Commun. Comput. Phys.
doi: 10.4208/cicp.OA-2022-0055

Vol. **33**, No. 1, pp. 214-244
January 2023

Heterogeneous LBM Simulation Code with LRnLA Algorithms

Vadim Levchenko[1,*] and Anastasia Perepelkina

Keldysh Institute of Applied Mathematics RAS.

Received 19 February 2022; Accepted (in revised version) 24 August 2022

Abstract. A design of a new heterogeneous code for LBM simulations is proposed. By heterogeneous computing we mean a collaborative computation on CPU and GPU, which is characterized by the following features: the data is distributed between CPU and GPU memory spaces taking advantage of both parallel hierarchies; the capabilities of both SIMT GPU and SIMD GPU parallelization are used for calculations; the algorithms in use efficiently conceal the CPU-GPU data exchange; the subdivision of the computing task is performed with an account for the strong points of both processing units: high performance of GPU, low latency, and advanced memory hierarchy of CPU. This code is a continuation of our work in the development of LRnLA codes for LBM. Previous LRnLA codes had good efficiency both for CPU and GPU computing, and allowed GPU simulation performed on data stored in CPU RAM without performance loss on CPU-GPU data transfer. In the new code, we use methods and instruments that can be flexibly adapted to GPU and CPU instruction sets. We present the theoretical study of the performance of the proposed code and suggest implementation techniques. The bottlenecks are identified. As a result, we conclude that larger problems can be simulated with higher efficiency in the heterogeneous system.

AMS subject classifications: 65Y05, 65Y20, 65-04, 76-10

Key words: LBM, Roofline, memory-bound, GPU, LRnLA.

1 Introduction

In the field of high-performance computational fluid dynamics (CFD), the Lattice Boltzmann Method (LBM) [1, 2] is an extremely popular method. It was proposed as a development of Lattice Gas Automata methods [3], and achieved worldwide recognition after several classic publications [4, 5]. From its early days, researchers predicted the power of the method to simulate extreme-scale problems. This power is in the simplicity of formulation of the method.

*Corresponding author. *Email addresses:* lev@keldysh.ru (V. Levchenko), mogmi@narod.ru (A. Perepelkina)

LBM has been implemented on many cutting-edge supercomputers [6–9]. There exist impressive applied simulations, such as wind modeling [10], flow simulation around the skeletal structure of a depth sponge [11], simulation of cerebrovascular blood flow [12], automotive simulation [13].

1.1 Heterogeneous computing model

In practice, usefulness of a method depends on the performance of its code implementation. Higher performance requires efficient use of computer hardware. Modern computers, as a rule, contain several computing devices with different architectures for hardware and for software (Instruction Set Architecture — ISA). Let us discuss the problem of developing an LBM code that uses all available computing power with the most efficiency. For such complex hybrid systems, one has to motivate the choice of and adequate (1) computing model, (2) algorithms, (3) implementation tools.

In systems with several processing devices, one of them is the main processor (CPU, host), and at least one is considered to be a coprocessor, which is dedicated to accelerating specific tasks. In this work, GPUs are taken as coprocessors. Considering the computing model, one may treat coprocessor as an extension of the processor. The coding tools in this environment treat the hardware uniformly, hiding the heterogeneity from the user with a common API (Application Programming Interface), such as OneAPI.

This method had its success in the previous generation. In 1980s, coprocessors such as Intel 8087, Weitek X167 were used to speed up floating point operations. Their architectures were extensions of ISA x86. Later, such coprocessors became integrated into mainstream CPU, and their ISAs was integrated into CPU ISA. Now, SIMD extensions for handling vector data (SSE/AVX2/AVX512) and matrix data (AMX) are also integrated in CPU. At the same time, these extensions remain optional. They are used for acceleration of some specific computing tasks. From this perspective, state-of-the art SIMD tools can be seen as a stage of coprocessor evolution. Their use is available through specific methods, compiled into libraries, and the interface can be obscured with compiler options or programming language extensions. The common tools for SIMD extensions of ISA x86+ include intrinsic methods, vectorized data types and vectorized operations implemented as special objects, extensions of compilers and programming language, loop auto-vectorization with OpenMP or optimizing compilers.

In our research of CPU/GPU heterogeneous computing models, we choose to avoid this evolution path. Its advantage is the fact that software code can be easily made universal for systems with and without coprocessors. Among LBM implementations, this path was followed in [14]. As a consequence, the software that was developed before introduction of coprocessors can be effortlessly adapted to new architectures. This is promoted in the Intel OneAPI standard [15]. On the other hand, GPU devices have their own address space and memory hierarchy, and memory bus with CPU RAM often presents itself as a performance bottleneck. If the fact is hidden from the programmer and ignored, computation efficiency often decreases. With other computing models, it can be taken

advantage of. That is why we prefer to use CUDA instead of unified directives such as OpenACC.

The second wide-spread computing model is based in the parallel computing platform CUDA [16] and OpenCL [17]. Its foundation is SIMT parallelism. This concept is different both from classical CPU parallelism (threads/processes) and from CPU vectorization in SIMD. Therefore, many common CPU programming patterns can not be transferred to GPU efficiently. At this cost, CUDA provides its users all means to exploit all levels of GPU parallelism and memory hierarchy. In the CUDA programming environment, there are sufficient tools for efficient implementation of stencil computation in 3D Cartesian grids. Thus, even the earliest CUDA implementations of LBM has shown successful acceleration in comparison with CPU codes [18, 19]. This trend continues in the recent works [20–23].

In the computing model popularized by nVidia, the role of CPU is limited to the role of a control unit. This makes its SIMD-extended computing power redundant. As a result, the most advanced GPU workstations, such as nVidia DGX, contain heavy nodes with 8-16 GPU for each 1-2 CPU. The recent exaFlOps-scale supercomputers with AMD GPU, such as Frontier, have similar architecture [24]. In such systems, the CPU-GPU bottleneck is circumvented by introduction of direct inter-GPU links, such as nVidia NVlink, AMD Infinity Fabric, or through a commutator. The communication may not involve CPU at all, and that makes heterogeneous computation even more GPU-centered. Some relevant LBM codes for hybrid supercomputer [10, 25] are made according to this model, while CPU is used for data management.

Thus, of the two models that we discussed here, the first one (uniform code for processor and device) does not provide enough control on parallelism and memory transfer, and the second one does not take advantage of the CPU SIMD extensions. Therefore, both can not be used in the search for the most efficient heterogeneous code implementations. Here, we aim to develop a model for heterogeneous computing in the CPU/GPU hybrid hardware with the following essential features: the data is distributed between CPU and GPU address spaces, and they are processed while taking full advantage of GPU SIMT and CPU SIMD capabilities.

LBM codes where both CPU and GPU are used for floating point computations have been developed, where CPU handles a part of LBM calculation [21, 26, 27] or some additional physics [28].

1.2 Memory wall problem

The main difficulty on the way to this goal is the so-called problem of 'memory wall'. The problem for fluid simulations is well illustrated by the Roofline model [29]. Roofline is the graph of theoretical performance limit in the axes of performance vs arithmetic intensity (AI). Arithmetic intensity of an computing task is one measure of its locality; it is defined as operations performed per byte of data throughput. One performance limit is the peak computing performance in FlOp/s (floating point operations per second)

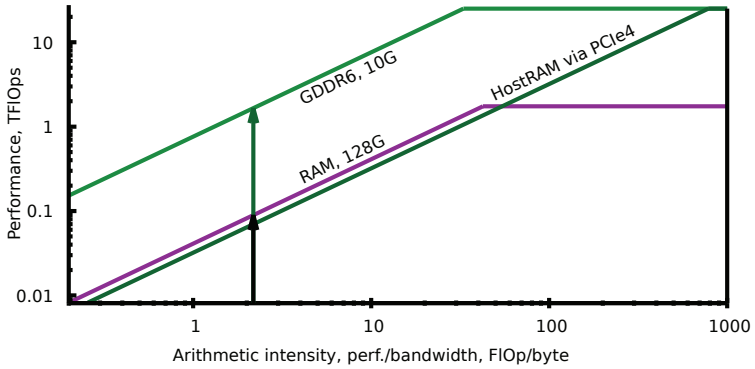


Figure 1: Roofline model for LBM (Section 2.1) on AMD Ryzen R9 5950X and GPU nVidia RTX 3080.

(horizontal line). The second one is computed as memory throughput times AI (slanted line). With less AI, the performance is limited by memory bandwidth. Such problems are called memory-bound. Stencil schemes are often memory-bound [29], as are many common LBM variations. Computationally heavy collision operators [30, 31] can make LBM codes compute-bound, but here we study the classic formulations. See Fig. 1 for a sample Roofline for the LBM problem studied in this paper (Section 2.1).

The value of the compute-bound peak is different whether parallelization methods are used or not. This value, however, does not matter if the problem is severely memory-bound. Therefore, to take advantage of SIMT/SIMD, one has to choose implementation algorithms with higher AI. Moreover, the memory-bound peak of the cache levels is different. As an example, in GPU, the acceleration in comparison to CPU codes is visible when the data is localized in the GPU memory. This is the reason why GPU accelerated simulations are often limited in size: if the data is localized in CPU RAM and sent through PCI-e, the performance peak is much less (Fig. 1).

The decrease in the ratio of memory bandwidth performance to peak computing performance is a trend in computer hardware evolution, thus, the memory wall problem is relevant for the future of simulation methods. Data localization is important for both computing models discussed in the beginning of this section.

1.3 Algorithms

In regards to minimizing the load on memory bandwidth, it is often seen as optimal to have one load and one store operation per one value update (according to the numerical scheme) per one time step [32]. This goal is stated however under the assumption of an implementation, which we refer to as *stepwise* or traditional. That is, there is an outer loop over time steps; all mesh data is updated at least once before the update for a second time step is started anywhere. Let us note that given data from a subdomain of a simulation task, one can update all nodes, then all nodes in a smaller area (with a halo of a stencil

size), and then all nodes in even smaller area, and so on. This way, the time update on a gradually decreasing subset of data can be performed without reading additional data.

We refer to non-stepwise implementations as *temporal blocking* [33–38] in this text, even though there exist different terms for similar notions in literature, such as polyhedral optimizations [39, 40], loop tiling, time skewing, wavefront blocking [41–45]. LBM codes with temporal blocking include [10, 34, 35, 45–47]

It appears that temporal blocking is the only method to obtain sufficient locality to implement efficient heterogeneous CFD codes. A prominent LBM code where temporal blocking allowed to overcome CPU-GPU communication bottleneck is described in [10]. LRnLA (Locally Recursive non-Locally Asynchronous) algorithms [48] can handle both temporal blocking and multi-level parallelism in a unified theoretical framework. With LRnLA algorithm construction, the localization of sub-tasks in the higher memory levels is possible for all levels of memory hierarchy [48], and all levels of parallelism are accounted for [49]. LRnLA algorithms were previously developed for the LBM method. With LRnLA implementations of D3Q19 LBGK (LBM with BGK collision term [4]), performance records were obtained on CPU and GPU. In [50] the obtained performance for single precision D3Q19 LBGK is up to 7.5 billion lattice cell updates per second (GLUps) on a single TeslaV100-PCIe GPU, and up to 10 GLUps on a single RTX 3090. In [51], the maximum obtained performance is 1 GLUps on one Ryzen R9 3900X processor. In all these cases, the performance is greater than the theoretical memory-bound peak of D3Q19 LBGK implemented with stepwise algorithms. The first LRnLA LBM code in which CPU was used to store data is reported in [52]. CPU computing power, however, was not used, but CPU-GPU communication bottleneck was overcome and data were stored in CPU RAM.

