LRnLA Algorithm ConeFold with non-Local Vectorization for LBM Implementation

Anastasia Perepelkina✉[0000–0003–2517–6064] and
Vadim Levchenko[0000-0003-3623-0556]

Keldysh Institute of Applied Mathematics RAS, Moscow, Russia
mogmi@narod.ru, lev@keldysh.ru

Abstract. We have achieved a ~ 0.3 GLUps performance on a 4 core CPU for the D3Q19 Lattice Boltzmann method by taking an advanced time-space decomposition approach. The LRnLA algorithm ConeFold was used with a new non-local mirrored vectorization. The roofline model was used for the performance estimation and parameter choice. There are many expansion possibilities, so the developed kernel may become a foundation for more complex LBM variations.

Keywords: Lattice Boltzmann Method, LRnLA algorithms, Parallel computation.

1 Introduction

One of the reasons for the popularity of the Lattice Boltzmann Method (LBM) [17] for Computational Fluid Dynamics (CFD) is the ease of its efficient computer implementation. However, some issues exist. LBM implementations remain memory-bound, and the vectorization is complicated since misaligned writes or reads require significant overhead. To try to reach the maximum performance efficiency authors vary the data storage method, data layout, propagation (streaming) algorithms. The efficient data synchronization for massively parallel implementations is also in a high demand since CFD problems are radically multiscale.

GPU makes the aforementioned issues more prominent. However, it provides better performance results than CPU in many cases. At the same time, CPU codes remain relevant. CPU codes are more flexible for multiphysics frameworks, as we see in the famous packages such as waLBerla [3] and OpenLB [5]. For large supercomputer simulations CPU has more memory so that the potentially bigger problems may be solved. And since the GPU computers are essentially heterogeneous, efforts are made to offload some work to the CPU kernels [15].

The aim of this work is to break the performance records of CPU implementation with the use of LRnLA algorithms [7]. LRnLA algorithms may be seen as an advanced temporal blocking method. Temporal blocking was previously applied to LBM [12, 4]. Indeed, it seems to grant better parallelization efficiency [19]. For GPU, it was used for host-device and intra-device communications [16]. While the apparent similarity in using the space-time parallelism

exists, LRnLA method is different. Its base lies in analyzing the dependency and influence conoids in the dependency graph, and the optimization is conducted with the account for memory and parallelism hierarchy of the computer. In fact, we disagree with [12] that the blocking method that is presented there is most efficient, and with the idea that 3D blocking is undesired.

In this paper, we show how the LRnLA algorithm ConeFold is built and implemented for LBM on CPU, how its performance may be estimated with the use of the roofline model. This implementation actually gives the performance per node record that surpasses every CPU result we found in the published work.

2 Lattice Boltzmann Method

In LBM, the simulation domain is split into $Nx \times Ny \times Nz$ cubic cells. In each cell, the probability distribution function is known for a set of discrete velocities $\mathbf{c}_{ijk}$. The specific method is denoted by a word like D3Q19, where the first number is the dimensionality of the model and the second number is the number of velocities. Discrete velocities are chosen as vectors that point from the center of the cell to the centers of its neighbors, and a zero velocity. In D3Q27, there is a set of vectors that point to each cell in a $3 \times 3 \times 3$ cube. In D3Q19, the longest vectors of D3Q27 are pruned.

For each velocity the update rule for its Distribution Function (DF) is split into two sub-steps: streaming $f_{ijk}(\mathbf{r}_{ijk}, t + \Delta t) \leftarrow f_{ijk}(\mathbf{r}_{000}, t)$, and collision $f_{ijk} \leftarrow f_{ijk} - (f_{ijk} - f^{eq}_{ijk})/\tau$; $i,\ j,\ k = -1, 0, 1$ for D3Q27. Streaming copies the f_{ijk} from cell with coordinates $\mathbf{r}_{000}$ to the cell with the relative position $\mathbf{r}_{ijk} = \mathbf{r}_{000} + \mathbf{c}_{ijk}\Delta t$, $\mathbf{c}_{ijk} = (i, j, k)$. The collision operates with the DF in the same spatial coordinates. The expression for the equilibrium DF $f^{eq}_{ijk}(\rho, \mathbf{u})$ ($\rho = \sum f_{ijk}$, $\mathbf{u} = \sum \mathbf{c}_{ijk} f_{ijk}$) is taken as the most commonly used second-order polynomial in $\mathbf{u}$ [17] to make the performance comparison easier, but any expression that operates on the data inside one LBM cell may be used in the current implementation.

