Accuracy and performance of the lattice Boltzmann method with 64-bit, 32-bit, and customized 16-bit number formats

Moritz Lehmann,[1,*] Mathias J. Krause,[2] Giorgio Amati,[3] Marcello Sega,[4] Jens Harting,[4,5] and Stephan Gekle[1]

[1]*Biofluid Simulation and Modeling–Theoretische Physik VI, University of Bayreuth, Bayreuth, Germany*
[2]*Institute of Mechanical Process Engineering and Mechanics, Karlsruhe Institute of Technology, Karlsruhe, Germany*
[3]*CINECA, SCAI–SuperComputing Applications and Innovation Department, Rome Branch, Italy*
[4]*Helmholtz Institute Erlangen-Nürnberg for Renewable Energy, Erlangen, Germany*
[5]*Department of Chemical and Biological Engineering and Department of Physics, Friedrich-Alexander-Universität, Erlangen, Germany*

(Received 12 January 2022; accepted 7 June 2022; published 26 July 2022)

Fluid dynamics simulations with the lattice Boltzmann method (LBM) are very memory intensive. Alongside reduction in memory footprint, significant performance benefits can be achieved by using FP32 (single) precision compared to FP64 (double) precision, especially on GPUs. Here we evaluate the possibility to use even FP16 and posit16 (half) precision for storing fluid populations, while still carrying arithmetic operations in FP32. For this, we first show that the commonly occurring number range in the LBM is a lot smaller than the FP16 number range. Based on this observation, we develop customized 16-bit formats—based on a modified IEEE-754 and on a modified posit standard—that are specifically tailored to the needs of the LBM. We then carry out an in-depth characterization of LBM accuracy for six different test systems with increasing complexity: Poiseuille flow, Taylor-Green vortices, Karman vortex streets, lid-driven cavity, a microcapsule in shear flow (utilizing the immersed-boundary method), and, finally, the impact of a raindrop (based on a volume-of-fluid approach). We find that the difference in accuracy between FP64 and FP32 is negligible in almost all cases, and that for a large number of cases even 16-bit is sufficient. Finally, we provide a detailed performance analysis of all precision levels on a large number of hardware microarchitectures and show that significant speedup is achieved with mixed FP32/16-bit.

DOI: 10.1103/PhysRevE.106.015308

I. INTRODUCTION

The lattice Boltzmann method (LBM) [1–4] is a powerful tool to simulate fluid flow. The parallel nature of the underlying algorithm has led to (multi-)GPU implementations [5–62], becoming a popular choice as speedup can be up to two orders of magnitude compared to CPUs at similar power consumption. However, most GPUs have only poor FP64 (double precision) arithmetic capabilities[1] and thus the vast majority of GPU codes have been implemented in FP32 (single precision), while most CPU codes are written in FP64. This difference, and, in particular, whether FP32 is sufficient for LBM simulations compared to FP64, has been a point of persistent discussion within the LBM community [15–20,31–36,52–58,60,63–65]. Nevertheless, only a few papers [19,35,36,52,60,66] provide some comparison on how floating-point formats affect the accuracy of the LBM and mostly find only insignificant differences between FP64 and FP32 except at very low velocity and where floating-point round-off leads to spontaneous symmetry breaking. Besides the question of accuracy, a quantitative performance comparison across different hardware microarchitectures is missing, as the vast majority of LBM software is either written only for CPUs [67–79] or only for Nvidia GPUs [30–56] or CPUs and Nvidia GPUs [18–29].

A second point of concern has been the amount of video memory on GPUs, which is in general smaller than standard memory on CPU systems and can thus lead to restrictions in domain size. LBM solely works on density distribution functions (DDFs) f_i (also called fluid populations)—floating-point numbers [80–83]—that need to be loaded from and written to video memory in every time step. These DDFs take up the majority of the consumed memory. If wanting to reduce the memory footprint of LBM with reduced floating-point precision, it comes to mind to store the DDFs in a lower precision number format (streaming step) while doing arithmetic in higher (floating-point) precision (collision step). This is equivalent to decoupling arithmetic precision and memory precision [84,85]. As a desirable side effect, since the limiting factor regarding compute time is memory bandwidth [12–21,30–45,52–55,59,60,63,64,67,86–88], lower precision DDFs also vastly increase performance. Such a mixed precision variant, where arithmetic is done in FP64 and DDF storage in FP32, has already been used by Refs. [35,57]. Using FP32 arithmetic and FP16 DDF storage would be even better, but has not yet been attempted due to concerns about possibly

[*]Corresponding author: moritz.lehmann@uni-bayreuth.de

[1]As of June 2022, the only GPUs with >2 TFLOPs/s in FP64 are H100, MI250(X), MI210, A100, CMP 170HX, MI100, A30, V100(S), Titan V, GV100, MI60, MI50, Radeon Pro VII, GP100, P100, Radeon VII, W9100, and W8100. All other data-center, gaming, and pro GPUs have limited FP64 capabilities.

insufficient accuracy. Lower 16-bit precision has already been successfully applied to other fluid solvers [89–91] and to a lot of other high-performance computing software [92,93].

The purpose of this paper is thus twofold: First, to render mixed FP32/16-bit precisions feasible for LBM, we introduce customized 16-bit number formats that turn out to be superior to standard IEEE-754 FP16 in LBM applications and in many cases perform as accurately as FP32. Therein, we leverage that some of the FP32 bits do not contain physical information or are entirely unused, similar to Ref. [89]. This approach requires minimal code interventions and can be easily combined with any velocity set, collision operator, swap algorithm, or LBM extension. In addition to using custom-built floating-point formats, we show that shifting the DDFs by subtracting the lattice weights and computing the equilibrium DDFs in a specific order of operations as originally proposed by Skordos [66] and further investigated by He and Luo [94] and Gray and Boek [60]—an optimization beneficial across all floating-point formats and already widely used [6–12,24–26,31,35,53,60,66,68–76,88,94]—turns out absolutely crucial for the 16-bit compression.

Second, we present an extensive comparison of FP64, FP32, FP16, shifted posit16 as well as our customized formats. Regarding LBM accuracy, we study Poiseuille flow through a cylinder [95], Taylor-Green vortex energy dissipation [66,96], Karman vortices [97] from flow around a cylinder, lid-driven cavity [30,33,37,39,49,52,98–103], deformation of a microcapsule in shear flow [104–106] with the immersed-boundary method (IBM) extension, and microplastic particle transport during a raindrop impact [10] with the volume-of-fluid and IBM extensions. Regarding performance, we exploit the capability of our FluidX3D LBM implementation written in OpenCL [6–12] to provide benchmarks for all floating-point variants on a large variety of hardware, from the world's fastest datacenter GPU over various consumer GPUs and CPUs from different vendors to even a mobile phone ARM system-on-a-chip (SoC), and show roofline analysis [64,87,107] for one hardware example.

II. LATTICE BOLTZMANN ALGORITHM

A. LBM—overview

The LBM is a Navier-Stokes flow solver that discretizes space into a Cartesian lattice and time into discrete time steps [1–4]. For each point on the lattice, density ρ and velocity $\vec{u}$ of the flow are computed from so-called density distribution functions (DDFs) f_i (also called fluid populations). The DDFs are floating-point numbers and represent how many fluid molecules move between neighboring lattice points in each time step. Because of the lattice, only certain directions are possible for this exchange and there are various levels of this directional discretization, in 3D typically 19 (including the center point), where space-diagonal directions are left out. After exchange of DDFs from and to neighboring lattice points (streaming), the DDFs are redistributed locally on each lattice point (collision). For the collision, there are various approaches, the most common being the single-relaxation-time (SRT), two-relaxation-time (TRT), and multirelaxation-time (MRT) collision operators [1,12].

The computation of the streaming part is done by copying the DDFs in memory to their new location. The algorithm is provided in Appendix A 2 a with notation as in Appendix A 3.

B. DDF-shifting

To achieve maximum accuracy, it is essential not to work with the DDFs f_i directly, but with shifted $f_i^{\text{shifted}} := f_i - w_i$ instead [53,60,66,88,94]. $w_i = f_i^{\text{eq}}(\rho = 1, \vec{u} = 0)$ are the lattice weights and ρ and $\vec{u}$ are the local fluid density and velocity. This requires a small change in the equilibrium DDF computation,

$$f_i^{\text{eq-shifted}}(\rho,\vec{u}) := f_i^{\text{eq}}(\rho,\vec{u}) - w_i \tag{1}$$

$$= w_i\,\rho\cdot\left(\frac{(\vec{u}\circ\vec{c}_i)^2}{2\,c^4} + \frac{\vec{u}\circ\vec{c}_i}{c^2} + 1 - \frac{\vec{u}\circ\vec{u}}{2\,c^2}\right) - w_i \tag{2}$$

$$= w_i\,\rho\cdot\left(\frac{(\vec{u}\circ\vec{c}_i)^2}{2\,c^4} - \frac{\vec{u}\circ\vec{u}}{2\,c^2} + \frac{\vec{u}\circ\vec{c}_i}{c^2}\right) + w_i\,(\rho-1), \tag{3}$$

and density summation:

$$\rho = \sum_i \left(f_i^{\text{shifted}} + w_i\right) = \left(\sum_i f_i^{\text{shifted}}\right) + 1. \tag{4}$$

We emphasize that it is key to choose Eq. (3) exactly as presented without changing the order of operations,[2] otherwise the accuracy may not be enhanced at all [60,66,94]. With this exact order, the round-off error due to different sums is minimized. This offers a large benefit, most prominently on FP16 accuracy, by substantially reducing numerical loss of significance at no additional computational cost. Since it is also beneficial for regular FP32 accuracy, it is already widely used in LBM codes such as our FluidX3D [6–12], OpenLB [68–71], ESPResSo [24–26], Palabos [72–76], and some versions of waLBerla [53]. In Appendix A 2, we provide the entire algorithm without and with DDF-shifting for comparison and in Appendix A 3 we clarify our notation.

We also recommend doing the summation of the DDFs in alternating + and − order during computation of the velocity $\vec{u}$ to further reduce numerical loss of significance, for example, $u_x = (f_1 - f_2 + f_7 - f_8 + f_9 - f_{10} + f_{13} - f_{14} + f_{15} - f_{16})/\rho$ for the x component in D3Q19.

Gray and Boek [60] also proposed computing $(\rho - 1) = \sum_i f_i^{\text{shifted}}$ as a separate variable and directly inserting it into Eq. (3); while we do not advise against this, we found its benefit to be insignificant at any floating-point precision while increasing complexity of the code and thus omit it in our implementation.

Although without DDF-shifting, the equation for the equilibria dictates the number distribution of the DDFs, with DDF-shifting applied, the DDFs are always centered around zero. Higher-order equilibria definitions such as Refs. [108–110], an alternative to Eq. (3), are likely to work as well with 16-bit compression if DDF-shifting is applied, but further validation is required.

[2]To minimize the overall number of floating-point operations, terms should be precomputed such that $f_i^{\text{eq-shifted}} = A\cdot(\frac{1}{2}\,(B^2 + C) \pm B) + D$ requires only three fused-multiply-add (FMA) operations.

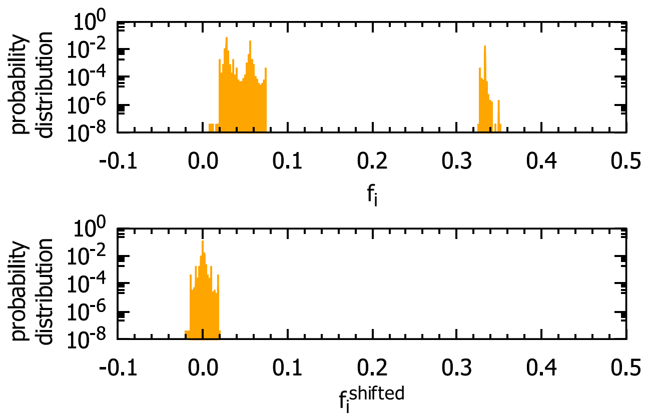


FIG. 1. Histogram of the DDFs for the lid-driven cavity simulation from Sec. IV D (Re = 1000, Ma = 0.17, grid resolution 128^3) after 100 000 LBM time steps. The simulation is performed without the DDF-shifting (top) and with DDF-shifting (bottom), both times in FP32/FP32.

C. Which range of numbers does the LBM use?

In Fig. 1, we present the distribution of f_i and f_i^{shifted} for the example system of the lid-driven cavity from Sec. IV D. A more detailed look at the DDF distributions of this system are provided in Figs. 19 and 20 in the Appendix. Similar data for the remaining setups are given in Appendix Fig. 21. It is quite remarkable how the number range in all cases is very limited. The f_i accumulate around the LBM lattice weights (for D3Q19 $w_i \in \{\frac{1}{36}, \frac{1}{18}, \frac{1}{3}\}$) and the f_i^{shifted} accumulate around 0, where floating-point accuracy is best. So for FP32 not only are the trailing bits of the mantissa expected to be nonphysical numerical noise [89], but also some bits of the exponent are entirely unused, meaning one can waive these bits without losing accuracy.

To find the theoretical maximum number range of f_i and f_i^{shifted}, we insert $\vec{u}_j = c\,\frac{\vec{c}_j}{|\vec{c}_j|}$ in Eqs. (A4) and (3) and find that $\frac{1}{\rho}\,|f_i^{\text{eq}}| \leqslant \delta$ or $\frac{1}{\rho}\,|f_i^{\text{eq-shifted}}| \leqslant \delta^{\text{shifted}}$, respectively, with the values of δ and δ^{shifted} depending on the velocity set in use (Table I).

With $\tau > 0.5$, through Eq. (A5), we get in the worst case

$$|f_i| \lessapprox |2\,f_i^{\text{eq}}| \lessapprox 2\,\rho\,\delta, \tag{5}$$

$$|f_i^{\text{shifted}}| \lessapprox |2\,f_i^{\text{eq-shifted}}| \lessapprox 2\,\rho\,\delta^{\text{shifted}}, \tag{6}$$

respectively, because the DDFs in stable simulations are expected to follow the equilibrium DDFs. The density ρ typically deviates only little from $\rho \approx 1$. Assuming $\rho < 2$ leads to $|f_i| \lessapprox 2$ being the worst-case maximum number range (D3Q13, no DDF-shifting). With the more typical D3Q19 and DDF-shifting, the same number range $|f_i^{\text{shifted}}| \lessapprox 2$ restricts the density to a less strict $\rho < 6$. Keeping the sign is required because f_i^{shifted} (and also f_i) can be negative.

$|f_i^{\text{shifted}}| \lessapprox 2$ and the resulting $\rho < 6$ is even sufficient for covering a fairly large class of compressible flows. The shock simulations in Ref. [108], for example, range in density from 0.6 to 2.2, so these simulations could possibly work as well. However, careful considerations need to be made for the individual setup to not exceed this limit. If a higher value for density is required, the floating-point formats with limited range could be shifted toward higher numbers; however, careful validation is required as this comes at the cost of worse accuracy at small numbers. The later proposed posit formats P16$_0$S and P16$_1$S do not put a limit on density, so they would be a better fit for simulations with large density variation.

TABLE I. The numerical value of δ and δ^{shifted} depending on the used velocity set.

	D2Q9	D3Q7	D3Q13	D3Q15	D3Q19	D3Q27
δ	0.45	0.47	0.50	0.42	0.34	0.30
δ^{shifted}	0.31	0.35	0.25	0.31	0.17	0.21

III. NUMBER REPRESENTATION MODELS

A 16-bit number can represent only 65 536 different values. The task is to spread these along the number axis in a way that the most commonly used numbers are represented with the best possible accuracy. There is a variety of number representations that come to mind as a 16-bit storage format: fixed-point, floating-point as well as the recently developed posit format [111], and each of them can be adjusted specifically for the LBM. Figure 2 illustrates the number formats investigated in this paper and Fig. 3 shows their accuracy characteristics.

A. Floating-point

1. Overview

In the normalized number range, a floating-point number [80–83] is represented as

$$x = \underbrace{(-1)^s}_{\text{sign}} \cdot \underbrace{2^{e-b}}_{\text{exponent}} \cdot \underbrace{(1 + 2^{-n_m}\,m)}_{\text{mantissa}}, \tag{7}$$

with s being the sign bit, e being an integer representing the exponent, and m being an integer representing the mantissa. b is a constant called exponent bias and n_m is the number of bits in the mantissa (values in Table II). The precision is $\log_{10}(2^{n_m+1})$ decimal digits[3] and the truncation error is $\epsilon = 2^{-n_m}$.

When the exponent e is zero, the mantissa is shifted to the right as a way to represent even smaller numbers close to zero, although at less precision. This is called the denormalized number range and making use of it during the conversions that will be described below is not straightforward, but essential alongside correct rounding to keep decent accuracy with the 16-bit formats.

2. Customized FP16 formats for the LBM

In our lattice Boltzmann simulations, we implement and test three different 16-bit floating-point formats:

(1) FP16: Standard IEEE-754 FP16, with FP32 ↔ FP16 conversion supported on all CPUs and GPUs from within the last 12 years.

[3]The +1 refers to the implicit leading bit of the mantissa.

TABLE II. Comparing the properties of the number formats used here to store the LBM DDFs f_i.

	Bits	n_m	b	Digits	ϵ	Range	Smallest normalized number	Smallest denormalized number
IEEE FP64	64	52	1023	16.0	2.2×10^{-16}	$\pm1.797693\times10^{308}$	2.225074×10^{-308}	4.940656×10^{-324}
IEEE FP32	32	23	127	7.2	1.2×10^{-7}	$\pm3.402823\times10^{38}$	1.175494×10^{-38}	1.401298×10^{-45}
IEEE FP16	16	10	15	3.3	9.8×10^{-4}	$\pm6.550400\times10^{4}$	6.103516×10^{-5}	5.960464×10^{-8}
FP16S	16	10	30	3.3	9.8×10^{-4}	$\pm1.999023\times10^{0}$	1.864464×10^{-9}	1.818989×10^{-12}
FP16C	16	11	15	3.6	4.9×10^{-4}	$\pm1.999512\times10^{0}$	6.103516×10^{-5}	2.980232×10^{-8}
Posit P16$_0$S	16	0–13	–	$\leqslant4.2$	$\geqslant1.2\times10^{-4}$	$\pm1.280000\times10^{2}$	–	4.768372×10^{-7}
Posit P16$_1$S	16	0–12	0	$\leqslant3.9$	$\geqslant2.4\times10^{-4}$	$\pm2.097152\times10^{6}$	–	2.910383×10^{-11}
Posit P16$_2$C	16	0–12	0	$\leqslant3.9$	$\geqslant2.4\times10^{-4}$	$\pm1.999756\times10^{0}$	–	1.734724×10^{-18}

(2) FP16S: We downscale the number range of IEEE-754 FP16 by $\times2^{-15}$ to ±2 and use the convenience that all modern CPUs and GPUs can do IEEE-754 floating-point conversion in hardware.

(3) FP16C: We allocate one bit less for the exponent (to decrease number range towards small numbers) and one bit more for the mantissa (to gain accuracy). The number range is also limited to ±2. This custom format requires manual conversion from and to FP32.

