Journal of Parallel and Distributed Computing 206 (2025) 105169

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

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

Optimization of single node load balancing for lattice Boltzmann method on heterogeneous high performance computers

Adrian Kummerländer [a,c,*], Fedor Bukreev [b,c], Dennis Teutscher [b,c], Marcio Dorn [d], Mathias J. Krause [a,b,c]

[a] Institute of Applied and Numerical Mathematics, Karlsruhe Institute of Technology, Karlsruhe, Germany
[b] Institute for Mechanical Process Engineering and Mechanics, Karlsruhe Institute of Technology, Karlsruhe, Germany
[c] Lattice Boltzmann Research Group, Karlsruhe Institute of Technology, Karlsruhe, Germany
[d] Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil

ARTICLE INFO

Keywords:
High performance computation
Lattice Boltzmann methods
Spatial domain decomposition
Genetic programming
Load balancing
Heterogeneous computation

ABSTRACT

Lattice Boltzmann Methods (LBM) are particularly suited for highly parallel computational fluid dynamics simulations on heterogeneous HPC systems combining CPUs and GPUs. However, the computationally dominant collide-and-stream loops commonly utilize only GPUs, leaving CPU resources underutilized. To overcome this limitation, this article proposes a novel load balancing strategy based on a genetic algorithm for bottom-up, cost-aware optimization of spatial domain decompositions. This approach generates subdomains and rank assignments inherently suited for cooperative execution on both CPUs and GPUs. Implemented in the open source framework OpenLB, the strategy is applied to turbulent flow reference cases, including a multi-physics reactive mixer. A detailed evaluation on heterogeneous HPC nodes demonstrates significant performance gains, achieving speedups of up to 87% compared to traditional GPU-only execution. This work therefore establishes cost-aware, bottom-up decomposition as a suitable strategy for exploiting the native heterogeneity of modern compute nodes.

1. Introduction

Heterogeneous computation utilizes exceptional features of the available hardware to improve application performance and energy usage [1–3]. Many modern *high performance computers* (HPC) are heterogeneous systems that combine e.g. *central processing units* (CPU) and *general purpose graphics processing units* (GPGPU) alongside dedicated high-speed interconnects. This trend extends to the world's fastest supercomputers of which one-third utilize accelerating co-processors alongside their primary CPUs [4].

Even single CPUs can be considered heterogeneous systems as their instruction sets commonly offer multiple different options for the same computation (e.g. scalar and vector arithmetic). This also extends to on-die GPUs in consumer offerings, *high bandwidth memory* (HBM) in dedicated HPC CPUs such as Intel Xeon Phi, or separate efficiency and performance cores.

The *lattice Boltzmann method* (LBM) is a mesoscopic approach to the simulation of various transport phenomena in *computational fluid dynamics* (CFD) [5–7]. Common target equations include the *Navier-Stokes* (NSE), *(reaction-)advection-diffusion* ((R)ADE) and radiative transport equations [8–12]. LB methods are particularly suited to massively parallel processing due to their algorithmic structure which can be split into a perfectly parallel *collision step* and a neighborhood local *streaming step*.

The utilization of (un)vectorized CPUs and GPGPU accelerators is well established [13–17] in LBM-based simulations. However, most common LB load balancing approaches homogeneously process the lattice either only on CPUs or only on GPUs while their respective CPUs are relegated to communication and input/output tasks. A spatial decomposition into a homogeneously sized subdomains follows from this approach.

As GPUs are co-processors that do not stand on their own but are attached to and tasked by CPUs, the latter are responsible for managing the overall execution flow, load balancing, and communication. As such, LB codes that homogeneously utilize only GPUs for the actual LB algorithm are already heterogeneous to a degree, especially if combined with asynchronous, CPU-based post processing and *input/output* (IO).

* Corresponding author at: Institute of Applied and Numerical Mathematics, Karlsruhe Institute of Technology, Karlsruhe, Germany.
E-mail address: adrian.kummerlaender@kit.edu (A. Kummerländer).

https://doi.org/10.1016/j.jpdc.2025.105169
Received 9 January 2024; Received in revised form 7 July 2025; Accepted 11 August 2025

Available online 21 August 2025

Full heterogeneous load balancing of LBM, i.e. splitting the collide-and-stream load between CPUs and GPUs, has been investigated previously [14, 18–22]. However, these approaches were restricted to either local axis-aligned fractional splits of a given homogeneous decomposition or processing overlap areas on the CPU [22].

The present work aims to extend load balancing of LBM to *bottom-up* decompositions using a local cost-estimate in a comprehensive heterogeneous load balancing approach enabling *cooperative* utilization of both SIMD CPUs and GPUs. That is, a heterogeneous cost-estimate of the simulation domain is used to guide the decomposition to be inherently suited for heterogeneous processing. For this purpose, the process of covering the simulation domain by non-intersecting cuboids is formulated as an optimization problem to be solved using a novel *genetic algorithm* (GA). Speedups up to 87% showcase that load balancing of LBM on heterogeneous hardware is not a settled question and warrants further investigation. The specific GA for block-structured domain decompositions opens up a new approach as a foundation for future work.

This work focuses on performance optimization within single, resource-constrained compute nodes that employ block-structured domain decompositions. The approach's effectiveness, and thus its resulting performance advantage, is inherently dependent on pronounced spatial cost-inhomogeneities, as the benefit stems from offloading regions where the relative advantage of the GPU is diminished.

At the top level, the present paper is structured into a method (Section 2) and an evaluation part (Section 3). The first part provides relevant details of the LB method (Section 2.1), its software implementation (Section 2.2), a detailed discussion of the spatial domain decomposition algorithm (Section 2.3) as well as rank assignment strategies (Section 2.4). The second part first describes the reference cases (Section 3.1) after which the genetic algorithm (Section 3.2) and the resulting simulation performance (Section 3.3) are evaluated in detail.

2. Methods

2.1. Lattice Boltzmann methods

Following the separation into local and non-local steps, the streaming step propagates information largely independent of the modeled physics while the specific transport phenomena is captured by the choice of collision operator and equilibrium distribution [5].

Definition 1 *(Collision step).* The relaxation of Q population values f_i towards a local equilibrium distribution f_i^{eq} according to a collision operator Ω

$$f_i^{\mathrm{post}}(x,t) = \Omega\left(f_i(x,t), f_i^{\mathrm{eq}}(x,t)\right)$$

is referred to as the *collision* step. This computation maps *pre-* to *post-collision* populations f_i^{post}.

A common choice for Ω is the BGK operator [23] with single relaxation time $\tau > 0.5$

$$\Omega = f_i(x,t) - \frac{1}{\tau}(f_i(x,t) - f_i^{\mathrm{eq}}(x,t)).$$

In combination with a formulation of the *Maxwell-Boltzmann equilibrium* f_i^{eq}, the moments of f_i can be shown to converge to solutions of the incompressible *Navier-Stokes equations* [5] (NSE)

$$\partial_t \rho + \nabla \cdot (\rho u) = 0,$$
$$\partial_t u + (u \cdot \nabla)u = -\frac{1}{\rho}\nabla p + 2\nu\nabla \cdot \Pi$$

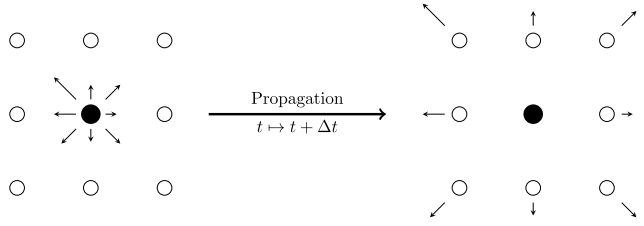
with macroscopic density ρ, velocity u, pressure p, kinetic viscosity ν and strain tensor Π. The BGK relaxation time τ is directly related to the modeled viscosity by $\nu = c_s^2(\tau - {}^{\Delta t}/_2)$. Other choices for collision operators and equilibrium distributions are possible, providing models for a wide range of transport phenomena [8–12]. In any case, the discretization of the collision operator depends on the choice of discrete velocities. A common set for three-dimensional NSE as a target equation is *D3Q19*

$$\{\xi_i\}_{i=0}^{18} = \left\{\xi \in \{-1,0,1\}^3 \,\middle|\, \exists j \in \{0,1,2\} : \xi_j = 0\right\}.$$

Definition 2 *(Streaming step).* Communication of post-collision populations f_i^{post} to neighboring cells

$$f_i(x + \xi_i, t + \Delta t) = f_i^{\mathrm{post}}(x,t)$$

according to discrete velocities ξ_i is called the *streaming* or *propagation* step.



In the context of LBM, load balancing consists of both the choice of spatial domain decomposition and the assignment of the resulting subdomains to the available set of processing units. Further details of the concrete case-specific collision operator and boundary conditions will be provided in Section 3.1. We emphasize that the approaches discussed in the remainder of this work apply to any block-structured LBM implementation independent of specific stencil, collision operator, equilibrium distribution and modeled physical phenomena.

2.2. Software context

The software foundation for the present work is provided by the open source LBM framework OpenLB [24] in version 1.6 [25]. OpenLB has received significant refactoring and performance engineering effort [26–28] over the last years, extending its previous CPU-only hybrid parallelization [24,29] to comprehensive support for NVIDIA GPUs [30] as well as vectorization on SIMD CPUs [17].

We note that the methods of the present paper translate to other openly available general purpose block-structured LBM frameworks with support for heterogeneous hardware such as Walberla [16], Palabos [31], and HemeLB [32], due to their similar top-level software architecture.

2.2.1. Heterogeneity in OpenLB

OpenLB's transparent heterogeneous hardware support is based on the implementation of collision-, non-local boundary- and multo-lattice coupling operators as abstract C++ template functions. These functions are instantiated on platform-specific types and data structures, e.g. using a vector pack wrapping SIMD intrinsics or GPU-side lattice structures. This framework also integrates automatic code generation for symbolic common subexpression elimination. All target platforms utilize the vectorization and GPU-friendly implicit *Periodic Shift* (PS) propagation pattern [17].

The top-level MPI parallelization in OpenLB follows a standard spatial decomposition scheme resulting in the assignment of directly addressed blocks to individual processes. Each block of this decomposition may be processed by different hardware and parallelization approaches, independent of any neighboring blocks. The set of per-block targets currently includes (un)vectorized CPU and NVIDIA GPU platforms (`CPU_SISD`, `CPU_SIMD` and `GPU_CUDA` respectively). CPU targets may optionally utilize OpenMP for shared memory parallelization. The resulting *hybrid* mode combining MPI for inter-block and OpenMP for intra-block parallelization was previously found to provide superior performance to MPI-only parallelization [28,29]. As of version 1.6, all supported per-block platforms can be heterogeneously assigned at the load balancing stage. The library transparently handles any resulting inter-block communications, providing the necessary foundation for the present work.

2.2.2. Load balancing in OpenLB

OpenLB's load balancing stage is split into the initial domain decomposition phase followed by the assignment to the available set of MPI ranks. Both steps may independently follow different algorithms, although the quality of the resulting balancing depends on both of them working together. This basic architecture was previously described in [24,29] with respect to unvectorized CPU-only execution. Additionally, graph-based approaches were applied [33] to optimize CPU-only balancing in complex geometries.

The present work extends OpenLB's load balancing to heterogeneous computation environments, specifically hybrid CPU-GPU utilization beyond pure offloading. Heterogeneous communication between ranks is handled via CUDA-aware MPI in the existing overlap synchronization pattern based on persistent asynchronous requests. While this approach is sufficient for intra-rank heterogeneous communication, additional CPU-GPU communication kernels were implemented due to better performance characteristics.

OpenLB supports XML (de)serialization of cuboid geometries, enabling the import of *externally* generated spatial decompositions. For the present work, the decomposition algorithm was implemented using the Python programming language together with the PyVista [34], SciPy [35] and PyGAD [36] libraries. The algorithm is implemented as an interactive workflow using the Jupyter computation environment.

2.3. Spatial domain decomposition

In a homogeneous setting, each parallel processing unit provides the same computational capability and should receive an equal part of the total problem. This is contrasted by the heterogeneous computational capabilities constituting a heterogeneous system and motivating non-equal work distribution. In any case, non-shared-memory parallelization in LBM necessarily relies on dividing the spatial domain into parts and distributing those parts to different processing units [5]. Common approaches to this decomposition include various cartesian slicing or bisection [37,38] or graph partitioning algorithms [39,40].

In order to meet the software context described in Section 2.2, spatial domain decompositions are restricted to *block decompositions*, matching OpenLB's parallelization primitives [29,30]. As such the *parts* mentioned in the previous paragraph are restricted to non-overlapping *cuboids* that together fully cover the spatial domain (cf. Figs. 9 and 11 for illustration).

Definition 3 *(Spatial Domain Decomposition).* Let $g : \mathbb{R}^3 \to \{0,1\}$ be a domain indicator function s.t. $g(x) = 1 \iff x \in D$ for the spatial simulation domain $D \subset \mathbb{R}^3$. Then we consider any set $\mathcal{D} := \{D_1, \dots, D_n\}$ of axis-aligned *cuboids*

$$D_i := \{x \in \mathbb{R}^3 | d_{\text{lower}} \leq x < d_{\text{upper}}\} \text{ for given } d_{\text{lower}} < d_{\text{upper}} \in \mathbb{R}^3 \tag{3.1}$$

that fully covers the spatial domain s.t.

$$\forall D_i, D_j \in \mathcal{D} : D_i \cap D_j = \emptyset \tag{3.2}$$

$$D \subseteq \bigcup_{D_i \in \mathcal{D}} D_i \tag{3.3}$$

to be a valid decomposition of the spatial simulation domain.

For the present work, heterogeneous block decompositions split the spatial simulation domain into a set of blocks that is heterogeneous w.r.t. the individual computational weights such that there is a bijection between the set of heterogeneous processing units and blocks. The per-block computational weight is a function of the cell count and complexity of the applied operators, including collision steps, coupling and non-local boundary conditions. In any case, side conditions include a minimal communication surface and communication locality.

Different from previous work [14,19–21] where each block of a homogeneous decomposition is *heterogenified* by slicing it into CPU- and GPU-parts, the present approach constructs heterogeneous block decompositions bottom-up. This bottom-up approach is based on a generic geometry indicator in conjunction with a computational cost estimate that marks GPU-unfriendly areas. Such cost estimates can either be provided as domain knowledge or be automatically inferred from minimal benchmark simulations in a auto-tuning setting.

Definition 4 *(Discretization of Spatial Simulation Domain).* Let $g : \mathbb{R}^3 \to \{0,1\}$ be an indicator function s.t. $g(x) = 1 \iff x \in D$ for spatial simulation domain $D \subset \mathbb{R}^3$. Domain D is *coarsely* discretized into a regular lattice indicated using

$$g_\# : \mathbb{Z}^3 \to \{0,1\}, x \mapsto \begin{cases} 1 & \int_{C_x} g(s)ds > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

for cell cuboids of coarse discretization spacing $\delta_b \gg \delta_x \in \mathbb{R}^+$

$$C_x := \{s \in \mathbb{R}^3 | s \geq x \wedge s \leq x + \delta_b\} \tag{4.2}$$

where δ_x is the spatial simulation discretization factor and

$$\mathcal{C}_\# := \{C_x | x \in \mathbb{Z}^3\} \tag{4.3}$$

is the set of all coarse cell cuboids.

Definition 5 *(Computational Cost Estimate).* In order to describe the spatially heterogeneous computational effort required to simulate a given coarsely discretized domain following Definition 4, a local computational weight estimate

$$p : \mathbb{Z}^3 \to \mathbb{N}_0 \text{ where } p(x) > p(y) \iff \text{simulation at } x \text{ is more costly than at } y \tag{5.1}$$

is used. Any coarse discretization cuboid C_x for $x \in \mathbb{Z}^3$ with $g_\#(x) = 0$ is estimated to have cost $p(x) = 0$.

Definition 6 *(Coarse Spatial Domain Decomposition).* Given a discretized spatial simulation domain with associated coarse cell cuboid set $\mathcal{C}_\#$ following Definition 4 we restrict the considered set of spatial domain decompositions to

$$\mathcal{D}_\# \subset \left\{ D_i \subset \mathbb{R}^3 \middle| \exists \mathcal{C}'_\# \subset \mathcal{C}_\# : D_i = \cup_{C_x \in \mathcal{C}'_\#} C_x \right\} \tag{6.1}$$

that are valid under Definition 3. Such decompositions $\mathcal{D}_\#$ are referred to as *coarse*.

Using the coarsely discretized and computationally-weighted domain indicator $g_\#$, generic algorithms can be applied to group the individual coarse cells into larger cuboids, i.e. to construct coarse spatial domain decompositions. The following optimization algorithm will be guided by per-cuboid and global properties cf. Definitions 7 and 8. Its target optimization problem is to find a coarse spatial domain decomposition following Definition 6 with maximum fitness as evaluated using Function (11.1).

