applied sciences

Article

Data-Oriented Language Implementation of the Lattice–Boltzmann Method for Dense and Sparse Geometries

Tadeusz Tomczak

Department of Computer Engineering, Wrocław University of Science and Technology, 50-370 Wrocław, Poland; tadeusz.tomczak@pwr.edu.pl

Abstract: The performance of lattice–Boltzmann solver implementations usually depends mainly on memory access patterns. Achieving high performance requires then complex code which handles careful data placement and ordering of memory transactions. In this work, we analyse the performance of an implementation based on a new approach called the data-oriented language, which allows the combination of complex memory access patterns with simple source code. As a use case, we present and provide the source code of a solver for D2Q9 lattice and show its performance on GTX Titan Xp GPU for dense and sparse geometries up to 4096^2 nodes. The obtained results are promising, around 1000 lines of code allowed us to achieve performance in the range of 0.6 to 0.7 of maximum theoretical memory bandwidth (over 2.5 and 5.0 GLUPS for double and single precision, respectively) for meshes of sizes above 1024^2 nodes, which is close to the current state-of-the-art. However, we also observed relatively high and sometimes difficult to predict overheads, especially for sparse data structures. The additional issue was also a rather long compilation, which extended the time of short simulations, and a lack of access to low-level optimisation mechanisms.

Keywords: parallel programming; CUDA; GPU; LBM

check for updates

Citation: Tomczak, T. Data-Oriented Language Implementation of the Lattice–Boltzmann Method for Dense and Sparse Geometries. *Appl. Sci.* **2021**, *11*, 9495. https://doi.org/10.3390/app11209495

Academic Editors: Wojciech Bożejko and Ugo Vaccaro

Received: 30 August 2021
Accepted: 10 October 2021
Published: 13 October 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Current high-performance computers use some form of parallel processing on many levels: beginning at instruction-level parallelism (ILP) and single instruction multiple data (SIMD) support, through the use of dynamic random access memories (DRAM), which transfer data in blocks containing several dozen bytes, up to multi/many-core chips and clusters of machines connected with a fast network. Thus, to effectively use the available hardware, the processed data should be carefully arranged in a way that allows usage of all available hardware with minimal losses. For example, DRAM block transactions connected with SIMD processing are tuned to large data sets containing elements processed in the same way. When neighbouring elements require different operations, the hardware is usually significantly underutilised. These limitations cause that many computational problems, for example in the physic simulations area, require not only sophisticated algorithms but also non-trivial data layouts in the memory to achieve high performance. Typical examples of such problems are simulations on sparse geometries, i.e., geometries for which computations must be performed only for a small part of area/volume.

The lattice–Boltzmann method (LBM) is a computational fluid dynamics (CFD) algorithm based on cellular automata idea, where automaton cells correspond to points (called *nodes*) of a uniformly discretised domain of computations. One of the main advantages of LBM is its inherent parallelism; thus, many high-performance LBM implementations are known.For dense geometries, the implementation may be relatively simple [1,2] and allow the achievement of high hardware utilisation (up to about 80% of peak theoretical memory bandwidth) [3,4]. However, when the significant part of geometry is solid and many nodes of a discretised domain do not take part in computations, then the more complex implementation techniques have to be used to avoid memory, bandwidth, and computational power waste.

The boundary between dense and sparse geometries may be difficult to define, especially when different constraints must be considered. From the performance point of view, the geometry can be treated as sparse if dedicated techniques allow decreasing simulation time or memory usage. An approximate distinction can also be based on application. For example, aerodynamic simulations for car, aviation and space industries can be often treated as dense because the simulated area is usually significantly larger than models of a car or an aircraft wing. Medical simulations of large fragments of vascular systems are usually sparse—the veins can occupy even less than 1% of the bounding volume. In oil or chemical industries, different porous materials are used: sandstones, chemical reactors, elements of fuel cells, etc. However, since their parameters can vary within wide limits, thus the decisions about using techniques for dense or sparse geometries should be made for each case separately.

LBM implementations for sparse geometries are based on two main approaches: indirect addressing, where each node contains additional information about localisation of neighbouring nodes, and spatial discretisation, where the information about geometry sparsity is stored for fragments containing a number of neighbouring nodes from the domain. Two main indirect addressing approaches are the *connectivity matrix* [5], used in MUPHY [6] and ILBDC [7] solvers, and the *fluid index array* [4,8,9]. Spatial discretisation is used in HemeLB [10], Palabos [11], OpenLB [12], Musubi [13] and WaLBerla [14] platforms. A more detailed review of LBM implementations for sparse geometries can be found in [15].

Although there are known many techniques that allow increase in the utilisation of hardware resources for computations on sparse geometries, implementation of these techniques requires a significant amount of work, especially for different machines. Even the rather simple implementation of one-level spatial discretisation from [16,17] requires more than ten thousand lines of heavily templated C++ code. In this context, an interesting approach to reduce the amount of work required to port the code to different platforms was presented in [4], where the final code for target machines was generated from templates written in the Python Mako library.

Recently, the data-oriented parallel programming language Taichi appeared [18], which simplifies the development of high-performance codes for computations on both sparse and dense data structures. It allows not only generation of the final code for different target machines, but the more important feature is decoupling information about data structures from computational kernels. By providing *structural nodes*, which can be used to build complex, hierarchical data structures, and at the same time offer a simple interface simulating access through `[]` operator like for dense data structures, the Taichi language allows design of only simple, basic codes describing computations only.

In this work, we present an implementation of the lattice–Boltzmann method in the Taichi language for both dense and sparse geometries and investigate its performance on a massively parallel graphic processing unit (GPU). To our knowledge, currently, there are no published studies on Taichi-based LBM solvers. A simple code of one existing attempt is available [19], but it was designed to keep the code simple, thus its performance is low. The implementation presented in this work is loosely based on this simple version, but we significantly redesigned the code to improve performance and handle a wider set of boundary conditions. Our implementation is created mainly for performance analysis, although we did some elementary correctness tests. We also provide the source code (available at https://github.com/tadeusz-tomczak/tilb (accessed on 30 August 20201)) which, as we believe, can be a good starting point for building high-performance simulations of more complex physical phenomena.

2. Materials and Methods

2.1. Taichi Language

The Taichi programming language [18] is an actively developed open-source [20] just-in-time compiler that translates Python-like source code into binary code for different hardware platforms (various CPUs, CUDA, AMDGPU and others). The language uses the standard Python syntax extended with decorators marking some functions as *kernels*. During compilation, the kernels are transformed into optimised binary codes, which are then called from the surrounding Python source. Such a solution allows easy binding of efficient, application-specific kernels with a wide variety of available libraries and tools, especially given that Taichi provides a basic GUI system and simple built-in interfaces to NumPy and PyTorch libraries. The kernel compiler applies a few levels of optimisations, including simple template instantiation, loop unrolling and vectorisation, constant folding, and others. Inside kernels, the highest level loops can be automatically parallelised provided that operations are performed on Taichi *fields*.

Fields are the multidimensional data structures that provide an array-like data access interface via the `[]` operator. Internally fields use a hierarchy of *structural nodes* (SNodes) which define dimensions, size and data arrangement in memory. Information from SNodes is used, both during kernel compilation and runtime, to optimise data access to field elements and to distribute workload onto available threads/processors. Compilation-time optimisations allow decreasing overheads caused by traversing nested SNodes but require that the field definition must be known at the moment of kernel compilation and enforce separate kernel compilations for different data structures.

The types of SNode allow the use of different data layouts. The *dense* layout is a simple, multidimensional array. The *bitmasked* layout allows addition of information whether given elements contain valid data or not. This layout does not reduce memory usage because all bitmasked elements must be still placed in memory. However, computational kernels are called only for data elements masked as valid, and thus the *bitmasked* layout can reduce the number of operations for sparse data. Reduction of memory usage for sparse data is possible with the *pointer* data layout that can mark pointers to non-existent data as invalid and thus does not require memory allocation in this case.

The data layout in Taichi can be defined as a hierarchy of different SNodes. For example, *ti.root.pointer (ti.i, 16).bitmasked (ti.i, 8).dense (ti.i, 4)* defines 16 pointers, each pointing to 8 bitmasked dense blocks of data where each dense block contains 4 data elements (not shown). After definition, this structure can be accessed using a simple *[i]* operator, where i $\in \{0, \ldots, 511\}$. The calculations of memory addresses, traversing and checking values of pointers and bitmasks, and launching of the appropriate number of computational kernels are internally handled by Taichi. Additionally, some optimisations are applied to reduce the overheads caused by additional memory accesses required to traverse multi-level, hierarchical data structures.