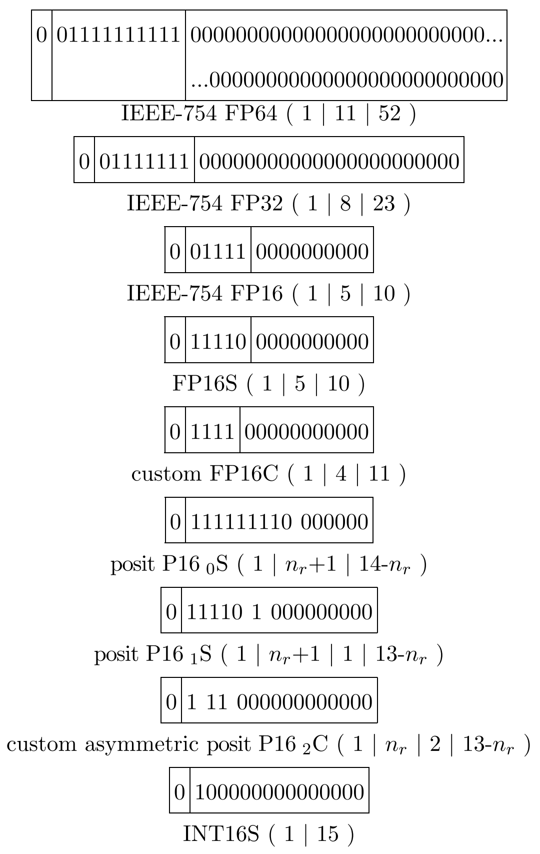


FIG. 2. The number 1.0 represented by the different formats we investigate here. The leftmost single bit is the sign s and the rightmost segment is the mantissa m. For floating-point (FP), the center segment is the exponent e. FP16S and FP16C are new formats specifically designed to store the DDFs. Fixed-point (INT16S) does not have an exponent. Posits have dynamic partitioning of the segments, with an extra regime segment and an optional exponent segment next to the mantissa.

When looking at Table II and Fig. 3, FP16S and FP16C differ in extended range toward small numbers versus halved truncation error ϵ. The question arises which of these two traits is more important for LBM. FP16 is inferior to both FP16S and FP16C as it combines lower mantissa accuracy and less range toward small numbers. Since FP16S comes at no additional computational cost and complexity compared to FP16, FP16S should always be preferred over FP16 for storing the DDFs.

3. *Floating-point conversion:* FP32 $\leftrightarrow$ FP16S

The IEEE-754 FP32 $\leftrightarrow$ FP16/FP16S conversion is supported in hardware and therefore only briefly described below.

FP32 → FP16S: For the FP32 → FP16 conversion, OpenCL provides the function `vstore_half_rte` that is executed in hardware. To convert to the FP16S format instead, we shift the number range up by 2^{15} via regular FP32 multiplication right before conversion. This is equivalent to increasing the exponent bias b by 15.

FP16S → FP32: For the FP16 → FP32 conversion, OpenCL provides the function `vload_half` that is executed in hardware. To convert from the FP16S format instead, we

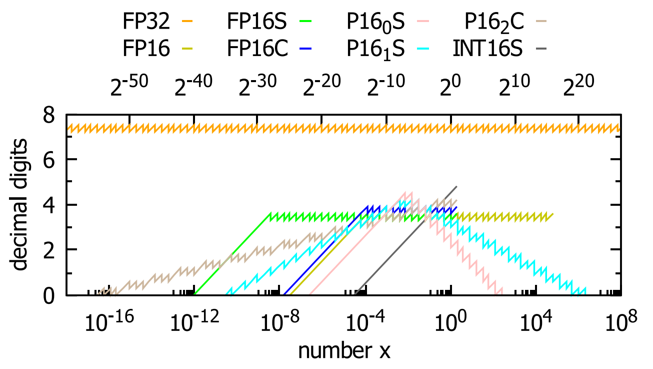


FIG. 3. Accuracy characteristics of the number formats investigated in this paper. This plot shows only the local minima (measured graphs see Fig. 18). FP16C reduces number range but increases accuracy in the normalized regime (horizontal part). For floating-point formats, the downward slope indicates the denormalized part, where accuracy behaves like fixed point. We also show 16-bit fixed-point scaled by $\times2^{-14}$ (INT16S). Posits (P16) have slopes left and right, with highest accuracy in the middle, which here is shifted from 1 to $\frac{1}{128}$, hence the "S". P16$_0$S/P16$_1$S have 0/1-bit exponents, making the slopes more (less) steep and decreasing (increasing) number range. P16$_2$C is a custom format with 2-bit exponent but asymmetric slope.

shift the number range down by 2^{-15} via regular FP32 multiplication right after conversion.

4. Floating-point conversion: FP32 ↔ FP16C

For FP32 ↔ FP16C conversion, we developed a set of fast conversion algorithms that work in any programming language and on any hardware which we describe in some more detail further below. An OpenCL C version is presented in Listing 1.

We ditch `NaN` and `Inf` definitions for an extended number range by a factor of 2 and less complicated and faster conversion. In PTX assembly [112], the FP32 → FP16C conversion takes 25 instructions and FP16C → FP32 takes 26 instructions.

FP32 → FP16C: The first step is to interpret the bits of the FP32 input number as `uint`, for which there is the `as_uint(float x)` function provided by OpenCL. The sign bit remains identical as the leftmost bit via bit-masking and bit-shifting. To assure correct rounding, we add a `1` to the 12th bit from left (`0x00000800`), because mantissa bits at positions 12 to 0 later are truncated. Next, we extract the exponent e by bit-masking and bit-shift by 23 places to the left.

For normalized numbers, the exponent is decreased by the difference in bias $127 - 15 = 112$ and bit-shifted to the right by 11 places. A final bit-mask ensures that there is no overflow into the sign bit. The mantissa is bit-shifted in place and or-ed to sign and exponent.

For denormalized numbers, we first add a `1` to place 24 (`0x00800000`) of the mantissa (to later figure out how many places the mantissa was shifted) and then bit-shift it to the right by as many places as the new exponent is below zero. Correct rounding, however, makes this a bit more difficult: We need to add `1` for rounding to the leftmost place of the mantissa that is cut off. To undo the initial rounding we did earlier, instead of `0x00800000`, we add `0x00800000`−`0x00000800`=`0x007FF800`, then shift by one place less than the new exponent is below zero, add `1` to the rightmost bit and finally shift right the one remaining place.

The exponent itself is the switch deciding whether the normalized or denormalized conversion is used. As an optional safety measure, we add saturation: If the number is larger than the maximum value, we override all exponent and mantissa bits to `1` (bitwise or with `0x7FFF`).

FP16C → FP32: To convert back to FP32, we first extract the exponent e and the mantissa m by bit-masking and bit-shifting. Additionally, since we intend to avoid branching, we already count the number of leading zeros[4] v in the mantissa for decoding the denormalized format: We cast m to `float`,[5] reinterpret the result as `uint`, bit-shift the exponent right by 23 bits and subtract the exponent bias, giving us the base-2 logarithm of m, equivalent to 31 minus the number of leading zeros.

The sign bit is bit-masked and bit-shifted in place. The exponent e again decides for normalized or denormalized numbers: For normalized numbers ($e \neq 0$), the exponent is increased by the difference in bias $127 - 15 = 112$ and or-ed together with the bit-shifted mantissa. For denormalized numbers ($e = 0$ and $m \neq 0$), the mantissa is bit-shifted to the right by the number of leading zeros and the shift-indicator `1` is removed by bit-masking. The mantissa is or-ed with the exponent which is set by the number of leading zeros and bit-shifted in place.

Finally, the `uint` result is reinterpret as `float` via the OpenCL function `as_float(uint x)`.

B. Posit

1. Overview

The novel posit format (type-III Unum) [90,111,113,114] is different from floating-point in that the bit segment for the mantissa (and also exponent) is variable in size and there is another bit segment, the regime, with variable size as well. The posit number representation is

$$x = \underbrace{(-1)^s}_{\text{sign}} \cdot \underbrace{(2^{n_e+1})^r}_{\text{regime}} \cdot \underbrace{2^e}_{\text{exponent}} \cdot \underbrace{(1 + 2^{-n_m}\, m)}_{\text{mantissa}}, \qquad (8)$$

with sign s, regime r, exponent e, and mantissa m. $n = 1 + (n_r + 1) + n_e + n_m$ is the total number of bits, whereby n_r, n_e, and n_m are the variable numbers of bits in regime, exponent and mantissa, respectively.

For very small numbers, the regime bit pattern looks like `000..01` (negative r), then gets shorter toward `01` ($r = -1$), flips to `10` ($r = 0$) and then gets longer again, looking like `111..10` (positive r). The last bit is the regime terminator bit that unambiguously tells the length of the regime. This bit is not included in the regime size n_r, so the size of the regime bit pattern is $n_r + 1$. n_r determines the value of the regime: $r = -n_r$ if the regime terminator bit is `1` or $r = n_r - 1$ if the regime terminator bit is `0`.

For increasing regime size, the remaining bits for exponent and mantissa are shifted to the right, so the mantissa (and if no mantissa bits are left also the exponent) become shorter and precision is reduced.

Posits can be designed with different (fixed) exponent sizes or no exponent at all. Just like for floating-point, larger exponent increases the range but decreases accuracy. This way, the posit format is designed to deliver variable accuracy based on where the number is in the regime: best accuracy is around ±1.0 where the regime is shortest (superior to floating-point) but for both tiny and large numbers, much precision is lost [114].

2. Customized posit formats for the LBM

As a storage format for LBM DDFs, where numbers close to 0 need to be resolved best and numbers outside the ±2 range are not required at all, the standard 16-bit posit formats seems an unfavorable choice. However, by multiplying a constant before and after conversion, similar to FP16S, we shift the most accurate part down to smaller numbers. We take a closer look at three different posit formats:

(1) P16$_0$S: 16-bit posit without exponent, shifted down by $\times 2^7$. In the interval $[2^{-11}, 2^{-3}]$, accuracy is equal to or better

[4]The OpenCL function `clz(m)` also counts the number of leading zeros. While translated into a single `clz.b32` PTX instruction (instead of `cvt.rn.f32.u32 mov.b32 shr.u32`), `clz.b32` executes much slower, leading to noticeably worse performance.

[5]Casting an `int` to `float` implicitly does a `log2` operation to determine the exponent.

than FP16S and in the interval $[2^{-10}, 2^{-4}]$, accuracy is equal to or better than FP16C. The range toward small numbers is very poor and for numbers $>2^{-3}$, accuracy is vastly degraded.

(2) P16$_1$S: 16-bit posit with one-bit exponent, shifted down by $\times 2^7$. In the interval $[2^{-13}, 2^{-1}]$, accuracy is equal to or better than FP16S and in the interval $[2^{-11}, 2^{-3}]$, accuracy is equal to or better than FP16C. For numbers $<2^{-13}$ or $>2^{-1}$, accuracy is reduced. The range toward small numbers is between FP16S and FP16C. This format poses no limitations on the density ρ because its number range is $\pm 2^{21}$.

(3) P16$_2$C: Custom asymmetric 16-bit posit with two-bit exponent, not shifted. By only covering the lower flank, we can get rid of the bit reserved for the regime sign, thus making the regime shorter by one bit and increasing the mantissa size by one bit in turn. The conversion algorithms are vastly simplified with the asymmetric regime. Accuracy is better or equal to FP16C in the interval $[2^{-7}, 2]$ and equal or better than FP16S in the interval $[2^{-11}, 2]$. For smaller numbers, accuracy is slowly reduced, but the range toward small numbers is excellent.

Both P16$_0$S and P16$_1$S provide numbers >2 that are unused in the LBM. Shifting the number range further down would degrade accuracy for larger numbers too much. Since the LBM with DDF-shifting uses numbers around 0 and it is not entirely clear in which order of magnitude accuracy is most important, it is also unclear if the increased accuracy in the center interval will benefit more than the decreased accuracy further away from the center will adversely affect.

3. FP32 ↔ *posit conversion*

Conversion between FP32 and posit is not supported in hardware (yet). Since the reference conversion algorithm in the SoftPosit library [115] is not written for speed primarily, we provide self-written, ultrafast conversion algorithms in Listing 1 in OpenCL C. These work on any hardware. A detailed description of how the algorithms work is omitted here but can be inferred by studying the provided listings. Note that the posit specification [111] does two's complement for negative numbers to have no duplicate zero and an infinity definition instead. To simplify the conversion algorithms and since infinity is not required in our applications, we just use the sign bit to reduce operations, so there is positive and negative zero.

C. Fixed-point

16-bit fixed-point format with a range scaling of ± 2 has discrete additive steps of $2^{-14} \approx 6.110^{-5}$, so this is also the smallest possible value. Compared to floating-point, precision is worse for small numbers and better for large numbers. For the LBM, this is insufficient and does not work.

D. Required code interventions

At all places where the DDFs are used as kernel parameters, their data type is made switchable with a macro (`fpXX`). At any location where the DDFs are loaded or stored in memory, the load (store) operation is replaced with another macro as provided in Listing 1 for FP32, FP16S, FP16C, P16$_0$S, P16$_1$S, and P16$_2$C. In the Appendix in Listing 2, we provide the core of our LBM implementation, exemplary for D3Q19 SRT.

IV. ACCURACY COMPARISON

A. 3D Poiseuille flow

A standard setup for LBM validation is a Poiseuille flow through a cylindrical channel [95]. For the channel walls, we use standard nonmoving midgrid bounce-back boundaries [1,12] and we drive the flow with a body force as proposed by Guo *et al.* [116]. Simulations are done with the D3Q19 velocity set and a single-relaxation-time (SRT) collision operator. We compare the simulated flow profile $u_{\text{sim}}(r)$ with the analytic solution [117]

$$u_{\text{theo}}(r) = \frac{f}{4\,\rho\,\nu}\,(R^2 - r^2) \tag{9}$$

to compute the error. Here, $\rho = 1$ is the average fluid density,

$$r = \sqrt{\left(y - \frac{L_y}{2}\right)^2 + \left(z - \frac{L_z}{2}\right)^2} \tag{10}$$

is the radial distance from the channel center, R is the channel radius,

$$\nu = \frac{2\,R\,u_{\max}}{Re} = \frac{\tau}{3} - \frac{1}{6} \tag{11}$$

is the kinematic shear viscosity, and τ is the relaxation time. The dimensions of the simulation box are

$$L_x = 1, \quad L_y = L_z := 2\,(R+1). \tag{12}$$

The flow is driven by a force per volume f that is calculated by rearranging Eq. (9) with $r = 0$:

$$f = \frac{4\,\rho\,\nu\,u_{\max}}{R^2}. \tag{13}$$

In accordance with Ref. [12], we define the error as the L_2 norm [1, p. 138]:

$$E = \sqrt{\frac{\sum_{r=0}^{R} |u_{\text{sim}}(r) - u_{\text{theo}}(r)|^2}{\sum_{r=0}^{R} |u_{\text{theo}}(r)|^2}}. \tag{14}$$

In Fig. 4, we keep the Reynolds number and center velocity constant at Re = 10 and $u_{\max} = 0.1$, and vary channel radius

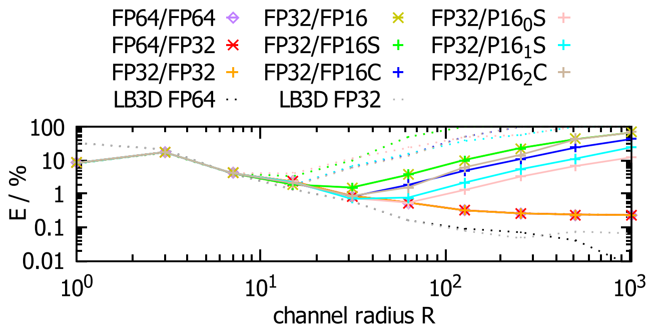
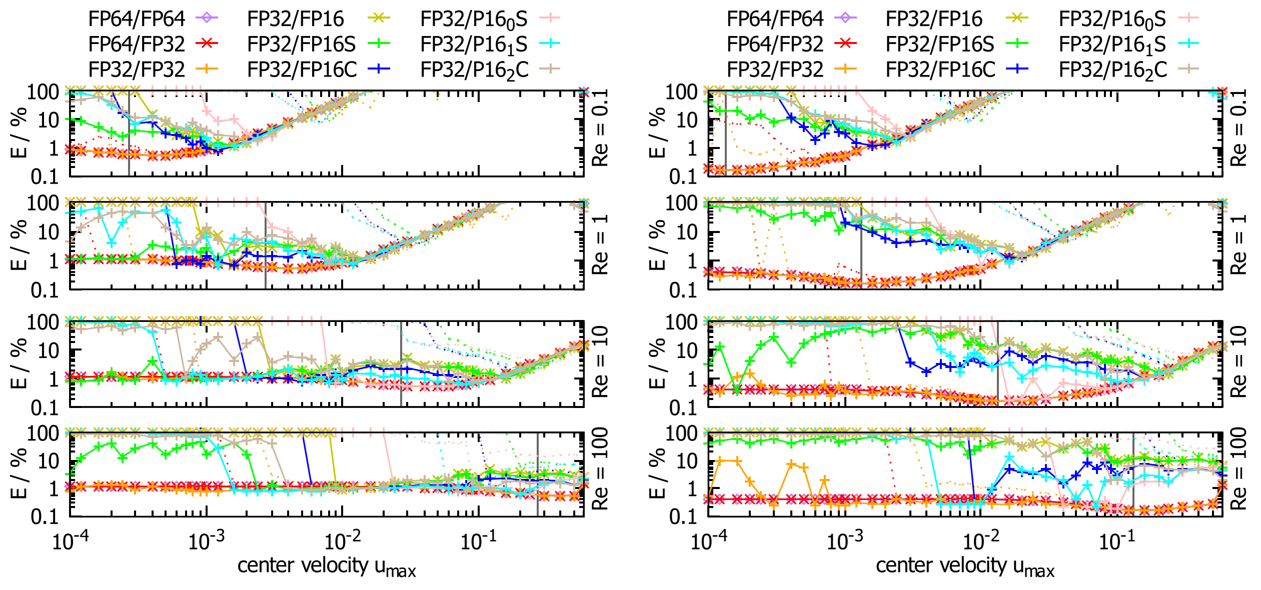


FIG. 4. Error of D3Q19 SRT Poiseuille flow for varying channel radius R (lattice resolution) at constant Reynolds number Re = 10 and constant center flow velocity $u_{\max} = 0.1$. The dashed lines represent corresponding simulations without DDF-shifting.



(a) $R = 31$ (b) $R = 63$

FIG. 5. Error of D3Q19 SRT Poiseuille flow for varying center velocity u_{max} at constant Reynolds number Re $\in \{0.1, 1, 10, 100\}$ and constant channel radius (a) $R = 31$ and (b) $R = 63$. The dashed curves represent corresponding simulations without DDF-shifting. The vertical lines represent the LBM relaxation time $\tau = 1$.

