computation

MDPI

Article

Esoteric Pull and Esoteric Push: Two Simple In-Place Streaming Schemes for the Lattice Boltzmann Method on GPUs

Moritz Lehmann

Biofluid Simulation and Modeling—Theoretische Physik VI, University of Bayreuth, 95447 Bayreuth, Germany; moritz.lehmann@uni-bayreuth.de

Abstract: I present two novel thread-safe in-place streaming schemes for the lattice Boltzmann method (LBM) on graphics processing units (GPUs), termed Esoteric Pull and Esoteric Push, that result in the LBM only requiring one copy of the density distribution functions (DDFs) instead of two, greatly reducing memory demand. These build upon the idea of the existing Esoteric Twist scheme, to stream half of the DDFs at the end of one stream-collide kernel and the remaining half at the beginning of the next, and offer the same beneficial properties over the AA-Pattern scheme—reduced memory bandwidth due to implicit bounce-back boundaries and the possibility of swapping pointers between even and odd time steps. However, the streaming directions are chosen in a way that allows the algorithm to be implemented in about one tenth the amount of code, as two simple loops, and is compatible with all velocity sets and suitable for automatic code-generation. The performance of the new streaming schemes is slightly increased over Esoteric Twist due to better memory coalescence. Benchmarks across a large variety of GPUs and CPUs show that for most dedicated GPUs, performance differs only insignificantly from the One-Step Pull scheme; however, for integrated GPUs and CPUs, performance is significantly improved. The two proposed algorithms greatly facilitate modifying existing code to in-place streaming, even with extensions already in place, such as demonstrated here for the Free Surface LBM implementation FluidX3D. Their simplicity, together with their ideal performance characteristics, may enable more widespread adoption of in-place streaming across LBM GPU codes.

Keywords: lattice Boltzmann method; GPU; in-place streaming; swap algorithm; Esoteric Twist; memory; memory bandwidth; Volume-of-Fluid; FluidX3D; OpenCL

check for updates

Citation: Lehmann, M. Esoteric Pull and Esoteric Push: Two Simple In-Place Streaming Schemes for the Lattice Boltzmann Method on GPUs. *Computation* **2022**, *10*, 92. https://doi.org/10.3390/computation10060092

Academic Editor: Sergey A. Karabasov

Received: 1 April 2022
Accepted: 27 May 2022
Published: 2 June 2022

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

Copyright: © 2022 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

The lattice Boltzmann method (LBM) [1] is a type of direct numerical simulation (DNS) to model fluid flow in a physically accurate manner. Its explicit algorithmic structure makes it ideal for parallelization on graphics processing units (GPUs) [2–59]. The LBM works on a mesoscopic scale, representing quantities of fluid molecules by density distribution functions (DDFs) that are exchanged (streamed) between neighboring points on a Cartesian lattice. These DDFs are represented as floating-point numbers, and the streaming consists of copying them to the memory locations associated with neighboring lattice points. So, the LBM algorithm, at its core, copies floating-point numbers in memory with little arithmetic computation in between, meaning its performance is bound by memory bandwidth [3–10,13–22,33–46,59–65].

Since each lattice point holds the same number of DDFs and they need to be exchanged between one another at every time step, a strategy for avoiding data dependencies on parallel hardware is needed. The most straightforward and most common approach [12–44] is to have two copies, A and B, of the DDFs residing in memory; the algorithms reads from A and writes to B in even steps and in odd steps vice versa. With two copies of the DDFs, the memory access can even be configured to result in only partially misaligned reads and only coalesced writes (the One-Step-Pull scheme), enabling peak memory efficiency on modern

GPUs [13–23]. This solves the data dependencies but comes at the cost of almost doubling memory demand. Unfortunately, memory capacity is the largest constraint on GPUs [58], limiting the maximum possible lattice resolution. To eliminate the higher memory demand and at the same time resolve data dependencies, a class of thread-safe in-place streaming algorithms have been developed. The first of these is termed the AA-Pattern [3], as it reads from A and at the same time writes to A in a special manner that does not violate data dependencies. The algorithm, however, is asymmetric for even and odd time steps, so it has not been widely adopted. A later variant of AA-Pattern is the Shift-and-Swap-Streaming method [4]. Geier and Schönherr recently found a more intricate solution, termed Esoteric Twist [2], that is symmetric for even and odd time steps and, moreover, has the advantage of slightly reduced memory bandwidth and thus higher performance compared to AA-Pattern; however, its implementation is very complicated, hindering widespread adoption. Few works have thus far considered in-place streaming on GPUs [2–12], and apart from the works introducing the methods [2–5,66], only a few have adopted in-place streaming in their code [6–8,11].

This work introduces two new thread-safe in-place streaming schemes, termed Esoteric Pull and Esoteric Push, that are suitable for GPU implementation. They build upon the same idea as Esoteric Twist and offer ideal performance characteristics while at the same time significantly simplifying implementation and allowing for automatic code generation. The very simple and modular implementation is especially well suited to modifying existing LBM implementations, even if various extensions, such as Free Surface LBM, are already in place.

2. Naive Implementation—One-Step Pull and One-Step Push

The most common LBM implementation uses two copies of the DDFs in memory to resolve data dependencies in a parallel environment [12–44]. There are two variants, namely One-Step Pull (Figure 1, Listing A1) and One-Step Push (Figure 2, Listing A2). The pull variant is generally preferred on GPUs because the penalty for non-coalesced reads is smaller than that for non-coalesced writes [13–23]. The coloring introduced in Figures 1 and 2 illustrates how loading/storing patterns compare to the regular DDF sequence during collision in registers, in other words, where exactly each DDF is loaded and stored in memory. This makes the more recently introduced, more sophisticated streaming patterns more comprehensible. In the One-Step-Pull scheme, DDFs are pulled in from neighbors (copy A of the DDFs), collided, and stored at the center node (copy B of the DDFs). In the One-Step-Push scheme, DDFs are loaded from the center node (copy A of the DDFs), collided, and then pushed out to neighbors (copy B of the DDFs). For both schemes, after every time step, the pointers to A and B are swapped.

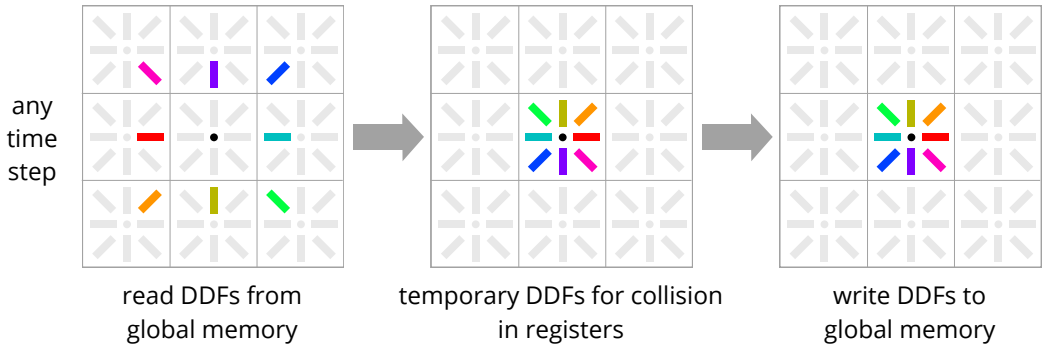


Figure 1. One-Step-Pull streaming scheme. Two copies of the DDFs are used to resolve data dependencies.

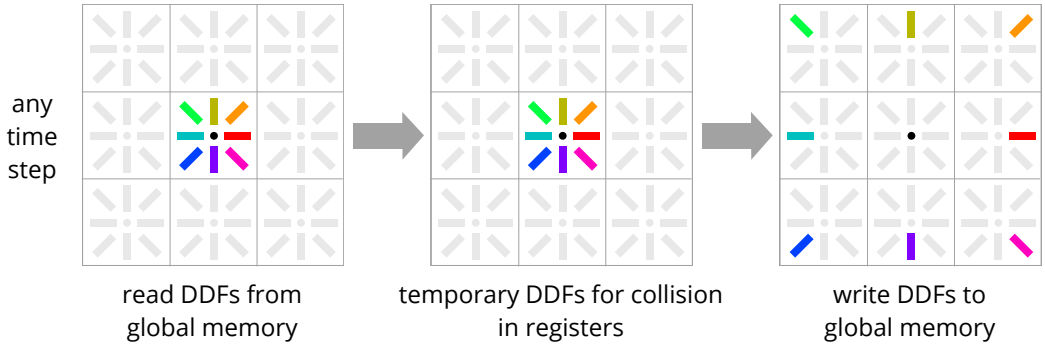


Figure 2. One-Step-Push streaming scheme. Two copies of the DDFs are used to resolve data dependencies.

3. State-of-the-Art Methods for In-Place Streaming on GPUs

The data dependency problem with in-place streaming on parallel hardware has already been solved by two major approaches, termed AA-Pattern and Esoteric Twist. Both provide the great advantage of significantly reducing memory demand for the LBM; however, both also pose various difficulties in GPU implementation, hindering widespread adoption.

3.1. AA-Pattern

When performing the LBM streaming step on parallel hardware, the issue arises that neighboring lattice points may be processed in parallel, and the exact order of execution is random. One or more threads may not write an updated value to a memory address from which another concurrent thread is reading, because then either the old or the new value may be used by the reading thread. This error is known as a race condition. To perform the LBM streaming step in parallel with only a single buffer for the DDFs, one must write updated values only to the same memory addresses that one thread has previously read the values from. Then, no two threads access the same memory addresses. Bailey et al. [3] have found that this is possible if, for even time steps combining one streaming step, the collision and a second streaming step are performed, and for odd time steps, only the collision step is performed. This makes the processed DDFs always end up in the same locations as they were read from, resolving data dependencies on concurrent hardware with only one copy of the DDFs in memory. To make the DDFs actually stream through memory locations, in even time steps, the DDFs are stored at the neighbor nodes in opposite orientation after the collision, and in odd time steps, before the collision, the DDFs are loaded from the center node in opposite orientation. The resulting algorithm reads the DDFs from copy A of the DDFs and writes to the same copy A in place, so it was termed AA-Pattern (Figure 3, Listing A3). It is a popular but mistaken belief that the different even and odd time steps require duplicate implementation of the stream_collide kernel because the pointers to the DDFs cannot be swapped in between time steps. The loading and storing of the DDFs before and after collision can be placed in functions, and when the LBM time step is passed as a parameter, these functions then switch between loading/storing the DDFs from/to neighbors or at the center point (see Listing A3). The stream_collide kernel then contains calls to these two functions before and after collision, and no duplicate implementation is required. Note that there are also two more modern variants of the AA-Pattern, termed Shift-and-Swap Streaming (SSS) [4] and Periodic Shift (PS) [5], offering benefits in programming languages where pointer arithmetic is available.

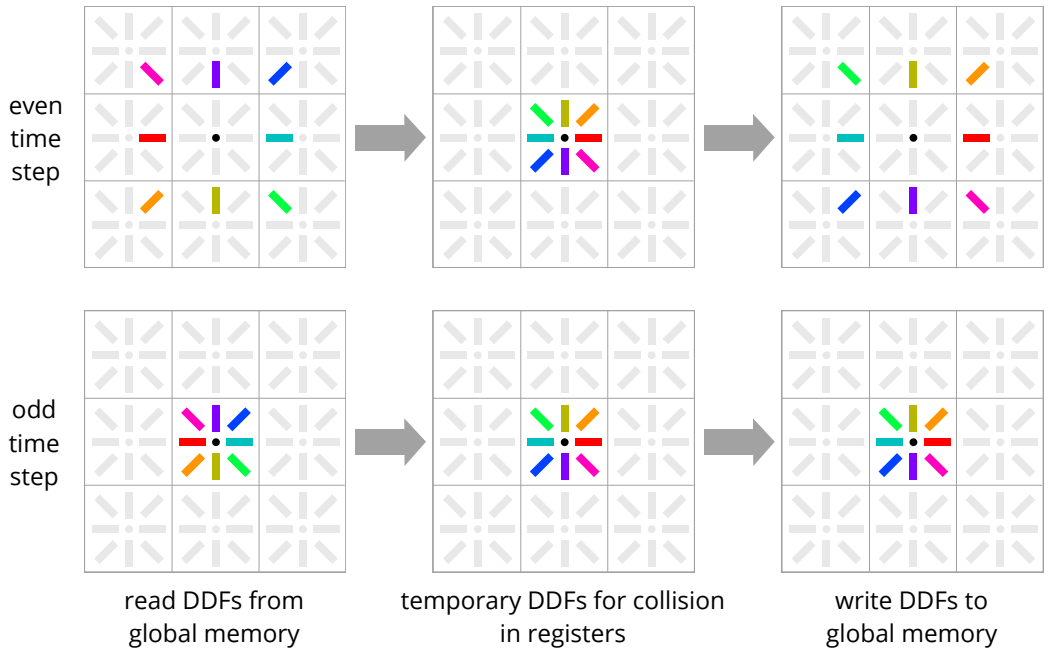


Figure 3. AA-Pattern in-place streaming scheme [3]. Even time steps: DDFs are pulled in from neighbors, collided, and then pushed out to the neighbors again, but stored in opposite orientation. Odd time steps: DDFs are loaded from the center node in opposite orientation, collided, and stored at the center node again in the same orientation as during collision. DDFs are always stored in the same memory locations where they were loaded from, so only one copy of the DDFs is required.

3.2. Esoteric Twist

The idea of the Esoteric Twist in-place streaming scheme (Figure 4, Listing A4) is to pull only DDFs for negative directions, execute the collision, and then push only DDFs for positive directions [2]. To resolve data dependencies on concurrent hardware, in even steps, the DDFs are stored in opposite orientation after the collision, and in odd time steps, before the collision, the DDFs are loaded in opposite orientation.

Esoteric Twist resembles a criss-cross access pattern shifted north-east by half a node, accessing the DDFs at a total of 4 nodes (in the 2D case) or up to 8 nodes (in 3D). For certain implementations, this reduces the number of ghost nodes required, but it makes the implementation of the index calculation tedious as it requires manually writing the indices, which are different across velocity sets. On top of this, for some velocity sets such as D3Q15, additional streaming directions must be computed beyond the streaming directions of the velocity set (see Listing A4), making the already tedious implementation even more cumbersome. This is an obstacle to implementing different velocity sets in a modular manner or with code generation. With some LBM extensions such as Volume-of-Fluid, duplicate (inverse) implementation of the streaming is required, so Esoteric Twist becomes very impractical.