2.2. Lattice–Boltzmann Method

The lattice–Boltzmann method (LBM) is a numerical approach to solve the Navier–Stokes equations, which describe the motion of fluids. The detailed LBM description with the theoretical background is available in many books [21–23]; thus, in this work, we only show a minimal introduction from the implementation point of view.

In LBM, the domain is discretised into a uniform, usually Cartesian, mesh containing *nodes* distant by the *lattice spacing* δx along all axes. During computations, nodes communicate with some neighbours—the choice of neighbours depends on *lattice arrangement* which defines the dimension of the problem and communication pattern between neighbouring nodes (*lattice linkage*). The lattice arrangement is usually described using DdQq notation, where d is the dimension, and q is the linkage. For example, the D2Q5 arrangement defines a 2D lattice where nodes communicate only with neighbours placed along the axes (left, right, top, bottom), whereas D2Q9 takes into account also the nodes placed at diagonal corners (see Figure 1).

A single iteration of simulation advances simulation time by the time step δt and includes one-time communication between all nodes and additional computations. To simplify equations, it is usually assumed that $\delta x = 1$ and $\delta t = 1$. In this case, the initial *characteristic velocity* U and the fluid viscosity ν have to be set up to keep the required Reynolds number

$$\mathrm{Re} = \frac{U \cdot L}{\nu}, \tag{1}$$

where L denotes the *characteristic length* which is dependent on the selected size in simulated geometry.

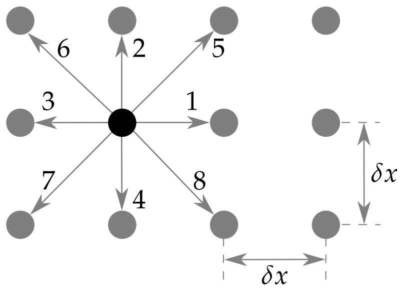

Figure 1. D2Q9 lattice arrangement and indices of lattice links (index 0 denotes the node itself). Circles denote lattice nodes.

Each node contains a set of *particle distribution functions* (PDF) $f_i(\mathbf{x}, t)$, where $\mathbf{x}$ denotes node position, t denotes time, and i denotes the index of function corresponding to lattice linkage. The PDF numbering can be chosen in different ways; in this work we use one of the most often presented in Figure 1. The macroscopic fluid density ρ and velocity $\mathbf{v}$ are related with PDFs according to equations

$$\rho(\mathbf{x}, t) = \sum_i f_i(\mathbf{x}, t) \quad \text{and} \quad \mathbf{v}(x, t) = \frac{1}{\rho(\mathbf{x}, t)} \frac{\delta x}{\delta t} \sum_i \mathbf{v_i} f_i(\mathbf{x}, t), \tag{2}$$

where $\mathbf{v_i}$ are called *lattice (microscopic) velocities* and are equal to vectors from the current to the neighbouring node. For the D2Q9 lattice shown in Figure 1, the selected values of $\mathbf{v_i}$ are the following: $\mathbf{v_0} = [0, 0], \mathbf{v_1} = [1, 0], \mathbf{v_4} = [0, -1], \mathbf{v_6} = [-1, 1]$, etc.

The LBM operations are described by equation

$$f_i(\mathbf{x} + \mathbf{v_i}\delta t, t + \delta t) = f_i(\mathbf{x}, t) + \Omega_i, \tag{3}$$

where Ω_i is named the *collision operator*. Simple LBM implementations are then often realised as two alternating steps: *collision* computing new PDF values according to the right-hand side of Equation (3), and *streaming* responsible for transferring the values computed during the collision step into the places defined by the left-hand side of Equation (3).

One of the simple, yet widely used, collision operators is the Bhatnagar, Gross and Krook (BGK) operator [24] defined as

$$\Omega_i^{BGK} = -\omega\Big(f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)\Big), \tag{4}$$

where ω is called the *collision frequency* and f_i^{eq} is the *local equilibrium distribution function*. Assuming both δx and δt equal to one, the collision frequency results from the fluid viscosity ν as

$$\omega = \frac{1}{3\nu + \frac{1}{2}}. \tag{5}$$

The local equilibrium distribution function is defined as

$$f_i^{eq}(\mathbf{x}, t) = w_i \rho \Big(1 + 3\mathbf{v_i} \cdot \mathbf{v} + \frac{9}{2}(\mathbf{v_i} \cdot \mathbf{v})^2 - \frac{3}{2}\mathbf{v} \cdot \mathbf{v}\Big), \tag{6}$$

where constants w_i depend on lattice arrangement and for D2Q9 are $w_0 = \frac{4}{9}, w_{1,2,3,4} = \frac{1}{9}$, $w_{5,6,7,8} = \frac{1}{36}$.

3. Implementation

The LBM implementation can directly follow Equation (3) but, in our version, we applied some of the techniques developed for higher performance: the *fused* kernel and the *pull* scheme [25], which are used on different hardware platforms, and a collection of low-level, GPU-only optimisations described below. The general idea of the presented implementation is shown in Algorithm 1. The algorithm contains three main operations.

Algorithm 1: General idea of the presented LBM implementation. $\rho, \mathbf{v}, f_i'$ and f_i'' denote temporary values used only inside the kernel.

```
1  for all nodes do
2      Set initial values of $\rho, \mathbf{v}$
3      Set $f_i^{pre} = f_i^{eq}$
4  repeat
5      for all nodes at positions $\mathbf{x}$ do in parallel
6          Kernel stream and collide
7              for all directions $i$ do
8                  gather $f_i' \leftarrow f_i^{pre}(\mathbf{x} - \mathbf{v_i})$
9              compute $\rho$ and $\mathbf{v}$ from $f_i'$ using Equation (2)
10             for all directions $i$ do
11                 compute $f_i^{eq}$ from $\rho$ and $\mathbf{v}$ using Equation (6)
12                 compute new $f_i''$ from $f_i', f_i^{eq}$, RHS of Equations (3) and (4)
13                 store $f_i^{post}(\mathbf{x}) \leftarrow f_i''$
14     for all nodes at positions $\mathbf{x}$ do in parallel
15         swap $f_i^{post}(\mathbf{x}) \rightleftarrows f_i^{pre}(\mathbf{x})$
16 until end of simulation;
```

Lines 1–3 are responsible for the initialisation of PDFs values f_i that are set to equilibrium state f_i^{eq} computed from initial values of velocity and density. Initialisation is done only once at the beginning of computations. Then, the simulation comes down to computing values of PDFs for the successive time steps.

A single iteration of lines 5–15 corresponds to a single time step. During the time step computations, all nodes are processed in parallel. To avoid race conditions, we use two copies of f_i functions: $f_i^{pre} = f_i(t)$ functions were computed during the previous time step and are only read, and $f_i^{post} = f_i(t + \delta t)$ functions are computed during the current time step and are only written. There are also known parallel implementations that use a single copy of PDFs only [26] at the cost of increased code complexity.

To minimise the memory bandwidth, we implemented the *fused* kernel (lines 6–13), where collision and streaming are conducted in one step with a single read and write of f_i values. Additionally, we use the reversed order of collision and streaming, also known as the *pull* scheme [25]. Direct implementation of Equation (3) computes the collision first and then *scatters* the new, computed f_i values to neighbour nodes. In the pull approach, first, the f_i values from the previous time step are *gathered* from neighbour nodes, then the collision is applied, and, eventually, the new f_i values are stored in the current node. This scheme keeps addresses of all writes to memory aligned what may additionally decrease memory traffic because unaligned memory writes often are more costly than unaligned reads (for example due to allocate-on-write policy).

After processing of all nodes, the values of f_i^{pre} and f_i^{post} are exchanged in lines 14–15. However, since we have not found an efficient way to exchange fields, then we use two kernels with identical computations, but one of the kernels reads f_i^{pre} and writes f_i^{post} and the second kernel does the opposite.

The operations shown in Algorithm 1 do not include support for boundary conditions, which is also present in the implemented kernel. To detect boundary nodes, we store in memory an additional field encoding each node type (fluid, solid, boundary type) along with a bitmask containing information about which neighbour nodes are present. We also use a separate field to store values for boundary nodes with fixed conditions, e.g., constant velocity. Supported boundary conditions are constant velocity, constant pressure, and bounce back according to [27].

