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

Yuankun Fu, Feng Li
*Purdue University*
*Indianapolis, Indiana*
{*fu121, li2251*}*@purdue.edu*

Fengguang Song
*Indiana University-Purdue University*
*Indianapolis, Indiana*
*fgsong@cs.iupui.edu*

Luoding Zhu
*Indiana University-Purdue University*
*Indianapolis, Indiana*
*luozhu@math.iupui.edu*

***Abstract*—Lattice Boltzmann method (LBM) is an important computational fluid dynamics (CFD) approach to solving the Naiver-Stokes equations and simulating complex fluid flows. LBM is also well known as a memory bound problem and its performance is limited by the memory access time on modern computer systems. In this paper, we design and develop both sequential and parallel memory-aware algorithms to optimize the performance of LBM. The new memory-aware algorithms can enhance data reuses across multiple time steps to further improve the performance of the original and fused LBM. We theoretically analyze the algorithms to provide an insight into how data reuses occur in each algorithm. Finally, we conduct experiments and detailed performance analysis on two different manycore systems. Based on the experimental results, the parallel memory-aware LBM algorithm can outperform the fused LBM by up to 347% on the Intel Haswell system when using 28 cores, and by 302% on the Intel Skylake system when using 48 cores.**

***Keywords*-Lattice Boltzmann method; Memory aware parallel algorithms; Multicore systems**

## I. Introduction

Computational fluid dynamics (CFD) is a crucial area that has a variety of numerical methods to solve a wide range of scientific, engineering, and life sciences problems. Among the numerical methods, Lattice Boltzmann method (LBM) is an important class of method for modeling the Naiver-Stokes equations and simulating complex fluid flows. Nevertheless, the performance of LBM is usually bounded by memory accesses in current multi-core CPU architectures.

To that end, we design a sequential and a parallel memory-aware Lattice Boltzmann algorithm to reduce the memory bottleneck on manycore systems. The objective of our algorithm redesign is to increase data reuses across multiple fused-time-steps to improve the performance. The original LBM algorithm scans each fluid point once, and then applies one *collision* operation and one *streaming* operation to each fluid point in each iteration. This results in a low arithmetic intensity and serious bottleneck in the main memory. In order to increase data reuse and LBM's arithmetic intensity, we merge two time steps into one iteration in our proposed memory-aware LBM algorithm such that two collision operations and two streaming operations can be applied to each fluid point consecutively. This way we are able to approximately double the data reuse rate and significantly improve the cache hit rate.

It is nontrivial to merge multiple time steps into one iteration to design the new memory-aware LBM algorithms. We have solved the following challenges to provide both correct results and faster performance: 1) handling the boundary conditions for two time steps within one iteration correctly; 2) using the same two buffers as the original LBM algorithm to compute two time steps in each iteration (i.e., no extra space); and 3) handling the overlapping fluid points between different threads correctly. In addition, we use theoretical algorithm analysis to provide an insight into how many data reuses occur in each algorithm.

We perform four types of experiment. The first type of experiment evaluates the performance of the sequential memory-aware algorithm. The second and the third experiments validate the strong and weak scalability of the parallel memory-aware algorithm, respectively. Based on the scalability experiments, our parallel memory-aware LBM can significantly outperform the fused LBM on both 28-core Intel Haswell and 48-core Intel Skylake manycore systems. For instance, the memory-aware LBM is up to 347% faster in terms of strong scalability on *Bridges* in Pittsburgh Supercomputer Center (PSC), and up to 302% faster in terms of weak scalability on *Stampede2* in Texas Advanced Computing Center (TACC). In the fourth experiment, we use Paraview [1] and Catalyst [2] to visualize and validate our experimental results.

This work makes the following contributions:

- New memory-aware LBM algorithms to alleviate the memory bottleneck and design of both sequential and parallel algorithms.
- A theoretical analysis to provide insight into data reuse for each algorithm.
- Thorough experiments and in-depth performance analysis to examine the performance of the sequential and parallel memory-aware LBM algorithm.

In the remainder of the paper, the following section introduces the background of the original and fused LBM algorithms. Section III and IV show the sequential and

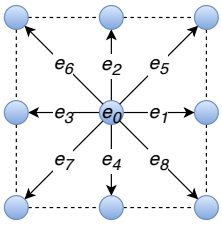



Figure 1. Illustration of the D2Q9 model.

parallel memory-aware LBM algorithm. Section V presents the analysis of the algorithms. Section VI compares the existing work with ours. Finally, Sections VII and VIII present the experimental results and summarize the paper.

## II. Background

### A. The Lattice Boltzmann Method

The Lattice Boltzmann method [3] is a modern approach in CFD to solving the incompressible, time-dependent Navier-Stokes equations numerically. The fundamental idea is that gases or fluids can be imagined as consisting of a large number of small particles moving with random motions. The exchange of momentum and energy is achieved through particle streaming and billiard-like particle collision. Compared to conventional analytical methods, LBM is relatively simple to use, easier to parallelize, and more convenient to incorporate additional physics to simulate new flow phenomena.

The Lattice Boltzmann equation with single relaxation time approximation of the collision operator (BGK model) is as follows:

$$f_i(\vec{x}+\vec{e_i}\Delta t, t+\Delta t) = f_i(\vec{x},t) - \frac{\Delta t}{\tau}[f_i(\vec{x},t) - f_i^{eq}(\rho(\vec{x},t),\vec{u}(\vec{x},t))]$$

where $f_i$ is the particle density distribution corresponding to the discrete velocity direction $\vec{e_i}$, and $\vec{x}$ and $t$ are the discrete location and time, respectively.

In this work, we simulate a classic problem of fluid flowing past a circular cylinder. Our simulation adopts the D2Q9 model for which a fluid particle has eight neighbors, and each fluid particle may move along 9 different directions including staying at the center, as shown in Fig. 1.

### B. The Original LBM Algorithm

The original LBM algorithm performs two entire sweeps over the whole data grid in every time step: one sweep for the *collision* operation, calculating the new distribution function values at each fluid node; and a subsequent sweep for the *streaming* operation, copying the distribution function values from each lattice node into its neighbors. Each fluid point uses two buffers (i.e., *buf1* and *buf2*) to store the particle distributions at the time step $t$ and $t+1$. More specifically, in the collision phase, the *buf1* of one fluid node is used to compute and then store the "intermediate" state after the collision but before streaming. Then during the streaming phase, the "intermediate" data in *buf1* is propagated to the *buf2* of this fluid node's neighbors. Algorithm 1 presents the original LBM in one time step. The baseline referential original LBM is the ANSI C implementation from Palabos [4] accessed at http://wiki.palabos.org/numerics:codes.