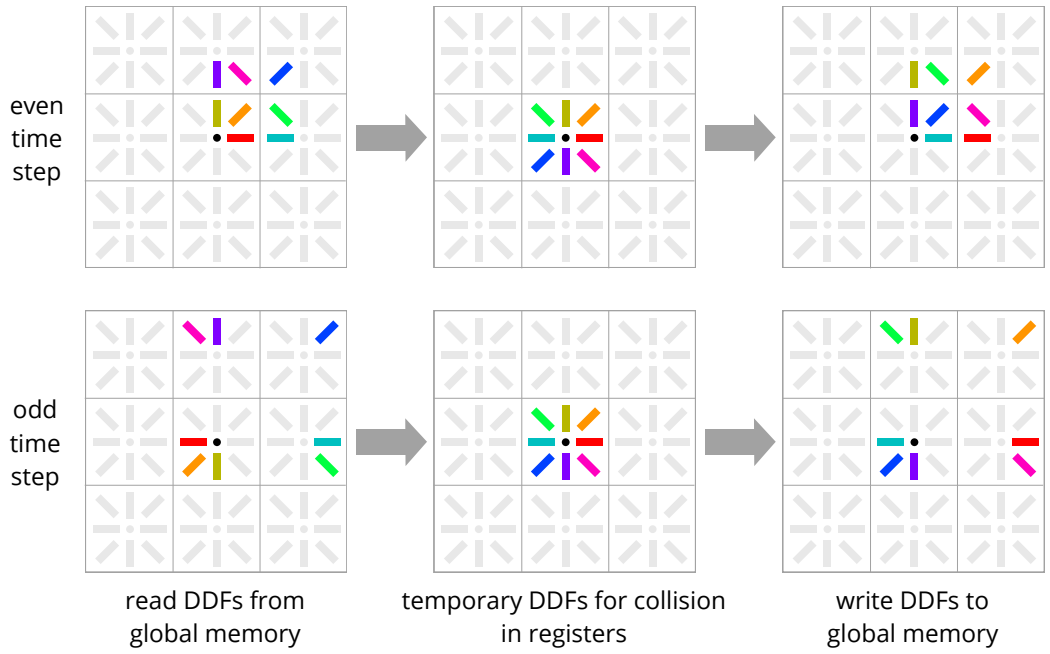


Figure 4. Esoteric Twist in-place streaming scheme [2]. Even time steps: DDFs are loaded in a criss-cross pattern shifted north-east by half a node. After collision, DDFs are stored in the same pattern but in opposite orientation. Odd time steps: DDFs are loaded in opposite orientation in a shifted criss-cross pattern that covers only DDFs not touched in the even time step. After collision, DDFs are written back in the same pattern but with regular orientation once again. DDFs are always stored in the same memory locations where they were loaded from, so only one copy of the DDFs is required.

4. New Methods: Esoteric Pull and Esoteric Push

My two novel in-place streaming algorithms are based on the same idea as the Esoteric Twist scheme, namely that only DDFs in negative directions are pulled before collision, and that after collision only DDFs in positive directions are pushed. Thus, half of the DDFs are streamed at the end of one stream_collide kernel and the other half at the beginning of the next.

The important observation I made is that the shifted criss-cross pattern of Esoteric Twist is not essential for the swap algorithm to work. I waive shifting the north-west to south-east DDFs north by one node in 2D, and waive shifting other diagonal directions in 3D, abandoning the shifted criss-cross pattern. Instead, the regular streaming direction neighbors are used. This enables a trivial index calculation in two four-line loops (unrolled by the compiler) for loading and storing in a way that works with all velocity sets out of the box. This also makes the implementation less redundant and much less prone to errors, and it further enables automatic code generation.

The Esoteric Pull scheme (Figure 5, Listing A5) in 2D differs from Esoteric Twist (Figure 4) only in the positions of the DDFs for the north-west to south-east directions, which are loaded/stored in their regular locations instead of shifted north by one node. In 3D, other diagonal directions are also not shifted, and their regular streaming directions are used instead to determine the streaming neighbor nodes. This has not only the advantage of trivial index calculation, but also improves memory coalescence for the DDFs in these diagonal directions that would be otherwise shifted by one lattice point with Esoteric Twist, leading to slightly higher performance demands.

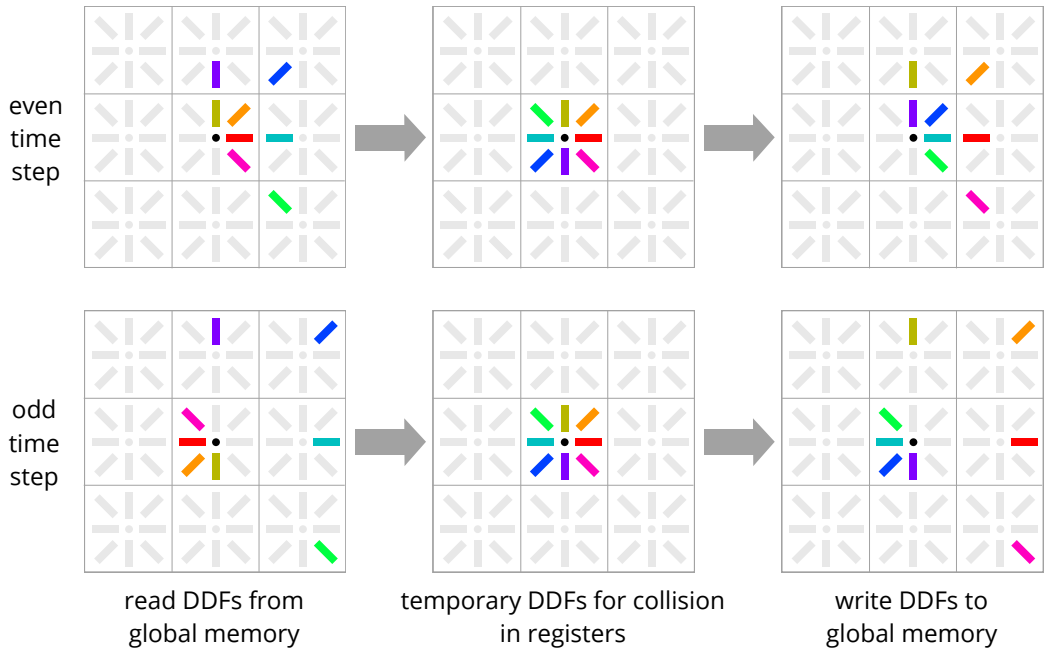


Figure 5. Esoteric Pull in-place streaming scheme. Even time steps: DDFs in positive directions are loaded from the center node, and DDFs from negative directions are pulled in from their regular streaming direction neighbors and are collided. Then, DDFs in positive directions are pushed out to neighbors and stored in opposite orientation, and DDFs in negative directions are stored at the center node in opposite orientation. Odd time steps: DDFs in positive directions are loaded from the center node in opposite orientation, and DDFs from negative directions are pulled in from their regular streaming direction neighbors in opposite orientation and are collided. Then, DDFs in positive directions are pushed out to neighbors, and DDFs in negative directions are stored at the center node. DDFs are always stored in the same memory locations where they were loaded from, so only one copy of the DDFs is required.

The Esoteric Push scheme (Figure 6, Listing A6) is essentially Figure 5 flipped by 180 degrees (except for the temporary DDFs in registers), highlighting two distinct symmetry flips that can be performed independently of each other: a) switching streaming for positive/negative directions, and b) switching even/odd time steps.

Both schemes yield simulations that are bitwise identical to Esoteric Twist. If pointer arithmetic is available, the pointers of DDFs in positive and negative directions can be swapped in between time steps such that memory addressing is the same for all time steps, in the very same manner as for Esoteric Twist.

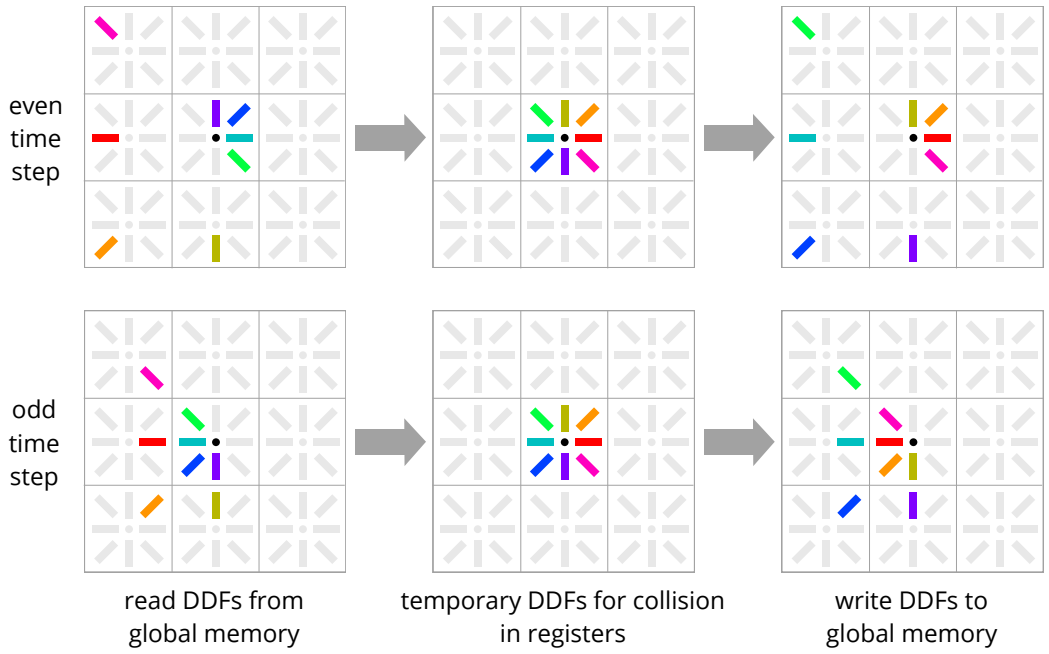


Figure 6. Esoteric Push in-place streaming scheme. Figure 5 flipped by 180 degrees (except for the temporary DDFs in registers).

4.1. Implicit Bounce-Back

In the very same manner as for the Esoteric Twist scheme [2], both Esoteric Pull and Esoteric Push offer the benefits of the ingeniously emerging implicit bounce-back boundaries. Due to the way the DDFs are flipped in orientation for regular fluid nodes, and because boundary nodes are not processed at all (with a guard clause at the very beginning of the stream_collide kernel), the DDFs of boundary nodes are not flipped in memory, so for neighboring fluid nodes, it appears as if their DDFs are already correctly flipped such that bounce-back boundaries automatically apply.

There are three distinct benefits to this side-effect of the Esoteric streaming schemes: (a) fluid nodes do not have to check the flags of their neighbors at all to apply bounce-back boundaries, reducing overall memory bandwidth and increasing performance slightly, (b) memory access is more coalesced, and (c) the implementation is simplified.

4.2. Comparison with Existing Streaming Schemes

When comparing the different streaming algorithms in Table 1 for DdQq LBM, the advantages of in-place streaming become evident, such as reduced storage and, for the Esoteric algorithms, also reduced bandwidth. In-place streaming reduces memory demand by $4\,q$ Bytes/node. The Esoteric algorithms further reduce memory bandwidth by $q-1$ Byte/node per time step, as neighbor flags do not have to be checked for implicit bounce-back boundaries. Based on their storage and performance properties, Esoteric Pull/Push appear identical to Esoteric Twist, apart from slightly improved memory coalescence for the DDFs in some of the diagonal directions. The main improvements of Esoteric Pull/Push are located in the much more straightforward implementation that is compatible with all velocity sets. Instead of having to manually unroll the loops over the streaming directions and make sure that all indices are typed correctly for each velocity set, the streaming can now be written in a generic way as two short loops that are unrolled by the compiler (compare Listings A4 and A5). This also allows for automatic code generation that many LBM implementations heavily rely on and significantly improves code maintainability.

A recently proposed method for reducing storage and bandwidth by resorting to FP32/16-bit mixed precision [13] makes in-place streaming with the Esoteric schemes even more compelling, reducing memory storage from 169 to 55 Bytes/node—less than one

third—and reducing bandwidth from 171 to 77 Bytes/node per time step for D3Q19. Less memory demand per node enables much larger lattice resolutions.

Table 1. Comparing memory storage (Bytes/node) and bandwidth (Bytes/node per time step) requirements of the different GPU-compatible streaming algorithms for DdQq LBM with FP32 arithmetic precision and eight available flag bits per node. With the in-place streaming and implicit bounce-back of the Esoteric schemes, and with FP32/16-bit mixed precision as proposed in [13], memory demands and bandwidth are significantly reduced.

Algorithm	Storage	Bandwidth
One-Step Pull	$8q + 4d + 5$	$9q$
One-Step Push	$8q + 4d + 5$	$9q$
AA-Pattern	$4q + 4d + 5$	$9q$
Esoteric Twist	$4q + 4d + 5$	$8q + 1$
Esoteric Pull	$4q + 4d + 5$	$8q + 1$
Esoteric Push	$4q + 4d + 5$	$8q + 1$
OSP + FP32/16-bit	$4q + 4d + 5$	$5q$
AA + FP32/16-bit	$2q + 4d + 5$	$5q$
ET/EP + FP32/16-bit	$2q + 4d + 5$	$4q + 1$

When comparing D3Q19 SRT performance of the different streaming algorithms on the Nvidia A100 40GB GPU with a FP32 single-precision floating-point, One-Step Pull serves as the baseline at 8816 MLUPs/s (100%). One-Step Push is slightly slower at 8675 MLUPs/s (98%). The AA-Pattern runs with 8269 MLUPs/s (94%), Esoteric Twist mitigates the efficiency losses with reduced bandwidth due to implicit bounce-back at 8483 MLUPs/s (96%), and Esoteric Pull/Push offer even slightly higher performance at 8522 MLUPs/s (97%), due to better memory coalescence for the diagonals that are shifted by one lattice point for Esoteric Twist. Looking at additional performance benchmarks across different hardware configurations (Figure 7), the benefit of less memory bandwidth usage due to implicit bounce-back approximately cancels out the drawback of more inefficient, partially misaligned writes due to the inherent symmetry of the memory access. In comparison with the One-Step-Pull streaming scheme [13], although memory efficiency is lower, performance changes only insignificantly on most dedicated GPUs, with some performance increase for FP32/FP16 mixed precision. On integrated GPUs and on CPUs, however, there is a significant increase in performance due to more efficient use of on-chip cache with in-place memory access. The benchmark case used is an empty box with a default size of 256^3, with no extensions enabled except bounce-back boundaries, following [13]. For devices where not enough memory was available, the box size was reduced, and for the AMD Radeon VII, the box size was increased to 464^3.