Therefore, adding to the requirements stated in the end of Section 1.1, we aim for a heterogeneous code where the communication bottlenecks are avoided by the use of advanced algorithms for data localization. Naturally, in the decomposition of a simulation task into sub-tasks, one should take account of the strength and weaknesses of CPU and GPU: GPU shows higher parallel performance, and CPU has advanced cache hierarchy and low computing latency.

In Section 2, we compile the required information from the previous works of LRnLA algorithm developments for LBM. The section is concluded (Section 2.6) with the proposition of a new heterogeneous code design. The relevance of such design is discussed in Section 3.1 on the basis of the Roofline performance analysis. The implementation techniques for the introduced algorithms are proposed in Section 4.

2 LRnLA algorithm construction

2.1 LBM

In classical LBM, fluid dynamics is represented by discrete distribution function values on a Cartesian mesh. There are Q values f_q (PDF — particle distribution function) per

node. LBM is a two-step numerical scheme. One of these steps is a local collision operation at node $\boldsymbol{x}_i$ at the discrete time instant t^k:

$$f_q(\boldsymbol{x}_i,t^k)=\Omega_q(f_1^*(\boldsymbol{x}_i,t^k),\cdots,f_Q^*(\boldsymbol{x}_i,t^k)),\quad q=1,\cdots,Q. \tag{2.1}$$

Here Ω_q is a collision operator. For example,

$$\Omega_q(f_1^*(\boldsymbol{x}_i,t^k),\cdots,f_Q^*(\boldsymbol{x}_i,t^k))=-\frac{f_q^*-f_q^{eq}}{\tau},$$

where τ is a relaxation parameter which controls fluid viscosity and f_q^{eq} is a weighted discrete approximation of the equilibrium function evaluated at density $\rho=\sum_{q=1}^{Q}f_q^*$, momentum density $\rho\boldsymbol{u}=\sum_{q=1}^{Q}f_q^*\boldsymbol{c}_q$. LBM with this collision operator is also called LBGK. We use the basic second-order variation of the equilibrium [1, 2]. The second step of LBM is the streaming step, which consists of Q separate transfers of f_q values in the $\boldsymbol{c}_q$ direction from each node to its neighbors.

$$f_q^*(\boldsymbol{x}_i+\boldsymbol{c}_q,t^k+1)=f_q(\boldsymbol{x}_i,t^k),\quad q=1,\cdots,Q. \tag{2.2}$$

In this paper the vectors $\boldsymbol{c}_q$ are taken from one or more of the following shells:

$$\begin{aligned}
&\text{shell 0, 1 point:} && (0,0,0);\\
&\text{shell 1, 6 points:} && (\pm1,0,0),(0,\pm1,0),(0,0,\pm1);\\
&\text{shell 2, 12 points:} && (\pm1,\pm1,0),(\pm1,0,\pm1),(0,\pm1,\pm1);\\
&\text{shell 3, 8 points:} && (\pm1,\pm1,\pm1).
\end{aligned} \tag{2.3}$$

2.2 Algorithm construction

Modern computers are among the most complex achievements of scientific progress, thus it is hard to find any specialist who understands the whole picture (including hardware, compilers, programming, numerical aspects of the method) enough to make an ideal solution to the high-performance implementation problem.

The LRnLA method provides a model of algorithm construction which makes writing temporal blocking enabled codes convenient, and at the same time gives control over hierarchical parallelism and data localization.

Let us review the basics of algorithm construction in the LRnLA method [48]. A dependency graph (DG) is an acyclic unidirectional graph. Its nodes represent operations, and the links are data transfers: the outgoing links are the results of the operations, the incoming links are arguments. LRnLA algorithm is defined through (1) a shape in the dependency graph space; (2) a rule of subdivision of the shape. The shape covers some of the dependency graph nodes and corresponds to a task of execution of these nodes. The subdivision of a shape corresponds to subdivision of the tasks into sub-tasks. The

dependencies across every subdivision plane have to be unilateral. The shapes that appear after subdivision either have data dependencies between them or not; in the latter case they can be performed asynchronously. The subdivision starts from a task of updating of all nodes in the simulation for N_T time steps. The subdivision is performed several times recursively, until one sub-task is an elementary scheme update.

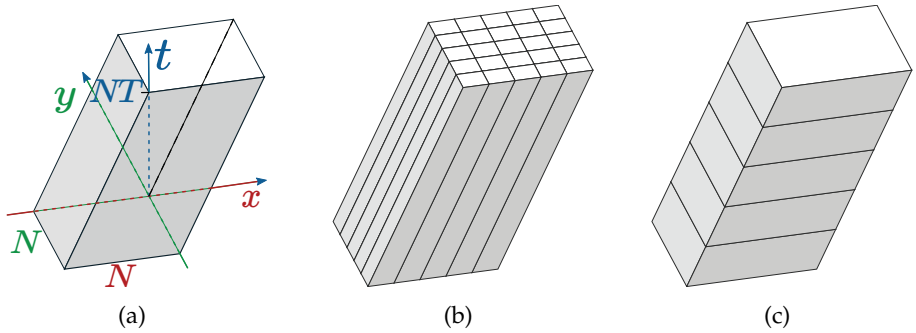
In LBM, the operation of collision in node x_j at time t^k can be placed in the $\{\boldsymbol{x}_j, t^k\}$ position in the 1T3D (3D and time) space of the dependency graph. The streaming operation can be positioned anywhere on the link between two collision nodes [50]. One full LBM update for a node is a set of one collision node and Q streaming operations to or from it, depending on the streaming pattern [50, 53, 54].

Let us take a rectangular grid with size $2N$ along each of the three coordinate axis directions, and define a task to perform the LBM update in all nodes N_T times. The dependency graph of this task fits in a rectangular box with size $2N \times 2N \times 2N \times N_T$; it is the initial task. Let us subdivide the task into sub-tasks of one LBM update for all nodes on a time layer. This corresponds to subdividing the rectangular box into flat boxes with size $2N \times 2N \times 2N \times 1$. The subdivision into flat $2N \times 2N \times 2N \times 1$ is a manifestation of the existence of an outer loop in time iterations in the majority of the simulation codes. This kind of algorithms are called *stepwise* in this paper. Here, we refer to *temporal blocking* as any non-stepwise kind of initial subdivision of the task. It is possible if the dependencies are local, i.e. the stencil has a finite size much smaller than $2N$. In this paper, the length of stencil dependencies is assumed to be equal to 1.

Let us state the desirable features of the subdivision. We desire higher operational intensity to conceal communication bottlenecks. Furthermore, it is convenient for a subdivision to use similar shapes that tile the whole 1T3D domain. The shapes should have adjustable parameters, so as to control localization in memory and degree of parallelism. The number of levels of recursive subdivision is flexible in theory. In practice, the subdivision has to correspond to the hardware architecture. It is reasonable to subdivide into task with the aim of either localization of a task in the faster memory level, or with the aim of distributing computations between parallel processes. ConeTorre and TorreFold algorithms are convenient for these purposes, and were successfully used with LBM on CPU an GPU before. That is why they are used here for a 3D heterogenous LBM code.

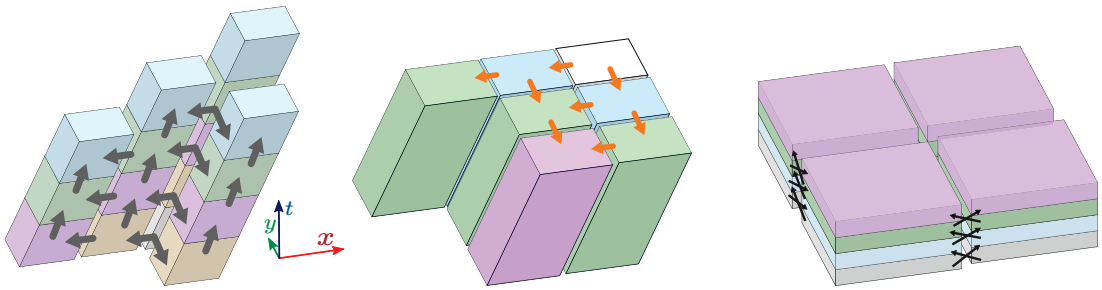
The `ConeTorre<L,NT>` shape is a prism in 1T3D (Fig. 2(a,c)). The bases of the prism are 3D cubes with size L mesh steps along each coordinate direction. The upper cube is shifted by $\{NT, NT, NT, NT\}$ in relation to the lower cube. It is subdivided with $t = const$ planes into `ConeTorre<L,nt>` algorithms, $n_t < N_T$, which have to be executed in a sequence. If $n_t = 1$, the subdivision is referred to as stepwise, since there exists a loop over time in the `ConeTorre<L,NT>` sub-task. The advantage of the `ConeTorre<L,NT>` is the potential for data reuse: at each iteration of the time loop, the data that follows the cube gnomon[†] are loaded and saved. With larger L, the amount of data in the gnomon is less than the data in the volume of the cube. Thus, the AI is high.

[†]In Euclidean geometry, a gnomon is the part of a shape that remains after a similar shape has been taken away from one of its corners.



(a) (b) (c)

Figure 2: `ConeTorre<N,NT>` shape ($N{=}N_T$) and its subdivision, projected onto 1T2D. TorreFold (b), ConeTorre (c).



(a) (b) (c)

Figure 3: Dependencies in a TorreFold: between ConeTorres (b), and between stages of ConeTorres (a). Shapes of the same color are independent. In (c), the traditional stepwise parallel decomposition is illustrated for comparison

The `TorreFold<L,NT>` algorithm has the same shape, but the subdivision is different (Fig. 2(b)). The both base cubes are subdivided into smaller cubes, and, in the result, several `ConeTorre<M,NT>` shapes are obtained, $M < L$. There are asynchronous `ConeTorre<M,NT>` sub-tasks among the shapes that are obtained in this subdivision (Fig. 3). The `TorreFold<L,NT>` sub-task includes implementation of a parallel execution and synchronization between `ConeTorre<M,NT>` sub-tasks, or between time iterations inside them.

The localization abilities of TorreFold and ConeTorre are further illustrated in Section 3.1.

In contrast to the parallel stepwise codes, the data exchange in TorreFold is only from right to left and from bottom to top (Fig. 3).

One more subdivision technique is relevant to the current work, which concerns the processing of the boundary condition. The initial task is not a prism, but a 1T3D rectangle, and this does not present an issue: empty operations can be added on the other side of the boundary, and the boundary nodes represent operations which correspond to the boundary condition. The boundary condition is assumed to have local dependencies as

well. However, one very common boundary condition can not be implemented this way: the periodic boundary. With it, the nodes on one side of the domain depend on the data produced on the other side of the domain. The use of wavefront-type temporal blocking is impossible without additional considerations.

To solve it, smashing [55] and folding [56] methods are used. The positioning of the DG operations in the 1TdD space is modified. The DG is folded in half in each coordinate direction. This way, the nodes in $x=N-j$ and $N+j-1$ are both positioned in $N-j$, $j=0,\cdots,N$. The points $x=0$ and $x=2N-1$ are both position in $x=0$. The boundaries of such dependency graph are at $x=0$ and $x=N$. Similar folding takes place in y and z, so that 8 nodes are superimposed onto each other in each node of a $N\times N\times N\times N_T$ rectangular domain. In the new DG, the dependencies of the periodic boundary are local, and `TorreFold<N,NT>` subdivision is applied.