3 ConeFold Algorithm

3.1 Algorithm as a Decomposition of a Dependency Graph

The core idea in the LRnLA method revolves around the existence of the influence and dependency region in space-time for each cell [8]. The dependencies in an LBM stencil fill a cube, so the data in one cell is influenced by a 4D pyramid in space-time. The whole simulation region may be decomposed in such regions.

That is why we illustrate algorithms as shapes in a dependency graph space with a subdivision rule [9]. A dependency graph consists of nodes (operations) and directed links (data dependencies between operations). A shape covers some nodes, so an algorithm described by this shape should perform all the operations of these nodes. It may be subdivided by planes with one restriction: if there are dependencies directed from one side of the plane to the other, there should not

be any that are directed backward. It ensures that if one part is influenced by the other, there is no backward dependency. This way the algorithm is decomposed into several sub-steps that should be processed in a sequence, that is determined by the direction of the dependencies. If there are no dependencies between the parts, the algorithms may be processed asynchronously. The decomposition continues until the shapes cover only one node. Thus, this definition of the algorithm is recursive, and leads to the existence of non-local asynchronous elements.

For example, the algorithm of parallel implementation of LBM for N_t time steps on 2 nodes with 4 cores and SIMD support.

- Decompose $N_t \times Nx \times Ny \times Nz$ domain (i.e. the whole dependency graph) into N_t flat (in time) layers with size $Nx \times Ny \times Nz$. They have data dependencies pointing upwards, so they need to be processed in sequence. We call algorithms that start with this kind of decomposition 'stepwise' or 'orthodox'.
- Decompose the layers into two rectangles by a vertical (in time) plane $x = const$. They have no data dependencies between each other, so they may be processed in parallel by the two processors.
- Decompose the rectangles into two smaller rectangles by vertical planes $y = const$. They have no data dependencies between each other, so they may be processed in parallel by the 4 cores.
- Decompose the rectangles into separate cells. These are traversed these in any loop sequence, which may be vectorized by the compiler.

This kind of implementation is far from optimal. The loop traversal over large blocks of data in the lowest level of decomposition cause the memory-bound limitation: at each time step all DF data should be loaded and stored in memory. At the multi-core stage cache conflicts may arise, their resolution may lead to overhead. The necessity of using two lattice copies for the propagation scheme arise here. For multi-node parallelism, the data exchange should be performed at each time step, and this becomes the bottleneck for the parallel scaling. The point where this algorithm went wrong is the first subdivision into flat time steps. There are plentiful possibilities of the dependency graph subdivision that lead to better data access locality and do not make the result incorrect.

In LRnLA, the choice of the subdivision planes comes from natural requirements. First, we have planes with $t = const$, the synchronization instants, where data may be visualized or analyzed in any other way. Second, we choose hyperplanes $Ct = x + const$, $Ct = y + const$ and $Ct = z + const$ where C is the discrete information propagation speed. Here, $C = \Delta x/\Delta t$.

The dependency graph for D1Q3 LBM with a 'swap' propagation scheme shown in Fig. 1(a). The subdivision planes in it surround an elementary ConeFold algorithm shape.

$Ct = -x + const$, $Ct = -y + const$, $Ct = -z + const$ hyperplanes may also be considered. This leads to a subdivision into pyramids, diamonds and other shapes to complement them in 4D space-time [7]. Otherwise, all simulation domain may be tiled by just one shape. It has many advantages, and the easier coding is just one of them.

