x}, t) \underbrace{-6 \, t_k \, \rho_w \, \boldsymbol{c}_k \cdot \boldsymbol{u}_w}_{\text{momentum exchange}}. \tag{3}$$

Here, $\bar{k}$ stands for the opposite direction of k (i.e., $\boldsymbol{c}_{\bar{k}} = -\boldsymbol{c}_k$). In addition, $\boldsymbol{u}_w$ is the velocity

imposed by the Dirichlet boundary condition, and ρ_w is the "wall density". In the present work, $\rho_w = 1$ because the flow runs in a quasi-incompressible regime and exhibits density fluctuations around 1. This boundary condition scheme places the wall half-way between a fluid node and the next solid node, although the exact wall location may be viscosity-depend for low Reynolds number flows. This problem is circumvented through the use of the TRT collision term [34] (one of the nine models implemented in this article), which should be the preferred choice in pore-scale porous media flows and other high-viscosity flows that are sensitive to the wall location.

The link-wise bounce-back (3) scheme only involves populations that originate from, or move toward, fluid cells. Therefore, no storage of populations is required on solid cells. The strategy chosen in stlbm to implement boundary conditions consists in allocating nevertheless a layer of wall cells in the solid domain, with 19 storage locations like common fluid cells. These wall cells do not store populations, but are rather assigned pre-computed values of the momentum exchange term appearing in Eq (3). Thus, the 19 storage locations $\mathtt{pop}_k$ of a cell are utilized as follows:

$$\mathtt{pop}_k = f_k \qquad \text{on fluid cells in pre-collision state (except for the AA-pattern),} \qquad (4)$$

$$\mathtt{pop}_k = -6\, t_k\, \rho_w\, \boldsymbol{c}_k \cdot \boldsymbol{u}_w \qquad \text{on solid cells.} \qquad (5)$$

As explained below, the situation is slightly more complex with one of the three implemented memory access schemes, the AA-pattern, which obeys Eq (4) at even time steps only, but retrieves pre-collision populations from neighboring fluid cells at odd steps instead. With the compact storage scheme expressed in Eqs (4) and (5), the streaming step is expressed in a uniform manner as an access of neighboring cells to achieve two goals, namely, (1) to copy populations and execute conventional streaming, and (2) to read the pre-computed momentum exchange term and sum it up with a reverted population. This latter possibility is expressed in the rightmost column of Table 1.

Table 1. The algorithms executed for the three implemented schemes.

nb: *neighbor*, ←: *assignment*, ↔: *swap*			
	Collision	**Communication**	
		nb is fluid	**nb is wall**
		Streaming: k runs over 18 values	
2Pop	$\mathtt{pop}_k \leftarrow \Omega_k(\mathtt{pop})$	$\mathtt{nb\_out}_k \leftarrow \mathtt{pop}_k$	$\mathtt{out}_{\bar{k}} \leftarrow \mathtt{pop}_k + \mathtt{nb\_pop}_k$
		Streaming: k runs over 9 values	
Swap	$\mathtt{pop}_{\bar{k}} \leftarrow \Omega_k(\mathtt{pop})$	$\mathtt{pop}_{\bar{k}} \leftrightarrow \mathtt{nb\_pop}_k$	$\mathtt{pop}_{\bar{k}} \leftarrow \mathtt{pop}_{\bar{k}} + \mathtt{nb\_pop}_k$
AA/Even	$\mathtt{pop}_{\bar{k}} \leftarrow \Omega_k(\mathtt{pop})$	*Pull*: k runs over 18 values	
AA/Odd		$\mathtt{tmp}_k \leftarrow \mathtt{nb\_pop}_{\bar{k}}$	$\mathtt{tmp}_k \leftarrow \mathtt{pop}_k + \mathtt{nb\_pop}_{\bar{k}}$
	$\mathtt{tmp}_k \leftarrow \Omega_k(\mathtt{tmp})$	*Push*: k runs over 18 values	
		$\mathtt{pop}_k \leftarrow \mathtt{tmp}_k$	$\mathtt{pop}_{\bar{k}} \leftarrow \mathtt{tmp}_k + \mathtt{nb\_pop}_k$

This table summarizes the algorithm that must be executed on all fluid cells in each of the three schemes. Only in the swap algorithm, wall cells are also included in the collision-streaming cycle, and apply the following operation to all fluid cells among the 9 considered neighbors: $\mathtt{nb\_pop}_k \leftarrow \mathtt{pop}_{\bar{k}}$. The following notation is used: $\mathtt{pop}$ denotes the 19 variables of the local cell, at position $\boldsymbol{x}$, and $\mathtt{nb\_pop}$ the variables of the neighbor cell in direction k, at position $\boldsymbol{x} + \boldsymbol{c}_k$ (*e.g.* $\mathtt{nb\_out}_{\bar{k}}$ denotes the variable of index $\bar{k}$ of the cell at position $\boldsymbol{x} + \boldsymbol{c}_k$). The double-population scheme uses a duplicate array of cells which are denoted as $\mathtt{out}$. Finally, the AA-pattern uses a temporary cell $\mathtt{tmp}$ which is entirely local and does not require the allocation of a global cell-array like $\mathtt{out}$.

Implementations of the LB algorithm can be found in the literature using both single-precision and double-precision floating point values. Single-precision implementations are particularly popular on GPUs, which are usually substantially more efficient in single precision, except on few high-end GPUs targeted at HPC platforms (which are not tested in the present work). Single-precision computations in LB simulations carry however the risk of round-off errors that are difficult to anticipate, especially in research codes, although mitigation strategies exist (see *e.g.* [35]). To emphasize its goal, which is to be reused in the context of general-purpose research and production codes, the stlbm library has been developed and tested with double precision variables only (although an adaptation to single precision would be easy).

2.2 LB collision models

In the present work, nine collision models are considered to evaluate the impact of the collision step (1) on the overall performance of the LB code. Every model is written based on the unified framework proposed by Coreixas et al. [36, 37], which is particularly useful to derive efficient formulations that do not rely on a matrix form.

More precisely, these collision models include standard operators such as the BGK operator with a weighted second-order equilibrium (BGK-W2 [38]), a non-weighted fourth-order equilibrium (BGK-NW4 [36, 39]), and the two-relaxation-time formulation (TRT [34, 40]). In addition, multi-relaxation-time collision models based on raw (RM [41–43]), Hermite (HM [44–47]), central (CM [39, 48–50]), central Hermite (CHM [51–53]) moment spaces, as well as cumulants (K [54–57]), are further considered. Finally, a multi-relaxation-time formulation of the recursive regularized (RR) collision model is included due to its interesting stability property [58–62]. It is worth noting that most of these multi-relaxation time collision models have been proposed for the D2Q9 and D3Q27 lattices, but in this work, we used *non-weighted* D3Q19 formulations that are summarized in Appendix G of Ref. [36]. It was also made possible for the user to freely adjust the bulk viscosity, through an extra parameter, in order to further increase the stability of the code when needed.

More details regarding the implementation of these collision models is provided in Section 2.6, whereas their impact on the performance is presented in Section 3.3.2.

2.3 Data structure and ordering of populations

To implement an LB scheme, the populations f_k must be stored at a provided memory location. This memory location can vary in time, and it may or may not be identical for pre- and post-collision variables (f_k and f_k^{out} respectively). A naive attempt to implement an in-place algorithm, which is to rely on the same storage space for populations at time t and $t + \delta t$, would lead to a conflict with the non-local nature of the streaming step. Indeed, as post-collision populations are streamed to neighboring cells, they potentially overwrite populations that have not yet completed their collision-streaming cycle, hence leading to erroneous results. In search of a solution, additional constraints to consider are:

- **Reduction of memory requirements**. A low memory footprint is desirable because (1) memory may be a critical resource on a given system, and (2) the total amount of memory processed by a system may impact its performance.

- **Reduction of memory accesses**. Although the actual picture is complicated by the presence of cache mechanisms, performance is generally improved by limiting the number of read- and write-accesses to the main memory.

- **Thread safety**. This feature is required to allow the multi-threaded execution models used in this article. Thread safety is achieved without synchronization primitives or locks, by applying schemes that naturally avoid race conditions. A global synchronization is applied only once per time iteration (or twice, in case of the swap algorithm). Within this context, achieving thread safety amounts to making sure that the collision-streaming algorithm is independent of the order of traversal of the cells.

This article relies on three popular thread-safe schemes (double-population scheme, swap algorithm, and AA-pattern), which are shortly described below. Their corresponding algorithms are summarized in Table 1. All three schemes rely on an array of $N_{tot} \times 19$ floating-point variables pop. This amounts to 19 variables for each node location $\boldsymbol{x}$, which are accessed through the syntax $\mathtt{pop_k}\,(\boldsymbol{x})$. The actual memory layout of the poparray is discussed at the end of this section.

2.3.1 Double-population scheme.   Probably the simplest of the three algorithms, the double-population scheme allocates a second array of size $N_{tot} \times 19$, hereafter called out. While this strategy doubles the total memory requirements, it simplifies the algorithm, as both the $f_k(t)$ and $f_k^{\mathrm{out}}(t)$ values are stored at the memory location $\mathtt{pop_k}$, while the streamed variables $f_k(t+1)$ are kept separately in $\mathtt{out_k}$. Therefore, the collision and streaming steps can be fused (*i.e.* they are executed within a single memory traversal), as any access conflicts are naturally avoided. After a time iteration, an exchange of the arrays pop and out guarantees that the streamed, temporary populations are reused for the subsequent cycle. While the large memory footprint of this scheme may result in severe performance penalties on CPUs (due to a less efficient use of cache memory), benchmark tests show that it performs quite well on GPUs (see Section 3.1). It is also noted that the overall number of memory accesses is kept low in this scheme, with a total of $N_{tot} \times 19$ read accesses and $N_{tot} \times 19$ write accesses per time iteration.

2.3.2 Swap algorithm.   This approach relies on the observation that for every population sent during streaming (from a cell to its neighbor), the cell also receives a population from the same neighbor. Therefore, the two copy operations can be fused into a single value swap, which prevents any value from being overwritten. Consequently, the algorithm is in-place and relies on no further global memory than the pop array. Given that a swap operation includes two copies of the streaming step, the loop executing the streaming operation on a cell spans over nine populations only instead of the usual 18—with the understanding that the nine opposite populations are taken care of by the swap operation of a neighboring cell. To guarantee that the swapped variables are found in matching locations, post-collision populations must be stored at a storage location associated to the opposite pre-collision population: $\mathtt{pop_{\bar{k}}} \leftarrow f_k^{\mathrm{out}}$. While it is memory efficient, this algorithm can unfortunately not at the same time offer a fused collision-streaming cycle and offer thread safety. Indeed, while carrying out a swap operation, it must be guaranteed that the populations of the neighbor involved in the swap are already in post-collision state. In a fused collision-streaming scheme, this can be enforced by adjusting the order of traversal of cells with the ordering of the discrete velocities, as it is for example done in the Palabos code [1]. The dependence on the order of traversal however violates the principle of threat safety. Instead, the version of the swap algorithm used presently splits collision and streaming in two steps requiring each a traversal of the full set of data in pop, and separated by a thread synchronization. Thus, the memory needs are half those of the double-population scheme, but the number of memory accesses is doubled, with a total of $2N_{tot} \times 19$ read accesses and $2N_{tot} \times 19$ write accesses per time iteration.

