Designing a 3D Parallel Memory-Aware Lattice Boltzmann Algorithm on Manycore Systems

Yuankun Fu[1] and Fengguang Song[2(✉)]

[1] Purdue University, Indianapolis, IN, USA
fu121@purdue.edu
[2] Indiana University-Purdue University, Indianapolis, IN, USA
fgsong@cs.iupui.edu

Abstract. Lattice Boltzmann method (LBM) is a promising approach to solving Computational Fluid Dynamics (CFD) problems, however, its nature of memory-boundness limits nearly all LBM algorithms' performance on modern computer architectures. This paper introduces novel sequential and parallel 3D memory-aware LBM algorithms to optimize its memory access performance. The introduced new algorithms combine the features of single-copy distribution, single sweep, swap algorithm, prism traversal, and merging two temporal time steps. We also design a parallel methodology to guarantee thread safety and reduce synchronizations in the parallel LBM algorithm. At last, we evaluate their performances on three high-end manycore systems and demonstrate that our new 3D memory-aware LBM algorithms outperform the state-of-the-art Palabos software (which realizes the Fuse Swap Prism LBM solver) by up to 89%.

Keywords: Lattice Boltzmann method · Memory-aware algorithms · Parallel numerical methods · Manycore systems

1 Introduction

Computational Fluid Dynamics (CFD) simulations have revolutionized the design process in various scientific, engineering, industrial, and medical fields. The current Reynolds averaged Navier-Stokes (RANS) methods can solve steady viscous transonic and supersonic flows, but are not able to reliably predict turbulent separated flows [28]. Lattice Boltzmann method (LBM) is a young and evolving approach to solving these problems in the CFD community [2]. It originates from a mesoscale description of the fluid (based on the Boltzmann equation), and directly incorporates physical terms to represent complex physical phenomena, such as multi-phase flows, reactive and suspension flows, etc. Besides, many *collision models* have been developed for LBM to improve its stability to the second order of numerical accuracy when simulating high Reynolds number flows [2].

However, it is challenging to achieve high performance for LBM algorithms, since LBM has large data storage costs and is highly memory-bound on current architectures [24]. Driven by our prior work [5] to merge multiple collision-streaming cycles (or time steps) in 2D, this study aims to augment the memory-awareness idea to support parallel 3D LBM to optimize data re-utilization. Although it might seem to be straightforward to move from the 2D space to 3D space, it is significantly much more difficult to design an efficient 3D memory-aware LBM algorithm. In this paper, we target solving the following three main challenges. (1) As geometries change from 2D to 3D, the required data storage increases from $O(N^2)$ to $O(N^3)$, meanwhile data dependencies of the lattice model becomes much more complicated. There exist single-copy distribution methods to reduce data storage cost by half, but they require following a particular traversal order. Can we combine the best single-copy distribution method with our idea of merging multiple collision-streaming cycles to design a 3D memory-aware LBM with higher performance? (2) If the combination is possible, since normal 3D tiling [21] does not apply to this case, how to additionally explore the spatial locality? (3) When designing the parallel 3D memory-aware LBM, a non-trivial interaction occurs at the boundaries between threads, how to guarantee thread safety and avoid race conditions? Although some existing works use wavefront parallelism to explore the temporal locality, they insert frequent layer-wise synchronizations among threads every time step [11,27]. In this paper, we aim to reduce the synchronization cost among parallel threads.

To the best of our knowledge, this paper makes the following contributions. First, we design both sequential and parallel 3D memory-aware LBM algorithms that combine five features: single-copy distribution, loop fusion (single sweep), swap algorithm, prism traversal, and merging two collision-streaming cycles. Second, we present a parallelization method to keep the thread safety on the intersection layers among threads and reduce the synchronization cost in parallel. At last, two groups of experiments are conducted on three different manycore architectures, followed by performance analysis. The first group of sequential experiments (i.e., using a single CPU core) shows that our memory-aware LBM outperforms the state-of-the-art Palabos (Fuse Swap Prism LBM solver) [17] by up to 19% on a Haswell CPU and 15% on a Skylake CPU. The second group evaluates the performance of parallel algorithms. The experimental results show that our parallel 3D memory-aware LBM outperforms Palabos by up to 89% on a Haswell node with 28 cores, 85% on a Skylake node with 48 cores, and 39% on a Knight Landing node with 68 cores.

2   Related Work

Existing research on designing efficient LBM algorithms mainly focuses on optimizing memory accesses within one time step of LBM due to its iterative nature. For instance, a few LBM algorithms (e.g., swap [13,25], AA [1], shift [19], and esoteric twist [6], etc.) retain a single copy of the particle distribution data (i.e., "single-copy distribution"), and optimize the memory access pattern in the LBM streaming kernel, but each of the algorithms needs to follow a set of constraints

(e.g., swap requires predefined order of discrete cell velocities [9], AA requires distinguishing between even and odd time steps, shift requires extra storage [9], esoteric twist requires only one version of the LB kernel [29], etc.) [26] uses a moment-based representation with extra distribution pseudo domain to further reduce the storage cost. Some works hide the inter-process communication cost on multicore accelerators [3], and achieve large-scale parallelization on HPC systems [20] and GPU [1]. [31] introduces a cache oblivious blocking 3D LBM algorithm, but it has an irregular parallelism scheme due to its recursive algorithm design. In summary, the above methods focus on optimizations within one time step. Differently, our 3D memory-aware LBM aims to adopt the efficient single-copy distribution scheme, and design new methodologies to merge two collision-streaming cycles to explore both temporal and spatial data locality at the same time for achieving higher performance.

Another category of works manages to accelerate LBM by wavefront parallelism, which generally groups many threads to successively compute on the same spatial domain. [11] presents a shared-memory wavefront 2D LBM together with loop fusion, loop bump, loop skewing, loop tiling, and semaphore operations. But due to its high synchronization cost incurred by many implicit barriers in wavefront parallelism, their parallel performance has only 10% of speedup on average. [7] presents a shared-memory wavefront 3D LBM with two-copy distributions, and does not use spatial locality techniques such as loop fusion and loop blocking. [27] presents a shared-memory wavefront 3D Jacobi approach together with spatial blocking. It uses two-copy distributions and has simpler 6-neighbors dependencies (rather than the 19 or 27 neighbors in 3D LBM). [12] combines the wavefront parallelism with diamond tiling. By contrast, our 3D memory-aware LBM does not use the wavefront parallelism, but judiciously contains three light-weight synchronization barriers every two collision-streaming cycles. In addition, we partition the simulation domain and assign a local sub-domain to every thread, rather than all threads work on the same sub-domain in wavefront parallelism. In each sub-domain, each thread in our algorithm computes multiple time steps at once, rather than one thread computes one time step at a time in wavefront parallelism. In addition, each of our threads also utilizes prism techniques to optimize spatial locality. This strategy in particular favors new manycore architectures, which tend to have increasingly larger cache sizes.