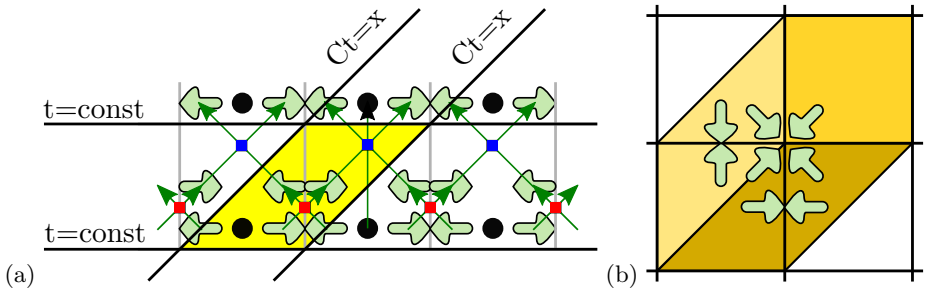

Fig. 1. (a) Dependency graph for LBM with 'swap' propagation scheme. Subdivision planes surround an elementary ConeFold (LRnLA cell). Operations are denoted by square markers. Red: non-local swap. Blue: local swap and collision.(b) The DFs that participate in a non-local swap in an LRnLA cell in D3Q27 case. The figure looks similar in x-y, y-z and z-x axis, the colored area is the CF projection

3.2 Implementation of LBM with ConeFold

The ConeFold (CF) algorithm is implemented with recursive templates in C++.

1D case The level of recursive subdivision of a CF is parametrized by an integer which is called rank. One 1D CF with rank $r = R$ (denoted by $\texttt{CF}\langle \texttt{d} = 1, \texttt{R} \rangle$) is a function call of 4 $\texttt{CF}\langle 1, R-1 \rangle$ in the order which satisfies data dependencies (Fig. 2(a)). On rank $r = 0$ the CF is an elementary update according to the scheme stencil (Fig. 1(a)). It is a portion of the dependency graph that is defined as an LRnLA cell for CF.

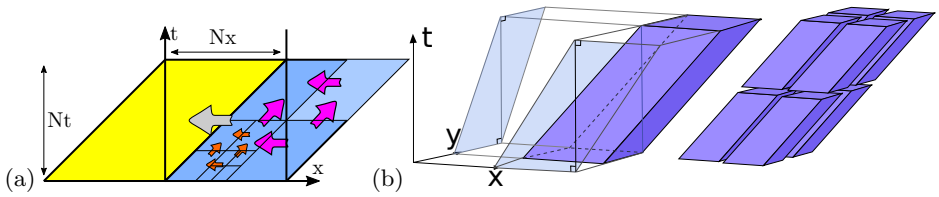

Fig. 2. (a) Two $\texttt{CF}\langle 1, \texttt{MaxRank} \rangle$ cover the domain and are recursively decomposed into smaller CFs. Arrows show data dependencies. (b) ConeFold for d=2.

If the top or bottom base is outside the computation region, the CF is specified as a right or left boundary CF. If $r > 0$ these call one CF inside the domain and two boundary CFs with $r - 1$ and a same type (left or right). If $r = 0$ the boundary condition of the scheme is applied. The periodic boundaries are not possible with this algorithm without additional techniques.

The computation is started with a right boundary $\texttt{CF}\langle 1, \texttt{MaxRank} \rangle$. Assuming the domain has $N_x = 2^{\texttt{MaxRank}}$ cells, after it is finished, the cells evolved to the

number of steps from 1 to $N_t = 2^{\texttt{MaxRank}}$, in a linear progression from left to right. Only the rightmost cell has been updated N_t times. The left boundary $\texttt{CF}\langle 1, \texttt{MaxRank}\rangle$ is required to update all other cells to the same time step.

Recursive d-binary subdivision of the CF algorithm makes it natural to use a Morton Z-curve [10] for data storage. Thus, the data storage cells are organized in a recursive structure, and the indices to the neighboring cells are computed accordingly. Inside one CF, the pointer to the data in its bottom base, and array offset to the data in the projection of its top base are known. Thus, one $\texttt{CF}\langle 1, 0\rangle$ has access to 2 data structure cells.

We need to find a LRnLA cell so that it uses the available data and homogeneously tiles the whole dependency graph of the domain evolution for $2^{\texttt{MaxRank}}$ steps, with a possible exception for boundary conditions.

For example, in D1Q3 the following variation of the propagation scheme would suffice (Fig. 1(a)). Each data storage cell contains the data for f_{-1}, f_0, f_1, there is access to the two adjacent cells: c_0 and c_1. The update rule is:

1. swap f_1 of cell c_0 and f_{-1} of the cell c_1 to the right of it (non-local swap);
2. swap f_1 and f_{-1} of c_1 (local swap).
3. collision in one cell c_1.

