Computers and Fluids 181 (2019) 1–7

Contents lists available at ScienceDirect

Computers and Fluids

journal homepage: www.elsevier.com/locate/compfluid

Benchmark solutions

An auto-vecotorization friendly parallel lattice Boltzmann streaming scheme for direct addressing

Markus Mohrhard[a,b,*], Gudrun Thäter[a], Jakob Bludau[c], Bastian Horvat[c], Mathias J. Krause[a,b]

[a] *Institute of Applied and Numerical Mathematics 2, Karlsruhe Institute of Technology, Englerstr. 2, Karlsruhe 76131, Germany*
[b] *Lattice Boltzmann Research Group, Karlsruhe Institute of Technology, Englerstr. 2, Karlsruhe 76131, Germany*
[c] *Chair of Helicopter Technology, Technical University Munich, Boltzmannstr. 15, Garching 85748, Germany*

ARTICLE INFO

Article history:
Received 14 November 2018
Accepted 2 January 2019
Available online 3 January 2019

Keywords:
Lattice Boltzmann methods
Fluid dynamics
Lattice Boltzmann streaming

ABSTRACT

Lattice Boltzmann methods (LBM) are used for massively parallel computational fluid dynamics simulations since they are easily parallelizable with a perfectly parallel and local in space collision step and a streaming step that only transfers data between neighboring grid points. Current CPU hardware architectures focus on increasing parallelism through additional CPU cores and wider vector instruction sets. To benefit from these developments parallel LBM schemes need to be designed with these concepts of parallelism in mind. This paper presents a new easily automatically vectorizable LBM streaming scheme for directly addressed grids which is based on the A-A pattern streaming algorithm. Combined with several implementation techniques the new algorithm provides a speedup of more than three compared to an unvectorized implementation. The algorithm also provides implementation benefits compared to the A-A pattern algorithm.

1. Introduction

Today's processors are based on multiple cores each containing advanced vector instruction sets for single instruction multiple data (SIMD) processing. Benefiting from these processors requires the use of parallel algorithms combined with implementations that efficiently use the vector instruction sets [1]. SIMD instructions can be added to an implementation through assembler instructions, compiler intrinsics or through automatic vectorization by the compiler. While intrinsics and assembler provide the developer with full control over the used vector instructions, they also require regular adaption of the code to new instruction sets whereas automatic compiler vectorization is able to target all available instruction sets. Automatic vectorization by the compiler is supported by many modern compilers but is has been shown that compilers have problems with complex memory access patterns or complicated control flows [2].