Modern parallel software packages that support LBM can be classified into two categories based upon their underlying data structures. One category adopts matrix-based memory alignment at the cell level (e.g., Palabos [10], OpenLB [8], HemeLB [14], HemoCell [30]). Since neighbors can be easily found through simple index arithmetics in this case, they are more suitable for simulations with dense geometries. The other category adopts adjacent list data structures (e.g., Musubi [15], waLBerla [4], HARVEY [20]). They are often used for simulating domains with sparse and irregular geometries, but their cells require additional memory of pointers, and double the memory consumption in the worst case. In this study, we choose the widely-used and efficient matrix-based data structure in the LBM community, and select the state-of-the-art Palabos library as the baseline, since Palabos provides a broad modeling framework, supports applications with complex physics, and shows high computational performance.

[18] designs a locally recursive non-locally asynchronous (LRnLA) conefold LBM algorithm, which uses recursive Z-curve arrays for data storage, and recursively subdivides the space-time dependency graph into polytopes to update lattice nodes. However, our work uses a more directly accessible matrix-based data storage and has a regular memory access pattern. Besides, our prism traversal can independently or integrate with merging two time steps to operate on the lattice nodes, while [18] operates on the dependency graph.

3   Baseline 3D LBM Algorithm

The baseline 3D LBM algorithm in this paper is called *Fuse Swap LBM* as shown in Algorithm 1, which involves three features: single-copy distribution, swap algorithm, and loop fusion. We choose the swap algorithm [13] since it is relatively simpler than the other single-copy distribution methods, and is more efficient to use simple index arithmetic to access neighbors in the matrix-based memory organization. The swap algorithm replaces the copy operations between a cell and its neighbors in the streaming kernel by a value swap, thereby it is in-place and does not require the second copy. But when combining it with loop fusion, we must guarantee that the populations of neighbors involved in the swap are already in a post-collision state to keep thread safety [9].

The work-around solution is to adjust the traversal order of simulation domains with a predefined order of discrete cell velocities [9]. Thus each cell can stream its post-collision data by swapping values with half of its neighbors pointed by the "red" arrows (1–9 directions for D3Q19 in Fig. 1a), if those neighbors are already in post-collision and have "reverted" their distributions. We define this operation as "*swap_stream*". The "*revert*" operation in Fig. 1b lets a cell locally swap its post-collision distributions to opposite directions. To make the Fuse Swap LBM more efficient, Palabos pre-processes and post-processes the boundary cells on the bounding box at line 2 and 7, respectively, so that it can remove the boundary checking operation in the inner bulk domain. Thus Algorithm 1 is divided into three stages in every time step as follows.
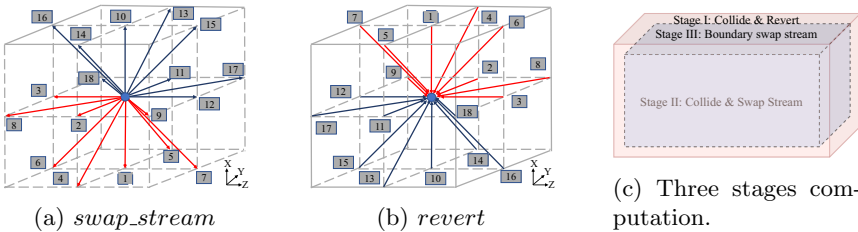

(a) *swap_stream*   (b) *revert*   (c) Three stages computation.

Fig. 1. Two operations and three stages computation used in sequential 3D fuse swap LBM. (Color figure online)

Algorithm 1. 3D Fuse Swap LBM

1: **for** iT = 0; iT < N; ++iT **do**
2:   Stage I: *collide* and *revert* on the bounding box, i.e., 6 surfaces of cuboid $(1,1,1)$ to (lx, ly, lz)
     // Stage II: bulk domain computation
3:   **for** iX = 2; iX $\leq lx - 1$; ++iX **do**
4:     **for** iY = 2; iY $\leq ly - 1$; ++iY **do**
5:       **for** iZ = 2; iZ $\leq lz - 1$; ++iZ **do**
6:         *collide* & *swap_stream* on (iX, iY, iZ) to half of its neighbors
7:   Stage III: *boundary_swap_stream* on the bounding box

---

4 The 3D Memory-Aware LBM Algorithm

4.1 Sequential 3D Memory-Aware LBM

We design and develop the sequential 3D memory-aware LBM (shown in Algorithm 2), based on the latest efficient Fuse Swap LBM, by adding two more features: merging two collision-streaming cycles to explore the temporal locality, and introducing the prism traversal to explore the spatial locality. Figure 2 shows an example on how to merge two collision-streaming cycles given a $4 \times 4 \times 4$ cube:

1. Figure 2a shows the initial state of all cells at the current time step t. Green cells are on boundaries, and blue cells are located in the inner bulk domain.
2. In Fig. 2b, we compute the first *collide*, *revert* and *boundary_swap_stream* row by row on the bottom layer iX = 1. After a cell completes the first computation, we change it to orange.
3. In Fig. 2c, we compute the first *collide* and *boundary_swap_stream* row by row till cell (2,2,1) on the second layer iX = 2.
4. In Fig. 2d, cell (2,2,2) completes its first *collide* and *swap_stream*, so we change it to red since they are inner cells. Then we observe that cell (1,1,1) is ready for the second *collide*, so we change it to yellow.
5. In Fig. 2e, we execute the second *collide* and *boundary_swap_stream* on cell (1,1,1), and change it to purple.

(a) Initialization. (b) First computation on layer iX=1. (c) First computation on layer iX=2. (d) First computation on cell (2,2,2). (e) Second computation on cell (1,1,1). (f) Legends.
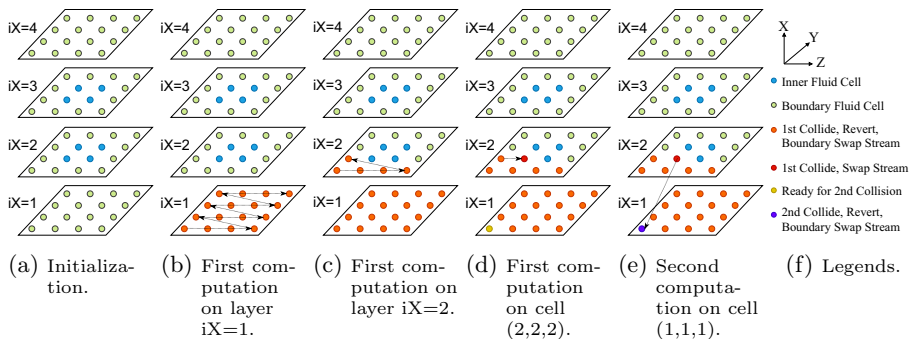
Fig. 2. 3D sequential two-step memory-aware LBM on a $4 \times 4 \times 4$ cube lattice.