In addition to the optimisations mentioned above, we also applied a few low-level optimisations: we used the structure-of-arrays instead of the array-of-structures data layout, minimised the number of memory operations and placed them in a non-divergent code to allow coalescing, and used numeric constants reducing the number of floating-point arithmetic operations. These optimisations were applied after analysis of the generated GPU assembly. Moreover, we resigned from encapsulating functionalities inside Python classes due to a small drop in performance. Additionally, the whole code (except auxiliary functions) is placed in a single source code file which simplified the management of memory allocation and kernel generation. The complete code contains slightly more than 1000 source lines of code, including geometry generation and storage of results.

For convenience, we also allocate memory for values of velocity $\mathbf{v}$ and density ρ, although these are not used during computations but only for data initialisation, storage of results, and visualisation. To allow run-time generation of images illustrating velocity fields, we also allocate an additional field for image memory, although it can be removed when not used.

The source code structure contains a single, main library file *tilb.py*, which is then used in separate scripts responsible for creating and running simulations. The file *tilb.py* provides a collection of functions that allow allocation of memory and perform computations according to Algorithm 1. These functions must be called in the correct order to properly initialise internal library data structures that are then needed to create Taichi fields (allocate memory) and generate computational kernels. The typical sequence is the following:

1. Set the selected data type (single- or double-precision), data layout, and geometry resolution;
2. Create Taichi fields containing PDFs and auxiliary data—this requires information from the previous step;
3. Set required values of node types—solid nodes can be left in an uninitialised state which can reduce the memory usage when sparse data layouts are used;
4. Set initial $\rho, \mathbf{v}$, compute ω, and initialise PDFs to local equilibrium;
5. Generate a kernel for stream and collide—this step is performed automatically during the first kernel call and requires information from the first step and about geometry sparsity to allow optimisations;
6. Compute a required number of time steps with optional saves of simulation state.

4. Results

The experiments were conducted on the computer containing NVIDIA GPU GTX Titan Xp with 12 GB GDDR5X memory with the 384-bit bus at 5.705 GHz (547.68 GB/s), Intel CPU i7-4930K at 3.4 GHz, and 48 GiB, four-channel 64-bit DDR3 memory at 1067 GHz (68.256 GB/s). We used a Linux operating system (Ubuntu 16.04 with x86_64 kernel version 4.10.0-38), CUDA compilation tools release 10.0, Python 3.8.8, and Taichi language version 0.7.26. Code profiling was performed in an NVIDIA Visual Profiler.

4.1. Validation

We validated the code for the three standard cases: lid-driven cavity, flow through a channel and flow past a cylinder. Due to simple boundary conditions and collision models, not all cases gave good agreement with physical models, but we used them as a method to validate code correctness. All computations were conducted in single precision using the dense Taichi data layout.

4.1.1. Lid-Driven Cavity

The lid-driven cavity flow is a standard CFD benchmark, where the flow inside a square chamber is driven by a constant velocity at the chamber top lid. Depending on the Reynolds number, different vortex structures can be observed. The characteristic length L equals the length of the chamber side ($L = n_y - 1$ in lattice units), the x-velocity of the top lid is the characteristic velocity U (we used $U = 0.1$ in lattice units), and y-velocity of the top lid is zero. The velocities at all other nodes are assumed to be zero. On the top wall, the constant velocity boundary condition was imposed. For other walls, we used the bounce-back boundary conditions.

The results were compared with data from [28] for different mesh resolutions $n_x \times n_y$, an example is shown in Figure 2. To obtain the correct Re values, the fluid viscosity was computed according to Equation (1) as $\nu = U \cdot (n_y - 1)/\mathrm{Re}$. A uniform fluid density $\rho = 1$ was imposed initially. For some combinations of mesh resolution and Reynolds numbers (for small resolutions and high Re numbers, as well as for large resolutions and small Re numbers), we observed numerical instabilities. Verified simulation settings are shown in Table 1. The simulation for Re = 10,000 was stopped before fully converging (mean squared error $\sum(v - v_{ref})^2/n$ was at the order of 10^{-3}, maximum relative error was about 30% for v_x at $y = 0.2813$) and stabilised (we could still observe small, fading swirling waves), but it was slowly approaching the reference velocity profiles. We also observed that, for $\mathrm{Re} = 3200$, a single reference data point at $(y = 0.4531, v_x = 0.86636)$ was significantly different than the others.

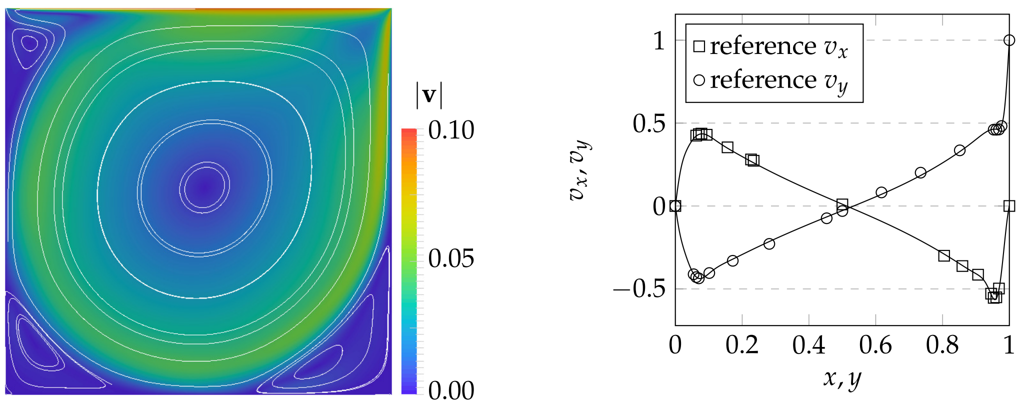

Figure 2. Lid-driven cavity results for Re = 5000 and D2Q9 lattice with 2048 × 2048 resolution. The picture on the **left** was generated from simulation results with ParaView [29] and shows the magnitude of velocity and streamlines (denoted with white lines) after 3×10^6 time steps. The plot on the **right** contains a comparison of velocity profiles (lines) with the reference solution (symbols) adapted from [28]. The complete simulation took about 44 min on a GPU.

Table 1. Validated cavity simulations for different Reynolds numbers Re. Time of computations is given for GTX Titan Xp GPU and includes the compilation of kernels (about 20 s) and a few dozens of saves of the simulation state to the disk.

Re	Mesh Resolution	Number of Time Steps	Computation Time
100	128 × 128	12,000	29 s
1000	256 × 256	100,000	56 s
3200	1024 × 1024	1,000,000	4 min
5000	2048 × 2048	3,000,000	44 min
10,000	4096 × 4096	10,000,000	12.5 h

4.1.2. Channel Flow

The flow through a channel can be solved analytically and is often used to validate the correctness of CFD solvers. In the channel flow case, the fluid flow is analysed for a long channel with a radius R (for the 2D case the radius equals half of the channel height), and the initial conditions force the flow along the channel. The stabilised flow should form a parabolic velocity profile

$$v_x = v_{max} \cdot \left(1 - \left(\frac{r}{R}\right)^2\right) \tag{7}$$

with maximum velocity v_{max} at the centre of the channel (r denotes the distance from the centre).

We tested two versions of channel flow differing with inlet boundary conditions. Simulation parameters were set to get similar v_{max} for both cases. The first case, denoted as chan_v, used the constant velocity $v_x = 0.1$ boundary condition at inlet. For fluid nodes, initial density was set to $\rho(t_0) = 1.0$. In the second case, chan_p, the inlet boundary condition was set to constant pressure with $\rho = 1.016$, and the initial density for fluid nodes was set to $\rho(t_0) = 1.008$. Both versions used channels containing 4096×512 nodes with bounce-back boundaries on the top and bottom walls. Outlet condition was set to constant pressure with $\rho = 1.0$, fluid viscosity was $\nu = 0.25$, initial velocity for fluid nodes was set to $\mathbf{v}(t_0) = [0, 0]$. We calculated 10^6 time steps which on the GPU took about 10 min per case, including saves of the simulation state every 10^4 time steps. The achieved computational kernel performance was 5.38 GLUPS, 387 GB/s for the single-precision version.

As can be seen in Figure 3, the obtained velocity profiles were close to parabolic. The maximum values of velocities were slightly different ($v_{max} = 0.158$ for chan_v and 0.160 for chan_p). Since we were using the quasi-compressible fluid model, then we were not able to achieve a complete agreement with theoretical models.