**Algorithm 1** Original LBM

```
1: // Collision:
2: for i=1; i≤X; i++ do
3:     for j=1; j≤Y; j++ do
4:         compute (i,j) collision using buf1
5: // Streaming:
6: for i=1; i≤X; i++ do
7:     for j=1; j≤Y; j++ do
8:         propagate (i,j) buf1 to neighbors' buf2
```

### C. The Fused LBM Algorithm

The most well-known improvement on the original LBM is to use *loop fusion* (combining the collision and the streaming step) to enhance the temporal locality [5], [6]. Instead of sweeping through the whole grid twice per time step, after calculating the distribution function values in *buf1* during collision operation, this algorithm immediately propagates the "intermediate" data to the neighbors' *buf2*. The fused LBM in one time step is shown in Algorithm 2.

**Algorithm 2** Fused LBM

```
1: // Use loop fusion to combine collision and streaming:
2: for i=1; i≤X; i++ do
3:     for j=1; j≤Y; j++ do
4:         compute (i,j) collision using buf1
5:         propagate (i,j)'s buf1 to its neighbors' buf2
```

In the next section III, our new memory-aware algorithm uses the fused LBM as the starting point to further optimize memory performance.

## III. The Memory-aware Algorithm

### A. Motivation to Focus on the Memory for LBM

Lattice Boltzmann methods, and many stencil computations (SC), are usually bounded by memory access in current multi-core CPU and GPU architectures [7]. There are mainly two sources for the memory bound: *memory latency* and *memory bandwidth*. We use LBM as an example to explain the effects of memory latency and memory bandwidth, respectively.

**Memory latency bound**. In the streaming phase, LBM updates the lattice (brings it into cache), without immediately reusing it for computation. For large-size lattices, such access pattern will cause frequent evictions of cache lines, thus introducing significantly high memory latency, due to miss penalties from both cache and TLB.

**Memory bandwidth bound**. LBM and stencil computations usually have rather high "bytes per operation",

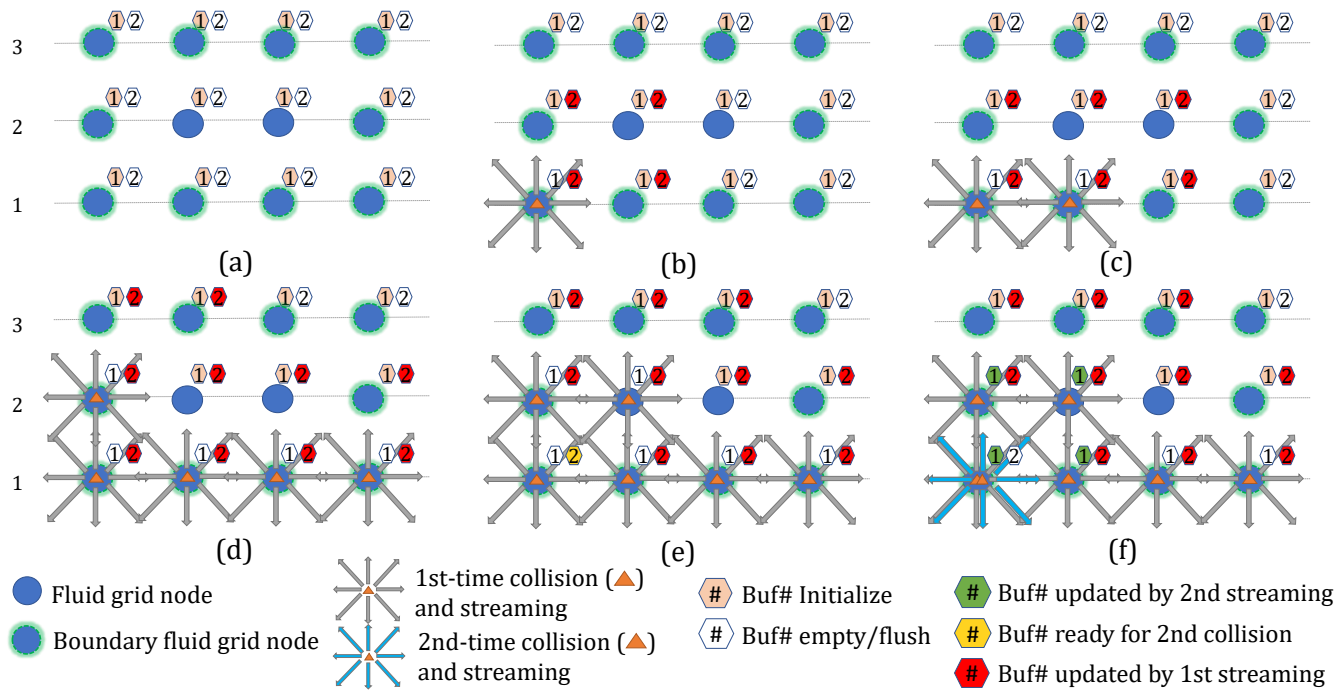



Figure 2. Sequential memory-aware algorithm. **(a)** Initialization. **(b)** First collision and streaming on $(1, 1)$. **(c)** First collision and streaming on $(1, 2)$. **(d)** Continue computing the first collision and streaming through $(2, 1)$. **(e)** First collision and streaming on $(2, 2)$ and fulfill the data dependency of $(1, 1)$ to compute the second collision and streaming. **(f)** Second collision and streaming on $(1, 1)$.

which means large memory bandwidth is required for the few computations. Nowadays computation capacity keeps climbing rapidly due to the use of more cores and wider SIMD units. Meanwhile, memory bandwidth has a slower increasing pace. To adapt to the increasing gap between CPU speed and memory speed in emerging architectures, extra effort must be made for memory-bound applications like LBM to achieve high performance, as reported by [8], [9], [10].

The good news is that both latency and bandwidth can benefit from better data reuse by carefully manipulating data access patterns. By this means, we expect to obtain better performance on modern architectures.

### B. Sequential Memory-aware LBM

As described in Section II, the fused LBM can improve the data reuse "within" each fused-step (i.e., reuse between collision and streaming). To do an even better job than the fused LBM, here we explore data reuses across *multiple* fused-steps for optimizing memory accesses.

The basic idea of our algorithm is as follows: 1) we compute the first collision and the first streaming on a block of points (assuming they fit in the cache) at the time step $t$; 2) while computing each point $(i, j)$ at the time step $t$, whenever a data dependency is fulfilled, we go back and compute the second collision and the second streaming on previously visited points at the time step $t + 1$. This idea is essentially simple, which leads to a relatively simple LBM code, as shown in Algorithm 3. A similar idea can be extended to merge more than two time steps.