To further increase data reuse, we optimize the algorithm's spatial locality by designing a "prism traversal" method, since the shape of this traversal constructs a 3D pyramid prism or a parallelpiped prism. We use an example to explain its access pattern in a $4 \times 16 \times 16$ cuboid with stride $tile = 4$. Figure 3a–d are the four separate 16×16 layers of the cuboid from bottom to top. The cells with the same number on the four layers construct a $prism$ (e.g., the cells with number 1 in Fig. 3a–d construct a pyramid-shape "Prism 1"). In each prism, we still firstly go along Z-axis, then along Y-axis, and upward along X-axis at last. Then we traverse prism-wise from Prism 1 to Prism 30. Finally, if a cuboid is much larger than this example, the majority of prisms are "parallelpiped" shapes like Prism 9 and 10 in Fig. 3e. The reason why the planar slice of a prism is either triangles or parallelograms is due to the $swap\_stream$ operation. When cutting Fig. 1a ($swap\_stream$) along the Y-Z plane, we have a planar slice as shown in Fig. 3f. We observe that a cell (star) swaps with its lower right neighbor (orange) at direction 9. In other words, when the orange cell swaps with the upward row, its neighbor "shifts" one cell $leftward$. Similarly, if cutting Fig. 1a ($swap\_stream$) along the X-Y plane, when a cell swaps data with the upward row, its neighbor "shifts" one cell $forward$. Thus when we traverse $tile$ number of cells on Z-axis at row iY, they can swap with $tile$ number of cells but shifted one cell leftward at row $iY+1$, thereby we get parallelograms in Fig. 3a–d. When the shift encounters domain boundaries, we truncate the parallelograms and get isosceles right triangles or part of parallelograms. At last, we can safely combine "prism traversal" with merging two collision-streaming cycles, since the cell at left forward down corner has been in a post-collision state and ready to compute the second computation when following the above traversal order.

Algorithm 2 presents the sequential 3D memory-aware LBM. Lines 6–10 traverse the domain prism-wise with stride $tile$. Lines 11–14 merge two time steps computation. The first $stream$ starting from the bottom layer iX = 1 in Line 10
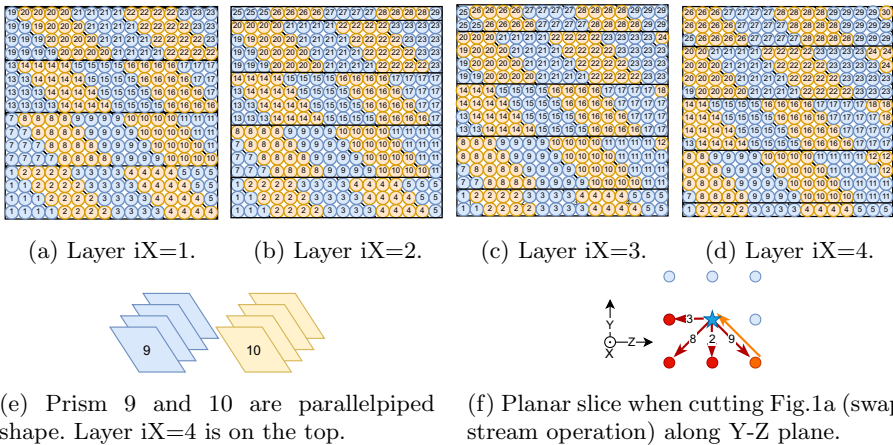

(a) Layer iX=1.   (b) Layer iX=2.   (c) Layer iX=3.   (d) Layer iX=4.

(e) Prism 9 and 10 are parallelpiped shape. Layer iX=4 is on the top.

(f) Planar slice when cutting Fig.1a (swap stream operation) along Y-Z plane.

Fig. 3. Sequential 3D prism traversal on a $4 \times 16 \times 16$ cuboid box.

Algorithm 2. 3D Sequential Memory-aware LBM

1: tile := stride of the prism traversal
2: **for** iT = 0; iT < N; iT += 2 **do**
3:   **for** outerX = 1; outerX ≤ lx; outerX += tile **do**
4:    **for** outerY = 1; outerY ≤ ly + tile - 1; outerY += tile **do**
5:     **for** outerZ = 1; outerZ ≤ lz + 2* (tile - 1); outerZ += tile **do**
6:      **for** innerX=outerX; innerX ≤ MIN(outerX+tile-1, lx); ++innerX, ++dx **do**
7:       minY = outerY - dx; maxY = minY + tile - 1; dy = 0; /* forward shift */
8:       **for** innerY=MAX(minY, 1); innerY ≤ MIN(maxY, ly); ++innerY, ++dy **do**
9:        minZ = outerZ - dx - dy; maxZ = minZ + tile - 1; /* leftward shift */
10:        **for** innerZ=MAX(minZ, 1); innerZ ≤ MIN(maxZ, lz); ++innerZ **do**
         /* (1) First computation at time step t. */
11:         *adaptive_collide_stream*(innerX, innerY, innerZ);
         /* (2) Second computation at time step $t + 1$. */
12:         **if** innerX > 1 && innerY > 1 && innerZ > 1 **then**
13:          *adaptive_collide_stream*(innerX-1, innerY-1, innerZ-1);
         /* (3) Second computation of neighbors at certain locations. */
14:         *boundary_neighbor_handler*(innerX, innerY, innerZ);
15:   Second *collide*, *revert* & *boundary_swap_stream* on the top layer iX = lx.
16: **function** *boundary_cell_comp*(iX, iY, iZ)
17: *collide*, *revert*, & *boundary_swap_stream* on (iX, iY, iZ) to half of its neighbors;
18: **function** *adaptive_collide_stream*(iX, iY, iZ)
19: **if** (iX, iY, iZ) is on the boundary **then**
20:   *boundary_cell_comp*(iX, iY, iZ);
21: **else**
22:   *collide* & *swap_stream* on (iX, iY, iZ) to half of its neighbors;
23: **function** *boundary_neighbor_handler*(iX, iY, iZ)
  // Handle the second computation of (iX, iY, iZ)'s neighbors at certain locations.
24: **if** iZ == lz **then** // (iX, iY, iZ) is the last cell of a row.
25:   *boundary_cell_comp* (iX-1, iY-1, iZ);
26: **if** iY == ly && iZ > 1 **then** // (iX, iY, iZ) is in the last row of a layer.
27:   *boundary_cell_comp*(iX-1, iY, iZ-1);
28: **if** iY == ly && iZ == lz **then** // (iX, iY, iZ) is the last cell on a layer.
29:   *boundary_cell_comp*(iX-1, iY, iZ);

