Article

Cross-Platform GPU-Based Implementation of Lattice Boltzmann Method Solver Using ArrayFire Library

Michal Takáč and Ivo Petráš *

Faculty BERG, Technical University of Košice, Némcevej 3, 042 00 Košice, Slovakia; michal.takac@tuke.sk

* Correspondence: ivo.petrash@tuke.sk; Tel.: +421-55-602-5194

Abstract: This paper deals with the design and implementation of cross-platform, D2Q9-BGK and D3Q27-MRT, lattice Boltzmann method solver for 2D and 3D flows developed with ArrayFire library for high-performance computing. The solver leverages ArrayFire's just-in-time compilation engine for compiling high-level code into optimized kernels for both CUDA and OpenCL GPU backends. We also provide C++ and Rust implementations and show that it is possible to produce fast cross-platform lattice Boltzmann method simulations with minimal code, effectively less than 90 lines of code. An illustrative benchmarks (lid-driven cavity and Kármán vortex street) for single and double precision floating-point simulations on 4 different GPUs are provided.

Dataset License: MIT

Keywords: lattice Boltzmann method (LBM); computational fluid dynamics (CFD); parallel computing; graphics processing unit (GPU) computing; ArrayFire library; numerical analysis

Citation: Takáč, M.; Petráš, I. Cross-Platform GPU-Based Implementation of Lattice Boltzmann Method Solver Using ArrayFire Library. *Mathematics* **2021**, *9*, 1793. <https://doi.org/10.3390/math9151793>

Academic Editor: Panagiota Tsompanopoulou

Received: 31 May 2021
Accepted: 26 July 2021
Published: 28 July 2021

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

Copyright: © 2021 by the authors. 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

Popularity of the lattice Boltzmann method (LBM) has steadily grown since its inception from lattice gas automata [1] more than three decades ago. The lattice gas automata are a type of cellular automaton used to simulate fluid flows and they were the precursor to the LBM. From lattice gas automata it is possible to derive the macroscopic Navier–Stokes equations. A disadvantage of the lattice gas automata method is the statistical noise. Another problem is the difficulty in expanding the model to 3D case. Because of these reasons the LBM started to rise in early 1990s as an alternative procedure [2]. As a mesoscopic method, filling the gap between macroscopic Navier–Stokes solvers and microscopic molecular dynamics, it has been an important tool for numerical simulations of multi-component, multiphase flows [3–5], flows in porous media [6,7], turbulent flows [8], and lately for more complex flows of fluids, as for instance, Bose–Einstein condensate [9], interaction of (2+1)-dimensional solitons [10], or modeling of viscous quasi-incompressible flows [11]. Thanks to the computational simplicity of LBM and its spatial and temporal locality, it is naturally suited to parallel computing [12].

Recently, the increase in computational power and advances in general-purpose computing on GPUs (GPGPU) opened the door for real-time and interactive computational fluid dynamics (CFD) simulations [13–17]. Together with the performance and speed of the LBM method, it is now possible to compute more than several hundreds of iterations per second which makes an interaction with the simulation in progress possible [18]. Getting instant feedback according to the change of various parameters in simulation gives researchers the ability to iterate faster toward the creation of accurate model, better understanding of underlying phenomena, or employing simulation within the control of industrial systems. It is, therefore, desirable to push the limits of execution speed of LBM simulations. Developers have to be careful with the memory limitations, even though GPUs provide high memory bandwidth, as LBM algorithms tend to consume large amounts of memory for storing the data. GPU architecture is designed for high

data throughput thanks to the combination of single instruction and multiple data (SIMD) execution model and multithreading, called single instruction and multiple threads (SIMT). With the parallel nature of the LBM algorithm, CFD simulations that use it can achieve high speeds not just on high-performance computing (HPC) systems but also PC workstations with a single GPU. For applications of real-time or online interactive visualization of the running simulation, getting to high frame rates is easier to achieve on such workstation PCs because of the high bandwidth of the PCIe slot. On networked HPC systems, the transfer speeds are limited by network throughput and higher latency. Therefore, transferring data from GPU on the HPC system back to the PC client for visualization is much slower [19]. In this study, we use GPUs that have between 70 and 900 GB/s theoretical maximum memory bandwidth.

To attain more computational power from GPUs, algorithm optimization techniques for parallel computing need to be considered. Compiled code needs to be vectorized and multithreaded to leverage parallel capabilities in massively parallel architectures [13]. This is typically done by using specific compilation commands to automatically vectorize code (NVC++ compiler from NVIDIA with `stdpar`), writing GPU-specific code (compute kernels) with frameworks like CUDA and OpenCL, or compiling both CUDA and OpenCL from the same code without specialized compiler directives using cross-platform library like ArrayFire. In multi-GPU setups, a message passing interface (e.g., MPI, OpenMP) is usually employed.

In this paper, the cross-platform solver for LBM simulations is presented, discussing the advantages of using ArrayFire library [20] for writing optimized CUDA and OpenCL software without worrying about hand-tuned optimization of custom-made kernels for specific platforms.

The present work is organized as follows. Section 1 introduces the problem. In Section 2, we present the computational framework of LBM with the structure of our cross-platform LBM solver, discussing the advantages and disadvantages of the computational ArrayFire library. In Section 3, we provide a performance analysis of implemented solver, benchmarking it on consumer-grade GPUs with well-known academic test cases. In Section 4, we provide a discussion about the next steps in future research building on the foundations from this study. Section 5 concludes this article with some comments.

2. Materials and Methods

The LBM solver with D2Q9 stencil (two-dimensional computational domain with fluid particle distributions moving in 9 velocity directions to neighboring nodes) with single-relaxation time and D3Q27 stencil (three-dimensions with 27 velocity directions) with multiple-relaxation time that uses GPUs for high-performance computation were implemented. Various optimization techniques exist to parallelize and optimize LBM algorithms [15,18,21–24]. We tried to keep manual optimizations in code to a minimum to see how performant the resulting ArrayFire application is.

