Title

Optimized GPU Implementation of Grid Refinement in Lattice Boltzmann Method

Permalink

https://escholarship.org/uc/item/0x86w4w1

Authors

Mahmoud, Ahmed H.
Salehipour, Hesam
Meneghin, Massimiliano

Publication Date

2024-06-01

Optimized GPU Implementation of Grid Refinement in Lattice Boltzmann Method

Ahmed H. Mahmoud[1,2], Hesam Salehipour[1], Massimiliano Meneghin[1]
[1]Autodesk Research, Canada, [2]University of California, Davis
{ahmed.mahmoud, hesam.salehipour, massimiliano.meneghin}@autodesk.com

***Abstract*—Nonuniform grid refinement plays a fundamental role in simulating realistic flows with a multitude of length scales. We introduce the first GPU-optimized implementation of this technique in the context of the lattice Boltzmann method. Our approach focuses on enhancing GPU performance while minimizing memory access bottlenecks. We employ kernel fusion techniques to optimize memory access patterns, reduce synchronization overhead, and minimize kernel launch latencies. Additionally, our implementation ensures efficient memory management, resulting in lower memory requirements compared to the baseline LBM implementations that were designed for distributed systems. Our implementation allows simulations of unprecedented domain size (e.g., $1596 \times 840 \times 840$) using a single A100-40 GB GPU thanks to enabling grid refinement capabilities on a single GPU. We validate our code against published experimental data. Our optimization improves the performance of the baseline algorithm by 1.3–2X. We also compare against state-of-the-art current solutions for grid refinement LBM and show an order of magnitude speedup.**

***Index Terms*—Parallel, GPU, Simulation, LBM, Boltzmann, Refinement**

I. Introduction

Complex fluid flows are prevalent in engineering systems and natural environments, and hence simulating their behavior is of paramount importance in many disciplines ranging from vehicle aerodynamics and building ventilation to weather forecasting and planetary currents. Often in common between all these various applications is the substantially wide range of spatial scales that need to be resolved numerically in order to capture the realistic fluid flow dynamics accurately. For example, considering the simulation of air flow around buildings, one needs to enclose the open environment using a computational box that is sufficiently large to eliminate the effects of the boundary conditions while resolving small geometric features of interest in the building (e.g., balconies and windows) notwithstanding the microscale turbulent features of the flow that are even orders of magnitude smaller.

Given this multiscale nature of such fluid-related phenomena, the field of computational fluid dynamics has evolved to enable accurate, efficient, and fast simulations that rely on nonuniform grids with enhanced refinements around the regions of interest [1]. Over the past two decades, the Lattice Boltzmann Method (LBM) has gained tremendous popularity as an alternative *mesoscopic* representation of fluid flows which have been classically formulated using a macroscopic representation based on the Navier-Stokes (N-S) equations and solved numerically using various methods such as finite-difference, finite-volume or finite element. While LBM is proven to be equivalent to the weakly compressible N-S equations up to second-order accuracy in space and time (e.g., through the Chapman-Enskong analysis [2]), its localized formulations make it ideally suited for massive parallelization, especially on GPUs. The massively parallel nature of LBM computation has made it an appealing CFD solution for commercial purposes, e.g., XFlow, PowerFLOW [3], and ultraFluidX [4].

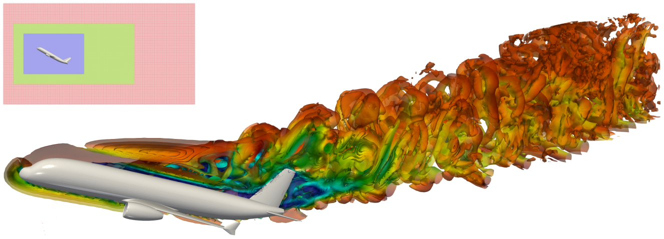
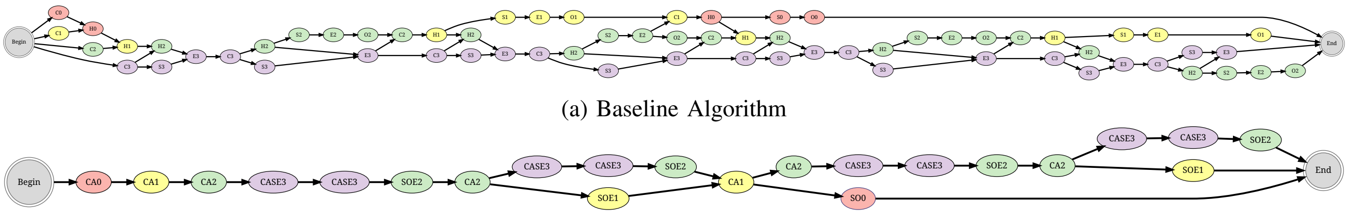


Fig. 1: Our optimized LBM grid refinement facilitates large-scale fluid simulations, e.g., airflow over an airplane within a virtual wind tunnel of size $1596 \times 840 \times 840$ on a single GPU.

Optimizing the GPU implementation of LBM has therefore gained much attention in recent years leading to the development of just-in-time compiler-based implementations [6] as well as advanced algorithms on uniform grid such as the AA-method [7], Esoteric Twist [8], and reduced/mixed precision [9]. The goal of all these optimizations is to reduce the memory requirements of LBM and ultimately improve the wall-clock performance of the simulation [10].

As discussed earlier, large-scale CFD simulations are only possible by relying on nonuniform grid refinement. The memory-bounded computations associated with LBM further necessitate the need for efficient implementations of this approach on the GPU. However, a robust grid refinement algorithm in LBM involves multiple kernels and complex data dependencies and synchronizations (Figure 2, top) which makes caching obsolete and may result in poor performance without careful hardware-specific optimizations. Various prior investigations have focused on implementing the grid refinement technique in LBM either on single-GPU or multi-GPU architecture. An implementation of the grid refinement technique in LBM on both CPU and a single GPU was presented for 2D domains in [11] which was then extended to heterogeneous CPU-GPU machines in [12]. Other works [13],



(a) Baseline Algorithm

(b) Our Optimized Implementation

Fig. 2: Data dependency graph of the baseline grid refinement LBM algorithm [5] ported to the GPU (top) and our optimized implementation (bottom). The baseline has complex dependencies because of multiple kernels required for different levels. Each node in this graph represents a GPU kernel where C is Collision, E is Explosion, S is Streaming, O is Coalescence and A is Accumulate as will be explained further in Section IV. The number represents the grid level where the coarsest level is 0. Our implementation is simpler because of our aggressive kernel fusion (around three times fewer kernels) which makes our algorithm 1.3–2X faster than the baseline algorithm.