We illustrate the algorithm using an example of $3 \times 4$ fluid grid in Fig. 2. Each fluid node is represented by a coordinate from bottom left corner $(1, 1)$ to top right corner $(3, 4)$. Same as the fused LBM algorithm, each fluid point has two buffers, i.e., *buf1* to store the distribution value in the time step $t$ and *buf2* in the time step $t+1$. The algorithm works as follows.

1) Fig. 2.a shows the initialization state of all fluid nodes in the current time step $t$.
2) In Fig. 2.b, we start computing the first fused collision and streaming on the fluid node $(1, 1)$ at the left corner. We use the data in *buf1* to perform collision. Then in the streaming phase, we propagate the data in the *buf1* of node $(1, 1)$ to its own and neighbors' *buf2*. Note that we draw these *buf2* with red colors. It means that their *buf2* are updated but still need other dependent data to be fulfilled for collision in the time step $t+1$ . Since *buf1* of node $(1, 1)$ has already been computed, we draw it with a white color, which means *buf1* is flushed and can be updated by other data.
3) Next in Fig. 2.c, we move right and continue computing the fluid node $(1, 2)$ and do the same fused

**Algorithm 3** Sequential Memory-aware LBM

```
1:  for i=1; i≤X; i+=block_size do
2:    for j=1; j ≤Y; j+=block_size do
3:      for ii=0; ii<block_size; ii++ do
4:        for jj=0; jj<block_size; jj++ do
5:          // First fused collision and streaming:
6:          compute (i+ii,j+jj) collision using buf1
7:          propagate (i+ii,j+jj) buf1 to neighbors' buf2
8:          if i+ii == X − 1 or X then
9:            save ρ at column X − 1 & X − 2
10:         // Second fused collision and streaming:
11:         if i+ii>1 and j+jj>1 then
12:           compute (i+ii-1,j+jj-1) collision using buf2
13:           propagate (i+ii-1,j+jj-1) buf2 to neighbors' buf1
14: handle boundary condition // see section III-C
```

collision and streaming just as node $(1,1)$ .

4) After finishing computing the bottom row 1, we move up one row and compute the first fused collision and streaming on fluid node $(2,1)$ on row 2 in Fig. 2.d.
5) In Fig. 2.e, we compute the first fused collision and streaming on node $(2,2)$ . Note that after node $(2,2)$'s streaming phase, since the *buf2* in node $(1,1)$ has collected all the particle distributions from its neighbors, the data dependency of *buf2* in node $(1,1)$ is fulfilled and ready for computing the second collision in the time step $t+1$. Thus we draw its *buf2* with a yellow color to represent that the data inside is ready for the second fused collision at the time step $t+1$.
6) In Fig. 2.f, we perform the second fused collision and streaming at the time step $t+1$ on node $(1,1)$ using *buf2*. After the streaming phase, since the *buf1* of node $(1,1)$ and its neighbors are flushed previously, we can safely propagate and store the "intermediate" data at the time step $t+1$ in those *buf1*. Thus we mark these *buf1* with green color to represent that they are updated by the data from the second streaming.

The sequential memory-aware LBM algorithm is shown in Algorithm 3, where we also use *loop blocking* in the nested loop. Thus, together with Fig. 2, we can observe that Algorithm 3 accesses fluid blocks regularly block by block, and then in each block accesses fluid points regularly line by line. With such a regular access pattern and better data locality, the sequential memory-aware LBM algorithm improves performance significantly.

### C. Special Handling of Boundary Conditions

Boundary conditions (BCs) are complex and affect greatly the stability and the accuracy of LBM. The discrete distribution functions on the boundary have to be taken care of to reflect the macroscopic BCs of the fluid. Fig. 3 shows the four BCs used in the LBM simulation:

1) The upper and lower boundaries use regularized BC.
2) At the inlet (left boundary), a parabolic Poiseuille profile is imposed on the velocity.

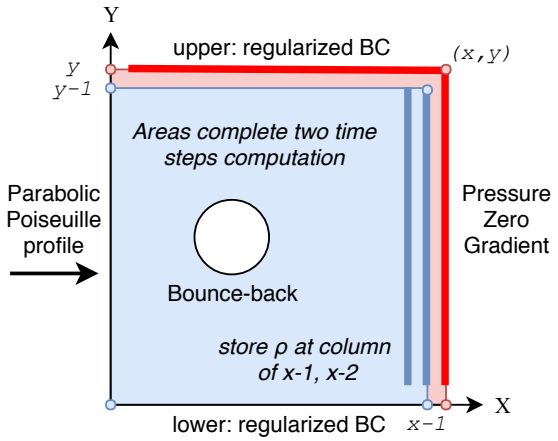



Figure 3. Four boundary conditions used in the LBM simulation. Fluid nodes in the blue area have completed two time step computation. The two red boundary lines need to be computed at the second time step using specific boundary conditions.

3) At the outlet (right boundary), we implement an outflow condition: $\nabla u = 0$. At every time step, we compute a second order extrapolation on the right boundary to ensure a zero-gradient BC on the pressure. Thus the velocity is constrained to be perpendicular to the outflow surface.
4) On the cylinder, we use the bounce-back BC.

Other different BCs can also be employed in our algorithms, as long as it follows the procedure in the next paragraph.

**Handling the Four BCs.** After executing line 1∼13 in Algorithm 3, the fluid grid area from $(0,0)$ to $(X-1,Y-1)$ has completed two time steps computation. Thus we draw it with blue color in Fig. 3. However, the upper and outlet boundaries have only completed the first time step computation. Specifically, the bottom and inlet BCs have been handled, whereas the upper and outlet BCs are left to be computed in the second time step. Therefore, we draw these two boundaries with red color in Fig. 3.

The regularized BC indicates that each fluid node's computation on upper boundary only relies on its own buffer. Thus, the upper BC can be handled simply by computing the second time step. However, to handle the right outlet zero-gradient BC, we need to conform to the formula $\rho_X = 4/3 \times \rho_{X-1} - 1/3 \times \rho_{X-2}$. This indicates the densities $\rho$ of fluid nodes at column $X$ depend on the $\rho$ at column $X-1$ and $X-2$. To ensure the correctness of the outlet BC, we use two arrays to store the $\rho$ at column $X-1$ and $X-2$ in the first time step, as reflected in line 8 and 9 in Algorithm 3. At last, after updating these densities on the outlet by the data stored in the two arrays, we can complete the outlet BC computation for the second time step correctly.