The same LBM solver was implemented in both C++ and Rust to study the ease of development across different languages for which ArrayFire wrappers exist. We picked Rust instead of Python as a second language because it is a modern, memory-safe language focused on performance, and the feature-parity with C++ API was better at the time of development.

In this section, we cover the basic theory behind LBM, followed by a description of the implementation of the solver algorithms on GPU, the pros and cons of using the ArrayFire library for the development of parallel programs and how the simulations results can be visualized in real-time with Forge library (included in ArrayFire) while the simulation is running.

2.1. Lattice Boltzmann Method

The lattice Boltzmann method originally grew out of the lattice gas automata [2,25]. It is positioned in the middle between Eulerian and Lagrangian methods for solving fluid flow problems. Instead of calculating the properties of individual particles, the particle

distribution function (PDF) is used for describing the distribution of particles that is computed for each node in the discretized domain. As we mentioned earlier, each node needs only its neighbors for the actual computation, allowing for good parallelization. A collision of particle distributions is described by Ω operator, that states the rate of change of PDF (denoted as f) is equal to the rate of collision in the limit of $dt \rightarrow 0$:

$$\frac{df}{dt} = \Omega(f). \quad (1)$$

The collision operator Ω is difficult to solve. It has been simplified by the work of Bhatnagar, Gross, and Krook (BGK) [26], that introduced the BGK operator defined as

$$\Omega_i = \frac{1}{\tau} (f_i^{eq} - f_i), \quad (2)$$

where f_i^{eq} is an PDF in an equilibrium state of the system obtained by Taylor expansion of the Maxwell–Boltzmann equilibrium function. The relaxation parameter τ is the reciprocal that presents a time in which the systems relaxes towards the equilibrium.

The lattice Boltzmann equation (LBE) in its discrete form, the fundamental part of the LBM, is obtained by discretization of the velocity space of the Boltzmann equation into a finite number of discrete velocities e_i , $i = 0, 1, \dots, Q$, with $Q = 9$ for D2Q9 and $Q = 27$ for D3Q27 stencil, respectively (see Figure 1).

Figure 1. Lattice stencils. (a) D2Q9; (b) D3Q27.

The LBE can be stated as

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) - \frac{1}{\tau} [f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)], \quad (3)$$

where $f_i(\mathbf{x}, t)$ denotes the individual direction of the PDF at each lattice point in particular time and $f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t)$ is equal to resulting PDF for all neighboring nodes in the next iteration step. In other words, it represents the probability of finding a particle at position $\mathbf{x}$ and time t with velocity $\mathbf{e}_i$. Necessary criterion for stability is that physical information should not travel faster than fastest speed supported by the lattice [25]. Discrete velocities $\mathbf{e}_i$ in D2Q9 model can be expressed in an array as

$$\mathbf{e}_i = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{bmatrix} \quad (4)$$

for the single node in 2D grid and for D3Q27 the array is extended to accommodate 27 velocities moving in 3D grid

$$\mathbf{e}_i = \begin{bmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 \end{bmatrix}$$

Macroscopic quantities are obtained from hydrodynamic moments of the distribution function (see Equation (5)), where ρ is the density and $\mathbf{u}$ is the velocity defined as

$$\rho = \sum_{i=0}^Q f_i, \quad \rho \mathbf{u} = \sum_{i=0}^Q \mathbf{e}_i f_i. \quad (5)$$

Equilibrium distribution function f^{eq} can be expressed by performing a Hermite expansion of the Boltzmann equilibrium function as

$$f_i^{eq} = \omega_i \rho \left(1 + 3 \frac{\mathbf{e}_i \cdot \mathbf{u}}{c_s} + \frac{9}{2} \frac{(\mathbf{e}_i \cdot \mathbf{u})^2}{c_s^2} - \frac{3}{2} \frac{u^2}{c_s^2} \right), \quad (6)$$

where c_s is the dimensionless speed of sound within the lattice, usually set to $c_s = \frac{1}{\sqrt{3}}$ and ω_i denotes different weights (7) for discrete velocity in D2Q9 stencil as follows

$$\omega_0 = 4/9, \quad \omega_{1,2,3,4} = 1/9, \quad \omega_{5,6,7,8} = 1/36. \quad (7)$$

With a proper set of discrete velocities, the LBE recovers the incompressible Navier–Stokes equations by the Chapman–Enskog expansion. For the flows within the incompressible limit, assumptions, such as low-Mach number $\mathbf{u}/c_s$ and variations in density of order $\mathcal{O}(M^2)$, where M is a transformation matrix, has to be made.

Two general steps of the LBM solver involve solving Equation (8) for collision and Equation (9) for streaming in each iteration, which are defined as follows:

$$f_i(\mathbf{x}, t + \Delta t) = f_i(\mathbf{x}, t) - \frac{1}{\tau} [f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)]. \quad (8)$$

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t + \Delta t). \quad (9)$$

To overcome difficulties of numerical instability in applying the 3D LBM method, the multiple-relaxation-time (MRT) scheme is useful to stabilize the solution and to obtain satisfactory results. The MRT model allows for independent tuning of the relaxation times for each physical process [8]. It is, therefore, natural to extend current work to include MRT. In this study we implemented non-orthogonal MRT-LBM in D3Q27 because it simplifies the transformation between the discrete velocity space and the moment space [5].