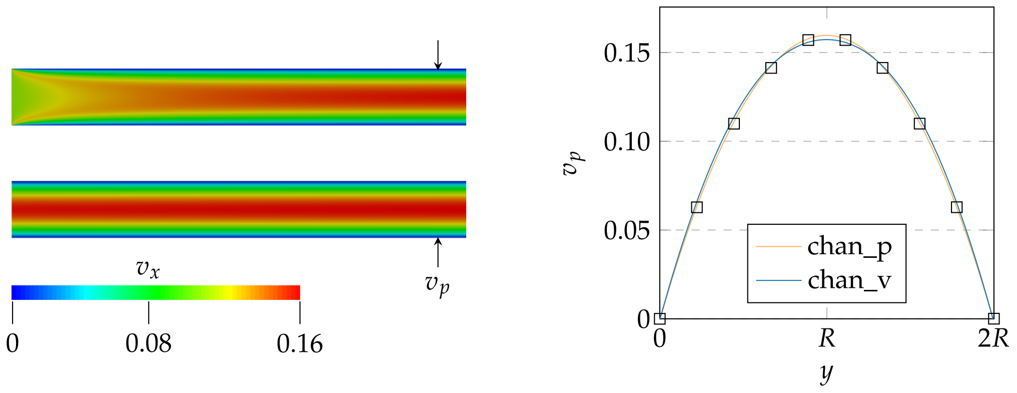

Figure 3. Velocity v_x for channel flows with constant inlet velocity $v_x = 0.1$ (picture in the **top left**) and constant pressure $\rho = 1.016$ (**bottom left**) together with velocity profiles v_p (plot on the **right**). The profiles were taken at points $x = 15 \times R$ (marked with arrows), where R is half the channel height. Squares denote reference values computed from Equation (7) for arbitrarily chosen $v_{max} = 0.159$.

4.1.3. Flow Past Cylinder

Flows past stationary cylinders of different shapes are also one of the typical CFD problems. For low Reynolds numbers, the flow is stationary. With increasing Re, the unsteady phenomenon called the Kármán vortex street appears.

Our simulations were based on the chan_v channel flow described above. All parameters were identical except for viscosity which was changed to observe different flow patterns. We used two often analysed, standard circular and square cylinders to find errors with handling boundary conditions at corner nodes. The cylinders were placed at a distance from the inlet equal to two heights of the channel. On the cylinder surface, we used the bounce-back boundary condition. The simulation time and performance were similar to those in the chan_v case. The results are shown in Figure 4 where typical behaviour can be observed.

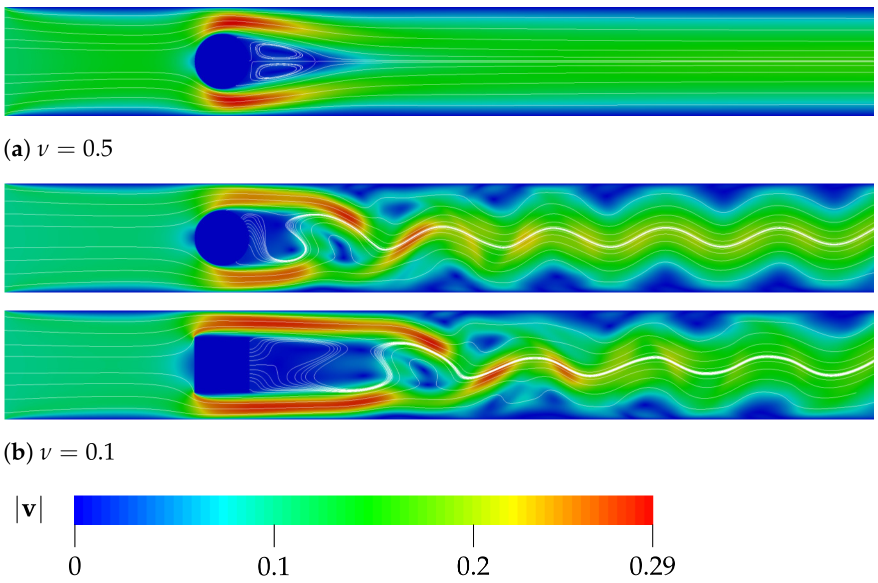

Figure 4. Velocity magnitude $|\mathbf{v}|$ and streamlines (white) after 10^6 time steps for flows past a cylinder at different viscosities ν. The picture on the top shows a separation bubble comprising two symmetric and counter-rotating recirculation zones. Other images contain unsteady Kármán vortex street patterns. At the inlet, constant velocity is set to $v_x = 0.1$. Mesh resolution is 4096×512 nodes, sphere diameter and square edge are equal to half of the channel height.

4.2. Performance

4.2.1. Memory Bandwidth

Before analysis of the implemented kernel, we first measured available memory bandwidth during a simple copy of data between one-dimensional arrays for different data layouts implemented in Taichi. The results are shown in Figure 5. Notice that we use both SI (k $= 10^3$, M $= 10^6$, G $= 10^9$) and binary (Ki $= 2^{10}$, Mi $= 2^{20}$, Gi $= 2^{30}$) prefixes. As an approximate reference, we run the NVidia *bandwidthTest* utility. Internally, *bandwidthTest* measures calls to the *cudaMemcpy* function, but it should be noted that this utility does not use preliminary "warm up" of the measured code. Furthermore, the *bandwidthTest* has no support for size arguments larger than 2^{31} bytes, and the results are displayed in MiB/s and must be scaled.

For Taichi, we prepared a simple kernel copying data from one one-dimensional array to the other. The number of threads per thread block was explicitly set to 512 since it resulted in high average performance. Time duration measurements were conducted using Python *time.perf_counter* for 100 kernel calls (as in *bandwidthTest*). Additionally, before each measurement, we initially called the measured kernel five times to force runtime compilation and initialise contents of cache memories and translation lookaside buffers (TLB).

To investigate the strange behaviour observed in Figure 5 for memory block sizes around 1 MiB, we prepared an additional kernel shown in Listing 1. We used custom kernel copying data because it is difficult to observe internal implementation details of the *cudaMemcpy* function. For a large amount of transferred data, the performance of the simple kernel is slightly lower than the performance of *cudaMemcpy*, but for small memory blocks, the copy kernel takes less time. We can thus suppose that *cudaMemcpy* uses an optimised method of data copying but introduces more overhead than a simple kernel launch.

When small data blocks are copied, for both data copying methods, the performance increases with block size due to a growing ratio of transfer time to overheads (transfer time grows with memory block size). Because of a high cache hit ratio, it is possible to obtain bandwidth even higher than the maximum memory bandwidth—for 1,383,048 bytes block size, the simple kernel achieved up to 663 GB/s and L2 cache hit ratio about 0.5. If the block size increases further, the performance rapidly drops and eventually stabilises at about 0.8 of peak memory bandwidth.

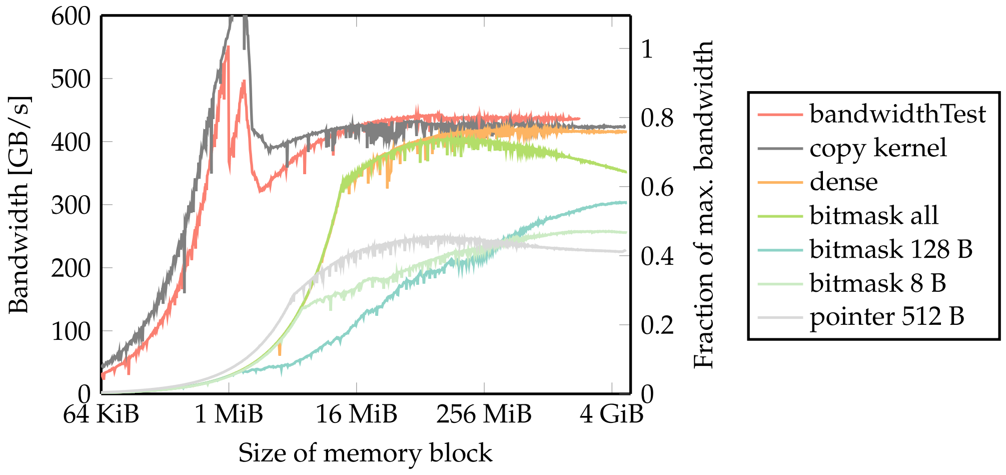

Figure 5. Bandwidth comparison of the simple CUDA copy kernel, *bandwidthTest* utility, and Taichi kernels copying linear memory. Numbers after "bitmask" and "pointer" denote the size of the data block assigned to a single bitmask/pointer.