## IV. Parallel Memory-aware LBM

To support manycore systems, we use OpenMP [11] to provide a parallel implementation of the memory-aware

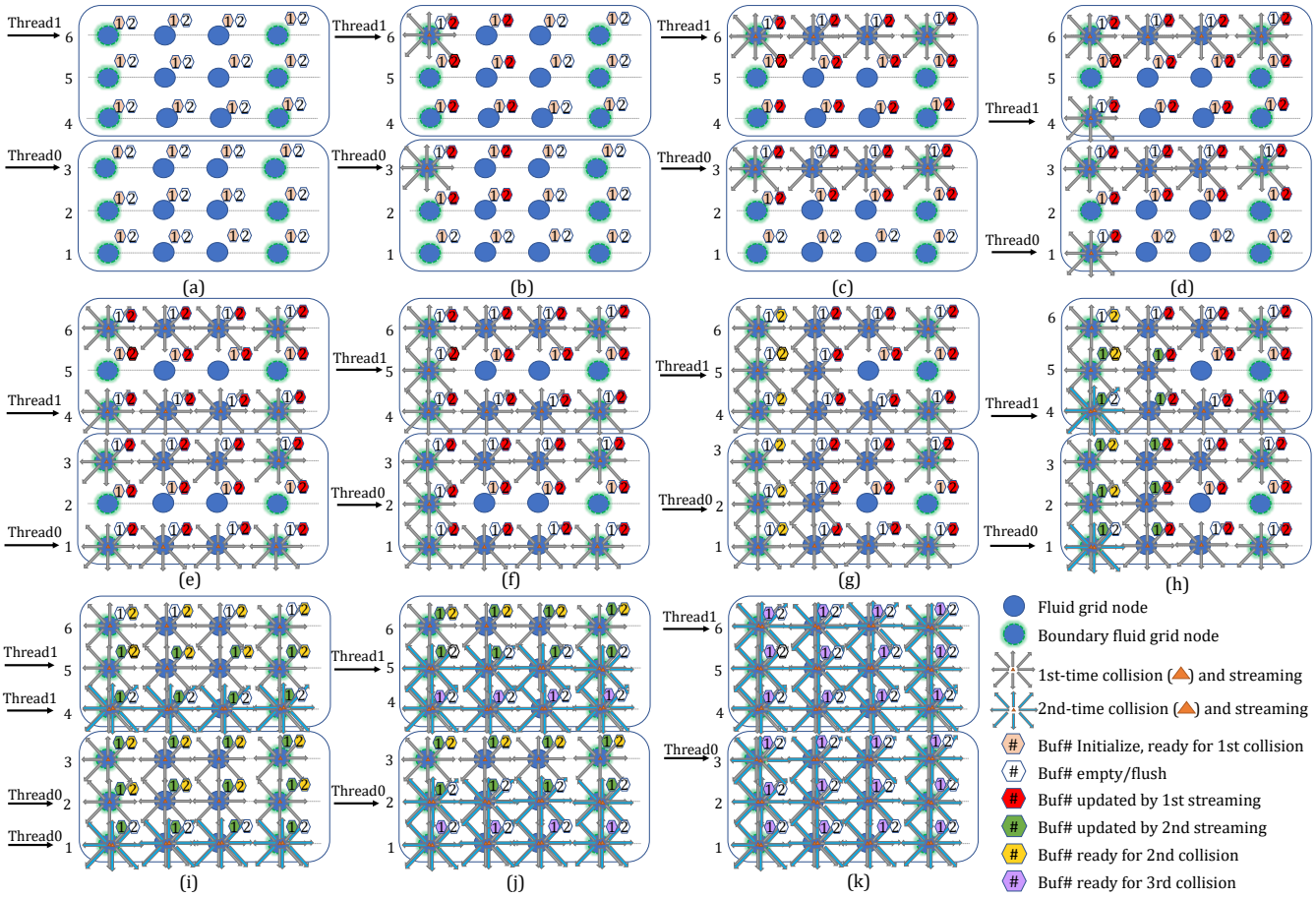



Figure 4. Parallel memory-aware algorithm using OpenMP. **(a)** Initialization. **(b)** Thread 0 and 1 start computing first collision and streaming on *thread boundaries* (i.e., row 3 and 6). **(c)** Complete computation on *thread boundaries*. **(d)** Start to compute the leftmost node on row 1 and 3. **(e)** Complete computation on row 1 and 3. **(f)** Start to compute the leftmost node on row 2 and 5. **(g)** When thread 0 and 1 complete the first computation on $(2, 2)$ and $(5, 2)$ respectively, the *buf2* in $(1, 1)$ and $(4, 1)$ fulfill the data dependency for second computation. **(h)** Compute the second collision and streaming on $(1, 1)$ and $(4, 1)$. **(i)** Complete the first computation on row 2 and 5, meanwhile complete second computation on row 1 and 4. **(j)** Complete second computation on row 2 and 5. **(k)** Complete the second computation on *thread boundaries* (i.e., row 3 and 6).

LBM algorithm. The main challenge is how to handle the overlapping area ($thread\_boundary$) between different threads, therefore to guarantee the correctness of the results.

For simplicity, Fig. 4 uses a $4 \times 6$ fluid grid as an example to illustrate the parallel memory-aware LBM algorithm 3 with OpenMP. The whole grid is computed by two threads, thus each gets a $4 \times 3$ grid region. Row 3 and 6 are the *thread boundaries*. The parallel memory-aware LBM works as follows:

1) Each thread computes the first fused collision and streaming on *thread boundary* (row 3 and 6) from Fig. 4.a to 4.c, as shown in the line 3∼7 of Algorithm 4.
2) Each thread computes the first fused collision and streaming on the bottom row of their own local region (row 1 & 4) in Fig. 4.d and 4.e. This is related to the line 14∼16 of Algorithm 4.
3) Each thread computes the first fused collision and streaming on the upper row (row 2 and 5) in Fig. 4.f.
4) When the *buf2* of some nodes fulfill the data dependency for the second collision, we draw them with the yellow color in Fig. 4.g.
5) In Fig. 4.h, each thread computes the second fused collision and streaming on these nodes, as shown in the line 20∼22 of Algorithm 4.
6) In Fig. 4.i and 4.j, each thread continues and repeats step 3 to step 5 until it finishes computing two time steps for the rest of the nodes in its local grid region except for the *thread boundary*.
7) Each thread computes the second fused collision and streaming on its *thread boundary* in Fig. 4.k. This is

involved in the line 24∼29 of Algorithm 4.

Step 1 and 7 guarantee the correctness of the results. For each thread, the *thread boundary* has data dependency both on the lower one row in its own region and the bottom row in the upper thread region. To get the correct second collision on the lower one row, step 1 propagates the first collision from *thread boundary* in advance. Similarly, step 7 needs the completion of the previous 6 steps to fulfill the data dependencies for the second collision on *thread boundary*.

Algorithm 4 shows the parallel memory-aware LBM. We distribute the whole grid into $n$ threads according to the $X$ axis. Each thread computes a local grid region with $X/n \times Y$ points. Lines of $X/n, 2X/n, ..., X$ are *thread boundaries*. This is done by using the OpenMP clause `#pragma omp for schedule(static,` $thread\_block = X/n$`)`. Note that Line 17 and 18 store the $\rho$ at column $X-1$ and $X-2$ in advance, to fulfill the BCs in line 30.