The collision operator in MRT-LBM is defined as

$$\Omega_{MRT} = M^{-1} S M, \quad (10)$$

where S is a diagonal relaxation matrix, constructed by the process described in [5]. The collision step is performed in moment space. Formally it can be defined as

$$m^* = m - S(m - m^{eq}), \quad (11)$$

where $m = Mf$ and $m^{eq} = Mf^{eq}$. After the collision step, the distribution function is reconstructed by $f^* = M^{-1}m^*$ and used in the streaming step as usual

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i^*(\mathbf{x}, t + \Delta t). \quad (12)$$

For the boundary condition, we implemented a simple no-slip boundary known as bounce-back, which effectively reverses the direction of f_i [12]. In places like input and

output of simulated pipe for Kármán vortex test case, we implemented periodic boundary conditions [27].

2.2. Using GPU for CFD Simulations

There has been an increase in studies focused on optimizing the execution speed of LBM algorithms, after the idea of using GPGPUs for CFD simulations started gaining traction more than a decade ago. It has been bolstered by the LBM's advantage in the locality of computations since only the values from neighboring nodes are needed.

In this space, the most used APIs for programming GPUs are CUDA and OpenCL. CUDA is a proprietary API used to program NVIDIA GPUs [28]. OpenCL is an open standard that supports different hardware from various vendors on the market [29].

Recently, multiple 2D and 3D LBM simulations used CUDA or OpenCL for their parallel implementation targeting GPUs [13–16,30–34]. There is an increasing amount of studies on memory access efficiency and optimization techniques [22,35]. However, to accomplish near-optimal performance, it is extraordinary amount of work. Programming software for GPGPU is still very difficult [36]. Developers need to optimize for specific hardware and, therefore, have to know each architecture thoroughly. For this reason, such hardware-specific implementations in cross-platform code tend to become very complex.

ArrayFire library can help by automatically leveraging the best hardware features available on multiple architectures, hiding the hardware-specific optimizations. Developers can write code in a high-level language like C++, Rust, or Python (for which the ArrayFire library wrapper exists) and have it compiled for CPU, GPU, or other accelerators like FPGA.

2.3. Compiling to CUDA and OpenCL from One Codebase

CUDA and OpenCL APIs differ from each other. They can be considered as extensions of the C/C++ language and require significant experience in low-level C/C++ programming. To write optimized, parallel software, developers need to employ different techniques, specific to the API they choose, which adds substantial overhead when trying to port one to another in case of a need. In heterogeneous computing systems, writing optimized cross-platform code for different GPUs means writing multiple hardware-specific kernels in CUDA and OpenCL.

In this work, a high-performance, parallel computing ArrayFire library has been used to significantly simplify programming for GPUs. Its easy-to-use API provides high-level abstractions in the form of hundreds of functions. They are automatically converted to optimized, fast GPU kernels, utilizing just-in-time (JIT) compilation and lazy evaluation [37]. ArrayFire's high-level object construct called Array is a data structure that acts as a container that represents memory stored on the device. On top of it, ArrayFire provides abstractions in the form of Unified Backend for working with different computational backends. This way it is possible to switch to CPU, GPU, FPGA, or another type of accelerator at runtime (see Listing 1), and permits programmers to write massively parallel applications in a high-level language with a much lower number of lines of code [20]. Arrays (or matrices) of up to 4 dimensions can be created.

Listing 1: C++ code for setting different computing backends.

```
#include <arrayfire.h>
int main()
{
    af::setBackend(AF_BACKEND_CUDA);
    // af::setBackend(AF_BACKEND_OPENCL);
    // af::setBackend(AF_BACKEND_CPU);
    return 0;
}
```

2.4. Employed Optimizations in LBM Solver

In the following section, we describe simple optimizations employed in our LBM-based solver. To achieve high throughput on different parallel architectures, a large portion of time-consuming, hardware-specific, low-level optimizations are done automatically by ArrayFire. The underlying JIT compilation engine converts expressions into the smallest number of CUDA or OpenCL kernels, but to decrease the number of kernel calls and unnecessary global memory operations, it tries to merge cooperating expressions into a single kernel. However, not everything can be optimized automatically. There are some specifics between LBM algorithms where some optimizations have a large impact on the performance of the simulation, namely coalesced memory writes resulting from better data organization scheme, removing branch divergence, improvement of cache locality, and thread parallelism.

2.4.1. Data Organization

To ensure the most efficient memory throughput when programming for GPU is to ensure that memory access is coalesced [22]. There are two common patterns for data organization: array of structures (AoS) and structure of arrays (SoA). Therefore, if we consider a 1D array, in AoS, all discrete velocities in each node occupy consecutive elements of the array, and in SoA, the value of one discrete velocity direction from all nodes is arranged consecutively in memory, then the next direction in all nodes and so on. In a GPU-based LBM algorithm, it is beneficial to store values of distribution functions for each node in the computational domain represented by a grid in 1-dimensional array. For D2Q9 and D3Q27 stencils, this translates into storing single velocity direction for 2D grid represented by $N_x * N_y$ or 3D grid represented by $N_x * N_y * N_z$ nodes at a time, consecutively 9-times for D2Q9 and 27-times for D3Q27, respectively. Using SoA is significantly faster than AoS [13,22]. This way the cache use is considerably improved [12].

To create a SoA structure with 1D array to store particle distributions in all directions along whole computational domain, we can set first dimension of array construct straightforwardly to the $n_x * n_y * n_z * Q$ number of elements, but for easier index creation further down the line, it is better to create 3D (or 4D in case of D3Q27) array initially and flatten it when used for heavy computation of actual simulation (see Listing 2).