[14] focused on scalability on multi-GPU systems, adaptive mesh refinement on multiple GPUs [15] and load-balancing in distributed CPU-based systems [16], [17]. While these related works also aimed to develop grid refinement for LBM on GPU, to our knowledge, there is no prior work that focuses primarily on optimizing the single GPU implementation of grid refinement in LBM. We believe this is a crucial precursory investigation before further optimizations are undertaken for multi-GPU systems.

In this paper, we propose an optimized implementation of the LBM grid refinement method that utilizes the GPU hardware efficiently. Our optimized implementation is based on aggressive kernel fusion (Figure 2, bottom) leading to improved memory accesses, reduced synchronization, reduced kernel launch overhead, and simpler code. Additionally, our implementation uses less memory than the baseline algorithm by reducing the size of the ghost layer. We show the effectiveness of our optimized implementation and compare it against CPU and GPU implementations where we achieve an order of magnitude speedup. While there exist many open-source LBM codes, ours is the first to implement the grid refinement method on the GPU. Our code is available at https://github.com/Autodesk/Neon.

II. Mathematical Background

The LBM equations determine the time-evolution of a collection of fictitious particles, represented by time-dependent *velocity distribution functions* (f_i) along a set of discrete lattice directions denoted by $e_i = (e_1, \ldots, e_q)$. In this work, we employ 3D lattice structures with 19 (D3Q19) or 27 (D3Q27) directions. The values of f_i are evolved in time through a *collide-and-stream* algorithm. If we denote the nonlinear and computationally local collision operation by $\mathcal{C}$, we may represent the collide-and-stream algorithm as,

$$\text{Collision:}\ \ f_i^*(x,t) = \mathcal{C}(f_i(x,t)) \tag{1}$$

$$\text{Streaming:}\ f_i(x + e_i\Delta x, t + \Delta t) = f_i^*(x,t) \tag{2}$$

Throughout this paper, we use two collision operators, namely (i) the single-relaxation collision model of Bhatnagar-Gross-Krook (BGK) [2], and (ii) the multi-relaxation entropic model of Karlin-Bösch-Chikatamarla (KBC) [18]. Without loss of generality and for conciseness, we only present the simple BGK model here in which $\mathcal{C}$ is defined as,

$$f_i^*(x,t) = f_i(x,t) - \frac{\Delta t}{\tau}\{f_i(x,t) - f_i^{eq}(x,t)\} \tag{3}$$

In the above equations Δt is the time step and τ is a *relaxation time*, determined by the total kinematic viscosity ν of the fluid as,

$$\tau = \frac{\nu}{c_s^2} + \frac{\Delta t}{2}. \tag{4}$$

or in other words, $\nu = c_s^2(\tau - \Delta t/2)$.

In Equation (1), f_i^{eq} represents the components of the *equilibrium* distribution function and is defined as,

$$f_i^{eq}(x,t) = \\ w_i\rho(x,t)\left\{1 + \frac{e_i \cdot u(x,t)}{c_s^2} + \frac{(e_i \cdot u(x,t))^2}{2c_s^4} - \frac{\|u(x,t)\|^2}{2c_s^2}\right\} \tag{5}$$

where w_i are weights associated with each lattice direction e_i. The fluid density $\rho(x,t)$, velocity $u(x,t)$ and pressure $p(x,t)$ are formally derived from f_i as,

$$\text{density:}\quad \rho(x,t) = \sum_i f_i(x,t) \tag{6}$$

$$\text{velocity:}\quad u(x,t) = \frac{1}{\rho(x,t)}\sum_i e_i f_i(x,t) \tag{7}$$

$$\text{pressure:}\quad p(x,t) = c_s^2\rho(x,t) \tag{8}$$

in which c_s is the constant speed of sound given by $c_s^2 = (1/3)(\Delta x/\Delta t)^2$.

A. Grid Refinement

Various techniques for grid-refinement in LBM have been proposed in the literature which may be broadly categorized into *node-based* methods [11], [19]–[22] and *volume-based* methods [16], [23]–[25]. In the node-based methods, f_i is stored at the cell vertices leading to shared vertices between coarse and fine cells. In order to impose conservation of mass and preserve second-order accuracy, the node-based methods often resort to scheme-dependent rescaling of f_i across transitions between levels and rely on ad hoc interpolation and filtering schemes that may lead to spurious results [19]. However, in volume-based techniques, f_i values of two subsequent grid levels are not colocated but staggered which renders the communication between coarse and fine levels more straightforward. As a result, in volume-based methods a simple homogeneous redistribution of f_i acts as the required interpolation scheme for coarse-to-fine communications while conserving mass and momentum [23].

In this paper, we adopt the volume-based method of Rohde et al. [25] as implemented by Schornbaum et al. [5], [16]. We introduce some novel changes to this algorithm that are aimed at optimizing the computational performance of the method when implemented on the GPU.

We select a refinement ratio of 2 where a coarse cell at level L is uniformly divided into 2^d cells–where d is the dimension– to arrive at level $L+1$, or in other words $\Delta x_{L+1} = \Delta x_L/2$. For neighboring cells that interface two grid levels, a maximum jump in grid level of $\Delta L = 1$ is allowed. Due to acoustic scaling which requires the speed of sound c_s to remain constant across various grid levels [11], $\Delta t_L \propto \Delta x_L$ and hence $\Delta t_{L+1} = \Delta t_L/2$. In addition, the fluid viscosity ν must also remain constant on each grid level which implies:

$$\nu_L = \nu_0 \rightarrow c_s^2(\tau_L - \frac{\Delta t_L}{2}) = c_s^2(\tau_0 - \frac{\Delta t_0}{2})$$

or equivalently:

$$\frac{\tau_L}{\Delta t_L} = 2^L\left(\frac{\tau_0}{\Delta t_0}\right) + \frac{1}{2}(1-2^L)$$

It is often convenient to define a non-dimensional relaxation parameter $\omega = \Delta t/\tau$ and hence, ω_L on various grid levels are obtained based on its value at the coarsest level or ω_0 as,