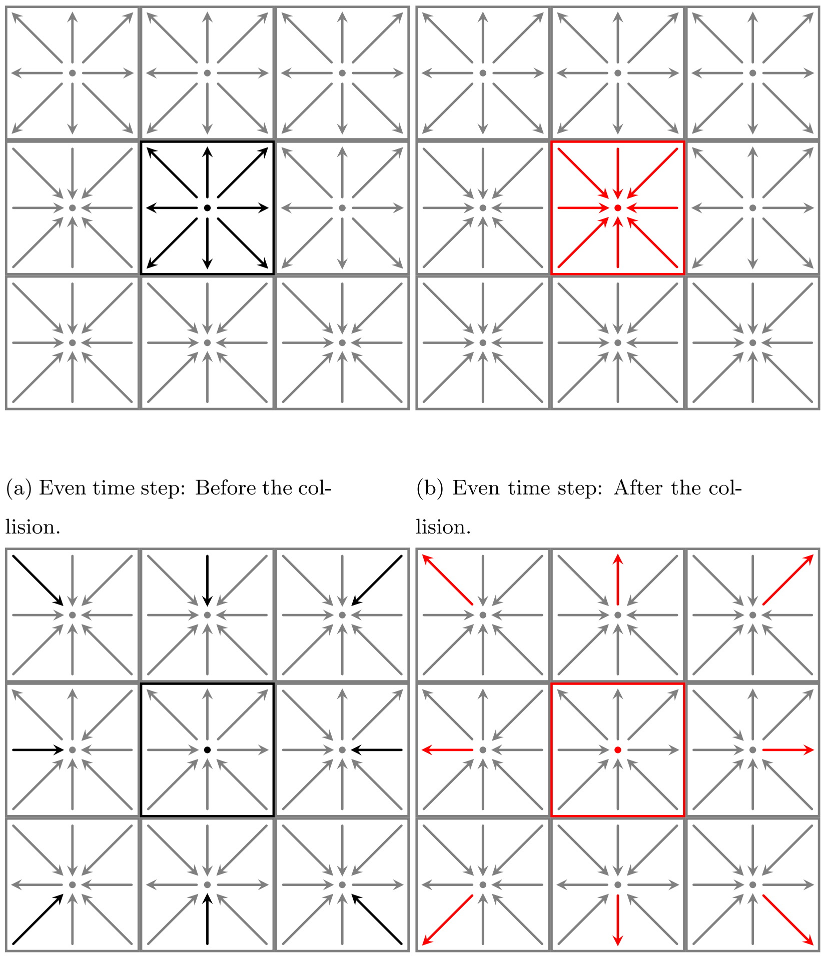
Lattice Boltzmann methods (LBM) [3] are inherently parallel methods through their purely local collision step and a data streaming that only transfers data between neighboring grid points. LBM are used for simulations in math [4], engineering [5] and medicine [6] as well as quantum mechanics [7]. Existing streaming schemes of the LBM can be separated into one grid and two grid algorithms. The classical two grid algorithms separate read and write operations through the use of independent grids. The one grid algorithms require complex strategies to deal with the resulting data dependencies, nevertheless they are able to use the available memory more efficiently [8]. Two commonly used fully parallel one grid streaming schemes have been described: A-A pattern by Bailey et al. [9] and esoteric twist by Geier et al. [10].

This work presents a new parallel LBM streaming scheme for directly addressed one grid implementations with focus on easy vectorization and simplified memory access based on the parallel LBM streaming scheme A-A pattern. The potential for efficient auto-vectorization is achieved by providing the data for the collision step always locally and therefore avoiding the necessity for A-A pattern's two collision implementations. The streaming in the new algorithm is handled through a pointer shift and pointer swaps inside of the mandatory structure of arrays data structure.

Based on a two dimensional test case with a BGK model [11] on a 128x128 grid, a speedup of more than 3.4 is measured compared to an unvectorized implementation and a factor of 2.5 compared to an array of structure with a collision optimized swap streaming [12] based implementation. The features of the new streaming algorithm, which allows efficient auto-vectorization and the possibility for simple collision implementations, make the new algorithm a good candidate for general purpose GPUs. Through a domain decomposition approach with shrinking [13] the advantages

* Corresponding author at: Institute of Applied and Numerical Mathematics 2, Karlsruhe Institute of Technology, Englerstr. 2, Karlsruhe 76131, Germany.
E-mail address: markus.mohrhard@kit.edu (M. Mohrhard).

https://doi.org/10.1016/j.compfluid.2019.01.001
0045-7930/© 2019 Elsevier Ltd. All rights reserved.


(a) Even time step: Before the collision.

(b) Even time step: After the collision.

(c) Odd time step: Before the collision.

(d) Even time step: After the collision.

Fig. 1. The A-A pattern algorithm. The processing order is bottom left to top right and the currently processed grid point is in the center.

of the new algorithm can be combined with the advantages of an indirect addressing scheme.

2. The lattice Boltzmann methods

LBM are a set of mesoscopic approaches that discretize the Bolzmann equation in time and space. The general form of the LBM can be written as

$$f_i(x + c_i\Delta t, t + \Delta t) = f_i(x, t) + \Omega(f_i(x, t))$$

with f_i being the discrete particle disribution functions (PDF), x being the discrete position in space, c_i being the discrete velocity, t being the current discrete point in time, Δt being the time step (usually $\Delta t = 1$). Ω represents a the collision operator which in the following work will be the BGK collision operator.