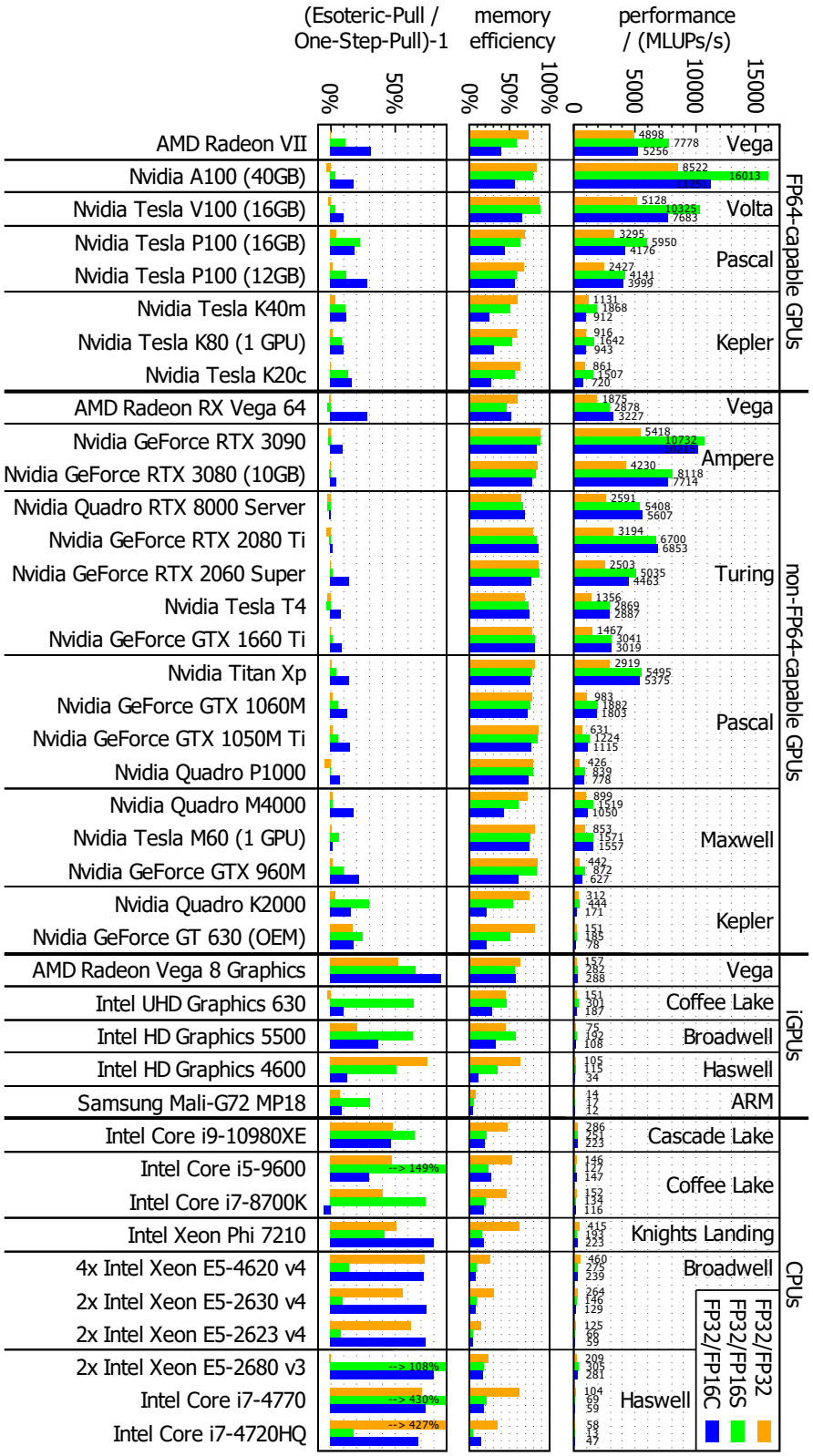


Figure 7. Performance of Esoteric Pull with D3Q19 SRT on different hardware configurations in the *FluidX3D* OpenCL implementation, in million lattice updates per second (MLUPs/s). Efficiency is calculated by dividing the measured MLUPs/s by the data sheet memory bandwidth times the number of bytes transferred per lattice point and time step (Table 1). CPU benchmarks are on all cores. Performance comparison with the One-Step-Pull streaming scheme [13] shows only insignificant differences on most dedicated GPUs, but large gains on integrated GPUs and CPUs.

5. Esoteric Pull for Free Surface LBM on GPUs

Here, to further underline the substantial perks of the added simplicity of Esoteric Pull over Esoteric Twist, the modification of existing Free Surface LBM (FSLBM) code from One-Step Pull to Esoteric Pull in-place streaming is briefly discussed using the example of *FluidX3D* [14].

Although the One-Step-Pull scheme makes an FSLBM implementation the simplest and most efficient, it can easily be modified to the Esoteric Pull in-place streaming scheme. FSLBM on a GPU requires three more kernels in addition to the stream_collide kernel. To distinguish between node types, three flag bits are required and can be represented as follows: fluid (001, F), interface (010, I), gas (100, G), interface→fluid (011, IF), interface→gas (110), and gas→interface (111, GI). These kernels are first introduced for the implementation of One-Step Pull:

- stream_collide: Immediately return for G nodes. Stream in DDFs from neighbors, but for F and I nodes also load outgoing DDFs from the center node to compute the mass transfer for Volume-of-Fluid [67]. Apply excess mass for F or I nodes by summing it from all neighboring F and I nodes. Compute the local surface curvature with PLIC [26] and reconstruct DDFs from neighboring G nodes. After collision, compare mass m and post-collision density ρ; along with neighboring flags, mark whether the center node should remain I or change to IF or IG. Store post-collision DDFs at the local node.
- surface_1: Prevent neighbors of IF nodes from becoming/being G nodes; update flags of such neighbors to either I (from IF) or GI (from G).
- surface_2: For GI nodes, reconstruct and store DDFs based on the average density and velocity of all neighboring F, I, or IF nodes. For IG nodes, turn all neighboring F or IF nodes to I.
- surface_3: Change IF nodes to F, IG nodes to G, and GI nodes to I. Compute excess mass for each case separately as well as for F, I, and G nodes, then divide the local excess mass by the number of neighboring F, I, IF, and GI nodes and store the excess mass on the local node.

After modifying the streaming scheme from One-Step Pull to Esoteric Pull, a fifth kernel must be added preceding the stream_collide kernel because in the stream_collide kernel, the outgoing DDFs cannot be loaded as neighboring nodes may overwrite them in memory within the same time step (race condition):

- surface_0: Immediately return for G nodes. Stream in DDFs as in Figure 5 (incoming DDFs), but also load outgoing DDFs in opposite directions to compute the mass transfer for Volume-of-Fluid. For I nodes, compute the local surface curvature with PLIC and reconstruct DDFs for neighboring G nodes; store these reconstructed DDFs in the locations at the neighbors from which they will be streamed in in the following stream_collide kernel. Apply excess mass for F or I nodes by summing from all neighboring F and I nodes.
- stream_collide: Immediately return for G nodes. Stream in DDFs as in Figure 5 and execute collision. For I nodes, compare mass m and post-collision density ρ; along with neighboring flags, mark whether the center node should remain I or change to IF or IG. Stream out the DDFs as in Figure 5.
- surface_1, surface_2, surface_3: unchanged

In the surface_0 kernel, additional streaming in inverted directions is necessary to load outgoing DDFs and store reconstructed DDFs for neighboring G nodes. Having to perform manual index calculations here, as with Esoteric Twist, would vastly elongate and over-complicate the code and reduce maintainability. With the Esoteric Pull/Push variants, only two simple loops of four lines of code each are required. The full FSLBM OpenCL C implementation with Esoteric Pull is provided in the Appendix B in Listing A7.

The change to the Esoteric Pull in-place streaming algorithm reduces the memory demand for FSLBM from 181 to 105 Bytes/node while only decreasing performance by

approximately 20% due to the duplicate loading of incoming DDFs and having to store reconstructed DDFs for G nodes. When combined with FP32/16-bit mixed precision [13], the memory demand is further reduced to 67 Bytes/node or about $\frac{1}{3}$ of vanilla FSLBM, comparable to or even less than the memory requirements of other (Navier–)Stokes solvers [68–71].

This, in turn, enables colossal lattice resolutions as illustrated by an example in Figure 8: a 7 mm (188 lattice point) diameter raindrop, 1.5 ms (7700 time steps) after impacting a deep pool at 9.55 $\frac{\mathrm{m}}{\mathrm{s}}$ [24] and 20° inclination, was simulated with the *FluidX3D* implementation with FP32/FP16C mixed precision [13]. Lattice resolution is $940 \times 940 \times 800$ or 707 million lattice points. The simulation was conducted on an Nvidia Quadro RTX 8000 GPU with 48 GB video memory and took 27 min, including the rendering of the image. The code for this setup is provided in Appendix C in Listing A8.

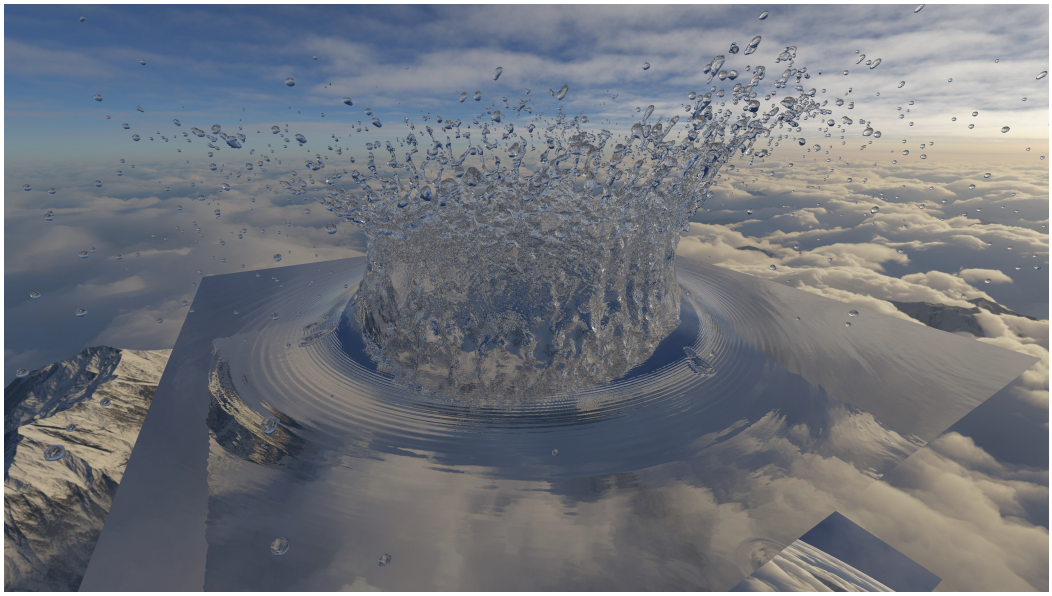


Figure 8. Esoteric Pull in-place streaming with FP16C memory compression [13] enables a colossal $940 \times 940 \times 800$ lattice resolution on a single 48 GB GPU, such as demonstrated here with a raindrop impact simulation. This figure is included in the Supplementary Files as a video.

6. Conclusions

In-place streaming is essential for any LBM GPU implementation as it significantly reduces memory demand and increases the maximum lattice resolution. However, existing thread-safe solutions for GPUs, such as AA-Pattern and Esoteric Twist, never gained widespread adoption due to the difficulty of implementation. The new Esoteric Pull and Esoteric Push schemes presented in this work should change that. They build upon the same idea as the Esoteric Twist scheme—streaming half of the DDFs at the end of one stream_collide kernel and the other half at the beginning of the next—but greatly simplify the implementation because of their trivial index calculations, even allowing for automatic code generation. For existing GPU implementations of the common One-Step-Pull scheme, the switch to Esoteric Pull requires only moderate modifications to the code, even if several extensions are already implemented, as demonstrated herein with Free Surface LBM. In contrast, the implementation of Esoteric Twist would be much more difficult and error-prone in such a case, as the index calculation has to be implemented twice in different variations for regular LBM streaming and for FSLBM mass exchange.

The Esoteric Pull and Esoteric Push schemes share the same performance advantage as Esoteric Twist over AA-Pattern: slightly reduced bandwidth due to implicit bounce-back. Moreover, compared to Esoteric Twist, memory coalescence is slightly improved on the otherwise shifted diagonal directions. This allows the Esoteric Pull/Push schemes, with only one DDF buffer, to provide GPU performance on par with the One-Step-Pull scheme

with double DDF buffers, despite requiring less efficient misaligned writes. In addition, on integrated GPUs and CPUs, performance is significantly increased.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/computation10060092/s1, Video S1: Esoteric Pull in-place streaming with FP16C memory compression.

Funding: This study was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—Project Number 391977956—SFB 1357.

Data Availability Statement: All data are archived and available upon request. All code beyond what is provided in the listings is archived and available upon reasonable request.

Acknowledgments: I acknowledge support through the computational resources provided by BZHPC, LRZ and JSC. I acknowledge the NVIDIA Corporation for donating a Titan Xp GPU and an A100 40GB GPU for my research. I acknowledge Stephan Gekle for motivating me to write this paper.