Listing 1. Simple CUDA kernel copying memory.

```
__global__ void
copy_simple (const double * src, double * dest, const unsigned long length)
{
  const unsigned idx = threadIdx.x + (blockIdx.x * threadsPerBlock);

  if (idx < length)
  {
    dest [idx] = src [idx];
  }
}
```

Low performance for Taichi kernels and small block sizes results mainly from large overheads between consecutive kernel launches. Code profiling revealed that Taichi brings about 100 µs long breaks between subsequent kernel calls, whereas raw CUDA-C implementation needs only single microsecond stoppages.

As can be seen in Figure 5, the memory bandwidth measured for Taichi kernels strongly depends on the used data layout. For the large size of transferred data, the maximum transfer (419.7 GB/s for 500 MB block size) was observed for the standard dense layout. In the gigabyte range, the average bandwidth was 416 GB/s which is only 5% lower than 436 GB/s reported by *bandwidthTest*.

When the bitmasked layout is used, the memory bandwidth strongly depends on the amount of data masked by a single bit. For a single masking bit per 8 bytes of data, the maximum bandwidth was 258 GB/s (at about a 4 GiB block size). When a single bit was used to mask the whole block of data and small data blocks were transferred, the measured bandwidth was similar to that for dense layout. However, starting from about 200 MiB block size, the bandwidth slowly dropped to 352 GB/s for a 5.41 GiB block size. For many different sizes of bytes per single bitmask, e.g., 64 KiB, the measured bandwidth dropped even to less than 30 GB/s. For fine-grained bitmasking, the highest bandwidth (up to 304 GB/s when a few GiB of data were transferred) was observed when 128-byte blocks were masked.

The pointer layout has the lowest maximum measured bandwidth, although it is less dependent on the size of the block of data assigned to a single pointer than the bitmask layout. For a single pointer per 8 bytes of data, the bandwidth was below 20 GB/s and, additionally, we had to significantly decrease the amount of allocated memory. With the increasing size of the pointed data block, the bandwidth steadily increased, achieving 247 GB/s at 124 MiB transferred data block for a single pointer per 512 bytes of data, as shown in Figure 5. Then, when increasing the amount of data per single pointer, the maximum bandwidths stayed in the 230–250 GB/s range for up to 16 MiB of data per pointer. After this limit, the bandwidth dropped to 144 GB/s when a single pointer was used for a block containing 1 GiB of data. We can also observe that the pointer layout has different overheads than other data layouts—for the small size of the data block, the performance plot for the pointer layout has a different shape than for other layouts.

The data from Figure 5 allow us to draw a general conclusion that, for dense data layouts, the Taichi language brings in a small, usually negligible overhead. However, sparse layouts (bitmask and pointer) reduce available bandwidth by at least about 40% for fine-grained resolution. It should be noticed that, for all data layouts available in Taichi, we were trying to find parameters giving the highest bandwidth, although we did not conduct the exhausting measurements for all available combinations. We could have then missed some significantly better settings, although this seems unlikely.

4.2.2. LBM Performance

The performance P of LBM implementations is often measured in *lattice updates per second* (LUPS), which define the number of processed nodes per unit of time. However, direct comparison of P_{LUPS} values for different implementations is difficult since the amount of processed data per node depends on both the lattice arrangement and the data type, usually either single (32 bits) or double (64 bits) precision floating-point number. Since LBM implementations are usually bandwidth-bound on available machines, in this work, we then define LBM performance as a theoretical, minimal bandwidth P_B required to achieve given LUPS for specific lattice arrangements and data types.

Let s_d denote the size (number of bytes) required to store a single number ($s_d \in \{4, 8\}$ for single- and double-precision floating-point numbers). Assuming that, in an ideal case, processing of a lattice node requires only read and write of all q functions f_i, the minimum amount of transferred data per single node is $B_{node} = 2 \cdot q \cdot s_d$ bytes. The LBM implementation performance can be then defined as $P_B = B_{node} \cdot P_{LUPS}$. We can also define a theoretical bandwidth utilisation $U_B = P_B / B_{peak}$, where B_{peak} denotes the maximum theoretical memory bandwidth of a given machine. For the GTX Titan Xp GPU, $U_B = C \cdot P_{GLUPS}$, where $C \in \{0.131, 0.263\}$ for single and double precision, respectively, and P_{GLUPS} is performance in 10^9 LUPS. The values of U_B can be then compared for different machines showing how much room for potential improvements is still available. However, the U_B coefficient does not take into account the additional limitations of the machine that prevent it from achieving full memory bandwidth.

The performance of the LBM kernel was measured for dense and sparse geometries, different data structures, and single and double precision numbers. Results for sparse data structures for dense geometry can be treated as an estimation of introduced overheads compared to the dense memory layout. Measurements were conducted using Python *time.perf_counter* for 1000 kernel calls. Before measurements, each kernel was called 100 times to force the runtime compilation and warm up the whole system. For sparse data layouts, we turned on the experimental *async_mode* available in Taichi that disables analysis of geometry sparsity before each kernel call because, in our case, the geometry is static during computations. This mode allowed increase in performance, but we observed sporadic problems with stability.

Performance for Dense Geometries

As an example of dense geometry, we used the cavity case at different mesh resolutions. Measurements for each data point require more than 20 s because of kernel compilation. Thus, we limited the number of checked mesh resolutions to about 100 on a logarithmic scale. To keep results comparable, all measurements for different data types and layouts are for the same set of geometry resolutions. The obtained performance is shown in Figure 6 and Table 2.

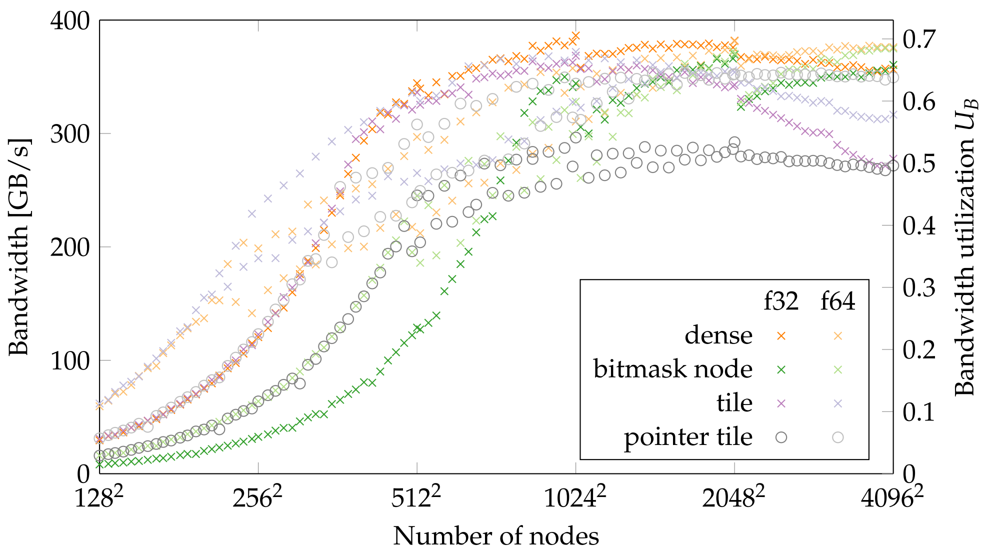

Figure 6. Performance of cavity GPU simulation for different mesh sizes, memory layouts and data types (f32 and f64 denote single- and double-precision floating-point numbers, respectively).

Table 2. Performance of cavity simulations for different data layouts. Performance P_{LUPS} is given in GLUPS, P_B in GB/s. Column "Mesh" contains mesh size for which maximum performance was observed. Average performance is computed for meshes containing at least 1024^2 nodes. Reference performances for other work are adapted from [3,4,16,17] and contain results for collision operators similar to the one used in this paper. For GPUs different to GTX Titan Xp, only U_B values can be directly compared. The works of [3,4] use optimised dense layouts, whereas [16,17] employ tiles.