R and kinematic shear viscosity ν accordingly. For $R \leqslant 15$, we see almost no difference between any of the floating-point variants. Here the staircase effect of the channel walls dominates the error. Moving toward larger radii, the error increases at first for FP32/FP16 and FP32/FP16S and later for FP32/FP16C as well, while FP64/xx and FP32/FP32 show no difference in this regime either. 16-bit posit formats hold up even better here with their increased peak accuracy. P16$_2$C for small R behaves like FP16C and then migrates over to FP16S as R becomes larger and the DDFs become smaller. We also simulate the same system without DDF-shifting (dashed lines) to quantify the difference. Already here we see that the 16-bit formats become unfeasible without DDF-shifting.

To confirm that the observed agreement between FP32/FP32 and FP64/FP64 is not a coincidence of our implementation, in Fig. 4 we include data from a simulation of the very same system with the LB3D code [79] that is further described in the Appendix.

We now investigate the error in more detail for a constant channel radius $R \in \{31, 63\}$ in Fig. 5. We simulate the flow in the channel for different Reynolds numbers Re $\in \{0.1, 1, 10, 100\}$ and vary the center velocity u_{max} and kinematic shear viscosity ν accordingly.

We find that the higher the Reynolds number, the further the minimal error is shifted toward larger u_{max}, always staying close to where $\tau = 1$ (vertical lines). The better small numbers can be resolved, the lower u_{max} can be chosen before the error suddenly becomes large. The better the accuracy of the mantissa, the lower the overall error, up to a certain point where discretization errors dominate at large u_{max}.

It is important to consider that compute time is proportional to u_{max} and that $u_{\text{max}} < u_{\text{max},\tau=1} = \frac{\text{Re}}{12\,R}$ smaller than at the error minimum is thus less practically relevant. In the domain $u_{\text{max}} \geqslant u_{\text{max},\tau=1}$ (in Fig. 5, right of the vertical lines), FP16C is almost always superior to FP16S, especially at higher Re. Posits show their superior precision most of the time, if the DDFs are just in the right interval.

We find that without DDF-shifting, the 16-bit formats become very inaccurate. For FP32/FP32, there is some benefit at higher Re and especially low velocities u_{max}. For FP64, the DDF-shifting does not make any noticeable difference in this setup as discretization errors dominate.

To better understand where the error comes from in the Poiseuille channel radially, we exemplary plot the error contribution as a function of the radial coordinate r for the parameters $R = 63$, $u_{\text{max}} = 0.1$, Re = 10 in Fig. 6. We find that for FP64 to FP32, the largest error contribution is near the channel wall (staircase effect along the no-slip bounce-back boundaries). For FP32/16-bit, there is equal error contribution near the wall, but the majority of the error comes from close to the channel center. The wall poses a boundary condition not only for velocity $[u(R) = 0]$ but also for the

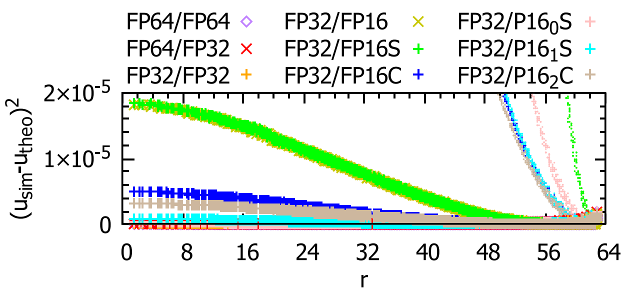


FIG. 6. Radial error profile of D3Q19 SRT Poiseuille flow for a channel radius $R = 63$ and center flow velocity $u_{\text{max}} = 0.1$ at constant Reynolds number Re = 10. The small dots on the right represent corresponding simulations without DDF-shifting. Note that the error contribution is on a linear scale here.

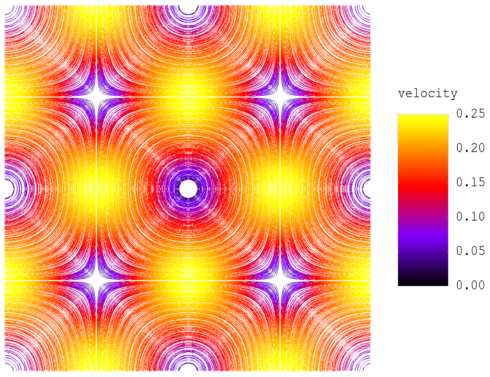


FIG. 7. Illustration of the velocity field at $t = 0$ with colored streamlines.

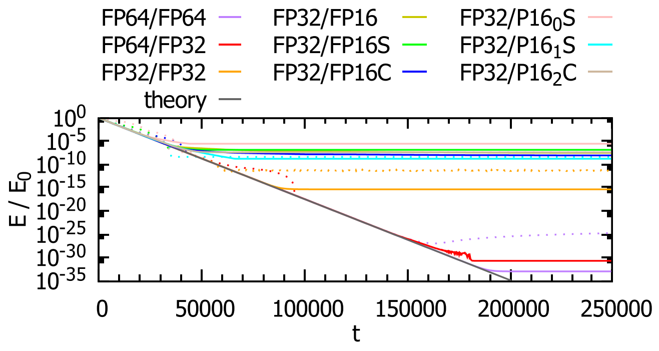


FIG. 8. Relative kinetic energy $E(t)/E_0$ of a D2Q9 SRT simulation of Taylor-Green vortices compared to the analytic solution in Eq. (18). Dashed lines represent corresponding simulations without DDF-shifting.

velocity error. Going radially inward from the channel wall, at first the staircase effect smooths out, lowering the error, but then each concentric ring of lattice points accumulates systematic floating-point errors, so at the channel center the error is largest. For FP32/FP32, this error behavior is barely noticeable but visible upon close inspection. For FP64, the floating-point errors are so tiny that the staircase smoothing continues all the way through the radial profile, making the error smallest in the center. Without DDF-shifting, there is no noticeable difference for FP64 and FP32 compared to when DDF-shifting is done, but the 16-bit formats become unfeasible.

B. Taylor-Green vortices

An especially well suited setup for testing the behavior at low velocities is Taylor-Green vortices. A periodic grid of vortices is initialized with velocity magnitude u_0 (illustrated in Fig. 7) and then over time viscous friction slows down the vortices while they remain in place on the grid. In 2D, the analytic solution [66,96] reads

$$u_x(t) = +u_0 \cos(k\,x)\,\sin(k\,y)\,e^{-2\,\nu\,k^2\,t}, \tag{15}$$

$$u_y(t) = -u_0 \sin(k\,x)\,\cos(k\,y)\,e^{-2\,\nu\,k^2\,t}, \tag{16}$$

$$\rho(t) = 1 - \frac{3\,u_0^2}{4}\,(\cos(2\,k\,x) + \cos(2\,k\,y))\,e^{-4\,\nu\,k^2\,t}, \tag{17}$$

and at $t = 0$ is used to initialize the simulation with $u_0 = 0.25$. Here $\nu = \frac{\tau}{3} - \frac{1}{6} = \frac{1}{6}$ is the kinematic shear viscosity at $\tau = 1$ and $k = \frac{2\pi N}{L}$. $L = 256$ is the side length of the square lattice and $N = 1$ is the number of periodic tiles in one direction. The kinetic energy

$$E(t) = \int_0^L \int_0^L \frac{\rho}{2}\,(u_x^2 + u_y^2)\,dx\,dy$$
$$= u_0^2\,\pi^2\,e^{-4\,\nu\,k^2\,t} \tag{18}$$

drops exponentially with time t. $E_0 = E(t = 0)$ denotes the initial kinetic energy. We compute the kinetic energy from the simulation as the discrete sum across all lattice points and compare it to the analytic solution in Fig. 8. The simulated kinetic energy drops exponentially as well, but at some point it does not drop further and remains constant as a result of floating-point errors. The relative energies of the plateaus are no coincidence: The plateaus are located at approximately the truncation error ϵ squared (Table II) for the respective number format in use. Particularly interesting is that for FP64/FP32 the plateau is much lower than for FP32 ϵ^2, being closer to FP64 ϵ^2. With P16$_0$S, the DDFs are outside of the most accurate interval, so accuracy is poor overall.

Finally, we note that the plateaus only reach down to ϵ^2 if DDF-shifting is properly implemented as presented in Eq. (3). Without DDF-shifting, there is significant loss in accuracy across all number formats.

C. Karman vortex street

Our next setup is a Karman vortex street in two dimensions [97]: a cylinder with radius $R = 32$ is placed into a simulation box with dimensions 512×1024. At the box perimeter, a velocity of $\vec{u} = (0,\ 0.15)$ is enforced using nonreflecting equilibrium boundaries [12,118]. The Reynolds number is set to $\text{Re} = \frac{2\,R\,|u|}{\nu} = 250$, defining the kinematic shear viscosity $\nu = \frac{\tau}{3} - \frac{1}{6}$ and relaxation time τ.

If starting the simulation with perfectly symmetric initial conditions, only floating-point errors can eventually trigger the Karman vortex instability. We notice that in some cases, the instability would not start at all even after several hundred thousand time steps. To avoid this nonphysical behavior, we initialize the velocity $\vec{u} = (0,\ 0.15)$ not only at the simulation box perimeter but also on the left half $x < 256$. This immediately triggers the Karman vortex instability regardless of floating-point setting.

We probe the velocity at the simulation box center (256, 512) over time in Fig. 9. This demonstrates that, when DDF-shifting is done, the 16-bit formats are almost indistinguishable from FP64 ground truth both qualitatively and quantitatively, with only minimal phase-shift for FP16, FP16S, and P16$_0$S.

To assess in detail where eventual differences may be present beyond a single velocity point probe, we look at the vorticity throughout the simulation box. In Fig. 10, we show the vorticity in the very much zoomed-in range of ± 0.001. For the 16-bit formats, in low vorticity areas there is noise present. Comparing FP16 and FP16S, the extended number range

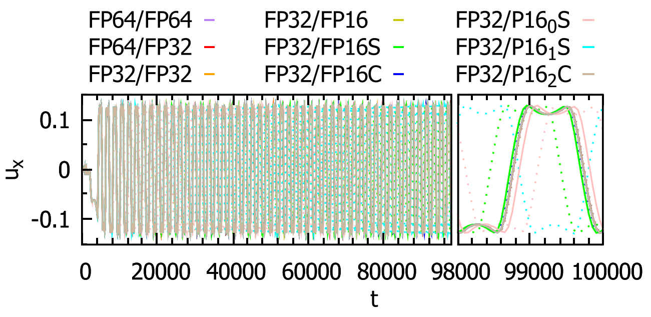


FIG. 9. Velocity x component of the Karman vortex street at the simulation box center (256, 512) over time for various floating-point precision. Dashed lines represent corresponding simulations without DDF-shifting. Even after 100 000 LBM time steps (50 vortex periods), the 16-bit graphs still cover the FP64 ground truth as amplitude, frequency, and even phase appear indistinguishable. Only zooming in at the last oscillation period reveals minuscule differences in phase for FP16, FP16S, and P16$_0$S. The phase shift in the 16-bit graphs is large without the DDF-shifting optimization. FP32/FP16C, FP32/FP32, and FP64/FP32 are indistinguishable from FP64 ground truth even when zooming in.

toward small numbers has no benefit here. FP16C with DDF-shifting mostly mitigates this noise, showing that the noise purely originates in smaller mantissa accuracy and numeric loss off significance. Our custom posit P16$_2$C has similarly low noise. P16$_0$S shows artifacts.

D. Lid-driven cavity

The lid-driven cavity is a common test setup for the LBM [30,33,37,39,49,52,98–100] and other Navier-Stokes solvers [101–103]. We here implement it in a cubic box at Reynolds number Re = 1000. On the lid, velocity parallel to the y-axis is enforced through moving bounce-back boundaries [1,12]. The box edge length is $L = 128$, the velocity at the top lid is $u_0 = 0.1$ in lattice units, and the kinematic shear viscosity is set by the Reynolds number $\mathrm{Re} = \frac{L u_0}{\nu} = 1000$. We simulate 100 000 LBM time steps with the D3Q19 SRT scheme.

Figure 11 shows the y (z) velocity along horizontal (vertical) probe lines through the simulation box center. All number formats except P16$_0$S look indistinguishable, even without DDF-shifting. Only when zooming in (not shown), for the simulations without DDF-shifting, deviations in relative velocity in the second digit become visible. With DDF-shifting, deviations are present only in the fourth digit, being smallest for FP16C, P16$_1$S, and P16$_2$C.

E. Capsule in shear flow

Here we test the number formats on a microcapsule in shear flow, one of the standard tests for microfluidics simulations in medical applications [105,106]. The D3Q19 multi-relaxation-time (MRT) [1,12] LBM is extended with the IBM [119] to simulate the deformable microcapsule in flow. For the IBM, we use the same level of precision as for the LBM arithmetic,

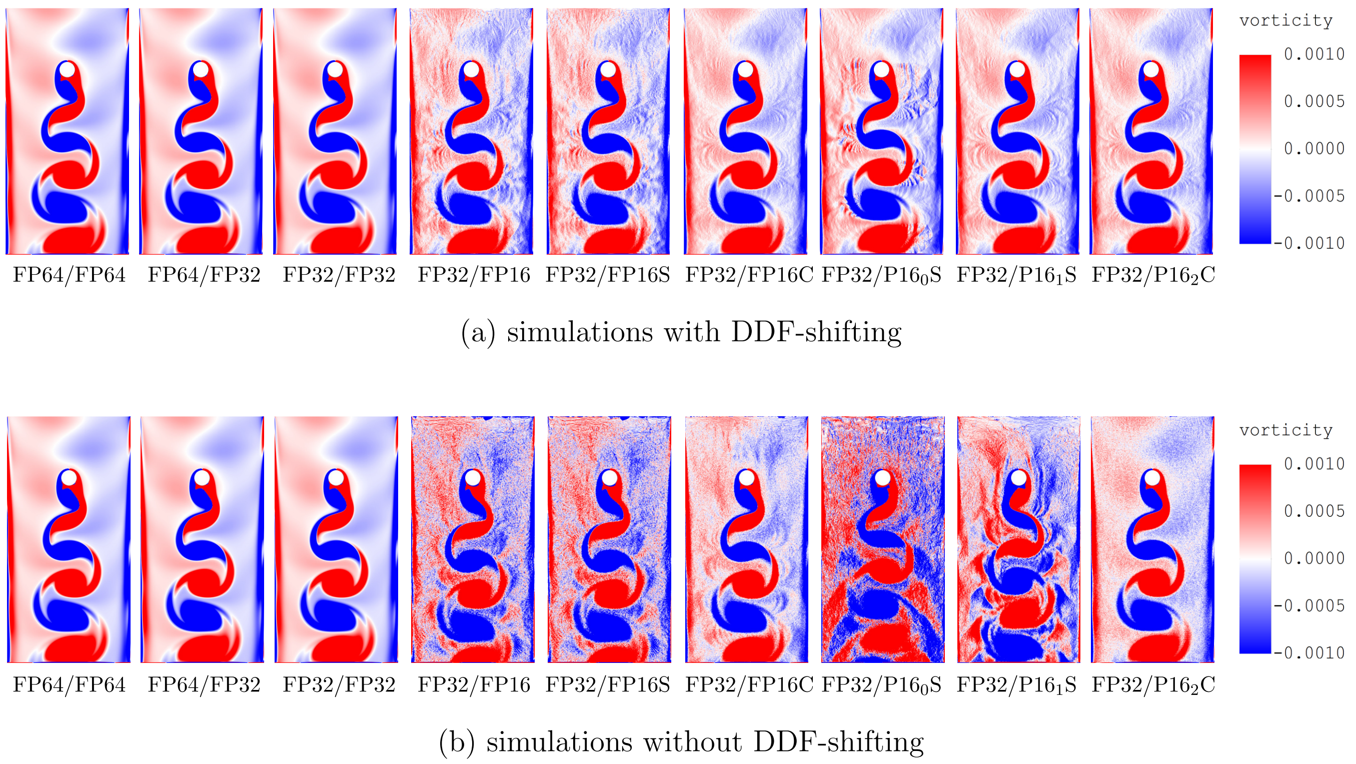


FIG. 10. Vorticity in the vastly overexposed range ±0.001 for simulations (a) with and (b) without DDF-shifting, after 100 000 LBM time steps. All simulations very accurately predict the vortex street, with frequency and amplitude of the vortices being identical and only insignificant differences in phase-shift even after 50 vortex periods. FP32 is indistinguishable from FP64 ground truth. For 16-bit, in the low vorticity range there is noise present, equally for FP16 and FP16S, but vastly reduced for FP16C and P16$_2$C. Omitting DDF-shifting vastly increases this noise and also adds significant phase shift as can be seen by comparing the position of the last red vortex at the bottom.

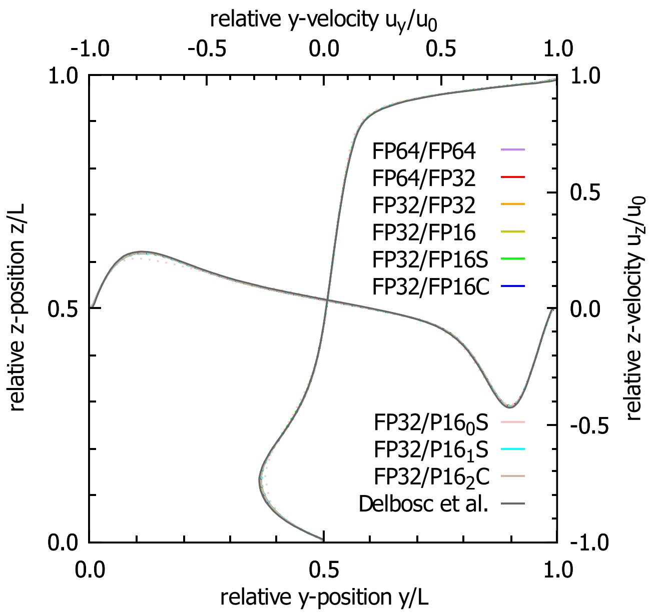


FIG. 11. The y-velocity along a vertical probe line through the simulation box center as well as the z-velocity along a horizontal line through the simulation box center. At the top lid, the velocity is fixed. As the flow goes one rotation clockwise, the width of the high-velocity peak increases and the height decreases. Dashed lines represent corresponding simulations without DDF-shifting. As a reference, we show the data from Delbosc *et al.* [33].

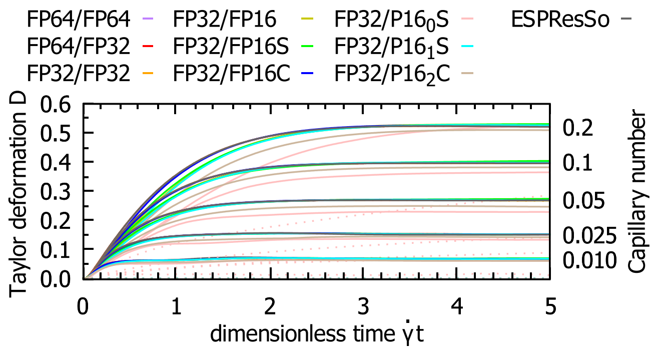


FIG. 13. Taylor deformation of the capsule first increases and then plateaus as the capsule starts tank-treading. This plateau depends on the Capillary number. Dashed lines represent corresponding simulations without DDF-shifting.

so either FP64 or FP32. As illustrated in Fig. 12, we place an initially spherical capsule of radius $R = 13.5$ in the center of a simulation box with the dimensions $128\times64\times192$, and we compute 385 000 time steps. At the top and bottom of the simulation box, a shear flow is enforced via moving bounce-back boundaries [120]. The membrane of the capsule is discretized into 5120 triangles and membrane forces, consisting of shear forces (neo-Hookean) [104,105,121] as well as volume forces (volume has to be conserved), are computed as in Ref. [105].