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

Abbreviations

The following abbreviations are used in this manuscript:

LBM	lattice Boltzmann method
DDF	density distribution function
GPU	graphics processing unit
CPU	central processing unit
SRT	single-relaxation-time
OSP	One-Step Pull/Push
AA	AA-Pattern
ET	Esoteric Twist
EP	Esoteric Pull/Push

Appendix A. OpenCL C Implementation of the Different Streaming Schemes

Appendix A.1. One-Step Pull

Listing A1. One-Step-Pull implementation in OpenCL C.

```
)+R(void load_f(const uint n, float* fhn, const global fpxx* fA, const uint* j, const uchar* flagsj) {
  fhn[0] = load(fA, index_f(n, 0u)); // One-Step-Pull
  for(uint i=1u; i<def_velocity_set; i+=2u) {
    fhn[i  ] = load(fA, flagsj[i+1u]&TYPE_W ? index_f(n, i+1u) : index_f(j[i+1u], i   )) ; // boundary : regular
    fhn[i+1] = load(fA, flagsj[i   ]&TYPE_W ? index_f(n, i   ) : index_f(j[i   ], i+1u)) ;
  }
}
)+R(void store_f(const uint n, const float* fhn, global fpxx* fB, const uint* j, const ulong t, const uchar* flagsj) {
  for(uint i=0u; i<def_velocity_set; i++) store(fB, index_f(n, i), fhn[i]); // One-Step-Pull
}
```

Appendix A.2. One-Step Push

Listing A2. One-Step-Push implementation in OpenCL C.

```
)+R(void load_f(const uint n, float* fhn, const global fpxx* fA, const uint* j, const uchar* flagsj) {
  for(uint i=0u; i<def_velocity_set; i++) fhn[i] = load(fA, index_f(n, i)); // One-Step-Push
}
)+R(void load_f(const uint n, float* fhn, const global fpxx* fA, const uint* j, const uchar* flagsj) {
  for(uint i=0u; i<def_velocity_set; i++) fhn[i] = load(fA, index_f(n, i)); // One-Step-Push
}
```

```
)+R(void store_f(const uint n, const float* fhn, global fpxx* fB, const uint* j, const
    ↪ uchar* flagsj) {
  store(fB, index_f(n, 0u), fhn[0]); // One-Step-Push
  for(uint i=1u; i<def_velocity_set; i+=2u) {
    store(fB, flagsj[i   ]&TYPE_W ? index_f(n, i+1u) : index_f(j[i   ], i   ), fhn[i
        ↪ ]); // boundary : regular
    store(fB, flagsj[i+1u]&TYPE_W ? index_f(n, i   ) : index_f(j[i+1u], i+1u), fhn[i+1u
        ↪ ]);
  }
}
```

Appendix A.3. AA-Pattern

Listing A3. AA-Pattern implementation in OpenCL C.

```
)+R(void load_f(const uint n, float* fhn, const global fpxx* fi, const uint* j, const
    ↪ ulong t, const uchar* flagsj) {
  fhn[0] = load(fi, index_f(n, 0u)); // AA-Pattern
  if(t%2ul) {
    for(uint i=1u; i<def_velocity_set; i+=2u) { // pull
      fhn[i  ] = load(fi, flagsj[i+1u]&TYPE_W ? index_f(n, i+1u) : index_f(j[i+1u], i
          ↪ )); // boundary : regular
      fhn[i+1] = load(fi, flagsj[i   ]&TYPE_W ? index_f(n, i   ) : index_f(j[i   ], i+1u
          ↪ ));
    }
  } else {
    for(uint i=1u; i<def_velocity_set; i+=2u) { // load local (inverse)
      fhn[i  ] = load(fi, index_f(n, i+1u));
      fhn[i+1] = load(fi, index_f(n, i   ));
    }
  }
}
)+R(void store_f(const uint n, const float* fhn, global fpxx* fi, const uint* j, const
    ↪ ulong t, const uchar* flagsj) {
  store(fi, index_f(n, 0u), fhn[0]); // AA-Pattern
  if(t%2ul) {
    for(uint i=1u; i<def_velocity_set; i+=2u) { // push
      store(fi, flagsj[i   ]&TYPE_W ? index_f(n, i   ) : index_f(j[i   ], i+1u), fhn[i
          ↪   ]); // boundary : regular
      store(fi, flagsj[i+1u]&TYPE_W ? index_f(n, i+1u) : index_f(j[i+1u], i   ), fhn[i+1
          ↪ u]);
    }
  } else {
    for(uint i=1u; i<def_velocity_set; i++) { // store local
      store(fi, index_f(n, i), fhn[i]);
    }
  }
}
```

Appendix A.4. Esoteric Twist

Listing A4. Esoteric-Twist implementation in OpenCL C.

```
)+R(void load_f(const uint n, float* fhn, const global fpxx* fi, const uint* j, const
    ↪ ulong t) {
  fhn[0] = load(fi, index_f(n, 0u)); // Esoteric-Twist
)+"#if defined(D2Q9)"+R(
  fhn[1] = load(fi, index_f(n   , t%2ul ? 1 : 2));
  fhn[2] = load(fi, index_f(j[1], t%2ul ? 2 : 1));
  fhn[3] = load(fi, index_f(n   , t%2ul ? 3 : 4));
  fhn[4] = load(fi, index_f(j[3], t%2ul ? 4 : 3));
  fhn[5] = load(fi, index_f(n   , t%2ul ? 5 : 6));
  fhn[6] = load(fi, index_f(j[5], t%2ul ? 6 : 5));
  fhn[7] = load(fi, index_f(j[3], t%2ul ? 7 : 8)); // attention
  fhn[8] = load(fi, index_f(j[1], t%2ul ? 8 : 7)); // attention
)+"#elif defined(D3Q15)"+R(
  uint x0, xp, xm, y0, yp, ym, z0, zp, zm;
  calculate_indices(n, &x0, &xp, &xm, &y0, &yp, &ym, &z0, &zp, &zm);
  fhn[ 1] = load(fi, index_f(n       , t%2ul ?  1 :  2));
  fhn[ 2] = load(fi, index_f(j[ 1]   , t%2ul ?  2 :  1));
  fhn[ 3] = load(fi, index_f(n       , t%2ul ?  3 :  4));
  fhn[ 4] = load(fi, index_f(j[ 3]   , t%2ul ?  4 :  3));
  fhn[ 5] = load(fi, index_f(n       , t%2ul ?  5 :  6));
  fhn[ 6] = load(fi, index_f(j[ 5]   , t%2ul ?  6 :  5));
  fhn[ 7] = load(fi, index_f(n       , t%2ul ?  7 :  8));
  fhn[ 8] = load(fi, index_f(j[ 7]   , t%2ul ?  8 :  7));
  fhn[ 9] = load(fi, index_f(j[ 5]   , t%2ul ?  9 : 10)); // attention
```

```
    fhn[10] = load(fi, index_f(xp+yp+z0, t%2ul ? 10 :  9)); // attention, additional streaming direction required beyond velocity set
    fhn[11] = load(fi, index_f(j[ 3]    , t%2ul ? 11 : 12)); // attention
    fhn[12] = load(fi, index_f(xp+y0+zp, t%2ul ? 12 : 11)); // attention, additional streaming direction required beyond velocity set
    fhn[13] = load(fi, index_f(j[ 1]    , t%2ul ? 13 : 14)); // attention
    fhn[14] = load(fi, index_f(x0+yp+zp, t%2ul ? 14 : 13)); // attention, additional streaming direction required beyond velocity set
)+"#elif defined(D3Q19)"+R(
    fhn[ 1] = load(fi, index_f(n     , t%2ul ?  1 :  2));
    fhn[ 2] = load(fi, index_f(j[ 1], t%2ul ?  2 :  1));
    fhn[ 3] = load(fi, index_f(n     , t%2ul ?  3 :  4));
    fhn[ 4] = load(fi, index_f(j[ 3], t%2ul ?  4 :  3));
    fhn[ 5] = load(fi, index_f(n     , t%2ul ?  5 :  6));
    fhn[ 6] = load(fi, index_f(j[ 5], t%2ul ?  6 :  5));
    fhn[ 7] = load(fi, index_f(n     , t%2ul ?  7 :  8));
    fhn[ 8] = load(fi, index_f(j[ 7], t%2ul ?  8 :  7));
    fhn[ 9] = load(fi, index_f(n     , t%2ul ?  9 : 10));
    fhn[10] = load(fi, index_f(j[ 9], t%2ul ? 10 :  9));
    fhn[11] = load(fi, index_f(n     , t%2ul ? 11 : 12));
    fhn[12] = load(fi, index_f(j[11], t%2ul ? 12 : 11));
    fhn[13] = load(fi, index_f(j[ 3], t%2ul ? 13 : 14)); // attention
    fhn[14] = load(fi, index_f(j[ 1], t%2ul ? 14 : 13)); // attention
    fhn[15] = load(fi, index_f(j[ 5], t%2ul ? 15 : 16)); // attention
    fhn[16] = load(fi, index_f(j[ 1], t%2ul ? 16 : 15)); // attention
    fhn[17] = load(fi, index_f(j[ 5], t%2ul ? 17 : 18)); // attention
    fhn[18] = load(fi, index_f(j[ 3], t%2ul ? 18 : 17)); // attention
)+"#elif defined(D3Q27)"+R(
    fhn[ 1] = load(fi, index_f(n     , t%2ul ?  1 :  2));
    fhn[ 2] = load(fi, index_f(j[ 1], t%2ul ?  2 :  1));
    fhn[ 3] = load(fi, index_f(n     , t%2ul ?  3 :  4));
    fhn[ 4] = load(fi, index_f(j[ 3], t%2ul ?  4 :  3));
    fhn[ 5] = load(fi, index_f(n     , t%2ul ?  5 :  6));
    fhn[ 6] = load(fi, index_f(j[ 5], t%2ul ?  6 :  5));
    fhn[ 7] = load(fi, index_f(n     , t%2ul ?  7 :  8));
    fhn[ 8] = load(fi, index_f(j[ 7], t%2ul ?  8 :  7));
    fhn[ 9] = load(fi, index_f(n     , t%2ul ?  9 : 10));
    fhn[10] = load(fi, index_f(j[ 9], t%2ul ? 10 :  9));
    fhn[11] = load(fi, index_f(n     , t%2ul ? 11 : 12));
    fhn[12] = load(fi, index_f(j[11], t%2ul ? 12 : 11));
    fhn[13] = load(fi, index_f(j[ 3], t%2ul ? 13 : 14)); // attention
    fhn[14] = load(fi, index_f(j[ 1], t%2ul ? 14 : 13)); // attention
    fhn[15] = load(fi, index_f(j[ 5], t%2ul ? 15 : 16)); // attention
    fhn[16] = load(fi, index_f(j[ 1], t%2ul ? 16 : 15)); // attention
    fhn[17] = load(fi, index_f(j[ 5], t%2ul ? 17 : 18)); // attention
    fhn[18] = load(fi, index_f(j[ 3], t%2ul ? 18 : 17)); // attention
    fhn[19] = load(fi, index_f(n     , t%2ul ? 19 : 20));
    fhn[20] = load(fi, index_f(j[19], t%2ul ? 20 : 19));
    fhn[21] = load(fi, index_f(j[ 5], t%2ul ? 21 : 22)); // attention
    fhn[22] = load(fi, index_f(j[ 7], t%2ul ? 22 : 21)); // attention
    fhn[23] = load(fi, index_f(j[ 3], t%2ul ? 23 : 24)); // attention
    fhn[24] = load(fi, index_f(j[ 9], t%2ul ? 24 : 23)); // attention
    fhn[25] = load(fi, index_f(j[ 1], t%2ul ? 25 : 26)); // attention
    fhn[26] = load(fi, index_f(j[11], t%2ul ? 26 : 25)); // attention
)+"#endif"+R( // D3Q27
}
)+R(void store_f(const uint n, const float* fhn, global fpxx* fi, const uint* j, const ulong t) {
    store(fi, index_f(n, 0u), fhn[0]); // Esoteric-Twist
)+"#if defined(D2Q9)"+R(
    store(fi, index_f(j[1], t%2ul ? 2 : 1), fhn[1]);
    store(fi, index_f(n   , t%2ul ? 1 : 2), fhn[2]);
    store(fi, index_f(j[3], t%2ul ? 4 : 3), fhn[3]);
    store(fi, index_f(n   , t%2ul ? 3 : 4), fhn[4]);
    store(fi, index_f(j[5], t%2ul ? 6 : 5), fhn[5]);
    store(fi, index_f(n   , t%2ul ? 5 : 6), fhn[6]);
    store(fi, index_f(j[1], t%2ul ? 8 : 7), fhn[7]); // attention
    store(fi, index_f(j[3], t%2ul ? 7 : 8), fhn[8]); // attention
)+"#elif defined(D3Q15)"+R(
    uint x0, xp, xm, y0, yp, ym, z0, zp, zm;
    calculate_indices(n, &x0, &xp, &xm, &y0, &yp, &ym, &z0, &zp, &zm);
    store(fi, index_f(j[ 1]   , t%2ul ?  2 :  1), fhn[ 1]);
    store(fi, index_f(n       , t%2ul ?  1 :  2), fhn[ 2]);
    store(fi, index_f(j[ 3]   , t%2ul ?  4 :  3), fhn[ 3]);
    store(fi, index_f(n       , t%2ul ?  3 :  4), fhn[ 4]);
    store(fi, index_f(j[ 5]   , t%2ul ?  6 :  5), fhn[ 5]);
    store(fi, index_f(n       , t%2ul ?  5 :  6), fhn[ 6]);
    store(fi, index_f(j[ 7]   , t%2ul ?  8 :  7), fhn[ 7]);
    store(fi, index_f(n       , t%2ul ?  7 :  8), fhn[ 8]);
    store(fi, index_f(xp+yp+z0, t%2ul ? 10 :  9), fhn[ 9]); // attention, additional streaming direction required beyond velocity set
```

```
  store(fi, index_f(j[ 5]   , t%2ul ?  9 : 10), fhn[10]); // attention
  store(fi, index_f(xp+y0+zp, t%2ul ? 12 : 11), fhn[11]); // attention, additional
      ↪ streaming direction required beyond velocity set
  store(fi, index_f(j[ 3]   , t%2ul ? 11 : 12), fhn[12]); // attention
  store(fi, index_f(x0+yp+zp, t%2ul ? 14 : 13), fhn[13]); // attention, additional
      ↪ streaming direction required beyond velocity set
  store(fi, index_f(j[ 1]   , t%2ul ? 13 : 14), fhn[14]); // attention
)+"#elif defined(D3Q19)"+R(
  store(fi, index_f(j[ 1], t%2ul ?  2 :  1), fhn[ 1]);
  store(fi, index_f(n    , t%2ul ?  1 :  2), fhn[ 2]);
  store(fi, index_f(j[ 3], t%2ul ?  4 :  3), fhn[ 3]);
  store(fi, index_f(n    , t%2ul ?  3 :  4), fhn[ 4]);
  store(fi, index_f(j[ 5], t%2ul ?  6 :  5), fhn[ 5]);
  store(fi, index_f(n    , t%2ul ?  5 :  6), fhn[ 6]);
  store(fi, index_f(j[ 7], t%2ul ?  8 :  7), fhn[ 7]);
  store(fi, index_f(n    , t%2ul ?  7 :  8), fhn[ 8]);
  store(fi, index_f(j[ 9], t%2ul ? 10 :  9), fhn[ 9]);
  store(fi, index_f(n    , t%2ul ?  9 : 10), fhn[10]);
  store(fi, index_f(j[11], t%2ul ? 12 : 11), fhn[11]);
  store(fi, index_f(n    , t%2ul ? 11 : 12), fhn[12]);
  store(fi, index_f(j[ 1], t%2ul ? 14 : 13), fhn[13]); // attention
  store(fi, index_f(j[ 3], t%2ul ? 13 : 14), fhn[14]); // attention
  store(fi, index_f(j[ 1], t%2ul ? 16 : 15), fhn[15]); // attention
  store(fi, index_f(j[ 5], t%2ul ? 15 : 16), fhn[16]); // attention
  store(fi, index_f(j[ 3], t%2ul ? 18 : 17), fhn[17]); // attention
  store(fi, index_f(j[ 5], t%2ul ? 17 : 18), fhn[18]); // attention
)+"#elif defined(D3Q27)"+R(
  store(fi, index_f(j[ 1], t%2ul ?  2 :  1), fhn[ 1]);
  store(fi, index_f(n    , t%2ul ?  1 :  2), fhn[ 2]);
  store(fi, index_f(j[ 3], t%2ul ?  4 :  3), fhn[ 3]);
  store(fi, index_f(n    , t%2ul ?  3 :  4), fhn[ 4]);
  store(fi, index_f(j[ 5], t%2ul ?  6 :  5), fhn[ 5]);
  store(fi, index_f(n    , t%2ul ?  5 :  6), fhn[ 6]);
  store(fi, index_f(j[ 7], t%2ul ?  8 :  7), fhn[ 7]);
  store(fi, index_f(n    , t%2ul ?  7 :  8), fhn[ 8]);
  store(fi, index_f(j[ 9], t%2ul ? 10 :  9), fhn[ 9]);
  store(fi, index_f(n    , t%2ul ?  9 : 10), fhn[10]);
  store(fi, index_f(j[11], t%2ul ? 12 : 11), fhn[11]);
  store(fi, index_f(n    , t%2ul ? 11 : 12), fhn[12]);
  store(fi, index_f(j[ 1], t%2ul ? 14 : 13), fhn[13]); // attention
  store(fi, index_f(j[ 3], t%2ul ? 13 : 14), fhn[14]); // attention
  store(fi, index_f(j[ 1], t%2ul ? 16 : 15), fhn[15]); // attention
  store(fi, index_f(j[ 5], t%2ul ? 15 : 16), fhn[16]); // attention
  store(fi, index_f(j[ 3], t%2ul ? 18 : 17), fhn[17]); // attention
  store(fi, index_f(j[ 5], t%2ul ? 17 : 18), fhn[18]); // attention
  store(fi, index_f(j[19], t%2ul ? 20 : 19), fhn[19]);
  store(fi, index_f(n    , t%2ul ? 19 : 20), fhn[20]);
  store(fi, index_f(j[ 7], t%2ul ? 22 : 21), fhn[21]); // attention
  store(fi, index_f(j[ 5], t%2ul ? 21 : 22), fhn[22]); // attention
  store(fi, index_f(j[ 9], t%2ul ? 24 : 23), fhn[23]); // attention
  store(fi, index_f(j[ 3], t%2ul ? 23 : 24), fhn[24]); // attention
  store(fi, index_f(j[11], t%2ul ? 26 : 25), fhn[25]); // attention
  store(fi, index_f(j[ 1], t%2ul ? 25 : 26), fhn[26]); // attention
)+"#endif"+R( // D3Q27
}
```