Layout	**Maximum**				**Average**		
	Mesh	P_{LUPS}	P_B	U_B	P_{LUPS}	P_B	U_B
f32 dense	1024^2	5.37	386	0.705	5.12	369	0.674
f32 bitmask node	2048^2	5.13	370	0.675	4.78	344	0.629
f32 tile	1024^2	5.12	369	0.673	4.53	326	0.596
f32 pointer tile	1024^2	4.12	296	0.541	3.85	277	0.506
f64 dense	2048^2	2.65	381	0.696	2.51	362	0.661
f64 bitmask node	3955^2	2.60	375	0.685	2.40	346	0.631
f64 tile	1024^2	2.58	372	0.679	2.37	341	0.622
f64 pointer tile	2509^2	2.46	354	0.646	2.42	348	0.635
Other work							
[4] D3Q19, f64, K40	243^3	0.77	235	0.816	-	-	-
[4] D3Q19, f32, K40	243^3	1.48	226	0.782	-	-	-
[3] D3Q19, f32, K20c	192^3	1.04	158	0.757	0.98	149	0.714
[16] D3Q19, f64, TITAN	-	0.64	194	0.674	-	-	-
[17] D3Q19, f32, Titan Xp	100^3	2.20	334	0.609	-	-	-
[17] D3Q19, f64, Titan Xp	252^3	0.99	301	0.550	-	-	-
[16] D2Q9, f64, TITAN	-	1.02	147	0.509	-	-	-

We compared performance for four different data layouts. The dense layout is a standard, multidimensional array containing all PDFs for all nodes. As excepted, this version offers the highest performance to 70% of peak GPU memory bandwidth. This result is close to the best-reported values for highly optimised codes from [3,4] and is consistent with performance reported for the same hardware in [17]. During code profiling, we observed that loads of all PDFs (excluding f_0, f_2 and f_5) cause uncoalesced transactions because, as shown in [4], neighbour f_i values are shifted in memory by one position. We found no method to correct this behaviour—the typical technique is the usage of shared memory but the Taichi language offers no such feature.

The bitmasked data structures can be used in many ways. At first, we applied a single bitmask per each f_i function because, due to the structure-of-arrays data layout, we were not able to apply a single bitmask per a whole lattice node. The observed performance dropped more than twice compared to the dense layout—maximum bandwidth was at the level of 140 GB/s for the single-precision version.

However, the bitmask layout does not save memory and serves only as a convenient way to skip computations for non-existent data. The reasonable way is then to use bitmasks only for a field containing encoded node type. This layout enables for simple management of sparse geometries and is marked as "bitmask node" on performance plots. As can be seen in Figure 6 and Table 2, when a single bitmask is used per whole data of a single lattice node, the performance loss is less than 10% compared to the dense layout for geometries containing at least 10^6 nodes. For smaller geometries, the bitmask layout has low performance. An additional advantage of this approach is that, due to the fine-grained masking of single nodes, only valid nodes are processed, even for very complex geometries.

The pointer layout available in Taichi can also be used in many ways, but the applied method should allow storage in memory only values used during computations. We then used a single pointer per *tile* containing data for 16^2 neighbour nodes. The resulting data layout is denoted as "pointer tile". Additionally, we measured performance for the "tile" layout defined as a dense array of tiles without an additional layer of pointers.

The measured performance of the tile layout was similar to the dense layout, but only for geometries with less than 2048^2 nodes. After this limit, the achieved bandwidth utilisation dropped even below 0.5 for single-precision data and the largest geometry. We observed two issues appearing for the tile data layout. First, the dimension of the CUDA thread block had to be reduced to the number of nodes per tile—in our case from 512 to 256 threads per block. For the dense layout, such a change of thread block size decreased bandwidth from 360 to 338 GB/s for the cavity 4096^2 case and single-precision data. Next, the Taichi has no support for low-level optimisations presented in [16,17], e.g., usage of shared memory, warp level programming, and LBM-optimised index calculations inside a tile. For example, we tried to store f_1 and f_3 functions using column-major order, but we encountered different behaviour than described in Taichi documentation. It is also probable that the observed decrease in performance may be caused by other, as yet undetected reasons.

For the pointer tile layout, the performance is slightly surprising. When double-precision data are used, then the performance stays high and steady even for the largest geometries, despite the drop-out observed for tile layout. On the other hand, performance for single-precision is also almost constant but at the low level ($U_B = 0.5$) given by the minimum value observed for tile layout and the largest geometry. We have not found the cause of such behaviour yet and only found that the code handling pointers in the kernel is quite complex and significantly increases register pressure—the kernel required 70 registers which decreased the theoretical occupancy to 37.5%.

The presented data show that we achieved high performance for each of the presented layouts despite the low bandwidth observed during simple data copy of sparse layouts. Only the tile-based layouts have lower performance, and for large geometries only (except the pointer tile layout for single-precision data). However, the measured performance was erratic and strongly dependent on geometry size for some of the analysed layouts, for example, the bitmask node layout, single-precision data, and geometries containing between 1024^2 and 2048^2 nodes.

Performance for Sparse Geometries

Performance for sparse geometries is measured in the same way as for dense ones, but only non-solid nodes are taken into account in the performance calculation. Thus, for each sparse geometry, we define its *porosity*

$$\phi = \frac{n_{\text{non-solid nodes}}}{n_{\text{all nodes}}} \tag{8}$$

which determines what proportion of all nodes are involved in the LBM calculations. We treat all non-solid nodes as computational because LBM implementation is bandwidth-bound on our system and, even for the bounce-back boundary nodes, which do not require computations, we need to read and write f_i functions from/to memory.

The performance for sparse geometries was measured for square geometries with 4096^2 nodes and different porosities $\phi \geq 0.1$. The large geometry size was chosen to keep at least 10^6 non-solid nodes, even for the lowest porosity. To obtain the required ϕ, the geometry was filled with solid, circle obstacles. We used two different arrangements of obstacles: a regular array and a random placement, with examples shown in Figure 7. The regular array contains a mesh of 8×8 circles, of which the radius depends on the required ϕ. We did not use regular arrays for $\phi < 0.3$ because, in such cases, all geometry walls are filled with solid nodes from overlapping circles. For the random placement, the geometry was filled with randomly placed circles. The radii of circles were also randomly chosen from $r \in [8, 256]$ nodes.

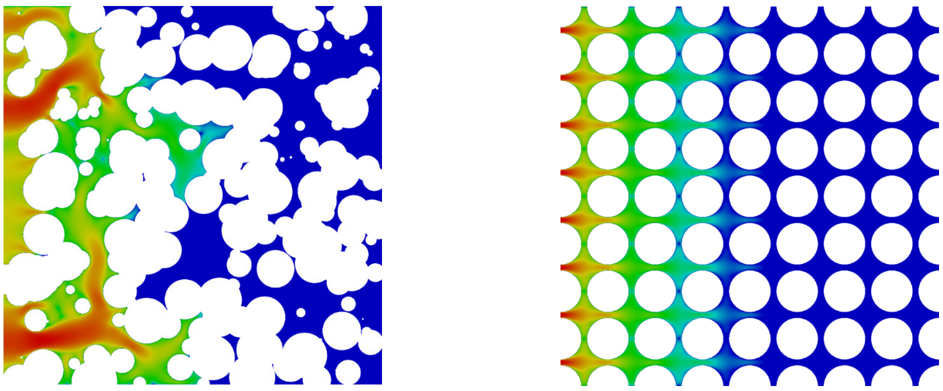
Figure 7. Examples of sparse geometries with 4096^2 nodes and porosity $\phi = 0.4$. Pressure boundary conditions were set on the inlet (**left** wall, $\rho = 1.016$) and outlet (**right** wall, $\rho = 1.0$), bounce back on the top and bottom walls and on borders of obstacles, viscosity $\nu = 0.5$. Colours correspond to the velocity magnitude after 10^6 time steps; a logarithmic scale is used. Calculations took about half an hour per geometry.

To estimate overheads for computations on sparse geometries, we define the *sparse computational efficiency*

$$\eta_P = \frac{P(\text{sparse geometry})}{P(\text{dense geometry})}, \tag{9}$$

where $P(\text{sparse geometry})$ and $P(\text{dense geometry})$ denote measured performances for the same data layout and geometry size. It should be noted that η_P does not take into account that the number of computational nodes in the sparse geometry is lower than in the dense one which may have an additional impact on performance. However, for dense data layouts, the memory is allocated for all nodes regardless of node type. Thus, the definition in Equation (9) seems reasonable.

The obtained performance results are shown in Figure 8. As can be seen, for all layouts but tile, the performance gradually drops with porosity. This performance decrease shows that the geometry sparsity introduces overheads, for example, creates redundant memory traffic caused by neighbouring solid and non-solid nodes, which are visible even after using data layouts (e.g., bitmasked) eliminating explicit operations for solid nodes.It should also be noted that the performance depends not only on porosity but also on the placement of solid and non-solid nodes. For randomly placed obstacles, the measured performance was slightly higher than for the regular placement, because randomly placed circles formed large, solid areas that minimised the interlacing in memory of the data for solid and non-solid nodes. Only for the tile layout, the performance was practically constant for porosities $\phi \geq 0.4$, which suggests that for the tile layout, other factors limit performance, as shown in Figure 6 for large geometry.