This transform invert the coordinate directions in the mirrored sub-domains. This results in the minimal changes in the result interpretation, and no changes in the collision operation are required. The boundary nodes contain operations that handle transition of the streamed PDF into the mirrored sub-domain.

Since 8 operations which are non-local (with exception of the boundary) are now placed in the same sub-task with any kind of DG subdivision, the most suitable parallelization method for their execution is vectorization.

The algorithms which are constructed with LRnLA subdivision on the folded domain are denoted ReFold.

2.3 Related work

The algorithms may be understood as a generalization of wavefront-type temporal blocking. Wavefront was introduced for loop optimization in [44]. As a loop optimization method, it had entered modern compiler optimization techniques. Wavefront loop traversal is still used and explicitly coded in modern simulation codes [45]. ConeTorre is a one of the proper many-dimensional generalization of the wavefront. Without ConeTorre, in a 3D simulation, if the common wavefront optimization is used in the nested loops in x and t, the resulting sub-task in the 1T3D domain is a prism with a base of at least $1\times N_y\times N_z$. For large problems, such base can not be localized on the higher memory levels. Thus, ConeTorre with a base of L^3 cells has a more compact base, and L is a parameter that can be adjusted to the hardware.

In the area of temporal blocking, there are two more popular approaches: decomposition into pyramids/diamonds/trapezoids, and the halo approach [41]. In the pyramids approach, one would update a cube of L^3 cells, then a cube of $(L-2)^3$ cells inside is, and continue to perform all updates for which the data exchanges are not needed until a pyramid in 1T3D is updated. This method is inconvenient, since, to perform all other updates, inverted pyramids, as well as other shapes such as differently oriented tetrahedra are required. A large variation of shapes is difficult for a human to describe and code, and for compiler to optimize. ConeTorre/TorreFold tile the space uniformly.

Another approach, the halo approach, was used in LBM Heterogenous code previously [10]. In it, parallel processors perform several computations on a group of L^3 cells without communication. After this, the data in the halo are incorrect and are erased; but due to the overlap of the L^3 cubes and excessive computations the final result is correct. ConeTorre is performed without redundant operations. Additionally, the LRnLA method of construction allows taking advantage of all levels of data localization, not just host-device communication as in [10].

Another related area of study is polygonal (polyhedral) optimization [57]. One fundamental paper of this topic [58] introduced the theory of dependency graph decomposition. However, we prefer to not use the achievements of this research field. In polygonal optimization, the rules for automatic search of best graph traversal and parallelization method are found and coded. This way, it is supposed that a program is written in a usual way, with ordinary loops over time and space; and the automatic optimization would transform the loops. In the LRnLA approach, the programmer is still considered the expert on the hardware model, and decides on the decomposition rules and parameters himself. The greatest disadvantage of the automatic polygonal optimization is the fact that it can not transform the data layout for the most efficient access, and works with whatever is provided to it. LRnLA approach gives simple advice to the design of the data layout: whatever is used in one sub-task should be localized in memory.

2.4 Data structure

LRnLA algorithms are built for better localization of data in the higher memory levels. The successful implementation of the constructed algorithm depends on the memory access pattern. This topic is widely discussed in LBM implementations [9, 53, 54, 59–61]. The goal is not only the locality of data that are accessed in one sub-task, but also data alignment and coalescing. Ideally, the data are aligned in the order that they are accessed.

For the purpose of locality the data are stored in the AoS (array of structure) fashion. That is, the data that are updated in a DG node are stored as a continuous structure in the memory: the data cell; and there is a space-filling curve traversing the coordinate space of the DG which converts the spatial x–y–z position into a 1D index of the data cell in the memory address space.

For the purpose of alignment, the space-filling curve follows the algorithm access patterns. This is not trivial for ConeFold algorithms. The sequentially accessed lattice cells are along the direction of the cube diagonal. That is why the recently developed FArSh (Functionally Arranged Shadow) data structure [62] is used here.

This way, two types of data storage are implemented. The data structure which is aligned with the axes directions is referred to as `tiles`. This structure covers the whole domain at a constant time.

The data which is aligned with the slope of ConeTorre is referred to as `FArSh`. It is composed of lines of cells $\{x'+j,y'+j,z'+j,j\}$, $j=0,\cdots,N_T$. The $\{x',y',z'\}$ coordinates are on the current computation wavefront (Fig. 4).

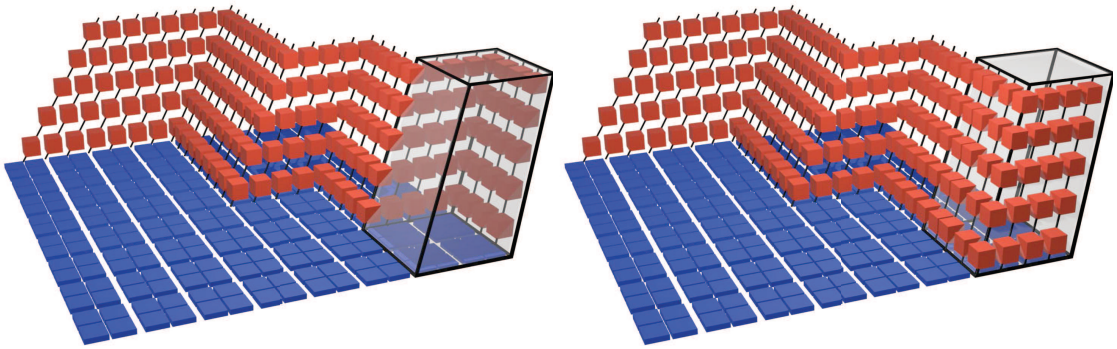


Figure 4: Data structure (1T2D). Groups of cells are stored in Tiles (blue) in the Z-order. Cells are stored in lines in FArSh (red). ConeTorre (one is pictured) loads bases (group by group) from Tiles and slopes (line by line) from FArSh. It stores the results overwriting the FArSh which it has read, so that the FArSh before and after ConeTorre stores cells that correspond to different positions. Left and right figures are the states of FArSh before/after the pictured ConeTorre.

The input for ConeTorre or ConeFold sub-task is the tile for its lower base and a FArSh portion for its slopes. Its output is the tile for its upper base and a FArSh portion for its upper slopes. The data that is input and output to and from a ConeTorre is organized in `FArSh` and `tiles` as well. The input and output `ConeTorre<n,NT>` bases is an array of $(n/2)^3$ groups. At each iteration, `ConeTorre<n,NT>` loads a gnomon of cells, $n_G = n^3-(n-1)^3$ cells total. Thus, `FArSh` for `ConeTorre<n,NT>` is an array of n_G lines of N_T cells. Tiles are implemented with a recursive Morton Z-order curve. The wavefront is a space with one less dimension. Thus, first, the 3D space coordinate is projected in the $\{1,1,1\}$ direction onto the cube gnomon $x',y',z' \to i_1,i_2$, where i_1, i_2 span a hexagon. Then the time coordinate t is added. Two dimensions of the FArSh i_1, i_2 that span a hexagon are implemented with a Z-curve in the 3 continuous areas of the hexagon [62]. Finally, `tiles` is a 3D Z-curve array of groups (group is a cube of 8 cells), and `FArSh` is a 2D Z-order array of lines of cells.

Each cell in the groups in `tiles`, and each cell in FArSh contains a full set of LBM populations in any order.

The same FArSh is used for input and output of data on the ConeTorre slopes: the cells are read from the right side, and the newly updated cells on the left side are written into the same memory positions.

2.5 LRnLA cell for LBM

The closure of the ConeFold subdivision is an LRnLA cell, the elementary update. By tiling the LRnLA cell in space and time, the whole dependency graph can be obtained, with exceptions for boundary conditions.

The type of the elementary LBM update can be chosen among many proposed stream-

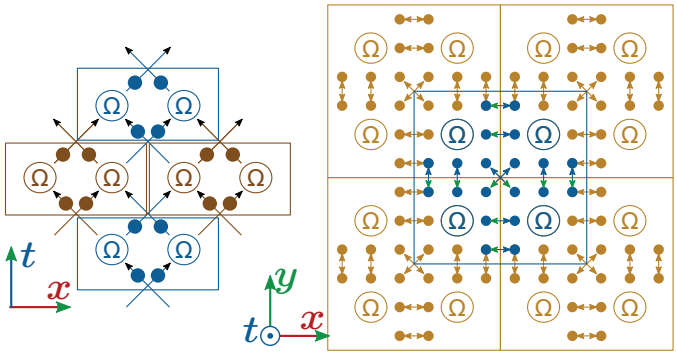


Figure 5: The compact scheme in the dependency graph. Ω are the collision operations. Streaming of f_q for $\boldsymbol{c}_q = \mathbf{0}$ is omitted.

ing patterns [54, 63, 64]. The positioning of each pattern on the dependency graph is discussed in [50]. The compact elementary update is proposed in [50, 51, 62]. We choose it for the implementation since it provides optimal use of the memory throughput.

Consider the following update scheme:

1. group cells into cubes of 2^d and perform the collision;
2. perform streaming operations which are between the cells of each group ('compact' sub-step);
3. regroup each cell with its other neighbors
4. perform streaming operations which are inside the group ('decompact' sub-step).

This is a full LBM step. Now, perform the cyclic shift of these sub-steps, so that the update starts with step 3. Steps 4–1–2 ('decompact'–'collision'–'compact') are enacted in the same group configuration. No outside neighbors of any group are required. This is a compact update.

Step 3 is where the data load and save takes place. Inside the base of the ConeTorre, the group are redefined, so that the new groups are shifted in relation to the previous configuration. At the gnomons of its base, the cells are read from FArSh (on the right side) and saved into FArSh (on the left side).

The advantage of the compact streaming scheme is that the update of a cubic group of 2^d cells requires only the data of this group, all values in a group are updated, and the elementary update contains a full set of streaming and collision operations of a full LBM update. The asynchronous ConeTorre in Fig. 2(b) do not access the same data cells at all during the iterations.

In the current work, an LRnLA cell contains two tiers: the identical updates of two compact groups with coordinates $(2i_x, 2i_y, 2i_z, 2i_t)$ and $(2i_x+1, 2i_y+1, 2i_z+1, 2i_t+1)$, for i_x, i_y, i_z, $i_t = 0, \cdots, N/2-1$. When RefoldConeFold is used, every update of a group at $(N-$

$i_x, N-i_y, N-i_z, i_t)$ is simultaneous with an update at $(N+i_x-1, N+i_y-1, N+i_z-1, i_t)$, i_x, i_y, i_z, $i_t = 0, \cdots, N-1$. This LRnLA cell can be used to construct the ConeTorre shape in a uniform direction.

The update is vectorized. Thus, the LRnLA cell contains the compact update of two vectorized groups, 8×16 cells in 16 groups total. The term *group* hereafter means a cube of 8 cells, and a *vectorized group* is a set of 8 groups in the positions mirrored around the center of the domain.

2.6 LRnLA subdivision on a heterogenous system

The previous LRnLA LBM code on GPU has shown performance that is superior to other known approaches [62]. Its GPU acceleration is efficient due to the uniformity of the computations. The presence of any inclusions introduces thread diversions, and some tiers of the ConeTorres have to be replaced with conditional statements or masks, and a notable performance drop is expected, so no boundary condition was used in [62]. That is why here it was decided to use GPU for the main part of computation and CPU for processing the boundary. In total, one or several GPUs process the 8 separate inner portions, and CPU cores process the boundaries of each N^3 block when the domain is folded and vectorized. ReFold algorithms [56] process the periodic boundary uniformly. The use of AVX vectorization is essential in the efficiency of ReFold implementation. Thus, we take advantage of the AVX vectorization at the same time.