Usually LBM equations can be split into a collision part (c) and a streaming part (s) as

$$f_i^*(x, t) = f_i(x, t) + \Omega(f_i(x, t)), \tag{c}$$

$$f_i(x + c_i\Delta t, t + \Delta t) = f_i^*(x, t). \tag{s}$$

The collision formula only uses data of each grid point whereas the streaming step exchanges the post-collision data with neighboring grid points. The data exchange between the grid points introduces data dependencies between these grid points that needs to be resolved by the implementation (cf. Section 3).

Besides the collision operator LBM models are also characterized by the selected discrete velocity set. Usually the velocity sets are written in a form DdQq with the d standing for the dimension (one dimensional, 2-dimensional or 3-dimensional) and the q value for the number of dicrete velocity directions (c_i with $i = 0, \ldots, q-1$). Common LBM velocity sets are D1Q3, D2Q9, D3Q19 and D3Q27, however more complex models have been proposed in the literature [14,15]. While the following work uses the D2Q9 velocity set the analyzed streaming schemes support any LBM velocity set.

3. Novel streaming scheme

The LBM streaming schemes can be classified into one grid and two grid schemes based on whether they require one or two sets of the grid data. Two grid streaming schemes are perfectly parallel by using separate grids for reading and storing the PDFs. One grid schemes need to resolve the data dependencies that appear in the streaming equation of the LBM through the streaming algorithm. One grid algorithms provide several benefits, especially lower memory requirements and better cache utilization, however