---

is necessary due to the data dependency for the second computation. In particular, the if-statement in Line 13 ensures that the cell to compute at time step $t+1$ is in a post-collision state, no matter using D3Q15, D3Q19, D3Q27 or extended lattice models. For simplicity, Lines 16–29 define three helper functions.

4.2 Parallel 3D Memory-Aware LBM

To support manycore systems, we choose OpenMP [16] to realize the parallel 3D memory-aware LBM algorithm[1]. Figure 4 illustrates its idea on a $8 \times 4 \times 4$ cuboid, which is evenly partitioned by two threads along the X-axis (*height*). Then each thread traverses a $4 \times 4 \times 4$ sub-domain with prism stride *tile* = 4. Line 4 in Algorithm 3 defines the start and end layer index of each thread's sub-domain, thus the end layers *myEndX* are "*intersections*" (e.g., layer 4 and 8). Figure 4a shows the initial state at time step t. In addition, the parallel 3D memory-aware Algorithm 3 consists of three stages: Preprocessing, Sub-domain computation, and Post-processing.

---

[1] [23] states that when the minimum effective task granularity (METG) of parallel runtime systems are smaller than task granularity of large-scale LBM simulations, all of these runtime system can deliver good parallel performance.
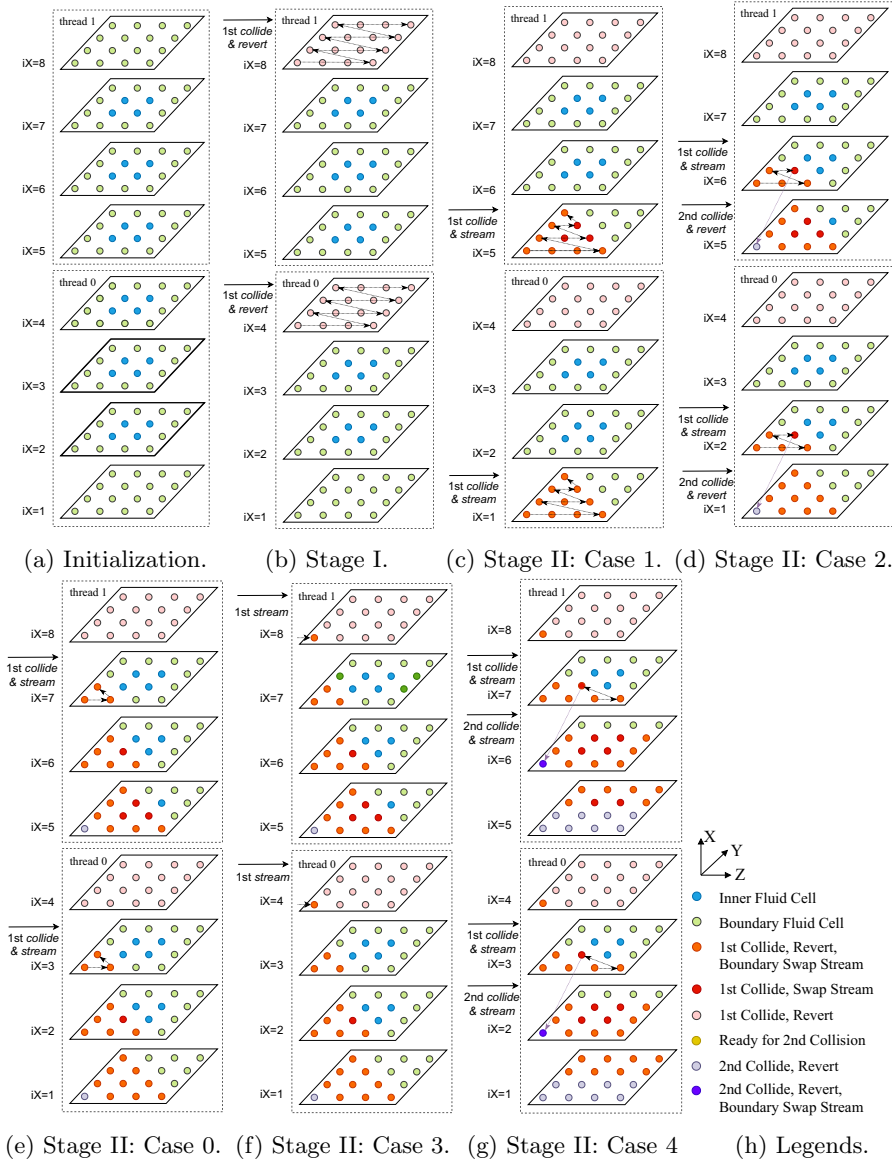
(a) Initialization. (b) Stage I. (c) Stage II: Case 1. (d) Stage II: Case 2.

(e) Stage II: Case 0. (f) Stage II: Case 3. (g) Stage II: Case 4   (h) Legends.

Fig. 4. Parallel 3D two-step memory-aware LBM on a $8 \times 4 \times 4$ cuboid.

1. **Stage I (Preprocessing)** *line 5 in Algorithm 3*: In Fig. 4b, thread 0 and 1 compute the first *collide* and *revert* on the "intersection" layers 4 and 8, respectively, and then change them to pink.