Note that before this $\texttt{CF}\langle 1, 0\rangle$ is performed, the f_1 value of c_1 would contain f_{-1} from the cell c_2 to the right of it. This is because the $\texttt{CF}\langle 1, 0\rangle$ had been already performed for cells c_1 and c_2, and its step 1 had swapped f_1 of c_1 and f_{-1} of c_2. This is why c_1 contains the required post-stream data. The local swap may be merged with the collision.

2D, 3D Case The d-dimensional algorithm is constructed by a direct product of 1D algorithms (Fig. 2(b)). There are the following changes.

- $\texttt{CF}\langle d, R\rangle$ is subdivided into 2^{d+1} CFs with R-1.
- There are 3^d types of boundary CFs. It is necessary to specify CFs with $r = 0$ and with $r > 0$ for all cases of boundary: faces, edges, corners. Obviously, a code generator is used for this task.
- $(3^d - 1)$ boundary CFs of the maximum rank are required to progress from one synchronization instant to the next one.
- The $\texttt{CF}\langle d, \texttt{MaxRank}\rangle$ base projection covers $2^{d\cdot\texttt{MaxRank}}$ cells.
- The top base is shifted from the bottom one by $2^{\texttt{MaxRank}}$ cells in d directions.

Note that the dimensionality of the CF is independent of the dimensionality of the model. If $d = 2$ for D3Q27 scheme, one LRnLA cell would be redefined to include the update for Nz LBM cells. In one data storage cell there are 27 N_z sized arrays for f_{ijk}.

Other Stencils The other possible reason to redefine an LRnLA cell is the existence of more extended dependencies since one $\texttt{CF}\langle d, 0\rangle$ has access only to 2^d cells. For example, the LRnLA cell for D1Q5 would update 2 full sets of distribution functions:

– swap f_1 with f_{-1} and f_2 with f_{-2} in c_0 and in c_1 (local swap);
– perform the collision in c_0, c_1;
– swap f_2 from c_0 with f_{-2} of c_1, f_2 from c_1 with f_{-2} of c_2;
– swap f_{-2} of c_1 with f_2 in c_1 and in c_2;
– swap f_1 from c_1 with f_{-1} of c_2, f_1 from c_2 with f_{-1} of c_3;
– swap f_2 from c_1 with f_{-2} of c_2, f_2 from c_2 with f_{-2} of c_3;

Here, local swap for all cells is required in initialization and in output. Some steps may be merged in optimization.

The velocities that are swapped in a 3D case of D3Q27 $\texttt{CF}\langle 3,0\rangle$ are depicted in Fig. 1(b). D3Q19, D3Q15, D3Q7 are devised from this pattern by pruning.

For clarity, we repeat the three terms for cells that are used here:

– LBM cell is the lattice node with the corresponding DFs;
– LRnLA cell is the portion of the dependency graph and consists of a number of operations for an elementary update;
– data storage cell is a set of variables that are organized as an element of a data structure that is used.

Non-local Vectorization There are several possibilities of vectorization in CF.

1. 2D CF may be used for 3D computation, as in [13]. In each cell, $27N_z$ DFs would be stored in a SoA manner, and in one $\texttt{CF}\langle 2,0\rangle$ they would be processed in a loop, that may be vectorized by a compiler or manually. The fact that one axis is detached from the LRnLA decomposition becomes the main issue here, since it brings back the problems of stepwise algorithms to this dimension.
2. The cell of the data structure may be redefined to contain $2\times2\times2$ LBM cell data. The 8 values for each DF in it would be collected in one SIMD vector. This vectorization is local, but requires many reshuffle operations, especially in the collision step. It may be useful to keep this method in mind for possible GPU implementation, but for CPU the performance is limited by the overhead.

In the first case the automatic vectorization is local as well, and the misalignment issues need to be resolved. The non-local vectorization may be implemented in a way that requires less overhead. For vectors of length 4, the cell data for cells $\{i, i+Nz/4, i+2Nz/4, i+3Nz/4\}$ with $i=0,1,..Nz/4-1$ are combined into vectors manually. This way, the domain is folded 4 times, the reshuffle is only required on the folds (Fig. 3).

Here a third method is proposed, which results in a non-local vectorization for 3D CF, and, moreover, fixes the issue of the impossibility of periodic boundaries.