Definition 7 *(Cuboid Properties).* Let $D_i \in \mathcal{D}_\#$ be a cuboid of coarse spatial decomposition $\mathcal{D}_\#$. For each such cuboid $D_i := \{C_x, \dots\} \subset \mathcal{C}_\#$ we define the cuboid indicator function as

$$g_{\#D_i} : \mathbb{Z}^3 \to \{0,1\}, x \mapsto \begin{cases} 1 & C_x \in D_i \\ 0 & \text{otherwise} \end{cases}. \tag{7.1}$$

Note that cuboids D_i may cover regions outside of the coarsely discretized simulation domain leading to the definition of their *volume fraction*

$$v(D_i) := \frac{\sum_{x \in \mathbb{Z}^3} g_\#(x) g_{\#D_i}(x)}{\sum_{x \in \mathbb{Z}^3} g_{\#D_i}(x)}. \tag{7.2}$$

Using the computational cost estimate p, the *computational cost set* of D_i is defined as

$$P(D_i) := \{n \in \mathbb{N}_0 | \exists x \in \mathbb{Z}^3 : g_{\#D_i}(x) = 1 \wedge p(x) = n\}. \tag{7.3}$$

Based on this, the *internal heterogeneity* is quantified using

$$h(D_i) := \begin{cases} \sum_{c \in P(D_i)} c & |P(D_i)| > 1 \\ 1 & \text{otherwise} \end{cases}. \tag{7.4}$$

Definition 8 *(Communication Surface).* The individual *communication surface* of cuboids in a given spatial decomposition $\mathcal{D}_\#$ is defined as

$$S(D_i) := \{C_x \in \mathcal{C}_\# | C_x \in D_i \wedge \exists \xi \in \Xi, j \neq i : C_{x+\delta} \in D_j\} \tag{8.1}$$

and quantified globally for the entire decomposition $\mathcal{D}_\#$ using

$$s(\mathcal{D}_\#) := \sum_{D_i \in D} |S(D_i)|. \tag{8.2}$$

This surface measures the amount of data that needs to be communicated between every timestep.

Definition 9 *(Seeded Spatial Decomposition).* A spatial decomposition $\mathcal{D}_\#$ consisting of $n \in \mathbb{N}$ cuboids $D_1, \dots, D_n$ satisfying constraint

$$v(D_i) \geq v_{\min} \text{ for given parameter constant } v_{\min} \in (0,1] \tag{9.1}$$

is constructed from a set of $m \in \mathbb{N}$ seed pairs

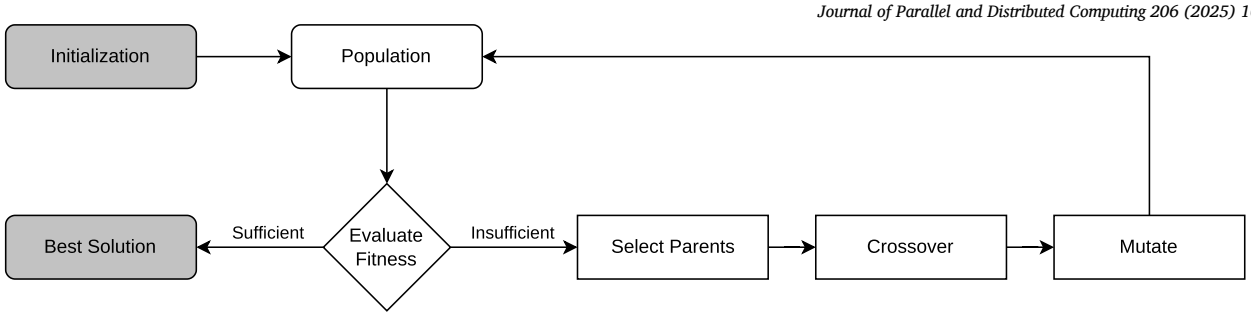


Fig. 1. Structure of a generic genetic algorithm.

$$\mathcal{S} := \{(S_1, v_1), \dots, (S_m, v_m)\} \text{ for } S_i \in \mathbb{Z}^3, v_i \in [v_{\min}, 1] \wedge g_\#(S_i) = 1 \tag{9.2}$$

following high-level construction rules:

1. Iterate over all active seeds and their associated cuboids until no more growth is possible.
2. Try to grow current cuboid in best possible direction w.r.t. volume fraction, internal heterogeneity.
3. Construct single cuboids to cover all not-yet-covered coarse cells.

The full construction algorithm following these rules is provided in Listing 2.

2.3.1. Optimization of decompositions

This section describes the specific *genetic algorithm* (GA) [41] based optimization approach that is employed to iteratively optimize spatial block decompositions of a given computationally-weighted domain indicator w.r.t. a fitness function. An outline of a generic GA's structure is provided in Fig. 1.

We state the abstract goals of the outlined algorithm as: to decompose the domain into a *minimal* number of high-volume-fraction cuboids with internally low variation of estimated computational cost. Notably, the number of decomposing cuboids is not an input but a result of the optimization process. These properties are desireable to minimize the memory footprint and communication overhead of simulations based on the results of this approach.

Definition 10 *(Genes and Chromosomes).* The foundation of any GA is the choice of chromosomes and their constituting genes on top of which fitness, mutation and crossover functions are applied [41]. Let a *gene* G_i be a tuple of seed position, minimum volume fraction and active state

$$G_i := (S_i, v_i, a_i) \text{ for } S_i \in \mathbb{Z}^3 : g_\#(S_i) = 1, v_i \in [v_{\min}, 1], a_i \in \{0, 1\} \tag{10.1}$$

and a *chromosome* be a list of $n_{\text{seeds}} \in \mathbb{N}$ genes. Each chromosome is associated with a unique spatial decomposition following Definition 9 and a fitness value computed using Function 11.1.

Based on this foundation, the fitness function in Definition 11 scores the quality of chromosomes that are generated by successive crossover and mutation functions in Definitions 12 and 13.

Definition 11 *(Fitness Function).* We define the minimization of the internal heterogeneity of a given spatial decomposition $\mathcal{D}_\# := \{D_1, \dots\}$ constructed following Definition 9 as the primary optimization goal, encoded by the fitness function

$$f(\mathcal{D}) := 2 - 2 \underbrace{\sum_{D_i \in \mathcal{D}} h(D_i)}_{\text{Heterogeneity}} - \underbrace{\frac{\sum_{D_i \in \mathcal{D}} 1 - v(D_i)}{|\mathcal{D}|}}_{\text{Volume fraction}} - \underbrace{\frac{s(\mathcal{D})}{s(\{C_x \in C_\# | \exists D_i \in \mathcal{D} : C_x \in D_i\})}}_{\text{Communication surface related to worst-case}} \leq 0 \tag{11.1}$$

with secondary goals being the minimization of the communication surface and the maximization of the aggregated per-cuboid volume fractions. An ideal decomposition $\mathcal{D}_{\text{ideal}}$ into a single cuboid with volume fraction 1, internally homogeneous cost and zero communication yields a fitness of $f(\mathcal{D}_{\text{ideal}}) = 0$.

As the first step in each new generation (cf. Fig. 1), the crossover function recombines the parents with highest fitness into new chromosomes. It also keeps the selected parents in the population to be mutated without recombination as an additional exploration vector.

Definition 12 *(Crossover Function).* An adapted variant of the uniform crossover approach [41] is used to recombine pairs of parent chromosomes selected via steady state into offspring. Specifically, the common per-gene uniform crossover is modified to preserve uniqueness of genes inside the offspring chromosome. Further, the offspring pool is initialized to contain unmodified copies of all parent chromosomes. The concrete implementation of this function is provided in Listing 3.

In addition to randomly replacing a number of seeds, the mutation function also randomly (de)activates a number of seeds with a generation-dependent probability. This additional complexity is intended to control the minimization pressure on the number of seeds to be very high for early generations while continuously decreasing for later generations to increase exploration of different cuboid configurations while still allowing for escape from low-seed-count local minima towards higher-seed-count chromosomes with better fitness.

Definition 13 *(Mutation Function).* An adaptive mutation [42,41] approach is used to increase the mutation probabilities for *below-mean-fitness* chromosomes. Specifically, only one seed is mutated for above-mean-fitness chromosomes while half of all seeds are mutated for all other chromosomes, including a configurable fraction of active seeds. For each selected gene there are two possible mutation operations: The mutation of its active state or the mutation of its seed position. The decision of which operation to apply is controlled by a threshold function

$$p_{\text{flip}}(i) := p_{\text{minFlip}} + (p_{\text{maxFlip}} - p_{\text{minFlip}}) \exp(-0.05i) \in [p_{\text{minFlip}}, p_{\text{maxFlip}}] \tag{13.1}$$

that yields a generation-dependently decreasing probability of mutating the active state. The mutatation of a gene's seed position consists of randomly sampling a replacement from the cost-weighted pool of positions not taken by other seeds of the same chromosome and by randomly sampling a new uniformely distributed individual minimum volume fraction in $[v_{\min}, 1]$. The concrete implementation of this function is provided in Listing 4.

Definitions 10, 11, 13 and 12 are implemented in the Python programming language using PyGAD [36] to form a complete genetic algorithm for heterogeneity-aware spatial block decompositions. The algorithm's performance when applied to reference problems alongside the specific choice of tuneable parameters such as the minimum volume fraction, population size, number of parents, and number of generations will be discussed in Section 3.2.

2.4. Rank assignment

Given a spatial decomposition according to Definition 3, the individual subdomains need to be assigned to ranks in the set of available processing units. For the heterogeneous case, the execution platform needs to be assigned in addition to the rank as each rank potentially offers multiple different options.

Definition 14 *(Rank Assignment).* Let $\mathcal{D} := \{D_1, \dots, D_n\}$ be the set of cuboids and

$$\mathcal{R} \subseteq \{R_1, \dots, R_m\} \times \{P_1, \dots, P_l\}$$

the set of available rank-platform tuples. Then $r : \mathcal{D} \to \mathcal{R}$ is a rank assignment of spatial decomposition $\mathcal{D}$.

The *heterogeneous* balancing approach maps a given decomposition onto CPU-GPU heterogeneous ranks, i.e. all ranks are expected to have both CPU and GPU compute units available.

Definition 15 *(Heterogeneous Rank Balancing).* Let $\mathcal{D} := \{D_1, \dots, D_n\}$ be the set of cuboids and

$$\mathcal{R} := \{R_1, \dots, R_m\} \times \{\texttt{GPU}, \texttt{CPU}\}$$

the set of available rank-platform tuples and $f_{\texttt{GPU}} \in (0, 1]$ the targeted GPU volume fraction. It is assumed that $n \geq m$, i.e. there is at least one cuboid per rank. The series of cuboids in by-volume descending order

$$\begin{aligned} (D)_1 &:= \arg\max_{D \in \mathcal{D}} \text{volume}(D) \\ (D)_i &:= \arg\max_{D \in \mathcal{D} \setminus \mathcal{D}_{i-1}} \text{volume}(D) \qquad \text{where } \mathcal{D}_i := \cup_{j=0,\dots,i} (D)_j \end{aligned} \tag{15.1}$$

is split at index

$$s := \min\left\{ j \;\middle|\; \sum_{D \in \mathcal{D}_j} \text{volume}(D) \geq f_{\texttt{GPU}} \sum_{D \in \mathcal{D}} \text{volume}(D) \right\} \tag{15.2}$$

into CPU- and GPU-eligible sections. GPU-eligible cuboids $(D)_0, \dots, (D)_s$ are iteratively assigned to the GPU platform while preferring the currently lowest utilized rank

$$\begin{aligned} (P)_i &:= \texttt{GPU} && \text{for } i \leq s \\ (R)_1 &:= R_1 \\ (R)_i &:= \arg\min_{R \in \mathcal{R}} \left(\sum_{j=1}^{i-1} \mathbb{1}_{(R)_j}(R) \, \text{volume}((D)_j) \right) && \text{for } i \leq s. \end{aligned}$$

In case $s < n$ the remaining CPU-eligible cuboids $(D)_{s+1}, \dots, (D)_n$ are assigned to their closest GPU-neighbor rank. This closest neighbor is determined as the rank with the highest number of GPU-assigned cuboids that neighbor the given CPU-eligible cuboid. The platform and rank series are continued by

$$\begin{aligned} (P)_i &:= \texttt{CPU} && \text{for } s < i \leq n \\ (R)_i &:= \arg\max_{R \in \mathcal{R}} \left(\sum_{D^* \in \text{neighbors}((D)_i)} \sum_{j=1}^{s} \mathbb{1}_{(R)_j}(R) \mathbb{1}_{(D)_j}(D^*) \right) && \text{for } s < i \leq n \end{aligned}$$

assuming that all CPU-eligible cuboids are direct neighbors of at least one GPU-eligible cuboid which is sufficient for our purposes. In summary we get the *heterogeneous rank balancing* as a rank assignment function

$$r_{\text{heterogeneous}} : \mathcal{D} \to \mathcal{R},$$
$$D \mapsto ((R)_j, (P)_j) \qquad \text{for } j := \arg\max_{i=1}^{n} \mathbb{1}_{(D)_i}(D). \tag{15.3}$$

The heterogeneous rank balancing described in Definition 15 takes into account only a given global GPU volume fraction, individual cuboid volumes and the emergent rank assignment neighborhood. Per-cuboid volumes function as a proxy for execution cost due to the minimization of internal cost-heterogeneity during decomposition (cf. Section 2.3) and the general requirement that high-cost regions make up only a small fraction of the total simulation domain.

As a modification of this strategy, the *orthogonal* approach balances the block set onto ranks that are declared either CPU- or GPU-only at the job scheduling stage, i.e. ranks that are internally homogeneous.

Definition 16 *(Orthogonal Rank Balancing).* Let $\mathcal{D} := \{D_1, \dots, D_n\}$ be the set of cuboids,

$$\mathcal{R} := \{(R_1, \texttt{GPU}), \dots, (R_m, \texttt{GPU}), (R_{m+1}, \texttt{CPU}), \dots, (R_l, \texttt{CPU})\}$$

the set of available rank-platform tuples and $f_{\texttt{GPU}} \in (0, 1]$ the targeted GPU volume fraction. It is assumed that $n \geq l$, i.e. there exist both CPU and GPU ranks and at least one cuboid per rank. Identically to Definition 15 we split the by-volume descending cuboid series (15.1) into GPU- and CPU-eligible sections (15.2) while additionally requiring $m \leq s$ s.t. each GPU-rank receives at least one cuboid. As such the platform $(P)_i$ and rank series $(R)_i$ are identical to Definition 15 for $i \leq s$.

Differently, for $i > s$ we define them as

$$(P)_i := \texttt{CPU} \qquad \text{for } s < i \leq l$$
$$(R)_{s+1} := R_{m+1}$$
$$(R)_i := \arg\min_{R \in \{R_{m+1}, \dots, R_l\}} \left(\sum_{j=s+1}^{l} \mathbb{1}_{(R)_j}(R)\, \text{volume}((D)_j) \right) \qquad \text{for } s+1 < i \leq l,$$

which *orthogonally* assigns all CPU-eligible cuboids to CPU-homogeneous ranks without considering the GPU-eligible section. In summary we get the *orthogonal rank balancing* as a rank assignment function

$$r_{\text{orthogonal}} : \mathcal{D} \to \mathcal{R},$$
$$D \mapsto ((R)_j, (P)_j) \qquad \text{for } j := \arg\max_{i=1}^{n} \mathbb{1}_{(D)_i}(D). \tag{16.1}$$

Both balancing strategies were implemented natively in OpenLB for the present investigation. In order to satisfy the rank requirements for decompositions whose cardinality is not determined by a user-intuited degree of parallelization but by cost-aware optimization (cf. Section 2.3), genetically optimized *base decompositions* are subdivided by an additional deterministic process when necessary. This subdivision applies the iterative by-volume decomposition approach commonly used for OpenLB-based simulations in complex geometries [33].

2.4.1. Resource allocation

In practice, the specific set of rank-platform tuples required for Definition 16's orthogonal rank balancing is realized by allocating only a single CPU core for GPU-enabled processes while assigning the remainder to CPU-only OpenMP-parallelized and vectorized processes that in turn do not get access to a GPU. Listing 1 illustrates how this can be achieved on an `accelerated` node of the HoreKa supercomputer (cf. Section 3.3 and Table 1) consisting of four NVIDIA A100 GPUs and two 38-core Intel Xeon Platinum CPUs.