$$\omega_L = \frac{2\omega_0}{2^{L+1} + (1-2^L)\omega_0} \tag{9}$$

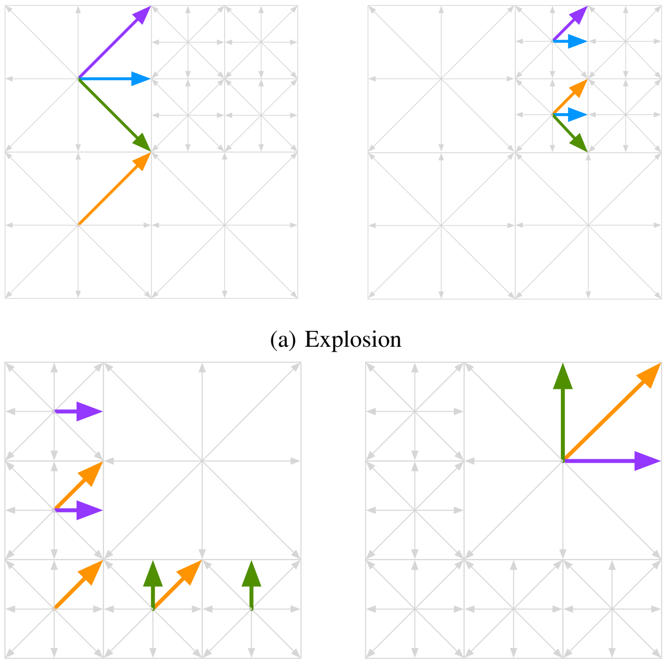
The coarse-to-fine and fine-to-coarse communication steps across the grids at level L and $L+1$ (also introduced later in the paper as the *Explosion* and *Coalescence* steps) are defined as,

$$\text{Explosion:} \quad f_i(x_{\beta,L+1}, t) = f_i(x_{\alpha,L}, t) \tag{10}$$

$$\text{Coalescence:} \quad f_i(x_{\alpha,L}, t) = \frac{1}{n}\sum_{\beta=1}^{n} f_i(x_{\beta,L+1}, t) \tag{11}$$

which correspond to a homogeneous redistribution of f_i for the coarse-to-fine communication (Explosion) and a simple averaging of f_i across n fine cells (denoted by $x_{\beta,L+1}$) that interface with a coarse cell (denoted by $x_{\alpha,L}$) for the fine-to-coarse communication (Coalescence).

In the remainder of this paper, we employ the *LBM units* in which it is conventional to non-dimensionalize space and time by Δx and Δt. As a result, in our accompanied open-source code all the parameters are assumed to be dimensionless, for instance, $\Delta x = \Delta t = 1$ and $c_s^2 = 1/3$.



(a) Explosion

(b) Coalescence

Fig. 3: Besides collision and streaming, the nonuniform grid refinement in LBM involves (a) coarse-to-fine *explosion*, and (b) fine-to-coarse *coalescence*. The colored arrows highlight how a few of the distribution functions f_i travel along certain directions e_i.

III. Baseline Grid Refinement Algorithm

Our baseline algorithm is that by Schornbaum et al. [5], [16], which has been designed for distributed systems. This algorithm uses a strongly balanced octree grid where, as indicated earlier, the transition in resolution from one level to another is strictly 1 in all directions. We denote an octree level with L, where $L = 0$ implies the coarsest level of the tree. The algorithm input is a grid that tiles the simulation domain at potentially different resolutions. The algorithm depends on four main computational components (Figure 3)

1) *Collision*: computes f_i^* using f_i as per Equation (1) similar to the uniform LBM algorithm. This step involves only local computation and does not require any communications with other cells and hence is not impacted by grid refinement.
2) *Streaming*: propagates the information in time and space by conducting a shift operation as per Equation (2). We designate the same-level Streaming simply by "Streaming" and denote the cross-level Streaming by *Explosion* and *Coalescence*.
3) *Explosion*: performs coarse-to-fine communications to propagate f_i from the coarse neighbor into finer cells as per Equation (10). Explosion is a one-to-many communication where a coarse cell distributes its values of f_i along e_i to fine cells that interface with that coarse cell

along $-e_i$ (Figure 3a).

4) *Coalescence*: performs fine-to-coarse communications to propagate f_i from the fine neighbor into coarse cells as per Equation (11). Coalescence is a many-to-one communication where a coarse cell averages the contributions of f_i from its neighboring fine cells along a certain direction indicated by the e_i vector (Figure 3b).

At a high level, the algorithm applies the standard LBM *collide-and-stream* algorithm at each level starting from the coarsest level. However, during the streaming step, the interface between different levels requires special treatment and this is where the uniform versus nonuniform LBM algorithm differs. Additionally, for a refinement ratio of two between levels L and $L+1$, the algorithm needs to complete two time steps at level $L+1$ before proceeding to the next time step of the coarse level due to the acoustic scaling (see Section II). In other words, given a grid with L levels, the finest grid will perform 2^{L-1} time steps to complete one time step on the coarsest level.

In order to facilitate the communication between different levels of the grid, the interface between a grid at level L and a finer neighbor at level $L+1$ is extended by a *ghost layer* (Figure 4a) which lives in the finer $L+1$ grid and covers two coarser layers from the grid at level L. These ghost layers are used for communicating interface information both ways, i.e., from coarse to fine and from fine to coarse. We note that all reads/writes could be done as a gather or scatter operation without the need for any atomic mechanism thanks to these added ghost layers.

The baseline method in Algorithm 1 shows a single time step of the *coarsest* level. This step is repeated in a loop until a user-specified stopping criterion is met, e.g., based on the maximum number of iterations. To succinctly describe one time step of the baseline method in Algorithm 1, we assume a two-level grid: L (coarse) and $L+1$ (fine). Starting from the post-streaming state, the algorithm performs a Collision step on L and $L+1$. Then, an Explosion operation populates ghost layers of $L+1$ by copying two layers of L along the interface between L and $L+1$. This is followed by a Streaming operation in L and $L+1$ including the innermost layer of the ghost cells. Then, another Collision and Streaming steps are performed on $L+1$. Finally, a Coalescence step is performed where the innermost ghost layer populations in $L+1$ are averaged and copied to the overlapping corresponding coarse cells in L.

IV. Analysis and Optimization

While the majority of previous work focused on distributed systems, here we analyze the performance of the baseline algorithm (Section III) on a single GPU and investigate its potential bottlenecks to pave the way for its optimization. The main drawback of implementing the baseline algorithm in its original form on the GPU is that every operation is performed in isolation—missing the opportunity for kernel fusion. This leads to loading the whole grid multiple times to operate only on a small subset of it, e.g., during Explosion and Coalescence. Here, we demonstrate how the baseline algorithm can be modified to enable kernel fusion, which is otherwise non-trivial. We will show that Streaming, Explosion, and Coalescence may all be combined into a single kernel.