**Algorithm 4** Parallel Memory-aware LBM with OpenMP

```
1: #pragma omp parallel default(shared) {
2: // First fused collision and streaming on thread boundary:
3: #pragma omp for schedule(static)
4: for i=thread_block; i≤X; i+=thread_block do
5:     for j=1; j≤Y; j++ do
6:         compute (i,j) collision using buf1
7:         propagate (i,j) buf1 to neighbors' buf2
8: // Parallel computation within each thread:
9: #pragma omp for schedule(static, thread_block)
10: for i=1; i≤X; i+=block_size do
11:     for j=1; j≤Y; j+=block_size do
12:         for ii=0; ii<block_size; ii++ do
13:             for jj=0; jj<block_size; jj++ do
14:                 // First fused collision and streaming
15:                 compute (i+ii,j+jj) collision using buf1
16:                 propagate (i+ii,j+jj) buf1 to neighbors' buf2
17:                 if i+ii == X-1 or X then
18:                     save ρ at column X-1 & X-2
19:                 // Second fused collision and streaming
20:                 if (i+ii-1)%thread_block != 0 then
21:                     compute (i+ii-1,j+jj-1) collision using buf2
22:                     propagate (i+ii-1,j+jj-1) buf2 to neighbors' buf1
23: // Second fused collision and streaming on thread boundary:
24: #pragma omp for schedule(static)
25: for i=thread_block; i≤X; i+=thread_block do
26:     for j=1; j≤Y; j++ do
27:         compute (i,j) collision using buf1
28:         propagate (i,j) buf1 data to neighbors' buf2
29: }
30: handle boundary conditions
```

## V. Analysis of the Algorithms

In this section, we theoretically analyze how many data are reused in the original, fused and memory-aware LBM. We consider the consecutive computation on two adjacent fluid nodes in the three sequential algorithms. Then we compute the average times of data reuse in each algorithm.

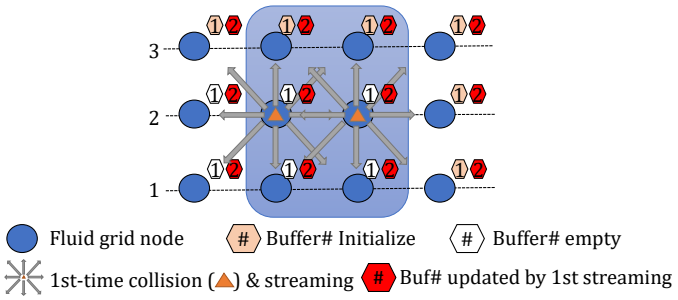



Figure 5. The original LBM has 6 $buf2$ reuses (in blue region) during the consecutive computation on two adjacent fluid nodes, while the fused LBM has 8 data reuses. (2 more $buf1$ reused.)

### A. Data reuse in Original LBM & Fused LBM

Fig. 5 shows the times of data reuse in the original LBM. For the consecutive computation on the two adjacent fluid nodes $(2, 2)$ and $(2, 3)$ during one time step, there is no data reuse in the collision since each node only uses its own *buf1*. However, during the two streaming of the two fluid nodes, since they propagate to the same 6 neighbors (in the blue region), these 6 nodes' *buf2* are reused. Thus in the original LBM, there are average $6 \div 2 = 3$ data reuses per fluid node.

Similarly for the fused LBM, there are 6 nodes' *buf2* reuses during streaming. But with *loop fusion*, we propagate the data in *buf1* immediately after collision on $(i, j)$. Thus each fluid node's *buf1* is reused during collision. With 2 nodes' *buf1* and 6 nodes' *buf2* reuses, there are average $(6 + 2) \div 2 = 4$ data reuses per node in the fused LBM.

### B. Data reuse in Memory-aware LBM

Fig. 6 illustrates the times of data reuse in the memory-aware algorithm with a $4 \times 5$ gird. Same as the fused LBM, when we compute the first fused collision and streaming on $(3, 3)$ and $(3, 4)$, there are 8 data reuses. Next, the memory-aware LBM will compute the second fused collision and

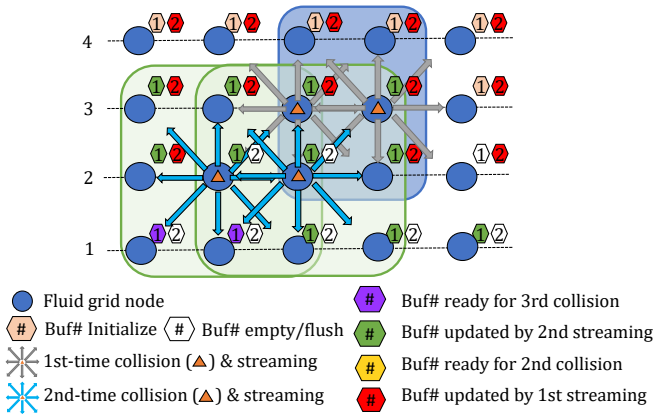



Figure 6. The memory-aware LBM has 28 data reuses during the consecutive computation on two adjacent fluid nodes (including 4 fluid nodes).

streaming on $(2,2)$ and $(2,3)$. Since we use *loop blocking*, their neighbors are still in the cache. Thus each node's *buf2* is reused during the second collision. Meanwhile, 9 nodes' *buf1* (i.e., from its own and 8 neighbors in the green region of Fig. 6) are also reused during the second streaming. Totally there are $8+(1+9)\times 2 = 28$ data reuses on 4 fluid nodes. Hence, the memory-aware LBM has average $28 \div 4 = 7$ data reuses per node.

## VI. Related Work

Wellein et al. present a pipelined wavefront parallelization approach [12] for stencil-based computations. It utilizes the wavefront parallelization scheme [13]. Within a fixed spatial domain successive, wavefronts are executed by all threads scheduled to a multicore processor with a shared last level cache. By reusing data from the same shared cache in the successive wavefronts, this strategy reduces cache misses. Instead of using all threads to compute one block, our memory-aware LBM algorithms let each thread compute a distinct data block in parallel.

T.Zeiser et al. introduce a parallel cache oblivious blocking algorithm (COLBA) [14] for the LBM in 3D. COLBA is based on a cache oblivious algorithm [15], and divides the space-time domain using *space cut* and *time cut*, thus tries to remove the explicit dependency on the cache size. However, it comes at the cost of irregular block access patterns, which causes many cache and branch-prediction misses. Due to the recursive structure of the algorithm, they also use an unconventional parallelism scheme to map the virtually decomposed domain to a tree. This work is quite different from ours since it not only uses the recursive method but also has irregular data accesses.