Listing 1: Excerpt of Resource Allocation for Orthogonally Balanced Single-Node Simulation on HoreKa

```
#!/usr/bin/env bash
#SBATCH --gres=gpu:4
#SBATCH --nodelist=hkn0810
#SBATCH --ntasks-per-node=76

# ranks.txt
# > rank  0=hkn0810 slot=0:36
# > rank  1=hkn0810 slot=0:37
# > rank  2=hkn0810 slot=1:36
# > rank  3=hkn0810 slot=1:37
# > rank  4=hkn0810 slot=0:0-35
# > rank  5=hkn0810 slot=1:0-35

export CUDA_VISIBLE_DEVICES=;
export OMP_NUM_THREADS=1;

mpirun --rankfile ranks.txt \
  -np 4 --bind-to core bash -c 'export CUDA_VISIBLE_DEVICES=${OMPI_COMM_WORLD_LOCAL_RANK}; ./application' \
: -np 2 --bind-to core bash -c 'export OMP_NUM_THREADS=36;                                 ./application'
```

In the authors' experience, resource configurations for simulations beyond basic MPI-only parallelization tend to be more complex than many users are comfortable with, frequently causing subpar performance due to e.g. overscaling. For this reason future work will extend the foundation of cost-aware decomposition and heterogeneous balancing to automatically generating matching scheduling scripts with rank binding and assignment. In this way the simulation user is separated from the performance-critical resource usage decision, allowing future work on automatically selecting a suitable set of resources within given constraints.

In general, we argue that spatial decompositions should not be derived from a user-selected degree of parallelism — rather the parallelization degree and resource usage should be informed by the optimized decomposition in a cost-aware auto tuning approach.

3. Evaluation

This section evaluates the spatial domain decomposition and rank assignment approach presented in Sections 2.3 and 2.4 w.r.t. convergence of the optimization algorithm respectively the resulting performance on heterogeneous reference systems. Both evaluations share common reference cases summarized in Section 3.1.

3.1. Reference cases

In the present work, heterogeneous processing is utilized to improve the execution performance of Lattice Boltzmann-based simulations with local cost inhomogeneities that can be taken into account during cost-aware decomposition. For the reference cases these inhomogeneities are caused due to the usage of a computationally-complex boundary condition for modeling turbulent velocity inlets. The details of this boundary condition are described in Section 3.1.1.

Extending upon the basic LBM scheme introduced in Section 2.1, both reference cases employ a *Large Eddy Simulation* (LES) model that locally modifies the BGK collision's relaxation frequency in order to model the small eddies in the turbulent regime without resolving them.

Definition 17 *(Smagorinsky LES BGK).* Let $\Delta_x > 0$ be the spatial and $\Delta_t > 0$ the temporal discretization parameters. The turbulent viscosity for given Smagorinsky constant $C_S > 0$ is modeled as

$$\nu_{\text{turb}} = (C_S \Delta_x)^2 \overline{\Pi}$$

where the shear rate $\overline{\Pi}$ is recoverable as the second-order moment of the distribution functions' non-equilibrium term. In combination with given global molecular viscosity $\nu_{\text{mol}} > 0$ we get the Smagorinsky LES BGK equation [43–45]

$$f_i(x + \xi_i \Delta_t, t + \Delta_t) - f_i(x,t) = -\frac{\Delta_t}{\tau_{\text{eff}}(x,t)}(f_i - f_i^{eq}) + \Omega$$

where

$$\tau_{\text{eff}}(x,t) := \frac{\nu_{\text{mol}} + \nu_{\text{turb}}(x,t)}{c_s^2} \frac{\Delta_t}{\Delta_x^2} + \frac{1}{2}$$

is the local effective relaxation time.

The LES scheme described in Definition 17 is implemented in OpenLB as the dynamics tuple [46]

```
using SmagorinskyBGK = dynamics::Tuple<
  float, descriptors::D3Q19<>,
  momenta::BulkTuple,
  equilibria::SecondOrder,
  collision::SmagorinskyEffectiveOmega<collision::BGK>
>;
```

for a single precision D3Q19 lattice. This declaration is fully amenable to OpenLB's automatic code generation yielding an efficient *common subexpression elimination* (CSE) optimized kernel for execution on SIMD CPUs and GPUs.

We emphasize that the discussed load balancing approaches are independent of the specific LB models employed by the reference cases and are employable to balance any cost-heterogeneous LB simulation cases targeting heterogeneous execution environments.

3.1.1. Boundary condition

The *Vortex Method* [47][48] (VM) is a resolution-independent turbulent velocity inlet condition that was adapted to LBM in OpenLB for usage in diverse turbulent flow cases, including cases targeted for heterogeneous processing in the present work. An extensive validation study of this method both w.r.t. numerical and experimental data is published separately in [49]. Its basic approach is to pertubate a given mean velocity profile through a fluctuating vorticity field generated from a number of discrete seed points.

Definition 18 *(Vortex Method).* Let $u_{\text{mean}}, u_{\text{prev}} : \mathbb{R}^3 \to \mathbb{R}^3$ be the mean velocity profile and the inflow velocity of the previous timestep, $A_{\text{inlet}} > 0$ the inlet area and $d_{\text{inlet}} \in \mathbb{R}^3$ the normalized inflow direction. Given the vortex size $\sigma > \delta_x$ and turbulence intensity $I > 0$ as parameters, $n_{\text{vortices}} \in \mathbb{N}$ discrete seed points are placed randomly on the *inflow plane* at positions $p_i \in \mathbb{R}^3$ and associated with signs $s_i \in \{-1, 1\}$. Based on this, per-vortex circulations Γ_i are computed from turbulent kinetic energies k_i:

$$k_i := \frac{3}{2}(|u_{\text{mean}}(p_i)| I)^2,$$

$$\Gamma_i := 4 s_i \sqrt{\frac{\pi}{6\ln 3 - 9\ln 2} \frac{k_i A_{\text{inlet}}}{n_{\text{vortices}}}}.$$

Together, this allows for recovering the pertubed velocity field

$$u_{\text{vortex}}(x) := \frac{1}{2\pi} \sum_{i=1}^{n_{\text{vortices}}} \Gamma_i \frac{((p_i - x) \times d_{\text{inlet}}) \left(1 - \exp\left(\frac{|x-p_i|^2}{2\sigma^2}\right)\right)}{|x - p_i|^2}$$

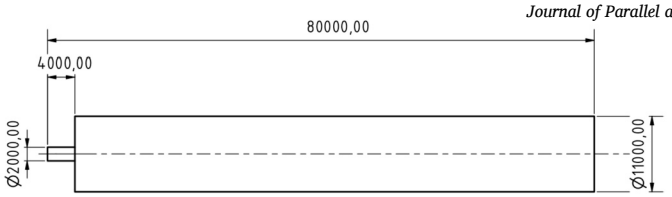


Fig. 2. Geometry of the turbulent free jet reference case [mm].

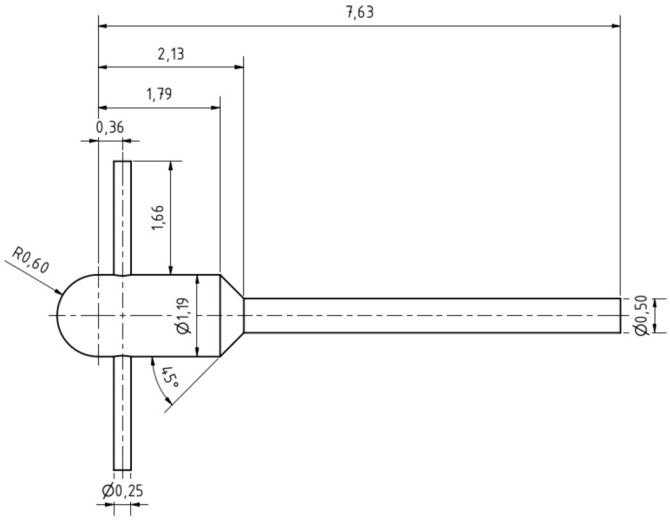


Fig. 3. Geometry of the turbulent mixer reference case [mm] (cf. [53]).

which results in the final inflow velocity field with streamwise fluctuation using the Langevin equation

$$u(x) := u_{\text{mean}}(x) + u_{\text{vortex}}(x) - \frac{u_{\text{vortex}}(x) \times (\nabla u_{\text{prev}})(x)}{|(\nabla u_{\text{prev}})(x)|} d_{\text{inlet}}.$$

The pertubed inflow velocity field is computed for each discrete LB timestep and applied to the lattice using an interpolated velocity boundary condition. Listing 5 details the concrete implementation of the VM boundary condition. Using this approach, the length scale of the perturbations is resolution independent, allowing the variation of turbulence level, length scale and lattice resolution.

The perfectly-parallel parts of the VM boundary condition are implemented as a platform-transparent operator [17,46] in OpenLB 1.6 [25] while the random (re)seeding is implemented for CPU-execution on the host-side using the standard libraries random number generation facilities. From the perspective of computational heterogeneity, the VM as implemented in OpenLB is of interest due to both its computational complexity and its dependence on host-side operations in each timestep. This yields a reduction of the gap between CPU- and GPU-based performance compared to simpler boundary schemes (cf. the simplified model in Section 3.3.1, particularly the timings in Table 2).

3.1.2. Turbulent free jet

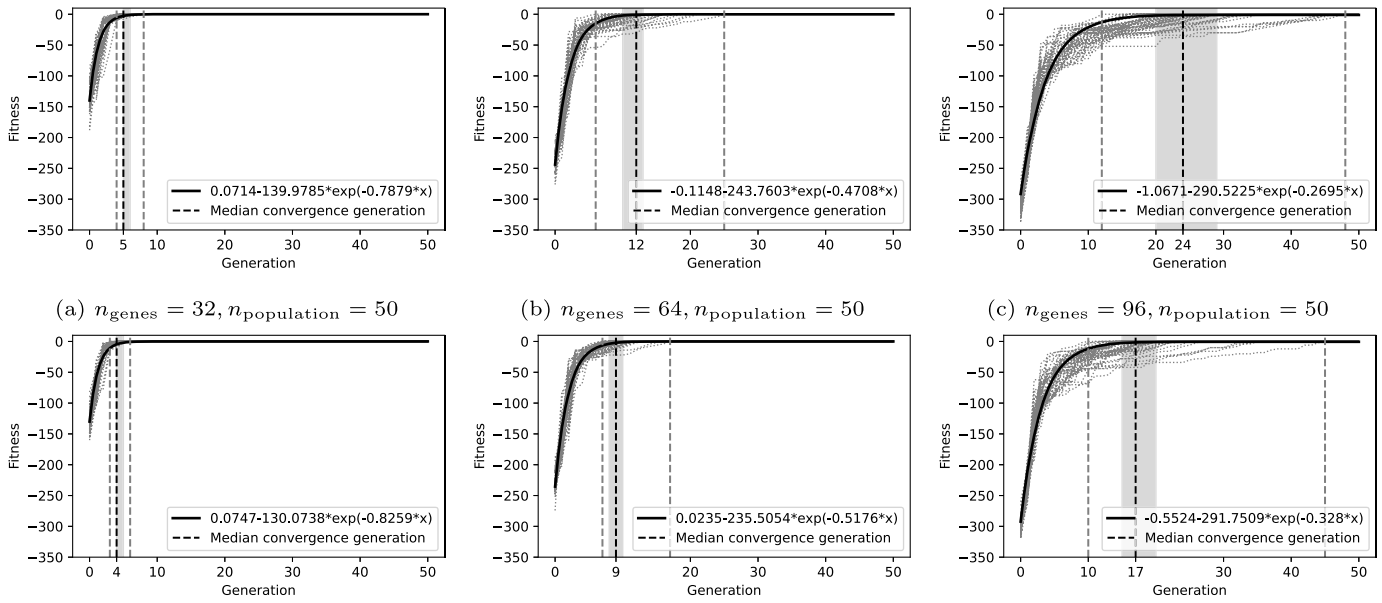
For the first case, an existing free jet simulation case included in OpenLB's example library is adapted to use cost-aware spatial decompositions generated via the presented approach. Fig. 2 details the simulation geometry which is discretized by a single-precision `D3Q19` lattice targeting approximation of the *Navier-Stokes* equation. For the inlet tube on the left hand side, a VM turbulent velocity profile is applied via an interpolated velocity condition while the outlet on the right hand side of the injection tube is modeled using an interpolated pressure condition [50,51]. To complete the boundary configuration, all solid walls are represented by an interpolated no-slip condition [52]. For the bulk, the LES BGK model detailed in Definition 17 is applied. Characteristic length and inflow velocity were chosen w.r.t. a target Reynolds number of $Re = 5000$.

3.1.3. Turbulent mixer

For the second case, the setup of a turbulent reactive mixer simulation of impinging jets following [53] was chosen to represent a more complex setup. Specifically, two lattices targeting *Navier-Stokes* (NSE) respectively a *(Reactive) Advection Diffusion Equation* ((R)ADE) are coupled, including LES turbulence modeling and interpolated boundary conditions in addition to two instances of the VM turbulent inlet condition. All computations were performed using single precision IEEE floating point values.

For the carried fluid modeled on a D3Q19 lattice, two turbulent inlets at Reynolds number $\text{Re} = 3000$ with diameter $d_{\text{inlet}} = 2.5$ mm force two chemical species into the mixing chamber where they are intensively mixed and react with each other. The turbulent carrier fluid is modeled by the Smagorinsky LES model following Definition 17. At each inlet, a Poiseuille velocity profile with VM turbulence initiation is applied. The maximal velocity is set to $u_{\text{inlet}} = \nu \,\text{Re}\, d_{\text{inlet}}^{-1}$ m/s with turbulent intensity at 5% and vortex diameter $d_{\text{vortex}} = 0.01\, d_{\text{inlet}}$. At the outlet an interpolated pressure boundary condition is applied [51] The setup of the NSE lattice is completed by interpolated no slip boundary conditions [52] to model the solid mixer walls.

The present setup does not yet represent the chemical reaction but reproduces only the mixing process as a starting point for a separate publication currently being prepared. In this context, the species are described by a singe *D3Q7* ADE lattice with a Schmidt-number based stabilization algorithm applied at the cells with a high shear rate [54]. Instead of specie concentrations, only one scalar is mapped that is set to $+0.5$ and -0.5 at the opposing



(a) $n_{\text{genes}} = 32, n_{\text{population}} = 50$ (b) $n_{\text{genes}} = 64, n_{\text{population}} = 50$ (c) $n_{\text{genes}} = 96, n_{\text{population}} = 50$

(d) $n_{\text{genes}} = 32, n_{\text{population}} = 100$ (e) $n_{\text{genes}} = 64, n_{\text{population}} = 100$ (f) $n_{\text{genes}} = 96, n_{\text{population}} = 100$

All parameterizations of the GA were applied to a coarse $11 \times 11 \times 11$ base discretization over 50 randomly seeded runs. Vertical dashed lines mark the minimum, median, and maximum generation of convergence over all runs of the particular constellation while the shaded area indicates the interquartile range.

Fig. 4. Convergence of homogeneous cuboidal case for different parameters.

inlets. The scalar value of 0 then represents the fully homogeneous state. Completing the ADE setup are outlets modeled by zero gradient boundary conditions and solid walls modeled by the interpolated no slip boundary condition [52] analogously to the NSE lattice.

3.2. Optimization of spatial decomposition

This section evaluates the generation of spatial decompositions following the specific genetic algorithm documented in Section 2.3. The simulation performance resulting from mapping those decompositions onto heterogeneous target hardware is evaluated in Section 3.3.

3.2.1. Homogeneous cuboidal domain

As a first sanity test of the proposed algorithm we apply it to a cost-homogeneous cuboid

$$D := \{x \in \mathbb{R}^3 | \forall i : x_i \in [-1, 1]\}$$

that is coarsely discretized using $\delta_b = 2/11$ for decomposition. Given the stated optimization goals, the resulting decomposition should be a single cuboid, exactly covering the spatial simulation domain. The fitness of such decompositions with cardinality $|\mathcal{D}_{\text{single}}| = 1$ and volume fraction 1 is ideal at $f(\mathcal{D}_{\text{single}}) = 0$.

As the coarsely discretized spatial simulation domain contains no empty cells for this basic setup, the global minimum volume fraction has no impact on the algorithm and can be ignored. Fixing the fraction of steady state selected parents at 0.1, the set of relevant parameters reduces to the population size and the number of genes. Fig. 4 compares the convergence rate distribution over 50 independent runs of the genetic algorithm for numbers of genes $n_{\text{genes}} \in \{32, 64, 96\}$ and population sizes $n_{\text{population}} \in \{50, 100\}$.

Comparing the results we observe that increasing the number of genes per chromosome decreases the convergence speed while it is increased for larger population sizes. This observation is explained by all genes of the initial population being active initially and only being deactivated during evolution by random mutation. As the ideal solution consists only of a single cuboid seeded by a single gene, larger chromosomes render this solution harder to reach. Similarly, increasing the population size while fixing the gene count improves performance due to more permutations being covered per generation.