In the current version, the domain with size $2N \times 2N \times 2N \times N_T$ cells is folded and ReFoldTorreFold is applied. Two parameters `MaxRank` and `minRank` control the size of the problem and CPU/GPU load balancing: $N = 2^{MaxRank+1}$, $n = 2^{minRank+1}$. Here $n = N_T$.

The `TorreFold<N,NT>`, which covers the whole folded dependency graph, is subdivided into `TorreFold<n,NT>`. After the first subdivision, the boundary `TorreFold<n,NT>` are assigned to CPU, and the inner ones are to be processed in GPU (Fig. 6).

On CPU, the implementation is based on the previous works [49,56]. The sub-tasks of `TorreFold<n,NT>` are distributed between standard C++ threads. They are subdivided into `TorreFold<n/2,NT>`, and recursively down to `ConeTorre<2,NT>`. This ConeTorre is a `for` loop of tiers of compact updates on a groups shifted by $\{1,1,1\}$ with each iteration.

Since the domain is folded, the data array element is a vectorized group. Manual AVX vectorization is used, that is, the vector and vector operations are defined explicitly in the code. One AVX vector contains PDF values in 8 cells in the domain, which are in the same place after folding.

On GPU, the implementation is inherited from [62]. `TorreFold<n,NT>` sub-tasks are performed on GPU, one per GPU at a time. The GPU computation is scalar, so, the `FArSh` data which is output from the `TorreFold<n,NT>` on CPU is unfolded: from vectorized cells into scalar cells, and from one `FArSh` array of vectorized cells to 8 `FArSh` arrays of scalar cells. Thus, there are 8 `FArSh` arrays, which are potentially distributed among 8 GPUs on a heavy node. If this is realized, there are 8 absolutely independent tasks of updating $(N-2n)^3$ lattice cites for N_T steps in time. A GPU stores all required `FArSh`, and

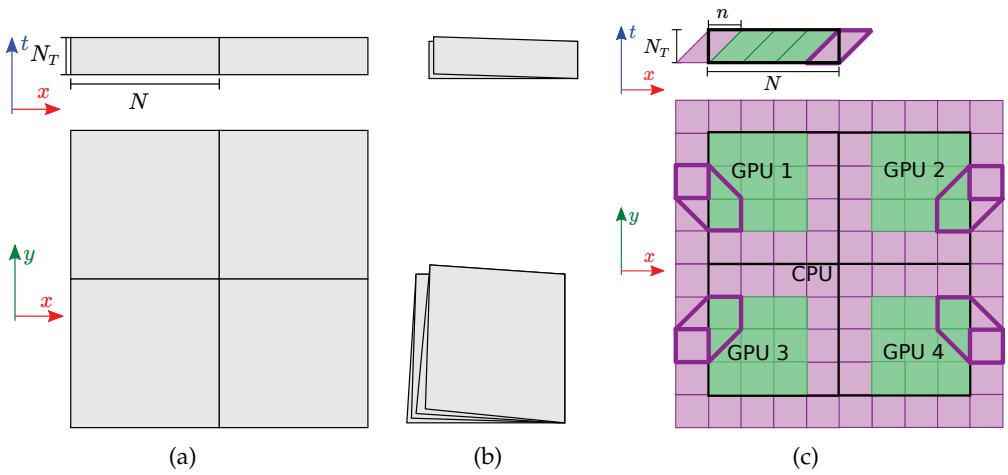


Figure 6: (a) The simulation domain projected to 1T1D and 2D. (b) The simulation domain is folded for `ReFold<N,NT>`. (c) ReFold is subdivided into `TorreFold<n,NT>`. In 1T1D projection the TorreFold shape is shown, the boundary shapes are executed on CPU, the inner shapes are executed in GPU. In the 2D projection, only the lower bases are depicted. Only one TorreFold is outlined in full. This TorreFold updates the 4 outlined data potions simultaneously.

the data for one `TorreFold<n,NT>` base cube. `TorreFold<n,NT>` sub-tasks are performed one-by-one on it: the base is loaded from CPU RAM, `TorreFold<n,NT>` is processed, and the data is saved back to GPU.

On GPU, `TorreFold<n,NT>` is recursively subdivided into `ConeTorre<8,NT>`, which are distributed between SMs. `ConeTorre<8,NT>` is a loop, and on each iteration of this loop, 16 CUDA-thread warps execute the tiers of the compact update on 8^3 cells.

3 Theoretical performance analysis

3.1 Roofline analysis

With the recursive subdivision of LRnLA algorithm construction, the memory hierarchy can be included in the Roofline analysis [48]. For a specific illustration, let us take CPU AMD Ryzen R9 5950X, 16 cores, 2 channels of DDR4@3200, 128GB RAM and GPU nVidia RTX 3080, 10 GB GDDR6X RAM. As a sample problem, let us take a cube of $2N \times 2N \times 2N$ cells and the problem of LBGK update in all cells $n = N_T$ steps in time.

We plot the third axis of the Roofline to show the storage size of the memory level. If the task is localized on some level of the memory hierarchy, its sub-tasks are executed with the data bandwidth of that storage. Thus, the localization shows which Roofline of the cache-aware Roofline model limits the performance.

We estimate the peak by following the recursive LRnLA decomposition. The AI of some task is estimated and its peak is found. The performance of the task can not be higher than the performance of the sub-tasks, so the AI of the sub-tasks is found and the

Table 1: Computational parameters of single-precision LBGK variants.

Shells	-	1	2	1,2	3	1,3	2,3	1,2,3
Q	Q1	Q7	Q13	Q19	Q9	Q15	Q21	Q27
$\mathcal{S}$, byte per cell	4	28	52	76	36	60	84	108
FlOp per cell	30	78	150	198	126	174	246	294
FlOp per value	30	11.1	11.5	10.4	14	11.6	11.7	10.9
$\mathcal{O}$, 2× FMA per cell	30	114	246	330	206	290	422	506
$\mathcal{O}/N_Q$, 2× FMA per value	30	16.3	18.9	17.4	22.9	19.3	20.1	18.7

peak is estimated. The actual peak is the minimum of the two. Decomposition is followed from the task of update of all nodes down to one cell update. In this top-to-bottom approach, in general, the AI and the peak tend to decrease. However, when one task operates with a sufficiently small data size, it is localized in the higher level of memory, and its limit on the Roofline is higher. The correct estimation of the compute-bound peak should be made with account for FMA (fused multiply-add) operations. On the chosen hardware, the rates of execution of float addition operations, float multiplication operations, and fused multiply-add operations are the same. The peak performance of the processor is declared under the assumption that each FMA operation corresponds to two FlOp (floating-point operations). Such a perfect balance of multiplications and additions is uncommon in real calculations. For our code, we carefully computed both metrics: the total number of FlOp, and the total number of FMA operations.

Along with the standard GFlOps (billion of floating-point operations per second) metric, we measure the performance in GLUps (billions of lattice site updates per second). Since the problem is memory-bound, the number of values stored and updated in a lattice site is important. Thus, we introduce one more metric: GPUps, billions of population updates per second, which is GLUps$\times Q$.

The LBM scheme parameters are compiled in Table 1. All possible configurations of shells (2.3) were included in the study. Only a few of these are relevant in simulations. Others are included to study the performance trends.

3.2 Limits of stepwise implementation

Let us illustrate the problems of the stepwise code. To perform one LBGK step update, $(2N)^d\mathcal{O}$ operations are required, and $2(2N)^d\mathcal{S}$ bytes have to be loaded and stored. This is an ideal case. In practice, for many streaming schemes in 3D, some data are loaded from the memory at least twice, since the cell data are accessed from the neighbor cells. This overhead depends on the domain traversal. For the compact update, the AI is estimated as ideal, since it is guarantees that no cells are accessed twice, and the data are modified in-place. Let us consider the CPU computation first. For large enough problems, the data is localized in the CPU RAM. The AI is $\mathcal{I}_{\text{SW}} = (2N)^d\mathcal{O}/2(2N)^d\mathcal{S}$. The Roofline

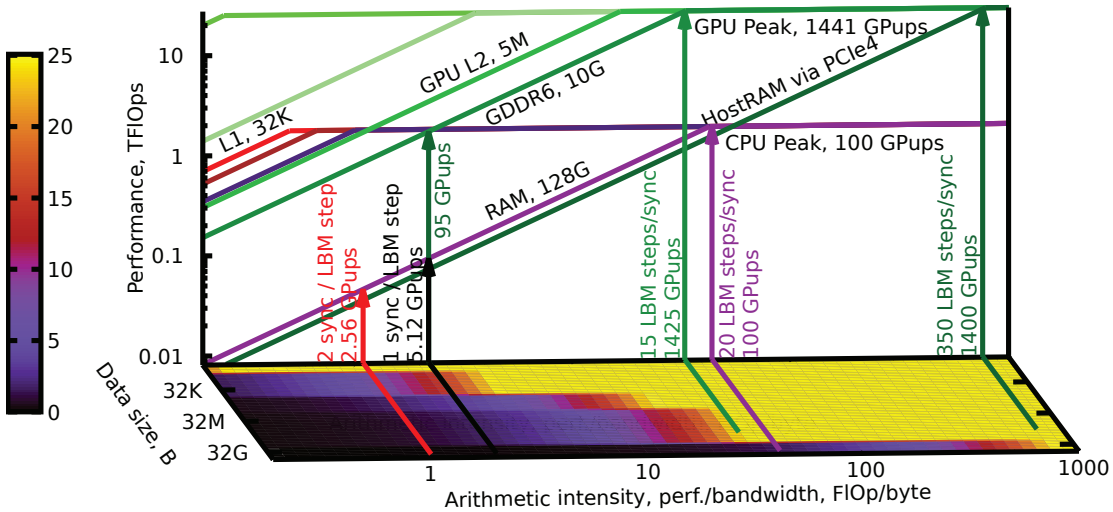


Figure 7: Limits of stepwise implementations. The limit of the two step implementation is shown with a red arrow (CPU), fused collision and streaming is shown with black arrow (CPU) and a green arrow at the same place (GPU). Other limits are illustrated to see how many steps have to be fused to reach compute-bound domain. The color scale on the color map below is the same as the vertical axis scale: performance.

limitation is shown in Fig. 7 (black), which is very low in comparison to the compute peak. Furthermore, the model illustrates that if the domain is iterated twice over in one LBM update, as it was in the early 2-step implementations [54] and even some recent ones [65], the limit is even less (red). Indeed, fusing collision and streaming doubles the performance peak.

It can be seen from the same graph that an LBGK code which could reach the compute peak for CPU needs to have 20 full steps fused. The memory bandwidth and compute peak for GPU computations is higher, and we estimate that 15 LBM steps should be fused to approach compute-bound domain. The situation for heterogenous computation is more difficult. If the data is stored in CPU RAM, the GPU performance is limited with PCIe4 bandwidth $\Theta_{\text{CPU-GPU}}$.

To reach compute-bound peak under these conditions, the AI should exceed $\frac{\Pi_{\text{GPU}}}{\Theta_{\text{CPU-GPU}}}$ where Π_{GPU} is the peak GPU performance. We estimate that 350 LBM steps should be fused. Such fusion is not adequate without temporal blocking approach, and any blocking introduces some overhead.

3.3 Limits of the current implementation

For the algorithm proposed in the current work, refer to Table 2 and Figs. 8, 9, 12. In the table, along with the estimated AI parameter, we compiled the estimates on the total data required to perform the corresponding sub-task: when the values are scalar (loc.S) and when the domain is folded and vectorized (loc.V). The former is relevant for the tasks performed on GPU, the latter is for the tasks performed on CPU.