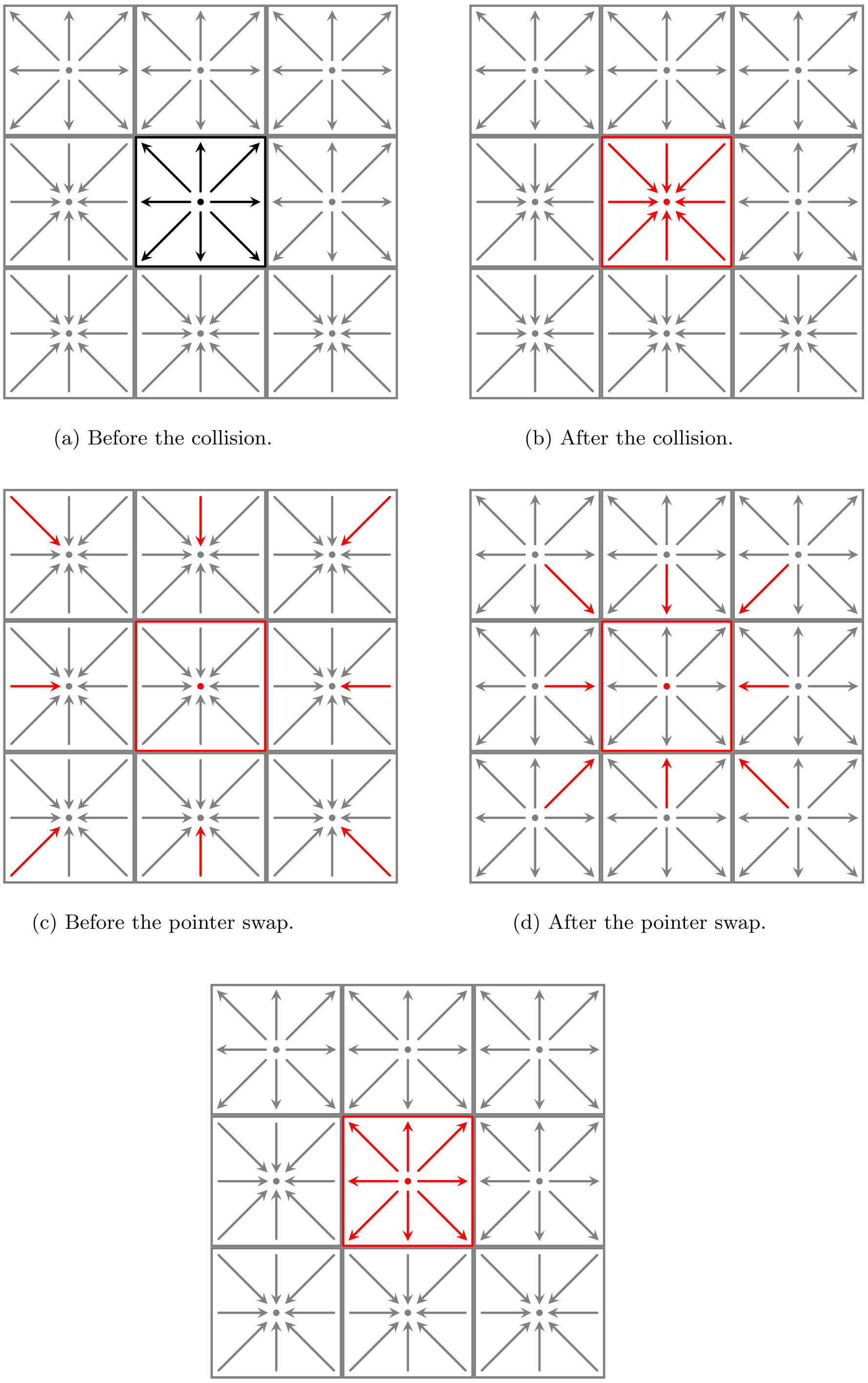

(a) Before the collision.
(b) After the collision.
(c) Before the pointer swap.
(d) After the pointer swap.
(e) After the pointer shift.

Fig. 2. The memory access for the new streaming algorithm. Figs. 2a and b represent a normal collision step. Figs. 2c, d and e represent the pointer operations necessary to shift the PDFs from their old location to the new location for the next collision.

they require more complex implementations. An extensive analysis of existing LBM streaming algorithms can be found in [8].

3.1. A-A pattern streaming

The A-A pattern streaming scheme by Bailey [9] is a one grid streaming scheme that resolves the data dependencies through different memory access patterns for even and odd time steps. As a parallel one grid streaming scheme the A-A pattern algorithm is a popular choice for GPU based LBM implementations. During the even time step the memory is read from its natural locations (Fig. 1a) and written back to the opposite direction (Fig. 1b). Therefore during the even time step the algorithm only performs a collision step and accesses memory only on the local grid point. In contrast to the even time step, during the odd time step memory is read from the grid points surrounding the grid point (Fig. 1c) and written back to the cells on the opposite side after the collision (Fig. 1d). The odd time step can also be interpreted as a combination of a streaming step, a collision step and another streaming step.

The A-A pattern algorithm provides several benefits from an implementation standpoint, starting with the property that only *q* load and *q* store operations are necessary. As both even and odd time steps only write back to memory locations that have just been read there is no need for non-temporal stores. Due to the different memory access patterns for even and odd time steps, A-A pattern based implementations usually require two implementations for each collision and boundary model which can significantly increase the code complexity and make maintenance more costly.