Listing 2: Creating SoA structure representation of D2Q9 lattice with ArrayFire in C++.

```
unsigned nx = 64, ny = 64, dirs = 9;
// SoA 1D array
array f_1d_soa = constant(0, nx * ny * dirs);
// Array flattened to SoA 1D array
array f = constant(0, nx, ny, dirs);
array f_1d_soa = flat(f)
```

For streaming operation in LBM, we use ArrayFire's `shift` function for each direction of particle distributions. Instead of running this function on every iteration, we create two indices for access to "current" nodes and their neighboring nodes at the initialization phase of the solver and store them for the whole lifetime of the simulation. Streaming is then as easy as accessing particle distributions in a 1D array with neighbours index. In ArrayFire, indexing is also executed in parallel, but is not a part of a JIT compilation. Instead, it is a handwritten optimized kernel. Any JIT code that is fed to indexing is evaluated in a single kernel if possible (see Listing 3).

Listing 3: Creating “current” (or main) index and neighboring index.

```

unsigned nx = 64, ny = 64, dirs = 9;
unsigned total_nodes = nx * ny;
// Index of all directions except center point 0
array CI = (range(dim4(1,8),1)+1) * total_nodes;
// Neighboring nodes index
unsigned int nb_index_arr[8] = {2,3,0,1,6,7,4,5};
array nbidx(8, nb_index_arr);
array NBI = CI(span,nbidx);
dim4 dims = dim4(total_nodes*dirs);
array main_index = moddims(range(dims),nx,ny,dirs);
array nb_index = flat(stream(main_index));

```

The result of streamed distribution functions is stored temporarily without overwriting the previous ones (see Listing 4).

Listing 4: Streaming step.

```

array F_streamed = F(nb_index);

```

Targeting any of the discrete speeds directions in 1D array can be done by computing $[node_position] + [total_nodes * directions]$.

2.4.2. Removing Branch Divergence

When writing classical, imperative code, handling control flow is usually done by using if-else blocks, creating different possible branches. In multithreaded execution model like SIMT used in GPUs, the processor’s threads execute different paths of the control flow, leading to poor utilisation due to thread-specific control flow using masking [38]. Branch divergence is a major cause for performance degradation in GPGPU applications [39]. To keep the flow coherent for the processing threads, it is recommended to remove if-else blocks from the code (see Listing 5).

Listing 5: Pseudo-code showcasing the removal of branch divergence by removing if statement.

```

// With branch divergence
if (cell_type == "solid") {
  x = a;
} else {
  x = b;
}
// Without branch divergence
let is_solid = cell_type == "solid";
x = a * is_solid + b * (!is_solid);

```

In practical LBM application, branch divergence occurs when doing different computations on different types of the nodes in computational domain, e.g., “if fluid node, do computation, else do nothing”. Branch removal in the C++ version of ArrayFire applications is shown in Listing 6.

Listing 6: Example C++ code of removing branch divergence using ArrayFire.

```
// Node types (0 = solid, 1 = fluid)
unsigned types[] = {0,0,0,1,1,1};
array T(3, 3, types);

// original array
array A = randu(3, 3);
// part of the domain to be replaced
array FLUID = constant(1, 3, 3);
// new values
array B = randu(3, 3);
array cond = FLUID == T;
array out = A * (1 - cond) + cond * B;
```

In Rust version, the branching removal is achieved in the same manner, but at the end to use the condition, function select can be used (see Listing 7).

Listing 7: Example Rust code of removing branch divergence using ArrayFire.

```
let out = af::select(&A, &cond, &B);
```

In LBM simulations, we are also concerned with setting up boundary conditions. It is necessary to tell the solver which cells are solid (e.g., for doing bounce-back in some step down the line) and which are other types of fluids. For the simplest case, let us consider only two types of cells—solid and fluid. Boolean mask, in this case represented as integers (fluid as 0 and solid as 1), is instantiated in mask variable. Indexes of the solid nodes within computational domain can be easily found with where function as it is in Listing 8.

Listing 8: C++ code for constructiong the index of all solid cells in 2D grid (D2Q9) using ArrayFire.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
int nx = 400, ny = 100;
array mask = constant(0,nx,ny);
// Rectangle obstacle of size 2x20 cells
mask(seq(100,102),seq(40,60))) = 1;
mask(span,0) = 1; // Top wall
mask(span,end) = 1; // Bottom wall
// Get the indices of each solid cell
array solids = where(mask);
// ... rest of the code ...
return 0;
}
```

With the solid indices, it is very easy to set the boundary conditions at solid nodes back to zero after the streaming step (see Listing 9).

Listing 9: Boundary conditions at solid nodes.

```
UX(solids) = 0; // velocity in X-direction
UY(solids) = 0; // velocity in Y-direction
UZ(solids) = 0; // in D3Q27, velocity in Z-direction also
DENSITY(solids) = 0;
```


2.4.3. Pull Scheme

Most common algorithms for the streaming phase in LBM solvers use push and pull schemes [22,35]. In the push scheme, the streaming step occurs after the collision step, at which point the particle distribution values are written to neighboring nodes. This presents a misalignment of the memory locations, resulting in uncoalesced writes, degrading the performance significantly. On the other hand, in pull scheme, streaming step occurs before collision step, at which point the neighboring particle distribution values are gathered to the current nodes and then used for computations ending with collision step, after which the results are written directly to the current nodes. This way, the writes are coalesced in memory.

The idea behind preferring coalesced writes to GPU device memory is that the requests for values that are stored at memory addresses within 128-byte range are combined into one, which saves memory bandwidth. It is generally accepted that the cost of the uncoalesced reading is smaller than the cost of the uncoalesced writing [22].

2.5. Real-Time Visualization

The progress of scientific computations can be viewed in real-time thanks to the high-performance OpenGL visualization library called Forge (see Figure 2). It is developed by the same team behind ArrayFire and distributed together with their library for high-performance parallel computing. It is written specifically for use with GPU-accelerated applications as it does not require the expensive copying from GPU to CPU and back to GPU for rendering, but instead it builds on CUDA/OpenCL interoperability with OpenGL and allows for direct reading of the data on GPU [40]. Forge provides various plotting and visualization functions for 2D and 3D domains. For the current study, we only used 2D visualizations.

Practical scientific simulations for in-depth study of complex physical phenomena from real world, e.g., direct numerical simulation of cellular blood flow [32], requires higher accuracy. Instead of single-precision floating-point type (f32) computation, double-precision floating-point (f64) has to be chosen. In LBM context, this practically doubles the amount of memory needed. For real-time visualizations of results with this type of precision, they have to be converted to a single-precision floating-point for Forge to effectively work with the data. In ArrayFire (and generally in programming), function for this operation is called *cast* (see Listing 10).

Listing 10: Converting to single precision floating point for Forge visualization in C, C++ and Rust.