3.2.2. Heterogeneous cuboidal domain

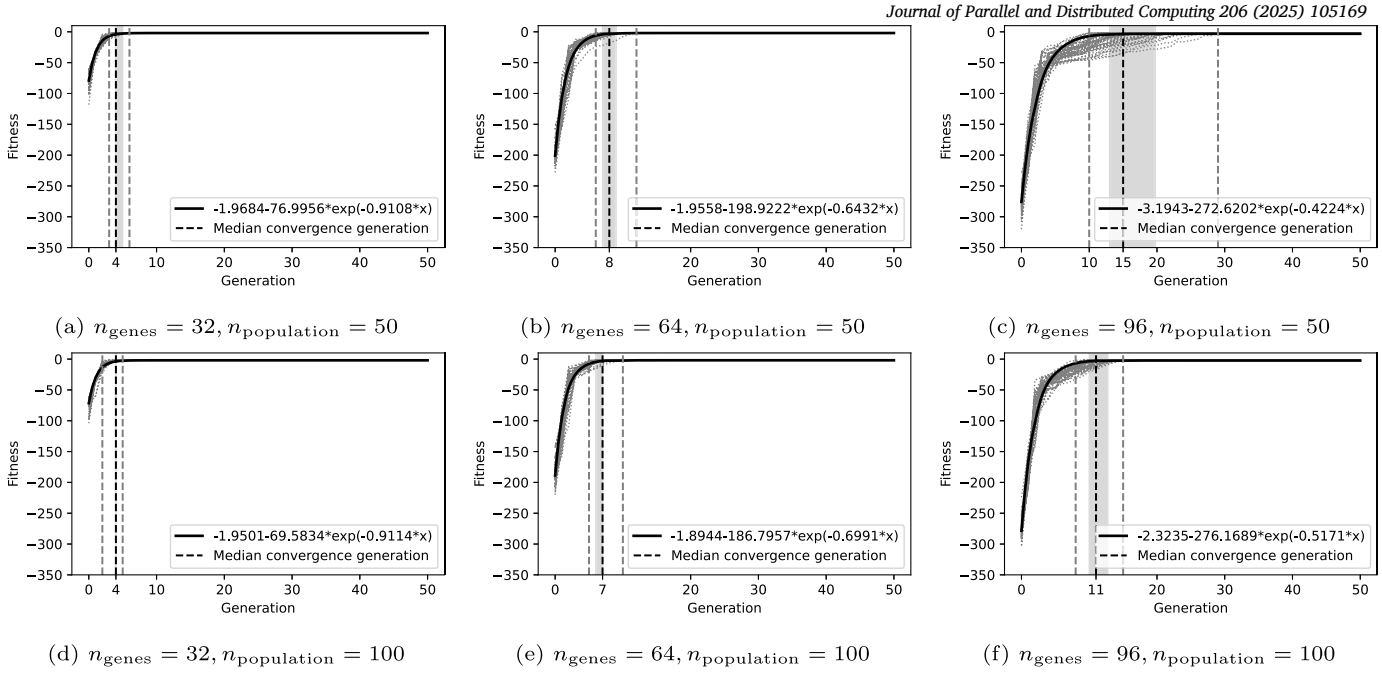
To validate the basic ability of the algorithm to spatially decompose cost-heterogeneous domains into individually homogeneous cuboids, the test case of Section 3.2.1 is separated into multiple cost-heterogeneous slices. Specifically, the cuboidal simulation domain (same as in the homogeneous case)

$$D := \{x \in \mathbb{R}^3 | \forall i : x_i \in [-1, 1]\}$$

is combined with a cost estimate

$$p(x) = \begin{cases} 10 & C_x \subset \{x \in \mathbb{R}^3 | x_0 \leq -1 + \delta_b\} \\ 1 & \text{otherwise} \end{cases},$$

using coarse discretization spacing $\delta_b \in \mathbb{R}^+$ (cf. Definition 4, chosen here again as $\delta_b := 2/11$) to generate a cost-heterogeneous test case.



(a) $n_{\text{genes}} = 32, n_{\text{population}} = 50$ (b) $n_{\text{genes}} = 64, n_{\text{population}} = 50$ (c) $n_{\text{genes}} = 96, n_{\text{population}} = 50$

(d) $n_{\text{genes}} = 32, n_{\text{population}} = 100$ (e) $n_{\text{genes}} = 64, n_{\text{population}} = 100$ (f) $n_{\text{genes}} = 96, n_{\text{population}} = 100$

All parameterizations of the GA were applied to a coarse $11 \times 11 \times 11$ base discretization over 50 randomly seeded runs. Vertical dashed lines mark the minimum, median, and maximum generation of convergence over all runs of the particular constellation while the shaded area indicates the interquartile range.

Fig. 5. Convergence of heterogeneous cuboidal case for different parameters.

Analogously to the homogeneous case, Fig. 5 compares the distribution of convergence rates over 50 independent runs of the genetic algorithm for parameters $n_{\text{genes}} \in \{32, 64, 96\}$ and $n_{\text{population}} \in \{50, 100\}$.

Comparing first the heterogeneous results on their own, we again observe that increasing the number of genes per chromosome decreases the convergence speed while it is increased for larger population sizes. Comparing the heterogeneous case to the homogeneous one, the former consistently converges faster to its ideal solution than the latter one (cf. Fig. 4). An explanation for this is the cost-weighted choice of seed positions at the mutation stage (see Definition 13), reducing the likelihood of seeds being placed in the large low-cost region. This renders the evolution of this region's covering closer to the comparably better converging low-seed-count versions of the homogeneous case.

Finally, in order to experimentally verify the algorithm's isotropy, Fig. 6 slices each axis by a high-cost layer at $x_i \in \{-0.5, 0, 0.5\}$ for $i \in \{0, 1, 2\}$ using fixed parameters $n_{\text{genes}} = 32$ and $n_{\text{population}} = 50$. No significant orientation dependence is exposed over 50 independent runs of each variant and all cases convergence quickly within 3 to 6 generations.

3.2.3. Heterogeneous free jet

The turbulent free jet reference case introduced in Section 3.1.2 is decomposed using the genetic algorithm in preparation for the performance evaluation in Section 3.3.2. For the cost estimate, the inlet tube is covered by a ten-fold cost compared to the bulk simulation domain in the injection tube due to its usage of the VM turbulent inlet condition. Given a minimum volume fraction $\upsilon_{\text{min}} = 0.7$ the expected cost-aware decomposition consists of two cuboids, one covering the injection tube and the other the inlet tube – resulting in the decompositions showcased alongside Tables 4 and 5 after subdivision to cover the provided GPUs.

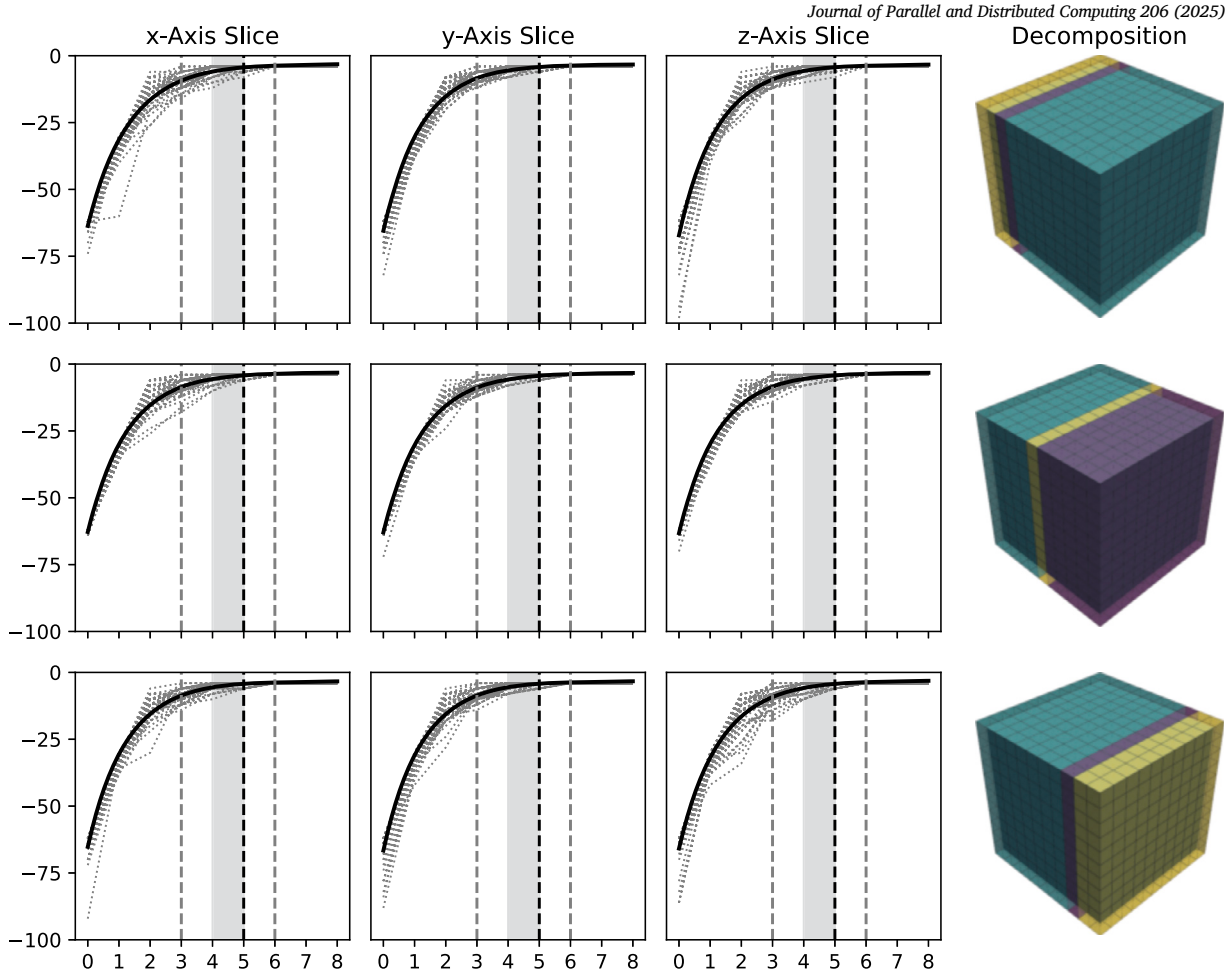
Fig. 7 summarizes the fitness evolution for different combinations of chromosome size and population count of the GA applied to the free jet geometry (cf. Fig. 2). Similarly to the cuboid reference cases, the median convergence generation is reduced by an increase of population size and increases in relation to the chromosome size. However, in all cases the optimization algorithm converged to the expected solution within at most 12 generations. This showcases the usability of the GA for a first simple application case.

3.2.4. Heterogeneous turbulent mixer

The turbulent mixer reference case introduced in Section 3.1.3 is decomposed using the genetic algorithm. Estimation of the inlet regions at a ten-fold cost compared to the bulk simulation domain yields the decompositions showcased in Fig. 9. The performance characteristics resulting from this cost-aware decomposition will be discussed in Section 3.3.3.

For this final evaluation of the GA's performance, Fig. 8 again summarizes the fitness evolution over different parameter combinations and minimum volume fraction $\upsilon_{\text{min}} = 0.7$. The more complex geometry and cost-landscape leads to a increase of the median convergence generation, e.g. at 51 generations for the parameter set with highest chromosome and population size. However, the basic characteristics observed in the previous evaluation cases continue to hold with the best convergence speed being observed for $n_{\text{genes}} = 16$ and $n_{\text{population}} = 200$ – i.e. the highest population of smallest chromosomes.

Fig. 10 provides context for the GA-obtained decomposition by showcasing decompositions obtained via OpenLB's standard *by-volume* strategy [33]. With respect to the goal of single node performance optimization the apparent simplicity of the GA decomposition is the advantage as it provides high-volume-fraction covering of the geometry with a minimal number of cuboids. The by-volume strategy only approaches a similarly close coverage for unnecessarily large cuboid counts.



Three different cost estimates resulting in the optimal decompositions displayed in the rightmost column are applied along the x-, y- resp. z-axes to investigate possible anisotropies of the genetic algorithm. All nine individual cases were decomposed from a coarse $11 \times 11 \times 11$ base discretization using 50 randomly seeded runs the GA. Vertical dashed lines mark the minimum, median, and maximum generation of convergence over all runs of the particular case while shaded regions indicate the interquartile range. Without exception, all runs converged within a median of 5 generations and identical minimum, maximum and quartile generations.

Fig. 6. Isotropy of heterogeneous cuboid decomposition case.

3.3. Performance

This section first evaluates the performance potential of CPU-GPU heterogeneous execution for a simplified model followed by the performance resulting of the heterogeneous balancing approach described in Section 2 applied to two representative case studies. The simplified model in Section 3.3.1 predicts the performance advantage obtainable via heterogeneous computation in a resource-restricted context that is then observed in practice for real-world applications in Sections 3.3.2 and 3.3.3. It also recovers the comparably smaller advantage obtainable via slicing approaches [14,18–21]. Finally, the obtained speedups are investigated in a detailed per-step timing breakdown in Section 3.3.4.

The base arithmetic type of all simulations was set to IEEE single-precision, utilizing AVX-512 vectorization and OpenMP on CPUs and CUDA on GPUs. Further details of the software environment are discussed in Section 2.2.

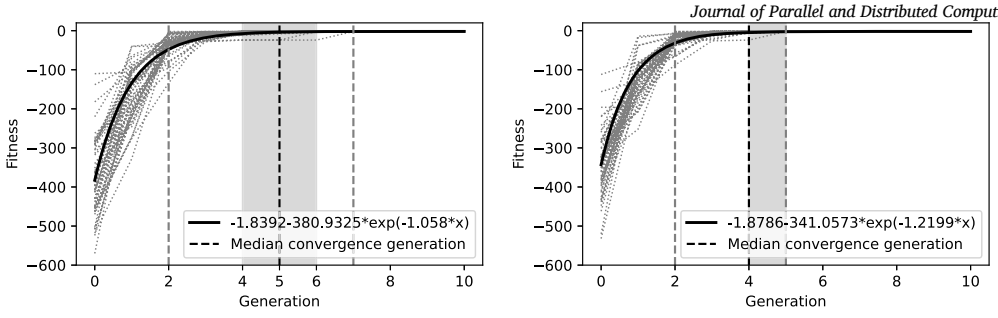
All cases utilize a turbulent velocity inlet condition described in Section 3.1.1 to create the cost-inhomogeneity utilized by heterogeneous processing. The two benchmark systems used for all measurements are detailed in Table 1.

All absolute performance measurements in *billions of cell updates per second* (GLUPs) are obtained as the mean over multiple independent runs of the respective benchmark case on the same system. For all evaluations, no significant fluctuations between measurements of the same benchmark were observed. This also applies to temporal fluctuations within a single benchmark run as the evaluated cases do not include any simulation-time-dependent changes to their performance. We note that performance values in GLUPs directly relate to *time-to-solution* values via Definition 19.

Definition 19 *(Relation of Performance Measurements to Time-to-Solution).* Let $\Sigma_{\text{cells}}(N)$ and $\Sigma_{\text{timesteps}}(N)$ be the total number of cells and the number of discrete timesteps for fixed resolution $N \in \mathbb{N}$. Then the *time-to-solution* $t_{\text{solution}}(N)$ of the entire simulation case can be obtained from time-invariant performance measurements $p(N)$ in GLUPs:

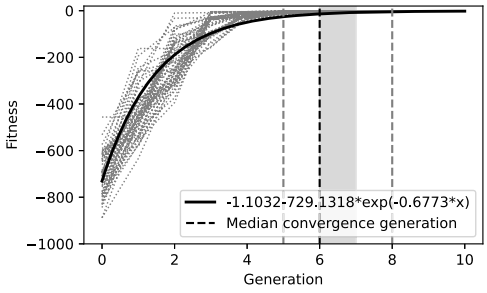
$$t_{\text{solution}}(N) = \frac{\Sigma_{\text{cells}}(N)\Sigma_{\text{timesteps}}(N)}{1e9\, p(N)}.$$

As such, speedups of $p(N)$ directly correspond to reductions of the time-to-solution.