2.3.3 AA-pattern.   This pattern combines the advantages of the double-population scheme and the swap algorithm, as it offers (1) a fused collision-streaming step, and (2) a single-memory implementation (since it requires no other global data than the pop array). To achieve

this, the data is however stored in different locations at two subsequent time step, and the algorithm must distinguish even and odd time steps. More precisely, the AA-pattern transfers the streaming step performed at the end of an even iteration to the beginning of the next odd iteration. Hence, two communication steps are incorporated at odd iterations, in terms of a "Pull" operation that gathers the populations from the neighbor cells to a local and temporary array for collision, before a "Push" operation that eventually writes the post-collision variables back to the same locations at the neighbors. Thread safety is guaranteed by virtue of the fact that a given cell (handled by a single thread) accesses the same non-local data during its read (Pull) and write (Push) access. Going into more details, at even steps the populations are stored locally, and the value of $f_k(\boldsymbol{x})$ is available at the location $\mathrm{pop_k}\,(\boldsymbol{x})$. Regarding post-collision values, they are also stored locally but at opposite locations, like in the swap algorithm: $\mathrm{pop_{\bar{k}}} \leftarrow f_k^{\mathrm{out}}$. At odd steps, the populations are available on neighboring nodes, with $f_k(\boldsymbol{x})$ stored at $\mathrm{pop_{\bar{k}}}\,(\boldsymbol{x} + \boldsymbol{c}_k)$.

All in all, the AA-pattern is a compressed scheme which, like the swap algorithm, cuts the memory requirements of the double-population scheme in half, but favorably maintains the same number of memory accesses, with an average of $N_{tot} \times 19$ read accesses and $N_{tot} \times 19$ write accesses per time iteration. It should however also be mentioned that the AA-pattern is more complex than the two other schemes, and introduces technical difficulties for the maintenance of an LB software framework, because of the separation into even and odd time steps.

Finally, the data can be aligned in two different ways in memory, independently of the chosen data structure. In the first layout, referred to as array-of-structure (aos), the populations of a given cell are aligned consecutively in memory. In the second approach, which carries the name of structure-of-array (soa), all populations corresponding to a given direction k are consecutive. The two following statements summarize the essence of the two data layouts using a C-array syntax:

$$\text{Array-of-structure}: \quad \texttt{double pop[Ntot][19];}$$

$$\text{Structure-of-array}: \quad \texttt{double pop[19][Ntot];}$$

By combining these two data layouts with the three proposed LB implementation schemes, the `stlbm` library offers a total of six implementation strategies for each collision model, which produce identical results but exhibit different, platform-dependent performance figures, as explored in this article.

2.4 STL and Parallel Algorithms

The C++ Standard Template Library (STL) supports a functional programming style, in which algorithms are supplemented with element access functions which customize their behavior. The algorithms apply these functions repeatedly to the elements of a data container, either through function pointers or function objects. At the application programmer level, an explicit loop is then replaced by a single call to an STL algorithm, hence leading to a programming style that is sometimes considered to be more expressive. The machine codes generated by an algorithm invocation or an explicit loop do not necessarily differ, as a compiler can take advantage of the mechanisms of C++ templates to translate the algorithm to an identical or similar loop as the hand-written one. This is especially true when the implementation of the element access function is known by the compiler at the time of the algorithm instantiation, in which case the function body can be efficiently inlined.

The following C++ code extract illustrates this concept with the STL algorithm `for_each`, used to compute macroscopic variables in the manner of a typical C++ program:

```
using namespace std;
struct Cell {
    array <double, 19> f;
    double rho;
    array<double, 3> u;
};
vector <Cell> cells(nCells);
array <int, 3>* c = ...; // Lattice velocities
// ... Initialize data, start principal time loop.
for_each(execution::par_unseq, begin(cells), end(cells),
            [c](Cell const& cell)
{
    cell.rho = 0.;
    cell.u = {0., 0., 0.};
    for(int k = 0; k < 19; ++k)
{
            cell.rho += cell.f[k];
            cell.u[0] += cell.f[k] * c[k][0];
            cell.u[1] += cell.f[k] * c[k][1];
            cell.u[2] += cell.f[k] * c[k][2];
}
});
```

In this example, the data layout follows the principles of an *array-of-structure* (aos), as all variables related to a given cell are located at consecutive memory addresses. The element access function provided to the `for_each` algorithm takes in this case the form of a lambda expression (a *lambda* for short), which defines an anonymous function object. A lambda can obtain access to variables from the surrounding scope, in a manner specified in the capture clause. In this example, the clause `[c]` indicates that the variable c, which refers to the discrete velocities, is captured by value. For reasons linked to limitations of the GPU backend of Parallel Algorithms, as explained below, the `stlbm` library defines `[c]` as a raw pointer to a heap array.

Finally, the code extract makes use of a parallel version of the `for_each` algorithm. Parallel algorithms are available starting with the C++17 standard. They receive an *execution policy* as an additional argument to indicate a parallelization strategy they may use to accelerate the execution of the algorithm. The four possible strategies are

2.4.1 Sequenced.   Requires that the algorithm's execution may not be parallelized and that the calls to the element access functions must be sequenced. The result is usually the same as in a non-parallel version of the algorithm. The policy is usually invoked through `execution::seq`.

2.4.2 Parallel.   Indicates that the algorithm's execution may be parallelized using multiple threads. The policy is usually invoked through `execution::par`.

2.4.3 Unsequenced.   Provided since C++20 (but available within the C++17 standard in all tested compilers), the policy indicates that the algorithm's execution may be vectorized, by using for example instructions that operate on multiple data items. The policy is usually invoked through `execution::unseq`.

2.4.4 Parallel and unsequenced.   Indicates that the algorithm's execution may be parallelized in a multi-threaded fashion, and that vectorization is allowed within each thread. The policy is usually invoked through `execution::par_unseq`.

In this article, only the `execution::par_unseq` policy has been applied, which allowed to achieve best performance on both CPUs and GPUs.

The methodology proposed by C++ Parallel Algorithms faces a challenge on heterogeneous systems. While the memories of the host and the device are often physically distinct on such systems, the C++ standard does not provide any means to transfer data between different memories. The Parallel Algorithms backend for NVidia GPUs circumvents this problem thanks to a memory model called *CUDA Unified Memory*, which provides a single memory address space to access data from both the host and the device. Through support in both the CUDA device driver and the NVidia GPU hardware, a Unified Memory manager automatically transfers data between the two physical memories based on usage. The model is extremely convenient at the user programmer level who entirely avoids explicit data transfers, yet needs to remain aware of the cost of implicit transfers. Current versions of CUDA Unified Memory are however limited to sharing data allocated on the heap, and cannot share stack memory. On NVidia GPU systems, pointers provided to the lambda capture of a parallel algorithm, or variables captured by reference, must necessarily reference heap data. For this reason, all shared data, including the lattice constants, is allocated on the heap in the `stlbm` library.

While the above code listing efficiently parallelizes the computation of macroscopic variables (which is a fully local step), the same strategy cannot be applied directly to non-local aspects of the LB algorithm. Indeed, STL algorithms provide only limited support to express an interaction between container elements. This is done either through highly specialized algorithms, such as the parallelized sorting algorithm, or in terms of reduction operations, implemented for example in the `transform_reduce` algorithm. None of these are sufficient to tackle the two following problems that are addressed in this manuscript:

1. Implementation of a structure-of-array (soa) data layout. In this case, the populations needed for a collision are no longer consecutive in memory, and a cell can therefore no longer be treated as a single element of a C++ container.

2. Implementation of the streaming step, which requires access to nearby spatial neighbors.

STL algorithms do however not prohibit neighbor access from being implemented manually, as the full data container can be disclosed to an element access function through the capture. This strategy is further explained in the following section.

2.5 Parallel LB algorithm

The code extract of the previous section uses an anonymous lambda to express the element access function of an STL algorithm, following standard practices in simple STL usage scenarios. This strategy however forces the access function to be implemented at its point of usage, which strongly limits the possibility to properly organize a larger code base. In the following, function objects are therefore instantiated from named classes, both to encourage the reuse of the proposed code in other projects, and to avoid misunderstandings linked to the implicit syntax of lambda capture clauses.

The encapsulation of element access functions in named classes opens the possibility for their instantiation (the function object) to be named and obtain an extended lifetime. It is even possible for the data such as the LB populations to be owned and managed by the same object, in which case the LB data and algorithms are unified under the same abstraction. In the present manuscript, we prefer however to treat these objects as stateless algorithms, and to manage data in a different scope. The purpose of this approach is again the ease of reuse of the proposed codes which, in a considered frequent use scenario, should be applied to accelerate

selected parts of an existing code with a pre-existing memory management strategy. This LB algorithm class adopts the following canonical shape:

```
struct LBM {                                          // (1)
      using CellData = array <double, 19>; // (2)
      CellData*lattice;                               // (3)
      array <int, 3>* c; // lattice velocities   // (4)
      double* t; // lattice weights
      double omega; // relaxation parameter
      Dim dim; // nx x ny x nz dimensions
      void operator () (CellData& cell) {             // (5)
            // Implement collision-streaming for a cell.
      }
};
```

The enumerated lines require the following clarifications:

(1). The class is declared with the `struct` keyword to provide public access to all data and methods, as data encapsulation or any other elements of object-oriented programming are outside the scope of the presented work.

(2). `CellData` names the type of a single data element provided to the `for_each` algorithm, which must be chosen according to the data layout. Here, the example of an aos layout with consequent cell data is chosen.

(3). The class variables of the five following lines are the articulated counterpart of a lambda capture. A pointer to the full lattice is captured to allow access to the neighboring cells. In the case of the soa data alignment, the full lattice is also required to gather all populations of the current cell.

(4). We chose to capture the lattice constants (velocities and weights) by reference and the relaxation time and lattice dimensions by value.

(5). To turn class instances into a function object, the function call operator is overloaded to implement a collision-streaming cycle for a single mesh grid cell.