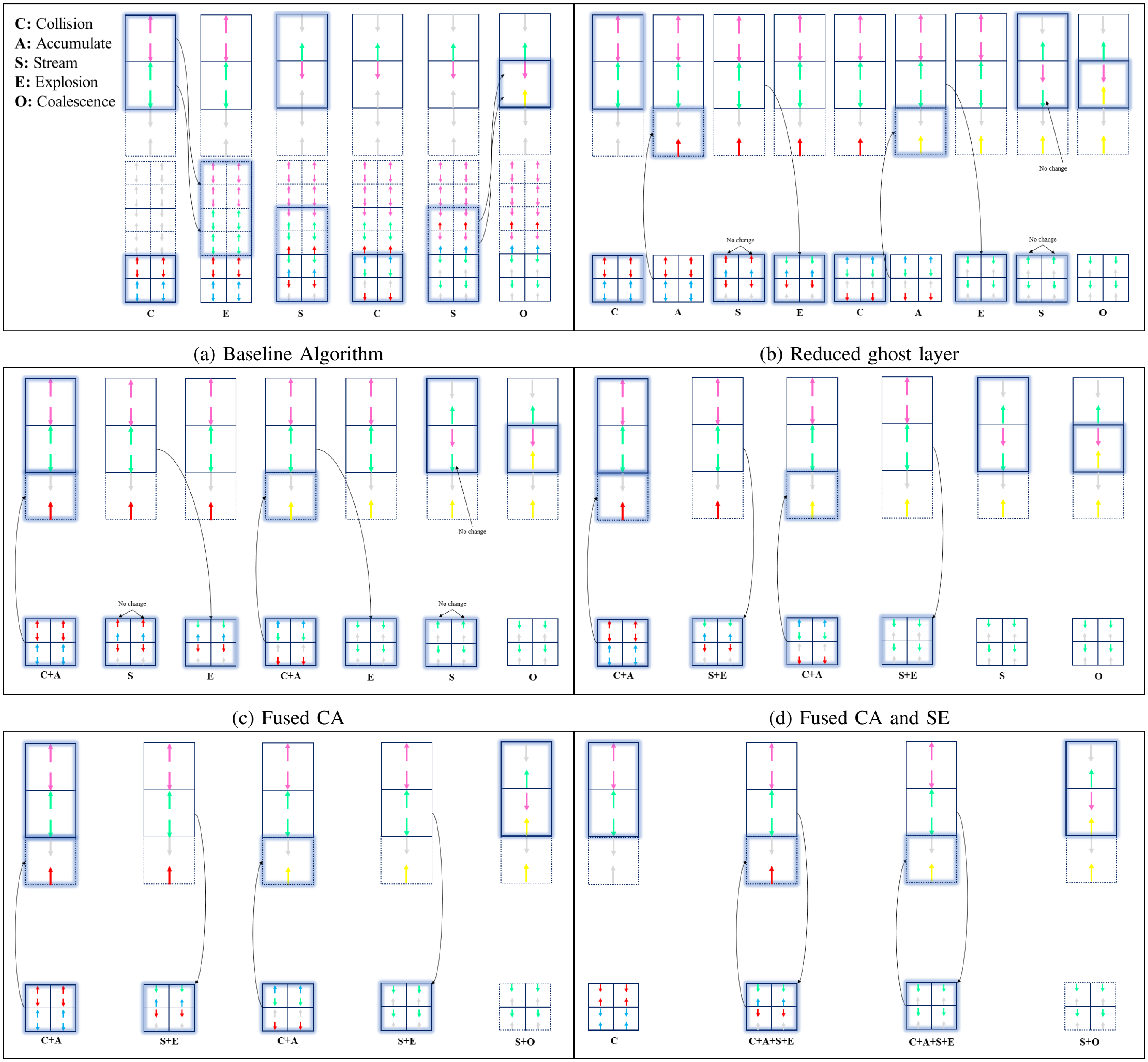
Algorithm 1: Baseline Grid Refinement LBM Algorithm [5]

```
Function NonUniformTimeStep($L$):
    Collision($L$)
    if $L \neq L_{max} - 1$ then
        NonUniformTimeStep($L+1$)
    end
    if $L \neq 0$ then
        Explosion($L, L-1$)
    end
    Streaming($L$)
    if $L \neq L_{max} - 1$ then
        Coalescence($L, L+1$)
    end
    if $L == 0$ then
        return
    end
    Collision($L$)
    if $L \neq L_{max} - 1$ then
        NonUniformTimeStep($L+1$)
    end
    if $L \neq 0$ then
        Explosion($L, L-1$)
    end
    Streaming($L$)
    if $L \neq L_{max} - 1$ then
        Coalescence($L, L+1$)
    end
```

A. Reducing Ghost Layers

The baseline algorithm uses four ghost layers on the fine grid which overlap two coarse layers. These ghost layers are employed to duplicate the overlapping coarse layers in order to be leveraged (only the two innermost layers) during the Streaming step of the fine level. The fine ghost layers also allow the Coalescence step to be performed as gather operations (initiated by the coarse layer) avoiding atomic operations. However, this approach is only reasonable if the distance in memory between the fine and coarse grid is large. On a single GPU, four ghost layers are excessive and may limit the maximum size of the problem that fits in a single GPU. Additionally, depending on the data structure (see Section V), the distance between grids at different resolutions may not be large enough to warrant manual caching.

Instead of allocating the ghost layer on the fine level, we allocate only a single layer on the coarse layer effectively reducing its size to 1/3 of what is needed by the baseline algorithm. In this case, during the Explosion step (i.e., coarse-



(a) Baseline Algorithm

(b) Reduced ghost layer

(c) Fused CA

(d) Fused CA and SE

(e) Fused CA, SE, and SO

(f) Fused CASE and SO

Fig. 4: Comparing the baseline algorithm against different optimization opportunities using a two-level grid. While the baseline algorithm (top left) uses a larger ghost layer (dotted cells), the ghost layer in our implementation (top right) is allocated on the coarse level effectively reducing its size to 1/3. Additional optimizations include fusing Collision (C) and Accumulate (A) steps by performing the Accumulate step as atomic write (middle left). We could also fuse Streaming (S) and Explosion (E) (middle right) as well as Streaming and Coalescence (O) steps (bottom left). Finally, we could fuse all steps of a single time step of the finest resolution (bottom right).

to-fine communication), the fine grid reads directly from the coarse grid.

We rely on this single ghost layer on the coarse grid to prepare the information needed by the interface cells of the coarse level. This approach turns the Coalescence step into a simple Streaming-like step (Figure 4b). As a result, the Coalescence step is split into two operations: (i) an *Accumulate* step on the ghost layer after every fine-level Collision step, and (ii) a Coalescence step by the coarse level to read and average the accumulated information. The Accumulate step is the same as the coarse-level accumulation in the baseline algorithm (Equation 11) but without division. The division (to average the data) is done now during the Coalescence on the coarse level. The Accumulate step could be performed either as a gather read operation from the ghost layer or scatter *atomic* write operation from the fine level. Here, we note that the contention is not too high as every ghost cell will be written by a maximum of 8 other fine cells. After two Accumulate steps, the information is ready for the coarse layer to do its Coalescence step.