2. **Stage II (Sub-domain computation)** handles five cases from step 2 to 7. In *case 0* (*lines 15–17 in Algorithm 3*), when thread 0 and 1 access the cells

---

Algorithm 3. Parallel 3D Memory-aware LBM

1: **for** iT = 0; iT < N; iT += 2 **do**
2:   **#pragma omp parallel default(shared){**
3:    $sub\_h = lx/nthreads$; // height of each thread's sub-domain
4:    myStartX = 1 + $thread\_id \times sub\_h$; myEndX = $(thread\_id + 1) \times sub\_h$;
     /* Stage I: First collide & revert on the intersection layer.*/
5:    collide & revert on all $ly \times lz$ cells on layer iX = myEndX;
6:   **#pragma omp barrier**
     /* Stage II: Main computation in each thread's sub-domain.*/
7:   **for** outerX = myStartX; outerX ≤ myEndX; outerX += tile **do**
8:    **for** outerY = 1; outerY ≤ ly + tile - 1 ; outerY += tile **do**
9:     **for** outerZ = 1; outerZ ≤ lz + 2 * (tile - 1); outerZ += tile **do**
10:     **for** innerX=outerX; innerX≤MIN(outerX+tile-1, myEndX); ++innerX, ++dx **do**
11:      minY = outerY - dx; maxY = minY + tile - 1; dy = 0; /* forward shift */
12:      **for** innerY=MAX(minY, 1); innerY≤MIN(maxY, ly); ++innerY, ++dy **do**
13:       minZ = outerZ - dx - dy; maxZ = minZ + tile - 1; /* leftward shift */
14:       **for** innerZ = MAX(minZ, 1); innerZ ≤ MIN(maxZ, lz); ++innerZ **do**
     // Case 0: First collide & stream on the first row and column of each layer except the intersection layers.
15:        **if** innerX != myEndX && (innerX == 1 or innerY == 1 or innerZ == 1) **then**
16:         First $boundary\_cell\_comp(innerX, innerY, innerZ)$;
17:         **continue**;
     // Case 1: First collide & stream on layer myStartX:
18:        **if** innerX == myStartX **then**
19:         First $adaptive\_collide\_stream$(innerX, innerY, innerZ);
     // Case 2: First collide & stream on myStartX + 1; Second collide & revert on myStartX:
20:        **else if** innerX == myStartX + 1 **then**
21:         First $adaptive\_collide\_stream$(innerX, innerY, innerZ);
22:         Second collide & revert on (innerX-1, innerY-1, innerZ-1);
23:         Handle the second collide & revert of neighbors at certain boundary locations;
     // Case 3: First stream on layer myEndX; Second collide & stream under one layer:
24:        **else if** innerX == myEndX **then**
25:         First $adaptive\_stream$(innerX, innerY, innerZ);
26:         Second $adaptive\_collide\_stream$(innerX-1, innerY-1, innerZ-1);
27:         $boundary\_neighbor\_handler$ (innerX, innerY, innerZ);
     // Case 4: first collide & stream on other layers; Second collide & stream under one layer:
28:        **else**
29:         First $adaptive\_collide\_stream$(innerX, innerY, innerZ);
30:         Second $adaptive\_collide\_stream$(innerX-1, innerY-1, innerZ-1);
31:         $boundary\_neighbor\_handler$(innerX, innerY, innerZ);
32:   **#pragma omp barrier**
     /* Stage III: second collide & stream on the intersection; then second stream on the layer myStartX. */
33:    $adaptive\_collide\_stream$ at all $ly \times lz$ cells on layer iX = myEndX;
34:   **#pragma omp barrier**
35:    stream at all $ly \times lz$ cells on layer iX = myStartX;
36:   **}**

---

on the first row and column of each layer except the "intersection" layers, we execute the first *boundary_cell_comp* on them and change them to orange.

3. Figure 4c shows *case 1* (*lines 18–19 in Algorithm 3*). When thread 0 and 1 access the cells on layer $myStartX$ (iX = 1 & 5), respectively, we execute the *adaptive_collide_stream* on them to compute at time step t, and then change the boundary cells to orange and the inner cells to red.

4. Figure 4d shows *case 2* (*lines 20–23 in Algorithm 3*). When thread 0 and 1 are on layer $myStartX + 1$ (iX = 2 & 6), respectively, we execute the first *adaptive_collide_stream* at time step t and change boundary cells to orange and inner cells to red. Meanwhile, cell (5,1,1) and (1,1,1) have collected the
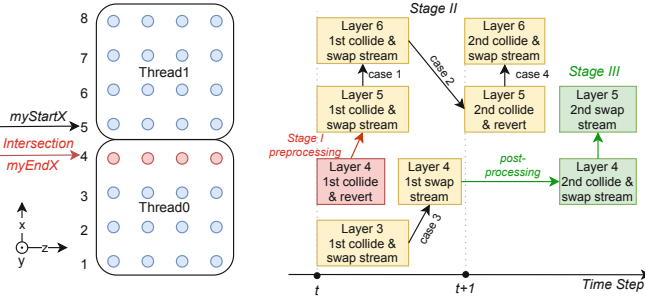
Fig. 5. Handle thread safety on intersection layers.

data dependencies to *collide* at time step $t + 1$, we execute the second *collide* and *revert* but without *stream* on them, and change to light purple.

5. Figure 4e shows that when continuing traversal in Prism 1, thread 0 and 1 are on layer iX = 3 & 6. Since the cells traversed in this figure are in the first row and column, case 0 is used here, otherwise, case 4 is used.

6. Figure 4f shows *case 3* (*lines 24–27 in Algorithm* 3). When thread 0 and 1 are on the intersection layers (iX = 4 & 8), we execute the remaining first *stream* at time step t due to preprocessing in Stage I. Then if cells under one layer (iX = 3 & 7) collect their data dependency at time step $t + 1$, we execute the second *adaptive_collide_stream* on them.

7. Figure 4g shows *case 4* (*lines 28–31 in Algorithm* 3). When thread 0 and 1 are on the other layers of sub-domain, we conduct the first *adaptive_collide_stream* on (innerX, innerY, innerZ) at time step t, and then the second *adaptive_collide_stream* on (innerX-1, innerY-1, innerZ-1) at time step $t + 1$. Then we call *boundary_neighbor_handler* to compute the neighbors of (innerX, innerY, innerZ) at certain locations at time step $t + 1$.

8. **Stage III (Post-processing)** *lines 33–35 in Algorithm* 3: Firstly, since Stage I and case 3 have completed the first computation on intersection layers, we wrap up the second *collide* and *stream* on intersections. Secondly, since case 2 have executed the second *collide* and *revert* on the first layers *myStartX* of each sub-domain, the second *stream* remains to be executed.