The Reynolds number is Re = 0.05, the kinematic shear viscosity is $\nu = \frac{1}{3}$, and we simulate various capillary numbers $\text{Ca} = \frac{\dot{\gamma}\,\mu\,R}{k_1} \in \{0.010, 0.025, 0.05, 0.1, 0.2\}$ by varying the membrane shear modulus k_1. The shear rate is $\dot{\gamma} = 1.310^{-5}$ in simulation units. To cross validate our results, we perform the same simulations with ESPResSo (FP32 for LBM, FP64 for IBM) [24], which has been cross validated with boundary-integral simulations and many others in Ref. [105]. In Fig. 13, we plot the Taylor deformation $D = \frac{a-c}{a+c}$ over time, with the largest and smallest semiaxes a and c of the deformed capsule [105]. We see that even in this complex scenario, the FP16C simulations produce physically accurate results with only insignificant deviations from FP64. The other 16-bit formats, especially posits, perform noticeably worse here. Without DDF-shifting, while FP32 still appears identical to ground truth, all 16-bit simulations do not produce the correct outcome (deformation remains close to zero). This emphasizes that DDF-shifting is essential for the lower precision formats.

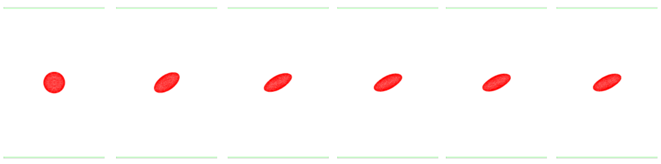


FIG. 12. Illustration of the capsule in shear flow (FP32/FP32, Ca = 0.1) simulation at dimensionless times $\dot{\gamma}\,t \in \{1, 2, 3, 4, 5\}$. Each image shows the simulation box from the side, with the top and bottom moving bounce-back boundaries marked in green. The capsule initially deforms to an elongated shape and then performs tank-treading, i.e., rotating the membrane while keeping its deformed shape.

F. Raindrop impact

Finally, we examine how number formats affect a volume-of-fluid LBM simulation of a 4 mm diameter raindrop impacting a deep pool at 8.8 $\frac{\text{m}}{\text{s}}$ terminal velocity. This system is described and extensively validated in Ref. [10] to study microplastic particle transport from the ocean into the atmosphere. The particles are simulated with the IBM. There, simulations are performed in FP32/FP32 with the maximum lattice size that fits into memory, so FP64 is not used here as it does not fit into the memory of a single GPU. The dimensionless numbers for this setup are Reynolds number $\text{Re} = \frac{d\,u}{\nu} = 33498$, Weber number $\text{We} = \frac{d\,u^2\,\rho}{\sigma} = 4301$, Froude number $\text{Fr} = \frac{u}{\sqrt{d\,g}} = 44.4$, Capillary number $\text{Ca} = \frac{u\,\rho\,\nu}{\sigma} = 0.1284$ and Bond number $\text{Bo} = \frac{d^2\,\rho\,g}{\sigma} = 2.179$. The simulated domain is $464\times464\times394$ lattice points and runs on a single AMD Radeon VII GPU. The impact is simulated for 10 ms time, equivalent to 20 416 time steps in LBM units.

The raindrop impact is illustrated in Fig. 14. Note that the fully parallelized GPU implementation of the IBM with floating-point `atomic_add_f` makes the simulation nondeterministic [10,12] and that the exact breakup of the crown into droplets is expected to be randomly different every time. We see minor artifacts at the bottom of the cavity for FP32/P16$_1$S, but otherwise no qualitative differences in random crown breakup.

To be able to obtain statistics of ejected droplets and particles, we run the simulation 100 times each with FP32/FP32, FP32/FP16S, FP32/FP16C, and FP32/P16$_1$S. The microplastic particles each time are initialized at different

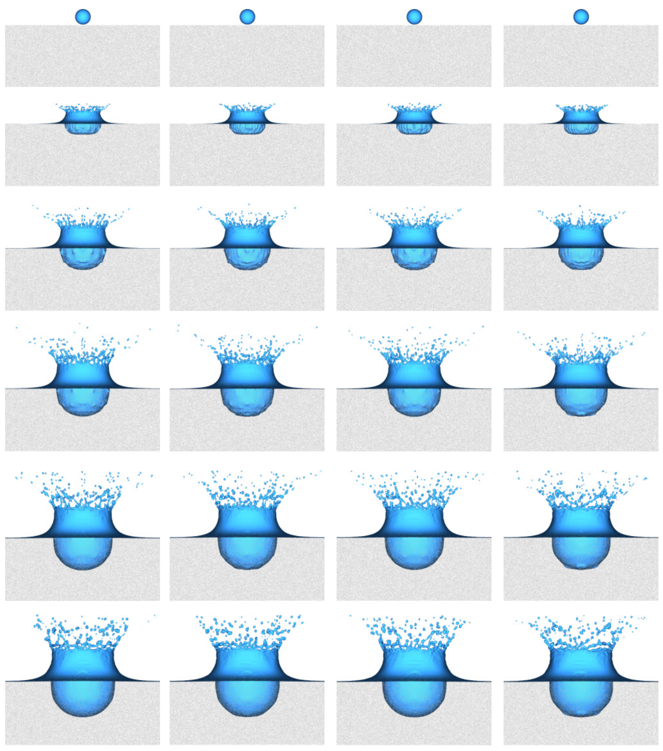


FIG. 14. A 4 mm diameter raindrop impacting a deep pool at 8.8 $\frac{\text{m}}{\text{s}}$ terminal velocity, illustrated at times $t \in \{0, 1, 2, 3, 4, 5\}$ ms after impact as used in Ref. [10].

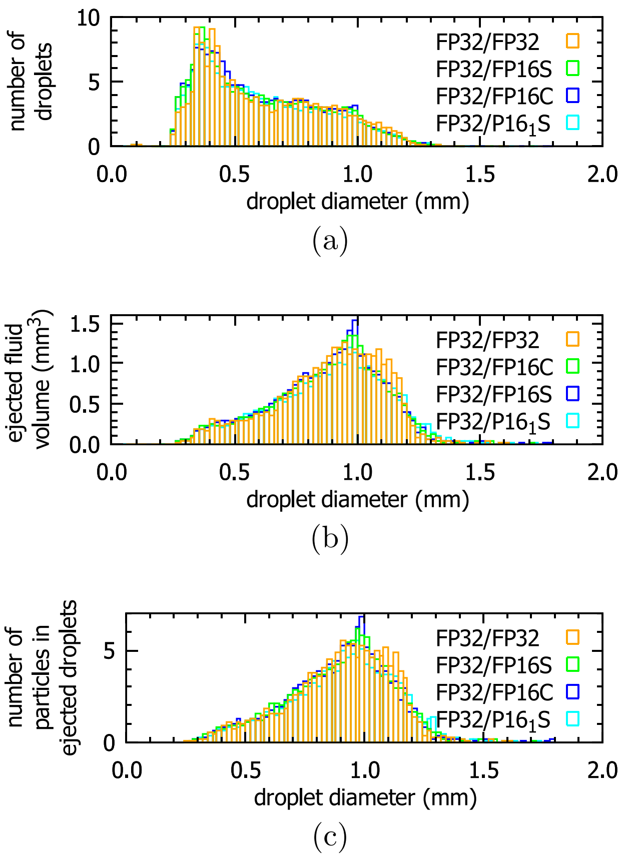


FIG. 15. (a) The size distribution of droplets, (b) the distribution of ejected fluid volume by droplet diameter, and (c) the distribution of microplastic particles in droplets for 100 simulations each conducted with FP32/FP32, FP32/FP16S, FP32/FP16C and FP32/P16$_1$S.

random positions, resulting in slightly different random crown breakup. Ejected droplets that touch the top of the simulation box are measured and then deleted as detailed in Ref. [10]. In histograms of the size, volume, and particle count depending on droplet diameter (Fig. 15), we see no significant differences across the data sets.

To conclude this section, we find that all FP32/FP16S, FP32/FP16C, and FP32/P16$_1$S are able to recreate the results of FP32/FP32 in raindrop impact simulations without negative impact on the accuracy of the results, while significantly reducing the memory footprint of these simulations. This in turn enables simulations higher lattice resolution, potentially increasing accuracy by resolving smaller droplets.

V. MEMORY AND PERFORMANCE COMPARISON

For GPUs, the most efficient streaming step implementation [63] is the One-Step-Pull scheme (AB-Pattern) with two copies of the DDFs in memory, because the noncoalesced memory read penalty is lower than the noncoalesced write penalty on GPUs [12,15,30,33,35,37,38,51,53,54], see Fig. 22 in the Appendix. One-Step-Pull further greatly facilitates implementing LBM extensions like Volume-of-Fluid, so it is a popular choice. Our FluidX3D base implementation (no-slip bounce-back boundaries, no extensions, as in Listing 2) with DdQq velocity set has memory requirements per lattice point as shown in Table III. For D3Q19, going from FP32/FP32 to FP32-16x reduces the memory footprint by ≈45%, to 93 bytes per node. If 16-bit compression was combined with in-place streaming schemes like AA-Pattern [34], Esoteric-Twist [62], Shift-and-Swap-Streaming [59], or the simple Esoteric-Pull [9], the memory footprint can even be reduced by ≈67%, to only 55 bytes per node.

Although our main goal with FP16 is to reduce memory footprint and allow for larger simulation domains, as a side effect, performance is vastly increased as a result of less memory transfer in every LBM time step. For our base implementation with the DdQq velocity set, the amount of memory transfers per lattice point per time step is shown in Table IV. Writing velocity and density to memory in each time step is not required for LBM without extensions. Theoretical speedup from FP32/FP32 to FP32/16-bit is 80% for all velocity sets and swap algorithms.

While most LBM implementations are limited to one particular hardware platform—either CPUs [67–79], Nvidia

TABLE III. Memory requirements in byte per lattice point of LBM floating-point variants for the One-Step-Pull swap algorithm with two copies of the DDFs for the DdQq velocity set.

	$\vec{u}$	ρ	flags	f_i	$\sum$
FP64/FP64	$8\,d$	8	1	$16\,q$	$8\,d+9+16\,q$
FP64/FP32	$8\,d$	8	1	$8\,q$	$8\,d+9+\ 8\,q$
FP32/FP32	$4\,d$	4	1	$8\,q$	$4\,d+5+\ 8\,q$
FP32/16-bit	$4\,d$	4	1	$4\,q$	$4\,d+5+\ 4\,q$

TABLE IV. Memory transfer in byte per lattice point per time step of LBM floating-point variants for the DdQq velocity set.

	flags	f_i	$\sum$
FP64/FP64	q	$16\,q$	$17\,q$
FP64/FP32	q	$8\,q$	$9\,q$
FP32/FP32	q	$8\,q$	$9\,q$
FP32/16-bit	q	$4\,q$	$5\,q$

GPUs [30–56], CPUs and Nvidia GPUs [18–29] or mobile SoCs [122,123]—only few use OpenCL [5–17]. With FluidX3D also being implemented in OpenCL, we are able to benchmark our code across a large variety of hardware, from the world's fastest data-center GPUs over gaming GPUs and CPUs to even the GPUs of mobile phone ARM SoCs. This enables us to determine LBM performance characteristics on various hardware microarchitectures. In Fig. 16, we show performance and efficiency on various hardware for D3Q19 SRT without extensions (only no-slip bounce-back boundaries are enabled in the code). The benchmark setup consists of a cubic box without any boundary nodes and with periodic boundary conditions in all directions. The standard domain size for the benchmark is 256^3, except where device memory is not enough; there we use the largest cubic box that fits into memory.

We group the tested devices into four classes with different performance characteristics:

(1) FP64-capable dedicated GPUs (high FP64:FP32 compute ratio) provide excellent efficiency for FP64/xx, FP32/FP32, and FP32/FP16S. They have such fast memory bandwidth that the FP32 ↔ FP16C software conversion brings FP32/FP16C from the bandwidth limit into the compute limit, reducing its efficiency.

(2) Non-FP64-capable dedicated GPUs (low FP64:FP32 compute ratio) have a particularly high FP32 arithmetic hardware limit, so even with the FP32 ↔ FP16C software conversion the algorithm remains in the memory bandwidth limit. FP32/xx efficiency is excellent except for older Nvidia Kepler. However, due to the poor FP64 arithmetic capabilities, FP64/xx efficiency is low as LBM here runs entirely in the compute limit rather than memory bandwidth limit. Surprisingly, FP64/FP32 runs even slower than FP64/FP64. This is because there is additional overhead for the FP64 ↔ FP32 cast conversion in the compute limit, despite less memory bandwidth being used.

(3) Integrated GPUs (iGPUs) overall show low performance and low efficiency. This is expected due to the slow system memory and cache hierarchy. Some older models do not support FP64 arithmetic at all.

(4) CPUs also show low performance and low efficiency. The low efficiency on CPUs is less of a property of the implementation or a result of OpenCL, and more related to CPU microarchitectures in general [67]. Other native CPU implementations of the LBM have equally low hardware efficiency [67,68,71,73] as a result of multilevel caching, inter-CPU communication, and other hardware properties unfavorable for LBM. To illustrate this further, our implementation runs about as fast on the Mali-G72 MP18 mobile phone GPU as CPU codes on between 2 and 16 cores, depending on the CPU model [19,23,27,67,68,71,73,86].

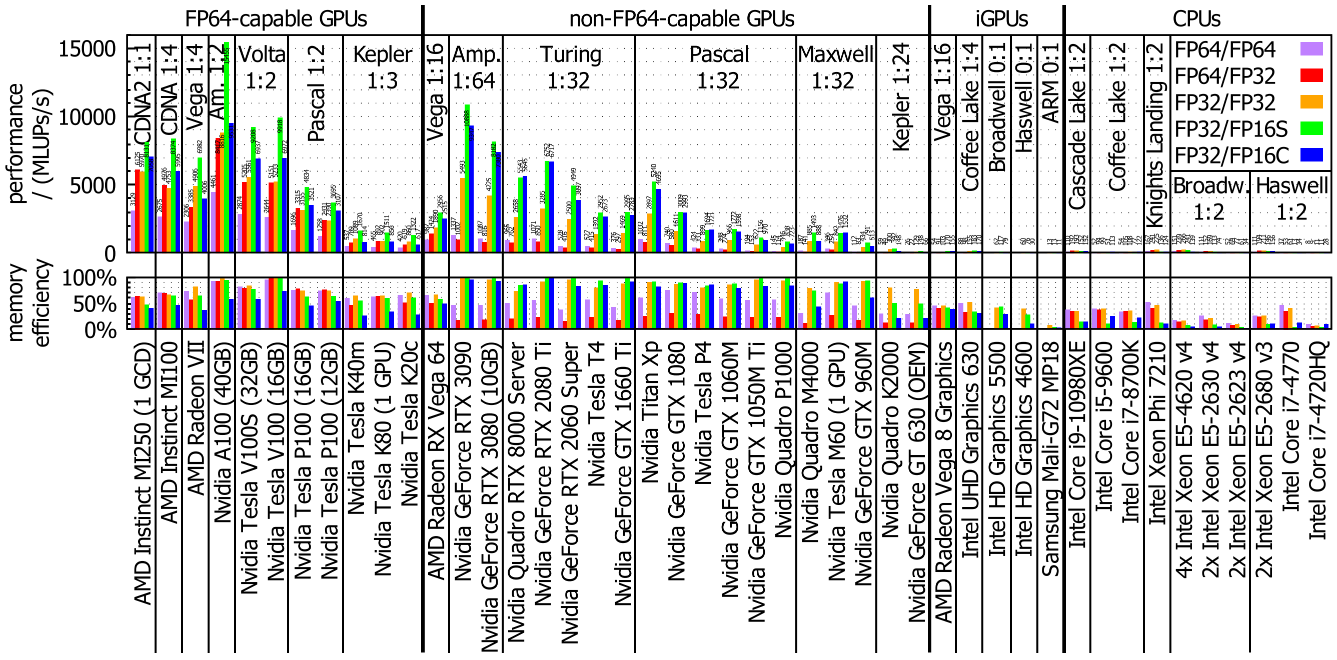


FIG. 16. Performance of FluidX3D with D3Q19 SRT on different hardware (code as in listing 2). The unit MLUPs/s is an acronym for mega lattice updates per second, meaning how many times 10^6 LBM lattice points are computed every second. To obtain the efficiency, we divide the measured MLUPs/s by the data sheet memory bandwidth times the number of bytes transferred per lattice point and time step (Table IV). Performance characteristics differ depending on the FP64 arithmetic capabilities as indicated by the FP64:FP32 compute ratios of the microarchitectures. The two GCDs of the MI250 are separate GPUs with 64 GB unified memory each, similar to dual-GPU cards such as the Tesla K80; driver 3423.0 (HSA1.1,LC) and ROCm 5.1.3 was used. CPU benchmarks are on all cores. Values in Table V.

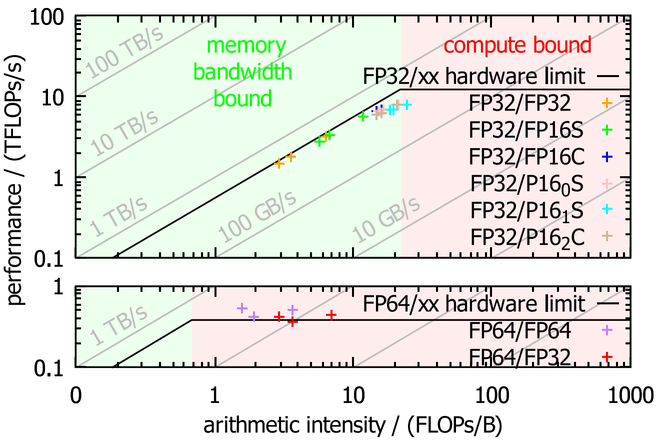


FIG. 17. Roofline model analysis of FluidX3D with the D3Q19 velocity set, running on an Nvidia Titan Xp GPU. For each floating-point type, the three data points (left to right) correspond to the SRT, TRT, and MRT collision operators. The arithmetic hardware limit is different for FP64/xx and FP32/xx, so we use two plots.

It is of note that performance on CPUs with large cache greatly depends on the domain size: If a large fraction of the domain fits into the L3 cache, efficiency (relative to memory bandwidth) is significantly better. Our CPU tests use a domain size of 256^3, so only an insignificant $\approx$1% is covered by L3 cache—a scenario representative of typical applications.