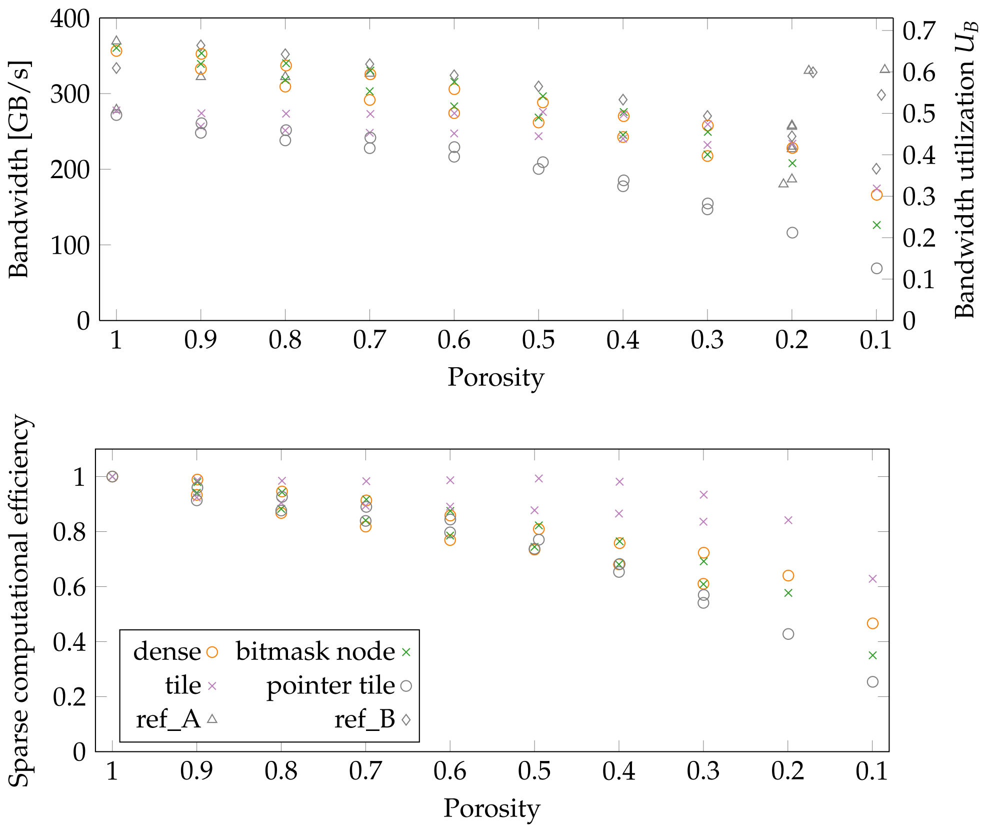

Figure 8. Performance of GPU simulations (**top**) and sparse performance efficiency (**bottom**) for single-precision numbers, different data layouts, and sparse geometries with 4096^2 nodes and different porosities. For a given data layout, the lower sequence of points shows performance for regularly placed circles, and the upper points correspond to performance for randomly placed circles. Reference performance for tile-based implementations optimised for sparse geometries is adapted from [16] (ref_A) and [17] (ref_B), accordingly. Performance for implementation from [17] is for GTX Titan Xp, D3Q19 lattice arrangement and single-precision numbers, whereas work [16] uses double-precision, both D2Q9 and D3Q19 lattices, and different GPU (GTX TITAN) thus positions of markers (triangles) result from U_B only.

The lowest performance was observed for the pointer tile layout. For porosity 0.1, the performance dropped to 25% of performance for dense geometry. However, in contrast to other layouts, performance for the pointer tile layout has the lowest differences between geometries with regularly and randomly placed obstacles. It may suggest additional overheads not connected with a sparsity of geometry, although more detailed research is needed.

Figure 8 also contains reference results from [16,17] for high-performance, tile-based GPU implementations optimised for sparse geometries. In many cases, the tile-based implementations with low-level optimisations outperform all Taichi versions, but the presented values should be compared carefully. Performance of implementation from [17] was measured for similar geometries (randomly arranged spheres) and the same data type and GPU as in the current work, but for a different lattice arrangement (D3Q19). In addition, two sparse geometries used in [17] , cerebral aneurysm and aorta with coarctation, resulted in much higher performance than randomly arranged spheres due to high spatial locality. Results for these geometries are shown in the right top corner in Figure 8. Work [16] used different GPU, double-precision numbers, and a limited set of geometries, including high-performance cerebral aneurysm and aorta with coarctation. Moreover, the performance was measured for two different lattice arrangements, D2Q9 and D3Q19, and significantly higher values were observed for D3Q19. Geometries with porosity close to 0.2, and a single dense geometry, used D2Q9, whereas geometries with porosities $\phi \geq 0.7$ were three-dimensional. Additionally, the implementation from [16] does not use memory layout optimisations presented in [17] , thus its performance is lower, as shown in Figure 8 for $\phi \geq 0.7$.

4.2.3. Memory Usage

Memory usage is difficult to measure for the Taichi language since it internally pre-allocates and manages GPU memory. However, we observed that, for the dense layout, the maximum size of allocated arrays was very similar for raw CUDA and Taichi implementations (two arrays containing about 720×2^{20} 64-bit elements). After exceeding this limit, we observed excessive page faults and a significant performance decrease. When we turned off the unified memory support, then memory allocation errors appeared sporadically. Although we did not conduct an in-depth analysis, we did not find any serious problems; thus, it seems that Taichi effectively manages memory and introduces minimal overhead only.

5. Conclusions

In this work, we presented the implementation of the lattice–Boltzmann solver in Taichi, the interpreted, data-oriented language which decouples computations and data arrangement in memory. We showed that, although sparse data layouts provided by Taichi bring significant overheads when used on a fine-grained level during simple data copying, it was possible to a design high-performance code of the lattice–Boltzmann solver for non-trivial cases. Four data layouts were tested: the dense layout which is a simple, multidimensional array; the tile layout with data arranged in square tiles containing 16^2 neighbouring nodes; the bitmask node layout where single nodes could be masked; and the pointer tile layout allowing allocation of memory per single tiles but at the cost of additional, indirect addressing. An additional advantage of the presented solution is short and simple code (about 1000 source lines) which is freely available in the hope that it can be a practical basis for further experiments due to its low complexity and high performance.

The obtained performance is comparable to the best existing implementations but strongly dependent on used data layouts. For the dense layout, we achieved up to about 70% of peak memory bandwidth available on GTX Titan Xp GPU, which corresponds to 5.37 GLUPS for single- and 2.65 GLUPS for double-precision computations. Slightly lower performance, but still up to over 67% of peak memory bandwidth, was observed for other dense data layouts, the tile and the bitmask node. The lowest performance, up to 54% of the peak, was obtained for the pointer tile layout and single-precision computations. Thus, Taichi implementations of LBM for dense geometries should have the highest performance with the simple, dense layout.

For sparse geometries, the layout resulting in the highest performance depends on geometry sparsity. Best performance for low porosities $\phi \leq 0.3$ was observed for the tile and the dense layouts, although it was still about two times slower than for dense geometries. Geometries with higher porosities were processed the fastest with the bitmask node and the dense layouts. The pointer tile layout has the lowest performance (about two times lower than the tile layout for porosity 0.1) but, in contrary to the other layouts, allows saving of memory by skipping data for some solid nodes.

Apart from data layouts available in Taichi, other factors may also have a significant impact on the code performance, which sometimes was difficult to predict. In the presented measurements, we observed that both geometry size and placement of solid and non-solid nodes has a visible impact on performance. Full performance requires geometries containing at least 10^6 nodes, but such behaviour was also reported in other papers about LBM implementations on GPU. However, we observed uncommon performance drops for the tile layout and large geometries, and significant performance limitations for the pointer tile layout for single-precision data. We have not found the reason yet and believe that significantly more thorough studies are needed to analyse overheads introduced by the Taichi language and its internal architecture.

The source code in Taichi is clean and concise, but we observed a few limitations. For example, we did not find a way for passing a static argument to function and use it as a compile-time constant index, thus we had to inline some operations manually to enable compile-time optimisations. Moreover, it is difficult to control register usage per kernel, and setting the number of CUDA threads per block is limited. For async mode, we sometimes observed problems with stability and code profiling by the NVIDIA Visual Profiler.