```
// C
af_array A_f64 = af_randu(100,100);
af_array B_f32;
cast(*B_f32, A_f64, f32);
// C++
array A_f64 = randu(100,100);
array B_f32 = A_f64.as(f32);
// Rust
let dims = af::Dim4::new(&[100, 100, 1, 1]);
let A_f64 = af::randu::<f64>(dims);
let B_f32 = A_f64.cast::<f32>();
```

In the current implementation, we used `image()` and `vector_field_2()` functions for visualizing density and velocity of on both lid-driven cavity (Figure 3) and Kármán vortex street test cases (Figure 4).

Figure 2. Differences between CPU-bound visualization with expensive copying bottleneck and high-performance GPU-bound visualization keeping full speeds of high-bandwidth of PCIe interface. (a) High-performance expensive visualization pipeline; (b) High-performance OpenGL visualization pipeline.

Figure 3. Lid-driven cavity test case at 128×128 resolution after 5000 iterations. (a) Visualization of lid-driven cavity simulation; (b) Velocity vector field.

Figure 4. Kármán vortex street (channel flow past circle-shaped obstacle) test case at 1000×300 resolution after 5000 iterations. (a) Visualization of Kármán vortex street simulation; (b) Velocity vector field (filtered for better presentation).

3. Results

For the current work, the LBM was applied to simulate incompressible steady and unsteady low Reynolds number flows in 2D and 3D lid-driven cavity using full bounce-back walls and 3D flow around confined circle also known as Kármán vortex street. These classic, academic tests were simulated in different grid densities going from coarse to denser, within memory limits of GPUs that were used.

Performance of LBM simulations is measured in million lattice updates per second (MLUPS), which is a standard unit of measurement within the LBM research community. The same metric is used for both single and double-precision floating-point operations in benchmarked simulations.

We also show different visualization functions from the Forge library that can be used to study fluid flows and other types of CFD simulations.

3.1. Hardware

The solver performance on 4 different GPUs with multiple architectures as it is shown in Table 1 was tested. The AMD Radeon R9 M370X is of mobile GPU type installed in laptops. In the current study, the AMD GPU was tested on the higher-end Macbook Pro 2015. The NVIDIA GTX 1070 is the average desktop GPU and its price at the time of writing this article is \$443.78 USD according to PassMark G3D Mark (a GPU benchmarking website). The NVIDIA RTX 3090 is a top-of-the-line, consumer-grade, enthusiast-level GPU with the price of \$2139.99 USD according to PassMark G3D Mark at the time of writing. We have also included NVIDIA GeForce RTX 2080 Ti.

Table 1. Hardware specifications for GPUs used for testing the simulation software described in this paper (taken from <https://www.techpowerup.com/gpu-specs/> accessed on 15 February 2021), where SM—streaming multiprocessor and CU—computing units.

	R9 M370X (AMD)	GTX 1070 (NVIDIA)	RTX 2080 Ti (NVIDIA)	RTX 3090 (NVIDIA)
Architecture	GCN 1.0	Pascal	Turing	Ampere
Number of cores (CU/SM)	640 (10 CU)	1920 (15 SM)	4352 (68 SM)	10,496 (82 SM)
Peak f32 performance (TFLOPS)	1.024	6.463	13.45	35.58
Memory clock (MHz)	1125	2002	1750	1219
Memory bandwidth (GB/s)	72.00	256.3	616.0	936.2
L1 cache size (KB)	16	48	64	128

3.2. Performance Analysis

Since ArrayFire allows for using not only GPU backends but also CPU, we added a CPU benchmarks executed on Intel Core i7 6800 K running at 3.40 GHz. Performance peaked at around 19 MLUPS and stayed the same between various domain sizes across both academic test cases.

Most of the GPU benchmarks were done using both CUDA and OpenCL backends, although differences between them were minimal. Therefore, in following tables and graphical representations of the data, we show MLUPS numbers solely from OpenCL benchmarks, since testing on this platform allowed us to perform benchmarks not only on NVIDIA hardware, but also on AMD GPU.

Single floating-point (f32) calculations perform significantly faster than double-precision (f64). The difference between f32 and f64 is the doubling of the problem data size, hence an increase in memory bandwidth is needed. This difference can be seen in D2Q9 lid-driven cavity and Kármán vortex test cases (see Figures 5–8).

For the 2D Kármán vortex test case, benchmarks showed similar results (see Figures 7 and 8). The performance spikes in so-called “warm-up” phase at the start of simulation were less significant in double-precision benchmarks than in lid-driven cavity test.

Figure 5. Single-precision performance analysis of 2D lid-driven cavity on D2Q9 stencil. (a) Performance on Radeon R9 M370X GPU; (b) Performance on GTX 1070 GPU; (c) Performance on RTX 2080 Ti GPU; (d) Performance on RTX 3090 GPU.

Figure 6. Double-precision performance analysis of 2D lid-driven cavity on D2Q9 stencil. (a) Performance on Radeon R9 M370X GPU; (b) Performance on GTX 1070 GPU; (c) Performance on RTX 2080 Ti GPU; (d) Performance on RTX 3090 GPU.