A canonical code for the application of an `LBM` instance to an LB lattice is expressed as follows:

```
vector <LBM::CellData> lattice_vect(num_elements); //  (1)
LBM::CellData* lattice = &lattice_container[0];      //  (2)
LBM lbm{lattice, &c[0], &t[0], omega, {nx, ny, nz}}; //  (3)
for_each (execution::par_unseq,
          lattice, lattice + num_elements, lbm);
```

This extract is again commented at the enumerated lines:

(1). The number of elements equals the number of cells (aos) or 19 times the number of cells (soa).

(2). The vector is simply used here for automatic management of heap data. The data can also be allocated with the `new` operator and deleted manually.

(3). Just as in a lambda capture, any variable visible in the present scope can be provided to initialize a `LBM` object.

Independently of the memory layout strategy (aos or soa), the data container `lattice_vect` is required to allocate a number of floating point variables equal to 19

times the number of cells. Differences only appear in their usage pattern. In the algorithmically simpler case of aos, these variables are regrouped by cells, as the type `CellData` refers to sequences of 19 floating point variables. The `for_each` algorithm is then simply applied to the full data space, cell by cell. In the soa layout on the other hand, the cell data is non-contiguous in memory, and the `CellData` type is chosen to refer to a single floating-point variable. The `for_each` algorithm can consequently not be applied to the full data space – as this would lead to too frequent invocations of a cell-level collision-streaming cycle– but spans only over the elements of the first population f_0. It is the responsibility of the methods of `LBM` to compute the indices of the other populations inside the array `lattice` and access them manually. Similarly, indices are computed to access neighboring cells for the streaming step.

The index of the currently processed cell is not explicitly provided by the `for_each` algorithm to its element access function. While the index could be reconstructed in a non-parallel version of `for_each` by giving up the stateless nature of `LBM` and incrementing an internal index at each function call, such an approach would be in obvious violation with a multithreaded or unsequenced execution of the algorithm. Instead, we choose to rely explicitly on the fact that the iterators in this invocation of `for_each` are raw pointers, and deduce the index from the difference of memory address between the current and first element of the lattice. While it is clear that such an approach could not be extended to other types of C++ containers and iterators, it should also be mentioned that this limitation to the scope of raw data arrays is rather expected in the context of massive HPC. It is further pointed out that the proposed code preserves a consistent high-level structure. Violations of the safety of data types and memory bounds are limited to a small group of data access function which encapsulate the cases of pointer-based arithmetic operations. Consequently, the data types and data access functions are defined as follows in a array-of-structure respectively a structure-of-array layout:

```
// Implementation for array-of-structure layout
struct LBM {
    using CellData = array<double, 19>;
    double & f (int i, int k) {
        return lattice[i][k];
    }
    // ... implementation of collision models
};
// Implementation for structure-of-array layout
struct LBM {
    using CellData =double;
    double & f (int i, int k) {
        return lattice[k * dim.nelem + i];
    }
    // ... implementation of collision models
};
```

Based on these access functions, the collision models can then be defined in an identical manner for both data layouts. Furthermore, the strategy for evaluating the index value and manipulating 3D indices is layout independent:

```
struct LBM {
        // ... data definitions
        // Function i_to_xyz transforms a linear index
        // into a Cartesian coordinate triplet
        auto i_to_xyz (int i) {
```

```
                int iX = i / (dim.ny * dim.nz);
                int i_yz = i % (dim.ny * dim.nz);
                int iY = i_yz / dim.nz;
                int iZ = i_yz % dim.nz;
                return std::make_tuple(iX, iY, iZ);
            };
            void operator() (CellData& cell)
                {size_t i = &cell-lattice;
                // ... implementation of collision-streaming
            }
    }
```

The complete source code of the `stlbm` can be accessed under the terms of an open-source license under S1 Source Code.

2.6 Minimization of number of instructions

Two major strategies for improving the efficiency of an HPC code consist in (1) optimizing memory accesses (which we do by testing six different data access strategies), and (2) reducing the number of instructions executed by the code. In LB applications, the latter is typically achieved by unrolling the loops over populations, and regrouping and eliminating terms in the resulting arithmetic expressions. In the present work, two versions of the second-order BGK (BGK-W2) and TRT collision models were implemented. The first, referred to as the *educational* version, uses only few superficial optimizations and relies on the capabilities of the compiler for further in-depth improvements, in order to encourage readability and re-use of the code. The second, referred to as the *unrolled* version uses aggressive loop unrolling and regrouping of terms to achieve manual performance gains. Interestingly, while the unrolled code versions systematically yield substantially better performance than the educational ones on GPUs, the flagship AMD CPU used in the tests obtained better results with the educational code for the BGK model.

In the educational code versions, loop unrolling was applied to the computation of the macroscopic variable density and velocity, to eliminate the frequent multiplications with zero-valued components of the discrete velocities c_i. Furthermore, partial sums that were shared by the expressions for density and velocity were regrouped. While the loops for the computation of equilibrium and for executing the collision models were not unrolled, the similarity of the equilibrium term for opposite populations was explicitly exploited. For example, as the second-order weighted equilibrium (BGK-W2) for a direction k is written as

$$E_k = \rho\, t_k (1 + 3\, \mathbf{c}_k \cdot \mathbf{u} + 4.5\, (\mathbf{c}_k \cdot \mathbf{u})^2 - |\mathbf{u}|^2), \tag{6}$$

the equilibrium value in the opposite direction $\bar{k}$, defined by the relation $\mathbf{c}_{\bar{k}} = -\mathbf{c}_k$, can be computed from E_k through the equation

$$E_{\bar{k}} = E_k - 6\, \rho\, t_k\, \mathbf{c}_k \cdot \mathbf{u}. \tag{7}$$

In the unrolled code, all loops in the collision term were additionally unrolled, and redundant terms eliminated.

Regarding more complex collision models, their implementation relies on the computation of post-collision moments of interest, which are converted back to raw moments (RMs) in order to compute post-collision populations f_k^{out}. This methodology allows for a drastic reduction of the number of instructions since it naturally (1) discards multiplication by zero-valued terms, and (2) highlights existing symmetries between opposite post-collision populations. By

taking advantage of the latter property, one can reduce the number of instructions required to compute $f_{\bar{k}}^{out}$ by first computing f_k^{out} –as already done for $E_{\bar{k}}$ in Eq (7). For instance,

$$f_0^{out} = \frac{\rho}{2}(-u_x + M_{200}^* + M_{120}^* + M_{102}^* - M_{220}^* - M_{202}^*), \tag{8}$$

$$f_{10}^{out} = \frac{\rho}{2}(u_x + M_{200}^* - M_{120}^* - M_{102}^* - M_{220}^* - M_{202}^*), \tag{9}$$

with $\boldsymbol{c}_0 = (-1, 0, 0) = -\boldsymbol{c}_{10}$ based on the velocity ordering convention used in the accompanying codes. Hence,

$$f_0^{out} = f_{10}^{out} - \rho(u_x - M_{120}^* - M_{102}^*), \tag{10}$$

where M_{pqr}^* $(p, q, r \in \mathbb{N})$ are post-collision RMs defined as

$$M_{pqr}^* = (1 - \omega_{pqr})M_{pqr} + \omega_{pqr}M_{pqr}^{eq}. \tag{11}$$

ω_{pqr} are adjustable relaxation frequencies, and M_{pqr}^{eq} only depend on ρ and $\boldsymbol{u}$ (their full expression can be found for all types of moment, e.g., in Ref. [36]). In addition, the computation of RMs

$$M_{pqr} = \sum_k c_{kx}^p c_{ky}^q c_{kz}^r f_k, \tag{12}$$

as well as other types of moments, is of paramount importance since it drastically impacts the performance of the code. As a matter of fact, the performance can be divided by a factor two or three if their general formulation is used. Consequently, aggressive optimizations (loop unrolling and regrouping of terms) were also applied to increase the performance of these collision models. For all models other than BGK-W2 and TRT, no "educational" version was produced.

As a sidenote, the multi-relaxation-time formulations considered in this study could be further optimized by limiting their generality in part, e.g. by imposing $\omega_{pqr} = 1$ in Eq (11). The latter value is commonly used in the LB community for high-order $(p + q + r \geq 3)$ post-collision moments, as well as those related to bulk viscosity, as it leads to an increased stability for high-Reynolds number flow simulations and/or in under-resolved conditions. Such a choice would allow us to discard the computation of the corresponding moment (12) which remains time consuming despite our optimizations. More quantitative investigations regarding performance gains induced by the equilibration of high-order moments (and those related to the bulk viscosity) will be discussed in a future study. The interested reader may also refer to Refs. [39, 63] for a brief discussion of this aspect in the context of multiphysics flow simulations based on CM-LBMs.

Finally, an extra parameter has been added to all multi-relaxation-time formulations, so that the bulk viscosity can be freely adjusted. This is done by modifying the relaxation of second-order moments as follows [36, 64]

$$M_{200}^* = M_{200} - \omega_+(M_{200}^{eq} - M_{200}^{eq}) - \omega_-(M_{020}^{eq} - M_{020}^{eq}) - \omega_-(M_{002}^{eq} - M_{002}^{eq}), \tag{13}$$

$$M_{020}^* = M_{020} - \omega_-(M_{200}^{eq} - M_{200}^{eq}) - \omega_+(M_{020}^{eq} - M_{020}^{eq}) - \omega_-(M_{002}^{eq} - M_{002}^{eq}), \tag{14}$$

$$M_{002}^* = M_{002} - \omega_-(M_{200}^{eq} - M_{200}^{eq}) - \omega_-(M_{020}^{eq} - M_{020}^{eq}) - \omega_+(M_{002}^{eq} - M_{002}^{eq}), \tag{15}$$

where $\omega_- = (\omega_{v_b} - \omega_v)/3$ and $\omega_+ = \omega_- + \omega_v$. The kinematic viscosity v, and its bulk

counterpart v_b, are related to these relaxation parameters through $v = (1/\omega_v - 0.5)c_s^2$ and $v_b = (1/\omega_{v_b} - 0.5)c_s^2$. The lattice constant c_s is defined as $c_s = 1/\sqrt{3}$ [3].

2.7 Hardware platforms and compilers

The computers used to carry out the benchmark simulations are listed below, along with the compiler and compiler options that were used.