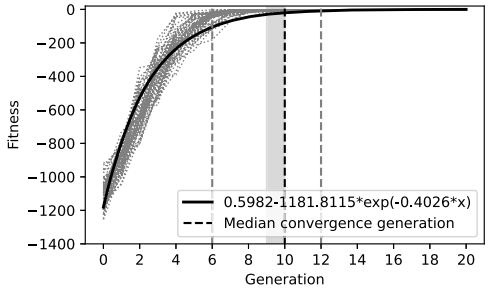


(a) $n_{\text{genes}} = 16, n_{\text{population}} = 100$

(b) $n_{\text{genes}} = 16, n_{\text{population}} = 200$



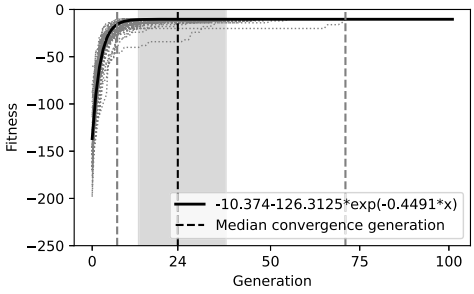
(c) $n_{\text{genes}} = 32, n_{\text{population}} = 200$



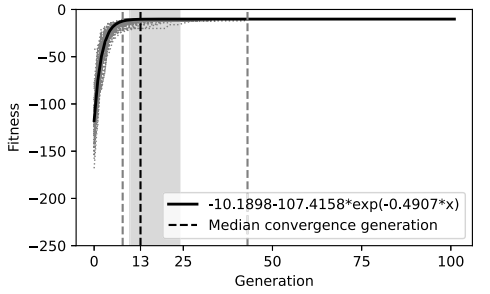
(d) $n_{\text{genes}} = 64, n_{\text{population}} = 200$

All four parameter tuples were applied to a coarse $67 \times 10 \times 10$ base discretization over 50 randomly seeded runs of the GA. Vertical dashed lines mark the minimum, median, and maximum generation of convergence over all runs of the particular constellation while the shaded area indicates the interquartile range. All runs converged within 2 to 12 generations to the expected two-cuboid decomposition.

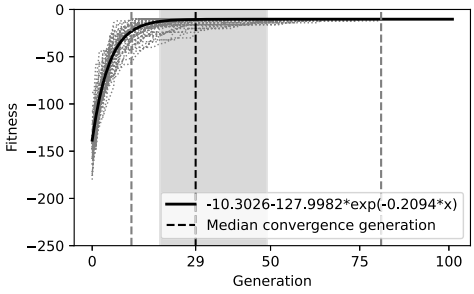
Fig. 7. Convergence of heterogeneous free jet case for different parameters.



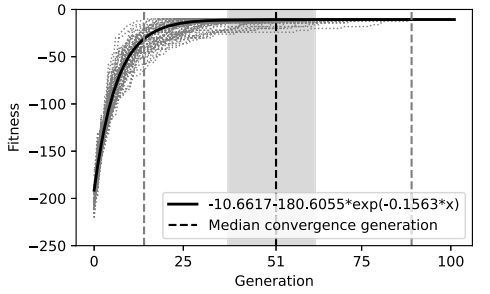
(a) $n_{\text{genes}} = 16, n_{\text{population}} = 100$



(b) $n_{\text{genes}} = 16, n_{\text{population}} = 200$



(c) $n_{\text{genes}} = 32, n_{\text{population}} = 200$



(d) $n_{\text{genes}} = 64, n_{\text{population}} = 200$

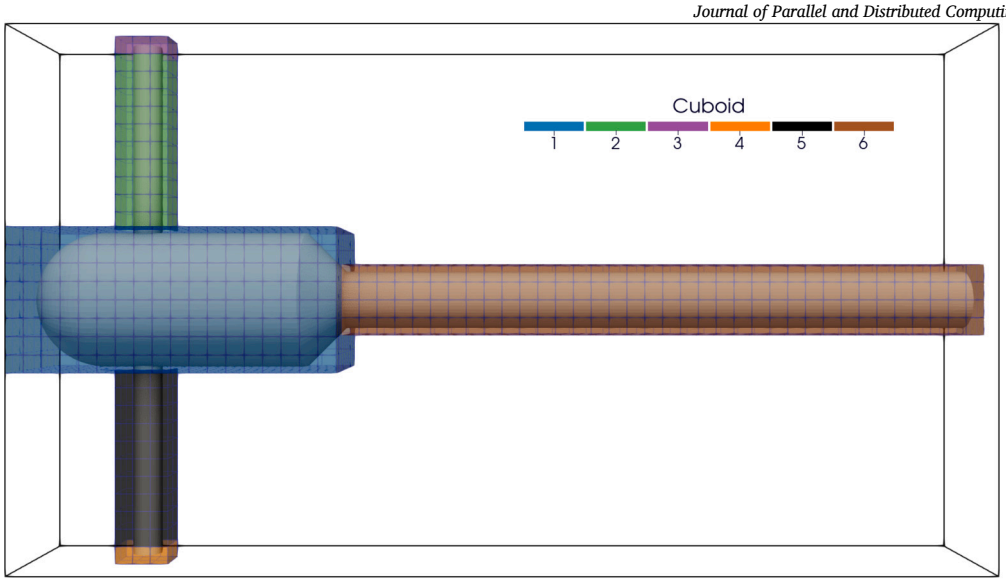
All four parameter tuples were applied to a base coarse $39 \times 22 \times 6$ base discretization over 50 randomly seeded runs of the GA. Vertical dashed lines mark the minimum, median, and maximum generation of convergence over all runs of the particular constellation while the shaded area indicates the interquartile range.

Fig. 8. Convergence of heterogeneous mixer case for different parameters.

3.3.1. Simplified model

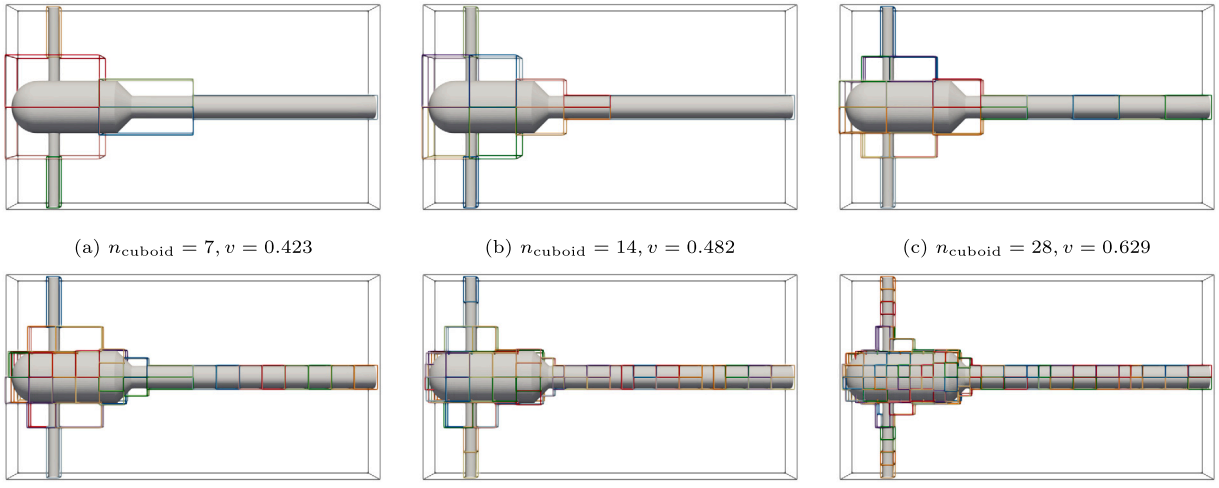
In order to estimate the potential advantage due to heterogeneous computation based on homogeneous performance measurements, we combine reference timings of a minimal square channel flow for different resolutions in a simplified heterogeneity model.

Definition 20 *(Simplified Model).* Let the spatial simulation domain $[0,1]^3 \subset \mathbb{R}^3$ be discretized by a single-precision `D3Q19` lattice with discretization spacing $\delta_x := 1/N$ given by resolution $N \in \mathbb{N}$. For the boundary conditions, cells at $x_0 = 0$ are covered by local or VM velocity boundaries, the ones



The genetic algorithm consistently converges to a *sensible* spatial decomposition of a given mixer geometry (cf. Section 3.3.3) with estimated ten-fold cost at its two inlets (cuboids 3 and 4). Correspondingly, the homogeneous high-cost regions marking the inlets are covered by two small cuboids while the remaining simulation domain is covered by high-volume-fraction cuboids (cuboids 1, 2, 5, 6) given the minimum volume fraction constraint at 0.7. The total volume fraction of the resulting decomposition is $v = 0.726$.

Fig. 9. Evolved spatial decomposition of a mixer geometry with high-cost inlets.



(a) $n_{\text{cuboid}} = 7, v = 0.423$ (b) $n_{\text{cuboid}} = 14, v = 0.482$ (c) $n_{\text{cuboid}} = 28, v = 0.629$

(d) $n_{\text{cuboid}} = 56, v = 0.684$ (e) $n_{\text{cuboid}} = 112, v = 0.741$ (f) $n_{\text{cuboid}} = 224, v = 0.796$

Showcased are decompositions obtained via OpenLB's standard *by-volume* strategy [33] for $n_{\text{cuboid}} \in \{7, \dots, 224\}$ alongside their respective volume fractions v. It can be observed that this strategy only results in high-volume-fraction coverings of the given geometry for high cuboid counts. This property is acceptable for CPU-based MPI-only execution due to the commonly high core count of HPC nodes. Considering the structure of single GPU-accelerated HPC nodes, this illustrates an advantage of the GA-based decomposition approach (cf. Fig. 9, $v = 0.726$): The ability to decompose geometries into a small number of high-volume-fraction cuboids suited for bulk execution on GPUs.

Fig. 10. Decompositions of a mixer geometry using the baseline by-volume approach.

at $x_0 = 1$ using a local pressure boundary and all other faces as bounce back boundaries. The interior cells are modeled using a LES BGK collision (cf. Definition 17). For this simulation case, reference values of the *time-per-simulation-step*

$$t : \mathbb{N} \times \{\texttt{LOCAL}, \texttt{VM}\} \times \{\texttt{GPU}, \texttt{CPU}\} \to \mathbb{R}^+$$

in seconds are obtained for different resolutions, boundary conditions and execution platforms.

Assuming that each processing unit in the CPU-GPU system can process a single lattice at a given time independent of the other processing unit, simultaneous processing of two simulation cases A and B with resolutions $N_A, N_B \in \mathbb{N}$ using only the GPU takes

$$t_{\text{homogeneous}} := t(N_A, \cdot, \texttt{GPU}) + t(N_B, \cdot, \texttt{GPU})$$

while heterogeneous CPU-GPU processing requires

Table 1
System specifications for case studies.

Name Description	LWS Custom workstation	HKN HoreKa `accelerated` node
Sockets	1	2
Processor	Intel Xeon Gold 6326	Intel Xeon Platinum 8368
Cores	16	38
SIMD ISA	AVX-512	AVX-512
Memory[GiB]	64	512
Number of GPUs	2	4
Type	NVIDIA RTX A5000	NVIDIA A100
Memory[GiB]	24	40
Interconnect	NVlink	NVlink
Operating system	NixOS 23.05	RHEL 8
Compiler	GCC 11.3[1]	ICC 2022.0.2[2]
CUDA[3]	11.7	NVIDIA HPC Toolkit 22.7 (11.7)
OpenMPI	4.1.4	4.0.7

[1] Using flags "`-O3 -march=tigerlake -mtune=tigerlake`".
[2] Using flags "`-O3 -ipo -axMIC-AVX512, CORE-AVX2`".
[3] Using flags "`-O3 -rdc=true -extended-lambda -expt-relaxed-constexpr`".

Table 2
Reference timings for the application of the simplified heterogeneity model.

		CPU [µs/Step]		GPU [µs/Step]		Speedup [GPU/CPU]	
N	Σ_{cells} [1e3]	LOCAL	VM	LOCAL	VM	LOCAL	VM
30	27	385.71	415.39	78.49	409.09	4.91	1.02
40	64	864.86	673.69	96.97	457.14	8.92	1.47
50	125	786.16	932.84	131.30	510.20	5.99	1.83
60	216	1317.07	1430.01	174.05	580.65	7.57	2.46
70	343	1854.05	1874.32	230.51	663.44	8.04	2.83
80	512	2547.26	2089.80	302.06	807.57	8.43	2.59
90	729	3328.77	2963.41	391.72	1004.13	8.50	2.95
100	1000	4291.85	3952.57	495.29	1114.83	8.67	3.55
200	8000	-	-	2905.92	-	-	-
300	27000	-	-	8520.04	-	-	-

Averaged per-step timings for a square channel flow utilizing either computationally cheap local boundaries or the more expensive vortex method (VM) approach. These values are used to estimate the speedup potential for combined CPU-GPU usage in Table 3. Notably the gab between CPU- and GPU-based execution of the VM case is much smaller than for the LOCAL case (cf. Speedup columns).

$$t_{\text{heterogeneous}} := \max\{t(N_A, \cdot, \text{CPU}), t(N_B, \cdot, \text{GPU})\}.$$

Thus we can compute a theoretical heterogeneity speedup

$$s := \frac{t_{\text{homogeneous}}}{t_{\text{heterogeneous}}}$$

for combinations of resolution, boundary condition and processing platform.

Table 3 investigates two orthogonal combination axes of the simplified model in Definition 20: The combination of sizes N_A and N_B and the cost of A given due to the boundary condition $\text{A}_{\text{bc}} \in \{\text{LOCAL}, \text{VM}\}$. Performance advantages can be obtained both using size-heterogeneity s.t. $N_A \ll N_B$ (this is what existing slicing approaches [14,18–21] utilize) or cost-heterogeneity due to $\text{A}_{\text{bc}} = \text{VM}$ compared to $\text{B}_{\text{bc}} = \text{LOCAL}$. It can be observed that, given a sufficiently large block size difference, the increased concurrency of single node CPU-GPU heterogeneity offers a theoretical speedup compared to GPU-only utilization (e.g. up to 1.06 for $N_A = 100$ and $N_B = 300$ in the cost-homogeneous case). However, the theoretical advantage is significantly higher (up to 1.83) for the cost-heterogeneous cases where a smaller lattice A is processed on CPU alongside a larger but *locally cheaper* lattice B. This is also visible in Table 2 where the per-case GPU advantage is significantly lower for VM boundaries than for the LOCAL ones (e.g. the $N = 30$ case is roughly five times faster on GPU than on CPU for LOCAL but nearly equally fast when using VM boundaries).

3.3.2. Turbulent free jet

For the first case, the turbulent free jet described in Section 3.1.2 was selected with the goal of demonstrating an advantage of CPU-GPU processing heterogeneity in a bottom-up heterogeneous decomposition not generated as a modified homogeneous decomposition.

On both systems, each GPU was assigned one section of the bulk injection tube while the CPU(s) were assigned the penalized inlet tube using the load balancing approach detailed in Section 2.4. The specific decomposition is showcased alongside the performance measurements in Tables 4 and 5.

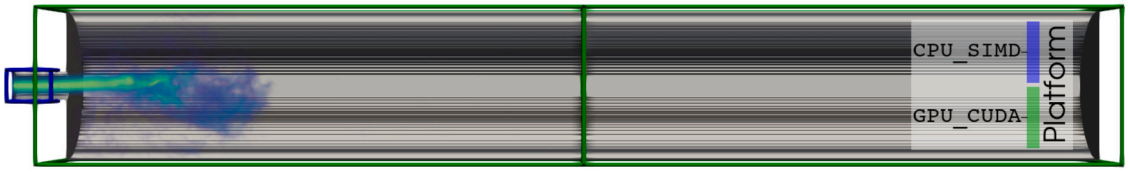
Considering first the performance on LWS, a local dual-GPU workstation, Table 4 summarizes the resulting performance for problem sizes between $1.3e6$ and $231e6$ cells using the homogeneous, plain heterogeneous and orthogonal heterogeneous balancing strategies detailed in Section 2.4. The minimum targeted GPU volume fraction was set to 0.99 and a fraction of 0.998 was reached. In all samples the heterogeneous balancing is observed to provide superior performance compared to the homogeneous GPU-only balancing with speedups between 1.04 and 1.34. Focusing on heterogeneous

Table 3
Theoretical speedup for simplified heterogeneity model.

	Speedup [(CPU for A, GPU for B) / GPU for both]							
	Homogeneous cost				Heterogeneous cost			
N_A \ N_B	N_A	100	200	300	N_A	100	200	300
30	0.41	1.16	1.03	1.01	1.17	1.83	1.14	1.05
40	0.22	0.68	1.03	1.01	0.82	1.41	1.16	1.05
50	0.33	0.80	1.05	1.02	0.69	1.08	1.18	1.06
60	0.26	0.51	1.06	1.02	0.52	0.75	1.20	1.07
70	0.25	0.39	1.08	1.03	0.48	0.62	1.23	1.08
80	0.24	0.31	1.10	1.04	0.53	0.62	1.28	1.09
90	0.24	0.27	0.99	1.05	0.47	0.51	1.32	1.12
100	0.23	0.23	0.79	1.06	0.41	0.41	1.02	1.13

Speedup comparison in simplified model of two lattices A and B with different sizes and processing platforms on a test system consisting of a single SIMD CPU and a single NVIDIA GPU (cf. System LWS in Table 1, benchmark case implemented in OpenLB, raw timings provided in Table 2). All values compare the speedup estimate given by Definition 20 for varying combinations of resolutions and boundary conditions. For the homogeneous cost cases both lattices use local boundaries, causing their computational cost to be dominated by the bulk model. In contrast, the heterogeneous cost cases use the VM boundary for lattice A and local boundaries for lattice B. Combinations with heterogeneity advantage are shaded in gray.