Figure 7. Single-precision performance analysis of 2D Kármán vortex on D2Q9 stencil. (a) Performance on Radeon R9 M370X GPU; (b) Performance on GTX 1070 GPU; (c) Performance on RTX 2080 Ti GPU; (d) Performance on RTX 3090 GPU.

Figure 8. Double-precision performance analysis of 2D Kármán vortex on D2Q9 stencil. (a) Performance on Radeon R9 M370X GPU; (b) Performance on GTX 1070 GPU; (c) Performance on RTX 2080 Ti GPU; (d) Performance on RTX 3090 GPU.

For the 3D LBM simulation with D3Q27 stencil and MRT, results are shown in Figure 9, where the computation speeds reaching up to 1500 MLUPS in single-precision and up to 490 MLUPS in double-precision benchmarks as it is depicted in Figure 10. The MRT is much more computationally intensive than BGK and with the 27 particle distributions to

track, the difference between D2Q9 and D3Q27 is significant. The memory bandwidth for D3Q27 lid-driven cavity tests is much higher, therefore, we were limited to grids under 128^3 in current implementation.

Figure 9. Single-precision performance analysis of 3D lid-driven cavity on D3Q27 stencil with multiple-relaxation time. (a) Performance on Radeon R9 M370X GPU; (b) Performance on GTX 1070 GPU; (c) Performance on RTX 2080 Ti GPU; (d) Performance on RTX 3090 GPU.

Figure 10. Double-precision performance analysis of 3D lid-driven cavity on D3Q27 stencil with multiple-relaxation time. (a) Performance on Radeon R9 M370X GPU; (b) Performance on GTX 1070 GPU; (c) Performance on RTX 2080 Ti GPU; (d) Performance on RTX 3090 GPU.

4. Discussion

Focus of this study was to test performance of ArrayFire library implementation of LBM codes on a single GPU. Some literature mentions simple extension to algorithms written with ArrayFire to add multiple GPU support in 4 lines of code [36]. Implementation like this is great for simple use cases and systems with the same type of GPU to prevent load-balancing issues. Proper multi-GPU support is planned in future versions of ArrayFire library that will be compatible with its Unified Backend convention. Therefore, we are eager to integrate multi-GPU support in future and test performance on heterogenous HPC systems when explicit multi-GPU will be ready.

In current implementations of D2Q9-BGK and D3Q27-MRT codes analyzed in this study, the lines of code are pretty low, comparable to MATLAB implementations of the same algorithms, but they run significantly faster without employing other than standard code changes required by ArrayFire library documentation (vectorize for-loops, use 1st dimension of matrices for coalesced writes, etc.). When we tried implementing codes for 3D LBM simulations using D3Q27 stencil, the performance was not good enough in comparison to state-of-art solvers in this category (e.g., Sailfish [30] or ELBE [41]). Therefore, for such case, more optimizations should be considered [35].

5. Conclusions

In this article, we presented a simple cross-platform implementation of LBM solver that can be used on variety of parallel accelerators (e.g., GPUs, CPUs, or FPGAs) by using ArrayFire, a high-performance parallel computing library (we used version 3.8.0). Reference implementation in C++ and ported the same code to Rust was created. We chose Rust because it has modern capabilities, is memory safe, and its performance is comparable to C/C++. In this way, we were able to produce sufficient LBM solver in under 150 lines of code, including real-time visualization code. Benchmarks were concluded for both C++ and Rust versions, resulting in identical performance. Data reported in this study are taken from the Rust version.

For the benchmarks, we analyzed two classical, academic test cases, the lid-driven cavity and Kármán vortex street (flow around the obstacle in pipe). Commonly used metric for measuring speed of LBM implementations, the MLUPS was employed. We benchmarked our LBM implementation on 4 different GPUs and one CPU. Between CPU and GPU, we saw from 4 to more than 300 times speed-up on various GPUs.

The code is available under MIT license and is freely published on GitHub. We also included the “condensed” implementation of Kármán vortex street in 87 lines of code C++ to showcase how little amount of code is needed to create high-performance, cross-platform simulation software with ArrayFire library.

Author Contributions: Conceptualization, methodology, software, data curation, visualization, original draft preparation, M.T.; supervision, formal analysis, funding acquisition, writing review, and editing, I.P. Both authors have read and agreed to the published version of the manuscript. Both authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Scientific Grant Agency (VEGA) under grant No. VEGA 1/0365/19 and by the Slovak Research and Development Agency (APVV) under the grants No. APVV-18-0526 and No. APVV-14-0892.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Code with the implementation of LBM algorithms for this article is provided freely at <https://github.com/michaltakac/lbm-arrayfire> (accessed on 16 June 2021). All the data from the benchmarks that were performed for this article are available at the same repository under the rust/benchmarks folder.