How to Handle Thread Safety near Intersection Layers: We aim to keep thread safety and minimize the synchronization cost during parallel executions. To this end, we need to carefully design the initial state of each thread so that the majority of computation stays in each threads' local sub-domain. The left part of Fig. 5 shows the view of Fig. 4 along X-Z axis, and layer 4 is the intersection layer that partitions two threads' sub-domains. The right part shows the data dependencies near the intersection layer in two time steps. In the figure, the red block represents Stage I of Algorithm 3, yellow blocks Stage II, and green blocks Stage III. The arrows indicate that data are transferred from layer A to B by using a procedure (or B depends on A). There are three non-trivial

dependencies requiring to handle thread safety near intersection layers. (1) Since the swap algorithm only streams data to half of the neighbors under one layer, the *swap_stream* on layer 5—the first layer of thread 1's sub-domain—should be delayed after the *revert* on layer 4 in thread 0's sub-domain. Thus, in Stage I, we pre-process *collide* and *revert* at time step t but without *stream* on layer 4, since *stream* on layer 4 depends on the post-collision on layer 3, which has not been computed yet. (2) In Stage II, the second *swap_stream* on layer 6 called by the case 4 procedure should be delayed after the second *revert* but without *swap_stream* on layer 5. This is because thread 1 cannot guarantee that thread 0 has completed the second *swap_steam* on layer 4. To keep thread safety, *swap_stream* on layer 5 is delayed to Stage III. (3) Thus, in Stage III, the second *swap_stream* on layer 5 is delayed after the second *swap_stream* on layer 4. Above all, since the major computation happens in Stage II of each thread's sub-domain, we avoid the frequent "layer-wise" thread synchronizations that occur in the wave-front parallelism. Besides, we only synchronize at the intersection layers every two time steps, hence the overhead of three *barriers* of Algorithm 3 becomes much less.

5   Experimental Evaluation

In this section, we first present the experimental setup and validations on our 3D memory-aware LBM. Then we evaluate its sequential and parallel performance.

5.1   Experiment Setup and Verification

The details of our experimental hardware platforms are provided in Table 1. To evaluate the performance of our new algorithms, we use the 3D lid-driven cavity flow simulation as an example. The 3D cavity has a dimension of $lz \times ly \times lx$, and its top lid moves with a constant velocity v. Our 3D memory-aware LBM algorithms have been implemented as C++ template functions, which are then added to the Palabos framework. For verification, we construct a *cavity* with the same procedure, and then separately execute four algorithms on it, i.e., Palabos solvers $fuse()$ and $fuse\_prism()$ for N time steps, and our memory-aware algorithms $two\_step\_prism()$ and $two\_step\_prism\_omp()$ for $N/2$ time steps. Then, we compute the velocity norm of each cell and write to four separate logs. At last, we verify that our algorithms produce the same result as Palabos for guaranteeing software correctness.

5.2   Performance of Sequential 3D Memory-Aware LBM

The first set of experiments with 3D cavity flows compare the sequential performance of four different LBM algorithms, which are the Fuse Swap LBM (with/without prism traversal), and the Two-step Memory-aware LBM (with/without prism traversal). For simplicity, we use the abbreviations of fuse LBM, fuse prism LBM, 2-step LBM and 2-step prism LBM, respectively. The problem input are

Table 1. Details of our experimental platforms.

Microarchitecture	Bridges at PSC	Stampede2 at TACC	
	Haswell'14	Skylake'17	Knight Landing'16
Intel CPU product code	Xeon E5-2695v3	Xeon Platinum 8160	Xeon Phi 7250
Total # Cores/node	28 on 2 sockets	48 on 2 sockets	68 on 1 socket
Clock rate (GHz)	2.1–3.3	2.1 nominal (1.4–3.7)	1.4
L1 cache/core	32 KB	32 KB	32 KB
L2 cache/core	256 KB	1 MB	1 MB per 2-core tile
L3 cache/socket	35 MB	33 MB (Non-inclusive)	16 GB MCDRAM
DDR4 Memory (GB)/node	128 (2133 MHz)	192 (2166 MHz)	96 (2166 MHz)
Compiler	icc/19.5	icc/18.0.2	
AVX extension	AVX2	AVX512	

3D cubes with edge size $L = 64$–896. Every algorithm with a prism stride configuration is executed five times, and the average MFLUPS (millions of fluid lattice node updates per second) is calculated. For the "prism" algorithms, different prism strides (ranging from 8, 16, 32,..., to 448) are tested, and we select the best performance achieved.
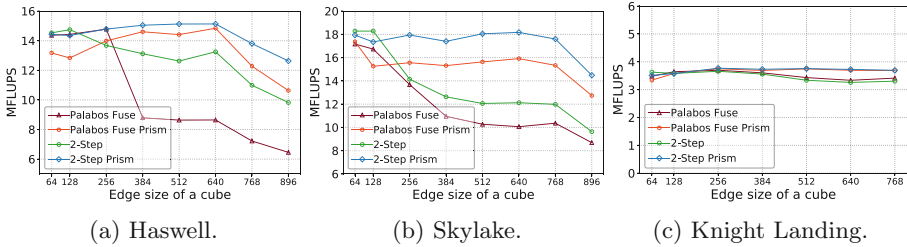

Fig. 6. Sequential performance using four LBM algorithms on three types of CPUs.

Figure 6 shows the sequential performance on three types of CPUs. When we use small edge sizes (e.g., $L = 64, 128$), 2-step LBM is the fastest. But when $L \geq 256$, 2-step prism LBM performs the best and is up to 18.8% and 15.5% faster than the second-fastest Palabos (Fuse Prism LBM solver) on Haswell and Skylake, respectively. But since KNL does not have an L3 cache, 2-step prism LBM is only 1.15% faster than Palabos (Fuse Prism LBM solver).

We observe that the performance of algorithms without prism traversal starts to drop when $L \geq 384$. Since the swap algorithm streams to half of its neighbors on its own layer and the layer below, $23.9 \, \text{MB}/layer \times 2 \, layers = 47.8 \, \text{MB}$ (when $L = 384$), which exceeds the L3 cache size (35 MB per socket on Haswell). Thus we need to use spatial locality by adding the feature of prism traversal. Consequently, on Haswell and Skylake, fuse LBM is improved by up to 71.7%

and 58.2%, respectively, 2-step LBM is improved by up to 28.6% and 50.4%, respectively. When only adding the feature of merging two steps, 2-step LBM is faster than Palabos (Fuse) by up to 53.3% on Haswell and 20.5% on Skylake. Hence, we conclude that both prism traversal and merging two steps significantly increase cache reuse on the large domain.
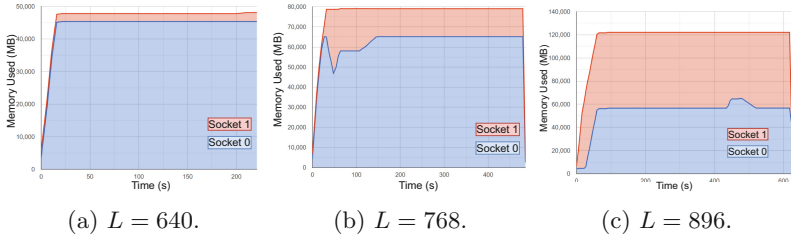

(a) $L = 640$.　　　　　(b) $L = 768$.　　　　　(c) $L = 896$.