On the vast majority of hardware, we actually reach the theoretical 80% speedup as indicated by the hardware efficiency remaining equal for FP32/FP32 and FP32/FP16S. Some hardware, namely, the Nvidia Turing and Volta microarchitectures, do actually reach 100% efficiency with FP32/FP32 and FP32/FP16S. The Nvidia RTX 2080 Ti is at 100% efficiency even with FP32/FP16C, since the Nvidia Turing microarchitecture can do concurrent floating-point and integer computation and the 2080 Ti has high enough compute power per memory bandwidth to entirely remain in the memory bandwidth limit. Some efficiency values are even above 100% as Nvidia Turing and Ampere A100 are capable of memory compression to increase effective bandwidth beyond the memory specifications [124,125]. Nvidia Pascal GeForce and Titan GPUs (that lack ECC memory) lock into P2 power state with reduced memory clock for compute applications to prevent memory errors [126], lowering maximum bandwidth and making perfect (data sheet) efficiency impossible.

FP32/P16$_0$S and FP32/P16$_2$C performance is very similar to FP32/FP16C (data not shown), since the conversion needs to be emulated in software as well. FP32/P16$_1$S performance is a bit lower because the conversion algorithm is slightly more complex.

To better understand why performance is excellent with FP32/xx but not with FP64/xx on non-FP64-capable GPUs, we perform a roofline analysis [64,107] for the Nvidia Titan Xp in Fig. 17. The number of arithmetic operations and memory transfers is determined by automated counting of the corresponding PTX assembly instructions [112] of the stream-collide kernel. We note that we count the arithmetic intensity as the sum of floating-point and integer operations because the Pascal microarchitecture computes floating-point and integer on the same CUDA cores. For D3Q19 SRT FP32/FP32, for example, we count 255 floating-point operations and 248 integer operations. LBM performance scales proportionally to memory bandwidth, which is indicated by diagonal lines. The factor of proportionality is different for FPxx/64 (323 byte memory transfer per LBM time step), FPxx/32 (171 byte), and FPxx/16 (95 byte) as the amount of memory transfer is different (Table IV). FP16 reduces the number of memory transfers, so the arithmetic intensity (number of arithmetic operations divided by memory transfers) is increased. The manual conversion from and to FP16C significantly increases the number of arithmetic operations, further raising arithmetic intensity. Nevertheless, even with the arithmetic-heavy matrix multiplication of the MRT collision operator, all data points are still within the memory bandwidth limit and thus almost equally efficient compared to FP32. Actual memory clocks during the benchmark are 3.5% lower than the data sheet value (hardware limit) due to the Titan Xp locking into P2 power state [126], inhibiting perfect efficiency for FP32/xx. In contrast, FP64/xx is in the compute limit, greatly reducing performance. The data points in the compute limit can be a bit above the hardware limit if core clocks are boosted beyond official data sheet values.

VI. CONCLUSIONS

In this paper, we studied the consequences of the employed floating-point number format on accuracy and performance of lattice Boltzmann simulations. We used six different test systems ranging from simple, pure fluid cases (Poiseuille flow, Taylor-Green vortices, Karman vortex streets, lid-driven cavity) to more complex situations such as immersed-boundary simulations for a microcapsule in shear flow or a Volume-of-Fluid simulation of an impacting raindrop. For all of these, we thoroughly compared how FP64, FP32, FP16, and posit16 (mixed) precision affect the accuracy of the LBM. In the mixed variants, a higher precision floating-point format is used for arithmetics and a lower precision format is used for storing the DDFs. Based on the observation that a number range of ± 2 is sufficient for storing DDFs, we designed two customized 16-bit number formats specifically tailored to the needs of LBM simulations: a custom 16-bit floating-point format (FP16C) with halved truncation error compared to the standard IEEE-754 FP16 format by taking one bit from the exponent to increase the mantissa size and a specifically designed asymmetric posit variant (P16$_2$C). Conversion to these formats can be implemented highly efficiently and code interventions are only a few lines.

In all setups that we have tested and for the majority of parameters, FP32 turned out to be as accurate as FP64, provided that proper DDF-shifting [66] is used. Our custom FP16C format considerably diminished errors and noise and turned out to be a viable option for FP32/16-bit mixed precision in many cases. 16-bit posits with their variable precision have shown to be very compelling options too. Especially, P16$_1$S in some cases could beat our FP16C. In other cases, however, where the DDFs are outside the most favorable number range, the simulation error is increased significantly for the FP32/posit16 simulations.

Regarding performance, we find that pure FP64 runs very poorly on the vast majority of GPUs, with the exception of very few data-center GPUs with extended FP64 arithmetic capabilities such as MI250/MI100/A100/V100(S)/P100. FP64/FP32 mixed precision can be almost as fast as pure FP32 on these special data-center GPUs. However, somewhat counterintuitively, on all GPUs with poor FP64 capabilities, FP64/FP32 is even slower than pure FP64 due to the conversion overhead. In general, pure FP32 then is a better choice since it enables excellent computational efficiency across all GPUs, especially considering that it is equally accurate to FP64 in all but edge cases. Computational efficiency is also excellent for FP32/FP16S mixed precision across all GPUs, reaching a maximum performance of 15455 MLUPs (D3Q19) on a single 40 GB Nvidia A100. On almost all GPUs that we have tested, we see the theoretical speedup of 80% that FP32/16-bit mixed precision offers for D3Q19, alongside 45% reduced memory footprint. Our custom format FP32/FP16C requires manual floating-point conversion which is heavy on integer computation. Nevertheless, FP32/FP16C runs efficiently on most GPUs with good FP32 arithmetic capabilities compared to their respective memory bandwidth and the theoretically expected 80% speedup can be achieved.

In conclusion, we show that pure FP32 precision is sufficient for most application scenarios of the LBM and that with our specifically tailored FP16C number format, in many cases even mixed FP32/FP16C precision can be used without significant loss of accuracy.

ACKNOWLEDGMENTS

We acknowledge support through the computational resources provided by BZHPC, LRZ, CINECA, and JSC JURECA-DC Evaluation Platform. We acknowledge the NVIDIA Corporation for donating a Titan Xp GPU and an A100 40GB GPU for our research. We thank M. Lehmann, M. Meinhart, and R. Kellnberger for running the benchmarks on their PCs. This study was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) No. SFB 1357–391977956. We further acknowledge funding from Deutsche Forschungsgemeinschaft in the framework of FOR 2688, Instabilities, Bifurcations and Migration in Pulsating Flow, Projects No. B3 (417989940) and No. B2 (417989464).

M.L. and M.K. contributed the original concept. M.L. conducted the simulations and wrote the paper. M.L., M.K., G.A., and S.G. contributed essential ideas, paper review, and literature research. M.L., G.A., M.S., and J.H. contributed test setups and benchmarks.

APPENDIX

1. The LB3D code

While in the most of this paper, we used the FluidX3D code [6–12], we also confirmed selected results with the LB3D lattice Boltzmann simulation package [79]. For this, we ported the FP64/FP64 routines to FP32/FP32 also in LB3D. LB3D is an MPI-based, general-purpose simulation package that includes various multicomponent and multiphase lattice Boltzmann methods, coupled to point particle molecular dynamics, discrete element methods [127], and immersed boundary [128,129] methods, as well as finite element solvers for advection-diffusion problems, including the Nernst-Planck equation [130]. For the Poiseuille test, we used second-order accurate, midgrid bounce-back boundary conditions.

2. LBM equations in a nutshell

The coloring indicates the level of precision for the equations below:
lower precision storage, conversion, higher precision arithmetic.

a. Without DDF-shifting

(1) Streaming:

$$f_i^{\text{temp}}(\vec{x},t) = f_i^{A}(\vec{x}-\vec{e}_i,\ t). \tag{A1}$$

(2) Collision (SRT):

$$\rho(\vec{x},t) = \sum_i f_i^{\text{temp}}(\vec{x},t), \tag{A2}$$

$$\vec{u}(\vec{x},t) = \frac{1}{\rho(\vec{x},t)} \sum_i \vec{c}_i\, f_i^{\text{temp}}(\vec{x},t), \tag{A3}$$

$$f_i^{\text{eq}}(\vec{x},t) = f_i^{\text{eq}}(\rho(\vec{x},t),\ \vec{u}(\vec{x},t)) = w_i\,\rho\cdot\left(\frac{(\vec{u}\circ\vec{c}_i)^2}{2\,c^4} + \frac{\vec{u}\circ\vec{c}_i}{c^2} + 1 - \frac{\vec{u}\circ\vec{u}}{2\,c^2}\right), \tag{A4}$$

$$f_i^{B}(\vec{x},\ t+\Delta t) = \left(1-\frac{\Delta t}{\tau}\right) f_i^{\text{temp}}(\vec{x},t) + \frac{\Delta t}{\tau}\, f_i^{\text{eq}}(\vec{x},t). \tag{A5}$$

b. With DDF-shifting

(1) Streaming:

$$f_i^{\text{temp}}(\vec{x},t) = f_i^{A}(\vec{x}-\vec{e}_i,\ t). \tag{A6}$$

(2) Collision (SRT):

$$\rho(\vec{x},t) = \left(\sum_i f_i^{\text{temp}}(\vec{x},t)\right) + 1, \tag{A7}$$

$$\vec{u}(\vec{x},t) = \frac{1}{\rho(\vec{x},t)} \sum_i \vec{c}_i\, f_i^{\text{temp}}(\vec{x},t), \tag{A8}$$

$$f_i^{\text{eq-shifted}}(\vec{x},t) = f_i^{\text{eq-shifted}}(\rho(\vec{x},t),\ \vec{u}(\vec{x},t)) = w_i\,\rho \cdot \left(\frac{(\vec{u}\circ\vec{c}_i)^2}{2\,c^4} - \frac{\vec{u}\circ\vec{u}}{2\,c^2} + \frac{\vec{u}\circ\vec{c}_i}{c^2}\right) + w_i\,(\rho - 1), \tag{A9}$$

$$f_i^{\text{B}}(\vec{x},\ t+\Delta t) = \left(1 - \frac{\Delta t}{\tau}\right) f_i^{\text{temp}}(\vec{x},t) + \frac{\Delta t}{\tau}\, f_i^{\text{eq}}(\vec{x},t). \tag{A10}$$

3. List of physical quantities and nomenclature

Quantity	SI-units	Defining equation(s)	Description
$\vec{x}$	m	$\vec{x} = (x, y, z)$	3D position in Cartesian coordinates
t	s	–	Time
Δx	m	$\Delta x := 1$	Lattice constant (in lattice units)
Δt	s	$\Delta t := 1$	Simulation time step (in lattice units)
c	$\frac{m}{s}$	$c := \frac{1}{\sqrt{3}}\frac{\Delta x}{\Delta t}$	Lattice speed of sound (in lattice units)
ρ	$\frac{kg}{m^3}$	$\rho = \sum_i f_i$	Mass density
$\vec{u}$	$\frac{m}{s}$	$\vec{u} = \sum_i \vec{c}_i f_i$	Velocity
f_i	$\frac{kg}{m^3}$	(A1)	Density distribution functions (DDFs)
f_i^{eq}	$\frac{kg}{m^3}$	(A4)	Equilibrium DDFs
i	1	$0 \leqslant i < q$	LBM streaming direction index
q	1	$q \in \{7, 9, 13, 15, 19, 27\}$	Number of LBM streaming directions
$\vec{c}_i$	$\frac{m}{s}$	[12], Eq. (11)	Streaming velocities
$\vec{e}_i$	m	$\vec{e}_i = \vec{c}_i\,\Delta t$	Streaming directions
w_i	1	[12], [Eq. (10)], $\sum_i w_i = 1$	Velocity set weights
τ	s	$\tau = \frac{\nu}{c^2} + \frac{\Delta t}{2}$	LBM relaxation time
ν	$\frac{m^2}{s}$	$\nu = \frac{\mu}{\rho}$	Kinematic shear viscosity
$\vec{f}$	$\frac{kg}{m^2\,s^2}$	$\vec{f} = \frac{\vec{F}}{V}$	Force per volume
(L_x, L_y, L_z)	(m, m, m)	$L_x\,L_y\,L_z = V$	Simulation box dimensions
g	$\frac{m}{s^2}$	$g := 9.81\,\frac{m}{s^2}$	Gravitational acceleration
σ	$\frac{kg}{s^2}$	–	Surface tension coefficient

4. Measured number format characteristics

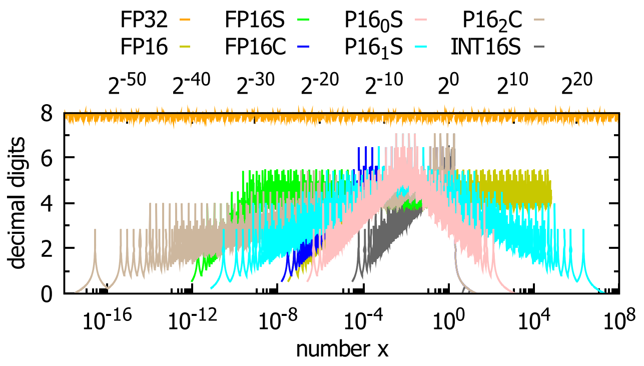
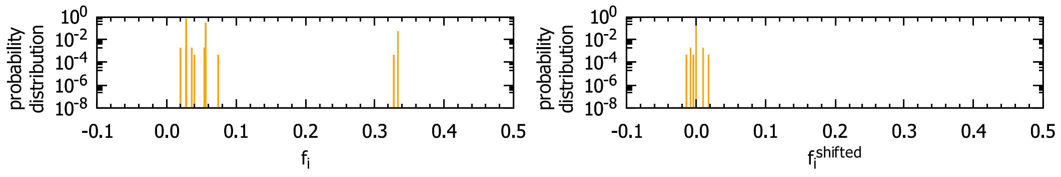
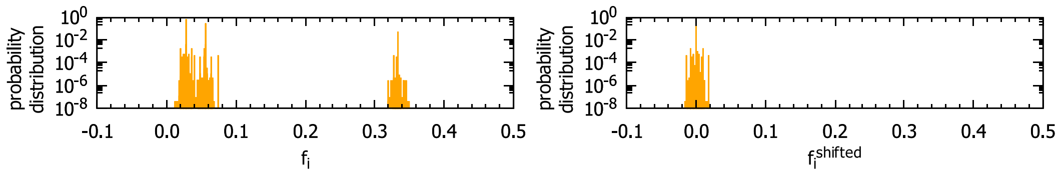


FIG. 18. Measured accuracy characteristics of the number formats investigated in this paper. The number of decimal digits for a given number x is computed via $-\log_{10}(|\log_{10}(\frac{x_{\text{represented}}}{x})|)$ [90,111,113]. Only the local minima are the relevant criterion for the error. Note that this definition for the number of decimal digits is off from the $\log_{10}(2^{n_m+1})$ definition by about 0.4.

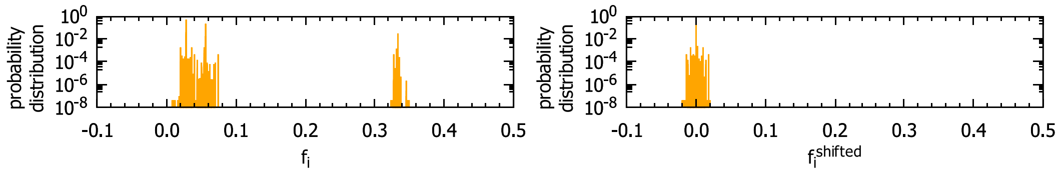
5. Numerical values of f_i and f_i^{shifted} for the lid-driven cavity (FP32/FP32)



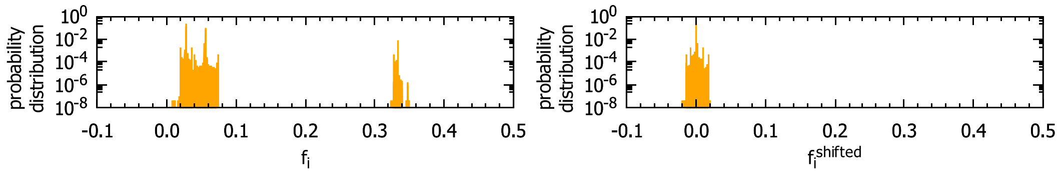
(a) Lid-driven cavity, after $t = 1$ time steps.



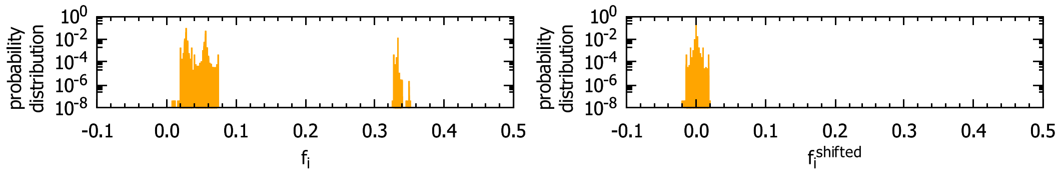
(b) Lid-driven cavity, after $t = 10$ time steps.



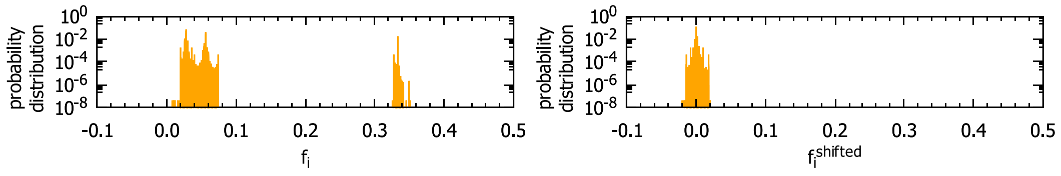
(c) Lid-driven cavity, after $t = 100$ time steps.



(d) Lid-driven cavity, after $t = 1000$ time steps.



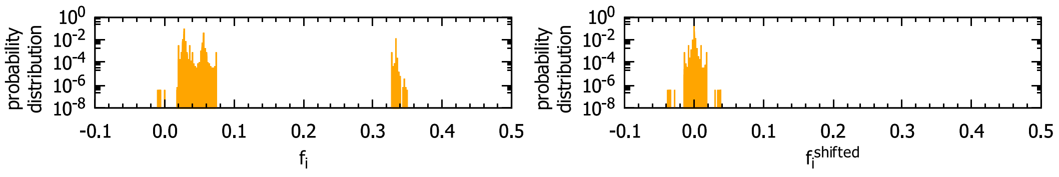
(e) Lid-driven cavity, after $t = 10000$ time steps.



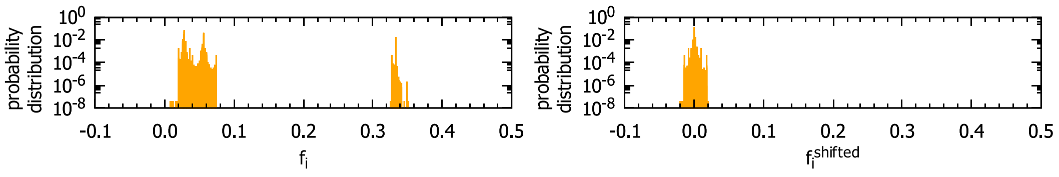
(f) Lid-driven cavity, after $t = 100000$ time steps.