AMD Ryzen Threadripper (8 cores)

Model	AMD Ryzen Threadripper 1900X 8−Core Processor
Cores	8 Cores, 16 Threads (SMT on), 1 Socket
Compiler	`CLang 10.0.0 − 4` / `−std=c++17 −O3`

Intel Xeon (8 cores)

Model	Xeon E5 − 2637 v4 @ 3.50GHz
Cores	8 Cores, 8 Threads (Hyperthreading off), 2 Sockets
Compiler	`GCC/g++ 9.3.0` / `−std=c++17 −O3 −march=native`

Intel Xeon (48 cores)

Model	Intel Xeon Gold 6240R @ 2.40GHz
Cores	48 Cores, 48 Threads (Hyperthreading off), 2 Sockets
Compiler	`GCC/g++ 9.3.0` / `−std=c++17 −O3 −march=native`

AMD EPYC (64 cores)

Model	AMD EPYC 7742 64−Core Processor @ 2.24 GHz
Cores	64 Cores, 64 Threads (SMT off), 1 Socket
Compiler	`CLang 10.0.1` / `−std=c++17 −O3`

NVIDIA GTX 1080 Ti

Model	GP102 [GeForce GTX 1080 Ti]
Compiler	`nvc++ 20.7–0` / `−stdpar −O1 −std=c++17`

NVIDIA RTX 2080 Ti

Model	TU102 [GeForce RTX 2080 Ti]
Compiler	`nvc++ 20.7–0` / `−stdpar −O1 −std=c++17`

NVIDIA V100

Model	GV100 [Tesla V100−PCIE−32GB]
Compiler	`nvc++ 20.7–0` / `−stdpar −O1 −std=c++17`

The compiler and compiler options were selected in informal comparative tests designed to identify the most efficient candidates. Except for the GPU versions compiled with `nvc++`, all binaries were linked with the `−ltbb` option to link with Intel's Threading Building Blocks.

3 Results and discussion

3.1 Benchmark case and code validation

The code of this article is applied to the benchmark case of a 3D flow in a lid-driven, cubic cavity, under the conditions of a steady, a laminar unsteady, and a turbulent regime. This test is
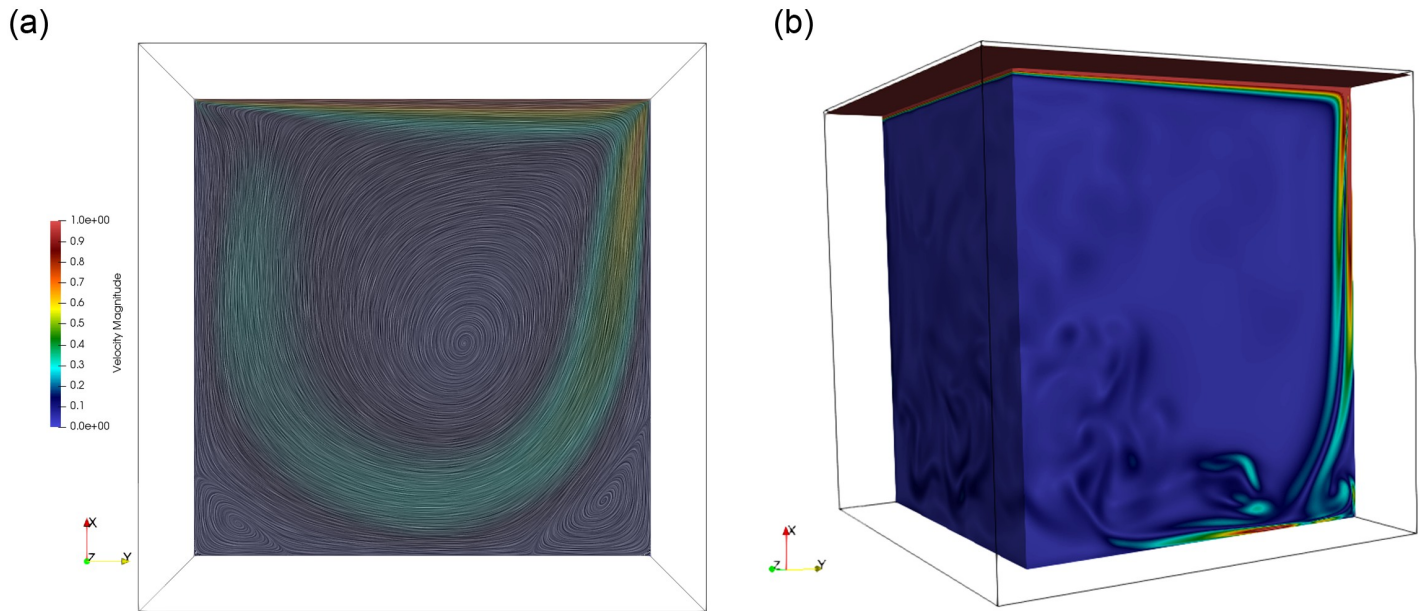
(a)   (b)

Fig 1. Visualization of simulation results for the lid-driven cavity. Left image: velocity norm and streamlines of the steady-state results at Re = 1000, on the plane defined by $z = 0$. Right image: vorticity norm on a selected surface at an instantaneous time step for the fully developed flow at Re = 10000.

used both to demonstrate the correctness of the code and to explore its efficiency on different hardware platforms depending on the implementation strategy and the LB collision model. This benchmark case, which can be found in numerous articles (see e.g. [67]), is defined as follows. An incompressible, non-thermal, viscous, Newtonian, and homogeneous fluid, the dynamics of which is described by the Navier-Stokes equations, is enclosed in a cubic domain. The 3D fluid-domain is defined by the open interval $(] - L/2, + L/2[)^3$, and the boundaries consist of the closure of this domain. The velocity vector has three components $\boldsymbol{u} = (u_x, u_y, u_z)$, and the space positions are given by $\boldsymbol{x} = x, y, z$. A Dirichlet boundary condition for the velocity is enforced on the boundaries, consisting of a no-slip condition on all walls except on the moving lid at $x = + L/2$. The velocity on the lid is constant with a value $\boldsymbol{u}(x = + L/2, y, z) = (0, U, 0)$. It should be pointed that this problem is mathematically not strictly well defined given the velocity discontinuities in the edges and corners of the lid. For this reason, a smoothly increasing profile is sometimes defined in these regions. This issue is however ignored in the present study because, as the subsequent results show, the LBM copes with this discontinuity without problem. The Reynolds number is defined as Re = UL/v. For illustrative purposes, the flow field obtained by our solver in a steady and in a turbulent regime is shown in Fig 1.

The problem is numerically solved with a homogeneous mesh of $N \times N \times N$ nodes, which includes a wall node on all extremities of the domain. Thus, the cell spacing is $\delta x = L/(N - 2)$. The time discretization was fixed by setting the reference lattice velocity $u_{\mathrm{LB}} \equiv U(\delta t/\delta x)$ to a constant value of 0.06. The Mach number, which in the present context of incompressible flow has a purely numerical significance, is therefore of Ma $= 0.06\sqrt{3} \approx 0.1$. The discrete time step is consequently defined as $\delta t = (u_{\mathrm{LB}}/U)\delta x$. The dimensionless time elapsed after N_{iter} iterations is $t = N_{\mathrm{iter}} \delta t(U/L) = N_{\mathrm{iter}} t_c$, with the dimensionless characteristic time being $t_c = \delta t(U/L) = u_{\mathrm{LB}}/(N - 2)$.

In spite of the simple setup, this problem poses some challenges due to the velocity discontinuity between the top lid and lateral walls, and because of the complexity of the flow field and boundary layers that develop at higher Reynolds numbers. This case is sometimes argued to
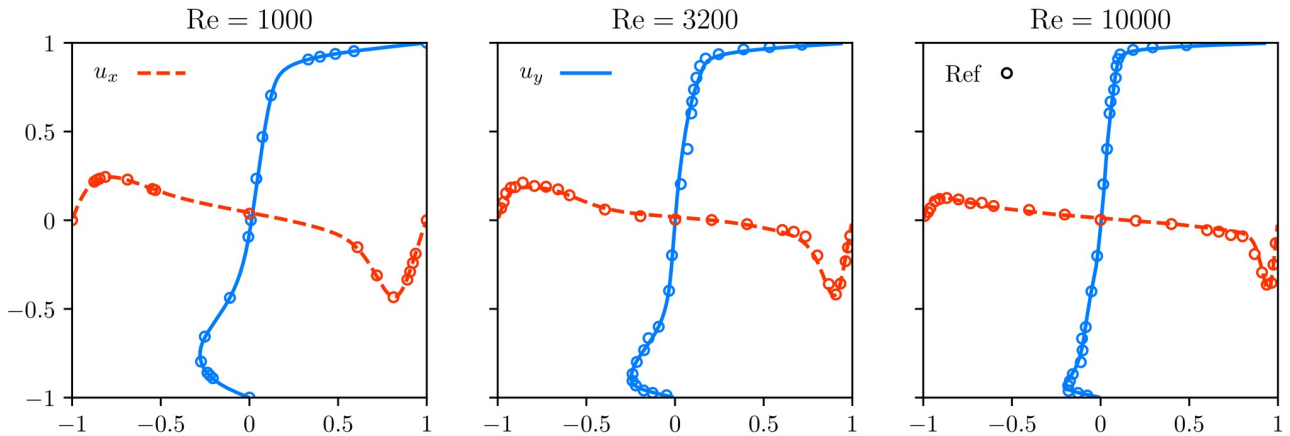
Fig 2. Profile of cubic cavity flow for two Reynolds numbers. Simulation results (solid lines) in a cubic lid-driven cavity along two center lines. Left image: Re = 1000, resolution $N = 256$, collision model BGK-W2, comparison of the results with reference values (circles) from spectral simulations provided in [67]. Middle image: non-stationary laminar flow at Re = 3200 simulated with a resolution $N = 256$ and collision model BGK-W2, averaged between a dimensionless time $t = 50$ and $t = 250$. Right image: turbulent flow at Re = 10000 simulated with a resolution $N = 400$ and RR collision model, averaged between a dimensionless time $t = 150$ and $t = 500$. The curves of the middle and right image are compared with experimental data extracted from figures in [66].

https://doi.org/10.1371/journal.pone.0250306.g002

pose substantial issues of accuracy and numerical stability for the LB method, which are for example overcome in [65] with help of a sophisticated boundary condition scheme. The results in this section show however that the LB framework of the `stlbm` project is sufficient to achieve satisfying results in all regimes, using either the standard BGK-W2 or the RR collision model.

Fig 2 plots the y-component of the velocity u_y along the center line $y = z = 0$ against the non-dimensionalized coordinate $2x/L$, and the x-component of the velocity u_x along the center line $x = z = 0$ against the coordinate $2y/L$. On the left image, at Re = 1000, the flow is steady. The flow is measured at an instantaneous time step after reaching stationary state and compared against simulation data of a spectral code. On the middle image, at Re = 3200 the flow is laminar and unsteady. The profile is averaged between dimensionless time values of $t = 50$ and $t = 250$ and is compared against experimental data published in [66]. On the right image, at Re = 10000 the flow is turbulent. The profile is averaged between dimensionless time values of $t = 150$ and $t = 500$ and is again compared with [66]. Excellent agreements are achieved in all three cases.