Pohl et al. design a sequential algorithm [5] to optimize the cache performance of LBM on a single processor core. This sequential algorithm diagonally and recursively accesses blocks down and left to avoid violating data dependencies, and also requires handling of various special cases. Differently, we introduce a regular and simple algorithm, and support parallelism on manycore systems.

As shown in section III-A that LBM is a memory bound problem, Pedro et al. propose two methods to reduce memory usage in LBM [16]. The "LBM-ghost" method uses extra ghost cells to store the intermediate results. Besides, it changes the fluid data layout and accesses data irregularly. The other "LBM-swap" method uses one fluid lattice space to avoid extra memory and accesses data regularly. But in every time step, it needs synchronization between two separate kernels, i.e., kernel collision and kernel streaming and swapping. However, our memory-aware algorithms use two lattices to combine collide and streaming kernels in two time steps, meanwhile access fluid data regularly.

Table I
Details of the experimental platforms.

| Platform | *Bridges* | *Stampede2* |
|---|---|---|
| CPU | Intel Xeon E5-2695v3 | Intel Xeon 8160 |
| # Cores | 28 on 2 sockets | 48 on 2 sockets |
| Clock rate (GHz) | 2.1∼3.3 | 2.1 nominal (1.4∼3.7) |
| L1 cache | $14 \times 32$KB | $24 \times 32$KB |
| L2 cache | $14 \times 256$KB | $24 \times 1$MB |
| L3 cache (MB) | 35 | 33 |
| Memory (GB) | 128 DDR4-2133MHz | 192 DDR4-2166MHz |
| Compiler | icc/17.4 | icc/18.0.0 |

## VII. Experimental Results

In this section, we first evaluate the performance of the original, fused and memory-aware LBM on two Intel CPU architectures deployed in two supercomputers: Haswell on *Bridges* and Skylake on *Stampede2*. Secondly, we visualize and validate the results using Paraview and Catalyst.

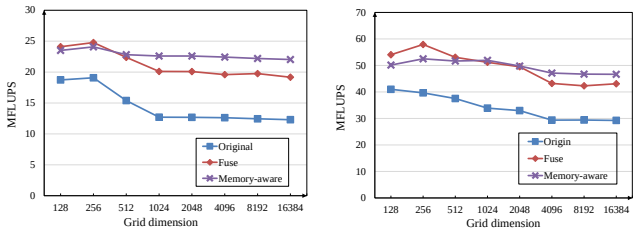
The *Bridges* system in the Pittsburgh Supercomputer Center has 752 regular nodes (128GB memory each). Each node has 28 Intel Haswell cores. The *Stampede2* system in the Texas Advanced Computing Center has 1,736 SKX nodes (192GB memory each). Each node has 48 Intel Skylake cores. More details about the architectures used in our experiments are given in Table I.

For all the implementation, we make the compiler use "`-O3`" flag on *Bridges* and "`-O3 -xCORE-AVX512`" on *Stampede2* to enable vectorization. The compiler optimization has resulted in significant speedup due to using vector instructions. All our experiments are performed using double precision. Besides, we use the conventional MFLUPS metric (millions of fluid lattice updates per second) to evaluate the performance of each LBM algorithm.

### A. Sequential performance

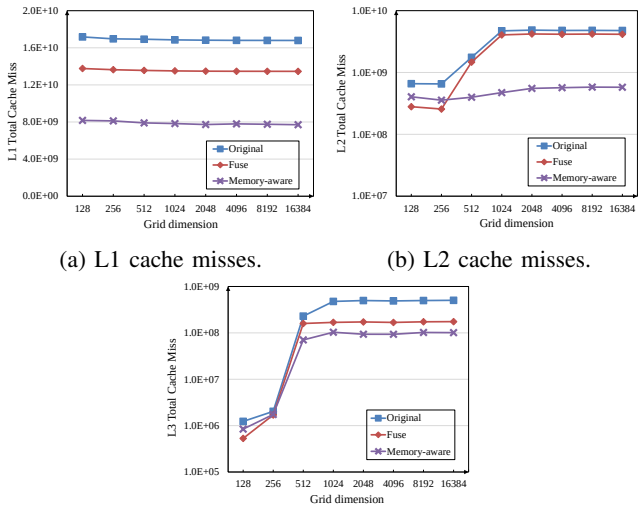
We have described the sequential memory-aware LBM in Section III. Our first experiment is intended to compare the sequential performance of original, fused and memory-aware LBM. The experiments are performed on a single core and the grid size ranges from $128\times128$ to $16384\times16384$. Since we use *loop blocking* in the sequential memory-aware LBM Algorithm 3. We search the best $block\_size$ that can achieve the fastest performance and the best $block\_size$ is 64. Then we set $block\_size = 64$ in the sequential memory-aware LBM experiments to compare with the other two algorithms.

Fig. 7 shows the sequential performance of the three LBM algorithms on Intel Haswell and Skylake CPU, respectively. Fig. 7.a shows that the fused LBM is up to 1.6 times faster than the original LBM using Haswell CPU. On the other hand, the memory-aware LBM achieves 22.0 MFLUPS and is faster than the fused LBM by up to 115% when the grid size is $16384\times16384$. Fig. 7.b shows that the fused LBM is up to 1.5 times faster than original LBM using Skylake CPU. Also, the memory-aware LBM obtains up to 51.9 MFLUPS and is faster than the fused LBM by up to 110%. We observe




(a) *Bridges*. (b) *Stampede2*.

Figure 7. Sequential performance using three LBM algorithms.



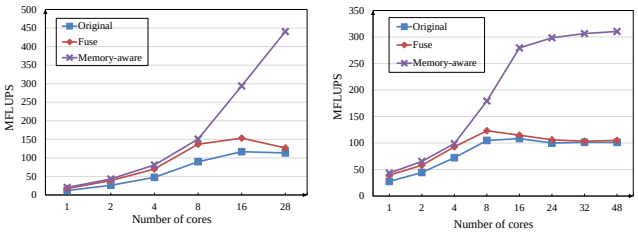

(a) L1 cache misses. (b) L2 cache misses.

(c) L3 cache misses.

Figure 8. Sequential performance analysis using PAPI on *Birdges*.