B. Kernel Fusion

The modifications explained thus far open the door for a variety of options for kernel fusion. For example, the Collision and Accumulate steps could be fused in a single kernel (Figure 4c). In this kernel, every cell performs its Collision operations, and if it is on the interface with a coarse cell, it will simply Accumulate its populations which are now stored in registers. When the coarse cell performs its Coalescence step, it will reset the ghost layer allowing subsequent Accumulate steps to be done correctly. Note that if the Accumulate step is done as a gather operation by the coarse cell, the data dependency would not allow kernel fusion since the coarse level should wait for the fine level to finish its Collision step before performing the Accumulate step.

Additionally, the Explosion and Streaming steps (Figure 4d), and Coalescence and Streaming steps (Figure 4e) could be fused since the Explosion and Coalescence are now simply a sub-step of the Streaming step where the missing populations live on a grid at a different resolution. When there are more than two levels, the middle level can fuse the Streaming, Explosion, and Coalescence steps all in one kernel. Finally, and similar to uniform LBM [10], the Collision and Streaming can be fused in one kernel (Figure 4f) for the finest resolution. Since the majority of the computation is done on the finest level, this fusion is expected to be extremely beneficial. Unlike the uniform LBM, this final kernel fusion also requires the Collision and Streaming steps to perform the Accumulate and Explosion steps respectively.

V. Implementation Details

The grid refinement LBM algorithm is based on a typical octree-based spatial organization—since the branching factor from one level to another is two. The algorithm defines two distinct sets of operations: single-level operations that work at one level (e.g., Collision and Streaming) and multi-level operations that exchange information between two grids at different levels (e.g., Coalescence and Explosion). There are two main approaches for designing an LBM data structure [5]: a forest of octrees or a stack of uniform sparse grids. Each can support LBM single-level operations, with some *glue* to transition between different levels aimed at supporting multi-level operations. However, since the majority of the work occurs on the finest level, a stack of uniform sparse grids is expected to have superior performance compared to an octree structure. This is because most of the operations involved are stencil operations which may be handled better by a uniform grid rather than a tree traversal. Thus, we choose the sparse grid as the basis for our data structure.

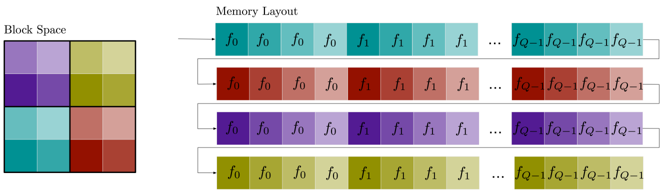

Block Space
Memory Layout
f_0 f_0 f_0 f_0 f_1 f_1 f_1 f_1 ... f_{Q-1} f_{Q-1} f_{Q-1} f_{Q-1}
f_0 f_0 f_0 f_0 f_1 f_1 f_1 f_1 ... f_{Q-1} f_{Q-1} f_{Q-1} f_{Q-1}
f_0 f_0 f_0 f_0 f_1 f_1 f_1 f_1 ... f_{Q-1} f_{Q-1} f_{Q-1} f_{Q-1}
f_0 f_0 f_0 f_0 f_1 f_1 f_1 f_1 ... f_{Q-1} f_{Q-1} f_{Q-1} f_{Q-1}

Fig. 5: A simplified example of our block-based memory layout for vector field data where each 2×2 block coincides with a 2×2 CUDA block

A. Block Sparse Data Structure

We partition the input domain into 3D blocks of fixed size. We place the blocks in active regions of the space associated with the fluid domain. For each block, we allocate a bitmask to track the active cells within the block. Additionally, we store the indices of all the neighboring blocks in all lattice directions (e.g., 26 directions for D3Q27) to be used for stencil operations. Given a cell, we can identify its intra-block and inter-block neighbor cells via division and modulo operations either using inexpensive bit operations (for intra-block neighbors) or explicitly (for inter-block neighbors). The block size B^3 is known at compile time which allows for various optimizations. To maximize locality, at runtime, a block is assigned to one CUDA block and every CUDA thread is assigned to a cell within the given block.

To represent vector fields over a grid, e.g., the 19-component vector field for D3Q19 populations, we adopt an Array of Structures of Arrays (AoSoA) (Figure 5) layout i.e., we store block's field data in a contiguous memory region, where data is then grouped by field's component. The memory layout and the mapping to CUDA blocks guarantee coalesced accesses and maximize memory locality within a block. Then, to improve the data locality between blocks, we arrange blocks in memory using space-filling curves (Sweep, Morton, or Hilbert).

B. Grid Refinement Data Structure

We implement our grid refinement data structure by stacking L_{max} block sparse data structures. We then extend the block data structure with information to allow access to neighbor

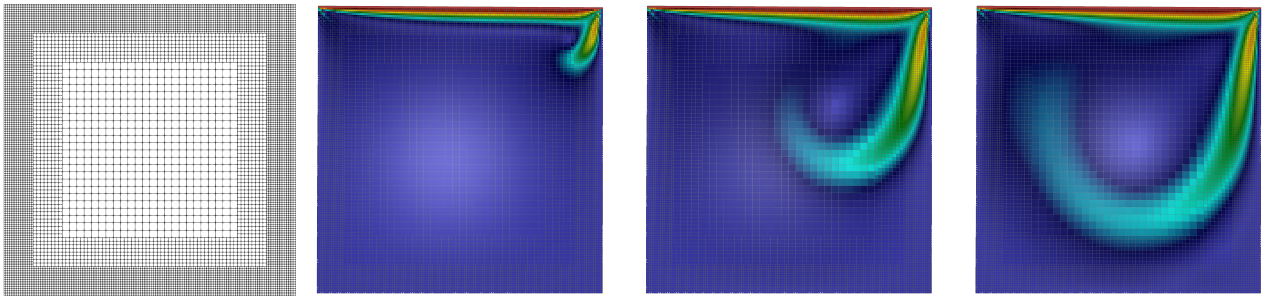

(a) Nonuniform Grid (b) Iteration 1000 (c) Iteration 5000 (d) Iteration 10000

Fig. 6: Snapshots of the lid-driven cavity flow at $Re = 100$ based on our nonuniform LBM implementation using the BGK collision model and D3Q19 lattice. The flow is incompressible, steady, and laminar characterized by a Reynolds number of $Re = u_{lid}/\nu = 100$ where ν is the kinematic viscosity of the fluid and the cubic box is assumed to have a dimensionless size of 1 in all directions. For this example, 3 levels of refinement with 240 voxels on the finest level (across all sides of the box) were used.

interface cells at a different resolution. For a grid at level L, we store the block index of an overlapping ghost block at level $L-1$. We also store the block index containing the fine cells only for the ghost cells on a non-leaf level. Octree-based organization of the baseline and optimized algorithm defines a 3D blocking of 2^3 because of its branching factor. Directly using such an organization for the memory may negatively impact the performance since 2^3 memory blocks provide low locality for stencil operations, and 2^3 CUDA blocks do not declare enough threads to fill up an entire CUDA warp. Thus, we decouple the branching factor from the memory layout. We do this by grouping a set of 2^3 *sub-blocks* into B^3-sized memory blocks as shown in Figure 5.