3.2 Parallel performance on different platforms

The case of a lid-driven cavity has been executed on different platforms to compare the performances. The `for_each` loops over the cells were executed with the `par_unseq` execution policy. Furthermore, reported values correspond to a median computed over 30 measurements, the latter being distributed other three days to reduce the impact on the benchmark of potential special events in the tested hardware or operating system. Each measurement consisted of 1000 warm-up iterations without time measurement, followed by 1000 benchmarked time steps. Finally, all test cases were first executed with a BGK-W2 collision model at a resolution of $N = 128$. A comparative study of the impact of the collision model on performances is also proposed in Section 3.3.2. As conventional in the field of LB, the performance is measured in terms of million lattice node updates per second (MLups), a value that identifies the number of lattice nodes, divided by one million, that are driven through a collision-streaming cycle in a second. To transpose this measurement in terms of time required to update one grid point
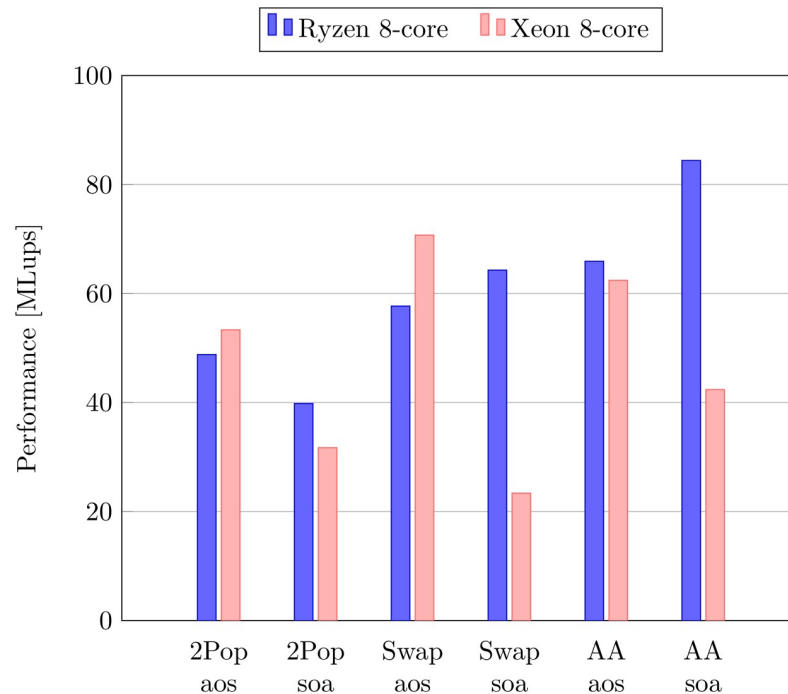
Fig 3. Performance of CPUs with a modest core count. All simulations are executed with the BGK-W2 collision model and $N = 128$. On both CPUs, best performance was quite unpredictably obtained sometimes with and sometimes without loop unrolling. The plotted value shows therefore always the best of the two.

per iteration (which is more commonly used in the Computational Fluid Dynamics community [68]), one simply needs to take the inverse of the number of lattice node update per second. As an example, a performance of 10 MLups corresponds to 0.1 *μs/pt/ite*.

The benchmark test was first executed on conventional CPUs with a modest core count. Fig 3 shows the performance obtained using the six different data structures on a 8-core AMD and a 8-core Intel CPU. The results should not be used to set up a competitive comparison between the involved CPU models, which differ in the year of market release, the clock rate, and other parameters. All in all, the results show overall comparable performance across models. The Intel CPU yields consistently best performances with an aos layout, while best performances on the AMD CPU are most frequently obtained with the soa approach, in agreement with the many-core AMD and the GPU architectures presented further down. This difference is mostly explained by the fact that the multi-core scaling performance of the code depends on the CPU and on the data structure. As shown below, the aos implementation strategy has better single-core performance than soa across architectures, but loses its edge over soa at a larger core count. While the AMD CPU reaches its best results with the AA-pattern, it is interesting to note that the swap-aos approach is the optimal choice on the Intel CPU, in spite of its apparent disadvantage due to the separated collision and streaming steps.

In Fig 4, the performances obtained on five many-core platforms are compared, namely a 64-core AMD EPYC, a 48-core Intel Xeon, two consumer-class GPUs and a data-center GPU. It is not typical to include GPUs and CPUs on a same comparison plot, as CPUs are often shown to perform one or two orders of magnitude worse than GPUs. In this case, the common comparison is motivated by the fact that they are measured with the exact identical code, but also by the fact that in our comparison, and if the best performing implementation scheme is selected for each platform, the faster of the two consumer-class GPUs outpaces the CPUs by
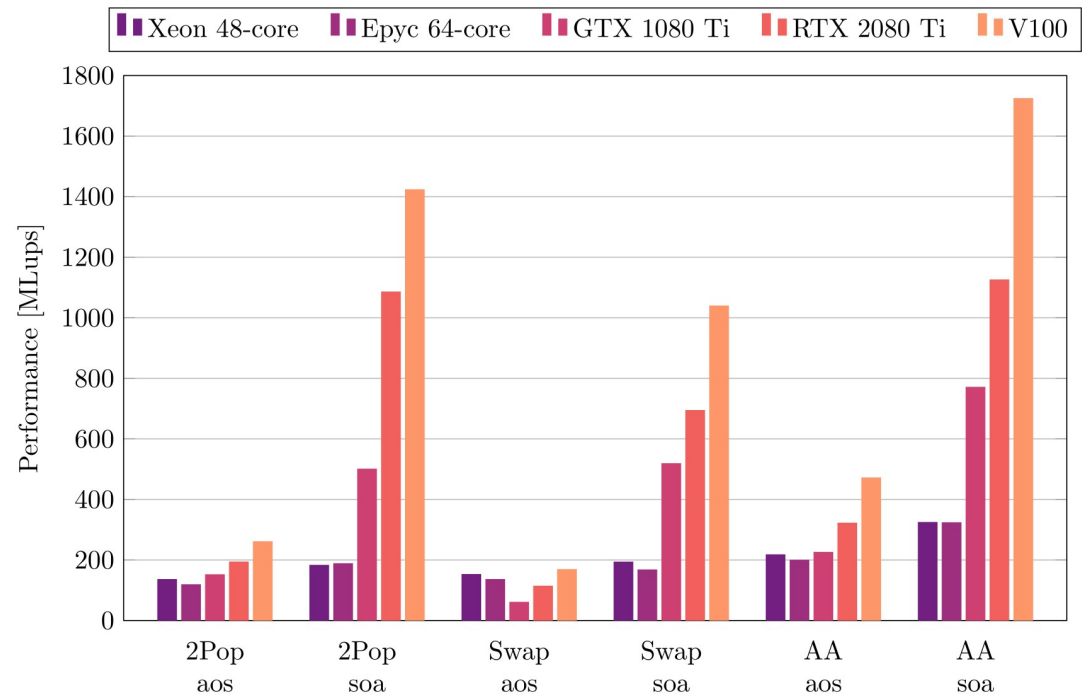
Fig 4. Performance of many-core CPU and GPU architectures. All simulations are executed with BGK-W2 collision model and $N = 128$. For best overall performance, the simulations on the 64-core Epyc CPU were executed without loop unrolling, while the 48-core Intel CPU and the two GPUs used loop unrolling.

https://doi.org/10.1371/journal.pone.0250306.g004

only roughly a factor four. To achieve best performance, the loops were unrolled on the GPUs and on the Intel Xeon, but were used without unrolling on the AMD CPU. Best performances are systematically obtained with the soa data layout, and the AA-pattern is the clear winner across all CPU and GPU platforms. For this reason, and also because of its lower memory needs, the AA-pattern (combined with the soa data layout) can be considered the most generally well performing choice for many-core platforms.

The real life implications of the performance metrics of the code are illustrated for the case of the turbulent cavity flow, which is run with the RR collision model at $N = 400$ up to a dimensionless time of $t = 500$ (to gather sufficient statistics), thus requiring a total of 3.3 million iterations. The simulation does not make use of any subgrid-scale turbulence model, but it is stabilized by adjusting the relaxation parameter associated to the bulk viscosity to a value of 1. On the 48-core Intel Xeon CPU, the execution time without outputs is of 7 days and 15 hours, while the RTX 2080 Ti completes the same task in 2 day and 15 hours, and the V100 PCIE in just 1 day and 3 hours.

Table 2 shows the numerical values of the performances obtained on the two CPUs with the aos and the soa implementation. The performances obtained if the program uses only a single core are also shown. It can be seen that on both CPUs, the aos implementation is more performant on a single core, but the soa approach is preferable when all cores are exploited. It can be hypothesized that although in general, the aos approach makes better use of memory caching mechanisms, this advantage is lost when multiple cores compete for access to shared memory space, and cache coherence must be guaranteed. It is further remarked that both CPUs exhibit a comparable parallel efficiency, and that the Intel CPU compensates the lower core count with a stronger single-core performance.

Table 2. Performance and speedup of AMD and Intel CPU with large core counts. Detailed performance figures (in MLups) are provided for the cavity benchmark case with N = 128, BGK collision, and AA-pattern implementation, for the 64-core AMD EPYC CPU and the 48-core Intel Xeon CPU, for program executions using 1 core or all cores. The speedup is the ratio between these two values, and the efficiency is a measure of the scaling performance (100% efficiency stands for an ideal speedup).

	Performance of 1 core	Performance of full CPU	Speedup (Efficiency)
64-core Epyc / AA aos	11.68	199.0	17.0 (26.6%)
64-core Epyc / AA soa	7.42	322.7	43.5 (68.0%)
48-core Xeon / AA aos	12.32	216.9	17.6 (31.9%)
48-core Xeon / AA soa	10.13	323.7	36.7 (66.6%)

Table 3 provides the numerical values of the performances obtained on the two GPUs with the double-population implementation and the AA-pattern and compares them with a reference performance obtained on the same GPUs with a code extracted from the CuBoltz code, an open-source library that evolved from the work described in [69]. Written in the Cuda language, this code is written by one of the authors of the article and is carefully optimized for performance. The extracted code has been reshaped to solve the exact same problem as the present `stlbm` tests, as verified through comparative regression tests. The comparison shows that the generic `stlbm` code ends up only 21% (GTX 1080 Ti) respectively 17% (RTX 2080 Ti) below the performance of a hardware-specific, carefully optimized code. It can be concluded that the formalism of C++ Parallel Algorithms allows to achieve performances adequate for high performance computing, while at the same time offering an elegant and generic programming style.