Table 2: Algorithm decomposition for $N=256$.

	x-scale	t-scale	loc.V	loc.S	AI
C^s_{512}	512	32		9.5G	69.6
`RefoldTF<256,32>`	256	32	3.2G	9.5G	37.1
`ConeFold<32>`	32	32	69M	8.7M	9.6
`TorreFold<16,32>`	16	32	26.7M	3.3M	6.2
`TorreFold<8,32>`	8	32	5.6M	719K	3.7
`TorreFold<4,32>`	4	32	1.04M	134K	2.5
`TorreFold<2,32>`	2	32	152K	19K	2.2
`LRnLA cell`	2	2	8.9K	1.1K	2.3
`Cell`	1	1	608	76	1.1
`StepWise`	512	1		9.5G	2.2

Let us study the Roofline limitation of the code proposed in the current work. The whole task is an update of a 3D cube C^s_{2N} for n time iterations. Its data is stored in the array of 8 instances of structure `cubeLR<Group, MaxRank>` (Section 4, Listing 1, line 30), where `MaxRank` is $R_{\max}$ and $N=2^{R_{\max}+1}$. The elements of this structure are scalar groups located in the Z-order curve. To perform the C^s_{2N}, $(2N)^d n\mathcal{O}$ operations are required and $(2N)^d\mathcal{S}$ bytes are to be loaded and saved. Thus the AI is $\mathcal{I}_{C^s_{2N}}=n(2N)^d\mathcal{O}/2(2N)^d\mathcal{S}$.

In the ReFold, the domain is folded to obtain a cube of vectorized cells with size N. It is subdivided into `TorreFold<n,n>`. Such shape with equal sizes is also referred to as `ConeFold<n>` [48,56]. ReFold contains all operations of the cube, and it controls the data exchange between CPU and GPU. The `TorreFold<n,n>` tasks that are performed in CPU input and output the tiles `cubeLR<GroupV, minRank>` (Section 4, Listing 1, line 38) on the cube gnomon, where `minRank` is $R_{\min}$ and $n=2^{R_{\min}+1}$. There are $\Gamma^-_{N^d/n^d}$ such `cubeLR` instances in total, where $\Gamma^-_{M^d}=M^d-(M-1)^d$ is the number of cells in the gnomon of a cube with size M. `TorreFold<n,n>` tasks on CPU output these data, then they are unfolded into 8 instances of scalar structures `cubeLR<Group, minRank>`, which are sent to GPU. In total, $\mathcal{S}^{\text{exch}}_{\text{RTF}}=2^d n\Gamma^-_{(N-n)^d}\mathcal{S}$ data are exchanged. This size is used to estimate the AI of the `ReFold<N,n>` on the Roofline. Its total data size and operations number are the same as for the C^s_{2N}, but more exchanges are required, thus, its AI is lower. In the Table 2, for `ReFoldTF<256,32>`, loc.S and loc.V show the estimations of the whole size of the domain and its vectorized portions correspondingly.

Next, the number of operations and the data IO for `TorreFold<n,n>` (`ConeFold<n>`) depend on the type of the cells (scalar $^{\text{in}}$ or vector $^{\text{BC}}$), and number of boundary planes of the cube this `TorreFold<n,n>` intersects. Let us introduce ν_O, so that $\nu_O=2$ if the `TorreFold<n,n>` intersects only one face of the cube, $\nu_O=1$ if it intersects two faces (it is

near the edge of the cube), and $\nu_O = 0$ if it is in the corner. Then

$$\mathcal{O}_{\mathrm{CF}}^{\mathrm{in,BC}} = \frac{n^{d+1}\mathcal{O}^{\mathrm{in,BC}}}{d+1-\nu_O}, \quad \mathcal{S}_{\mathrm{CF}}^{\mathrm{in,BC}} = \frac{2n^d(d+1)\mathcal{S}^{\mathrm{in,BC}}}{d+1-\nu_O}, \quad \mathcal{O}^{\mathrm{BC}} = 2^d\mathcal{O}^{\mathrm{in}}, \quad \mathcal{S}^{\mathrm{BC}} = 2^d\mathcal{S}^{\mathrm{in}}. \tag{3.1}$$

The AI for both inner and boundary shapes is

$$\mathcal{I}_{\mathrm{CF}}^{\mathrm{in,BC}} = \frac{n\mathcal{O}}{2(d+1)\mathcal{S}}. \tag{3.2}$$

The `TorreFold<n,n>` task is decomposed into `TorreFold<m,n>` sub-tasks, $2 < m < n$ on CPU and $8 < m < n$ on GPU. Both for vector and scalar operations, the AI is

$$\mathcal{I}_{\mathrm{CT}}^{\mathrm{in,BC}} = \frac{nm^d\mathcal{O}}{2\left(m^d + n\Gamma^-_{m^d}\right)\mathcal{S}}. \tag{3.3}$$

$\mathcal{I}_{\mathrm{CT}}^{\mathrm{in}}$ and $\mathcal{I}_{\mathrm{CT}}^{\mathrm{BC}}$ are equal for scalar calculations on GPU and vector calculations on CPU. Thus, the arrows that illustrate them on our Roofline graphs overlap. However, as it can be seen in the Table 2, the data footprint is different.

3.4 Roofline of CPU computation

Fig. 8 illustrates the Roofline analysis for the one thread performance. That is, if all TorreFolds are executed in CPU, and only one CPU thread is used. The compute peak is lower, so even the stepwise algorithm is close to compute-bound. In a subdivision, we start from the 'Cube' and subdivide the tasks until one cell update. At first, the limitation is the CPU RAM, but the data required in the `TorreFold<8,32>` task fits in the L3 cache. Thus, the performance of its sub-tasks, `TorreFold<4,32>`, is limited with the L3 cache bandwidth. The AI of `TorreFold<2,32>` overlap the AI of the ideal stepwise algorithm. Finally, the compact update is localized in the L2 cache, the resulting peak is compute-bound. The performance obtained in our code implementation is 4.3 GPUps, and it is close to the peak. This proves that vectorization is used efficiently.

Fig. 9 illustrates the Roofline analysis for the multicore CPU performance. The compute peak is higher, so the performance drop in the result of the first major subdivision is relevant. In this version, after the first subdivision, both scalar (inner) and vector (BC) sub-tasks are performed on CPU. The AI match, so the arrows in the Roofline graph overlap. But scalar sub-tasks require 8 times less data, so the scalar `ConeFold<32>` is localized in the L3 cache, and its sub-tasks are not limited with the RAM bandwidth, unlike similar vectorized shapes. The performance result, obtained in the code, is 15.5 GPUps, and it is close to the estimated peak as well. Moreover, it is larger than the theoretical peak performance of the ideal stepwise implementation.

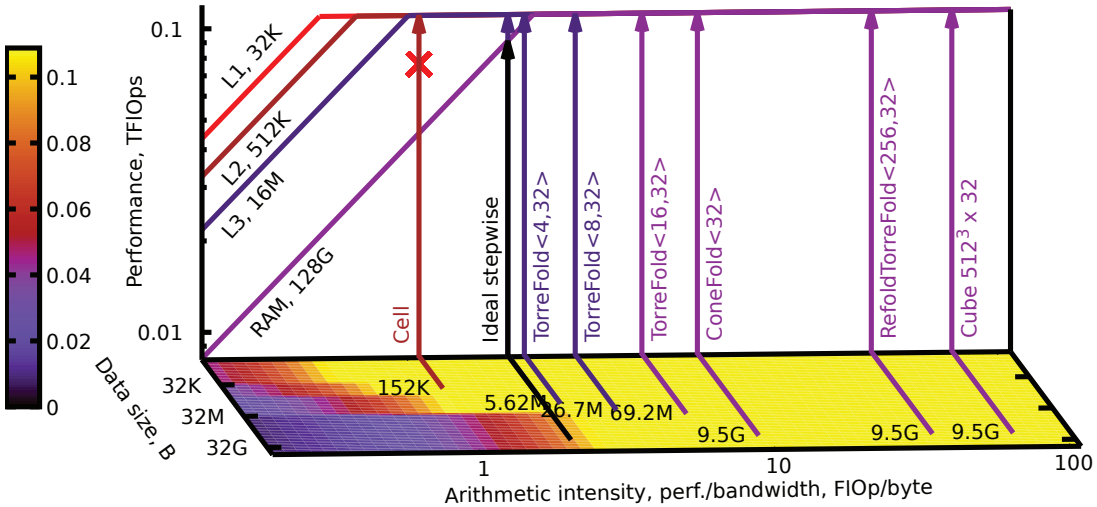


Figure 8: Performance limit study of the one thread execution. The performance obtained in the benchmarks is shown with the cross mark, it is 4.3 GPUps (0.23 GLUps). The color scale on the color map below is the same as the vertical axis scale: performance.

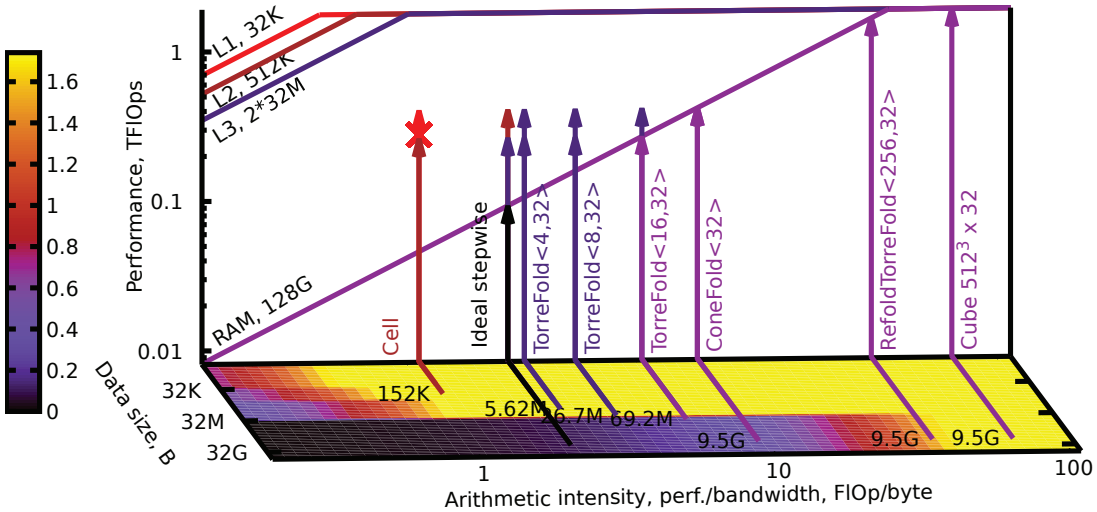


Figure 9: Performance limit study of the 16 cores. The performance obtained in the benchmarks is shown with the cross mark, it is 15.5 GPUps (0.82 GLUps). The color scale on the color map below is the same as the vertical axis scale: performance.

3.5 Heterogeneous implementation efficiency

We run several tests to see the efficiency of heterogeneous approach (Fig. 10). In the current code, for a fixed size of the domain N, the fraction of the computations performed on CPU is controlled with the n parameter (see Section 2.6).