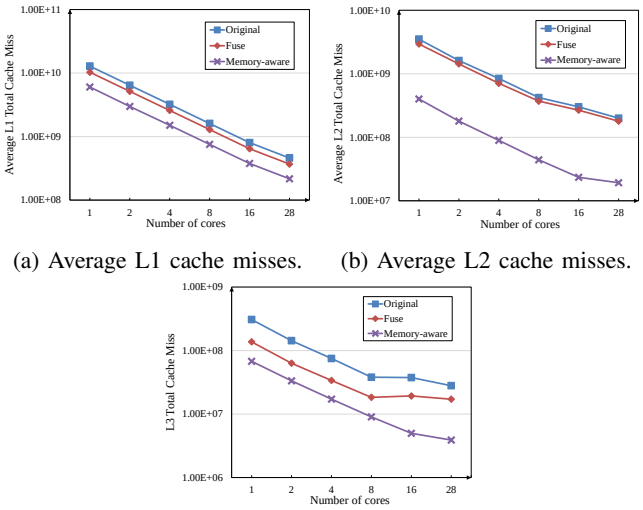
from Fig. 7.a and 7.b that when the grid size is small ($128 \times 128$ and $256 \times 256$), the fused LBM is better. However, the memory-aware LBM is consistently faster than the fused LBM starting from ($1024 \times 1024$) on both CPUs.

To dig into the reason, we use the PAPI CPU component [17] to collect the number of L1, L2, L3 cache misses on the Intel Haswell CPU, with results shown in Fig. 8. Fig. 8.a shows the number of L1 cache misses using three different algorithms, where memory-aware LBM has the lowest L1 cache misses (nearly half of that in the fused LBM). This is expected since our carefully-designed memory-aware method has better locality and cache-reuse. However, the memory-aware LBM can have worse L2/L3 cache misses in small grid sizes($128\times128$ & $256\times256$), as we can see from Fig. 8.b and 8.c. This is also easy to explain: if the grid can fully fit into the CPU cache, our cache-reuse optimization method simply won't help much. For instance, the $256 \times 256$ grid needs 9MB memory space, which is less than the 35MB L3 cache size in the Haswell CPU of the *Bridges*. However, considering that LBM simulation usually runs on a large scale, we can still claim that our memory-aware LBM will outperform the other two methods in most use cases.




(a) *Bridges*. (b) *Stampede2*.

Figure 9. Strong scalability using three LBM algorithms.




(a) Average L1 cache misses. (b) Average L2 cache misses.

(c) L3 cache misses.

Figure 10. Strong scalability performance analysis using PAPI.

## B. *Strong Scalability*

Our second experiment evaluates the strong scalability performance of the parallel original, fused and memory-aware LBM. The grid size on a single node is $14336\times14336$ on *Bridges*, and $24576 \times 24576$ on *Stampede2*. We still set $block\_size = 64$ in our parallel memory-aware LBM. Fig. 9.a shows the strong scalability of the three LBM algorithms on a single Haswell node of *Bridges*. We can see that the original and fused LBM algorithm cannot scale well when the number of cores is larger than 8. In contrast, our memory-aware algorithm scales well even when the number of cores reaches 16 and 28, which gets up to 440.6 MFLUPS and 3.5 times of speedup compared to the fused LBM. Similarly, as depicted in Fig. 9.b, on a Skylake node of *Stampede2*, the original and fused algorithms scales at most 8 cores. However, the memory-aware LBM algorithm can scale to 16 cores, obtain up to 310.6 MFLUPS and outperform parallel fused LBM by up to 3.0 times.

To give an in-depth examination of the strong scalability performance of the three algorithms, we again use PAPI to compare and analyze the relevant hardware counters on *Bridges*. Results are shown in Fig. 9.a, where we count the number of L1, L2, L3 misses for each thread and then

Table II
CONFIGURATION FOR WEAK SCALABILITY EXPERIMENTS FOR THE INTEL HASWELL CPU ON *Bridges*.

| # Cores | 1 | 2 | 4 | 8 | 16 | 28 |
|---|---|---|---|---|---|---|
| X ($\times 10^3$) | 5.12 | 7.2 | 10.24 | 14.4 | 20.48 | 27.104 |
| Y ($\times 10^3$) | 5.12 | 7.2 | 10.24 | 14.4 | 20.48 | 27.104 |
| Memory (GB) | 3.52 | 6.95 | 14.06 | 27.81 | 56.25 | 98.52 |

Table III
CONFIGURATION FOR WEAK SCALABILITY EXPERIMENTS FOR THE INTEL SKYLAKE CPU ON *Stampede2*.

| # Cores | 1 | 2 | 4 | 8 | 16 | 24 | 32 | 48 |
|---|---|---|---|---|---|---|---|---|
| X ($\times 10^3$) | 5 | 7.07 | 10 | 14.08 | 20 | 24.48 | 28.16 | 33.6 |
| Y ($\times 10^3$) | 5 | 7.07 | 10 | 14.08 | 20 | 24.48 | 28.16 | 33.6 |
| Memory(GB) | 3.35 | 6.70 | 13.41 | 26.59 | 53.64 | 80.37 | 106.35 | 151.41 |

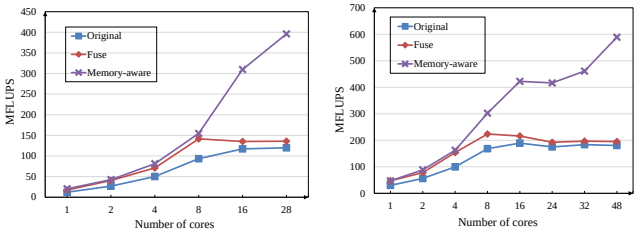
calculate the average among all threads. Fig. 10.a, 10.b and 10.c show that the memory-aware LBM algorithm has the smallest number of L1, L2, L3 cache misses, which is 1.7, 11.4 and 4.4 times less than the fused LBM, respectively. So we can see that the advantage of better cache-use in our parallel memory-aware LBM provides better performance. The reason why the memory-aware LBM scale relatively slower from 24 to 48 cores on the Skylake CPU in Fig. 9.b can be that the Skylake CPU clock speed decreases as the number of active cores used and the vector instruction set, as reported by the user guide of *Stampede2*. [1]

### C. Weak Scalability

Our third experiment evaluates the weak scalability performance of the three parallel LBM algorithms. The grid size and memory used on a single node of *Bridges* and *Stampede2* are presented in Table II and III, receptively. Fig. 11.a shows the weak scalability on the Haswell CPU. We can see that our memory-aware method has the advantage of scaling well on 16 and 28 cores, and can achieve up to 396.1 MFLUPS and 2.9 times of speedup over the fused LBM. Similarly, Fig. 11.b shows the weak scalability on the Skylake CPU. Again, the original and fused LBM algorithm are not scalable from 16 to 48 cores. The memory-aware LBM scales the best, reaches up to 589.0 MFLUPS and outperforms the fused LBM by up to 3 times.