3.2. New streaming scheme

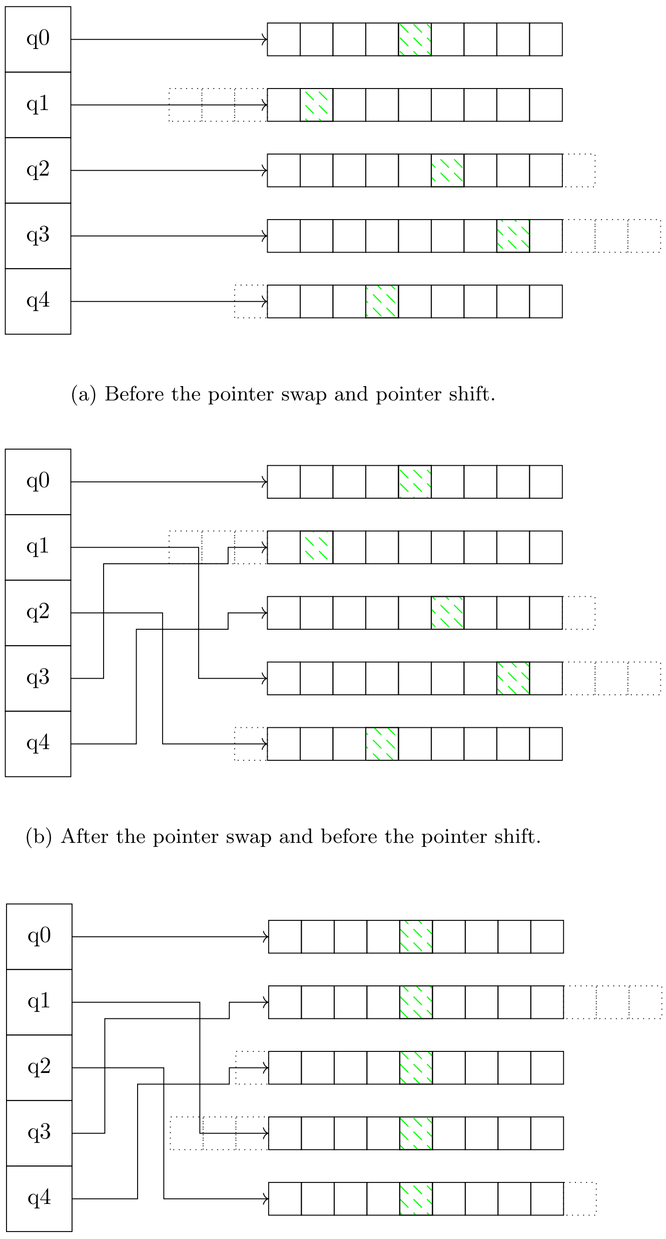
The new streaming scheme, named shift and swap streaming algorithm (SSS), uses the same memory locations as the A-A pattern algorithm combined with a directly addressed grid and a mandatory structure of array (SoA) data layout where each PDF direction is stored in a separate array. The algorithm's even time step looks exactly the same as the A-A pattern's even time step for the same data layout (Figs. 2a b). In contrast to A-A pattern's streaming through switching the memory access pattern, the new algorithm updates the control structure of the SoA data structure. After the switch from the even to the odd time step the data is in the same positions as in the A-A pattern's pre-collision location (Fig. 2c). As part of the next step the pointers to the arrays for opposite directions are exchanged, resulting in Fig. 2d. During the final step, the starting point of the arrays are shifted to provide the same index based access to the data for all PDF directions (Fig. 2e). The steps for the SoA data updates during streaming can also be seen in Fig. 3. After these data array updates the data is in the same locations for the collision as during the even time step of the A-A pattern streaming scheme.

The SSS shares many properties with the A-A pattern streaming, especially the perfect parallel execution and having only *q* load and *q* store operations. In addition the algorithm only requires one collision and boundary implementation as even and odd time steps use the same data access patterns.

3.3. Techniques for improved auto-vectorization

The proposed algorithm already allows efficient vectorized execution of the collision step. However, in a directly addressed grid the data contains also the data for boundary grid points and points not belonging to the simulation domain. In non-vectorized implementations these cases can be handled through explicit or implicit – e.g. virtual function calls – branching whereas the new algorithm's vectorization potential depends on the absence of any branching. By introducing a binary mask that stores whether a grid point is a fluid grid point, the branching can be simplified to an unconditional fluid collision whose result is thrown away for non-fluid grid points. The ignored boundary grid points can be processed after all the fluid nodes have been processed. In addition, explicit fixed size loops are introduced that combine the update of several consecutive grid points. The additional loops with fixed width support the automatic vectorization by simplifying the instructions that the compiler should vectorize. Listing 1 shows the combination of these techniques based on a simplified implementation.