FIG. 19. Numerical values of f_i and f_i^{shifted} for the lid-driven cavity (FP32/FP32, $\text{Re} = 1000$, $\text{Ma} = 0.17$, grid resolution $L = 128$) at various points in time.

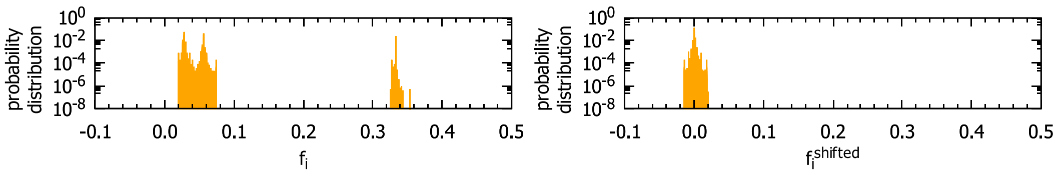
6. Numerical values of f_i and f_i^{shifted} for all setups (FP32/FP32)



(a) Lid-driven cavity at grid resolution $L = 64$.

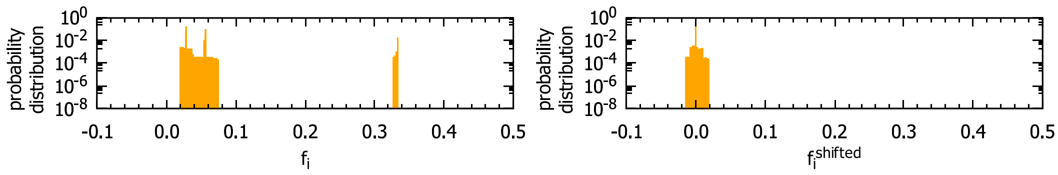


(b) Lid-driven cavity at grid resolution $L = 128$.

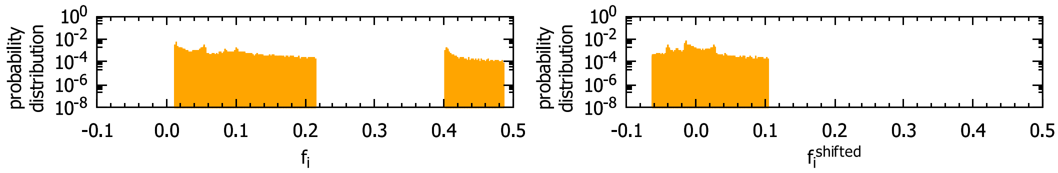


(c) Lid-driven cavity at grid resolution $L = 256$.

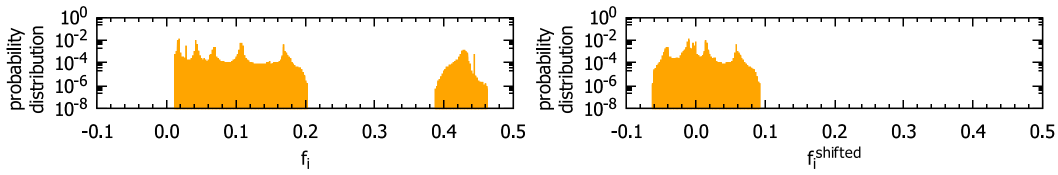
FIG. 20. Numerical values of f_i and f_i^{shifted} for the lid-driven cavity (FP32/FP32, Re = 1000, Ma = 0.17, after t = 100 000 time steps) at various grid resolutions.



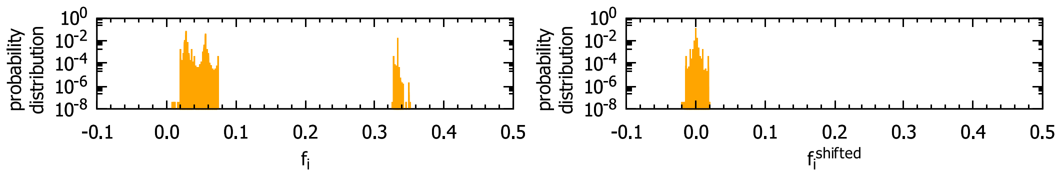
(a) Poiseuille flow, $R = 63$, $u_{\max} = 0.1$, $Re = 10$, at the time of convergence, like in figure 6.



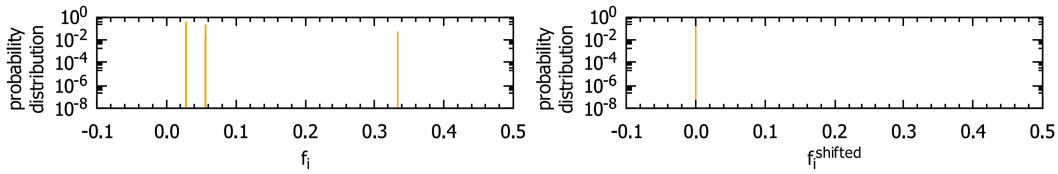
(b) Taylor-green vortices, at time of initialization $t = 0$.



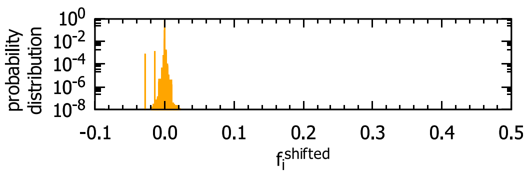
(c) Karman vortex street, after $t = 100000$ time steps.



(d) Lid-driven cavity, after $t = 100000$ time steps, like in figures 1 and 11.



(e) Microcapsule in shear flow at $Ca = 0.1$, after dimensionless time $\dot{\gamma}\, t = 5$.



(f) Raindrop impact at resolution $464 \times 464 \times 394$, with IBM particles, after $t = 10\,\mathrm{ms}$ time.

FIG. 21. Numerical values of f_i and f_i^{shifted} for all setups (FP32/FP32).

7. Properties of benchmarked hardware

TABLE V. Properties of benchmarked hardware.

	Data sheet				Measured memory bandwidth/ GB/s				LBM performance (D3Q19 SRT) / MLUPs/s				
Device	FP64/ TFLOPs/ s	FP32/ TFLOPs/ s	memory/ GB	bandwidth/ GB/ s	coalesced read	misaligned read	coalesced write	misaligned write	FP64/ FP64	FP64/ FP32	FP32/ FP32	FP32/ FP16S	FP32/ FP16C
AMD Instinct MI250 (1 GCD)	45.26	45.26	64	1638	969	1290	1034	634	3129	6125	5970	8131	7058
AMD Instinct MI100	11.54	46.14	32	1229	812	936	776	567	2675	4976	4753	8374	5995
AMD Radeon VII	3.46	13.83	16	1024	647	825	698	397	2306	3385	4906	6982	4006
Nvidia A100 (40GB)	9.75	19.49	40	1555	1334	1235	1514	207	4461	8417	8816	15455	9531
Nvidia Tesla V100S (32GB)	8.18	16.35	32	1134	911	930	951	153	2874	5205	5561	9208	6937
Nvidia Tesla V100 (16GB)	7.07	14.13	16	900	811	806	888	133	2644	5151	5233	9918	6972
Nvidia Tesla P100 (16GB)	4.76	9.52	16	732	504	306	544	90	1696	3315	3155	4834	3521
Nvidia Tesla P100 (12GB)	4.76	9.52	12	549	426	308	403	47	1258	2431	2390	3695	3107
Nvidia Tesla K40m	1.43	4.29	12	288	158	42	182	42	537	789	1089	1670	814
Nvidia Tesla K80 (1 GPU)	1.37	4.11	12	240	135	42	153	43	465	892	902	1511	856
Nvidia Tesla K20c	1.17	3.52	5	208	131	34	147	35	420	619	860	1322	617
AMD Radeon RX Vega 64	0.83	13.35	8	484	251	368	252	391	987	1424	1890	2956	2515
Nvidia GeForce RTX 3090	0.61	39.05	24	936	886	886	909	204	1337	1002	5493	10888	9347
Nvidia GeForce RTX 3080	0.47	29.77	10	760	692	693	709	171	1087	816	4225	8182	7380
Nvidia Quadro RTX 8000 Server	0.47	14.93	48	624	495	474	433	144	965	762	2658	5543	5645
Nvidia GeForce RTX 2080 Ti	0.42	13.45	11	616	533	531	579	159	1071	850	3285	6752	6717
Nvidia GeForce RTX 2060 S.	0.22	7.18	8	448	392	264	423	82	528	416	2500	4949	3897
Nvidia Tesla T4	0.25	8.14	16	300	259	264	226	85	527	415	1392	2952	2673
Nvidia GeForce GTX 1660 Ti	0.17	5.48	6	288	246	187	263	53	376	297	1469	2995	2783
Nvidia Titan Xp	0.38	12.15	12	548	445	163	514	112	1032	811	2897	5240	4695
Nvidia GeForce GTX 1080	0.31	9.78	8	320	257	201	275	80	740	580	1611	3009	2993
Nvidia Tesla P4	0.18	5.70	8	192	139	121	144	44	424	333	899	1694	1721
Nvidia GeForce GTX 1060M	0.14	4.44	6	192	156	104	165	44	348	274	966	1772	1598
Nvidia GeForce GTX 1050M Ti	0.08	2.49	4	112	98	65	102	27	194	153	622	1156	970
Nvidia Quadro P1000	0.06	1.89	4	82	70	47	73	21	145	114	446	838	723
Nvidia Quadro M4000	0.08	2.57	8	192	154	35	159	23	187	141	885	1493	888
Nvidia Tesla M60 (1 GPU)	0.15	4.82	8	160	143	56	145	39	346	254	842	1476	1532
Nvidia GeForce GTX 960M	0.05	1.51	4	80	73	31	70	14	112	83	434	791	513
Nvidia Quadro K2000	0.03	0.73	2	64	35	7	57	8	59	49	300	342	148
Nvidia GeForce GT 630 (OEM)	0.02	0.46	2	29	13	3	26	4	26	22	129	148	66
AMD Radeon Vega 8 Graphics	0.08	1.23	7	38	30	33	26	33	54	91	103	170	155
Intel UHD Graphics 630	0.12	0.46	7	51	38	34	28	18	80	98	155	183	170
Intel HD Graphics 5500	–	0.35	3	26	16	19	8	9	–	–	62	117	79
Intel HD Graphics 4600	–	0.38	2	26	19	10	11	7	–	–	60	76	30
Samsung Mali-G72 MP18	–	0.24	4	29	22	21	21	21	–	–	13	13	11
Intel Core i9-10980XE	1.81	3.23	128	94	60	65	26	30	110	192	193	152	152
Intel Core i5-9600	0.30	0.60	16	43	35	39	14	16	52	94	99	51	113
Intel Core i7-8700K	0.36	0.71	16	51	38	51	26	18	54	105	108	77	122
Intel Xeon Phi 7210	2.66	5.32	192	102	35	65	62	61	167	241	275	136	124
4x Intel Xeon E5-4620 v4	1.34	2.69	512	273	104	126	52	54	151	239	266	241	139
2x Intel Xeon E5-2630 v4	0.70	1.41	64	137	65	88	66	86	111	152	169	133	74
2x Intel Xeon E5-2623 v4	0.33	0.67	64	137	25	43	24	26	52	69	77	61	34
2x Intel Xeon E5-2680 v3	0.96	1.92	64	137	110	102	43	50	111	194	211	146	156
Intel Core i7-4770	0.22	0.44	16	26	24	26	15	12	37	53	61	13	34
Intel Core i7-4720HQ	0.17	0.33	16	26	22	22	9	11	8	9	11	11	28

8. Memory benchmarks

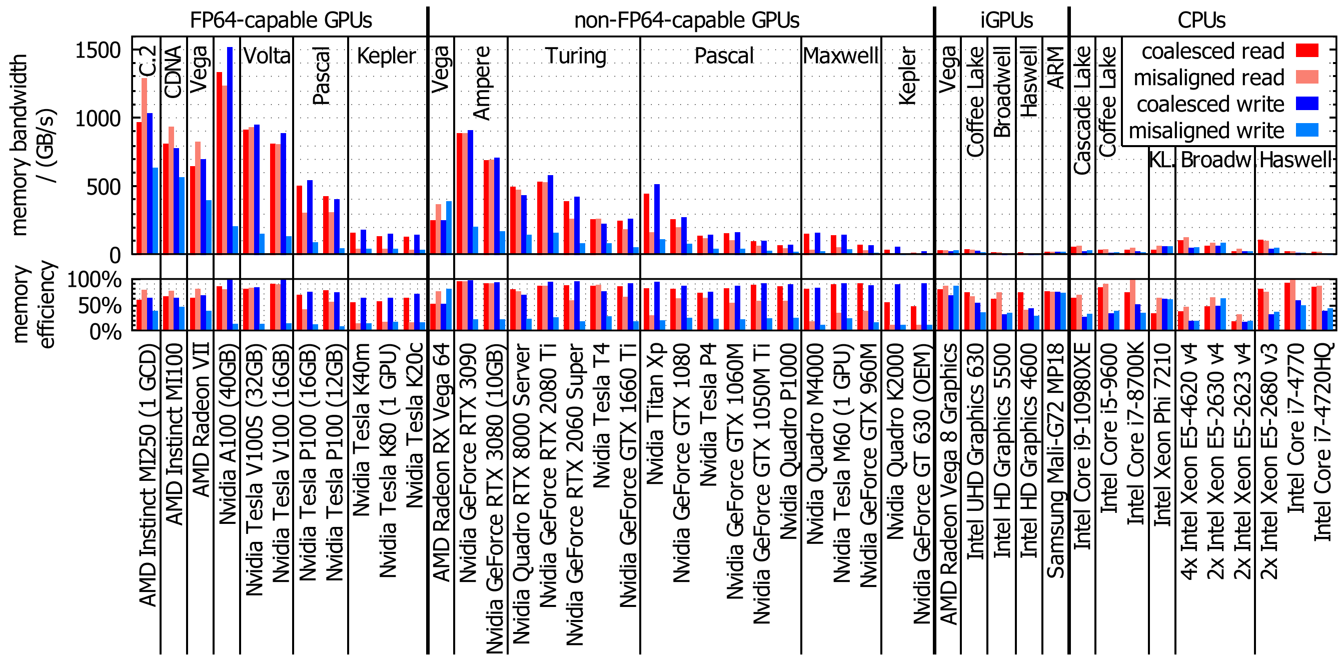


FIG. 22. Synthetic OpenCL memory benchmarks to measure coalesced/misaligned read/write performance. The misaligned write penalty is much larger than the misaligned read penalty across almost all tested devices. Values in Table V.

9. Ultrafast conversion algorithms

```
// FP32/FP32 macros
#define fpXX float // switchable data type
#define load(p,o) p[o] // regular float read
#define store(p,o,x) p[o]=x // regular float write

// FP32/FP16S macros
#define fpXX half // switchable data type
#define load(p,o) vload_half(o,p)*3.0517578E-5f // use OpenCL function
#define store(p,o,x) vstore_half_rte((x)*32768.0f,o,p) // use OpenCL function

// FP32/FP16C macros
#define fpXX ushort // switchable data type
#define load(p,o) half_to_float_custom(p[o]) // call conversion function
#define store(p,o,x) p[o]=float_to_half_custom(x) // call conversion function

// FP32/P160S macros
#define fpXX ushort // switchable data type
#define load(p,o) p160_to_float(p[o])*0.0078125f // call conversion function
#define store(p,o,x) p[o]=float_to_p160((x)*128.0f) // call conversion function

// FP32/P161S macros
#define fpXX ushort // switchable data type
#define load(p,o) p161_to_float(p[o])*0.0078125f // call conversion function
#define store(p,o,x) p[o]=float_to_p161((x)*128.0f) // call conversion function

// FP32/P162C macros
#define fpXX ushort // switchable data type
#define load(p,o) p162C_to_float(p[o]) // call conversion function
#define store(p,o,x) p[o]=float_to_p162C(x) // call conversion function



ushort float_to_half_custom(const float x) {
  const uint b = as_uint(x)+0x00000800; // round-to-nearest-even: add last bit after truncated mantissa
  const uint e = (b&0x7F800000)>>23; // exponent
  const uint m = b&0x007FFFFF; // mantissa; in line below: 0x007FF800 = 0x00800000-0x00000800 = decimal indicator flag -
      ↪ initial rounding
  return (b&0x80000000)>>16 | (e>112)*((((e-112)<<11)&0x7800)|m>>12) | ((e<113)&(e>100))*((((0x007FF800+m)>>(124-e))+1)
      ↪ >>1) | (e>127)*0x7FFF; // sign | normalized | denormalized | saturate
}
float half_to_float_custom(const ushort x) {
  const uint e = (x&0x7800)>>11; // exponent
  const uint m = (x&0x07FF)<<12; // mantissa
  const uint v = as_uint((float)m)>>23; // evil log2 bit hack to count leading zeros in denormalized format
  return as_float((x&0x8000)<<16 | (e!=0)*((e+112)<<23|m) | ((e==0)&(m!=0))*((v-37)<<23|((m<<(150-v))&0x007FF000))); //
      ↪ sign | normalized | denormalized
}



ushort float_to_p160(const float x) {
  const uint b = as_uint(x);
  const int e = ((b&0x7F800000)>>23)-127; // exponent-bias
  int m = (b&0x007FFFFF)>>9; // mantissa
  const int v = abs(e); // shift
  const int r = (e<0 ? 0x0002 : 0xFFFE)<<(13-v); // generate regime bits
  m = ((m>>(v-(e<0)))+1+(e<-13)*0x2)>>1; // rounding: add 1 after truncated position; in case of lowest numbers, saturate
  return (b&0x80000000)>>16 | (e>-16)*((r+m)&0x7FFF) | (e>13)*0x7FFF; // sign | regime+mantissa ("+" handles rounding
      ↪ overflow) | saturate
}
```

```
float p160_to_float(const ushort x) {
  const uint sr = (x>>14)&1; // sign of regime
  ushort t = x<<2; // remove sign and first regime bit
  t = sr ? ~t : t; // positive regime r>=0 : negative regime r<0
  const int r = 142-(as_int((float)t)>>23); // evil log2 bit hack to count leading zeros for regime
  const uint m = (x<<(r+10))&0x007FFFFF; // extract mantissa and bit-shift it in place
  const int rs = sr ? r : -r-1; // negative regime r<0 : positive regime r>=0
  return as_float((x&0x8000)<<16 | (r!=158)*((rs+127)<<23 | m)); // sign | regime | mantissa
}



ushort float_to_p161(const float x) {
  const uint b = as_uint(x);
  const int e = ((b&0x7F800000)>>23)-127; // exponent-bias
  int m = (b&0x007FFFFF)>>10; // mantissa
  const int ae = abs(e);
  const int v = ae>>1; // shift, ">>1" is the same as "/2"
  const int e2 = ae&1; // "&1" is the same as "%2"
  const int r = ((e<0 ? 0x0002 : 0xFFFE<<e2)+e2)<<(13-v-e2); // generate regime bits, merge regime+exponent and shift in
      ↪ place
  m = ((m>>(v-(e<0)*(1-e2)))+(e>-28)+(e<-26)*0x3)>>1; // rounding: add 1 after truncated position; in case of lowest
      ↪ numbers, saturate
  return (b&0x80000000)>>16 | (e>-31)*((r+m)&0x7FFF) | (e>26)*0x7FFF; // sign | regime+exponent+mantissa ("+" handles
      ↪ rounding overflow) | saturate
}
float p161_to_float(const ushort x) {
  const uint sr = (x>>14)&1; // sign of regime
  ushort t = x<<2; // remove sign and first regime bit
  t = sr ? ~t : t; // positive regime r>=0 : negative regime r<0
  const int r = 142-(as_int((float)t)>>23); // evil log2 bit hack to count leading zeros for regime
  const uint e = (x>>(12-r))&1; // extract mantissa and bit-shift it in place
  const uint m = (x<<(r+11))&0x007FFFFF; // extract mantissa and bit-shift it in place
  const int rs = (sr ? r : -r-1)<<1; // negative regime r<0 : positive regime r>=0, "<<1" is the same as "*2"
  return as_float((x&0x8000)<<16 | (r!=158)*((rs+e+127)<<23 | m)); // sign | regime+exponent | mantissa
}



ushort float_to_p162C(const float x) {
  const int b = as_int(x);
  const int e = ((b&0x7F800000)>>23)-127; // exponent-bias
  int m = (b&0x007FFFFF)>>10; // mantissa
  const int ae = -e;
  const int v = ae>>2; // shift, ">>2" is the same as "/4"
  const int r = 0x4000>>v; // generate regime bits, merge regime+exponent and shift in place
  const int e4 = (3-(ae&3))<<(12-v); // generate exponent and shift in place
  m = ((m>>v)+(e>-54)+(e<-51)*0x3)>>1; // rounding: add 1 after truncated position; in case of lowest numbers, saturate
  return (b&0x80000000)>>16 | (e>-59)*((r+e4+m)&0x7FFF) | (e>0)*0x7FFF; // sign | regime+exponent+mantissa ("+" handles
      ↪ rounding overflow) | saturate
}
float p162C_to_float(const ushort x) {
  const int r = 158-(as_int((float)((uint)x<<17))>>23); // remove sign bit, evil log2 bit hack to count leading zeros for
      ↪ regime
  const int e = (x>>(12-r))&3; // extract mantissa and bit-shift it in place
  const int m = (x<<(r+11))&0x007FFFFF; // extract mantissa and bit-shift it in place
  return as_float((x&0x8000)<<16 | (r!=158)*((124-(r<<2)+e)<<23 | m)); // sign | regime+exponent | mantissa
}
```