Appendix A.5. Esoteric Pull

Listing A5. Esoteric Pull implementation in OpenCL C.

```
)+R(void load_f(const uint n, float* fhn, const global fpxx* fi, const uint* j, const
    ↪ ulong t) {
  fhn[0] = load(fi, index_f(n, 0u)); // Esoteric-Pull
  for(uint i=1u; i<def_velocity_set; i+=2u) {
    fhn[i   ] = load(fi, index_f(n   , t%2ul ? i    : i+1u));
    fhn[i+1u] = load(fi, index_f(j[i], t%2ul ? i+1u : i   ));
  }
}
)+R(void store_f(const uint n, const float* fhn, global fpxx* fi, const uint* j, const
    ↪ ulong t) {
  store(fi, index_f(n, 0u), fhn[0]); // Esoteric-Pull
  for(uint i=1u; i<def_velocity_set; i+=2u) {
    store(fi, index_f(j[i], t%2ul ? i+1u : i   ), fhn[i   ]);
    store(fi, index_f(n   , t%2ul ? i    : i+1u), fhn[i+1u]);
  }
}
```

Appendix A.6. Esoteric Push

Listing A6. Esoteric Push implementation in OpenCL C.

```
)+R(void load_f(const uint n, float* fhn, const global fpxx* fi, const uint* j, const ulong t) {
  fhn[0] = load(fi, index_f(n, 0u)); // Esoteric-Push
  for(uint i=1u; i<def_velocity_set; i+=2u) {
    fhn[i   ] = load(fi, index_f(j[i+1u], t%2ul ? i+1u : i   ));
    fhn[i+1u] = load(fi, index_f(n      , t%2ul ? i    : i+1u));
  }
}
)+R(void store_f(const uint n, const float* fhn, global fpxx* fi, const uint* j, const ulong t) {
  store(fi, index_f(n, 0u), fhn[0]); // Esoteric-Push
  for(uint i=1u; i<def_velocity_set; i+=2u) {
    store(fi, index_f(n      , t%2ul ? i    : i+1u), fhn[i   ]);
    store(fi, index_f(j[i+1u], t%2ul ? i+1u : i   ), fhn[i+1u]);
  }
}
```