For A-A pattern and the new streaming scheme unaligned memory access can not be avoided during odd time steps. However, AVX and AVX2 in modern Intel processors have relaxed the alignment requirements [16] which provides improved automatic vectorization possibilities for the new algorithm.



(a) Before the pointer swap and pointer shift.

(b) After the pointer swap and before the pointer shift.

(c) After the pointer swap and after the pointer shift.

Fig. 3. The control structure in the SSS algorithm before and after the pointer swap for a D2Q5 stencil on a 3×3 grid. The memory with the green pattern represents the memory location that corresponds to the grid point in the middle in the new time step. The dotted memory locations represent the locations that don't correspond to the 9 grid points but are necessary due to the shift. q0 to q4 are the pointers of the control structure. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

4. Numerical results

The SSS's vectorization is evaluated based on a 2-dimensional 128x128 grid point double precision test case with a BGK collision model. The test case is implemented as part of the *OpenLB* LBM framework [17]. The *OpenLB* framework uses a direct addressing mode and allows the selection of several different collision and boundary models as well as the selection between single and double precision calculations. The Intel compiler version 14 is used with optimization settings *O3* and *xHost* on a Intel i7-4770K pro-

```
constexpr int loop_width = 8;
constexpr int q = 9;

constexpr int opposite(int dir) {
    if (dir == 0) return 0;
    if (dir <= q/2) return dir + q/2;
    else return dir - q/2;
}

void lbm (double** data, bool* mask, int data_points)
{
  // special handling for data_points not being
  // a multiple of loop_width not shown
  for (int i = 0; i < data_points; i += loop_width) {

    // calculate a few shared variables (e.g velocity)
    // before the next loops
    for (int lp_index = 0; lp_index < loop_width; ++lp_index) {

    }

    // special handling for direction == 0 necessary here
    for (int direction = 1; direction <= q/2; ++direction) {

      // explicit fixed size loop for efficient vectorization
      for (int lp_index = 0; lp_index < loop_width; ++lp_index) {

        // calculate new values for direction and opp_direction
        // and store results in new_value1 and new_value2
        int opp_index = opposite(direction);
        int index = i + lp_index;
        bool fluid = mask[index];
        data[opp_index][index] =
           data[opp_index][index] * !fluid + fluid * new_value1;
        data[direction][index] =
           data[direction][index] * !fluid + fluid * new_value2;
      }
    }
  }
}
```

Listing 1. Structure of the code supporting automatic vectorization across grid points.