One of the issues was also the long compilation time. Although an additional time required for runtime compilation is typical for interpreted languages, it should be reported, especially that the presented code takes more than 20 s to compile. Comparing this with a few seconds needed to obtain the stationary solution for small meshes (up to 256^2 nodes), the kernel compilation enlarges the simulation time by order of magnitude. We suspect that maybe some form of precompiled kernels could significantly decrease the time for short simulations. It should be noted that we used only one simple collision model and a reduced set of boundary conditions. For more complex computational models or universal kernels with support for many different collision models, the time required to generate kernel code can be longer.

In many situations, the compilation time has a low impact on total performance or even can be neglected, especially for long, complex simulations taking hours or more, or when the time for simulation is not strictly constrained. However, there are some edge cases where the compilation time may be of importance. One of the examples may be real-time systems where simulations are required to predict the results of available actions, and these predictions must be available under time constraints. Another example may be the calibration of a numerical model, where many simulations with different settings are run to search the given parameter space. The performance measurements presented in this work can be considered as some form of parameter space searching where we analyse the impact of different parameters on the performance of the simulation. Moreover, the compilation time can decrease performance even for relatively complex cases, for which GPU implementation allows quick finishing of computations. For example, we have shown that 10^6 time steps, for geometry containing 1024^2 nodes, can be computed in few minutes on a single GPU with Pascal architecture. Usage of newer GPUs, Volta or Ampere, could decrease simulation time further—memory bandwidth for the A100 GPU is about three times higher than for the GPU used in this work, thus we may expect a similar decrease in simulation time.

Future work includes searching for methods that allow the use of other optimisation techniques used for LBM implementations. We are also planning the implementation of more complex collision models, boundary conditions, and support for three-dimensional geometries, although this may require the new design of code due to a larger amount of data per node which can increase register pressure.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The source code presented in this study is openly available in Github at https://github.com/tadeusz-tomczak/tilb (accessed on 30 August 20201).

Acknowledgments: The authors are very grateful for the reviewer's valuable comments, which improved the manuscript, and for the support from NVIDIA Corporation for providing them with the Titan Xp GPU used in this research.

Conflicts of Interest: The author declares no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BGK	Bhatnagar–Gross–Krook operator
CFD	Computational fluid dynamics
CPU	Central processing unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic random-access memory
GPU	Graphics processing unit
GUI	Graphical user interface
ILP	Instruction-level parallelism
LBM	Lattice–Boltzmann method
LUPS	Lattice updates per second
PDF	Particle distribution function
SI	Système international (d'unités)
SIMD	Single instruction, multiple data
SNode	Structural node
TLB	Translation lookaside buffer

References

1. Tölke, J. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Comput. Vis. Sci.* **2008**, *13*, 29–39. [CrossRef]
2. Tölke, J.; Krafczyk, M. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *Int. J. Comput. Fluid Dyn.* **2008**, *22*, 443–456. [CrossRef]
3. Mawson, M.J.; Revell, A.J. Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs. *Comput. Phys. Commun.* **2014**, *185*, 2566–2574. [CrossRef]
4. Januszewski, M.; Kostur, M. Sailfish: A flexible multi-GPU implementation of the lattice Boltzmann method. *Comput. Phys. Commun.* **2014**, *185*, 2350–2368. [CrossRef]
5. Schulz, M.; Krafczyk, M.; Tölke, J.; Rank, E. Parallelization Strategies and Efficiency of CFD Computations in Complex Geometries Using Lattice Boltzmann Methods on High-Performance Computers. In *High Performance Scientific and Engineering Computing, Proceedings of the 3rd International FORTWIHR Conference on HPSEC, Erlangen, Germany, 12–14 March 2001*; Breuer, M., Durst, F., Zenger, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; pp. 115–122.
6. Bernaschi, M.; Melchionna, S.; Succi, S.; Fyta, M.; Kaxiras, E.; Sircar, J. MUPHY: A parallel MUlti PHYsics/scale code for high performance bio-fluidic simulations. *Comput. Phys. Commun.* **2009**, *180*, 1495–1502. [CrossRef]
7. Zeiser, T.; Hager, G.; Wellein, G. Benchmark analysis and application results for lattice Boltzmann simulations on NEC SX vector and Intel Nehalem systems. *Parallel Process. Lett.* **2009**, *19*, 491–511. [CrossRef]
8. Martys, N.S.; Hagedorn, J.G. Multiscale modeling of fluid transport in heterogeneous materials using discrete Boltzmann methods. *Mater. Struct.* **2002**, *35*, 650–658. [CrossRef]
9. Nita, C.; Itu, L.; Suciu, C. GPU accelerated blood flow computation using the lattice Boltzmann method. In Proceedings of the High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 10–12 September 2013; pp. 1–6.
10. Mazzeo, M.; Coveney, P. HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. *Comput. Phys. Commun.* **2008**, *178*, 894–914. [CrossRef]
11. Parmigiani, A.; Huber, C.; Bachmann, O.; Chopard, B. Pore-scale mass and reactant transport in multiphase porous media flows. *J. Fluid Mech.* **2011**, *686*, 40–76. [CrossRef]
12. Krause, M.; Kummerländer, A.; Avis, S.; Kusumaatmaja, H.; Dapelo, D.; Klemens, F.; Gaedtke, M.; Hafen, N.; Mink, A.; Trunk, R.; et al. OpenLB—Open source lattice Boltzmann code. *Comput. Math. Appl.* **2021**, *81*, 258–288. [CrossRef]
13. Hasert, M.; Masilamani, K.; Zimny, S.; Klimach, H.; Qi, J.; Bernsdorf, J.; Roller, S. Complex fluid simulations with the parallel tree-based Lattice Boltzmann solver Musubi. *J. Comput. Sci.* **2014**, *5*, 784–794. [CrossRef]

14. Feichtinger, C.; Donath, S.; Köstler, H.; Götz, J.; Rüde, U. WaLBerla: HPC software design for computational engineering simulations. *J. Comput. Sci.* **2011**, *2*, 105–112. [CrossRef]
15. Tomczak, T. Lattice Boltzmann Method for Sparse Geometries: Theory and Implementation. In *Analysis and Applications of Lattice Boltzmann Simulations*; Valero-Lara, P., Ed.; IGI Global: Hershey, PA, USA, 2018; Chapter 5, pp. 152–187.
16. Tomczak, T.; Szafran, R.G. Sparse Geometries Handling in Lattice Boltzmann Method Implementation for Graphic Processors. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *29*, 1865–1878. [CrossRef]
17. Tomczak, T.; Szafran, R.G. A new GPU implementation for lattice-Boltzmann simulations on sparse geometries. *Comput. Phys. Commun.* **2019**, *235*, 258–278. [CrossRef]
18. Hu, Y.; Li, T.M.; Anderson, L.; Ragan-Kelley, J.; Durand, F. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* **2019**, *38*, 201:1–201:16. [CrossRef]
19. Wang, Z. LBM_Taichi. Available online: https://github.com/hietwll/LBM_Taichi (accessed on 30 August 2021).
20. Taichi Programming Language. Available online: https://github.com/taichi-dev/taichi (accessed on 30 August 2021).
21. Krüger, T.; Kusumaatmaja, H.; Kuzmin, A.; Shardt, O.; Silva, G.; Viggen, E. *The Lattice Boltzmann Method: Principles and Practice*; Graduate Texts in Physics; Springer International Publishing: Cham, Switzerland, 2017.
22. Guo, Z.; Shu, C. *Advances in Computational Fluid Dynamics—Lattice Boltzmann Method and Its Applications in Engineering*; World Scientific Publishing Co. Pte. Ltd: Singapore, 2013; Volume 3.
23. Mohamad, A.A. *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes*; Springer: London, UK, 2019; Volume 70.
24. Bhatnagar, P.L.; Gross, E.P.; Krook, M. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Phys. Rev.* **1954**, *94*, 511–525. [CrossRef]
25. Rinaldi, P.; Dari, E.; Vénere, M.; Clausse, A. A Lattice-Boltzmann solver for 3D fluid simulation on GPU. *Simul. Modell. Pract. Theory* **2012**, *25*, 163–171. [CrossRef]
26. Bailey, P.; Myre, J.; Walsh, S.D.; Lilja, D.J.; Saar, M.O. Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors. In Proceedings of the 2009 International Conference on Parallel Processing, Vienna, Austria, 22–25 September 2009; pp. 550–557.
27. Zou, Q.; He, X. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Phys. Fluids* **1997**, *9*, 1591–1598. [CrossRef]
28. Ghia, U.; Ghia, K.; Shin, C. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *J. Comput. Phys.* **1982**, *48*, 387–411. [CrossRef]
29. Ayachit, U. *The ParaView Guide: A Parallel Visualization Application*; Kitware: Clifton Park, NY, USA, 2015.