C. Programming Model

We implement our data structure in Neon [26]. Neon is a multi-GPU programming model that transforms a sequential user code into a multi-GPU efficient parallel implementation. We use Neon's capability of extracting the data dependency graph by analyzing the input and output fields of each kernel. This allows us to run independent kernels concurrently and to place synchronization points only when necessary leading to maximizing concurrency. For composing kernels, Neon exposes a for-each abstraction to define computations that manipulate fields' data. In Neon, the user only writes what happens to a single cell then Neon automatically launches and assigns CUDA blocks and threads to all cells.

Neon was originally designed for uniform grid computations with built-in dense, element sparse, and block sparse data structures. We extend Neon's capability to automatically construct a data dependency graph of multi-resolution applications thanks to a stack of uniform sparse grids. For multi-level kernels, the application defines at what level the kernel should run. Then, Neon launches the kernel on the specified grid level while relying on its capabilities for resolving data dependencies and assigning CUDA blocks and threads to active grid cells.

VI. Results

We now show the effectiveness of our optimization on a set of use cases. We analyze both the correctness and performance of our implementation. The LBM community relies on MLUPS (Million Lattice Updates Per Second) as the primary metric for performance. For a uniform grid, MLUPS is simply defined as VN/T where V is the number of voxels and T is the wall-clock time in microseconds to complete N LBM iterations. For nonuniform LBM, we calculate MLUPS as $\sum_{L=0}^{L_{max}-1} V_L N_L/T_L$, where L_{max} is the number of levels of the multi-resolution grid, V_L is the number of active voxels at level L (excluding any ghost cells), and T_L is the wall-clock time in microseconds to complete N_L iterations on level L i.e., $N_L = 2^L N$. We conduct all experiments on an A100 GPU with 40 GB of memory inside an NVIDIA DGX machine. Our implementation and comparisons use double precision unless stated otherwise. All tests are run in an NVIDIA docker container with CUDA 11.2.

We use the KBC model—compatible only with D3Q27 lattice—(Section II) in the experiments with turbulent flow simulation (e.g., wind tunnel). For the laminar (e.g., lid-driven cavity), we use the BGK model which can be employed with either D3Q19 or D3Q27. Since the results are identical, we reported those based on D3Q19.

A. Lid-driven Cavity

Validation: A canonical benchmark problem in CFD is the flow inside a lid-driven cavity. In this problem, fluid inside a cubic box is driven by the tangential movement of the box lid. The no-slip boundary condition on all the walls as well as the moving wall boundary condition on the top lid are all imposed by the halfway bounce-back method [27]. In order to capture the boundary layer near the walls accurately, we successively refine the voxels in all directions as they get closer to the boundaries. Figure 6 shows different iterations of our simulation based on a 3-level nonuniform grid. We validated our simulation against the accepted reference results of Ghia et al. [28] where we sampled the domain along x and y

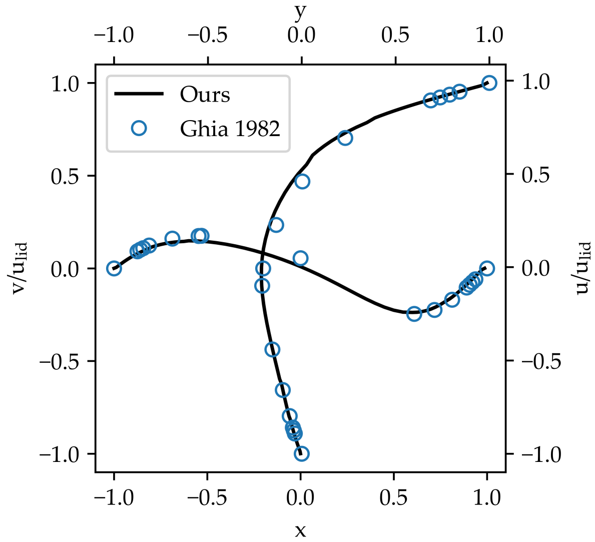


Fig. 7: Validating our nonuniform LBM implementation against reference results of Ghia et al. [28]. The normalized velocity components in x and y directions, namely u/u_{lid} and v/u_{lid} (where $\vec{u} = (u, v, w)$ and $\vec{u}_{lid} = (u_{lid}, 0, 0)$) are probed along the y and x axes respectively. The indicated distance is measured with respect to the coordinate axis located at the box center.

axes that run through the center of the domain. Figure 7 shows that our results are well-aligned with the reference data.

Comparison: As mentioned in Section I, there are no open-source and optimized GPU implementations of the grid refinement method in LBM. As such, our choice for comparison was limited to Palabos [29] and waLBerla [30]. Palabos is a multi-core CPU implementation of LBM targeting applications with complex coupled physics. We used their nonuniform LBM implementation to compare the same lid-driven cavity example against our implementation. Given the same domain size and refinement levels, Palabos took 2.3 seconds while our implementation took 0.015 seconds per iteration which is more than two orders of magnitude faster. Of course, a big factor of this speedup is due to differences between CPU and GPU hardware architectures where LBM is more amenable to massively parallel architectures like the GPU. For that, we compared also against waLBerla; an LBM block-structured implementation that relies on meta-programming techniques to generate highly efficient code for CPUs and GPUs. Again using the lid-driven cavity example, waLBerla performs $\mathcal{O}(10)$ MLUPS while our implementation is more than 2250 MLUPS i.e., more than two orders of magnitude speedup. We acknowledge that waLBerla's developers have only recently ported their grid refinement algorithm to GPU, and therefore it is not well-optimized. However, this highlights the importance of our careful optimizations and illustrates that merely porting CPU code to GPU is not enough to utilize the GPU hardware efficiently.

The comparison between grid refinement and uniform resolution may not be straightforward as the refinement of the grid is problem-dependent. For the lid-driven cavity and the level of refinement we selected, the time to solution for both well-optimized uniform [26] and grid refinement LBM is almost the same—grid refinement is only faster by 1.18X.

B. Wind Tunnel

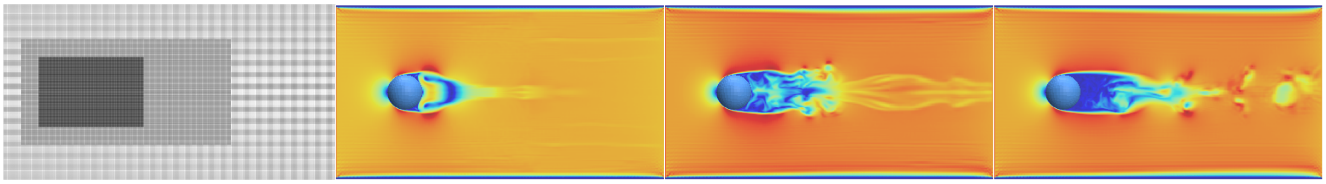
In order to showcase the benefits of employing nonuniform grid refinement in LBM and to further demonstrate our efficient implementation of this technique on a single GPU, we constructed a virtual wind tunnel experiment with two configurations.