Listing 1: OpenCL C macros for regular FP32, for FP16S using hardware-accelerated IEEE-754 FP16 floating-point conversion and for our FP16C format with calls to our manual floating-point conversion functions. Manual floating-point conversion functions for FP32 $\leftrightarrow$ FP16C (`float`$\leftrightarrow$`half`) in OpenCL C. We also provide macros and conversion algorithms for FP32 $\leftrightarrow$ P16$_0$S/P16$_1$S/P16$_2$C posit formats. The saturation term in the algorithms can be omitted if it is made sure that larger than maximum numbers are never used, which is the case in this LBM application.

10. LBM core of the FluidX3D OpenCL C implementation (D3Q19 SRT FP32/xx)

```
float __attribute__((always_inline)) sq(const float x) {
  return x*x;
}
uint3 __attribute__((always_inline)) coordinates(const uint n) { // disassemble 1D index to 3D coordinates (n -> x,y,z)
  const uint t = n%(def_sx*def_sy);
  return (uint3)(t%def_sx, t/def_sx, n/(def_sx*def_sy)); // n = x+(y+z*sy)*sx

uint __attribute__((always_inline)) f_index(const uint n, const uint i) { // 32-bit indexing (maximum box size for D3Q19:
     ↪ 608x608x608)
  return i*def_s+n; // SoA (229% faster on GPU compared to AoS)
}
void __attribute__((always_inline)) equilibrium(const float rho, float ux, float uy, float uz, float* feq) { // calculate
     ↪ f_equilibrium
  const float c3 = -3.0f*(sq(ux)+sq(uy)+sq(uz)), rhom1=rho-1.0f; // c3=-2*sq(u)/(2*sq(c))
  ux *= 3.0f;
  uy *= 3.0f;
  uz *= 3.0f;
  feq[ 0] = def_w0*fma(rho, 0.5f*c3, rhom1); // 000 (identical for all velocity sets)
  const float u0=ux+uy, u1=ux+uz, u2=uy+uz, u3=ux-uy, u4=ux-uz, u5=uy-uz;
  const float rhos=def_ws*rho, rhoe=def_we*rho, rhom1s=def_ws*rhom1, rhom1e=def_we*rhom1;
  feq[ 1] = fma(rhos, fma(0.5f, fma(ux, ux, c3), ux), rhom1s); feq[ 2] = fma(rhos, fma(0.5f, fma(ux, ux, c3), -ux), rhom1s
       ↪ ); // +00 -00
  feq[ 3] = fma(rhos, fma(0.5f, fma(uy, uy, c3), uy), rhom1s); feq[ 4] = fma(rhos, fma(0.5f, fma(uy, uy, c3), -uy), rhom1s
       ↪ ); // 0+0 0-0
  feq[ 5] = fma(rhos, fma(0.5f, fma(uz, uz, c3), uz), rhom1s); feq[ 6] = fma(rhos, fma(0.5f, fma(uz, uz, c3), -uz), rhom1s
       ↪ ); // 00+ 00-
  feq[ 7] = fma(rhoe, fma(0.5f, fma(u0, u0, c3), u0), rhom1e); feq[ 8] = fma(rhoe, fma(0.5f, fma(u0, u0, c3), -u0), rhom1e
       ↪ ); // ++0 --0
  feq[ 9] = fma(rhoe, fma(0.5f, fma(u1, u1, c3), u1), rhom1e); feq[10] = fma(rhoe, fma(0.5f, fma(u1, u1, c3), -u1), rhom1e
       ↪ ); // +0+ -0-
  feq[11] = fma(rhoe, fma(0.5f, fma(u2, u2, c3), u2), rhom1e); feq[12] = fma(rhoe, fma(0.5f, fma(u2, u2, c3), -u2), rhom1e
       ↪ ); // 0++ 0--
  feq[13] = fma(rhoe, fma(0.5f, fma(u3, u3, c3), u3), rhom1e); feq[14] = fma(rhoe, fma(0.5f, fma(u3, u3, c3), -u3), rhom1e
       ↪ ); // +-0 -+0
  feq[15] = fma(rhoe, fma(0.5f, fma(u4, u4, c3), u4), rhom1e); feq[16] = fma(rhoe, fma(0.5f, fma(u4, u4, c3), -u4), rhom1e
       ↪ ); // +0- -0+
  feq[17] = fma(rhoe, fma(0.5f, fma(u5, u5, c3), u5), rhom1e); feq[18] = fma(rhoe, fma(0.5f, fma(u5, u5, c3), -u5), rhom1e
       ↪ ); // 0+- 0-+
}
void __attribute__((always_inline)) fields(const float* f, float* rhon, float* uxn, float* uyn, float* uzn) { // calculate
     ↪  density and velocity from fi
  float rho=f[0], ux, uy, uz;
  #pragma unroll
  for(uint i=1; i<def_set; i++) rho += f[i]; // calculate density from f
  rho += 1.0f; // add 1.0f last to avoid digit extinction effects when summing up f
  ux = f[ 1]-f[ 2]+f[ 7]-f[ 8]+f[ 9]-f[10]+f[13]-f[14]+f[15]-f[16]; // calculate velocity from fi (alternating + and - for
       ↪  best accuracy)
  uy = f[ 3]-f[ 4]+f[ 7]-f[ 8]+f[11]-f[12]+f[14]-f[13]+f[17]-f[18];
  uz = f[ 5]-f[ 6]+f[ 9]-f[10]+f[11]-f[12]+f[16]-f[15]+f[18]-f[17];
  *rhon = rho;
  *uxn = ux/rho;
  *uyn = uy/rho;
  *uzn = uz/rho;
}
void __attribute__((always_inline)) neighbors(const uint n, uint* j) { // calculate neighbor indices
  const uint3 xyz = coordinates(n);
  const uint x0 =   xyz.x; // pre-calculate indices (periodic boundary conditions on simulation box walls)
  const uint xp =  (xyz.x       +1)%def_sx;
  const uint xm =  (xyz.x+def_sx-1)%def_sx;
  const uint y0 =   xyz.y                 *def_sx;
  const uint yp = ((xyz.y       +1)%def_sy)*def_sx;
  const uint ym = ((xyz.y+def_sy-1)%def_sy)*def_sx;
  const uint z0 =   xyz.z                  *def_sy*def_sx;
  const uint zp = ((xyz.z       +1)%def_sz)*def_sy*def_sx;
  const uint zm = ((xyz.z+def_sz-1)%def_sz)*def_sy*def_sx;
  j[0] = n;
  j[ 1] = xp+y0+z0; j[ 2] = xm+y0+z0; // +00 -00
  j[ 3] = x0+yp+z0; j[ 4] = x0+ym+z0; // 0+0 0-0
  j[ 5] = x0+y0+zp; j[ 6] = x0+y0+zm; // 00+ 00-
  j[ 7] = xp+yp+z0; j[ 8] = xm+ym+z0; // ++0 --0
  j[ 9] = xp+y0+zp; j[10] = xm+y0+zm; // +0+ -0-
  j[11] = x0+yp+zp; j[12] = x0+ym+zm; // 0++ 0--
  j[13] = xp+ym+z0; j[14] = xm+yp+z0; // +-0 -+0
  j[15] = xp+y0+zm; j[16] = xm+y0+zp; // +0- -0+
  j[17] = x0+yp+zm; j[18] = x0+ym+zp; // 0+- 0-+
}
kernel void initialize(global fpXX* fc, global float* rho, global float* u) {
  const uint n = get_global_id(0); // n = x+(y+z*sy)*sx
  float feq[def_set]; // f_equilibrium
  equilibrium(rho[n], u[n], u[def_s+n], u[2*def_s+n], feq);
  #pragma unroll
  for(uint i=0; i<def_set; i++) store(fc, f_index(n,i), feq[i]); // write to fc
} // initialize()
kernel void stream_collide(const global fpXX* fc, global fpXX* fs, global float* rho, global float* u, global uchar* flags
     ↪ ) {
  const uint n = get_global_id(0); // n = x+(y+z*sy)*sx
  const uchar flagsn = flags[n]; // cache flags[n] for multiple readings
  if(flagsn&TYPE_W) return; // if node is boundary node, just return (slight speed up)
  uint j[def_set]; // neighbor indices
  neighbors(n, j); // calculate neighbor indices
  uchar flagsj[def_set]; // cache neighbor flags for multiple readings
  flagsj[0] = flagsn;
  #pragma unroll
  for(uint i=1; i<def_set; i++) flagsj[i] = flags[j[i]];
  // read from fc in video memory and stream to fhn
  float fhn[def_set]; // cache f_half_step[n], do streaming step
  fhn[0] = fc[f_index(n, 0)]; // keep old center population
  #pragma unroll
  for(uint i=1; i<def_set; i+=2) { // perform streaming
    fhn[i  ] = load(fc, flagsj[i+1]&TYPE_W ? f_index(n, i+1) : f_index(j[i+1], i  )); // boundary : regular
    fhn[i+1] = load(fc, flagsj[i  ]&TYPE_W ? f_index(n, i  ) : f_index(j[i  ], i+1));
  }
  // collide fh
  float rhon, uxn, uyn, uzn; // cache density and velocity for multiple writings/readings
  fields(fhn, &rhon, &uxn, &uyn, &uzn); // calculate density and velocity fields from f
  uxn = clamp(uxn, -def_c, def_c); // limit velocity (for stability purposes)
  uyn = clamp(uyn, -def_c, def_c);
  uzn = clamp(uzn, -def_c, def_c);
  float feq[def_set]; // cache f_equilibrium[n]
  equilibrium(rhon, uxn, uyn, uzn, feq); // calculate equilibrium populations
  #pragma unroll
  for(uint i=0; i<def_set; i++) store(fs, f_index(n,i), fma(1.0f-def_w, fhn[i], def_w*feq[i])); // write to fs in video
       ↪ memory
} // stream_collide()
```

Listing 2: LBM core of the FluidX3D OpenCL C implementation (D3Q19 SRT FP32/xx).

[1] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen, in *The Lattice Boltzmann Method* (Springer International Publishing, Cham, Switzerland, 2017), Vol. 10, p. 978.

[2] S. Chapman, T. G. Cowling, and D. Burnett, *The Mathematical Theory of Non-Uniform Gases: An Account of the Kinetic Theory of Viscosity, Thermal Conduction and Diffusion in Gases* (Cambridge University Press, Cambridge, UK, 1990).

[3] H. Pradipto and A. Purqon, Accuracy and numerical stability analysis of lattice Boltzmann method with multiple relaxation time for incompressible flows, in J. Phys.: Conf. Ser. (IOP Publishing, Bristol, UK, 2017), Vol. 877, p. 012035.

[4] R. Benzi, S. Succi, and M. Vergassola, The lattice Boltzmann equation: Theory and applications, Phys. Rep. **222**, 145 (1992).

[5] P. M. Tekic, J. B. Radjenovic, and M. Rackovic, Implementation of the lattice Boltzmann method on heterogeneous hardware and platforms using OpenCL, Adv. Electr. Comput. Eng. **12**, 51 (2012).

[6] F. Häusl, MPI-based multi-GPU extension of the lattice Boltzmann method, Bachelor's Thesis, University of Bayreuth (2019), https://epub.uni-bayreuth.de/5689/.

[7] F. Häusl, Soft objects in newtonian and non-Newtonian fluids: A computational study of bubbles and capsules in flow, Master's Thesis, University of Bayreuth (2022), https://epub.uni-bayreuth.de/5960/.

[8] M. Lehmann and S. Gekle, Analytic solution to the piecewise linear interface construction problem and its application in curvature calculation for volume-of-fluid simulation codes, Computation **10**, 21 (2022).

[9] M. Lehmann, Esoteric Pull and Esoteric Push: Two simple inplace streaming schemes for the lattice Boltzmann method on GPUs, Computation **10**, 92 (2022).

[10] M. Lehmann, L. M. Oehlschlägel, F. P. Häusl, A. Held, and S. Gekle, Ejection of marine microplastics by raindrops: A computational and experimental study, Microplast. Nanoplast. **1**, 1 (2021).

[11] H. Laermanns, M. Lehmann, M. Klee, M. G. Löder, S. Gekle, and C. Bogner, Tracing the horizontal transport of microplastics on rough surfaces, Microplast. Nanoplast. **1**, 11 (2021).

[12] M. Lehmann, High performance free surface LBM on GPUs, Master's Thesis, University of Bayreuth (2019), https://epub.uni-bayreuth.de/5400/.

[13] M. Schreiber, P. Neumann, S. Zimmer, and H.-J. Bungartz, Free-surface lattice-Boltzmann simulation on many-core architectures, Proc. Comput. Sci. **4**, 984 (2011).

[14] M. Holzer, M. Bauer, and U. Rüde, Highly efficient lattice-Boltzmann multiphase simulations of immiscible fluids at high-density ratios on CPUs and GPUs through code generation, arXiv:2012.06144.

[15] M. Takáč and I. Petráš, Cross-platform GPU-based implementation of lattice Boltzmann method solver using ArrayFire library, Mathematics **9**, 1793 (2021).

[16] M. Q. Ho, C. Obrecht, B. Tourancheau, B. D. de Dinechin, and J. Hascoet, 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)* (IEEE, San Diego, California, 2017), pp. 1–9.

[17] J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein, Performance engineering for the lattice Boltzmann method on GPGPUs: Architectural requirements and performance results, Comput. Fluids **80**, 276 (2013).

[18] C. Riesinger, A. Bakhtiari, M. Schreiber, P. Neumann, and H.-J. Bungartz, A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters, Computation **5**, 48 (2017).

[19] E. O. Aksnes and A. C. Elster, Porous rock simulations and lattice Boltzmann on GPUs, in *Parallel Computing: From Multicores and GPU's to Petascale* (IOS Press, Amsterdam, Netherlands, 2010), pp. 536–545.

[20] A. Kummerländer, M. Dorn, M. Frank, and M. J. Krause, Implicit propagation of directly addressed grids in lattice Boltzmann methods (2021), doi: 10.13140/RG.2.2.35085.87523.

[21] M. Geveler, D. Ribbrock, D. Göddeke, and S. Turek, 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.

[22] J. Bény, C. Kotsalos, and J. Latt, Toward full GPU implementation of fluid-structure interaction, in *Proceedings of the 2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)* (IEEE, Amsterdam, 2019), pp. 16–22.

[23] G. Boroni, J. Dottori, and P. Rinaldi, Full GPU implementation of lattice-Boltzmann methods with immersed boundary conditions for fast fluid simulations, Int. J. Multiphys. **11**, 1 (2017).

[24] M. Griebel, M. A. Schweitzer *et al.*, *Meshfree Methods for Partial Differential Equations II* (Springer, Cham, Switzerland, 2005).

[25] H.-J. Limbach, A. Arnold, B. A. Mann, and C. Holm, ESPResSo—an extensible simulation package for research on soft matter systems, Comput. Phys. Commun. **174**, 704 (2006).

[26] Institute for Computational Physics, Universität Stuttgart, ESPResSo user's guide, http://espressomd.org/wordpress/wp-content/uploads/2016/07/ug_07_2016.pdf (2016), accessed June 15, 2018.

[27] S. D. Walsh, M. O. Saar, P. Bailey, and D. J. Lilja, Accelerating geoscience and engineering system simulations on graphics hardware, Comput. Geosci. **35**, 2353 (2009).

[28] S. Zitz, A. Scagliarini, and J. Harting, Lattice Boltzmann simulations of stochastic thin film dewetting, Phys. Rev. E **104**, 034801 (2021).

[29] C. Wei, W. Zhenghua, L. Zongzhe, Y. Lu, and W. Yongxian, An improved lbm approach for heterogeneous gpu-cpu clusters, in *Proceedings of the 2011 4th International Conference on Biomedical Engineering and Informatics (BMEI)* (IEEE, Shanghai, China, 2011), Vol. 4, pp. 2095–2098.

[30] M. J. Mawson and A. J. Revell, Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs, Comput. Phys. Commun. **185**, 2566 (2014).

[31] J. Tölke and M. Krafczyk, TeraFLOP computing on a desktop PC with GPUs for 3D CFD, Int. J. Comput. Fluid Dyn. **22**, 443 (2008).

[32] G. Herschlag, S. Lee, J. S. Vetter, and A. Randles, GPU data access on complex geometries for D3Q19 lattice Boltzmann method, in *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (IEEE, Vancouver, BC, Canada, 2018), pp. 825–834.