Table 4
Performance of Turbulent Free Jet case on System LWS.



		Performance [1e9 Cell Updates / s]			Speedup		
		GPU-only	CPU-GPU				
N	Σ_{cells}[1e6]	Homogeneous	Heterogeneous	Orthogonal	Het/Hom	Ort/Het	Ort/Hom
5	1.3	1.418	1.907	2.515	1.344	1.318	1.773
7	3.5	2.443	2.746	3.314	1.124	1.206	1.356
9	7.3	3.241	3.603	4.020	1.111	1.115	1.240
11	13	3.874	4.061	4.525	1.048	1.114	1.167
13	21	4.265	4.528	4.687	1.061	1.034	1.098
15	33	4.542	4.732	4.917	1.041	1.039	1.082
17	47	4.751	5.007	5.173	1.054	1.033	1.088
19	66	4.969	5.170	5.347	1.040	1.034	1.076
21	89	5.133	5.350	5.505	1.042	1.029	1.072
23	116	5.256	5.469	5.602	1.040	1.024	1.065
25	149	5.373	5.589	5.701	1.040	1.020	1.061
27	187	5.458	5.703	5.787	1.044	1.014	1.060
29	231	5.549	5.826	5.920	1.049	1.016	1.066

Comparing the performance of GPU-only balancing of the cost-aware spatial decomposition obtained via the GA to different CPU-GPU assignment strategies for the turbulent free jet on LWS. Observed speedups due to processing heterogeneity range between 1.04 and 1.77. Decomposition was fixed at $n_{\text{cuboid}} = 3$ s.t. each of the two NVIDIA RTX A5000 GPUs receives an equal workload share. Figure displays domain decomposition and platform assignment with illustrative velocity magnitude volume rendering overlay.

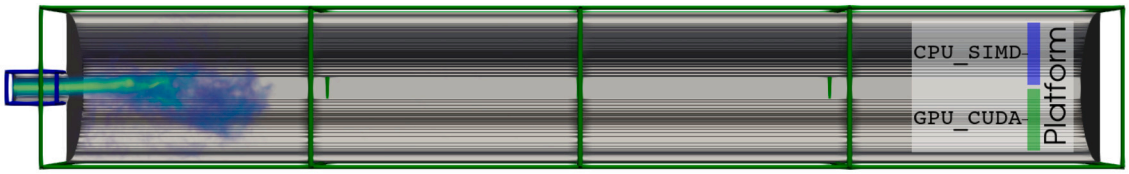
strategies, the orthogonal approach yields an additional speedup between 1.014 and 1.31 adding up to total speedups w.r.t. the homogeneous approach between 1.06 and 1.77.

A downwards trend for growing resolutions can be observed for all samples. The reason for this is the decreasing competitiveness of CPU-based processing of the high-cost inflow region for growing resolutions. This observation matches up with the prediction of the simplified performance model in Section 3.3.1 when fixing the ratio between N_A and N_B.

This downwards trend also highlights a limitation of the approach for this specific application du to a difference in workload scaling: the GPU-assigned bulk simulation is volumetric with cost $O(N^3)$, while the CPU-assigned inlet condition is areal with cost $O(N^2)$. As N grows, the total execution time becomes overwhelmingly dominated by the GPU's cubic workload, thus reducing the relative speedup from heterogeneous co-processing. The advantage in this case is therefore most pronounced in a regime where the areal-scaling workload still constitutes a non-trivial fraction of the total computational cost.

Considering this relationship in a general LBM context and the existing literature, heterogeneous computation can in any case only be expected to yield an incremental performance improvement given a fixed set of computational resources and its usefulness decreases given the ability to *freely scale* the number of compute nodes without cost considerations. This puts the decreasing (but still notable) performance advantage for growing problem sizes into context as these simulations are approaching the limits of the fixed hardware.

Table 5
Performance of Turbulent Nozzle case on System HKN.



		Performance [1e9 Cell Updates / s]			Speedup		
		GPU-only	CPU-GPU				
N	$\Sigma_{cells}[1e6]$	Homogeneous	Heterogeneous	Orthogonal	Het/Hom	Ort/Het	Ort/Hom
5	1.3	1.491	2.121	2.592	1.423	1.222	1.738
7	3.5	2.827	3.733	4.925	1.320	1.319	1.741
9	7.3	4.338	5.640	6.998	1.300	1.240	1.613
11	13	5.714	7.385	8.542	1.292	1.156	1.495
13	21	6.919	8.804	9.953	1.272	1.130	1.438
15	33	7.953	10.019	11.086	1.259	1.106	1.393
17	47	8.884	11.251	12.211	1.266	1.085	1.374
19	66	9.714	12.338	13.134	1.270	1.064	1.352
21	89	10.270	13.053	13.737	1.270	1.052	1.337
23	116	10.984	13.851	14.548	1.260	1.050	1.324
25	149	11.486	14.557	15.057	1.267	1.034	1.310
27	187	11.954	15.190	15.718	1.270	1.034	1.314
29	231	12.355	15.494	16.138	1.254	1.041	1.306
31	280	12.551	15.996	16.583	1.274	1.036	1.321
33	337	12.960	16.445	17.025	1.268	1.035	1.313
35	402	13.310	17.012	17.448	1.278	1.025	1.310
37	474	13.460	17.200	17.712	1.277	1.029	1.315
39	555	13.750	17.613	18.061	1.280	1.025	1.313

Comparing the performance of GPU-only balancing of the cost-aware spatial decomposition obtained via the GA to different CPU-GPU assignment strategies for the turbulent free jet on HKN (single accelerated node of HoreKa supercomputer). Observed speedups due to processing heterogeneity range between 1.02 and 1.74. Decomposition was fixed at $n_{cuboid} = 5$ s.t. each of the 4 NVIDIA A100 GPUs receives an equal workload share. Figure displays domain decomposition and platform assignment with illustrative velocity magnitude volume rendering overlay.

Table 6
Performance of NSE-only Turbulent Mixer case on System LWS.

		Performance [1e9 Cell Updates / s]						Speedup		
		Homogeneous			Orthogonal			Orthogonal cost-aware /		
N	$\Sigma_{cells}[1e6]$	*by-volume*	*equal-cost*	*cost-aware*	*by-volume*	*equal-cost*	*cost-aware*	*homogeneous by-volume*	*orthogonal equal-cost*	*homogeneous cost-aware*
20	3	1.188	1.284	1.309	1.471	1.813	2.146	1.806	1.183	1.639
40	23	3.258	3.363	3.587	2.319	4.024	4.266	1.309	1.060	1.189
60	77	4.098	4.459	4.884	2.721	4.627	5.284	1.289	1.142	1.081
80	182	4.739	5.195	5.217	2.995	5.128	5.521	1.165	1.077	1.058

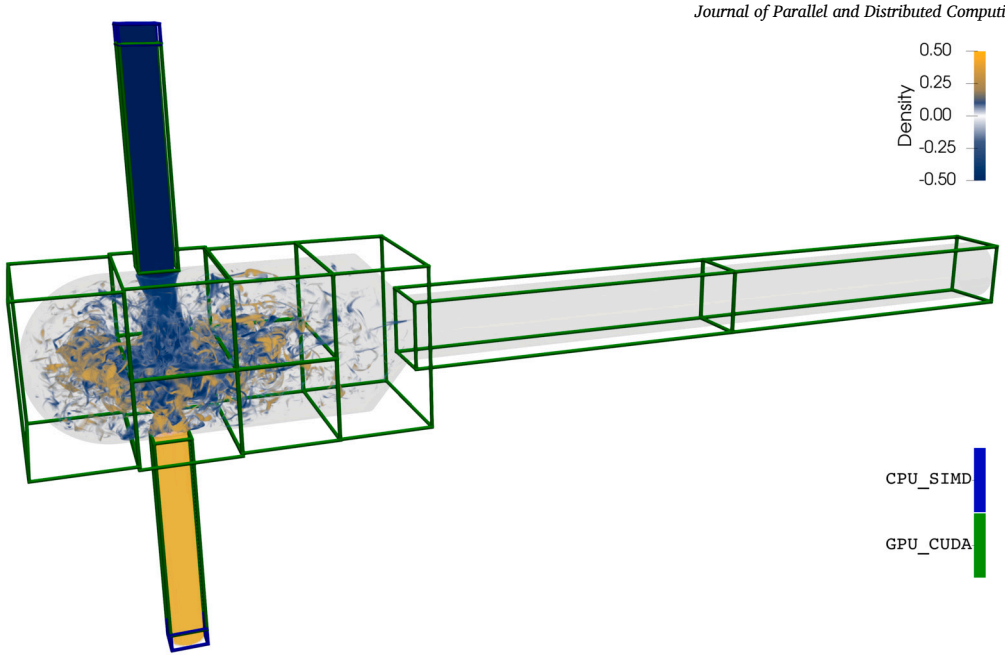
Comparing the performance of GPU-homogeneous heuristic balancing of by-volume, equal-cost and cost-aware spatial decomposition to orthogonal CPU-GPU balancing for the turbulent mixer using only a single NSE-targeting lattice on LWS. Equal-cost and cost-aware decompositions were obtained using the GA while by-volume used OpenLB's default decomposition strategy [33]. All decompositions were fixed at $n_{cuboid} = 12$ to reach equal workload distribution between the two GPUs. Speedup values relate the performance of orthogonally assigned cost-aware decomposition to the homogeneous GPU-only performance prior to the present work, to the performance obtained when executing the GA-optimized equal-cost decomposition orthogonally and finally to GPU-only execution of the cost-aware decomposition.

Qualitatively similar results are observed for the same case on System HKN, the HoreKa `accelerated` node. The larger minimal speedup at 1.306 compared to the LWS results are explained by the smaller performance gap between the system's CPU and GPU.

3.3.3. Turbulent mixer

For the second case, the turbulent mixer described in Section 3.1.3 was selected from current application research [54] in order to improve resource-constrained performance for parameter studies. The base decomposition showcased in Fig. 9 was straight forwardly obtained via the GA given an estimation of the two inlet regions at a ten-fold cost compared to the bulk domain together with a minimum volume fraction of 0.7 (cf. Section 3.2.4). The minimum targeted GPU volume fraction was set to 0.99 and a fraction of 0.998 was reached. The subdivided decomposition used on the concrete systems together with its platform-assignment is displayed in Fig. 11. Absolute performance results and speedups for systems LWS and HKN are listed in Tables 6, 7 and 8. Further details on the resource assignment during scheduling are provided in Section 2.4.1.

Relating the results first to the previous free jet case, a similar downwards trend for growing resolutions can be observed for all speedup samples. Considering the NSE-only results on LWS in Table 6, the lower maximum speedup (~ 1.3 for the full mixer compared to ~ 1.7 for the free jet) can



Illustrative volume rendering of the turbulent mixer's density field superimposed by a cost-aware cuboid decomposition obtained via the GA. Cuboid edges are colored to mark the assigned processing platform. Simulation geometry is detailed in Fig. 3.

Fig. 11. Showcase of turbulent mixer case for $N = 100$ utilizing heterogeneous processing on System HKN.

Table 7
Performance of NSE-only Turbulent Mixer case on System HKN.

		Performance [1e9 Cell Updates / s]						Speedup		
		Homogeneous			Orthogonal			Orthogonal cost-aware /		
N	$\Sigma_{\text{cells}}[1e6]$	*by-volume*	*equal-cost*	*cost-aware*	*by-volume*	*equal-cost*	*cost-aware*	*homogeneous by-volume*	*orthogonal equal-cost*	*homogeneous cost-aware*
20	3	1.642	1.943	1.961	1.373	2.993	3.075	1.873	1.027	1.568
40	23	6.968	8.097	8.362	8.998	9.961	10.900	1.564	1.094	1.304
60	77	10.423	12.318	12.438	10.812	13.485	14.479	1.389	1.074	1.164
80	182	13.924	16.044	16.121	12.585	15.699	16.777	1.205	1.069	1.041
100	355	15.493	16.136	17.010	13.242	17.614	18.030	1.164	1.024	1.060
120	614	16.380	17.740	18.647	13.985	18.761	19.256	1.176	1.026	1.033

Comparing the performance of GPU-homogeneous heuristic balancing of by-volume, equal-cost and cost-aware spatial decomposition to orthogonal CPU-GPU balancing for the turbulent mixer using only a single NSE-targeting lattice on HKN. Equal-cost and cost-aware decompositions were obtained using the GA while by-volume used OpenLB's default decomposition strategy [33]. All decompositions were fixed at $n_{\text{cuboid}} = 12$ to reach equal workload distribution between the four NVIDIA A100 GPUs. Speedup values relate the performance of orthogonally assigned cost-aware decomposition to the homogeneous GPU-only performance prior to the present work, to the performance obtained when executing the GA-optimized equal-cost decomposition orthogonally and finally to GPU-only execution of the cost-aware decomposition.

Table 8
Performance of the full Turbulent Mixer case.

	Performance [1e9 Cell Updates / s]		Speedup
N	Homogeneous Cost-aware	Orthogonal Cost-aware	
20	0.496	0.606	1.222
40	0.856	0.901	1.053
60	0.940	0.990	1.053
80	0.966	1.004	1.039

(a) Performance on System LWS

	Performance [1e9 Cell Updates / s]		Speedup
N	Homogeneous Cost-aware	Orthogonal Cost-aware	
20	1.018	1.344	1.320
40	3.008	3.368	1.120
60	3.994	4.223	1.057
80	4.427	4.566	1.031
100	4.689	4.790	1.022
120	4.830	4.938	1.022

(b) Performance on System HKN

Comparing the performance of GPU-homogeneous heuristic balancing of cost-aware spatial decomposition to orthogonal CPU-GPU balancing for the turbulent mixer case. All decompositions were fixed at $n_{\text{cuboid}} = 12$ to reach equal workload distribution between the GPUs. The CPUs were assigned the two high-cost inlet regions by orthogonal balancing.

Table 9
Average Per-step Timing of NSE-only Turbulent Mixer Case on LWS using Cost-aware Decomposition.

	Homogeneous [µs]			Orthogonal [µs]				Speedup
	Per-Rank		Global	Per-Rank			Global	
Step	0 (GPU)	1 (GPU)		0 (GPU)	1 (GPU)	2 (CPU)		
`prepareVM`	79.49	76.25		0	0	73.16		
`deviceSync`	4.12	3.40		0	0	0		
`applyVM`	572.87	572.64		0	0	80.02		
`collide`	28.99	31.67		23.42	24.89	56.72		
`deviceSync`	374.18	366.24		395.91	388.31	0		
`communicate`	201.36	208.82		199.74	204.91	388.57		
	1261.00	1259.03	1261.00	619.07	618.10	598.46	619.07	2.04
`stream`	20.05	20.01		19.69	19.67	0.60		
`deviceSync`	2.98	2.91		2.89	2.82	0		
`communicate`	0.30	0.30		0.23	0.22	0.09		
	23.33	23.22	23.33	22.81	22.71	0.69	22.81	1.02
`postProcess`	420.18	440.09		334.89	354.68	31.80		
`communicate`	19.99	20.24		20.89	18.54	372.52		
	440.17	460.33	460.33	355.78	373.21	404.32	404.32	1.14
µs/Step			1744.66				1046.20	1.67

(a) Average Per-step Timings for $N = 20$

	Homogeneous [µs]			Orthogonal [µs]				Speedup
	Per-Rank		Global	Per-Rank			Global	
Step	0 (GPU)	1 (GPU)		0 (GPU)	1 (GPU)	2 (CPU)		
`prepareVM`	66.88	66.14		0	0	90.95		
`deviceSync`	8.18	8.67		0	0	0		
`applyVM`	323.14	323.74		0	0	281.63		
`collide`	33.56	30.48		29.16	27.06	160.22		
`deviceSync`	2946.43	2915.03		2836.15	2605.94	0		
`communicate`	388.55	528.20		311.92	531.22	2584.25		
	3691.67	3797.45	3797.45	3177.23	3164.23	3117.05	3177.23	1.20
`stream`	20.48	20.06		20.12	19.50	0.62		
`deviceSync`	3.01	3.09		3.20	2.93	0		
`communicate`	0.29	0.30		0.29	0.26	0.15		
	23.78	23.45	23.78	23.60	22.68	0.77	23.60	1.01
`postProcess`	602.52	605.06		547.79	540.21	68.04		
`communicate`	94.28	94.65		70.56	79.96	545.25		
	696.81	699.71	699.71	618.35	620.17	613.29	620.17	1.13
µs/Step			4520.93				3821.00	1.18

(b) Average Per-step Timings for $N = 40$

Breakdown of average per-step timings for core operations of the NSE-only turbulent mixer case on LWS for resolutions $N \in \{20, 40\}$. Both the GPU-homogenous and the orthogonal CPU-GPU heterogeneous balancing used the GA-generated cost-aware decomposition into $n_{\text{cuboid}} = 12$ cuboids. Individual timings were obtained using OpenLB's internal performance instrumentation. Compare Table 6 for raw averaged performance values. Each timestep is decomposed into three distinct sections by the MPI-based synchronization between neighboring blocks (after collision and streaming respectively) and the global lattice statistics reduction after the application of *post-stream* boundary conditions. Global timings are the maximum of the per-rank and section summation of timings.

be explained by the comparatively increased complexity of the simulation setup, consisting of two separate lattices with distinct collide-and-stream cycles and boundary conditions decreasing the advantage generated by the additional concurrency due to CPU utilization. This also relates to the lower absolute throughput measured in billions of cell updates per second due to each lattice location being modeled by two separate populations and requiring an additional inter-lattice coupling operation after the separate collide-and-stream cycles. For this reason, the actual bandwidth required for each timestep is increased approximately four-fold compared to the simpler single-lattice case.