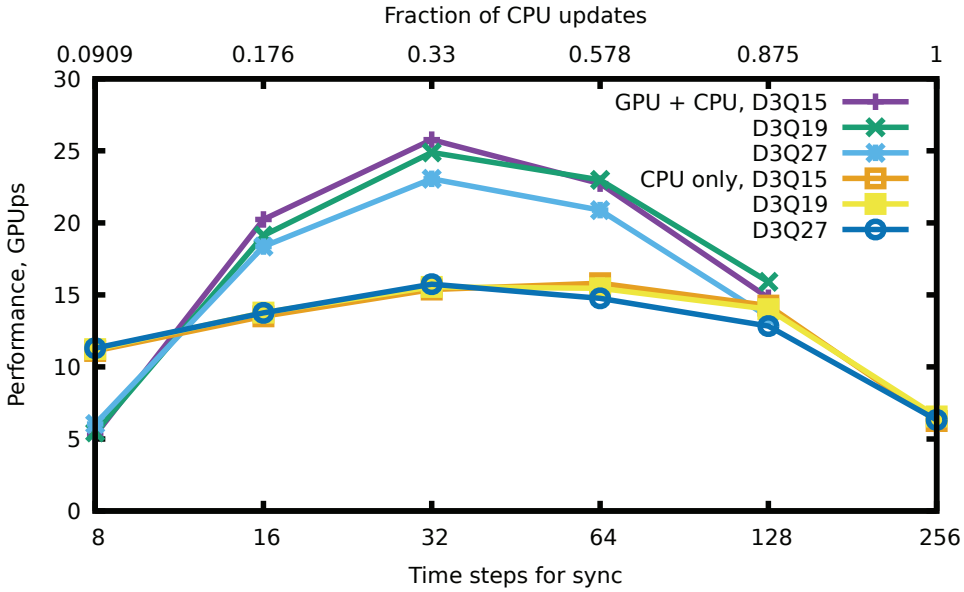


Figure 10: Performance vs GPU/CPU balance. Lattice grid $(2N)^3 = 512^3$.

n is also an algorithmic parameter, it is the number of steps fused in a ConeTorre. We compare the performance to CPU-only runs, and the dependency of performance on n is also present. With higher n, a less portion of computations is performed with GPU. With lower n, the PCie4 bandwidth becomes the bottleneck.

The GPU-only code, implemented with ConeTorre, shows higher performance [62]. However, the tests in [62] are made for homogeneous simulation domain, and if any conditional statements are introduced the performance deteriorates. CPU computation is introduced here to enable the inclusions, and to add to the computing power as well.

In Fig. 11 we study the performance dependence on the data updated per cell. Larger shapes in the LRnLA subdivision provide higher AI in general, so the performance is higher with higher $R_{\max}$. With less Q the data are localized in the faster memory level on the earlier stages of the recursive subdivision, thus, performance declines with Q. With lower $R_{\max}$, the lower performance, and the atypical behavior is caused by an additional reason: the degree of parallelism is lower.

In Fig. 12, the proposed heterogeneous code is studied. The ReFold is applied. The boundary part is processed on CPU in the vectorized code. Then the domain is unfolded, and eight inner parts are performed on GPU. Then the domain is folded and the remaining boundary is processed on CPU. The CPU part on the right boundary is performed first. It outputs `FArSh` data that is unfolded and sent to GPU. After the GPU part is processed, `FArSh` is folded back and sent back to CPU.

The AI of the each of the eight inner parts of the `ReFold` (the combinations of all inner

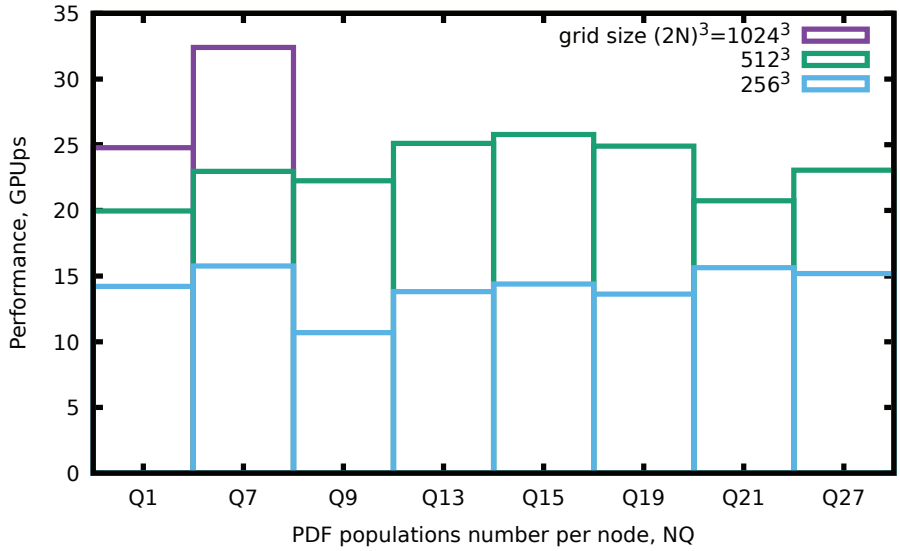


Figure 11: Performance vs number of updated values per cell.

`ConeFold<n>` of one eighth of the domain) is

$$\mathcal{I}^{\text{in}}_{\text{RTF}} = \frac{n(N-n)^d\mathcal{O}}{2\left((N-n)^d + n\Gamma^-_{(N-n)^d}\right)\mathcal{S}} = \frac{n}{1+d\frac{n}{N-n}+o\left(\frac{n}{N-n}\right)}\frac{\mathcal{O}}{2\mathcal{S}}.$$

The AI of the boundary part is:

$$\mathcal{I}^{\text{BC}}_{\text{RTF}} = \frac{2^d n\left(N^d-(N-n)^d\right)\mathcal{O}}{2\cdot 2^d\left(n^d\Gamma^-_{N^d/n^d}+n\Gamma^-_{N^d}\right)\mathcal{S}} = \frac{n}{2+\frac{d-1}{2}\frac{n}{N}+o\left(\frac{n}{N}\right)}\frac{\mathcal{O}}{2\mathcal{S}}.$$

The result for these two AI is different, and, in Fig. 12, the arrow for `ReFoldTF<224,32>` (inner part) is to the right of the arrow for `ReFoldTF<256,32>`. The arrow for `ReFoldTF<224,32>` is limited by the PCie4 bandwidth, since the CPU-GPU data exchange takes place.

On GPU, all `FArSh` data for the current inner GPU part is assumed to be already in GPU, and the tiles are in the CPU memory. The tiles are sent to GPU for each `ConeFold<n>` execution, and saved back after `ConeFold<n>` is finished. The AI of `ConeFold<n>` is

$$\mathcal{I}^{\text{in}}_{\text{CF}} = \frac{n\mathcal{O}}{2\mathcal{S}}. \tag{3.4}$$

It is higher than the AI of the shape of all inner `ConeFold<n>` tasks combined. This is not shown in Fig. 12 since it does not introduce new limits.

The sub-tasks of `ConeFold<n>` are localized in the GPU.

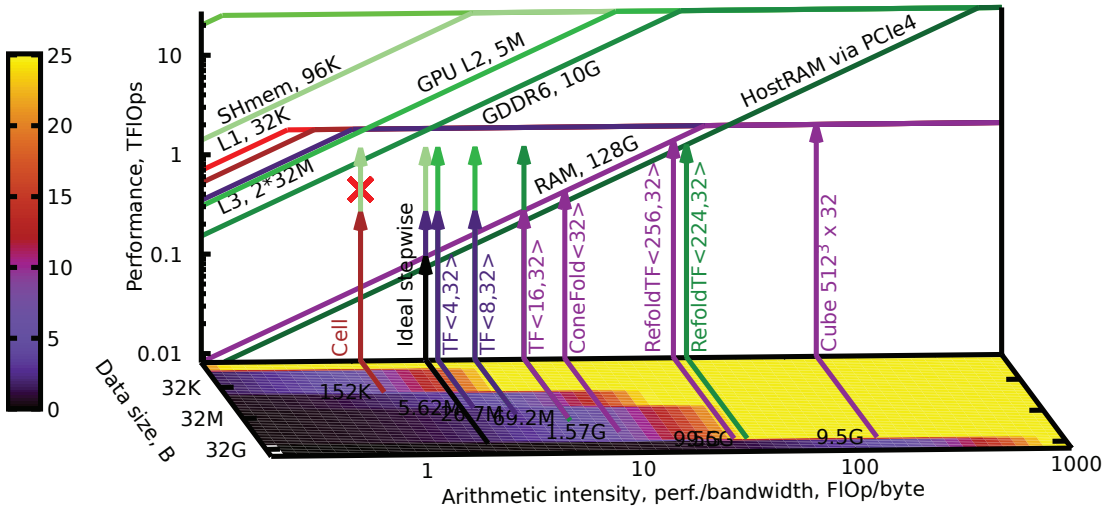


Figure 12: Performance limit study of the heterogeneous code. The performance obtained in the benchmarks is shown with the cross mark, it is 25 GPUps (1.31 GLUps). The color scale on the color map below is the same as the vertical axis scale: performance.

Thus, we obtain the two performance limits from the heterogeneous Roofline study. One is the performance limit of the boundary computation on CPU ($\Pi^{\mathrm{BC}}_{\mathrm{CPU}}$)). The second one is performance limit of the 8 inner portions of the domain on GPU that is determined the GPU performance and PCIe4 bandwidth ($\Pi^{\mathrm{in}}_{\mathrm{GPU}}$).

The total peak is estimated with the use of the ratio of the data processed on GPU and CPU:

$$\frac{\mathcal{O}_{\mathrm{CPU}}}{\mathcal{O}_{C^v_N}} = \frac{nN^d - n(N-n)^d}{nN^d}, \quad \frac{\mathcal{O}_{\mathrm{GPU}}}{\mathcal{O}_{C^v_N}} = \frac{n(N-n)^d}{nN^d}.$$

And we obtain

$$\Pi \le \min\left(\Pi^{\mathrm{in}}_{\mathrm{GPU}}\left(1-\frac{n}{N}\right)^d, \Pi^{\mathrm{BC}}_{\mathrm{CPU}}\left(1-\left(1-\frac{n}{N}\right)^d\right)\right). \tag{3.5}$$

This is higher than the CPU peak. The bottleneck is determined by the n parameter, which should be smaller than the N size for efficient parallelization. It can be increased in the implementation if more computer RAM is installed.

4 Implementation details

4.1 Heterogeneous code

There are many tools and libraries for the use of acceleration techniques of the modern hardware. In the heterogeneous code, only one tool is not sufficient. The use of universal programming directives such as openACC does not allow enough control over the data

Listing 1: A sample of the code that can be compiled for all types of required instruction sets

```
namespace lbm {//LBM constants
  const ftype w0 = 1./3., w1 =1./18., w2 =1./36., w3 =    0.0;
// prefix for constant arrays are different in C++ and CUDA
  CONST_PREFIX int ci[][3] = {
    {-1, 0, 0,}, //  0
    { 0,-1, 0,}, //  1
    { 0, 0,-1,}, //  2
    ...}
// Cells are either scalar or vectorized; controlled with floatT.
template<class floatT> struct Cell {
  floatT fq[lbm::NQ];
  ...
// Functions are compiled for host or for the device
  FUNC_PREFIX floatT bgk(int iq, floatT eqfq) {
    using namespace lbm;
    return fq[iq] - dtau*( fq[iq] - eqfq * get_wiq(iq) );
    ...}
...};
// A group is 8 cells; it is convenient to use one cell vector instead
// This way the update of cells in a group is also vectorized
typedef Cell<ftypeV> group;
// A vectorized group is a set of 8 groups mirrored around the center of the domain.
typedef array<Cell<ftypeV>,8> groupV;

//All data is stored in tiles and FArSh
// The main data portion is scalar
// There are 8 instances of theCube structure that can be distributed between GPU
struct theCube {
  // Recursive implementation of 3D Z-order curve of groups
  cubeLR<3, group, MaxRank>* tiles=NULL;
  // scalar FArSh is allocated on GPU
  // gnomon of a cube with (N-n)^3 cells
  // the length of the line in time is n
  FArSh4cube<Cell<ftype>, N-n, n>  mold;
};
//on the boundary, data is vectorized
struct theCubeGnomon {
  cubeLR<3, groupV, minRank>* tiles=NULL;
  FArSh4cube<Cell<ftypeV>, N, n> mold;  //< vector FArSh is stored in CPU
...};
```

localization throughout the simulations (see also Section 1.1). The code, however, should not become too complex. The use of same data structures (`FArSh` and `Tile`) for all input and output in GPU and CPU is one step in this directions.