On GPUs, it is useful to compare these performance values against a theoretical peak performance obtained under the assumption that computations are negligible and the overall performance is limited solely by optimally coalesced accesses to central memory. With the two-population scheme and the AA-pattern, all populations need to be read and written once per iteration, which in double-precision representation requires two memory transactions of an 8-byte floating-point number per population and iteration (with the Swap algorithm, the number of memory transactions is doubled). For these two implementation schemes, the peak performance p (measured in LUPS) is therefore established by the formula

$$p = bw/(2 \cdot 19 \cdot 8), \tag{16}$$

where bw is the memory bandwidth (in bytes per second). On Table 3, the ratio of measured performance over the theoretical peak performance is indicated in parentheses. It can be seen that the `stlbm` code reaches performances close to, and in some cases even distinctly superior to, 50% of the theoretical peak performance that can be expected on the respective GPUs. It is worth noting that these performances are obtained for lightly loaded GPUs, i.e., N = 128. By further increasing the load to N = 400, e.g., for the data-center GPU V100 PCIE, the ratio of measured performance is increased to 84%, 63% and 72% for CuBoltz, the two-population

Table 3. Performance values of `stlbm` and the reference Cuda code CuBoltz for the lid-driven cavity at N = 128, with the BGK-W2 collision model. Performance is indicated in MLups, followed in parenthesis by the ratio between this performance and the theoretical peak performance, computed for a memory bandwidth of 480 GB/s (GTX 1080 Ti), 616 GB/s (RTX 2080 Ti), and 900 GB/s (V100 PCIE).

	CuBoltz	`stlbm` (2Pop)	`stlbm` (AA)
GTX 1080 Ti	877.7 (56%)	499.7 (32%)	770.1 (49%)
RTX 2080 Ti	1415.8 (70%)	1084.9 (54%)	1124.8 (56%)
V100 PCIE	1823.6 (62%)	1422.6 (48%)	1723.7 (58%)

scheme and the AA pattern respectively. In any case, this study demonstrates both the quality of the `stlbm algorithm`, and the suitability of LB codes for GPU platforms.

3.3 Analysis of performance-impacting features

3.3.1 Loop unrolling. Fig 5 compares the performance of the educational and the unrolled codes on the five tested many-core platforms for the BGK-W2 collision model. The results show that the GPUs experience substantial speed improvements from loop unrolling (the improvement is of 72%, 66%, and 34% on the GTX 1080 Ti, the RTX 2080 Ti, and the V100 PCIE respectively), while the AMD EPYC performs better with the educational code, which presumably allows more targeted compiler optimizations. Interestingly, when the educational code is used, and thus, readability is emphasized over hand-tuned optimizations, the consumer-class GPUs seem to lose their clear-cut performance edge over CPUs, as they are only 39% (GTX 1080 Ti) respectively 110% (RTX 2080 Ti) faster than the AMD CPU. The data-center GPU V100 PCIE on the other hand remains substantially faster even for this version of the code, with a speed up of 298% over the AMD.

3.3.2 Collision model. Hereafter, the performance obtained with different LB collision models are investigated on the fastest consumer-level platform, the RTX 2080 Ti GPU. In all cases, the optimized codes with loop unrolling were used, and multi-relaxation-time formulations of collision models were considered.

Generally speaking, Fig 6 shows that the speed difference between the slowest and the fastest collision model remains within a factor two. This means that adopting modern collision models, e.g. to simulate high-Reynolds number flows in a stable and accurate manner, does not come at the expense of a drastic performance loss. Hence, LBMs based on these collision models remain particularly competitive as compared to Navier-Stokes solvers. Quantitatively
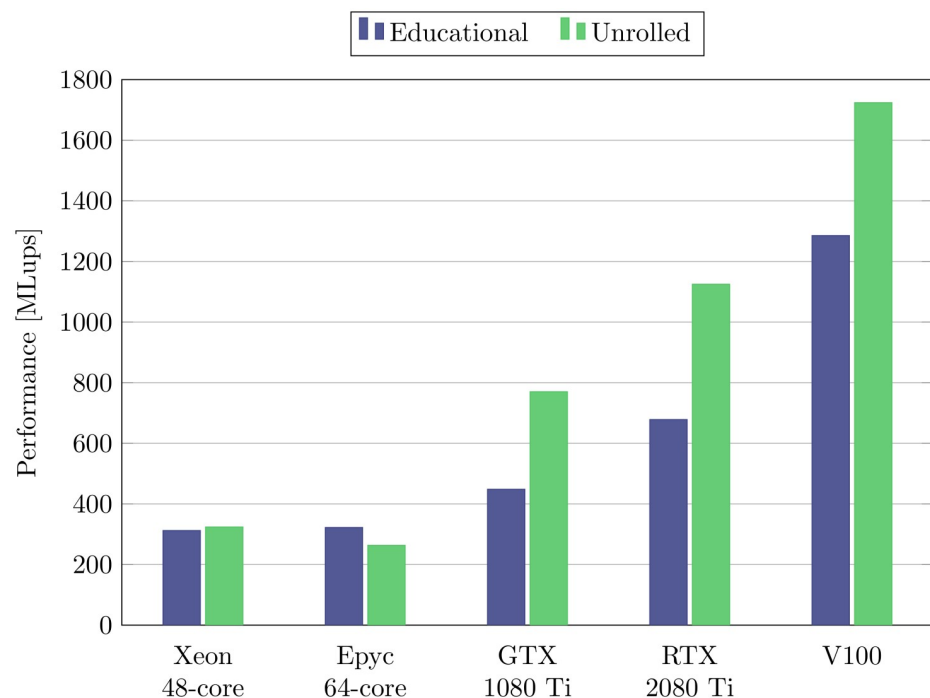

Fig 5. Influence of loop unrolling. All simulations are executed with the BGK-W2 collision model, with the AA-soa scheme, and with $N = 128$.
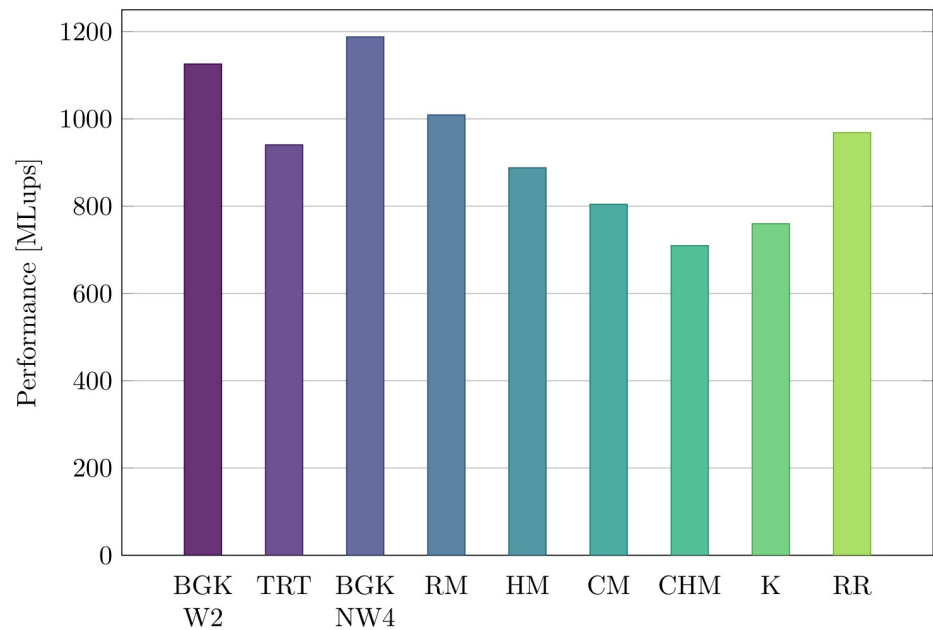
Fig 6. Performance of different LB models. All simulations are executed with the AA-soa data structure at $N = 128$ on the RTX 2080 Ti GPU.

speaking, single-relaxation-time formulations (BGK-W2 and BGK-NW4) are the fastest. For the latter model, more redundant terms can be pre-computed, hence its better performance. Regarding collision models based on multiple relaxation times, it seems clear that increasing the number of moment space transitions automatically increases the number of floating point operations, hence a decrease of performance. As an example, the CHM-LBM requires two intermediate moment transformations before computing post-collision populations from their RMs, i.e. transformations from CHMs to HMs and from HMs to RMs. The same goes for the K-LBM, whereas the HM-LBM only requires a single moment transformation (from HMs to RMs). In addition, even if the RR-LBM is also based on a collision applied in the HM space, it is faster than the HM-LBM. This is explained by the fact that it only requires the computation of second-order HMs, and recursive computations of high-order HMs only add a few floating point operations.

In the end, the present study confirms that the good performances obtained with the `stlbm` implementation strategy are consistent across all collision models.

3.3.3 Domain size. Fig 7 shows the performance achieved by `stlbm` on the fastest consumer-level hardware platform, the RTX 2080 Ti GPU, using different domain sizes, with increments of 1 for the number of cells N in every direction. The performance figures are shown for the AA-soa and double-population-soa implementation strategies, and for the BGK-W2 and RR collision models, to test both a simple and a complex model. In general, the performance increases up to a resolution of roughly $100 \times 100 \times 100$, and stays roughly constant afterwards. The double-population implementation further shows a drop of performance close to $N = 90$, which is not further explained at this point. This investigation finally leads to the conclusion that the performances of the double-population scheme and the AA-pattern are very similar, and that the slight superiority of the double-population approach observed in Sec 3.2 would be less clear-cut or even inexistent at a different domain size or with a different collision model.
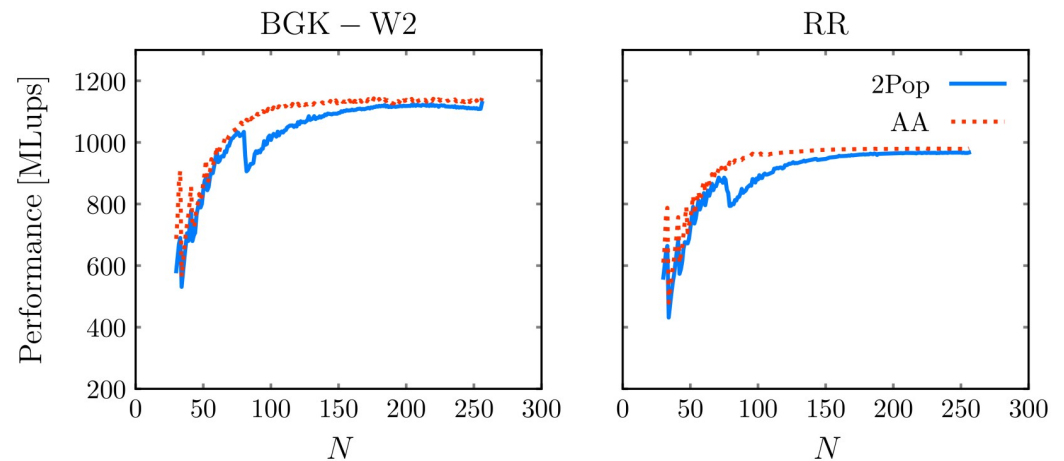
Fig 7. Dependence of domain size on performance on GPU. Left: BGK-W2; Right: RR. Simulations are run on an RTX 2080 Ti GPU.