[33] N. Delbosc, J. L. Summers, A. Khan, N. Kapur, and C. J. Noakes, Optimized implementation of the lattice Boltzmann method on a graphics processing unit towards real-time fluid simulation, Comput. Math. Appl. **67**, 462 (2014).

[34] P. Bailey, J. Myre, S. D. Walsh, D. J. Lilja, and M. O. Saar, Accelerating lattice Boltzmann fluid flow simulations using

graphics processors, in *Proceedings of the 2009 International Conference on Parallel Processing* (IEEE, Vienna, Austria, 2009), pp. 550–557.

[35] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, Multi-GPU implementation of the lattice Boltzmann method, Comput. Math. Appl. **65**, 252 (2013).

[36] W. B. de Oliveira Jr, A. Lugarini, and A. T. Franco, Performance analysis of the lattice Boltzmann method implementation on GPU, in *XL CILAMCE 2019 Ibero-Latin Congress on Computational Methods in Engineering (ABMEC), Natal, Brazil* (2019).

[37] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, A new approach to the lattice Boltzmann method for graphics processing units, Comput. Math. Appl. **61**, 3628 (2011).

[38] N.-P. Tran, M. Lee, and S. Hong, Performance optimization of 3D lattice Boltzmann flow solver on a GPU, Sci. Program. **2017** (2017).

[39] P. R. Rinaldi, E. Dari, M. J. Vénere, and A. Clausse, A lattice-Boltzmann solver for 3D fluid simulation on GPU, Simul. Modell. Pract. Theory **25**, 163 (2012).

[40] P. R. Rinaldi, E. A. Dari, M. J. Vénere, and A. Clausse, Fluid simulation with lattice Boltzmann methods implemented on GPUs using CUDA, in *High-Performance Computing Symposium (HPC2009), San Diego, California, USA* (2009).

[41] J. Beny and J. Latt, Efficient LBM on GPUs for dense moving objects using immersed boundary condition, arXiv:1904.02108.

[42] J. Ames, D. F. Puleri, P. Balogh, J. Gounley, E. W. Draeger, and A. Randles, Multi-GPU immersed boundary method hemodynamics simulations, J. Comput. Sci. **44**, 101153 (2020).

[43] Q. Xiong, B. Li, J. Xu, X. Fang, X. Wang, L. Wang, X. He, and W. Ge, Efficient parallel implementation of the lattice Boltzmann method on large clusters of graphic processing units, Chin. Sci. Bull. **57**, 707 (2012).

[44] H. Zhu, X. Xu, G. Huang, Z. Qin, and B. Wen, An efficient graphics processing unit scheme for complex geometry simulations using the lattice Boltzmann method, IEEE Access **8**, 185158 (2020).

[45] J. Duchateau, F. Rousselle, N. Maquignon, G. Roussel, and C. Renaud, 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)* (IEEE, Krakow, Poland, 2015), pp. 315–322.

[46] C. F. Janßen, D. Mierke, M. Überrück, S. Gralher, and T. Rung, Validation of the GPU-accelerated CFD solver ELBE for free surface flow problems in civil and environmental engineering, Computation **3**, 354 (2015).

[47] J. Habich, T. Zeiser, G. Hager, and G. Wellein, Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA, Adv. Eng. Software **42**, 266 (2011).

[48] E. Calore, D. Marchi, S. F. Schifano, and R. Tripiccione, Optimizing communications in multi-GPU lattice Boltzmann simulations, in *Proceedings of the 2015 International Conference on High Performance Computing & Simulation (HPCS)* (IEEE, Amsterdam, 2015), pp. 55–62.

[49] P.-Y. Hong, L.-M. Huang, L.-S. Lin, and C.-A. Lin, Scalable multi-relaxation-time lattice Boltzmann simulations on multi-GPU cluster, Comput. Fluids **110**, 1 (2015).

[50] W. Xian and A. Takayuki, Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster, Parallel Comput. **37**, 521 (2011).

[51] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, 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* (Springer, Berkeley, California, 2010), pp. 151–161.

[52] F. Kuznik, C. Obrecht, G. Rusaouen, and J.-J. Roux, LBM based flow simulation using GPU computing processor, Comput. Math. Appl. **59**, 2380 (2010).

[53] C. Feichtinger, J. Habich, H. Köstler, G. Hager, U. Rüde, and G. Wellein, A flexible Patch-based lattice Boltzmann parallelization approach for heterogeneous GPU–CPU clusters, Parallel Comput. **37**, 536 (2011).

[54] E. Calore, A. Gabbana, J. Kraus, E. Pellegrini, S. F. Schifano, and R. Tripiccione, Massively parallel lattice–Boltzmann codes on large GPU clusters, Parallel Comput. **58**, 1 (2016).

[55] A. Horga, With lattice Boltzmann models using CUDA enabled GPGPUs, Master's thesis, University of Timisoara, Romania (2013).

[56] N. Onodera, Y. Idomura, S. Uesawa, S. Yamashita, and H. Yoshida, Locally mesh-refined lattice Boltzmann method for fuel debris air cooling analysis on GPU supercomputer, Mech. Eng. J. **7**, 19-00531 (2020).

[57] G. Falcucci, G. Amati, P. Fanelli, V. K. Krastev, G. Polverino, M. Porfiri, and S. Succi, Extreme flow simulations reveal skeletal adaptations of deep-sea sponges, Nature (London) **595**, 537 (2021).

[58] S. Zitz, A. Scagliarini, S. Maddu, A. A. Darhuber, and J. Harting, Lattice Boltzmann method for thin-liquid-film hydrodynamics, Phys. Rev. E **100**, 033313 (2019).

[59] M. Mohrhard, G. Thäter, J. Bludau, B. Horvat, and M. Krause, An auto-vecotorization friendly parallel lattice Boltzmann streaming scheme for direct addressing, Comput. Fluids **181**, 1 (2019).

[60] F. Gray and E. Boek, Enhancing computational precision for lattice Boltzmann schemes in porous media flows, Computation **4**, 11 (2016).

[61] W. Li, Y. Ma, X. Liu, and M. Desbrun, Efficient kinetic simulation of two-phase flows, ACM Trans. Graphics **41**, 114 (2022).

[62] M. Geier and M. Schönherr, Esoteric twist: An efficient in-place streaming algorithmus for the lattice Boltzmann method on massively parallel hardware, Computation **5**, 19 (2017).

[63] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein, Comparison of different propagation steps for lattice Boltzmann methods, Comput. Math. Appl. **65**, 924 (2013).

[64] M. Wittmann, Ph.D. thesis, Friedrich-Alexander-Universitt Erlangen-Nürnberg, Erlangen, Germany (2016), https://nbn-resolving.org/urn:nbn:de:bvb:29-opus4-74586.

[65] F. Bonaccorso, A. Montessori, A. Tiribocchi, G. Amati, M. Bernaschi, M. Lauricella, and S. Succi, Lbsoft: A parallel open-source software for simulation of colloidal systems, Comput. Phys. Commun. **256**, 107455 (2020).

[66] P. Skordos, Initial and boundary conditions for the lattice Boltzmann method, Phys. Rev. E **48**, 4823 (1993).

[67] G. Wellein, P. Lammers, G. Hager, S. Donath, and T. Zeiser, Towards optimal performance for lattice Boltzmann applications on terascale computers, in *Parallel Computational Fluid Dynamics 2005* (Elsevier, Amsterdam, Netherlands, 2006), pp. 31–40.

[68] M. J. Krause, A. Kummerländer, S. J. Avis, H. Kusumaatmaja, D. Dapelo, F. Klemens, M. Gaedtke, N. Hafen, A. Mink, R. Trunk *et al.*, OpenLB—Open source lattice Boltzmann code, Comput. Math. Appl. **81**, 258 (2021).

[69] J. Latt and M. Krause, OpenLB release 0.3: Open source lattice Boltzmann code (2007).

[70] V. Heuveline and M. Krause, OpenLB: Towards an efficient parallel open source library for lattice Boltzmann fluid flow simulations, in *PARA'08 Workshop on State-of-the-Art in Scientific and Parallel Computing, May 13-16, 2008*, Lecture Notes in Computer Science (LNCS) Vols. No. 6126 and No. 6127 (Springer, Trondheim, Norway, 2011) published online 2011, https://para08.idi.ntnu.no/docs/submission_37.pdf.

[71] M. Krause, S. Avis, H. Kusumaatmaja, D. Dapelo, M. Gaedtke, N. Hafen, M. Haußmann, J. Jeppener-Haltenhoff, L. Kronberg, A. Kummerländer, J. Marquardt, T. Pertzel, S. Simonis, R. Trunk, M. Wu, and A. Zarth, OpenLB release 1.4: Open source lattice Boltzmann code (2020).

[72] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M. B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li *et al.*, Palabos: Parallel lattice Boltzmann solver, Comput. Math. Appl. **81**, 334 (2021).

[73] T. Min, G. Weidong, P. Jingshan, and G. Meng, Performance analysis and optimization of PalaBos on petascale Sunway BlueLight MPP Supercomputer, Procedia Eng. **61**, 241 (2013).

[74] L. Mountrakis, E. Lorenz, O. Malaspinas, S. Alowayyed, B. Chopard, and A. G. Hoekstra, Parallel performance of an IB-LBM suspension simulation framework, J. Comput. Sci. **9**, 45 (2015).

[75] C. Kotsalos, J. Latt, and B. Chopard, Bridging the computational gap between mesoscopic and continuum modeling of red blood cells for fully resolved blood flow, J. Comput. Phys. **398**, 108905 (2019).

[76] C. Kotsalos, J. Latt, and B. Chopard, Palabos-npFEM: Software for the simulation of cellular blood flow (digital blood), J. Open Res. Software **9**, 16 (2021).

[77] G. Wellein, T. Zeiser, G. Hager, and S. Donath, On the single processor performance of simple lattice Boltzmann kernels, Comput. Fluids **35**, 910 (2006).

[78] A. Lintermann and W. Schröder, Lattice–Boltzmann simulations for complex geometries on high-performance computers, CEAS Aeronaut. J. **11**, 745 (2020).

[79] S. Schmieschek, L. Shamardin, S. Frijters, T. Krüger, U. D. Schiller, J. Harting, and P. V. Coveney, LB3D: A parallel implementation of the lattice-Boltzmann method for simulation of Interacting amphiphilic fluids, Comput. Phys. Commun. **217**, 149 (2017).

[80] IEEE Computer Society. Standards Committee and American National Standards Institute, IEEE standard for binary floating-point arithmetic, IEEE Std 754-2019 (Revision of IEEE 754-2008) 754 (1985).

[81] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Comput. Surv. **23**, 5 (1991).

[82] IS Committee and others, 754–2008 IEEE standard for floating-point arithmetic, IEEE Comput. Soc. Std. **2008** (2008).

[83] W. Kahan, IEEE standard 754 for binary floating-point arithmetic, Lect. Notes Status IEEE **754**, 11 (1996).

[84] T. Grützmacher and H. Anzt, A modular precision format for decoupling arithmetic format and storage format, in *European Conference on Parallel Processing* (Springer, Turin, Italy, 2018), pp. 434–443.

[85] H. Anzt, G. Flegar, T. Grützmacher, and E. S. Quintana-Ortí, Toward a modular precision ecosystem for high-performance computing, Int. J. High Perform. Comput. Appl. **33**, 1069 (2019).

[86] M. Krause, Fluid flow dimulation 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), 2010, http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019768.

[87] S. Succi, G. Amati, M. Bernaschi, G. Falcucci, M. Lauricella, and A. Montessori, Towards exascale lattice Boltzmann computing, Comput. Fluids **181**, 107 (2019).

[88] D. d'Humières, Multiple–relaxation–time lattice Boltzmann models in three dimensions, Proc. R. Soc. London, Ser. A **360**, 437 (2002).

[89] M. Klöwer, M. Razinger, J. J. Dominguez, P. D. Düben, and T. N. Palmer, Compressing atmospheric data into its real information content, Nat. Comput. Sci. **1**, 713 (2021).

[90] M. Klöwer, P. Düben, and T. Palmer, Number formats, error mitigation, and scope for 16-bit arithmetics in weather and climate modeling analyzed with a shallow water model, J. Adv. Model. Earth Syst. **12**, e2020MS002246 (2020).

[91] S. Hatfield, M. Chantry, P. Düben, and T. Palmer, Accelerating high-resolution weather models with deep-learning hardware, in *Proceedings of the Platform for Advanced Scientific Computing Conference* (ACM Press, Zurich, Switzerland, 2019), pp. 1–11.

[92] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems), in *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (IEEE, Tampa, Florida, USA, 2006), pp. 50–50.

[93] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham, Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems, Proc. R. Soc. A **476**, 20200110 (2020).

[94] X. He and L.-S. Luo, Lattice Boltzmann model for the incompressible Navier–Stokes equation, J. Stat. Phys. **88**, 927 (1997).

[95] T. Krüger, Unit conversion in LBM, in LBM Workshop. Dostupné z: http://lbmworkshop.com/wp-content/uploads/2011/08/2011-08-22_Edmonton_scaling.pdf (2011).

[96] G. I. Taylor and A. E. Green, Mechanism of the production of small eddies from large ones, Proc. R. Soc. London, Ser. A **158**, 499 (1937).

[97] T. v. Karman, Ueber den Mechanismus des Widerstandes, den ein bewegter Körper in einer Flüssigkeit erfährt, Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse, 509 (1911).

[98] A. Cattenone, S. Morganti, and F. Auricchio, Basis of the lattice Boltzmann method for additive manufacturing, Arch. Comput. Methods Eng. **27**, 1109 (2020).

[99] U. R. Alim, A. Entezari, and T. Moller, The lattice-Boltzmann method on optimal sampling lattices, IEEE Trans. Visualization Comput. Graphics **15**, 630 (2009).

[100] P. Neumann and T. Neckel, A dynamic mesh refinement technique for lattice Boltzmann simulations on octree-like grids, Comput. Mech. **51**, 237 (2013).

[101] U. Ghia, K. N. Ghia, and C. Shin, High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method, J. Comput. Phys. **48**, 387 (1982).

[102] B.-N. Jiang, T. Lin, and L. A. Povinelli, Large-scale computation of incompressible viscous flow by least-squares finite element method, Comput. Methods Appl. Mech. Eng. **114**, 213 (1994).

[103] J.-Y. Yang, S.-C. Yang, Y.-N. Chen, and C.-A. Hsu, Implicit weighted ENO schemes for the three-dimensional incompressible Navier–Stokes equations, J. Comput. Phys. **146**, 464 (1998).

[104] D. Barthes-Biesel, Motion and deformation of elastic capsules and vesicles in flow, Annu. Rev. Fluid Mech. **48**, 25 (2016).

[105] A. Guckenberger, M. P. Schraml, P. G. Chen, M. Leonetti, and S. Gekle, On the bending algorithms for soft objects in flows, Comput. Phys. Commun. **207**, 1 (2016).

[106] A. Guckenberger and S. Gekle, Theory and algorithms to compute Helfrich bending forces: A review, J. Phys.: Condens. Matter **29**, 203001 (2017).

[107] S. Williams, A. Waterman, and D. Patterson, Roofline: An insightful visual performance model for floating-point programs and multicore architectures (Lawrence Berkeley National Lab., Berkeley, CA, 2009).

[108] J. Latt, C. Coreixas, J. Beny, and A. Parmigiani, Efficient supersonic flow simulations using lattice Boltzmann methods based on numerical equilibria, Philos. Trans. R. Soc. A **378**, 20190559 (2020).

[109] N. Frapolli, S. S. Chikatamarla, and I. V. Karlin, Entropic lattice Boltzmann model for gas dynamics: Theory, boundary conditions, and implementation, Phys. Rev. E **93**, 063302 (2016).

[110] M. Atif, M. Namburi, and S. Ansumali, Higher-order lattice Boltzmann model for thermohydrodynamics, Phys. Rev. E **98**, 053311 (2018).

[111] J. L. Gustafson and I. T. Yonemoto, Beating floating point at its own game: posit arithmetic, Supercomputing Front. Innovations **4**, 71 (2017).

[112] NVIDIA Corporation, Parallel Thread Execution ISA Version 7.2 (2021).

[113] J. L. Gustafson, posit arithmetic, Mathematica Notebook describing the posit number system **30** (2017).

[114] F. De Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, posits: The good, the bad and the ugly, in *Proceedings of the Conference for Next Generation Arithmetic 2019, Singapore* (2019), pp. 1–10.

[115] C. Leong, SoftPosit library, accessed Sept, 24, 2019, https://gitlab.com/cerlane/SoftPosit.

[116] Z. Guo, C. Zheng, and B. Shi, Discrete lattice effects on the forcing term in the lattice Boltzmann method, Phys. Rev. E **65**, 046308 (2002).

[117] C. K. Batchelor and G. Batchelor, *An Introduction to Fluid Dynamics* (Cambridge University Press, Cambridge, UK, 2000).

[118] S. Izquierdo, P. Martínez-Lera, and N. Fueyo, Analysis of open boundary effects in unsteady lattice Boltzmann simulations, Comput. Math. Appl. **58**, 914 (2009).

[119] T. Krüger, Introduction to the immersed boundary method, in *LBM Workshop* (Edmonton, Canada, 2011).

[120] A. J. Ladd, Numerical simulations of particulate suspensions via a discretized Boltzmann equation. Part 1. Theoretical foundation, J. Fluid Mech. **271**, 285 (1994).

[121] D. Barthes-Biesel, J. Walter, and A.-V. Salsac, *Flow-Induced Deformation of Artificial Capsules* (CRC Press, Boca Raton, 2010), pp. 35–70.

[122] A. R. Harwood and A. J. Revell, Parallelisation of an interactive lattice-Boltzmann method on an Android-powered mobile device, Adv. Eng. Software **104**, 38 (2017).

[123] P. Neumann and M. Zellner, Lattice Boltzmann flow simulation on Android devices for interactive mobile-based learning, in *European Conference on Parallel Processing* (Springer, Grenoble, France, 2016), pp. 3–15.

[124] NVIDIA Corporation, NVIDIA Turing GPU Architecture (2018), https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[125] NVIDIA Corporation, NVIDIA A100 Tensor Core GPU Architecture (2020), https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[126] R. Gonzales, NVIDIA CUDA force P2 state - performance analysis (off vs. on) (2021), https://babeltechreviews.com/nvidia-cuda-force-p2-state/.

[127] J. Harting, S. Frijters, M. Ramaioli, M. Robinson, D. E. Wolf, and S. Luding, Recent advances in the simulation of particle-laden flows, Eur. Phys. J.: Spec. Top. **223**, 2253 (2014).

[128] T. Krüger, S. Frijters, F. Günther, B. Kaoui, and J. Harting, Numerical simulations of complex fluid-fluid interface dynamics, Eur. Phys. J.: Spec. Top. **222**, 177 (2013).

[129] T. Krüger, B. Kaoui, and J. Harting, Interplay of inertia and deformability on rheological properties of a suspension of capsules, J. Fluid Mech. **751**, 725 (2014).

[130] N. Rivas, S. Frijters, I. Pagonabarraga, and J. Harting, Mesoscopic electrohydrodynamic simulations of binary colloidal suspensions, J. Chem. Phys. **148**, 144101 (2018).