The memory allocation and task scheduling are essentially different. However, here, we made a successful attempt to make the computation kernel itself portable between CUDA and vectorized CPU code. This concerns the LBM method itself: its constants, methods for initialization, calculation of moments, collision. It is implemented through preprocessor commands (Listing 1).

We define data types: `ftype` for scalar (single or double precision) type and `ftypeV` for vectors of 8 values.

4.2 Elementary update

The implementation of the compact streaming can differ with different memory layout [50, 51, 62]. Here we use its latest development that is described in [62] (Listing 2). In it, the code is succinct due to the special numbering of PDF in a group. In each group, the cells are mirrored across the center. For example, in cells with $i_x=0$, f_0 corresponds to the $\boldsymbol{c}_0=\{-1,0,0\}$ velocity, and in cells with $i_x=1$ it corresponds to the $\boldsymbol{c}_0=\{1,0,0\}$ velocity.

In the current code, the implementation is improved further by integrating collision into streaming (Listing 3). To perform the collision, all PDF at the current lattice cite should be collected to compute the moments. On the other hand, all the required data is already inside the group at the start of the compact tier. The moments can be computed before the data is streamed. A similar trick was used before in [66]. When the moments are known beforehand, the PDF values can be updated in the collision without reading all other PDF values.

Listing 2: Elementary update on GPU

```
__device__ const int3 ci3[] = {{-1, 0, 0,}, ... };
...

Cell ct; // temporary cell in the CUDA thread register
// in the ConeTorre loop:

 ct.collision(dtau);

 for (int iq=0; iq < Nq; iq++) { // compact
   if (ci3[iq].x > 0) ct.fi[iq] = __shfl_xor_sync(0xFFFFFFFF, ct.fi[iq], 1);
   if (ci3[iq].y > 0) ct.fi[iq] = __shfl_xor_sync(0xFFFFFFFF, ct.fi[iq], 2);
   if (ci3[iq].z > 0) ct.fi[iq] = __shfl_xor_sync(0xFFFFFFFF, ct.fi[iq], 4);
 }

 .... //FArSh exchange and cell ct shift

 for (int iq=0; iq < Nq; iq++) { //de-compact
   if (ci3[iq].x < 0) ct.fi[iq] = __shfl_xor_sync(0xFFFFFFFF, ct.fi[iq], 1);
   if (ci3[iq].y < 0) ct.fi[iq] = __shfl_xor_sync(0xFFFFFFFF, ct.fi[iq], 2);
   if (ci3[iq].z < 0) ct.fi[iq] = __shfl_xor_sync(0xFFFFFFFF, ct.fi[iq], 4);
 }
```

The elementary update on GPU is implemented exactly as in [62] (Listing 2).

4.3 ConeTorre

ConeTorre is essentially a loop over time iterations (Listing 4, line 29). On CPU, one iteration is the `fused_tier` (Listing 3). On GPU, one iteration is the loop over L_f (an integer parameter) iterations. It is implemented exactly as was reported in [62].

Listing 3: Fused Tier

```
void fused_tier(groupT& g, FArSh4group<ftypeT>& f4g) {
  array<MFCell<ftypeT>,8> mM; //moments required for the collision
#pragma unroll 1
  for(int ic=0; ic<8; ic++) {
    g[ic].calculate_moments(mM[ic]);
    //the PDF with zero velocity can be updated as is:
    g[ic].fq[Nq] = g[ic].bgk(Nq, mM[ic].F0);
  }
  //zero velocity PDF is not streamed. Its FArSh exchange is here.
  //7 cells are in a gnomon of the 8-cell cube: the data is swapped with FArSh
  for(int ic=1; ic<7; ic++) swap(f4g[ic]->fq[Nq], g[ic].fq[Nq]);
  //the eighth element of the group is cyclically shifted inside the base
  swap_left(f4g[0]->fq[Nq], g[0].fq[Nq], g[7].fq[Nq]);
#pragma unroll Nq
  for (int iq=0; iq < Nq; iq++) { //loop over all streamed PDFs
    array<ftypeT,7> ffq; //temporary value for swapping
    for(int ic=0; ic<7; ic++) ffq[ic] = f4g[ic]->fq[iq];
#pragma unroll 2
    //the collision is before streaming:
    for(int ic=0; ic<8; ic++) g[ic].fq[iq] = g[ic].bgk(iq, mM[ic]);
    //compact + FArSh exchange + decompact
    const int flagC=(ci[iq][0]>0?1:0)+(ci[iq][1]>0?2:0)+(ci[iq][2]>0?4:0);
    const int flagD=(ci[iq][0]<0?1:0)+(ci[iq][1]<0?2:0)+(ci[iq][2]<0?4:0);
    for(int ic=0; ic<7; ic++) f4g[ic]->fq[iq] = g[ic^flagC].fq[iq];
    g[0^flagD].fq[iq] = g[7^flagC].fq[iq];
    for(int ic=1; ic<7; ic++) g[ic^flagD].fq[iq] = ffq[ic];
    g[7^flagD].fq[iq] = ffq[0];
  }
}
```

4.4 TorreFold

`TorreFold<n,NT>` is executed either on CPU, or on GPU (Listing 4). On CPU, it is processed with one thread. It contains iteration over the bottom base $(n/2)^3$ groups in the reverse order of the Z-order curve, that is, from the rightmost corner of the base. At each iteration, a `ConeTorre<2,NT>` is started from that group. Thus, the recursive decomposition of `TorreFold<n,NT>` into `TorreFold<n/2,NT>` and so on until `ConeTorre<2,NT>` is not coded explicitly. The decomposition is realized through the traversal over the groups of the domain.

On GPU, `TorreFold<n,NT>` distributes `ConeTorre<8,NT>` sub-tasks between GPU SMs (see [62]).

5 Conclusion

As the trend towards heterogeneous hardware continues for future supercomputers, future HPC simulations should also be heterogeneous and take advantage of all computing acceleration methods. We used LRnLA methods both for CPU and GPU computations, and, in the current work, we presented the first LRnLA code for heterogeneous LBM

Listing 4: `TorreFold<n,NT>` on CPU

```
template<int dim, int rank> void TorreFold::update( // here dim=3, n=2^(minRank+1)
        array<int, dim> coor, // 3D index of the upper base cube of the TorreFold
        int it0, // time index
        ConeFold* CF) { // wait condition
  //The TorreFold is started if the 3 (or less for boundary) TorreFolds
  // that it depends on are executed
  if(/* wait */) return;
  // Here the height of TorreFold is Nt=n
  const int Nt=1<<(rank+1)
  // Position of the lower base
  array<int,dim> Cbot; for(int c=0; c<dim; c++) Cbot[c] = coor[c]-1;
#ifdef CALC_on_GPU
  update_on_GPU(rank, LRindex<dim>::zip(Cbot), LRindex<dim>::zip(coor), it0, Nt);
#else//CALC_on_GPU
  // indexes of the top and bottom bases in the tiles array
  long int Itop=LRindex<dim>::zip(coor); Itop <<= (dim*rank);
  long int Ibot=LRindex<dim>::zip(Cbot); Ibot <<= (dim*rank);
  const long int NNN=1L<<(dim*rank);//number of groups in the base
#pragma omp parallel for num_threads(8)
  for(int ic0=0; ic0<(1<<dim); ic0++) {
    auto& cube=cube_data_arr[ic0];
    group* grBot=&cube.tiles->get_data(Ibot);// tiles array of the bottom base
    group* grTop=&cube.tiles->get_data(Itop);// tiles array of the upper base
    // loop over groups in the base in the reverse Z-curve index order:
    for(long int i=NNN-1; i>=0; i--) {
      group floor=grBot[i];// the current group
      //Find the position in FArSH
      FArSh4group<ftype> f4g; f4g.align(Ibot+i, cube.mold, it0);
      for(f4g.it=0; f4g.it<Nt; ++f4g) // this loop is the ConeTorre<2,Nt>
        fused_tier(floor, f4g);
      grTop[i] = floor;//group in the upper base is saved
    }
  }
#endif//CALC_on_GPU
  // communicate execution to the next TorreFold
  if(CF) /* post */;
}
```

simulations.

The code design uses temporal blocking. The recursive LRnLA subdivision, along with the convenient data storage structures, allows a uniform description for all levels of parallelism, data localization sites, and data exchange configurations.

The use of compact streaming in LBM simplifies the task. Without it, in ConeTorre, a halo of one cell around the ConeTorre projection has to be loaded. The compact update has no outside dependencies. This leads to better localization, fewer data in `FArSh` arrays, and even more asynchrony between ConeTorres.

The code is implemented for a heterogeneous system. The computations are written in such a way as to be understood as scalar or vector instructions on CPU, or GPU. Efficient use of AVX vectorization is an essential step in making the contribution of many-core CPU to the overall performance significant, and our benchmarks have proven our

success in this effort.

With the Roofline model, we have illustrated the advantages of the proposed design. The theoretical performance study revealed the bottlenecks. The main bottleneck is set by the n parameter. n should be much smaller than the N size for two reasons. With a higher N/n ratio, a larger portion of the calculation is delegated to GPU, and a higher degree of parallelism is available in the algorithm. The size of the domain, in turn, can be increased in the implementation if more computer RAM is installed. In a result, we conclude that larger problems can be simulated with higher efficiency in the heterogeneous system.

Acknowledgments

The work is supported by Russian Science Foundation, grant # 18-71-10004.

References

[1] Sauro Succi. *The Lattice Boltzmann Equation: For Complex States of Flowing Matter*. Oxford University Press, 2018.

[2] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. The lattice Boltzmann method. *Springer International Publishing*, 10(978-3):4–15, 2017.

[3] Guy R McNamara and Gianluigi Zanetti. Use of the Boltzmann equation to simulate lattice-gas automata. *Physical Review Letters*, 61(20):2332, 1988.

[4] Yue-Hong Qian, Dominique d'Humières, and Pierre Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479, 1992.

[5] Pierre Lallemand, Dominique d'Humieres, Li-Shi Luo, and Robert Rubinstein. Theory of the lattice Boltzmann method: Three-dimensional model for linear viscoelastic fluids. *Physical Review E*, 67(2):021203, 2003.

[6] Christian Feichtinger, Johannes Habich, Harald Köstler, Ulrich Rüde, and Takayuki Aoki. Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU–GPU clusters. *Parallel Computing*, 46:1–13, 2015.

[7] Fredrik Robertsen, Jan Westerholm, and Keijo Mattila. Lattice Boltzmann simulations at petascale on multi-GPU systems with asynchronous data transfer and strictly enforced memory read alignment. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 604–609. IEEE, 2015.