On both systems and all resolutions, ranging between 3 and 600 million cells, heterogeneous processing of the turbulent mixer case provided a clear performance advantage over GPU-only execution up to a speedup of 1.32 for the coupled NSE-ADE and 1.63 for the simpler NSE-only variant. Totaling the speedup obtained for this case by the present work over both the improved decomposition and the improved resource utilization we obtain speedups between 1.16 and 1.87 compared to GPU-only execution of by-volume decomposition. The highest achieved raw throughput of 19.26 GLUPs for the $N = 120$ reaches around 90% of the performance for a computationally much simpler *lid driven cavity* benchmark case on the same HoreKa GPU node [28].

Combining the by-volume, equal-cost and cost-aware domain decompositions with either GPU-homogeneous or CPU-GPU heterogeneous (orthogonal) rank assignment in Table 6, demonstrates that the performance advantage is produced by the combination of cost-aware decomposition with processing heterogeneity and not only due to better domain decomposition. This also holds for the fully coupled mixer in Table 8a. While one can observe a speedup due to the GA-optimized decomposition alone (cf. the GPU-only equal-cost and cost-aware decompositions compared to by-volume in Tables 6 and 7), this is improved further when combined with heterogeneous rank assignment.

Table 10
Average Per-step Timing of NSE-only Turbulent Mixer Case on LWS using By-volume Decomposition.

	Homogeneous [µs]				Orthogonal [µs]	Speedup	
Step	By-volume Per-Rank 0 (GPU)	1 (GPU)	Global	Cost-aware Global (cf. Table 9a)	Cost-aware Global (cf. Table 9a)	*homogeneous cost-aware / homogeneous by-volume*	*orthogonal cost-aware /*
`prepareVM`	81.35	80.40					
`deviceSync`	4.52	3.66					
`applyVM`	575.06	577.88					
`collide`	30.66	35.88					
`deviceSync`	427.80	401.59					
`communicate`	268.52	305.28					
	1387.91	1404.69	1404.69	1261.0	619.07	1.11	2.27
`stream`	20.59	20.56					
`deviceSync`	2.53	2.57					
`communicate`	0.33	0.36					
	23.45	23.48	23.48	23.33	22.81	1.01	1.03
`postProcess`	402.10	418.06					
`communicate`	119.23	103.60					
	521.33	521.66	521.66	460.33	404.32	1.13	1.29
µs/Step			1949.83	1744.66	1046.20	1.12	1.87

Breakdown of average per-step timings for core operations of the NSE-only turbulent mixer case on LWS for resolution $N = 20$ and using OpenLB's default by-volume decomposition strategy [33]. These values represent the baseline GPU-homogeneous performance prior to the present work and are related to the GA-obtained cost-aware decomposition executed either GPU-homogeneously or CPU-GPU heterogeneously (cf. Table 9a).

3.3.4. Turbulent mixer timing profile

In order to explore the origin of the speedup results in Section 3.3.3, a detailed per-step timing analysis of the NSE-only case is performed on LWS. The averaged timings of the individual algorithm steps are listed in Table 9 for two resolutions $N \in \{20, 40\}$. These values were obtained using OpenLB's internal performance instrumentation and validated to match the total cell throughput values by a constant factor to ensure that no critical component of the LB algorithm was missed. A key observation in this data is that the speedups due to heterogeneous processing are produced not by reductions of any per-process load imbalance but by performance improvements in the critical collision block. This first block of the per-step algorithm includes two operations specific to the VM boundary condition (cf. Section 3.1.1): The CPU-only (re)seeding in `prepareVM` and the platform-transparent operator application in `applyVM`. While there are speedups for the post-stream boundary conditions despite no significant cost-heterogeneity – as predicted by the simplified model – the majority of the total speedup originates in the increased concurrency of the expensive VM application during the collision block. Specifically, this part is sped up approximately two-fold for the $N = 20$ case and 1.2-fold for the $N = 40$ case, reasonably closely matching the global speedups observed in the previous GLUPs-based throughput measurement.

Finally, Table 10 relates the GPU-only baseline timings for $N = 20$ using by-volume decomposition to the cost-aware decomposition executed both homogeneously and heterogeneously. This confirms that the total speedup results of two parts: A spatial decomposition optimized for improved simulation geometry matching (~ 1.1 speedup of the critical collision section) and increased concurrency due to CPU utilization in regions with reduced GPU performance advantage (additional ~ 1.67 speedup yielding a total of ~ 1.87).

4. Conclusion

This work addressed the efficient utilization of heterogeneous CPU-GPU systems for LBM-based simulations using a novel cost-aware domain decomposition and load balancing approach based on genetic programming. The algorithm optimizes a *seeded spatial decomposition* by creating cost-homogeneous cuboids, which are then mapped to the hardware using either a heterogeneous or an orthogonal rank assignment strategy. The full approach was implemented in OpenLB and evaluated on two test systems featuring multiple NVIDIA GPUs and Intel CPUs with AVX-512 vectorization. Application-dependent speedups up to 87% were observed for heterogeneous CPU-GPU execution over the original cost-unaware decomposition. This gain is attributable to both the optimized decomposition and the cooperative use of hardware. Specifically, activating CPU-GPU execution delivered a speedup of up to 77% over a GPU-only run using the optimized decomposition. Within this heterogeneous framework, the orthogonal assignment strategy was another important factor, improving performance by up to 31% over the heterogeneous assignment model.

This novel bottom-up approach marks an advance in methodology and performance compared to existing works focused on top-down subdivision of homogeneous decompositions. The documented improvements in a resource-constrained single-node setting are of particular interest where scaling is not financially feasible. However, the method's applicability is predicated on block-structured geometries and the availability of a meaningful cost-estimate, with performance advantages being most pronounced for problems where spatially-distinct, lower-complexity cost-heterogeneities have a significant impact on the total runtime.

Future research directions include multi-node balancing and incorporating auto-tuning mechanisms to further automate load balancing. The genetic algorithm could also be applied to cost-homogeneous cases and its fitness function refined to include different properties of the evaluated decomposition.

5. Listings

Listing 2: Construction of Seeded Spatial Decomposition

```
function growCuboid(grid     : Current grid 3D array of integers
                    cost     : Cost estimate as 3D array of integers
                    lower    : Lower bound position of cuboid
                    upper    : Upper bound position of cuboid
                    fraction : Minimum volume fraction of cuboid
                    material : Integer identifier of cuboid cells)
begin
  best := grid[lower:upper]
  best.nonEmptySize  := count(best == material)
  best.fraction      := best.nonEmptySize / size(best)
  best.heterogeneity := computeInternalHeterogeneity(cost[lower:upper])
  for (lG, uG) in {c ∈ {0,−1}^3 | ∃!i ∈ {0,1,2} : c_i = −1} × {c ∈ {0, 1}^3 | ∃!i ∈ {0,1,2} : c_i = 1}
    newLower := max(0,           lower + lG)
    newUpper := min(extent(grid), upper + uG)
    sub := grid[newLower:newUpper]
    sub.nonEmptySize := count(sub == material | sub == 1)
    sub.homogeneousSize := sub.nonEmptySize + count(sub == 0 | sub == -material)
    isConsistent := sub.homogeneousSize == size(sub)
    sub.fraction := sub.nonEmptySize / size(sub)
    sub.heterogeneity := computeHeterogeneity(cost[newLower:newUpper])
    isCoveringMoreOrSameByLess := sub.nonEmptySize >= best.nonEmptySize and size(sub) <  size(best)
    isCoveringMoreByMoreOrSame := sub.nonEmptySize >  best.nonEmptySize and size(sub) >= size(best)
    isReducingHeterogeneity := sub.heterogeneity < best.heterogeneity
    if isConsistent and sub.fraction >= fraction and sub.heterogeneity <= best.heterogeneity
      if isCoveringMoreOrSameWithLess or isCoveringMoreWithMoreOrSame or isReducingHeterogeneity
        best := sub
      end
    end
  end
end

function growGridFromSeeds(grid  : Initial grid as 3D array of integers
                           cost  : Cost estimate as 3D array of integers
                           seeds : List of seeds with properties active, origin, fraction)
begin
  cuboids := Map() // Map between seeds and associated cuboids
  material := 2
  for seed in seeds
    cuboids[seed.origin] := (seed.active, seed.origin, seed.origin, seed.fraction, material)
    material += 1
  end
  growing := True
  while growing
    growing := False
    for origin, (active, lower, upper, fraction, m) in cuboids if active
      grew, newLower, newUpper := growCuboid(grid, cost, lower, upper, fraction, m)
      if grew
        sub := grid[lower:upper]
        sub[sub == 1] :=  m
        sub[sub == 0] := -m
        cuboids[origin] := (grew, newLower, newUpper, fraction, m)
        growing := True
      else
        cuboids[origin] :=  (False, lower, upper, fraction, m)
      end
    end
  end
  uncoveredIndices := where(grid == 1)
  for index in uncoveredIndices
      grid[index] := material
      material += 1
  end
  return |grid|
end
```

Listing 3: Uniform Crossover Function

```
function crossover(parents           : List of all current chromosomes to be mutated
                   nOffspring        : Number of children to be created
                   minCuboidFraction : Global per-cuboid minimum volume fraction
                   grid              : Initial grid as 3D array of integers
                   cost              : Cost estimate as 3D array of integers)
begin
  offspring := [ ]
  iOffspring := 0
  // Add unmodified parents to offspring
  while size(offspring) < size(parents)
    offspring.append(parents[iOffspring])
    iOffspring += 1
```

```
  end
  // Fill up offspring pool with new children from parents
  iParents := randomShuffle([ (i,j)∈[0,size(parents)−1]² | i != j ])
  while size(offspring) < nOffspring
    parent1 := parents[iParents[iOffspring % size(iParents)][0]]
    parent2 := parents[iParents[iOffspring % size(iParents)][1]]
    child := Empty child chromosome
    seeds := Set of seed positions
    fractions := Map between seed positions and minimum volume fractions
    for i in range(size(parent1))
      seeds.add(parent1[i])
      if parent1[i] not in fractions
        fractions[parent1[i]] := parent1[i].fraction
      end
      seeds.add(parent2[i])
      if parent2[i] not in fractions
        fractions[parent2[i]] := parent2[i].fraction
      end
    end
    j := 0
    for i in randomSampleWithoutReplacement([0,...,size(seeds)-1], size(child))
      child[j].origin := seeds[i]
      child[j].fraction := fractions[seeds[i]]
      j += 1
    end
    offspring.append(child)
    iOffspring += 1
  end
  return offspring
end
```

Listing 4: Adaptive Mutation Function with increasing Exploration Pressure

```
function mutate(offspring          : List of all current chromosomes to be mutated
                iGeneration        : Count of past generations
                meanFitness        : Mean fitness of population
                minCuboidFraction  : Global per-cuboid minimum volume fraction
                minFlipProbability : Minimum probability of mutating active state
                maxFlipProbability : Maximum probability of mutating active state
                activeFraction     : Fraction of nMutate to allocate for active seeds
                grid               : Initial grid as 3D array of integers
                cost               : Cost estimate as 3D array of integers)
begin
  possibleSeeds := Locations of grid cells with value 1
  for chromosome in offspring
    nMutate := 1
    if computeFitness(chromosome) < meanFitness
      nMutate := nSeeds / 2
    end
    nActive := Count of active genes in chromosome
    nInactive := size(chromosome) - nActive
    mutationProbability := Vector of per-gene probability
    for i, gene in enumerate(chromosome)
      if gene.active
        mutationProbability[i] :=    activeFraction  / nActive
      else
        mutationProbability[i] := (1-activeFraction) / nActive
      end
    end
    mutationProbability /= sum(mutationProbability)
    flipProbability := minFlipProbability+(maxFlipProbability-minFlipProbability)*(exp(-0.05*iGeneration))
    for i in randomSampleWithoutReplacement(range(size(chromosome)), nMutate, mutationProbability)
      if uniformRandom(0.0, 1.0) < flipProbability
        chromosome[i].active := !chromosome[i].active
      else
        availableSeeds := possibleSeeds - seeds in chromosome
        availableSeedsProbabilities := cost[availableSeeds] / sum(cost[availableSeeds]))
        chromosome[i].origin := randomSample(availableSeeds, availableSeedsProbabilities)
        chromosome[i].fraction := randomValueBetween(minCuboidFraction, 1.0)
      end
    end
  end
  return offspring
end
```

Listing 5: Computation of Pertubed Velocity Profile via Vortex Method

```
function vortexMethod(seeds     : List of vortex seeds and their associated properties
                      inlet     : Geometric description of the inlet
                      u_mean    : Mean velocity profile
                      sigma     : Vortex size
                      intensity : Turbulence intensity
                      charT     : Characteristic time scale
```

```
                      iT        : Index of current discrete lattice timestep)
begin
  for seed in seeds
    seed.position := sampleRandomPositionIn(inlet)
    u := u_mean(seed.position)
    k := 3/2 * pow(norm(u) * intensity, 2)
    seed.vorticity := 4 * sqrt(pi/(6*ln(3)-9*ln(2))*(k*area(inlet))/count(seeds))
    if iT == latticeTime(charT*sigma/norm(u))
      seed.sign := randomSample([-1, 1])
    end
    seed.vorticity *= seed.sign
  end
  for latticeR in latticePositionsAt(inlet)
    u_vortex := [0,0,0]
    for seed in seeds
      diff := physicalPositionOf(latticeR) - seed.position
      cross := crossProduct(diff, inlet.normal)
      expTerm := exp(norm(diff)/(2*pow(sigma,2)))
      u_vortex += 1/(2*pi) * seed.vorticity * (cross * (1-expTerm)) / norm(diff)
    end
    u_grad := computeVelocityGradient(lattice[latticeR])
    u := u_mean(physicalPositionOf(latticeR))
    u += u_vortex - crossProduct(u_vortex, u_grad)/norm(u_grad) * inlet.normal
    lattice[latticeR].defineU(u)
  end
end
```

Glossary

CFD	Computational Fluid Dynamics
NSE	Navier-Stokes Equation
(R)ADE	(Reaction) Advection Diffusion Equation
LBM	Lattice Boltzmann Method
VM	Vortex Method Turbulent Velocity Inlet Condition
GLUPs	Giga Lattice Updates per Second (Number of Billions of Cells updated per Second)
GA	Genetic Algorithm

CRediT authorship contribution statement

Adrian Kummerländer: Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Conceptualization. **Fedor Bukreev:** Writing – review & editing, Visualization, Methodology. **Dennis Teutscher:** Writing – review & editing. **Marcio Dorn:** Writing – review & editing, Funding acquisition. **Mathias J. Krause:** Writing – review & editing, Supervision, Software, Resources, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This project was funded by the NHR@KIT Call for Collaboration project “OpenLB – An Open Source High Performance Lattice Boltzmann Code for Heterogeneous CPU-GPU Clusters” as well as DFG project number 436212129 “Increase of efficiency in phosphate recovery by understanding the interaction of flow and loading processes with modeling and simulation”. This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

The decomposition and rank assignment strategies as well as both application cases developed in the present work will be made available as open source in OpenLB.