Appendix B. OpenCL C Implementation of the FSLBM with Esoteric Pull

Listing A7. OpenCL C FSLBM implementation as in *FluidX3D* [13,14,24].

```
#define TYPE_W 0b00000001 // static boundary (wall)
#define TYPE_E 0b00000010 // equilibrium boundary or moving bounce-back boundary (inflow/outflow)
#define TYPE_F 0b00001000 // fluid
#define TYPE_I 0b00010000 // interface
#define TYPE_G 0b00100000 // gas

#define TYPE_IF 0b00011000 // change from interface to fluid
#define TYPE_IG 0b00110000 // change from interface to gas
#define TYPE_GI 0b00111000 // change from gas to interface
#define TYPE_SU 0b00111000 // any flag bit used for SURFACE

#define fpxx float // switchable data type (regular 32-bit float)
#define load(p,o) p[o] // regular float read
#define store(p,o,x) p[o]=x // regular float write

)+R(ulong index_f(const uint n, const uint i) { // 64-bit indexing (maximum 2^32 lattice points (1624^3 lattice resolution, 225GB)
  return (ulong)i*(ulong)def_N+(ulong)n; // SoA (229% faster on GPU)
}

)+R(void load_f(const uint n, float* fhn, const global fpxx* fi, const uint* j, const ulong t) {
  fhn[0] = load(fi, index_f(n, 0u)); // Esoteric-Pull
  for(uint i=1u; i<def_velocity_set; i+=2u) {
    fhn[i   ] = load(fi, index_f(n    , t%2ul ? i    : i+1u));
    fhn[i+1u] = load(fi, index_f(j[i], t%2ul ? i+1u : i   ));
  }
}
)+R(void store_f(const uint n, const float* fhn, global fpxx* fi, const uint* j, const ulong t) {
  store(fi, index_f(n, 0u), fhn[0]); // Esoteric-Pull
  for(uint i=1u; i<def_velocity_set; i+=2u) {
    store(fi, index_f(j[i], t%2ul ? i+1u : i   ), fhn[i   ]);
    store(fi, index_f(n    , t%2ul ? i    : i+1u), fhn[i+1u]);
  }
}

)+"#ifdef SURFACE"+R(
)+R(void load_f_outgoing(const uint n, float* fon, const global fpxx* fi, const uint* j, const ulong t) { // load outgoing DDFs, even: 1:1 like stream-out odd, odd: 1:1 like stream-out even
  for(uint i=1u; i<def_velocity_set; i+=2u) { // Esoteric-Pull
    fon[i   ] = load(fi, index_f(j[i], t%2ul ? i    : i+1u));
    fon[i+1u] = load(fi, index_f(n    , t%2ul ? i+1u : i   ));
  }
}
)+R(void store_f_reconstructed(const uint n, const float* fhn, global fpxx* fi, const uint* j, const ulong t, const uchar* flagsj) { // store reconstructed gas DDFs, even: 1:1 like stream-in even, odd: 1:1 like stream-in odd
  for(uint i=1u; i<def_velocity_set; i+=2u) { // Esoteric-Pull
    if(flagsj[i+1u]&TYPE_G) store(fi, index_f(n    , t%2ul ? i    : i+1u), fhn[i   ]); // only store reconstructed gas DDFs to locations from which
```

```
45      if(flagsj[i   ]&TYPE_G) store(fi, index_f(j[i], t%2ul ? i+1u : i   ), fhn[i+1u]); //
            ↪  they are going to be streamed in during next stream_collide()
46    }
47  }
48  )+"#endif"+R( // SURFACE
49
50
51
52  )+"kernel void initialize(global fpxx* fi, global float* rho, global float* u, global
        ↪ uchar* flags"+R( // ) {
53  )+"#ifdef SURFACE"+R(
54    , global float* mass, global float* massex, global float* phi // argument order is
          ↪ important
55  )+"#endif"+R( // SURFACE
56  )+") {"+R( // initialize()
57    const uint n = get_global_id(0); // n = x+(y+z*Ny)*Nx
58    uchar flagsn = flags[n];
59    uint j[def_velocity_set]; // neighbor indices
60    neighbors(n, j); // calculate neighbor indices
61    uchar flagsj[def_velocity_set]; // cache neighbor flags for multiple readings
62    for(uint i=1u; i<def_velocity_set; i++) flagsj[i] = flags[j[i]];
63    if(flagsn&TYPE_W) { // node is wall
64      bool TYPE_ONLY_W = true; // has only wall neighbors
65      for(uint i=1u; i<def_velocity_set; i++) if(!(flagsj[i]&TYPE_W)) TYPE_ONLY_W =
            ↪ TYPE_ONLY_W&&(flagsj[i]&TYPE_W);
66      if(TYPE_ONLY_W) {
67        u[                        n] = 0.0f; // reset velocity for wall lattice points
              ↪ with only boundary neighbors
68        u[    (ulong)def_N+(ulong)n] = 0.0f;
69        u[2ul*(ulong)def_N+(ulong)n] = 0.0f;
70      }
71  )+"#ifndef MOVING_BOUNDARIES"+R(
72      if(flagsn&TYPE_W) {
73        u[                        n] = 0.0f; // reset velocity for all wall lattice points
74        u[    (ulong)def_N+(ulong)n] = 0.0f;
75        u[2ul*(ulong)def_N+(ulong)n] = 0.0f;
76      }
77  )+"#else"+R( // MOVING_BOUNDARIES
78    } else if(!(flagsn&TYPE_T)) { // local lattice point is not wall
79      bool next_to_moving_boundary = false;
80      for(uint i=1u; i<def_velocity_set; i++) {
81        next_to_moving_boundary |= (u[j[i]]!=0.0f||u[(ulong)def_N+(ulong)j[i]]!=0.0f||u[2
              ↪ ul*(ulong)def_N+(ulong)j[i]]!=0.0f)&&(flagsj[i]&TYPE_W);
82      }
83      if(next_to_moving_boundary) flagsn = flagsn = flagsn|TYPE_E; // only flag lattice
            ↪ points next to a wall with non-zero velocity with TYPE_E
84  )+"#endif"+R( // MOVING_BOUNDARIES
85    }
86    float feq[def_velocity_set]; // f_equilibrium
87    calculate_f_eq(rho[n], u[n], u[(ulong)def_N+(ulong)n], u[2ul*(ulong)def_N+(ulong)n],
          ↪ feq);
88  )+"#ifdef SURFACE"+R( // automatically generate the interface layer between fluid and
        ↪ gas
89    { // separate block to avoid variable name conflicts
90      float phin = phi[n];
91      if(!(flagsn&(TYPE_W|TYPE_E|TYPE_T|TYPE_F|TYPE_I))) flagsn = (flagsn&~TYPE_SU)|TYPE_G
            ↪ ; // change all non-fluid and non-interface flags to gas
92      if(flagsn&TYPE_G) { // node is gas
93        bool change = false; // check if node has to be changed to interface
94        for(uint i=1u; i<def_velocity_set; i++) change = change||(flagsj[i]&TYPE_F); // if
              ↪  neighbor flag fluid is set, the node must be interface
95        if(change) { // create interface automatically if phi has not explicitely defined
              ↪ for the interface layer
96          flagsn = (flagsn&~TYPE_SU)|TYPE_I; // node must be interface
97          phin = 0.5f;
98          float rhon, uxn, uyn, uzn; // initialize interface nodes with average density/
                ↪ velocity of fluid neighbors
99          average_neighbors_fluid(n, rho, u, flags, &rhon, &uxn, &uyn, &uzn); // get
                ↪ average rho/u from all fluid neighbors
100         calculate_f_eq(rhon, uxn, uyn, uzn, feq); // calculate equilibrium DDFs
101       }
102     }
103     if(flagsn&TYPE_G) { // node is still gas
104       u[                        n] = 0.0f; // reset velocity for gas nodes
105       u[    (ulong)def_N+(ulong)n] = 0.0f;
106       u[2ul*(ulong)def_N+(ulong)n] = 0.0f;
107       phin = 0.0f;
108     } else if((flagsn&TYPE_I) && (phin<0.0f||phin>1.0f)) {
109       phin = 0.5f; // node should be interface, but phi was invalid
110     } else if(flagsn&TYPE_F) {
111       phin = 1.0f;
```

```
112       }
113       phi[n] = phin;
114       mass[n] = phin*rho[n];
115       massex[n] = 0.0f; // reset excess mass
116       flags[n] = flagsn;
117     }
118   )+"#endif"+R( // SURFACE
119     store_f(n, feq, fi, j, 1ul); // write to fi
120   } // initialize()
121
122
123
124   )+"kernel void stream_collide(global fpxx* fi, global float* rho, global float* u,
          ↪ global uchar* flags, const ulong t"+R( // ) {  // for all but SURFACE, flags is
          ↪ const
125   )+"#ifdef SURFACE"+R(
126     , global float* mass // argument order is important
127   )+"#endif"+R( // SURFACE
128   )+") {"+R( // stream_collide()
129     const uint n = get_global_id(0); // n = x+(y+z*Ny)*Nx
130     const uchar flagsn = flags[n]; // cache flags[n] for multiple readings
131     if(flagsn&(TYPE_W|TYPE_G)) return; // if node is boundary or gas, just return
132
133     uint j[def_velocity_set]; // neighbor indices
134     neighbors(n, j); // calculate neighbor indices
135
136   )+"#if defined(MOVING_BOUNDARIES)||defined(SURFACE)"+R( // don't leave any spaces in #if
          ↪  arguments
137     uchar flagsj[def_velocity_set]; // only required for MOVING_BOUNDARIES, SURFACE or
          ↪ TEMPERATURE
138     flagsj[0] = flagsn;
139     for(uint i=1u; i<def_velocity_set; i++) flagsj[i] = flags[j[i]];
140   )+"#endif"+R( // MOVING_BOUNDARIES or SURFACE
141
142     float fhn[def_velocity_set]; // local DDFs
143     load_f(n, fhn, fi, j, t); // perform streaming (part 1)
144
145   )+"#ifdef MOVING_BOUNDARIES"+R(
146     if(flagsn&TYPE_E) { // apply Dirichlet velocity boundaries if necessary (reads
          ↪ velocities of only neighboring boundary nodes, which do not change during
          ↪ simulation)
147       uint ji;
148       for(uint i=1u; i<def_velocity_set; i+=2u) { // Krueger p.180, rho_wall=1, loop is
            ↪ totally unrolled by compiler
149         const float w6 = -6.0f*w(i); // w(i) = w(i+1) if i is odd
150         ji = j[i+1u]; fhn[i   ] = flagsj[i+1u]&TYPE_W ? fma(w6, c(i+1u)*u[ji]+c(
              ↪ def_velocity_set+i+1u)*u[(ulong)def_N+(ulong)ji]+c(2u*def_velocity_set+i+1u
              ↪ )*u[2ul*(ulong)def_N+(ulong)ji], fhn[i   ]) : fhn[i   ]; // boundary :
              ↪ regular
151         ji = j[i   ]; fhn[i+1u] = flagsj[i   ]&TYPE_W ? fma(w6, c(i   )*u[ji]+c(
              ↪ def_velocity_set+i   )*u[(ulong)def_N+(ulong)ji]+c(2u*def_velocity_set+i
              ↪ )*u[2ul*(ulong)def_N+(ulong)ji], fhn[i+1u]) : fhn[i+1u];
152       }
153     }
154   )+"#endif"+R( // MOVING_BOUNDARIES
155
156     float rhon, uxn, uyn, uzn; // calculate local density and velocity for collision
157   )+"#ifndef EQUILIBRIUM_BOUNDARIES"+R(
158     calculate_rho_u(fhn, &rhon, &uxn, &uyn, &uzn); // calculate density and velocity
          ↪ fields from fi
159   )+"#else"+R( // EQUILIBRIUM_BOUNDARIES
160     if(flagsn&TYPE_E) {
161       rhon = rho[                      n]; // apply preset velocity/density
162       uxn  = u[                        n];
163       uyn  = u[    (ulong)def_N+(ulong)n];
164       uzn  = u[2ul*(ulong)def_N+(ulong)n];
165     } else {
166       calculate_rho_u(fhn, &rhon, &uxn, &uyn, &uzn); // calculate density and velocity
            ↪ fields from fi
167     }
168   )+"#endif"+R( // EQUILIBRIUM_BOUNDARIES
169     float fxn=def_fx, fyn=def_fy, fzn=def_fz; // force starts as constant volume force,
          ↪ can be modified before call of calculate_forcing_terms(...)
170     float Fin[def_velocity_set]; // forcing terms
171
172     { // separate block to avoid variable name conflicts
173   )+"#ifdef VOLUME_FORCE"+R( // apply force and collision operator, write to fi in video
          ↪ memory
174       const float rho2 = 0.5f/rhon; // apply external volume force (Guo forcing, Krueger p
            ↪ .233f)
```

```
    uxn = clamp(fma(fxn, rho2, uxn), -def_c, def_c); // limit velocity (for stability purposes)
    uyn = clamp(fma(fyn, rho2, uyn), -def_c, def_c); // force term: F*dt/(2*rho)
    uzn = clamp(fma(fzn, rho2, uzn), -def_c, def_c);
    calculate_forcing_terms(uxn, uyn, uzn, fxn, fyn, fzn, Fin); // calculate volume force terms Fin from velocity field (Guo forcing, Krueger p.233f)
)+"#else"+R( // VOLUME_FORCE
    uxn = clamp(uxn, -def_c, def_c); // limit velocity (for stability purposes)
    uyn = clamp(uyn, -def_c, def_c); // force term: F*dt/(2*rho)
    uzn = clamp(uzn, -def_c, def_c);
    for(uint i=0u; i<def_velocity_set; i++) Fin[i] = 0.0f;
)+"#endif"+R( // VOLUME_FORCE
  }

)+"#ifdef SURFACE"+R(
  if(flagsn&TYPE_I) { // node was interface, eventually initiate flag change
    bool TYPE_NO_F=true, TYPE_NO_G=true; // temporary flags for no fluid or gas neighbors
    for(uint i=1u; i<def_velocity_set; i++) {
      const uchar flagsji_su = flags[j[i]]&TYPE_SU; // extract SURFACE flags
      TYPE_NO_F = TYPE_NO_F&&flagsji_su!=TYPE_F;
      TYPE_NO_G = TYPE_NO_G&&flagsji_su!=TYPE_G;
    }
    const float massn = mass[n]; // load mass
         if(massn>rhon || TYPE_NO_G) flags[n] = (flagsn&~TYPE_SU)|TYPE_IF; // set flag interface->fluid
    else if(massn<0.0f || TYPE_NO_F) flags[n] = (flagsn&~TYPE_SU)|TYPE_IG; // set flag interface->gas
  }
)+"#endif"+R( // SURFACE

)+"#ifndef EQUILIBRIUM_BOUNDARIES"+R(
)+"#ifdef UPDATE_FIELDS"+R(
  rho[                      n] = rhon; // update density field
  u[                        n] = uxn; // update velocity field
  u[    (ulong)def_N+(ulong)n] = uyn;
  u[2ul*(ulong)def_N+(ulong)n] = uzn;
)+"#endif"+R( // UPDATE_FIELDS
)+"#else"+R( // EQUILIBRIUM_BOUNDARIES
)+"#ifdef UPDATE_FIELDS"+R(
  if(!(flagsn&TYPE_E)) {
    rho[                      n] = rhon; // update density field
    u[                        n] = uxn; // update velocity field
    u[    (ulong)def_N+(ulong)n] = uyn;
    u[2ul*(ulong)def_N+(ulong)n] = uzn;
  }
)+"#endif"+R( // UPDATE_FIELDS
)+"#endif"+R( // EQUILIBRIUM_BOUNDARIES

  float feq[def_velocity_set]; // equilibrium DDFs
  calculate_f_eq(rhon, uxn, uyn, uzn, feq); // calculate equilibrium DDFs

)+"#ifdef VOLUME_FORCE"+R(
  const float c_tau = fma(def_w, -0.5f, 1.0f);
  for(uint i=0u; i<def_velocity_set; i++) Fin[i] *= c_tau;
)+"#endif"+R( // VOLUME_FORCE
)+"#ifndef EQUILIBRIUM_BOUNDARIES"+R(
  for(uint i=0u; i<def_velocity_set; i++) fhn[i] = fma(1.0f-def_w, fhn[i], fma(def_w, feq[i], Fin[i])); // perform collision (SRT)
)+"#else"+R( // EQUILIBRIUM_BOUNDARIES
  for(uint i=0u; i<def_velocity_set; i++) fhn[i] = flagsn&TYPE_E ? feq[i] : fma(1.0f-def_w, fhn[i], fma(def_w, feq[i], Fin[i])); // perform collision (SRT)
)+"#endif"+R( // EQUILIBRIUM_BOUNDARIES

  store_f(n, fhn, fi, j, t); // perform streaming (part 2)
} // stream_collide()



)+"#ifdef SURFACE"+R(
)+R(kernel void surface_0(global fpxx* fi, const global float* rho, const global float* u, const global uchar* flags, global float* mass, const global float* massex, const global float* phi, const ulong t) { // capture outgoing DDFs before streaming
  const uint n = get_global_id(0); // n = x+(y+z*Ny)*Nx
  const uchar flagsn = flags[n]; // cache flags[n] for multiple readings
  if(flagsn&(TYPE_W|TYPE_G)) return; // node processed here is fluid or interface

  uint j[def_velocity_set]; // neighbor indices
  neighbors(n, j); // calculate neighbor indices
  float fhn[def_velocity_set]; // incoming DDFs
```

```
246     load_f(n, fhn, fi, j, t); // load incoming DDFs
247     float fon[def_velocity_set]; // outgoing DDFs
248     fon[0] = fhn[0]; // fon[0] is already loaded in fhn[0]
249     load_f_outgoing(n, fon, fi, j, t); // load outgoing DDFs
250
251     float massn = mass[n];
252     for(uint i=1u; i<def_velocity_set; i++) {
253       massn += massex[j[i]]; // distribute excess mass from last step which is stored in
            ↪ neighbors
254     }
255     if(flagsn&TYPE_F) { // node is fluid
256       for(uint i=1u; i<def_velocity_set; i++) massn += fhn[i]-fon[i]; // neighbor is fluid
            ↪  or interface node
257     } else if(flagsn&TYPE_I) { // node is interface
258       float phij[def_velocity_set]; // cache fill level of neighbor lattice points
259       for(uint i=1u; i<def_velocity_set; i++) phij[i] = phi[j[i]]; // cache fill level of
            ↪ neighbor lattice points
260       float rhon, uxn, uyn, uzn, rho_laplace=0.0f; // no surface tension if rho_laplace is
            ↪  not overwritten later
261   )+"#ifndef EQUILIBRIUM_BOUNDARIES"+R(
262       calculate_rho_u(fon, &rhon, &uxn, &uyn, &uzn); // calculate density and velocity
            ↪ fields from fon (not fhn)
263   )+"#else"+R( // EQUILIBRIUM_BOUNDARIES
264       if(flagsn&TYPE_E) {
265         rhon = rho[                      n]; // apply preset velocity/density
266         uxn  = u[                        n];
267         uyn  = u[    (ulong)def_N+(ulong)n];
268         uzn  = u[2ul*(ulong)def_N+(ulong)n];
269       } else {
270         calculate_rho_u(fon, &rhon, &uxn, &uyn, &uzn); // calculate density and velocity
              ↪ fields from fon (not fhn)
271       }
272   )+"#endif"+R( // EQUILIBRIUM_BOUNDARIES
273       uxn = clamp(uxn, -def_c, def_c); // limit velocity (for stability purposes)
274       uyn = clamp(uyn, -def_c, def_c);
275       uzn = clamp(uzn, -def_c, def_c);
276       phij[0] = calculate_phi(rhon, massn, flagsn); // don't load phi[n] from memory,
            ↪ instead recalculate it with mass corrected by excess mass
277       rho_laplace = def_6_sigma==0.0f ? 0.0f : def_6_sigma*curvature_calculation(n, phij,
            ↪ phi); // surface tension least squares fit (PLIC, most accurate)
278       float feg[def_velocity_set]; // reconstruct f from neighbor gas lattice points
279       const float rho2tmp = 0.5f/rhon; // apply external volume force (Guo forcing,
            ↪ Krueger p.233f)
280       const float uxntmp = clamp(fma(def_fx, rho2tmp, uxn), -def_c, def_c); // limit
            ↪ velocity (for stability purposes)
281       const float uyntmp = clamp(fma(def_fy, rho2tmp, uyn), -def_c, def_c); // force term:
            ↪  F*dt/(2*rho)
282       const float uzntmp = clamp(fma(def_fz, rho2tmp, uzn), -def_c, def_c);
283       calculate_f_eq(1.0f-rho_laplace, uxntmp, uyntmp, uzntmp, feg); // calculate gas
            ↪ equilibrium DDFs with constant ambient pressure
284       uchar flagsj[def_velocity_set]; // cache neighbor flags for multiple readings
285       for(uint i=1u; i<def_velocity_set; i++) flagsj[i] = flags[j[i]];
286       for(uint i=1u; i<def_velocity_set; i+=2u) { // calculate mass exchange between
            ↪ current node and fluid/interface nodes
287         massn += flagsj[i   ]&(TYPE_F|TYPE_I) ? flagsj[i   ]&TYPE_F ? fhn[i+1]-fon[i   ] :
              ↪  0.5f*(phij[i   ]+phij[0])*(fhn[i+1 ]-fon[i   ]) : 0.0f; // neighbor is
              ↪ fluid or interface node
288         massn += flagsj[i+1u]&(TYPE_F|TYPE_I) ? flagsj[i+1u]&TYPE_F ? fhn[i  ]-fon[i+1u] :
              ↪  0.5f*(phij[i+1u]+phij[0])*(fhn[i   ]-fon[i+1u]) : 0.0f; // fluid :
              ↪ interface : gas
289       }
290       for(uint i=1u; i<def_velocity_set; i+=2u) { // calculate reconstructed gas DDFs
291         fhn[i   ] = feg[i+1u]-fon[i+1u]+feg[i   ];
292         fhn[i+1u] = feg[i   ]-fon[i   ]+feg[i+1u];
293       }
294       store_f_reconstructed(n, fhn, fi, j, t, flagsj); // store reconstructed gas DDFs
            ↪ that are streamed in during the following stream_collide()
295     }
296     mass[n] = massn;
297   }
298   )+R(kernel void surface_1(global uchar* flags) { // prevent neighbors from interface->
        ↪ fluid nodes to become/be gas nodes
299     const uint n = get_global_id(0); // n = x+(y+z*Ny)*Nx
300     const uchar flagsn_su = flags[n]&(TYPE_SU|TYPE_W); // extract SURFACE flags
301     if(flagsn_su==TYPE_IF) { // flag interface->fluid is set
302       uint j[def_velocity_set]; // neighbor indices
303       neighbors(n, j); // calculate neighbor indices
304       for(uint i=1u; i<def_velocity_set; i++) {
305         const uchar flagsji = flags[j[i]];
306         const uchar flagsji_su = flagsji&(TYPE_SU|TYPE_W); // extract SURFACE flags
307         const uchar flagsji_r = flagsji&~TYPE_SU; // extract all non-SURFACE flags
```

```
308         if(flagsji_su==TYPE_IG) flags[j[i]] = flagsji_r|TYPE_I; // prevent interface neighbor nodes from becoming gas
309         else if(flagsji_su==TYPE_G) flags[j[i]] = flagsji_r|TYPE_GI; // neighbor node was gas and must change to interface
310       }
311     }
312   } // possible types at the end of surface_1(): TYPE_F / TYPE_I / TYPE_G / TYPE_IF / TYPE_IG / TYPE_GI
313   )+R(kernel void surface_2(global fpxx* fi, const global float* rho, const global float* u, global uchar* flags, const ulong t) {  // apply flag changes and calculate excess mass
314     const uint n = get_global_id(0); // n = x+(y+z*Ny)*Nx
315     const uchar flagsn_su = flags[n]&(TYPE_SU|TYPE_W); // extract SURFACE flags
316     if(flagsn_su==TYPE_GI) { // initialize the fi of gas nodes that should become interface
317       float rhon, uxn, uyn, uzn; // average over all fluid/interface neighbors
318       average_neighbors_non_gas(n, rho, u, flags, &rhon, &uxn, &uyn, &uzn); // get average rho/u from all fluid/interface neighbors
319       float feq[def_velocity_set];
320       calculate_f_eq(rhon, uxn, uyn, uzn, feq); // calculate equilibrium DDFs
321       uint j[def_velocity_set];
322       neighbors(n, j);
323       store_f(n, feq, fi, j, t); // write feq to fi in video memory
324     } else if(flagsn_su==TYPE_IG) { // flag interface->gas is set
325       uint j[def_velocity_set]; // neighbor indices
326       neighbors(n, j); // calculate neighbor indices
327       for(uint i=1u; i<def_velocity_set; i++) {
328         const uchar flagsji = flags[j[i]];
329         const uchar flagsji_su = flagsji&(TYPE_SU|TYPE_W); // extract SURFACE flags
330         const uchar flagsji_r = flagsji&~TYPE_SU; // extract all non-SURFACE flags
331         if(flagsji_su==TYPE_F||flagsji_su==TYPE_IF) {
332           flags[j[i]] = flagsji_r|TYPE_I; // prevent fluid or interface neighbors that turn to fluid from being/becoming fluid
333         }
334       }
335     }
336   } // possible types at the end of surface_2(): TYPE_F / TYPE_I / TYPE_G / TYPE_IF / TYPE_IG / TYPE_GI
337   )+R(kernel void surface_3(const global float* rho, global uchar* flags, global float* mass, global float* massex, global float* phi) { // apply flag changes and calculate excess mass
338     const uint n = get_global_id(0); // n = x+(y+z*Ny)*Nx
339     const uchar flagsn_su = flags[n]&(TYPE_SU|TYPE_W); // extract SURFACE flags
340     if(flagsn_su&TYPE_W) return;
341     const float rhon = rho[n]; // density of node n
342     float massn = mass[n]; // mass of node n
343     float massexn = 0.0f; // excess mass of node n
344     float phin = 0.0f;
345     if(flagsn_su==TYPE_F) { // regular fluid node
346       massexn = massn-rhon; // dump mass-rho difference into excess mass
347       massn = rhon; // fluid node mass has to equal rho
348       phin = 1.0f;
349     } else if(flagsn_su==TYPE_I) { // regular interface node
350       massexn = massn>rhon ? massn-rhon : massn<0.0f ? massn : 0.0f; // allow interface nodes with mass>rho or mass<0
351       massn = clamp(massn, 0.0f, rhon);
352       phin = calculate_phi(rhon, massn, TYPE_I); // calculate fill level for next step (only necessary for interface nodes)
353     } else if(flagsn_su==TYPE_G) { // regular gas node
354       massexn = massn; // dump remaining mass into excess mass
355       massn = 0.0f;
356       phin = 0.0f;
357     } else if(flagsn_su==TYPE_IF) { // flag interface->fluid is set
358       flags[n] = (flags[n]&~TYPE_SU)|TYPE_F; // node becomes fluid
359       massexn = massn-rhon; // dump mass-rho difference into excess mass
360       massn = rhon; // fluid node mass has to equal rho
361       phin = 1.0f; // set phi[n] to 1.0f for fluid nodes
362     } else if(flagsn_su==TYPE_IG) { // flag interface->gas is set
363       flags[n] = (flags[n]&~TYPE_SU)|TYPE_G; // node becomes gas
364       massexn = massn; // dump remaining mass into excess mass
365       massn = 0.0f; // gas mass has to be zero
366       phin = 0.0f; // set phi[n] to 0.0f for gas nodes
367     } else if(flagsn_su==TYPE_GI) { // flag gas->interface is set
368       flags[n] = (flags[n]&~TYPE_SU)|TYPE_I; // node becomes interface
369       massexn = massn>rhon ? massn-rhon : massn<0.0f ? massn : 0.0f; // allow interface nodes with mass>rho or mass<0
370       massn = clamp(massn, 0.0f, rhon);
371       phin = calculate_phi(rhon, massn, TYPE_I); // calculate fill level for next step (only necessary for interface nodes)
372     }
373     uint j[def_velocity_set]; // neighbor indices
```

```
374     neighbors(n, j); // calculate neighbor indices
375     uint counter = 0u; // count (fluid|interface) neighbors
376     for(uint i=1u; i<def_velocity_set; i++) { // simple model: distribute excess mass
            ↪ equally to all interface and fluid neighbors
377       const uchar flagsji_su = flags[j[i]]&(TYPE_SU|TYPE_W); // extract SURFACE flags
378       counter += (uint)(flagsji_su==TYPE_F||flagsji_su==TYPE_I||flagsji_su==TYPE_IF||
              ↪ flagsji_su==TYPE_GI); // avoid branching
379     }
380     massn += counter>0u ? 0.0f : massexn; // if excess mass can't be distributed to
            ↪ neighboring interface or fluid nodes, add it to local mass (ensure mass
            ↪ conservation)
381     massexn = counter>0u ? massexn/(float)counter : 0.0f; // divide excess mass up for all
            ↪  interface or fluid neighbors
382     mass[n] = massn; // update mass
383     massex[n] = massexn; // update excess mass
384     phi[n] = phin; // update phi
385   } // possible types at the end of surface_3(): TYPE_F / TYPE_I / TYPE_G
386   )+"#endif"+R( // SURFACE
```