[8] Zhao Liu, XueSen Chu, Xiaojing Lv, Hongsong Meng, Shupeng Shi, Wenji Han, Jingheng Xu, Haohuan Fu, and Guangwen Yang. SunwayLB: Enabling extreme-scale lattice Boltzmann method based computing fluid dynamics simulations on Sunway TaihuLight. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 557–566, 2019.

[9] Seiya Watanabe and Changhong Hu. Performance evaluation of lattice Boltzmann method for fluid simulation on A64FX processor and supercomputer Fugaku. In *International Conference on High Performance Computing in Asia-Pacific Region*, pages 1–9, 2022.

[10] Takashi Shimokawabe, Toshio Endo, Naoyuki Onodera, and Takayuki Aoki. A stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers. In *Cluster Computing (CLUSTER)*, pages 525–529. IEEE, 2017.

[11] Giacomo Falcucci, Giorgio Amati, Pierluigi Fanelli, Vesselin K Krastev, Giovanni Polverino, Maurizio Porfiri, and Sauro Succi. Extreme flow simulations reveal skeletal adaptations of deep-sea sponges. *Nature*, 595(7868):537–541, 2021.
[12] Derek Groen, James Hetherington, Hywel B. Carver, Rupert W. Nash, Miguel O. Bernabeu, and Peter V. Coveney. Analysing and modelling the performance of the HemeLB lattice-Boltzmann simulation environment. *Journal of Computational Science*, 4(5):412–422, 2013.
[13] Christoph A Niedermeier, Christian F Janßen, and Thomas Indinger. Massively-parallel multi-GPU simulations for fast and accurate automotive aerodynamics. In *7th European Conference on Computational Fluid Dynamics*, 2018.
[14] Martin Bauer, Harald Köstler, and Ulrich Rüde. lbmpy: A flexible code generation toolkit for highly efficient lattice Boltzmann simulations. *arXiv preprint arXiv:2001.11806*, 2020.
[15] Intel Corporation. *Intel® oneAPI Programming Guide*, 2022.1 edition, Dec 2021.
[16] NVIDIA Corporation. *CUDA C Programming Guide*, August 2022.
[17] Khronos OpenCL Working Group. *The OpenCL Specification*, May 2022.
[18] Peter Bailey, Joe Myre, Stuart DC Walsh, David J Lilja, and Martin O Saar. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *2009 International Conference on Parallel Processing*, pages 550–557. IEEE, 2009.
[19] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers & Mathematics with Applications*, 61(12):3628–3638, 2011.
[20] Senyou An, Huidan Whitney Yu, and Jun Yao. Gpu-accelerated volumetric lattice Boltzmann method for porous media flow. *Journal of Petroleum Science and Engineering*, 156:546–552, 2017.
[21] Christoph Riesinger, Arash Bakhtiari, Martin Schreiber, Philipp Neumann, and Hans-Joachim Bungartz. A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters. *Computation*, 5(4):48, 2017.
[22] Enrico Calore, Alessandro Gabbana, Jiri Kraus, Elisa Pellegrini, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Massively parallel lattice–Boltzmann codes on large gpu clusters. *Parallel Computing*, 58:1–24, 2016.
[23] Vadim Levchenko, Anastasia Perepelkina, and Andrey Zakirov. LRnLA lattice Boltzmann method: A performance comparison of implementations on GPU and CPU. In M Sokolinsky, L; Zymbler, editor, *Parallel Computational Technologies. PCT 2019. Communications in Computer and Information Science*, volume 1063, pages 139–151, Cham, 2019. Springer.
[24] ORNL, Oak Ridge National Laboratory. *Crusher Quick-Start Guide*, Jan 2022.
[25] Fabio Bonaccorso, Marco Lauricella, Andrea Montessori, Giorgio Amati, Massimo Bernaschi, Filippo Spiga, Adriano Tiribocchi, and Sauro Succi. LBcuda: a high-performance cuda port of lbsoft for simulation of colloidal systems, 2021.
[26] Gabriel Freytag, Matheus S. Serpa, João V.F. Lima, Paolo Rech, and Philippe O.A. Navaux. Collaborative execution of fluid flow simulation using non-uniform decomposition on heterogeneous architectures. *Journal of Parallel and Distributed Computing*, 152:11–20, 2021.
[27] Enrico Calore, Alessandro Gabbana, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Optimization of lattice Boltzmann simulations on heterogeneous computers. *The International Journal of High Performance Computing Applications*, 33(1):124–139, 2019.
[28] Christos Kotsalos, Jonas Latt, Joel Beny, and Bastien Chopard. Digital blood in massively parallel CPU/GPU systems for the study of platelet transport. *Interface Focus*, 11(1):20190116, 2021.
[29] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual

performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[30] Martin Geier, Andreas Greiner, and Jan G Korvink. Cascaded digital lattice Boltzmann automata for high Reynolds number flow. *Physical Review E*, 73(6):066705, 2006.

[31] Christophe Coreixas, Bastien Chopard, and Jonas Latt. Comprehensive comparison of collision models in the lattice Boltzmann framework: Theoretical investigations. *Physical Review E*, 100(3):033305, 2019.

[32] Martin Bauer, Sebastian Eibl, Christian Godenschwager, Nils Kohl, Michael Kuron, Christoph Rettinger, Florian Schornbaum, Christoph Schwarzmeier, Dominik Thönnes, Harald Köstler, et al. waLBerla: A block-structured high-performance framework for multiphysics simulations. *Computers & Mathematics with Applications*, 81:478–501, 2021.

[33] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *ACM Sigplan Notices*, volume 26, pages 30–44. ACM, 1991.

[34] Johannes Habich, T Zeiser, G Hager, and G Wellein. Enabling temporal blocking for a lattice Boltzmann flow solver through multicore-aware wavefront parallelization. In *21st International Conference on Parallel Computational Fluid Dynamics*, pages 178–182, 2009.

[35] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep 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*, pages 1–13. IEEE, 2010.

[36] Matthias Korch and Tim Werner. Improving locality of explicit one-step methods on GPUs by tiling across stages and time steps. *Future Generation Computer Systems*, 2019.

[37] Toshio Endo, Hiroko Midorikawa, and Yukinori Sato. *Software Technology That Deals with Deeper Memory Hierarchy in Post-petascale Era*, pages 227–248. Springer Singapore, Singapore, 2019.

[38] Toshio Endo. Applying recursive temporal blocking for stencil computations to deeper memory hierarchy. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 19–24. IEEE, 2018.

[39] Francois Irigoin and Remi Triolet. Supernode partitioning. In *Symposium on Principles of Programming Languages (POPL'88)*, pages 319–328, San Diego, CA, January 1988.

[40] Paul Feautrier. Some efficient solutions to the affine scheduling problem: Part I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[41] David Wonnacott. Time skewing for parallel computers. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 477–480. Springer, 1999.

[42] David G Wonnacott and Michelle Mills Strout. On the scalability of loop tiling techniques. *IMPACT 2013*, page 3, 2013.

[43] David Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):181–221, 2002.

[44] Michael Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986.

[45] Madhurima Vardhan, John Gounley, Luiz Hegele, Erik W Draeger, and Amanda Randles. Moment representation in the lattice Boltzmann method on massively parallel hardware. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–21, 2019.

[46] Thomas Zeiser, Gerhard Wellein, Aditya Nitsure, Klaus Iglberger, U Rude, and Georg Hager. Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method. *Progress in Computational Fluid Dynamics, an International Journal*, 8(1-4):179–188, 2008.

[47] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 579–586. IEEE, 2009.

[48] Vadim Levchenko and Anastasia Perepelkina. Locally recursive non-locally asynchronous algorithms for stencil computation. *Lobachevskii Journal of Mathematics*, 39(4):552–561, 2018.

[49] Anastasia Perepelkina and Vadim Levchenko. Synchronous and asynchronous parallelism in the LRnLA algorithms. In Leonid Sokolinsky and Mikhail Zymbler, editors, *Parallel Computational Technologies. PCT 2020. Communications in Computer and Information Science*, volume 1263, pages 146–161, Cham, 2020.

[50] Andrey Zakirov, Anastasia Perepelkina, Vadim Levchenko, and Sergey Khilkov. Streaming techniques: Revealing the natural concurrency of the lattice Boltzmann method. *The Journal of Supercomputing*, 77(10):11911–11929, 2021.

[51] Vadim Levchenko, Anastasia Perepelkina, and Andrey Zakirov. New compact streaming in LBM with ConeFold LRnLA algorithms. In Sergey Sobolev Vladimir Voevodin, editor, *Supercomputing. RuSCDays 2020. Communications in Computer and Information Science*, volume 1331, pages 50–62, 2020.

[52] Anastasia Perepelkina, Vadim Levchenko, and Andrey Zakirov. Extending the problem data size for GPU simulation beyond the GPU memory storage with LRnLA algorithms. *Journal of Physics: Conference Series*, 1740:012054, 2021.

[53] Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. Comparison of different propagation steps for lattice Boltzmann methods. *Computers & Mathematics with Applications*, 65(6):924–935, 2013.

[54] Keijo Mattila, Jari Hyväluoma, Jussi Timonen, and Tuomo Rossi. Comparison of implementations of the lattice-Boltzmann method. *Computers & Mathematics with Applications*, 55(7):1514–1524, 2008.

[55] Nissa Osheim, Michelle Mills Strout, Dave Rostron, and Sanjay Rajopadhye. Smashing: Folding space to tile through time. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 80–93, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[56] Anastasia Perepelkina and Vadim Levchenko. LRnLA algorithm ConeFold with non-local vectorization for LBM implementation. In Vladimir Voevodin and Sergey Sobolev, editors, *Supercomputing. RuSCDays 2018. Communications in Computer and Information Science*, volume 965, pages 101–113, Cham, 2019.

[57] Aravind Acharya and Uday. Bondhugula. PLUTO+: Near-complete modeling of affine transformations for parallelism and locality. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[58] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 319–329, New York, NY, USA, 1988. Association for Computing Machinery.

[59] Adrian Kummerländer, Marcio Dorn, Martin Frank, and Mathias Krause. Implicit propagation of directly addressed grids in lattice Boltzmann methods. 08 2021.

[60] Mark J. Mawson and Alistair J. Revell. Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture NVIDIA GPUs. *Computer Physics Communications*, 185(10):2566–2574, 2014.

[61] Markus Mohrhard, Gudrun Thäter, Jakob Bludau, Bastian Horvat, and Mathias J. Krause. Auto-vectorization friendly parallel lattice Boltzmann streaming scheme for direct addressing. *Computers & Fluids*, 181:1–7, 2019.

[62] Anastasia Perepelkina and Vadim Levchenko. Functionally arranged data for algorithms with space-time wavefront. *Parallel Computational Technologies. PCT 2021. Communications in Computer and Information Science*, 1437:134–148, 2021.

[63] Peter Bailey, Joe Myre, Stuart DC Walsh, David J Lilja, and Martin O Saar. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 550–557. IEEE, 2009.

[64] Martin Geier and Martin Schönherr. Esoteric twist: An efficient in-place streaming algorithms for the lattice Boltzmann method on massively parallel hardware. *Computation*, 5(2):19, 2017.

[65] Maoqiang Jiang, Jing Li, and Zhaohui Liu. Efficient implementation of immersed boundary-lattice Boltzmann method for massive particle-laden flows part I: Serial computing. *arXiv preprint arXiv:2002.08855*, 2020.

[66] Maciej Matyka and Michał Dzikowski. Memory-efficient lattice Boltzmann method for low Reynolds number flows. *Computer Physics Communications*, 267:108044, 2021.