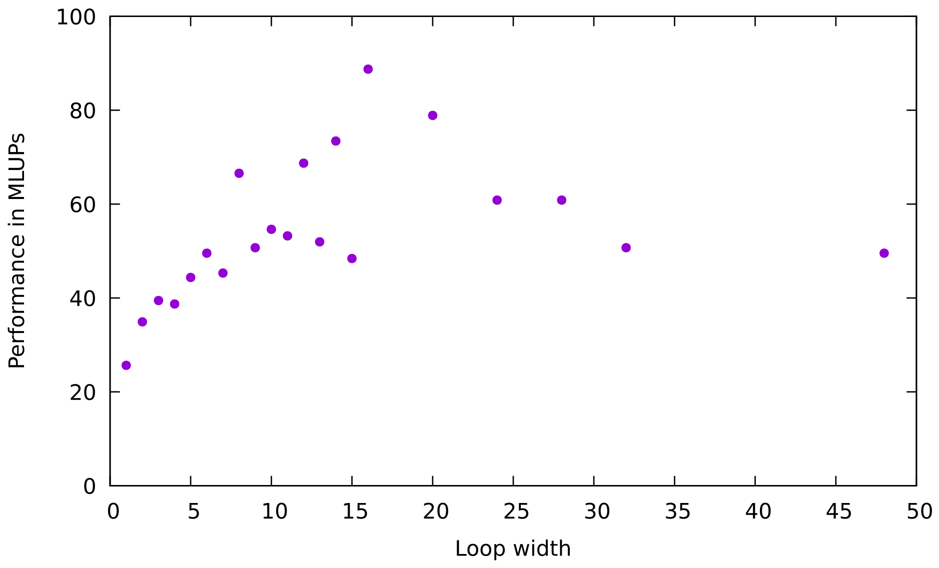


Fig. 4. Performance in MLUPs for the test problem based on the explicit fixed width loop as measured on a Intel i7-4770K.

cessor. For the performance evaluation the existing AoS data structures in *OpenLB 1.2* are changed to a SoA data layout and the fixed loop width pattern described in Section 3.3 is used with various loop widths. The performance for the same test case based on the existing swap [12] and AoS based implementation of *OpenLB* release 1.2 is 34 million lattice updates per second (MLUPs).

Fig. 4 shows the performance in MLUPs for the selected explicit loop width. The graph shows that the new implementation with larger explicit loop widths is faster than the old implementation. The optimal performance is achieved for a loop width of 16 which is 4 times larger than the native vector instruction set width of 4 for double precision numbers. For a loop width of one the performance of the new implementation is slightly worse than the old implementation, however for a loop width of 16 the performance of the new implementation is about 2.5 times faster.

The unexpected result that the performance is optimal for a loop width of 16 appears to be related to the unaligned memory access during odd time steps. The larger loop width results in a loop unrolling effect with 4 equal instructions being performed after each other which reduces some of the unaligned cache access problems. The results show that using an additional explicit fixed size loop is a mandatory step to improve the performance for the new implementation.

4.1. Bandwidth

An efficient LBM implementation is expected to be memory bandwidth limited [18–20]. For the new algorithm the bandwidth is compared against the theoretical maximum memory bandwidth to show that the implementation easily saturates the available memory bandwidth. The used D2Q9 model for double precision data requires $9 * 8 = 72$ bytes for both reading and writing the data for each grid point for a combined 144 bytes per time step and grid point. Based on the 88 MLUPs reached with a loop width of 16 the used bandwidth can be calculated as:

$$\textbf{Bandwith} = 144\frac{\text{bytes}}{\text{point}} * 88 * 10^6\frac{\text{points}}{\text{s}} = 11.8\frac{\text{GB}}{\text{s}}.$$

The Intel i7-4770K processor used for the experiments has a theoretical memory bandwidth for the whole processor (4 cores) of 25.6 GB/s which implies that already with 2 cores the memory bandwidth would be saturated. Based on the measurements even processors with larger memory bandwidth limits like Intel Xeon E5-2699v4 (22 cores and 76.8 GB/s bandwidth) can be easily saturated.

5. Conclusions and outlook

The paper introduces a new LBM streaming scheme adapted from the A-A pattern scheme for directly addressed grids with a SoA data layout. Additionally, the paper shows that the new streaming scheme is vectorizable by the compiler if combined with simple additional implementation techniques. Based on the measured performance parallel implementations are expected to be memory bandwidth bound which suggests that more computationally complex collision models might benefit from this streaming scheme. Initial tests with complex test cases including complex boundaries have shown that the new algorithm is also applicable to complex geometries. As the algorithm exhibits all the necessary properties for an efficient GPU LBM streaming scheme – being one grid without data dependencies based on a SoA data layout – further research needs to be done to analyze the usability of the algorithm for current GPU hardware.