Appendix C. Setup Script for the Raindrop Impact Simulation

Listing A8. C++ setup script for the raindrop impact simulation in Figure 8.

```
1   #include "lbm.hpp"
2   void main_setup() { // raindrop setup (#define D3Q19, SRT, VOLUME_FORCE,
        ↪ EQUILIBRIUM_BOUNDARIES, SURFACE, FP16C)
3     const int box_diameter = 940;
4     float drop_diameter = box_diameter/5;
5     const int select_drop_size = 12;
6     const float alpha_sim = 20.0f;
7
8     if(drop_diameter==-1.0f) drop_diameter = 0.1f*(float)box_diameter;
9     const float scale = (float)box_diameter/(10.0f*drop_diameter); // 256.0f/400.0f; //
          ↪ 1.0f;
10
11    // rain drop parameters from "Effects of Altitude on Maximum Raindrop Size and Fall
          ↪ Velocity as Limited by Collisional Breakup, Fig. 3" in SI-units
12    float const si_nu = 1.0508E-6f; // kinematic shear viscosity [m^2/s] at 20C and 35g/l
          ↪ salinity
13    const float si_rho = 1024.8103f; // fluid density [kg/m^3] at 20C and 35g/l salinity
14    const float si_sigma = 73.81E-3f; // fluid surface tension [kg/s^2] at 20C and 35g/l
          ↪ salinity
15    const float si_g = 9.81f; // gravitational acceleration [m/s^2]
16    const float alpha = alpha_sim; // impact angle [degree], 0 = vertical
17    const float si_Ds[] = { 1.0E-3f, 1.5E-3f, 2.0E-3f, 2.5E-3f, 3.0E-3f, 3.5E-3f, 4.0E-3f,
          ↪  4.5E-3f, 5.0E-3f, 5.5E-3f, 6.0E-3f, 6.5E-3f, 7.0E-3f, 4.1E-3f };
18    const float si_us[] = { 4.50f,   5.80f,   6.80f,   7.55f,   8.10f,   8.45f,   8.80f,
          ↪    9.05f,   9.20f,   9.30f,   9.40f,   9.45f,   9.55f,   7.21f };
19    const float si_D = si_Ds[select_drop_size]; // drop diameter [m] (1-7mm)
20    const float si_u = si_us[select_drop_size]; // impact velocity [m/s] (4.50-9.55m/s)
21    const float si_H  =  4.0f*si_D*scale; // liquid pool height [m] (4*D is sufficient for
          ↪  deep pool)
22    const float si_Lx = 10.0f*si_D*scale; // simulation box width [m]
23    const float si_Lz =  8.5f*si_D*scale; // simulation box height [m]
24
25    // determine a length, a velocity and the mean density in simulation units
26    const float Lx = (float)box_diameter; // simulation box width
27    const float u = 0.05f; // impact velocity
28    const float rho = 1.0f; // density
29    units.set_m_kg_s(Lx, u, rho, si_Lx, si_u, si_rho); // calculate 3 independent
          ↪ conversion factors (m, kg, s)
30    print_info("D  = "+to_string(si_D, 6u));
31    print_info("Re = "+to_string(units.si_Re(si_D, si_u, si_nu), 6u));
32    print_info("We = "+to_string(units.si_We(si_D, si_u, si_rho, si_sigma), 6u));
33    print_info("Fr = "+to_string(units.si_Fr(si_D, si_u, si_g), 6u));
34    print_info("Ca = "+to_string(units.si_Ca(si_u, si_rho, si_nu, si_sigma), 6u));
35    print_info("Bo = "+to_string(units.si_Bo(si_D, si_rho, si_g, si_sigma), 6u));
36    print_info("10ms = "+to_string(units.t(0.01f))+" LBM steps");
37    const float nu = units.nu(si_nu); // calculate values for remaining parameters in
          ↪ simulation units
38    const float sigma = units.sigma(si_sigma);
39    const float f = units.f(si_rho, si_g); // force per volume
40    const float Lz = 800;//units.x(si_Lz);
41    const float H = units.x(si_H);
42    const float R = 0.5f*units.x(si_D); // drop radius
43    // define simulation box size, viscosity and volume force
44    LBM lbm(to_uint(Lx), to_uint(Lx), to_uint(Lz), nu, 0.0f, 0.0f, -f, sigma, 0.5f); //
          ↪ largest box size on Titan Xp with FP32: 384^2, FP16: 464^3
45
```

```
46    const uint N=lbm.get_N(), Nx=lbm.get_Nx(), Ny=lbm.get_Ny(), Nz=lbm.get_Nz(); for(uint
          ↪ n=0u, x=0u, y=0u, z=0u; n<N; n++, lbm.coordinates(n, x, y, z)) {
47      // define geometry
48      lbm.rho[n] = rho; // set density everywhere
49      if(sphere(x, y, z, float3(0.5f*Nx, 0.5f*Ny-2.0f*R*tan(alpha*pif/180.0f), H+R+2.5f),
            ↪ R+2.0f)) {
50        const float b = sphere_plic(x, y, z, float3(0.5f*Nx, 0.5f*Ny-2.0f*R*tan(alpha*pif
              ↪ /180.0f), H+R+2.5f), R);
51        if(b!=-1.0f) {
52          lbm.u.y[n] =  sin(alpha*pif/180.0f)*u;//+random_symmetric(0.1f); // break
                ↪ symmetry by initializing with noise
53          lbm.u.z[n] = -cos(alpha*pif/180.0f)*u;//+random_symmetric(0.1f); // break
                ↪ symmetry by initializing with noise
54          if(b==1.0f) {
55            lbm.flags[n] = TYPE_F;
56            lbm.phi[n] = 1.0f;
57          } else {
58            lbm.flags[n] = TYPE_I;
59            lbm.phi[n] = b;
60          }
61        }
62      }
63      if(z==0) lbm.flags[n] = TYPE_W;
64      else if(z==H) {
65        lbm.flags[n] = TYPE_I;
66        lbm.phi[n] = 0.5f; // not strictly necessary, but should be clearer (phi is
              ↪ automatically initialized to 0.5f for TYPE_I if not initialized)
67      } else if(z<H) lbm.flags[n] = TYPE_F;
68      else if(z==Nz-1) { // make drops that hit the simulation box ceiling disappear
69        lbm.rho[n] = 0.5f;
70        lbm.flags[n] = TYPE_E;
71      }
72    }
73    lbm.run(0u);
74    key_1 = false; // turn off boundary
75    key_6 = true; // turn on surface raytracing
76    lbm.graphics.set_camera(-30.0f, 20.0f, 100.0f, 1.0f);
77    Clock clock;
78
79    // image
80    lbm.run(7700u/*units.t(0.0015f)*/);
81    print_info("compute time: "+print_time(clock.stop()));
82    clock.start();
83    lbm.graphics.write_frame();
84    print_info("render time: "+to_string(clock.stop(), 3u));
85
86    // video
87    //lbm.graphics.write_frame();
88    //while(units.si_t(lbm.get_t())<0.0015f) {
89    //  lbm.run(units.t(0.0015f)/300u);
90    //  lbm.graphics.write_frame();
91    //}
92    //print_info("compute + render time: "+to_string(clock.stop(), 3u));
93  } /**/
```

References

1. Krüger, T.; Kusumaatmaja, H.; Kuzmin, A.; Shardt, O.; Silva, G.; Viggen, E.M. *The Lattice Boltzmann Method*; Springer International Publishing: Cham, Switzerland, 2017; Volume 10, pp. 978–983.
2. Geier, M.; Schönherr, M. Esoteric twist: An efficient in-place streaming algorithmus for the lattice Boltzmann method on massively parallel hardware. *Computation* **2017**, *5*, 19. [CrossRef]
3. 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.
4. Mohrhard, M.; Thäter, G.; Bludau, J.; Horvat, B.; Krause, M. An Auto-Vecotorization Friendly Parallel Lattice Boltzmann Streaming Scheme for Direct Addressing. *Comput. Fluids* **2019**, *181*, 1–7. [CrossRef]
5. Kummerländer, A.; Dorn, M.; Frank, M.; Krause, M.J. Implicit Propagation of Directly Addressed Grids in Lattice Boltzmann Methods. *Comput. Fluids* **2021**. [CrossRef]
6. Schreiber, M.; Neumann, P.; Zimmer, S.; Bungartz, H.J. Free-surface lattice-Boltzmann simulation on many-core architectures. *Procedia Comput. Sci.* **2011**, *4*, 984–993. [CrossRef]
7. Riesinger, C.; Bakhtiari, A.; Schreiber, M.; Neumann, P.; Bungartz, H.J. A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters. *Computation* **2017**, *5*, 48. [CrossRef]
8. Aksnes, E.O.; Elster, A.C. Porous rock simulations and lattice Boltzmann on GPUs. In *Parallel Computing: From Multicores and GPU's to Petascale*; IOS Press: Amsterdam, The Netherlands, 2010; pp. 536–545.