For the explanation of the basic idea let us assume SIMD vector length 2 in D1Q3. Take a domain with $2\cdot2^{\texttt{MaxRank}}$ cells. Pack the data from cell $(2^{\texttt{MaxRank}}-1-i)$ with the data from cell $(2^{\texttt{MaxRank}}+i)$ for $i=0,1,..,(2^{\texttt{MaxRank}}-1)$ into one LRnLA cell. Combine all pairs of values into SIMD vector. Then, construct a CF for the

$2^{\texttt{MR}}$ cells of the resulting data structure. The code for LBM inside the domain stays the same but operates with SIMD vectors instead of scalars.

As a result, for the second value of the pairs, the computation is the same as for the non-vectorized case. For the first value of the pairs, the computation is mirrored. It has no impact on the streaming step. In the collision step, we need to change the sign of the directed macro-values, such as $\mathbf{u}$. The constant SIMD vector $\{-1, 1\}$ is used as a coefficient in the scheme.

Thus, f_1 in the left half of the domain represents propagation to the left. f_1 in the right half of the domain represents propagation to the right.

The right boundary $\texttt{CF}\langle 3, 0\rangle$ links the cell at $(2^{\texttt{MaxRank}} - 1)$ with the cell at $2^{\texttt{MaxRank}}$. The swap is performed between two values in SIMD vector for f_1. The left boundary $\texttt{CF}\langle 3, 0\rangle$ links the cell 0 with cell $(2 \cdot 2^{\texttt{MaxRank}} - 1)$ and the swap is performed between values in f_{-1}.

Thus, this is a non-local mirrored vectorization (Fig. 3).

To enable the use of SIMD vectors with size 4 or 8 the domain is either mirrored in another axis, or has more reflected copies along the same axis. Here we present simulation with SIMD length 8 for single precision. With the use of AVX512 the same may be done for double precision.

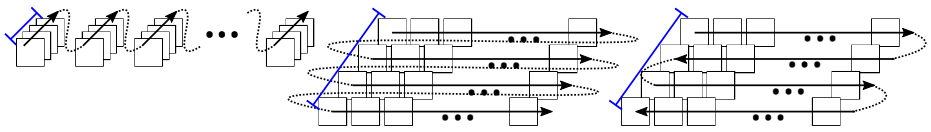

Fig. 3. Local, non-local, and non-local mirrored vectorization. Arrow shows the coordinate axis direction, blue segment shows SIMD vector pack. Vector length is 4.

Parallel Algorithm The TLP parallelization with CF is implemented with the TorreFold LRnLA algorithm [14]. We define an integer parameter `nLArank` (non-Locally Asynchronous rank). The TorreFold shape is similar to the shape of the CF with $r = \texttt{MaxRank}$, but the decomposition rule is different (Fig. 4). It is decomposed into $2^{(d+1)\texttt{nLArank}}$ CFs. Some of these are independent and may be processed by different threads. The asynchronous CFs are shown in a 1D case in Fig. 4. In 3D there is also asynchrony in an x–y–z–t diagonal cross-section.

The specific implementation may differ, the following is used in the current code. The CFs which stand on top of each other are collected in a tower (ConeTorre). All ConeTorres are distributed between threads in the Z-curve order. The dependencies between them are ensured by semaphores. Each tower has d semaphores. Initially, all semaphores are locked. The thread is assigned to some ConeTorre, and it processes CFs with $r = \texttt{MR} - \texttt{nLArank}$ in it one by one. Before starting a CF, the thread waits for one semaphore in each of the towers that are influencing it. After a thread finishes one CF it unlocks all its semaphores.

Some ConeTorres start outside of the domain, but the CFs that are outside are just skipped.

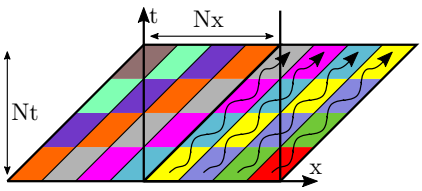

Fig. 4. TorreFold algorithm in 1D. Same color CFs may be processed in parallel. Threads (curled lines) process CFs in one ConeTorre.

3.3 Roofline Model

The roofline model [18] helps to analyze bottlenecks in the implementation on a given computer. It does not account for all possible code capabilities and hardware limitations, but still, it is valuable for its simplicity.