Data availability

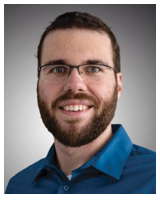
Data will be made available on request. The implementation will be made available as Open Source (GPL2) in OpenLB

References

[1] P. Carpenter, U.-H. Utz, S. Narasimhamurthy, E. Suarez, Heterogeneous high performance computing, https://doi.org/10.5281/zenodo.6090425, Feb. 2022.
[2] R. Caspart, S. Ziegler, A. Weyrauch, H. Obermaier, S. Raffeiner, L.P. Schuhmacher, J. Scholtyssek, D. Trofimova, M. Nolden, I. Reinartz, F. Isensee, M. Götz, C. Debus, Precise energy consumption measurements of heterogeneous artificial intelligence workloads, in: High Performance Computing. ISC High Performance 2022 International Workshops, Springer International Publishing, Cham, 2022.
[3] G. Freytag, M.S. Serpa, J.V.F. Lima, P. Rech, P.O.A. Navaux, Non-uniform partitioning for collaborative execution on heterogeneous architectures, in: 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2019.
[4] Highlights - June 2022 | TOP500 — top500.org, https://www.top500.org/lists/top500/2022/06/highs/, June 2022. (Accessed 9 May 2023).
[5] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, E.M. Viggen, The Lattice Boltzmann Method, 1st edition, Graduate Texts in Physics, Springer, Cham, 2017.
[6] D. Arumuga Perumal, A.K. Dass, A review on the development of lattice Boltzmann computation of macro fluid flows and heat transfer, Alex. Eng. J. 54 (4) (2015) 955–971, https://doi.org/10.1016/j.aej.2015.07.015.

[7] N. Fatima, I. Rajan, D. Arumuga Perumal, A. Sasithradevi, S.A. Ahmed, M. Gorji, Z. Ahmad, Simulation of fluid flow in a lid-driven cavity with different wave lengths corrugated walls using lattice Boltzmann method, J. Taiwan Inst. Chem. Eng. 144 (2023) 104748, https://doi.org/10.1016/j.jtice.2023.104748.
[8] S. Simonis, M. Frank, M. Krause, On relaxation systems and their relation to discrete velocity Boltzmann models for scalar advection–diffusion equations, Philos. Trans. R. Soc. A, Math. Phys. Eng. Sci. 378 (2175) (2020), https://doi.org/10.1098/rsta.2019.0400.
[9] S. Simonis, M. Haussmann, L. Kronberg, W. Dörfler, M.J. Krause, Linear and brute force stability of orthogonal moment multiple-relaxation-time lattice Boltzmann methods applied to homogeneous isotropic turbulence, Philos. Trans. R. Soc. A, Math. Phys. Eng. Sci. 379 (2208) (2021), https://doi.org/10.1098/rsta.2020.0405.
[10] A. Mink, K. Schediwy, C. Posten, H. Nirschl, S. Simonis, M.J. Krause, Comprehensive computational model for coupled fluid flow, mass transfer, and light supply in tubular photobioreactors equipped with glass sponges, Energies 15 (20) (2022), https://doi.org/10.3390/en15207671.
[11] S. Simonis, M. Frank, M.J. Krause, Constructing relaxation systems for lattice Boltzmann methods, Appl. Math. Lett. 137 (2023), https://doi.org/10.1016/j.aml.2022.108484.
[12] F. Bukreev, S. Simonis, A. Kummerländer, J. Jeßberger, M.J. Krause, Consistent lattice Boltzmann methods for the volume averaged Navier–Stokes equations, J. Comput. Phys. 490 (2023), https://doi.org/10.1016/j.jcp.2023.112301.
[13] M. Januszewski, M. Kostur, Sailfish: a flexible multi-GPU implementation of the lattice Boltzmann method, Comput. Phys. Commun. 185 (9) (Sep. 2014), arXiv:1311.2404, https://doi.org/10.1016/j.cpc.2014.04.018.
[14] C. Feichtinger, J. Habich, H. Köstler, U. Rüde, T. Aoki, Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on cpu–gpu clusters, Parallel Comput. 46 (2015), https://doi.org/10.1016/j.parco.2014.12.003.
[15] M. Mohrhard, G. Thäter, J. Bludau, B. Horvat, M.J. Krause, Auto-vectorization friendly parallel lattice Boltzmann streaming scheme for direct addressing, Comput. Fluids 181 (Mar. 2019), https://doi.org/10.1016/j.compfluid.2019.01.001.
[16] M. Bauer, S. Eibl, C. Godenschwager, N. Kohl, M. Kuron, C. Rettinger, F. Schornbaum, C. Schwarzmeier, D. Thönnes, H. Köstler, U. Rüde, walberla: a block-structured high-performance framework for multiphysics simulations, in: Computers & Mathematics with Applications 81, Development and Application of Open-Source Software for Problems with Numerical PDEs, 2021.
[17] A. Kummerländer, M. Dorn, M. Frank, M.J. Krause, Implicit propagation of directly addressed grids in lattice Boltzmann methods, Concurr. Comput. Pract. Exp. 35 (8) (2023) e7509, https://doi.org/10.1002/cpe.7509.
[18] C. Feichtinger, J. Habich, H. Köstler, G. Hager, U. Rüde, G. Wellein, A flexible patch-based lattice Boltzmann parallelization approach for heterogeneous gpu–cpu clusters, in: Emerging Programming Paradigms for Large-Scale Scientific Computing, Parallel Comput. 37 (9) (2011), https://doi.org/10.1016/j.parco.2011.03.005.
[19] C. Riesinger, A. Bakhtiari, M. Schreiber, P. Neumann, H.-J. Bungartz, A holistic scalable implementation approach of the lattice Boltzmann method for cpu/gpu heterogeneous clusters, Computation 5 (4) (2017), https://doi.org/10.3390/computation5040048.
[20] E. Calore, A. Gabbana, S. Schifano, R. Tripiccione, Optimization of lattice Boltzmann simulations on heterogeneous computers, Int. J. High Perform. Comput. Appl. 33 (1) (2019), https://doi.org/10.1177/1094342017703771.
[21] G. Freytag, M.S. Serpa, J.V. Lima, P. Rech, P.O. Navaux, Collaborative execution of fluid flow simulation using non-uniform decomposition on heterogeneous architectures, J. Parallel Distrib. Comput. 152 (2021), https://doi.org/10.1016/j.jpdc.2021.02.006.
[22] C. Wei, W. Zhenghua, L. Zongzhe, Y. Lu, W. Yongxian, An improved lbm approach for heterogeneous gpu-cpu clusters, in: 2011 4th International Conference on Biomedical Engineering and Informatics (BMEI), vol. 4, 2011.
[23] P.L. Bhatnagar, E.P. Gross, M. Krook, A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems, Phys. Rev. E 94 (3) (May 1954), https://doi.org/10.1103/PhysRev.94.511.
[24] M.J. Krause, A. Kummerländer, S.J. Avis, H. Kusumaatmaja, D. Dapelo, F. Klemens, M. Gaedtke, N. Hafen, A. Mink, R. Trunk, J.E. Marquardt, M.-L. Maier, M. Haussmann, S. Simonis, Openlb—open source lattice Boltzmann code, in: Development and Application of Open-Source Software for Problems with Numerical PDEs, Comput. Math. Appl. 81 (2021), https://doi.org/10.1016/j.camwa.2020.04.033.
[25] A. Kummerländer, S. Avis, H. Kusumaatmaja, F. Bukreev, M. Crocoll, D. Dapelo, N. Hafen, S. Ito, J. Jeßberger, J.E. Marquardt, J. Mödl, T. Pertzel, F. Prinz, F. Raichle, M. Schecher, S. Simonis, D. Teutscher, M.J. Krause, OpenLB Release 1.6: Open Source Lattice Boltzmann Code, Apr. 2023.
[26] A. Kummerländer, Lattice Boltzmann Performance Engineering in OpenLB, hiRSE Seminar Series, Dec. 2022.
[27] A. Kummerländer, M.J. Krause, Research software engineering in OpenLB: refactoring a legacy code to state-of-the-art performance, deRSE23 conference for research software engineering in Germany, Paderborn (2023), https://doi.org/10.5281/zenodo.7662082, Feb. 2023.
[28] A. Kummerländer, F. Bukreev, S. Berg, M. Dorn, M.J. Krause, Advances in computational process engineering using lattice Boltzmann methods on high performance computers, in: High Performance Computing in Science and Engineering '22, Springer, 2024.
[29] V. Heuveline, M.J. Krause, J. Latt, Towards a hybrid parallelization of lattice Boltzmann methods, in: Mesoscopic Methods in Engineering and Science, Comput. Math. Appl. 58 (5) (2009), https://doi.org/10.1016/j.camwa.2009.04.001.
[30] A. Kummerländer, S. Avis, H. Kusumaatmaja, F. Bukreev, D. Dapelo, S. Großmann, N. Hafen, C. Holeksa, A. Husfeldt, J. Jeßberger, L. Kronberg, J. Marquardt, J. Mödl, J. Nguyen, T. Pertzel, S. Simonis, L. Springmann, N. Suntoyo, D. Teutscher, M. Zhong, M. Krause, OpenLB Release 1.5: Open Source Lattice Boltzmann Code, Apr. 2022.
[31] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M.B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li, F. Marson, J. Lemus, C. Kotsalos, R. Conradin, C. Coreixas, R. Petkantchin, F. Raynaud, J. Beny, B. Chopard, Palabos: parallel lattice Boltzmann solver, in: Development and Application of Open-Source Software for Problems with Numerical PDEs, Comput. Math. Appl. 81 (2021), https://doi.org/10.1016/j.camwa.2020.03.022.
[32] I. Zacharoudiou, J. McCullough, P. Coveney, Development and performance of a hemelb gpu code for human-scale blood flow simulation, Comput. Phys. Commun. 282 (2023), https://doi.org/10.1016/j.cpc.2022.108548.
[33] J. Fietz, M.J. Krause, C. Schulz, P. Sanders, V. Heuveline, Optimized hybrid parallel lattice Boltzmann fluid flow simulations on complex geometries, in: C. Kaklamanis, T. Papatheodorou, P.G. Spirakis (Eds.), Euro-Par 2012 Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
[34] C.B. Sullivan, A. Kaszynski, PyVista: 3d plotting and mesh analysis through a streamlined interface for the visualization toolkit (VTK), J. Open Source Softw. 4 (37) (may 2019), https://doi.org/10.21105/joss.01450.
[35] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S.J. van der Walt, M. Brett, J. Wilson, K.J. Millman, N. Mayorov, A.R.J. Nelson, E. Jones, R. Kern, E. Larson, C.J. Carey, İ. Polat, Y. Feng, E.W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E.A. Quintero, C.R. Harris, A.M. Archibald, A.H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy 1.0 contributors, SciPy 1.0: fundamental algorithms for scientific computing in python, Nat. Methods 17 (2020), https://doi.org/10.1038/s41592-019-0686-2.
[36] A.F. Gad, Pygad: an intuitive genetic algorithm python library, arXiv:2106.06158 [abs], 2021, https://api.semanticscholar.org/CorpusID:235417393.
[37] N. Satofuka, T. Nishioka, Parallelization of lattice Boltzmann method for incompressible flow computations, Comput. Mech. 23 (2) (1999), https://doi.org/10.1007/s004660050397.
[38] D. Vidal, R. Roy, F. Bertrand, On improving the performance of large parallel lattice Boltzmann flow simulations in heterogeneous porous media, Comput. Fluids 39 (2) (2010), https://doi.org/10.1016/j.compfluid.2009.09.011.
[39] L. Axner, J. Bernsdorf, T. Zeiser, P. Lammers, J. Linxweiler, A. Hoekstra, Performance evaluation of a parallel sparse lattice Boltzmann solver, J. Comput. Phys. 227 (10) (May 2008), https://doi.org/10.1016/j.jcp.2008.01.013.
[40] D. Groen, D.A. Chacra, R.W. Nash, J. Jaros, M.O. Bernabeu, P.V. Coveney, Weighted decomposition in high-performance lattice-Boltzmann simulations: are some lattice sites more equal than others?, in: Solving Software Challenges for Exascale, vol. 8759, Springer International Publishing, 2015.
[41] S.N. Sivanandam, S.N. Deepa, Introduction to Genetic Algorithms, Springer, 2010.
[42] S. Marsili Libelli, P. Alba, Adaptive mutation in genetic algorithms, Soft Comput. 4 (2000), https://doi.org/10.1007/s005000000042.
[43] M. Krafczyk, J. Tölke, L.-S. Luo, Large-eddy simulations with a multiple-relaxation-time lbe model, Int. J. Mod. Phys. B 17 (2003), https://doi.org/10.1142/S0217979203017059.
[44] H. Yu, S.S. Girimaji, L.-S. Luo, Dns and les of decaying isotropic turbulence with and without frame rotation using lattice Boltzmann method, J. Comput. Phys. 209 (2) (2005), https://doi.org/10.1016/j.jcp.2005.03.022.
[45] P. Nathen, D. Gaudlitz, M. Krause, J. Kratzke, An extension of the lattice Boltzmann method for simulating turbulent flows around rotating geometries of arbitrary shape, in: 21st AIAA Computational Fluid Dynamics Conference, American Institute of Aeronautics and Astronautics, San Diego, CA, 2013.
[46] A. Kummerländer, S. Avis, H. Kusumaatmaja, F. Bukreev, M. Crocoll, D. Dapelo, S. Großmann, N. Hafen, S. Ito, J. Jeßberger, E. Kummer, J.E. Marquardt, J. Mödl, T. Pertzel, F. Prinz, F. Raichle, M. Sadric, M. Schecher, D. Teutscher, S. Simonis, M.J. Krause, OpenLB user guide: associated with release 1.6 of the code (2023), arXiv:2307.11752, https://doi.org/10.48550/arXiv.2307.11752.
[47] E. Sergent, Vers une méthodologie de couplage entre la simulation des grandes échelles et les modèles statistiques, Ph.D. thesis, thèse de doctorat dirigée par Bertoglio, Jean-Pierre Mécanique Ecully, Ecole centrale de Lyon 2002, 2002, http://www.theses.fr/2002ECDL0019.

[48] ANSYS Inc., Ansys Fluent Theory Guide Associated with Release 2021R2, July 2021.
[49] M. Hettel, F. Bukreev, E. Daymo, A. Kummerländer, M.J. Krause, O. Deutschmann, Calculation of single and multiple low Reynolds number free jets with a lattice-Boltzmann method, AIAA J. 63 (4) (2025) 1305–1318, https://doi.org/10.2514/1.J064280.
[50] P.A. Skordos, Initial and boundary conditions for the lattice Boltzmann method, Phys. Rev. E 48 (Dec 1993), https://doi.org/10.1103/PhysRevE.48.4823.
[51] Q. Zou, X. He, On pressure and velocity boundary conditions for the lattice Boltzmann BGK model, Phys. Fluids 9 (6) (1997), https://doi.org/10.1063/1.869307.
[52] M. Bouzidi, M. Firdaouss, P. Lallemand, Momentum transfer of a Boltzmann-lattice fluid with boundaries, Phys. Fluids 13 (11) (2001), https://doi.org/10.1063/1.1399290.
[53] B.K. Johnson, R.K. Prud'homme, Chemical processing and micromixing in confined impinging jets, AIChE J. 49 (9) (2003), https://doi.org/10.1002/aic.690490905.
[54] F. Bukreev, A. Kummerländer, J. Jeßberger, D. Teutscher, S. Simonis, D. Bothe, M.J. Krause, Benchmark simulation of laminar reactive micromixing using lattice Boltzmann methods, AIAA J. 63 (4) (2025) 1295–1304, https://doi.org/10.2514/1.J064234, publisher: American Institute of Aeronautics and Astronautics.



Adrian Kummerländer is a doctoral student of the Lattice Boltzmann Research Group (LBRG) at the Karlsruhe Institute of Technology (KIT) in Germany. He studied mathematics with specialization in computer science at KIT, with bachelor's and master's theses focused on grid refinement and high performance LBM. He is a professional software developer and has been a core contributor to the open source LBM framework OpenLB since joining the LBRG during his undergraduate studies. His research interests focus on performance engineering, fluid structure interaction and HPC applications of LBM.

Fedor Bukreev is a doctoral student of the Lattice Boltzmann Research Group (LBRG) at the Karlsruhe Institute of Technology (KIT) in Germany. He studied mechanical engineering and chemical processing technology at RWTH Aachen and Ruhr-University Bochum, with his bachelor's and master's theses focused on multiphysics flow modeling. In the LBRG, he contributes to the open source LBM framework OpenLB, where he develops and integrates physical models. His research interests focus on multiphysics flows with chemical reactions and turbulence models, as well as fluid-structure interactions applications of the Lattice Boltzmann Methods.

Mathias J. Krause studied mathematics with economics at the Universität Karlsruhe (TH) in Germany and the Cardiff University in Wales (2006). He completed his doctorate (2010) and habilitation (2020) at the Karlsruhe Institute of Technology (KIT). Since 2013, he heads the interdisciplinary Lattice Boltzmann Research Group (LBRG) at KIT. His research interests are mainly dedicated to the fields of applied mathematics, applied computer science with strong focus on HPC, CFD and optimization under the constraints of PDE. He is initiator and main author of OpenLB, an open source software which allows everyone to apply his scientific findings to solve large-scale biological, chemical, mechanical and medical engineering problems. His work was honored with several prizes and a membership as WIN-Kollegiat at the Heidelberg Academy of Science.