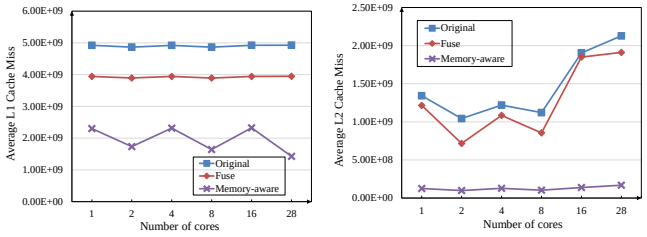
We use PAPI to analyze the weak scalability experiments on *Bridges*. Fig. 12.a, 12.b and 12.c show that the memory-aware LBM has the smallest number of L1, L2, L3 cache misses, which is up to 2.3, 13.8, and 4.6 times smaller compared to the fused LBM, respectively. We conclude that the best weak scalability performance in memory-aware LBM benefits from the least number of cache misses in all three level caches.

[1]The actual clock of the SKX CPU depends on the vector instruction set, number of active cores, and other factors affecting power and temperature limits. A single core serial code using the AVX2 instruction set may run at 3.7GHz, while a large, fully-threaded MKL dgemm may run at 1.4GHz.

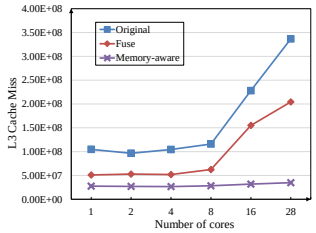



(a) *Bridges*. (b) *Stampede2*.

Figure 11. Weak scalability using three LBM algorithms.




(a) Average L1 cache misses. (b) Average L2 cache misses.



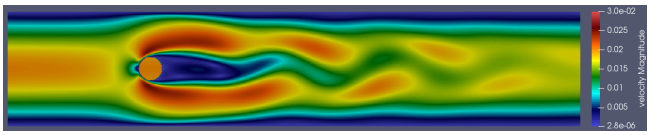

(c) L3 cache misses.

Figure 12. Weak scalability performance analysis using PAPI.

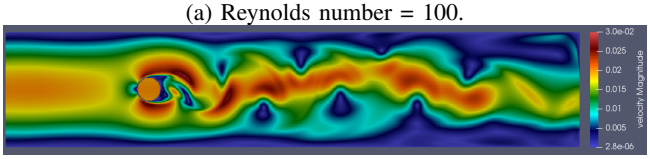
### D. Visualization

The last experiment is used to visualize and validate the memory-aware LBM algorithm. Our simulation examines the widely known and extended test scenario, a flow past a cylinder placed in a channel, which dates back to the design of wings of an aircraft and understanding the behavior of the flow past them in the early 20th century. It turns out that the *Reynolds number* (i.e., the ratio of a fluid's inertial force to its viscous force) plays an important role in characterizing the behavior of the flow. As the Reynolds number increases to 100 or higher, an unstable periodic pattern is created, which is called the *Karman vortex street*.

Our algorithm can compute and output the velocity of each fluid point. We use Catalyst to convert those output to VTK files. Next, Paraview reads the VTK files and generates figures and videos. The simulation is a flow past a $1280\times256$ channel. An uncompressed cylinder at location (320,128) with radius equaling to 26 makes the steady-state symmetrical flow unstable. In Fig. 13, after running 100,000 steps, a Karman vortex street is generated. Fig. 13.a and 13.b show the Karman vortex street with the Reynolds number 100 and 400, respectively. We can observe that more vortices are generated when the Reynolds number equals to 400. This is




(a) Reynolds number = 100.




(b) Reynolds number = 400.

Figure 13. Vorticity plot of flow past a cylinder, a Karman vortex street is generated

because inertial forces dominate the viscous forces at higher Reynolds numbers, which tend to produce more chaotic eddies and induce flow instabilities [18]. The full simulation videos are published at https://youtu.be/C5IqsZVPV0Y and https://youtu.be/hyNN6yxdn18.

## VIII. Conclusion

To address the memory bound limitation of Lattice Boltzmann method in manycore systems, we propose the memory-aware LBM algorithm, which improves the fused LBM by carefully tweaking the data access pattern. We provide a detailed algorithm analysis to demonstrate how our memory-aware LBM algorithm can enable more data reuses across multiple fused-steps. The sequential, strong and weak scalability experiments show that our method outperforms the fused LBM by up to 347% on the Intel Haswell system when using 28 cores, and by 302% on the Intel Skylake system when using 48 cores. Moreover, we use PAPI to give an insight into the speedup reasons. Our future work will extend and apply the memory-aware idea to 3D and distributed memory HPC systems.

## Acknowledgment

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by the NSF Grant# ACI-1548562. It is involved in the Project# TG-ASC170037 and also supported by the NSF Grant# 1522554.

## References

[1] U. Ayachit, "The paraview guide: a parallel visualization application," 2015.

[2] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. E. Jansen, "The paraview coprocessing library: A scalable, general purpose in situ visualization library," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. IEEE, 2011, pp. 89–96.

[3] S. Chen and G. D. Doolen, "Lattice Boltzmann method for fluid flows," *Annual review of fluid mechanics*, vol. 30, no. 1, pp. 329–364, 1998.

[4] Palabos, "http://www.palabos.org/," 2016.

[5] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde, "Optimization and profiling of the cache performance of parallel lattice Boltzmann codes," *Parallel Processing Letters*, vol. 13, no. 04, pp. 549–560, 2003.

[6] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann San Francisco, 2002, vol. 1.

[7] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–13.

[8] M. Reilly *et al.*, "When multicore isn't enough: Trends and the future for multi-multicore systems," in *HPEC*, 2008.

[9] 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," *Computers & Fluids*, vol. 80, pp. 276–282, 2013.

[10] P. Valero-Lara, A. Pinelli, and M. Prieto-Matias, "Accelerating solid-fluid interaction using lattice-boltzmann and immersed boundary coupled simulations on heterogeneous platforms," *Procedia Computer Science*, vol. 29, pp. 50–61, 2014.

[11] OpenMP, "http://www.openmp.org/," 2018.

[12] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, vol. 1. IEEE, 2009, pp. 579–586.

[13] A. Hoisie, O. Lubeck, and H. Wasserman, "Performance analysis of wavefront algorithms on very-large scale distributed systems," in *Workshop on wide area networks and high performance computing*. Springer, 1999, pp. 171–187.

[14] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rude, and G. Hager, "Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method," *Progress in Computational Fluid Dynamics, an International Journal*, vol. 8, no. 1-4, pp. 179–188, 2008.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium*. IEEE, 1999, pp. 285–297.

[16] P. Valero-Lara, "Reducing memory requirements for large size LBM simulations on GPUs," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 24, p. e4221, 2017.

[17] PAPI project, "http://icl.utk.edu/papi/," 2018.

[18] M. Van Dyke and M. Van Dyke, "An album of fluid motion," 1982.