For example, the memory-bound slope is usually shown for the RAM memory throughput. However, the memory hierarchy allows, in some cases, to break this ceiling. We propose to take into the account the caching ability of the hardware by a divide-and-conquer approach [8]. If a task A consists of N similar sub-tasks $B1$, $B2$, ..., BN, and all task A data is small enough to fit some level of cache, then the memory bound limit of tasks B is determined by the throughput of a higher level of cache. However, if we assume that each sub-task is carried out one-by-one, the load on the memory throughput is not less than the sum of the data required by each B task individually.

This conforms to the recursive definition of the algorithm as a decomposition of a task into subtasks. For a given algorithm we need to estimate the arithmetic intensity and the amount of data, that needs to be loaded into the cache.

The $\texttt{CF}\langle d, r\rangle$ consists of $2^{(d+1)r}$ LRnLA cells, so it has $O(r,d) = o2^{(d+1)r}$ operations, where o is the number of operations in an LRnLA cell. During its execution, the total amount of data loaded is $L(r,d) = N^d + N((N+1)^d - N^d)$ data storage cells, where $N = 2^r$. The amount of data stored is $S(r,d) = (N-1)^d + N \cdot N_T(N^d - (N-1)^d)$.

The arithmetic intensity is $O(r,d)/(L(r,d) + S(r,d))$. The cached data size is $L(r,d)$.

In the presented code the recursive subdivision is as follows:

- a 4D cube that covers $2^{3\texttt{MaxRank}}$ LBM cells for $2^{\texttt{MaxRank}}$ updates is decomposed into $2^{3(\texttt{nLArank}+1)}$ ConeTorres;
- ConeTorres are decomposed into $2^{\texttt{nLArank}}$ CFs with rank $r^* = \texttt{MaxRank} - \texttt{nLArank}$;

– CFs are recursively decomposed into the same shapes of smaller rank until the LRnLA cell is reached.

In this estimation, the boundary effects are not considered.

The Roofline for our implementation on Intel Core i5-6400 is shown in Fig. 5. The arithmetic intensity increases with r. However, the algorithm performance is limited by the roofline of all its smaller parts. Several arrows from the higher levels of decomposition to the lower levels are plotted one by one from right to left. If an arrow reaches memory bound slope, arrows to the left of it are not allowed to be higher than this arrow. The color of the arrow shows the color of the roofline that defines its memory throughput limit.

`nLArank` is set equal to 4. This choice is determined by the roofline: the advantages of the $(d+1)$-binary recursive subdivision are evident only for ranks smaller than $\texttt{MaxRank} - 4 = 4$.

With this model, we can estimate the disadvantage of the lower dimensionality of LRnLA decomposition. If a $\texttt{CF}\langle 2, 0\rangle$ contains N_z cell updates, it, and all the CFs of higher rank, require more memory, and may not be localized in higher levels of cache. For large N_z the arrows for $R = 0, 1, 2$ would all be limited by the RAM roofline. This goes in contrast with the guidelines from [12], and also suggests that the non-local vectorization without mirroring the domain decreases the efficiency.

The local vectorization with $d = 3$ would make the code compute-bound according to this roofline model. Still, we do not see an implementation solution for it without significant overhead.

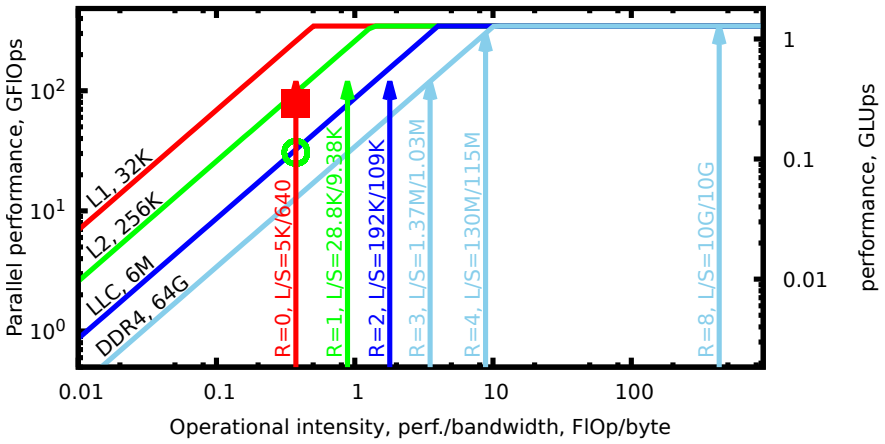

Fig. 5. The roofline for Intel Xeon i5-6400. The red marker shows the highest performance achieved in our implementation, the green marker shows the maximum achieved one thread performance.