References

[1] Cockshott P, Renfrew K. SIMD Instruction-sets. London: SIMD Programming Manual for Linux and Windows; 2004. p. 11–22. ISBN 978-1-4471-3862-4.
[2] Maleki S, Gao Y, Garzarán MJ, Wong T, Padua DA. An evaluation of vectorizing compilers. In: Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT, 7; 2011. p. 372–82. doi:10.1109/PACT.2011.68.
[3] Chen S, Doolen GD. Lattice Boltzmann method for fluid flows. Annu Rev Fluid Mech 1998;30(1):329–64.
[4] Krause MJ, Heuveline V. Parallel fluid flow control and optimisation with lattice Boltzmann methods and automatic differentiation. Comput Fluids 2013;80:28–36.
[5] Gaedtke M, Wachter S, Rädle M, Nirschl H, Krause MJ. Application of a lattice Boltzmann method combined with a Smagorinsky turbulence model to spatially resolved heat flux inside a refrigerated vehicle. Comput Math Appl 2018.
[6] Henn T, Thäter G, Dörfler W, Nirschl H, Krause MJ. Parallel dilute particulate flow simulations in the human nasal cavity. Comput Fluids 2016;124:197–207.
[7] Succi S, Benzi R. Lattice Boltzmann equation for quantum mechanics. Physica D 1993;69(3–4):327–32.
[8] Wittmann M, Zeiser T, Hager G, Wellein G. Comparison of different propagation steps for lattice Boltzmann methods. Comput Math Appl 2013;65(6):924–35. doi:10.1016/j.camwa.2012.05.002.
[9] Bailey P, Myre J, Walsh S, Lilja D, Saar M. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In: 2009 International Conference on Parallel Processing. IEEE; 2009. p. 550–7. ISBN 978-1-4244-4961-3. doi:10.1109/ICPP.2009.38.
[10] Geier M, Schönherr M. Esoteric twist: an efficient in-place streaming algorithmus for the lattice Boltzmann method on massively parallel hardware. Computation 2017;5(2):19.

[11] Bhatnagar PL, Gross EP, Krook M. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. Phys.Rev. 1954;94(3):511.
[12] Mattila K, Hyväluoma J, Rossi T, Aspnäs M, Westerholm J. An efficient swap algorithm for the lattice Boltzmann method. Comput Phys Commun 2007;176(3):200–10. doi:10.1016/j.cpc.2006.09.005.
[13] Fietz J, Krause MJ, Schulz C, Sanders P, Heuveline V. Optimized hybrid parallel lattice Boltzmann fluid flow simulations on complex geometries. In: Kaklamanis C, Papatheodorou T, Spirakis PG, editors. Euro-Par 2012 Parallel Processing. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 818–29. ISBN 978-3-642-32820-6. doi:10.1007/978-3-642-32820-6_81.
[14] Mantovani F, Pivanti M, Schifano S, Tripiccione R. Performance issues on many-core processors: a d2q37 lattice Boltzmann scheme as a test-case. Comput Fluids 2013;88:743–52. doi:10.1016/j.compfluid.2013.05.014.
[15] Meng J, Zhang Y. Diffuse reflection boundary condition for high-order lattice Boltzmann models with streaming collision mechanism. J Comput Phys 2014;258:601–12. doi:10.1016/j.jcp.2013.10.057.
[16] Lomont C. Introduction to Intel advanced vector extensions. Intel White Paper2011;1–21.
[17] Krause MJ, Mink A, Trunk R, Klemens F, Maier M-L, Mohrhard M, et al. OpenLB Release 1.2: Open Source Lattice Boltzmann Code URL http://www.openlb.net/download.
[18] Wittmann M, Zeiser T, Hager G, Wellein G. Modeling and analyzing performance for highly optimized propagation steps of the lattice Boltzmann method on sparse lattices. arXiv:14100412 2014.
[19] Feichtinger C, Habich J, Köstler H, Rüde U, Aoki T. Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU–GPU clusters. Parallel Comput 2015;46:1–13.
[20] Zeiser T, Wellein G, Hager G, Donath S, Deserno F, Lammers P, et al. Optimized lattice Boltzmann kernels as testbeds for processor performance. Regional Computing Center of Erlangen (RRZE), Martensstraße 2004;1.