Acknowledgments: Author would like to thank ArrayFire team, especially John Melonakos and Pradeep Garigipati for building ArrayFire library, helping with technical questions and additional benchmarking on NVIDIA Tesla K20c GPU.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CUDA	Compute unified device architecture
OpenCL	Open computing language
LBM	Lattice Boltzmann method
LBE	Lattice Boltzmann equation
SIMD	Single instruction multiple data
SIMT	Single instruction multiple threads
HPC	High performance computing
PDF	Particle distribution function
MRT	Multiple relaxation time
MLUPS	Million lattice updates per second
AoS	Array of structures
SoA	Structure of arrays
FPGA	Field programmable gate array
GPU	Graphics processing unit
GPGPU	General-purpose computing on graphics processing unit
D2Q9	Two-dimensional lattice stencil with 9 discrete velocity directions in each node
D3Q27	Three-dimensional lattice stencil with 27 discrete velocity directions in each node
JIT	Just-in-time
API	Application programming interface

References

- Hardy, J.; Pomeau, Y.; de Pazzis, O. Time Evolution of a Two-Dimensional Classical Lattice System. *Phys. Rev. Lett.* **1973**, *31*, 276–279. [\[CrossRef\]](#)
- Higuera, F.J.; Jiménez, J. Boltzmann Approach to Lattice Gas Simulations. *Europhys. Lett. (EPL)* **1989**, *9*, 663–668. [\[CrossRef\]](#)
- Li, Q.; Luo, K.; Kang, Q.; He, Y.; Chen, Q.; Liu, Q. Lattice Boltzmann methods for multiphase flow and phase-change heat transfer. *Prog. Energy Combust. Sci.* **2016**, *52*, 62–105. [\[CrossRef\]](#)
- Nabavizadeh, S.; Eshraghi, M.; Felicelli, S. A Comparative Study of Multiphase Lattice Boltzmann Methods for Bubble-Dendrite Interaction during Solidification of Alloys. *Appl. Sci.* **2018**, *9*, 57. [\[CrossRef\]](#)
- Fei, L.; Du, J.; Luo, K.H.; Succi, S.; Lauricella, M.; Montessori, A.; Wang, Q. Modeling realistic multiphase flows using a non-orthogonal multiple-relaxation-time lattice Boltzmann method. *Phys. Fluids* **2019**, *31*, 042105. [\[CrossRef\]](#)
- Mattila, K.; Puurtinen, T.; Hyvälouma, J.; Surmas, R.; Myllys, M.; Turpeinen, T.; Robertsén, F.; Westerholm, J.; Timonen, J. A Prospect for Computing in Porous Materials Research: Very Large Fluid Flow Simulations. *J. Comput. Sci.* **2016**, *12*, 62–76. [\[CrossRef\]](#)
- Xin, F.; Li, X.F.; Xu, M.; Huai, X.L.; Cai, J.; Guo, Z.X. Simulation of gas exothermic chemical reaction in porous media reactor with lattice Boltzmann method. *J. Therm. Sci.* **2013**, *22*, 42–47. [\[CrossRef\]](#)
- Suga, K.; Kuwata, Y.; Takashima, K.; Chikasue, R. A D3Q27 multiple-relaxation-time lattice Boltzmann method for turbulent flows. *Comput. Math. Appl.* **2015**, *69*, 518–529. [\[CrossRef\]](#)
- Palpacelli, S.; Succi, S. Quantum lattice Boltzmann simulation of expanding Bose-Einstein condensates in random potentials. *Phys. Rev. E* **2008**, *77*, 066708. [\[CrossRef\]](#)
- Wang, H.; Yan, G. Lattice Boltzmann Model for the Interaction of (2+1)-Dimensional Solitons in Generalized Gross–Pitaevskii Equation. *Appl. Math. Model.* **2016**, *40*, 5139–5152. [\[CrossRef\]](#)
- Ilyin, O. Discrete Velocity Boltzmann Model for Quasi-Incompressible Hydrodynamics. *Mathematics* **2021**, *9*, 993. [\[CrossRef\]](#)
- Mawson, M. Interactive Fluid-Structure Interaction with Many-Core Accelerators. Ph.D. Thesis, The University of Manchester, Manchester, UK, 2014; p. 199.
- Delbosc, N.; Summers, J.; Khan, A.; Kapur, N.; Noakes, C. Optimized implementation of the Lattice Boltzmann Method on a graphics processing unit towards real-time fluid simulation. *Comput. Math. Appl.* **2014**, *67*, 462–475. [\[CrossRef\]](#)
- Delbosc, N. Real-Time Simulation of Indoor Air Flow Using the Lattice Boltzmann Method on Graphics Processing Unit. Ph.D. Thesis, University of Leeds, Leeds, UK, 2015; p. 261.

15. Harwood, A.R.; Revell, A.J. Parallelisation of an interactive lattice-Boltzmann method on an Android-powered mobile device. *Adv. Eng. Softw.* **2017**, *104*, 38–50. [\[CrossRef\]](#)
16. Koliha, N.; Janßen, C.; Rung, T. Towards Online Visualization and Interactive Monitoring of Real-Time CFD Simulations on Commodity Hardware. *Computation* **2015**, *3*, 444–478. [\[CrossRef\]](#)
17. Glessmer, M.; Janßen, C. Using an Interactive Lattice Boltzmann Solver in Fluid Mechanics Instruction. *Computation* **2017**, *5*, 35. [\[CrossRef\]](#)
18. Wang, M.; Ferey, N.; Bourdot, P.; Magoules, F. Interactive 3D Fluid Simulation: Steering the Simulation in Progress Using Lattice Boltzmann Method. In Proceedings of the 2019 18th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES), Wuhan, China, 8–10 November 2019; IEEE: Wuhan, China, 2019; pp. 72–75. [\[CrossRef\]](#)