Fig. 7. Memory usage on two sockets of a Haswell node.

In Fig. 6, we observe that the performance of all algorithms starts to drop when $L \geq 768$ on Haswell and $L = 896$ on Skylake. To find out the reason, we use Remora [22] to monitor the memory usage on each socket of the Haswell node. As L increases from 640 to 896, the memory usage on socket 1 (red area) in Fig. 7a–c has enlarged from 2.4 GB to 63.9 GB. When memory usage exceeds the 64 GB DRAM capacity per socket on the Haswell node, foreign NUMA memory accesses are involved, thus the sequential performance reduces. Similar results also happen on the Skylake node. However, because the KNL node only has one socket, the performance on KNL does not drop.

5.3　Performance of Parallel 3D Memory-Aware LBM

Given N cores, Palabos LBM solvers partition the simulation domain evenly along three axes by $N_z \times N_y \times N_x = N$ MPI processes, which follows the underlying memory layout of cells along the axis of Z, then Y, and X at last. But our 3D memory-aware LBM partitions a domain only along X-axis by N OpenMP threads. Hence, Palabos LBM solvers have a smaller Y-Z layer size per core than our algorithm and have closer memory page alignment especially for a large domain. To exclude the factor caused by different partition methods, when the input of Palabos LBM solvers still uses cubes, 3D memory-aware LBM will take two different inputs. Firstly, it takes the input of the "equivalent dimension" of those cubes, such that a thread in our algorithm and a process in Palabos will compute a sub-domain with the same dimension after the respective partition method. Secondly, it simply takes the identical input of those cubes.

Figure 8 shows the strong scalability of three LBM algorithms on three types of compute nodes. The input of Palabos LBM solvers use cubes with edge size L from small to large. Table 2 gives an example of the equivalent input used by 3D memory-aware LBM when Palabos LBM solvers use a cube with $L = 840$

Table 2. Equivalent input used by 2-step prism LBM when the input of Palabos LBM solvers is a cube with $L = 840$ on a Haswell node.

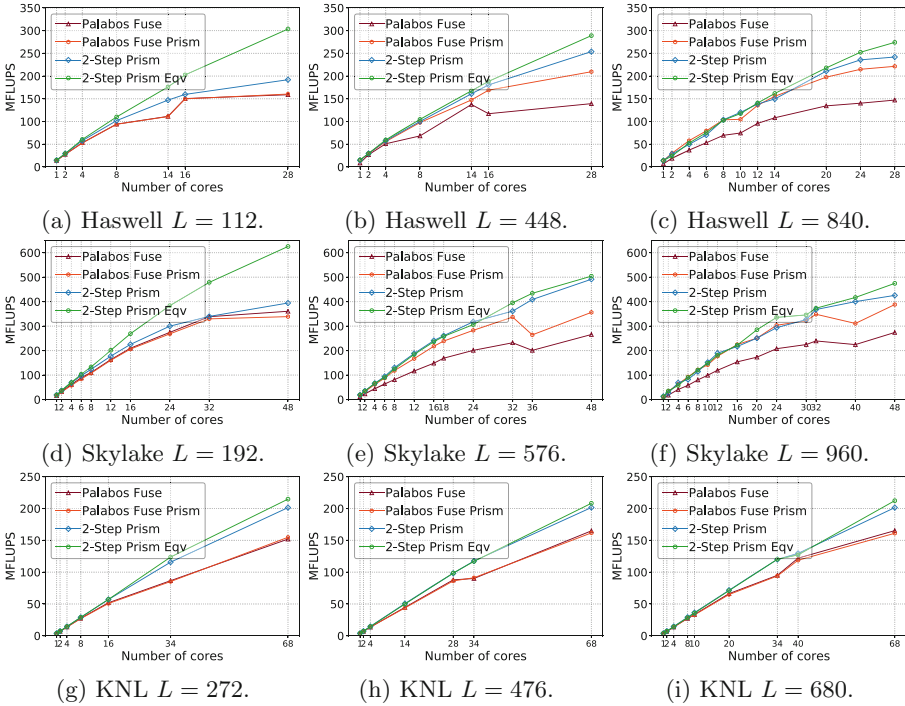| Cores	1	2	4	6	8	10	12	14	20	24	28
lx (height)	840	1680	3360	5040	3360	8400	5040	11760	8400	10080	11760
ly (width)	840	840	420	420	420	420	420	420	420	420	420
lz (length)	840	420	420	280	420	168	280	120	168	140	120

Fig. 8. Strong scalability performance on three types of compute nodes. "2-step prism eqv" = Parallel 3D memory aware LBM takes the equivalent input of cubes.

on a Haswell node. We observe that the 2-step prism LBM scales efficiently and always achieves the best performance in all cases. (1) When using the equivalent input of cubes, for small scale cubes (with $L = 112, 192, 272$) in Fig. 8a, d and g, 3D memory-aware LBM (green legend) is faster than the second-fastest Palabos (Fuse Prism) (orange legend) by up to 89.2%, 84.6%, and 38.8% on the Haswell, Skylake, and KNL node, respectively. Missing L3 cache on KNL prevents the similar speedup as other two CPUs. In Fig. 8b, e and h, for the middle scale cubes (with $L = 448, 576, 476$), it is still faster than Palabos (Fuse Prism) by up to 37.9%, 64.2%, and 28.8% on three CPU nodes, respectively. Due to unbalanced number of processes assigned on three axes, we observe that the performance of

Palabos Fuse and Fuse Prism drop on some number of cores. In Fig. 8c, f and i, for the large scale cubes (with $L = 840, 960, 680$), it is still faster than Palabos (Fuse Prism) by up to 34.2%, 34.2%, and 31.8%, respectively. (2) When using the identical input of cubes, although our 3D memory-aware LBM has larger Y-Z layer sizes, it is still faster than Palabos (Fuse Prism) but with less speedup than before, i.e., by up to 21.1%, 54.7%, and 30.1% on three CPU nodes, respectively. The less speedup suggests our future work to partition a 3D domain along three axes to utilize closer memory page alignment on smaller Y-Z layer size.