Flow over sphere: We place a sphere inside a virtual wind tunnel with three levels of refinement around the sphere (Figure 8 and Table I) where the no-slip boundary condition is again imposed on the side walls and on the sphere using the halfway bounce-back method [27]. The inlet velocity associated with the incoming flow inside the wind tunnel is also prescribed using the same bounce-back technique. For the outflow boundary condition where the flow exits the domain, the missing directions of the distribution function f_i at the boundary are simply assigned the corresponding lattice weights w_i.

Table I shows the performance of both the baseline and our most optimized implementation. The baseline here corresponds to the algorithm in Figure 4b which has two of our own improvements compared to the original algorithm proposed in [16]. More precisely, in the *modified* baseline here (i) the ghost layer is allocated on the coarse level and (ii) the Accumulate communication is initiated from the coarse level. Table I clearly demonstrates the speedups gained through our optimizations. We note that as the computational domain becomes larger, the speedup decreases because a larger fraction of time is spent on the domain interior (e.g., to perform Collision) and less time is spent on the interface between levels (e.g., to perform Coalescence) and hence the overall impact of the computational operations that were targeted by our optimized implementation become relatively less pronounced. Nevertheless, our optimized implementation is able to sustain significantly better performance with up to 30% speedup.

In addition, we carry out an ablation study to show the impact of different fusion options shown schematically in Figure 4. Figure 9 shows that the fusion of Collide and Streaming operation associated with the finest resolution has the largest contribution to the speedup. This is because the voxels at the finest resolution consume more than half of the number of active voxels (Table I). We emphasize that such fusion is only possible thanks to the introduction of the Accumulate step which breaks the data dependency allowing the finest level to fuse the Collision and Streaming steps into one.

Flow over airplane: In this scenario, we demonstrate the impressive capabilities offered by our efficient implementations of the grid refinement LBM running on a single GPU. We simulate a virtual domain with dimensions of $1596 \times 840 \times 840$



(a) Nonuniform Grid

(b) Iteration 5k

(c) Iteration 10k

(d) Iteration 30k

Fig. 8: Evolution of flow over a sphere with radius R at $Re = u_{inlet}R/\nu = 4000$ using the KBC collision model and D3Q27 lattice.

Size	Distribution ($\times 10^6$)	Baseline (MLUPS)	Ours (MLUPS)	Speedup
$272 \times 192 \times 272$	0.602, 0.296, 0.175	482.63	1081.67	2.20
$544 \times 384 \times 544$	4.81, 2.37, 1.40	1115.80	1646.37	1.40
$816 \times 576 \times 816$	16.25, 8.0, 4.74	1299.7	1805.03	1.30

TABLE I: Comparing the performance of the *modified* baseline algorithm (Figure 4b) to our optimized implementation (Figure 4f) using the example of flow over sphere where three levels of refinement were used to arrive at the finest level around the sphere. *Size* indicates the length of the virtual wind tunnel box described in the finest level along x, y, and z directions; *Distribution* indicates the distribution of active voxels across different levels starting from the finest level, *Baseline* refers to the MLUPS achieved by our implementation of the modified baseline algorithm; *Ours* refers to our most optimized/fused implementation; and *Speedup* = Ours/Baseline.

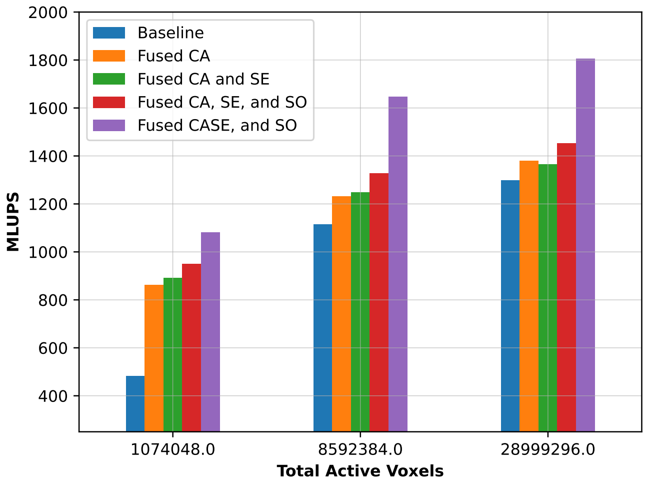


Fig. 9: The impact of different fusion configurations on the achieved MLUPS for the flow over sphere problem.

and place an aircraft model into this domain. We maintain consistent boundary conditions and employ the same collision model as in our previous case. In Figure 1, we depict the intricate turbulent flow patterns around the aircraft.

It is worth noting that with a uniform grid approach, the specified domain size of $1596 \times 840 \times 840$ would be unattainable due to limitations in accommodating such large grids within a single GPU. Even with advanced optimizations for uniform grids, such as the AA-method [7] that employs a single buffer, the largest feasible domain size on a single 40 GB GPU would be restricted to approximately $794 \times 794 \times 794$. This emphasizes the exceptional capabilities enabled by our grid refinement implementation that would otherwise be unachievable

VII. Conclusion and Future Work

Large-scale and accurate simulation of realistic turbulent flows is essential for many applications. In this work, we investigated the nonuniform grid refinement algorithm in the Lattice Boltzmann Method (LBM) and optimized its performance on a single GPU. The algorithm which was inherited from a CPU-focused implementation introduces complex data dependencies leading to inefficient implementation if ported to the GPU in its original form. We analyzed and optimized this complex data dependency. Our proposed algorithm, which highly leverages kernel fusion optimization and GPU atomic operations, features a lower memory footprint and reaches higher performance. Efficient nonuniform grid refinement on the GPU enables us to run large-scale problems on a single 40 GB GPU. Such large spatial scales open new avenues for tackling realistic engineering problems that were previously either impossible or only feasible on multi-GPU systems.

The foundation laid by our optimized single GPU algorithm positions us favorably for future research in extending this approach to multi-GPU frameworks, offering even higher scalability options for both speed and memory requirements. Additionally, we foresee promising research opportunities in Adaptive Mesh Refinement (AMR) for LBM, enabling dynamic grid resolution adjustments during runtime, and enhancing the flexibility of our simulations.

References

[1] J. H. Ferziger and M. Perić, *Computational Methods for Fluid Dynamics*, 3rd ed. Springer, 2019.

[2] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen, *The Lattice Boltzmann method*, 1st ed. Springer Cham, 2016.