19. Linxweiler, J.; Krafczyk, M.; Tölke, J. Highly interactive computational steering for coupled 3D flow problems utilizing multiple GPUs: Towards intuitive desktop environments for interactive 3D fluid structure interaction. *Comput. Vis. Sci.* **2010**, *13*, 299–314. [\[CrossRef\]](#)
20. Yalamanchili, P.; Arshad, U.; Mohammed, Z.; Garigipati, P.; Entschew, P.; Kloppenborg, B.; Malcolm, J.; Melonakos, J. ArrayFire—A High Performance Software Library for Parallel Computing with An Easy-to-Use API. AccelerEyes, 2015. Available online: <https://github.com/arrayfire/arrayfire> (accessed on 15 February 2021).
21. Wittmann, M.; Haag, V.; Zeiser, T.; Köstler, H.; Wellein, G. Lattice Boltzmann Benchmark Kernels as a Testbed for Performance Analysis. *Comput. Fluids* **2018**, *172*, 582–592. [\[CrossRef\]](#)
22. Tran, N.P.; Lee, M.; Hong, S. Performance Optimization of 3D Lattice Boltzmann Flow Solver on a GPU. *Sci. Program.* **2017**, *2017*, 1–16. [\[CrossRef\]](#)
23. Körner, C.; Pohl, T.; Rüde, U.; Thürey, N.; Zeiser, T. Parallel Lattice Boltzmann Methods for CFD Applications. In *Numerical Solution of Partial Differential Equations on Parallel Computers*; Series Title: Lecture Notes in Computational Science and Engineering; Bruaset, A.M., Tveito, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; Volume 51, pp. 439–466. [\[CrossRef\]](#)
24. Harwood, A.R.G.; Wenisch, P.; Revell, A.J. A Real-Time Modeling and Simulation Platform for Virtual Engineering Design and Analysis. In Proceedings of the 2018 6th European Conference on Computational Mechanics and the 7th European Conference on Computational Fluid Dynamics, Glasgow, UK, 11–15 June 2018; p. 11.
25. Succi, S. *The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond*; Numerical Mathematics and Scientific Computation, Clarendon Press: Oxford, UK, 2001.
26. 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. [\[CrossRef\]](#)
27. Succi, S. *The Lattice Boltzmann Equation: For Complex States of Flowing Matter*; Oxford University Press: Oxford, UK, 2018; pp. 1–762. [\[CrossRef\]](#)
28. Storti, D.; Yurtoglu, M. *CUDA for Engineers: An Introduction to High-Performance Parallel Computing*, 1st ed.; Addison-Wesley Professional: Boston, MA, USA, 2015.
29. Karimi, K.; Dickson, N.G.; Hamze, F. A Performance Comparison of CUDA and OpenCL. *arXiv* **2011**, arXiv:1005.2581.
30. Januszewski, M.; Kostur, M. Sailfish: A Flexible Multi-GPU Implementation of the Lattice Boltzmann Method. *Comput. Phys. Commun.* **2014**, *185*, 2350–2368. [\[CrossRef\]](#)
31. Boroni, G.; Dottori, J.; Rinaldi, P. FULL GPU Implementation of Lattice-Boltzmann Methods with Immersed Boundary Conditions for Fast Fluid Simulations. *Int. J. Multiphys.* **2017**, *11*. [\[CrossRef\]](#)
32. Kotsalos, C.; Latt, J.; Beny, J.; Chopard, B. Digital Blood in Massively Parallel CPU/GPU Systems for the Study of Platelet Transport. *arXiv* **2019**, arXiv:1911.03062.
33. Szőke, M.; Józsa, T.I.; Koleszár, Á.; Moulitsas, I.; Könözy, L. Performance Evaluation of a Two-Dimensional Lattice Boltzmann Solver Using CUDA and PGAS UPC Based Parallelisation. *ACM Trans. Math. Softw.* **2017**, *44*, 1–22. [\[CrossRef\]](#)
34. Harwood, A.R.; O'Connor, J.; Sanchez Muñoz, J.; Camps Santasmasas, M.; Revell, A.J. LUMA: A many-core, Fluid–Structure Interaction solver based on the Lattice-Boltzmann Method. *SoftwareX* **2018**, *7*, 88–94. [\[CrossRef\]](#)
35. Herschlag, G.; Lee, S.; Vetter, J.S.; Randles, A. GPU Data Access on Complex Geometries for D3Q19 Lattice Boltzmann Method. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada, 21–25 May 2018; IEEE: Vancouver, BC, Canada, 2018; pp. 825–834. [\[CrossRef\]](#)
36. Malcolm, J.; Yalamanchili, P.; McClanahan, C.; Venugopalakrishnan, V.; Patel, K.; Melonakos, J. ArrayFire: A GPU Acceleration Platform. In *SPIE Defense, Security, and Sensing*; Kelmelis, E.J., Ed.; International Society for Optics and Photonics: Bellingham, WA, USA, 2012; Volume 8403. [\[CrossRef\]](#)
37. Chrzesczyk, A. *Matrix Computations on GPU with ArrayFire—Python and ArrayFire—C/C++*. ArrayFire: Atlanta, GA, USA, 2017; p. 88.
38. McCool, M.; Robison, A.D.; Reinders, J. Chapter 8—Fork–Join. In *Structured Parallel Programming*; McCool, M., Robison, A.D., Reinders, J., Eds.; Morgan Kaufmann: Boston, MA, USA, 2012; pp. 209–251. [\[CrossRef\]](#)
39. Tan, J.; Fu, X. Addressing Hardware Reliability Challenges in General-Purpose GPUs. In *Advances in GPU Research and Practice*; Sarbazi-Azad, H., Ed.; Emerging Trends in Computer Science and Applied Computing, Morgan Kaufmann: Boston, MA, USA, 2017; pp. 649–705. [\[CrossRef\]](#)

-
40. Kloppenborg, B. *Real-Time Visualization of CUDA? Data Using ArrayFire Forge*; Technical Report; NVIDIA, GTC Silicon Valley: Santa Clara, CA, USA, 2016.
 41. Janßen, C.F.; Mierke, D.; Überrück, M.; Gralher, S.; Rung, T. Validation of the GPU-Accelerated CFD Solver ELBE for Free Surface Flow Problems in Civil and Environmental Engineering. *Computation* **2015**, *3*, 354–385. [[CrossRef](#)]