9. Holzer, M.; Bauer, M.; Rüde, U. Highly Efficient Lattice-Boltzmann Multiphase Simulations of Immiscible Fluids at High-Density Ratios on CPUs and GPUs through Code Generation. *arXiv* **2020**, arXiv:2012.06144.
10. Duchateau, J.; Rousselle, F.; Maquignon, N.; Roussel, G.; Renaud, C. Accelerating physical simulations from a multicomponent Lattice Boltzmann method on a single-node multi-GPU architecture. In Proceedings of the 2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), Krakow, Poland, 4–6 November 2015; pp. 315–322.
11. Li, W.; Ma, Y.; Liu, X.; Desbrun, M. Efficient Kinetic Simulation of Two-Phase Flows. *ACM Trans. Graph.* **2022**, *41*, 114.
12. Walsh, S.D.; Saar, M.O.; Bailey, P.; Lilja, D.J. Accelerating geoscience and engineering system simulations on graphics hardware. *Comput. Geosci.* **2009**, *35*, 2353–2364. [CrossRef]
13. Lehmann, M.; Krause, M.J.; Amati, G.; Sega, M.; Harting, J.; Gekle, S. On the accuracy and performance of the lattice Boltzmann method with 64-bit, 32-bit and novel 16-bit number formats. *arXiv* **2021**, arXiv:2112.08926.
14. Lehmann, M. High Performance Free Surface LBM on GPUs. Master's Thesis, University of Bayreuth, Bayreuth, Germany, 2019.
15. Takáč, M.; Petráš, I. Cross-Platform GPU-Based Implementation of Lattice Boltzmann Method Solver Using ArrayFire Library. *Mathematics* **2021**, *9*, 1793. [CrossRef]
16. 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]
17. Delbosc, N.; Summers, J.L.; Khan, A.; Kapur, N.; Noakes, C.J. 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]
18. Tran, N.P.; Lee, M.; Hong, S. Performance optimization of 3D lattice Boltzmann flow solver on a GPU. *Sci. Program.* **2017**, 1205892. [CrossRef]
19. Obrecht, C.; Kuznik, F.; Tourancheau, B.; Roux, J.J. Multi-GPU implementation of the lattice Boltzmann method. *Comput. Math. Appl.* **2013**, *65*, 252–261. [CrossRef]
20. Obrecht, C.; Kuznik, F.; Tourancheau, B.; Roux, J.J. A new approach to the lattice Boltzmann method for graphics processing units. *Comput. Math. Appl.* **2011**, *61*, 3628–3638. [CrossRef]
21. Feichtinger, C.; Habich, J.; Köstler, H.; Hager, G.; Rüde, U.; Wellein, G. A flexible Patch-based lattice Boltzmann parallelization approach for heterogeneous GPU–CPU clusters. *Parallel Comput.* **2011**, *37*, 536–549. [CrossRef]
22. Calore, E.; Gabbana, A.; Kraus, J.; Pellegrini, E.; Schifano, S.F.; Tripiccione, R. Massively parallel lattice–Boltzmann codes on large GPU clusters. *Parallel Comput.* **2016**, *58*, 1–24. [CrossRef]
23. Obrecht, C.; Kuznik, F.; Tourancheau, B.; Roux, J.J. Global memory access modelling for efficient implementation of the lattice Boltzmann method on graphics processing units. In Proceedings of the International Conference on High Performance Computing for Computational Science, Berkeley, CA, USA, 22–25 June 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 151–161.
24. Lehmann, M.; Oehlschlägel, L.M.; Häusl, F.P.; Held, A.; Gekle, S. Ejection of marine microplastics by raindrops: A computational and experimental study. *Microplastics Nanoplastics* **2021**, *1*, 18. [CrossRef]
25. Laermanns, H.; Lehmann, M.; Klee, M.; Löder, M.G.; Gekle, S.; Bogner, C. Tracing the horizontal transport of microplastics on rough surfaces. *Microplastics Nanoplastics* **2021**, *1*, 11. [CrossRef]
26. Lehmann, M.; Gekle, S. Analytic Solution to the Piecewise Linear Interface Construction Problem and Its Application in Curvature Calculation for Volume-of-Fluid Simulation Codes. *Computation* **2022**, *10*, 21. [CrossRef]
27. Häusl, F. MPI-Based Multi-GPU Extension of the Lattice Boltzmann Method. Bachelor's Thesis, University of Bayreuth, Bayreuth, Germay, 2019.
28. Häusl, F. Soft Objects in Newtonian and Non-Newtonian Fluids: A Computational Study of Bubbles and Capsules in Flow. Master's Thesis, University of Bayreuth, Bayreuth, Germay, 2021.
29. Limbach, H.J.; Arnold, A.; Mann, B.A.; Holm, C. ESPResSo—An extensible simulation package for research on soft matter systems. *Comput. Phys. Commun.* **2006**, *174*, 704–727. [CrossRef]
30. Institute for Computational Physics, Universität Stuttgart. ESPResSo User's Guide. 2016. Available online: http://espressomd.org/wordpress/wp-content/uploads/2016/07/ug_07_2016.pdf (accessed on 15 June 2018).
31. Hong, P.Y.; Huang, L.M.; Lin, L.S.; Lin, C.A. Scalable multi-relaxation-time lattice Boltzmann simulations on multi-GPU cluster. *Comput. Fluids* **2015**, *110*, 1–8. [CrossRef]
32. Xian, W.; Takayuki, A. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Comput.* **2011**, *37*, 521–535. [CrossRef]
33. Ho, M.Q.; Obrecht, C.; Tourancheau, B.; de Dinechin, B.D.; Hascoet, J. Improving 3D Lattice Boltzmann Method stencil with asynchronous transfers on many-core processors. In Proceedings of the 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC), San Diego, CA, USA, 10–12 December 2017; pp. 1–9.
34. Habich, J.; Feichtinger, C.; Köstler, H.; Hager, G.; Wellein, G. Performance engineering for the lattice Boltzmann method on GPGPUs: Architectural requirements and performance results. *Comput. Fluids* **2013**, *80*, 276–282. [CrossRef]
35. 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]
36. 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; pp. 825–834.

37. de Oliveira, W.B., Jr.; Lugarini, A.; Franco, A.T. Performance analysis of the lattice Boltzmann method implementation on GPU. In Proceedings of the XL Ibero-Latin-American Congress on Computational Methods in Engineering, ABMEC, Natal, Brazil, 11–14 November 2019.
38. Rinaldi, P.R.; Dari, E.; Vénere, M.J.; Clausse, A. A Lattice-Boltzmann solver for 3D fluid simulation on GPU. *Simul. Model. Pract. Theory* **2012**, *25*, 163–171. [CrossRef]
39. Rinaldi, P.R.; Dari, E.A.; Vénere, M.J.; Clausse, A. Fluid Simulation with Lattice Boltzmann Methods Implemented on GPUs Using CUDA. In Proceedings of the HPCLatAm 2009, Buenos Aires, Argentina, 26–27 August 2009.
40. Ames, J.; Puleri, D.F.; Balogh, P.; Gounley, J.; Draeger, E.W.; Randles, A. Multi-GPU immersed boundary method hemodynamics simulations. *J. Comput. Sci.* **2020**, *44*, 101153. [CrossRef]
41. Xiong, Q.; Li, B.; Xu, J.; Fang, X.; Wang, X.; Wang, L.; He, X.; Ge, W. Efficient parallel implementation of the lattice Boltzmann method on large clusters of graphic processing units. *Chin. Sci. Bull.* **2012**, *57*, 707–715. [CrossRef]
42. Zhu, H.; Xu, X.; Huang, G.; Qin, Z.; Wen, B. An Efficient Graphics Processing Unit Scheme for Complex Geometry Simulations Using the Lattice Boltzmann Method. *IEEE Access* **2020**, *8*, 185158–185168. [CrossRef]
43. Kuznik, F.; Obrecht, C.; Rusaouen, G.; Roux, J.J. LBM based flow simulation using GPU computing processor. *Comput. Math. Appl.* **2010**, *59*, 2380–2392. [CrossRef]
44. Horga, A. With Lattice Boltzmann Models Using CUDA Enabled GPGPUs. Master's Thesis, University of Timsoara, Timsoara, Romania, 2013.
45. Geveler, M.; Ribbrock, D.; Göddeke, D.; Turek, S. Lattice-Boltzmann simulation of the shallow-water equations with fluid-structure interaction on multi-and manycore processors. In *Facing the Multicore-Challenge*; Springer: Wiesbaden, Germany, 2010; pp. 92–104.
46. Beny, J.; Latt, J. Efficient LBM on GPUs for dense moving objects using immersed boundary condition. *arXiv* **2019**, arXiv:1904.02108.
47. Tekic, P.M.; Radjenovic, J.B.; Rackovic, M. Implementation of the Lattice Boltzmann method on heterogeneous hardware and platforms using OpenCL. *Adv. Electr. Comput. Eng.* **2012**, *12*, 51–56. [CrossRef]
48. Bény, J.; Kotsalos, C.; Latt, J. Toward full GPU implementation of fluid-structure interaction. In Proceedings of the 2019 18th International Symposium on Parallel and Distributed Computing (ISPDC), Amsterdam, The Netherlands, 3–7 June 2019; pp. 16–22.
49. Boroni, G.; Dottori, J.; Rinaldi, P. FULL GPU implementation of lattice-Boltzmann methods with immersed boundary conditions for fast fluid simulations. *Int. J. Multiphysics* **2017**, *11*, 1–14.
50. Griebel, M.; Schweitzer, M.A. *Meshfree Methods for Partial Differential Equations II*; Springer: Cham, Switzerland, 2005.
51. Zitz, S.; Scagliarini, A.; Harting, J. Lattice Boltzmann simulations of stochastic thin film dewetting. *Phys. Rev. E* **2021**, *104*, 034801. [CrossRef] [PubMed]
52. 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]
53. Habich, J.; Zeiser, T.; Hager, G.; Wellein, G. Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA. *Adv. Eng. Softw.* **2011**, *42*, 266–272. [CrossRef]
54. Calore, E.; Marchi, D.; Schifano, S.F.; Tripiccione, R. Optimizing communications in multi-GPU Lattice Boltzmann simulations. In Proceedings of the 2015 International Conference on High Performance Computing & Simulation (HPCS), Amsterdam, The Netherlands, 20–24 July 2015; pp. 55–62.
55. Onodera, N.; Idomura, Y.; Uesawa, S.; Yamashita, S.; Yoshida, H. Locally mesh-refined lattice Boltzmann method for fuel debris air cooling analysis on GPU supercomputer. *Mech. Eng. J.* **2020**, *7*, 19–00531. [CrossRef]
56. Falcucci, G.; Amati, G.; Fanelli, P.; Krastev, V.K.; Polverino, G.; Porfiri, M.; Succi, S. Extreme flow simulations reveal skeletal adaptations of deep-sea sponges. *Nature* **2021**, *595*, 537–541. [CrossRef]
57. Zitz, S.; Scagliarini, A.; Maddu, S.; Darhuber, A.A.; Harting, J. Lattice Boltzmann method for thin-liquid-film hydrodynamics. *Phys. Rev. E* **2019**, *100*, 033313. [CrossRef]
58. Wei, C.; Zhenghua, W.; Zongzhe, L.; Lu, Y.; Yongxian, W. An improved LBM approach for heterogeneous GPU-CPU clusters. In Proceedings of the 2011 4th International Conference on Biomedical Engineering and Informatics (BMEI), Shanghai, China, 15–17 October 2011; Volume 4, pp. 2095–2098.
59. Gray, F.; Boek, E. Enhancing computational precision for lattice Boltzmann schemes in porous media flows. *Computation* **2016**, *4*, 11. [CrossRef]
60. Wellein, G.; Lammers, P.; Hager, G.; Donath, S.; Zeiser, T. Towards optimal performance for lattice Boltzmann applications on terascale computers. In *Parallel Computational Fluid Dynamics 2005*; Elsevier: Amsterdam, The Netherlands, 2006; pp. 31–40.
61. Wittmann, M.; Zeiser, T.; Hager, G.; Wellein, G. Comparison of different propagation steps for lattice Boltzmann methods. *Comput. Math. Appl.* **2013**, *65*, 924–935. [CrossRef]
62. Wittmann, M. Hardware-effiziente, hochparallele Implementierungen von Lattice-Boltzmann-Verfahren für komplexe Geometrien. Ph.D. Thesis, Friedrich-Alexander-Universität, Erlangen, Germany, 2016.
63. Krause, M. Fluid Flow Simulation and Optimisation with Lattice Boltzmann Methods on High Performance Computers: Application to the Human Respiratory System. Ph.D. Thesis, Karlsruhe Institute of Technology (KIT), Universität Karlsruhe (TH), Karlsruhe, Germany, 2010. Available online: https://publikationen.bibliothek.kit.edu/1000019768 (accessed on 20 February 2019).

64. Succi, S.; Amati, G.; Bernaschi, M.; Falcucci, G.; Lauricella, M.; Montessori, A. Towards exascale lattice Boltzmann computing. *Comput. Fluids* **2019**, *181*, 107–115. [CrossRef]
65. D'Humières, D. Multiple–relaxation–time lattice Boltzmann models in three dimensions. *Philos. Trans. R. Soc. London. Ser. A Math. Phys. Eng. Sci.* **2002**, *360*, 437–451. [CrossRef]
66. Latt, J. *Technical Report: How to Implement Your DdQq Dynamics with Only q Variables per Node (Instead of 2q)*; Tufts University: Medford, MA, USA, 2007; pp. 1–8.
67. Bogner, S.; Rüde, U.; Harting, J. Curvature estimation from a volume-of-fluid indicator function for the simulation of surface tension and wetting with a free-surface lattice Boltzmann method. *Phys. Rev. E* **2016**, *93*, 043302. [CrossRef] [PubMed]
68. Crane, K.; Llamas, I.; Tariq, S. *Real-Time Simulation and Rendering of 3d Fluids*; GPU gems 3.1, Addison-Wesley Professional: Boston, MA, USA, 2007; Volume 3.
69. Gerace, S. A Model Integrated Meshless Solver (MIMS) for Fluid Flow and Heat Transfer. Ph.D. Thesis, University of Central Florida, Orlando, FL, USA, 2010.
70. Lynch, C.E. *Advanced CFD Methods for Wind Turbine Analysis*; Georgia Institute of Technology: Atlanta, GA, USA, 2011.
71. Keßler, A. Matrix-Free Voxel-Based Finite Element Method for Materials with Heterogeneous Microstructures. Ph.D. Thesis, der Bauhaus-Universität Weimar, Weimar, Germnay, 2019.