3.4 Performance Results

The described algorithm was implemented in code with the use of C++ (gcc compiler version 6.3) with parallelisation with POSIX threads. Code generation tools are made with Python3.6. Data visualization for verification of results uses aiwlib library [6]. The performance scaling of D3Q19 LBM implementation was verified on the Intel Core i5-4440 (2ch 32GB DDR3 RAM) and the Intel Core i5-6400 CPU (2ch 64GB DDR4 RAM).

On Fig. 6 the cube shaped domain is scaled up to the maximum size that still fits RAM memory. Starting from $\sim$ 2 MB data size the performance of about 25% from the peak is reached. Despite the fact that low-budget CPU is used, the achieved performance of > 0.25 GLUps (billions of Lattice Update per second) may even be comparable to some GPU implementations. Note that the performance rises with the increase of the data size. With stepwise approaches without any kind of temporal blocking the performance drops each time the data size exceeds some level of cache. For comparison, in [15] CPU kernels for 12 core CPU Xeon E5-2690v3 reached $\sim$ 0.22 GLUps performance, while GPU kernels reach 3 GLUps. Walberla framework reaches $\sim$ 0.08 GLUps on 8 core Intel Xeon E5-2680 [3].

We have performed performance tests on one node of the K60 cluster [1] (Intel Xeon E5-2690 v4, 8ch 256GB DDR4 RAM), and achieved the performance up to 1.2 GLUps. Although the strong scaling efficiency is unsatisfactory in this case, we see how the result may be improved. The LRnLA algorithm for NUMA architecture is developed [20], but not implemented in the current code. Nevertheless, the result of our CPU code for one node of K60 cluster is better than the CPU version in [15] for one node of the Piz Daint supercomputer, and $\sim$ 42% of its GPU version.

On Fig. 7 the strong parallel scaling results are presented. The reason for the low scaling efficiency ($\sim$ 72%) is the significant influence of the boundaries on the TorreFold algorithm. This may be improved by devising another implementation pattern. However, it may be reasonable to leave the idle resources for background tasks (i.e. MPI transfers) in larger codes.

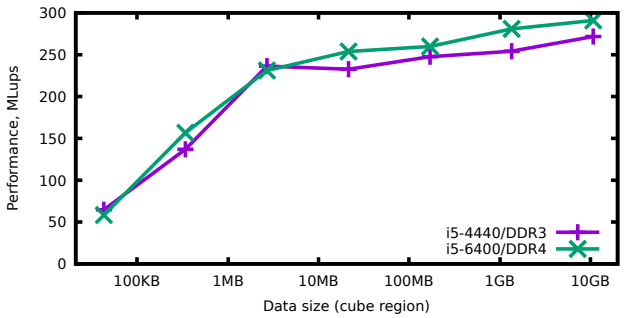

Fig. 6. Performance dependency on the data size.

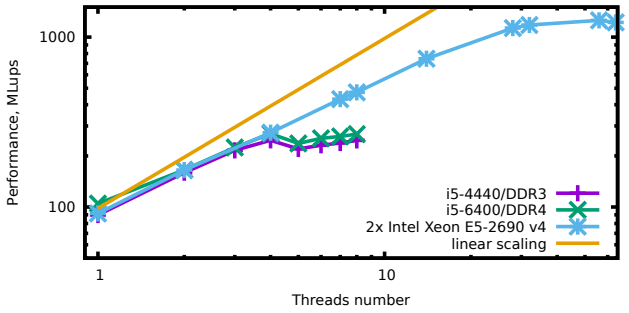

Fig. 7. Performance dependency on the number of POSIX threads.

4 Conclusion

We have used the LRnLA algorithm ConeFold to make a high-performance LBM simulation code. This approach optimizes the space-time traversal to take advantage of the memory hierarchy and all available levels of parallelism. The algorithm has been augmented by the non-local vectorization method, which not only increases the performance but also fixes the inability of performing simulation in periodic domains.

The LBM method proved to be simple for implementation, and the propagation method that conforms with the ConeFold decomposition has been found. It is interesting that the most memory-efficient propagation schemes of the stepwise codes [2, 11] share the similarity with the method that arose naturally from the scheme dependencies and data access locality requirements.