[3] D. S. S. Corp., "3DEXPERIENCE SIMULIA," 2023. [Online]. Available: https://www.3ds.com

[4] A. E. Inc., "ultraFluidX," 2023. [Online]. Available: https://altair.com

[5] F. Schornbaum and U. Rüde, "Extreme-scale block-structured adaptive mesh refinement," *SIAM Journal on Scientific Computing*, vol. 40, pp. 358–387, 2018.

[6] M. Ataei and H. Salehipour, "XLB: A differentiable massively parallel lattice boltzmann library in python," *CoRR*, vol. abs/2311.16080, no. 2311.16080, Dec. 2023.

[7] P. Bailey, J. Myre, S. D. Walsh, D. J. Lilja, and M. O. Saar, "Accelerating lattice boltzmann fluid flow simulations using graphics processors," in *2009 international conference on parallel processing*. IEEE, Sep. 2009, pp. 550–557.

[8] M. Geier and M. Schönherr, "Esoteric Twist: An efficient in-place streaming algorithmus for the lattice boltzmann method on massively parallel hardware," *Computation*, vol. 5, no. 2, p. 19, Mar. 2017.

[9] M. Lehmann, M. J. Krause, G. Amati, M. Sega, J. Harting, and S. Gekle, "Accuracy and performance of the lattice boltzmann method with 64-bit, 32-bit, and customized 16-bit number formats," *Physical Review E*, vol. 106, no. 1, Jul. 2022.

[10] J. Latt, C. Coreixas, and J. Beny, "Cross-platform programming model for many-core lattice Boltzmann simulations," *PLOS ONE*, vol. 16, no. 4, pp. 1–29, Apr. 2021.

[11] M. Schönherr, K. Kucher, M. Geier, M. Stiebler, S. Freudiger, and M. Krafczyk, "Multi-thread implementations of the lattice Boltzmann method on non-uniform grids for CPUs and GPUs," *Computers & Mathematics with Applications*, vol. 61, pp. 3730–3743, Jun. 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0898122111002999

[12] P. Valero-Lara and J. Jansson, "Heterogeneous CPU+GPU approaches for mesh refinement over Lattice-Boltzmann simulations," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 7, Apr. 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3919

[13] C.-M. Wu, Y.-S. Zhou, and C.-A. Lin, "Direct numerical simulations of turbulent channel flows with mesh-refinement lattice boltzmann methods on GPU cluster," *Computers & Fluids*, vol. 210, p. 104647, Oct. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045793020302188

[14] C. A., Niedermeier, C. F. Janssen, and T. Indinger, "Massively-parallel multi-gpu simulations for fast and accurate automotive aerodynamics," in *Proceedings of the 7th European Conference on Computational Fluid Dynamics*, ser. ECFD 7, Jun. 2018.

[15] S. Watanabe and T. Aoki, "Large-scale flow simulations using lattice Boltzmann method with AMR following free-surface on multiple GPUs," *Computer Physics Communications*, vol. 264, Jul. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465521000291

[16] F. Schornbaum and U. Rüde, "Massively parallel algorithms for the lattice boltzmann method on nonuniform grids," *SIAM Journal on Scientific Computing*, vol. 38, pp. C96–C126, 2016.

[17] Z. Liu, J. Ruan, W. Song, L. Zhou, W. Guo, and J. Xu, "Parallel scheme for multi-layer refinement non-uniform grid lattice boltzmann method based on load balancing," *Energies*, vol. 15, no. 21, 2022. [Online]. Available: https://www.mdpi.com/1996-1073/15/21/7884

[18] I. V. Karlin, F. Bösch, and S. Chikatamarla, "Gibbs' principle for the lattice-kinetic theory of fluid dynamics," *Physical Review E*, vol. 90, no. 3, Sep. 2014.

[19] D. Lagrava, O. Malaspinas, J. Latt, and B. Chopard, "Advances in multi-domain lattice Boltzmann grid refinement," *Journal of Computational Physics*, vol. 231, pp. 4808–4822, May 2012.

[20] A. Dupuis and B. Chopard, "Theory and applications of an alternative lattice Boltzmann grid refinement algorithm," *Physical Review E*, vol. 67, Jun. 2003. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.67.066707

[21] B. Dorschner, N. Frapolli, S. S. Chikatamarla, and I. V. Karlin, "Grid refinement for entropic lattice Boltzmann models," *Physical Review E*, vol. 94, Nov. 2016. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.94.053311

[22] O. Filippova and D. Hänel, "Grid refinement for lattice-BGK models," *Journal of Computational Physics*, vol. 147, pp. 219–228, Nov. 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999198960892

[23] H. Chen, O. Filippova, J. Hoch, K. Molvig, R. Shock, C. Teixeira, and R. Zhang, "Grid refinement in lattice Boltzmann methods based on volumetric formulation," *Physica A: Statistical Mechanics and its Applications*, vol. 362, pp. 158–167, Mar. 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378437105009696

[24] M. Hasert, K. Masilamani, S. Zimny, H. Klimach, J. Qi, J. Bernsdorf, and S. Roller, "Complex fluid simulations with the parallel tree-based Lattice Boltzmann solver Musubi," *Journal of Computational Science*, vol. 5, no. 5, pp. 784–794, Sep. 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750313001270

[25] M. Rohde, D. Kandhai, J. J. Derksen, and H. E. A. van den Akker, "A generic, mass conservative local grid refinement technique for lattice-boltzmann schemes," *International Journal for Numerical Methods in Fluids*, vol. 51, pp. 439–468, 2006. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.1140

[26] M. Meneghin, A. H. Mahmoud, P. K. Jayaraman, and N. J. W. Morris, "Neon: A multi-gpu programming model for grid-based computations," in *Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium*, Jun. 2022, pp. 817–827. [Online]. Available: https://escholarship.org/uc/item/9fz7k633

[27] A. J. Ladd, "Numerical simulations of particulate suspensions via a discretized boltzmann equation. part 1. theoretical foundation," *Journal of fluid mechanics*, vol. 271, pp. 285–309, 1994.

[28] U. Ghia, K. Ghia, and C. Shin, "High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method," *Journal of Computational Physics*, vol. 48, pp. 387–411, Dec. 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0021999182900584

[29] 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, and B. Chopard, "Palabos: Parallel lattice boltzmann solver," *Computers & Mathematics with Applications*, vol. 81, pp. 334–350, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0898122120301267

[30] M. Bauer, S. Eibl, C. Godenschwager, N. Kohl, M. Kuron, C. Rettinger, F. Schornbaum, C. Schwarzmeier, D. Thönnes, H. Köstler, and U. Rüde, "waLBerla: A block-structured high-performance framework for multiphysics simulations," *Computers & Mathematics with Applications*, vol. 81, pp. 478–501, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0898122120300146