4 Conclusion

This article presents an implementation approach for multi-threaded LB applications which is hardware agnostic, relies exclusively on standard C++ language features, yet exhibits very interesting performances on many-core CPUs and on GPUs. It is presented and disseminated in form of the open-source C++ library `stlbm`. While the computational kernels are kept very simple to serve as templates in general-purpose LB codes, and use a descriptive programming style to serve an educational purpose, the `stlbm` library targets a broad scope, as it offers six different implementation strategies, and nine LB collision models, which can be combined without restrictions.

The article presents performance metrics of the code for different implementation choices and on different platforms for the test case of a 3D lid-driven cavity flow. The results of these benchmarks show that the proposed framework leads to excellent performance on both CPUs and GPUs. In the latter case, very similar performances are obtained compared to those of an optimized code written in a domain-specific language, with a performance loss of only approximately 20%.

The results of these benchmarks also lead to an interesting conclusion that appears to question common prejudice regarding the use of CPUs vs. GPUs in computational science. While it is traditionally thought that GPU development is substantially more time consuming than CPU development, but more rewarding thanks to performance gains of one to two orders of magnitude, the `stlbm` library presented in this article paints rather the opposite picture. Indeed, the same effort leads to a code that compiles to a CPU and a GPU binary without changes. As for the performance, the two consumer-level GPUs (which, like the CPUs, were deployed within desktop computers) outperform the CPUs by roughly a factor 4 (or a factor 2 if the more readable version of the code is considered), while the data-center GPU achieves a factor of more than 5 (or a factor 4 for the educational code version).

The achieved performances are barely affected by the use of sophisticated collision terms. More precisely, the speed difference between standard and more sophisticated collision models remains within a factor two, which means the most robust collision models remain competitive (despite their increased complexity) as compared to Navier-Stokes solvers.

The numerical tests also allow to single out two implementation strategies that exhibit the best performances across the tested high-performance platforms, namely the double-

population scheme and the AA-pattern, both used in combination with a structure-of-array data layout. Among these two, the AA-pattern appears to be more versatile thanks to its consistently good performances across different platforms. It furthermore stands out because of its reduced memory needs. While the third tested approach, the swap scheme, performed poorly on GPU and AMD CPU hardware, it produced good performance on the 8-core Intel CPU, combined with an array-of-structure layout. We therefore retain also this approach as an interesting candidate, which has a low memory footprint like the AA-pattern. It furthermore proposes an algorithm that fits more easily into complex LB applications, as it does not require a separation into even and odd time steps.

As a consequence of these observations, the `stlbm` library offers, additionally to its full software framework, three standalone, single-file codes based on the two-population soa, the AA-pattern soa, and the swap aos strategies respectively. Each of these standalone files contains less than 500 lines of code and allows to understand the philosophy of LB simulations with Parallel Algorithms with fewer hurdles than the full code basis. They are written for the BGK-W2 collision model, which is however easily replaced by any of the other models by inserting corresponding code portions for the `stlbm` library.

We finally point out that the conclusions regarding the best performing implementation scheme appear not to be universal, and rather linked to the specificity of the presently tested hardware. The generic version of the `stlbm` code serves therefore an important purpose, allowing all implementation strategies to be rapidly tested on future many-core systems.

Supporting information

S1 Source Code. STLBM software library. The `stlbm` software library is released as an open-source project and can be used under the terms of an MIT license. The source code is available at https://gitlab.com/UniGeHPFS/stlbm. All performance measurements presented in this article were executed with the version of the code identified through the Git tag `benchmarks_plosone`.
(TXT)

S1 File. Performance measurements. This spreadsheet contains the detailed performance measurements used to produce the median values on the graphs and in the tables of this article.
(XLSX)

Acknowledgments

We would like to thank Orestis Malaspinas for useful discussions regarding the role of functional programming styles (in relationship with Futhark) in hardware-agnostic multi-threaded program development, and for supplying testing hardware for this article. We would further like to thank Christos Kotsalos and Anthony Boulmier for providing advice and help on hardware installation and configuration.

Author Contributions

Conceptualization: Jonas Latt, Christophe Coreixas, Joël Beny.

Data curation: Jonas Latt.

Formal analysis: Jonas Latt, Christophe Coreixas.

Investigation: Jonas Latt, Christophe Coreixas, Joël Beny.

Methodology: Jonas Latt, Joël Beny.

Project administration: Jonas Latt.

Software: Jonas Latt, Christophe Coreixas, Joël Beny.

Supervision: Jonas Latt.

Validation: Jonas Latt, Christophe Coreixas.

Visualization: Jonas Latt, Christophe Coreixas.

Writing – original draft: Jonas Latt, Christophe Coreixas.

Writing – review & editing: Jonas Latt, Christophe Coreixas, Joël Beny.

References

1. Latt J, Malaspinas O, Kontaxakis D, Parmigiani A, Lagrava D, Brogi F, et al. Palabos: Parallel Lattice Boltzmann Solver. Computers & Mathematics with Applications. 2021; 81: 334–350. https://doi.org/10.1016/j.camwa.2020.03.022

2. Ladd AJC. Numerical simulations of particulate suspensions via a discretized Boltzmann equation. Part 1. Theoretical foundation. Journal of Fluid Mechanics. 1994; 271: 285–309. https://doi.org/10.1017/S0022112094001783

3. Krüger T, Kusumaatmaja H, Kuzmin A, Shardt O, Silva G, Viggen EM. The Lattice Boltzmann Method: Principles and Practice. Springer International Publishing; 2017.

4. Marson F, Thorimbert Y, Chopard B, Ginzburg I, Latt J. Enhanced single-node boundary condition for the Lattice Boltzmann Method. Physical Review E. Forthcoming. Available from: https://arxiv.org/abs/2009.04604

5. Alfieri RA. An efficient kernel-based implementation of POSIX threads. In: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference—Volume 1. USTC'94. USA: USENIX Association; 1994. p. 5.

6. NVIDIA CUDA Toolkit. Available from: https://developer.nvidia.com/cuda-toolkit

7. Fang J, Huang C, Tang T, Wang Z. Parallel programming models for heterogeneous many-cores: a comprehensive survey. CCF Trans HPC. 2020; 2: 382–400. https://doi.org/10.1007/s42514-020-00039-4

8. The OpenMP API specification for parallel programming. Available from: https://www.openmp.org/

9. The OpenACC API specification for parallel programming. Available from: https://www.openacc.org/

10. Intel's OneAPI. Available from: https://software.intel.com/en-us/oneapi

11. HCC: An open source C++ compiler for heterogeneous devices. Available from: https://github.com/RadeonOpenCompute/hcc

12. Haidl M, Gorlatch S. PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14. In: 2014 LLVM Compiler Infrastructure in HPC; 2014. p. 1–11. https://doi.org/10.1109/LLVM-HPC.2014.9

13. Bell N, Hoberock J. Thrust: A productivity-oriented library for CUDA. In: GPU computing gems Jade edition. Elsevier; 2012. p. 359–371. https://doi.org/10.1016/B978-0-12-385963-1.00026-5

14. Kim W, Voss M. Multicore Desktop Programming with Intel Threading Building Blocks. IEEE Software. 2011; 28(1): 23–31. https://doi.org/10.1109/MS.2011.12

15. Pohl T, Kowarschik M, Wilke J, Iglberger K, Rüde U. Optimization and profiling of the cache performance of parallel lattice Boltzmann codes. Parallel Process Lett. 2003; 13(04): 549–560. https://doi.org/10.1142/S0129626403001501

16. Bauer M, Eibl S, Godenschwager C, Kohl N, Kuron M, Rettinger C, et al. waLBerla: A block-structured high-performance framework for multiphysics simulations. Computers & Mathematics with Applications. 2020; 81: 471–501. https://doi.org/10.1016/j.camwa.2020.01.007

17. Mattila K, Hyväluoma J, Rossi T, Aspnäs M, Westerholm J. An efficient swap algorithm for the lattice Boltzmann method. Computer Physics Communications. 2007; 176(3): 200–210. https://doi.org/10.1016/j.cpc.2006.09.005

18. Bailey P, Myre J, Walsh SDC, Lilja DJ, Saar MO. Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors. In: 2009 International Conference on Parallel Processing. Vienna: IEEE; 2009. p. 550–557.

19. Mohrhard M, Thäter G, Bludau J, Horvat B, Krause MJ. Auto-vectorization friendly parallel lattice Boltzmann streaming scheme for direct addressing. Computers & Fluids. 2019; 181: 1–7. https://doi.org/10.1016/j.compfluid.2019.01.001

20. Ryoo S, Rodrigues CI, Baghsorkhi SS, Stone SS, Kirk DB, Hwu WmW. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. PPoPP '08. New York, NY, USA: Association for Computing Machinery; 2008. p. 73–82. https://doi.org/10.1145/1345206.1345220

21. Tölke J. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. Comput Visual Sci. 2010; 13(1): 29–39. https://doi.org/10.1007/s00791-008-0120-2

22. Kuznik F, Obrecht C, Rusaouen G, Roux JJ. LBM based flow simulation using GPU computing processor. Computers & Mathematics with Applications. 2010; 59(7): 2380–2392. https://doi.org/10.1016/j.camwa.2009.08.052

23. Obrecht C, Kuznik F, Tourancheau B, Roux JJ. A new approach to the lattice Boltzmann method for graphics processing units. Computers & Mathematics with Applications. 2011; 61(12): 3628–3638. https://doi.org/10.1016/j.camwa.2010.01.054

24. Mawson MJ, Revell AJ. Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs. Computer Physics Communications. 2014; 185(10): 2566–2574. https://doi.org/10.1016/j.cpc.2014.06.003

25. Tran NP, Lee M, Hong S. Performance Optimization of 3D Lattice Boltzmann Flow Solver on a GPU. Scientific Programming. 2017; 2017: 1–16. https://doi.org/10.1155/2017/1205892

26. McIntosh-Smith S, Curran D. Evaluation of a performance portable lattice Boltzmann code using OpenCL. In: Proceedings of the International Workshop on OpenCL 2013 & 2014—IWOCL '14. Bristol, United Kingdom: ACM Press; 2014. p. 1–12. https://doi.org/10.1145/2664666.2664668

27. Obrecht C, Tourancheau B, Kuznik F. Performance Evaluation of an OpenCL Implementation of the Lattice Boltzmann Method on the Intel Xeon Phi. Parallel Process Lett. 2015; 25(03): 1541001. https://doi.org/10.1142/S0129626415410017

28. Januszewski M, Kostur M. Sailfish: A flexible multi-GPU implementation of the lattice Boltzmann method. Computer Physics Communications. 2014; 185(9): 2350–2368. https://doi.org/10.1016/j.cpc.2014.04.018

29. Lagrava D, Malaspinas O, Latt J, Chopard B. Advances in multi-domain lattice Boltzmann grid refinement. Journal of Computational Physics. 2012; 231(14): 4808–4822. https://doi.org/10.1016/j.jcp.2012.03.015

30. Astoul T, Wissocq G, Boussuge JF, Sengissen A, Sagaut P. Analysis and reduction of spurious noise generated at grid refinement interfaces with the lattice Boltzmann method. J Comput Phys. 2020; 418: 109645. https://doi.org/10.1016/j.jcp.2020.109645

31. Succi S. The Lattice Boltzmann Equation for Fluid Dynamics and Beyond. Illustrated edition ed. Oxford: New York: Clarendon Press; 2001.