The roofline model of the target CPU was built and the estimation of the code limits was aided by the LRnLA theory of algorithm construction. From this analysis, we see the necessity of using 4D localization in the algorithm and choose the parameters for parallelization.

We have measured the performance of the code on some low-cost CPU, and the results that were expected from the roofline construction were achieved. Namely, we have achieved > 0.100 GLUps performance per core, ~ 0.3 GLUps per CPU, which exceeds the results that were found in the published work.

This result proves the advantages of the LRnLA approach. Namely, the algorithmic optimization is more important than the low-level considerations. The algorithms are built independent of the numerical method and hardware but are successfully adapted by taking account of dependency propagation speed and operations count from the numerical method side and the hierarchy of memory and parallelism from the hardware side.

As a further study, we aim to apply the method to more complex variations of the LBM method, like free surface LBM or double distribution function methods. Further, the optimizations of memory managing for sparse geometries and non-uniform grids are apparent.

The work is partially supported by the Russian Science Foundation (project #18-71-10004).

References

1. Computational resources of Keldysh Institute of Applied Mathematics RAS, www.kiam.ru
2. Geier, M., Schönherr, M.: Esoteric twist: An efficient in-place streaming algorithmus for the lattice boltzmann method on massively parallel hardware. Computation 5(2), 19 (2017)
3. Godenschwager, C., Schornbaum, F., Bauer, M., Köstler, H., Rüde, U.: A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. p. 35. ACM (2013)
4. 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)
5. Heuveline, V., Latt, J.: The OpenLB project: an open source and object oriented implementation of lattice Boltzmann methods. International Journal of Modern Physics C 18(04), 627–634 (2007)
6. Ivanov, A., Khilkov, S.: Aiwlib library as the instrument for creating numerical modeling applications. Scientific Visualization 10(1), 110–127 (2018)
7. Levchenko, V.D.: Asynchronous parallel algorithms as a way to archive effectiveness of computations (in Russian). J. of Inf. Tech. and Comp. Systems (1), 68 (2005)
8. Levchenko, V.D., Perepelkina, A.Y.: Locally recursive non-locally asynchronous algorithms for stencil computation. Lobachevskii Journal of Mathematics 39(4), 552–561 (2018)
9. Levchenko, V.D., Perepelkina, A.Y., Zakirov, A.V.: DiamondTorre algorithm for high-performance wave modeling. Computation 4(3), 29 (2016)
10. Morton, G.M.: A computer oriented geodetic data base and a new technique in file sequencing (1966)
11. Neumann, P., Bungartz, H.J., Mehl, M., Neckel, T., Weinzierl, T.: A coupled approach for fluid dynamic problems using the PDE framework peano. Communications in Computational Physics 12(1), 65–84 (2012)
12. Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: High Performance Computing, Networking, Storage and Analysis (SC). pp. 1–13. IEEE (2010)
13. Perepelkina, A.Y., Levchenko, V.D., Goryachev, I.A.: Implementation of the kinetic plasma code with locally recursive non-locally asynchronous algorithms. In: Journal of Physics: Conference Series. vol. 510, p. 012042. IOP Publishing (2014)
14. Perepelkina, A.: 3D3V kinetic code for simulation of magnetized plasma (in Russian). Ph.D. thesis, Keldysh Institute of Applied Mathematics RAS, Moscow (2015)
15. Riesinger, C., Bakhtiari, A., Schreiber, M., Neumann, P., Bungartz, H.J.: A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters. Computation 5(4), 48 (2017)
16. Shimokawabe, T., Endo, T., Onodera, N., Aoki, T.: A stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers. In: Cluster Computing (CLUSTER). pp. 525–529. IEEE (2017)

17. Succi, S.: The lattice Boltzmann equation: for fluid dynamics and beyond. Oxford university press (2001)
18. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM 52(4), 65–76 (2009)
19. Wittmann, M.: Hardware-effiziente, hochparallele Implementierungen von Lattice-Boltzmann-Verfahren für komplexe Geometrien (in German). Ph.D. thesis, Friedrich-Alexander-Universitt Erlangen-Nrnberg (2016)
20. Zakirov, A.V., Levchenko, V.D.: The code for effective 3D modeling of electormagnetic wavesevolution in actual electrodynamics problems. Keldysh Institute Preprints (28) (2009)