6   Conclusion

To address the memory-bound limitation of LBM, we design a new 3D parallel memory-aware LBM algorithm that systematically combines single copy distribution, single sweep, swap algorithm, prism traversal, and merging two collision-streaming cycles. We also keep thread safety and reduce the synchronization cost in parallel. The parallel 3D memory-aware LBM outperforms state-of-the-art LBM software by up to 89.2% on a Haswell node, 84.6% on a Skylake node and 38.8% on a Knight Landing node, respectively. Our future work is to merge more time steps on distributed memory systems and on GPU.

References

1. Bailey, P., Myre, J., Walsh, S.D., Lilja, D.J., Saar, M.O.: Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In: 2009 International Conference on Parallel Processing, pp. 550–557. IEEE (2009)
2. Coreixas, C., Chopard, B., Latt, J.: Comprehensive comparison of collision models in the lattice Boltzmann framework: theoretical investigations. Phys. Rev. E **100**(3), 033305 (2019)
3. Crimi, G., Mantovani, F., Pivanti, M., Schifano, S.F., Tripiccione, R.: Early experience on porting and running a Lattice Boltzmann code on the Xeon-Phi coprocessor. Procedia Comput. Sci. **18**, 551–560 (2013)
4. Feichtinger, C., Donath, S., Köstler, H., Götz, J., Rüde, U.: WaLBerla: HPC software design for computational engineering simulations. J. Comput. Sci. **2**(2), 105–112 (2011)
5. Fu, Y., Li, F., Song, F., Zhu, L.: Designing a parallel memory-aware lattice Boltzmann algorithm on manycore systems. In: 30th International Symposium on Computer Architecture and High Performance Computing, pp. 97–106. IEEE (2018)
6. Geier, M., Schönherr, M.: Esoteric twist: an efficient in-place streaming algorithms for the lattice Boltzmann method on massively parallel hardware. Computation **5**(2), 19 (2017)
7. Habich, J., Zeiser, T., Hager, G., Wellein, G.: Enabling temporal blocking for a lattice Boltzmann flow solver through multicore-aware wavefront parallelization. In: 21st International Conference on Parallel Computational Fluid Dynamics, pp. 178–182 (2009)
8. Heuveline, V., Latt, J.: The OpenLB project: an open source and object oriented implementation of lattice Boltzmann methods. Int. J. Mod. Phys. C **18**(04), 627–634 (2007)

9. Latt, J.: Technical report: How to implement your DdQq dynamics with only q variables per node (instead of 2q), pp. 1–8. Tufts University (2007)

10. Latt, J., et al.: Palabos: parallel lattice Boltzmann solver. Comput. Math. Appl. **81**, 334–350 (2020)

11. Liu, S., Zou, N., et al.: Accelerating the parallelization of lattice Boltzmann method by exploiting the temporal locality. In: International Symposium on Parallel and Distributed Processing with Applications, pp. 1186–1193. IEEE (2017)

12. Malas, T., Hager, G., Ltaief, H., Stengel, H., Wellein, G., Keyes, D.: Multicore-optimized wavefront diamond blocking for optimizing stencil updates. SIAM J. Sci. Comput. **37**(4), C439–C464 (2015)

13. Mattila, K., Hyväluoma, J., Rossi, T., Aspnäs, M., Westerholm, J.: An efficient swap algorithm for the lattice Boltzmann method. Comput. Phys. Commun. **176**(3), 200–210 (2007)

14. Mazzeo, M.D., Coveney, P.V.: HemeLB: a high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. Comput. Phys. Commun. **178**(12), 894–914 (2008)

15. Musubi (2021). https://geb.sts.nt.uni-siegen.de/doxy/musubi/index.html

16. OpenMP (2021). http://www.openmp.org

17. Palabos (2021). https://palabos.unige.ch/

18. Perepelkina, A., Levchenko, V.: LRnLA algorithm ConeFold with non-local vectorization for LBM implementation. In: Voevodin, V., Sobolev, S. (eds.) RuSCDays 2018. CCIS, vol. 965, pp. 101–113. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05807-4_9

19. Pohl, T., Kowarschik, M., Wilke, J., Iglberger, K., Rüde, U.: Optimization and profiling of the cache performance of parallel lattice Boltzmann codes. Parallel Process. Lett. **13**(04), 549–560 (2003)

20. Randles, A.P., Kale, V., Hammond, J., Gropp, W., Kaxiras, E.: Performance analysis of the lattice Boltzmann model beyond Navier-Stokes. In: 27th International Symposium on Parallel and Distributed Processing, pp. 1063–1074. IEEE (2013)

21. Rivera, G., Tseng, C.W.: Tiling optimizations for 3d scientific computations. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC 2000, p. 32. IEEE (2000)

22. Rosales, C., et al.: Remora: a resource monitoring tool for everyone. In: Proceedings of the 2nd International Workshop on HPC User Support Tools, pp. 1–8 (2015)

23. Slaughter, E., et al.: Task bench: a parameterized benchmark for evaluating parallel runtime performance. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, pp. 1–15. IEEE (2020)

24. Succi, S., Amati, G., Bernaschi, M., Falcucci, G., et al.: Towards exascale lattice Boltzmann computing. Comput. Fluids **181**, 107–115 (2019)

25. Valero-Lara, P.: Reducing memory requirements for large size LBM simulations on GPUs. Concurrency Computat. Pract. Exp. **29**(24), e4221 (2017)

26. Vardhan, M., Gounley, J., Hegele, L., Draeger, E.W., Randles, A.: Moment representation in the lattice Boltzmann method on massively parallel hardware. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, pp. 1–21 (2019)

27. Wellein, G., Hager, G., Zeiser, T., Wittmann, M., Fehske, H.: Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In: 33rd Annual IEEE International Computer Software and Applications Conference, vol. 1, pp. 579–586. IEEE (2009)

28. Witherden, F.D., Jameson, A.: Future directions in computational fluid dynamics. In: 23rd AIAA Computational Fluid Dynamics Conference, p. 3791 (2017)

29. Wittmann, M., Zeiser, T., Hager, G., Wellein, G.: Comparison of different propagation steps for lattice Boltzmann methods. Comput. Math. Appl. **65**(6), 924–935 (2013)
30. Zavodszky, G., van Rooij, B., Azizi, V., Alowayyed, S., Hoekstra, A.: Hemocell: a high-performance microscopic cellular library. Procedia Comput. Sci. **108**, 159–165 (2017)
31. Zeiser, T., Wellein, G., Nitsure, A., Iglberger, K., Rude, U., Hager, G.: Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method. Prog. Comput. Fluid Dyn. Int. J. **8**(1–4), 179–188 (2008)