32. Chopard B. Cellular Automata Modeling of Physical Systems. In: Meyers RA, editor. Computational Complexity: Theory, Techniques, and Applications. New York, NY: Springer New York; 2012. p. 407–433. https://doi.org/10.1007/978-1-4614-1800-9_27

33. Ladd AJC. Numerical simulations of particulate suspensions via a discretized Boltzmann equation. Part 1. Theoretical foundation. J Fluid Mech. 1994; 271: 285–309. https://doi.org/10.1017/S0022112094001783

34. Ginzburg I, Verhaeghe F, d'Humières D. Two-relaxation-time Lattice Boltzmann Scheme: About Parametrization, Velocity, Pressure and Mixed Boundary Conditions. Communications in Computational Physics. 2008; 3(2): 427–478. Available from: https://www.global-sci.org/oldweb/cicp/issue/FULLPDF/3/427/paper.pdf.

35. Skordos PA. Initial and boundary conditions for the lattice Boltzmann method. Phys Rev E. 1993; 48(6): 4823–4842. https://doi.org/10.1103/PhysRevE.48.4823 PMID: 9961167

36. Coreixas C, Chopard B, Latt J. Comprehensive comparison of collision models in the lattice Boltzmann framework: Theoretical investigations. Phys Rev E. 2019; 100: 033305. https://doi.org/10.1103/PhysRevE.100.033305 PMID: 31639944

37. Coreixas C, Wissocq G, Chopard B, Latt J. Impact of collision models on the physical properties and the stability of lattice Boltzmann methods. Phil Trans R Soc A. 2020; 378(2175): 20190397. https://doi.org/10.1098/rsta.2019.0397 PMID: 32564722

38. Qian YH, D'Humières D, Lallemand P. Lattice BGK Models for Navier-Stokes Equation. Europhys Lett. 1992; 17(6): 479. https://doi.org/10.1209/0295-5075/17/6/001

39. De Rosis A, Coreixas C. Multiphysics flow simulations using D3Q19 lattice Boltzmann methods based on central moments. Phys Fluids. 2020; 32: 117101. https://doi.org/10.1063/5.0026316

40. d'Humières D, Ginzburg I. Viscosity independent numerical errors for Lattice Boltzmann models: From recurrence equations to "magic" collision numbers. Comput Math Appl. 2009; 58(5): 823–840. https://doi.org/10.1016/j.camwa.2009.02.008

41. d'Humières D. Generalized lattice-Boltzmann equations. Prog Astronaut Aeronaut. 1992; 159: 450–458. https://doi.org/10.2514/5.9781600866319.0450.0458

42. Lallemand P, Luo LS. Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. Phys Rev E. 2000; 61: 6546–6562. https://doi.org/10.1103/PhysRevE.61.6546 PMID: 11088335

43. d'Humières D. Multiple-relaxation-time lattice Boltzmann models in three dimensions. Philos Trans R Soc London, Ser A. 2002; 360(1792): 437–451. https://doi.org/10.1098/rsta.2001.0955 PMID: 16214687

44. Latt J, Chopard B. Lattice Boltzmann Method with Regularized Pre-collision Distribution Functions. Math Comput Simul. 2006; 72(2-6): 165–168. https://doi.org/10.1016/j.matcom.2006.05.017

45. Shan X, Chen H. A general multiple-relaxation-time Boltzmann collision model. Int J Mod Phys C. 2007; 18(04): 635–643. https://doi.org/10.1142/S0129183107010887

46. Adhikari R, Succi S. Duality in matrix lattice Boltzmann models. Phys Rev E. 2008; 78: 066701. https://doi.org/10.1103/PhysRevE.78.066701 PMID: 19256975

47. Chen H, Gopalakrishnan P, Zhang R. Recovery of Galilean invariance in thermal lattice Boltzmann models for arbitrary Prandtl number. Int J Mod Phys C. 2014; 25(10): 1450046. https://doi.org/10.1142/S0129183114500466

48. Geier M, Greiner A, Korvink JG. Cascaded digital lattice Boltzmann automata for high Reynolds number flow. Phys Rev E. 2006; 73: 066705. https://doi.org/10.1103/PhysRevE.73.066705 PMID: 16907021

49. Ishida T. Aerodynamic Simulations of a High-Lift configuration by Lattice Boltzmann Method with Block-Structured Cartesian Grid. In: AIAA Scitech 2019 Forum; 2019. p. 2306. https://doi.org/10.2514/6.2019-2306

50. Chávez-Modena M, Martínez JL, Cabello JA, Ferrer E. Simulations of Aerodynamic Separated Flows Using the Lattice Boltzmann Solver XFlow. Energies. 2020; 13(19): 5146. https://doi.org/10.3390/en13195146

51. Mattila KK, Philippi PC, Hegele LA Jr. High-order regularization in lattice-Boltzmann equations. Phys Fluids. 2017; 29(4): 046103. https://doi.org/10.1063/1.4981227

52. Shan X. Central-moment-based Galilean-invariant multiple-relaxation-time collision model. Phys Rev E. 2019; 100: 043308. https://doi.org/10.1103/PhysRevE.100.043308 PMID: 31771023

53. Hosseini SA, Darabiha N, Thévenin D. Compressibility in lattice Boltzmann on standard stencils: effects of deviation from reference temperature. Phil Trans R Soc A. 2020; 378(2175): 20190399. https://doi.org/10.1098/rsta.2019.0399 PMID: 32564724

54. Geier M, Schönherr M, Pasquali A, Krafczyk M. The cumulant lattice Boltzmann equation in three dimensions: Theory and validation. Comput Math Appl. 2015; 70(4): 507–547. https://doi.org/10.1016/j.camwa.2015.05.001

55. Gehrke M, Janßen CF, Rung T. Scrutinizing lattice Boltzmann methods for direct numerical simulations of turbulent channel flows. Comput Fluids. 2017; 156: 247–263. https://doi.org/10.1016/j.compfluid.2017.07.005

56. Sitompul YP, Aoki T. A filtered cumulant lattice Boltzmann method for violent two-phase flows. J Comput Phys. 2019; 390: 93–120. https://doi.org/10.1016/j.jcp.2019.04.019

57. Nishimura S, Hayashi K, Nakaye S, Yoshimoto M, Suga K, Inamuro T. Implicit Large-Eddy Simulation of rotating and non-rotating machinery with Cumulant Lattice Boltzmann method aiming for industrial applications. In: AIAA Aviation 2019 Forum; 2019. https://doi.org/10.2514/6.2019-3526

58. Malaspinas O. Increasing stability and accuracy of the lattice Boltzmann scheme: Recursivity and regularization. arXiv:150506900 [Preprint]. 2015. Available from: https://arxiv.org/pdf/1505.06900.

59. Coreixas C, Wissocq G, Puigt G, Boussuge JF, Sagaut P. Recursive regularization step for high-order lattice Boltzmann methods. Phys Rev E. 2017; 96: 033306. https://doi.org/10.1103/PhysRevE.96.033306 PMID: 29346972

60. Brogi F, Malaspinas O, Chopard B, Bonadonna C. Hermite regularization of the lattice Boltzmann method for open source computational aeroacoustics. J Acoust Soc Am. 2017; 142(4): 2332–2345. https://doi.org/10.1121/1.5006900 PMID: 29092578

61. Jacob J, Malaspinas O, Sagaut P. A new hybrid recursive regularised Bhatnagar-Gross-Krook collision model for Lattice Boltzmann method-based Large Eddy Simulation. J Turb. 2018; 19(11-12): 1051–1076. https://doi.org/10.1080/14685248.2018.1540879

62. Wissocq G, Coreixas C, Boussuge J. Linear stability of athermal regularized lattice Boltzmann methods. Phys. Rev. E. 2020; 102(5): 053305. https://doi.org/10.1103/PhysRevE.102.053305 PMID: 33327122

63. De Rosis A, Huang R, Coreixas C. Universal formulation of central-moments-based lattice Boltzmann method with external forcing for the simulation of multiphysics phenomena. Phys Fluids. 2019; 31(11): 117102. https://doi.org/10.1063/1.5124719

64. Fei L, Luo KH, Li Q. Three-dimensional cascaded lattice Boltzmann method: Improved implementation and consistent forcing scheme. Phys. Rev. E. 2018; 97(5): 053309. https://doi.org/10.1103/PhysRevE.97.053309 PMID: 29906988

65. Hegele LA, Scagliarini A, Sbragaglia M, Mattila KK, Philippi PC, Puleri DF, et al. High-Reynolds-number turbulent cavity flow using the lattice Boltzmann method. Phys Rev E. 2018; 98(4): 043302. https://doi.org/10.1103/PhysRevE.98.043302

66. Prasad AK, Koseff JR. Reynolds number and end-wall effects on a lid-driven cavity flow. Physics of Fluids A: Fluid Dynamics. 1989; 1(2): 208–218. https://doi.org/10.1063/1.857491

67. Albensoeder S, Kuhlmann HC. Accurate three-dimensional lid-driven cavity flow. Journal of Computational Physics. 2005; 206(2): 536–558. https://doi.org/10.1016/j.jcp.2004.12.024

68. Manoha E, Caruelle B. Summary of the LAGOON solutions from the Benchmark problems for Airframe Noise Computations-III Workshop. In: 21st AIAA/CEAS Aeroacoustics Conference; 2015. p. 2846. https://doi.org/10.2514/6.2015-2846

69. Beny J, Latt J. Efficient LBM on GPUs for dense moving objects using immersed boundary condition. In: Ibrahimbegovic A, Mattos Pimenta PD, editors. CILAMCE 2018 Proceedings of XXXIX Ibero-Latin American Congress on Computational Methods in Engineering; 2018. Available from: https://arxiv.org/pdf/1904.02108.