Acceleration of Wind Simulation Using Locally Mesh-Refined Lattice Boltzmann Method on GPU-Rich Supercomputers

Naoyuki Onodera(✉) and Yasuhiro Idomura

Japan Atomic Energy Agency, Chiba, Japan
onodera.naoyuki@jaea.go.jp

Abstract. A real-time simulation of the environmental dynamics of radioactive substances is very important from the viewpoint of nuclear security. Since airflows in large cities are turbulent with Reynolds numbers of several million, large-scale CFD simulations are needed. We developed a CFD code based on the adaptive mesh-refined Lattice Boltzmann Method (AMR-LBM). AMR method arranges fine grids in a necessary region, so that we can realize a high-resolution analysis including a global simulation area. The code is developed on the GPU-rich supercomputer TSUBAME3.0 at the Tokyo Tech, and the GPU kernel functions are tuned to achieve high performance on the Pascal GPU architecture. The code is validated against a wind tunnel experiment which was released from the National Institute of Advanced Industrial Science and Technology in Japan Thanks to the AMR method, the total number of grid points is reduced to less than 10% compared to the fine uniform grid system. The performances of weak scaling from 1 nodes to 36 nodes are examined. The GPUs (NVIDIA TESLA P100) achieved more than 10 times higher node performance than that of CPUs (Broadwell).

Keywords: High performance computing · GPU · Lattice boltzmann method
Adaptive mesh refinement · Real-time wind simulation

1 Introduction

A real-time simulation of the environmental dynamics of radioactive substances is very important from the viewpoint of nuclear security. In particular, high resolution analysis is required for resident areas or urban cities, where the concentration of buildings makes the air flow turbulent. In order to understand the details of the air flow there, it is necessary to carry out large-scale Computational Fluid Dynamics (CFD) simulations. Since air flows behave as almost incompressible fluids, CFD simulations based on an incompressible Navier-Stokes equation are widely developed. The LOcal-scale High-resolution atmospheric DIspersion Model using Large-Eddy Simulation (LOHDIM-LES [1]) has been developed in Japan atomic energy agency (JAEA). The LOHDIM-LES can solve turbulent wind simulation with Reynolds numbers of several million. However, an incompressible formulation sets the speed of sound to infinity, and thus, the pressure Poisson equation has to be solved iteratively with sparse matrix solvers. In such large-scale problems, it is rather difficult for sparse matrix solvers to converge efficiently because the problem becomes ill-conditioned with increasing the problem size and the overhead of node-to-node inter-communication increases with the number of nodes.

The Lattice Boltzmann Method (LBM) [2–5] is a class of CFD method that solves the discrete-velocity Boltzmann equation. Since the LBM is based on a weak compressible formulation, the time integration is explicit and we do not need to solve the pressure Poisson equation. This makes the LBM scalable, and thus, suitable for large-scale computation. As an example, researches performing large-scale calculation using the LBM were nominated for the Gordon Bell prize in SC10 [6] and SC15 [7]. However, it is difficult to calculate multi-scale analysis with a uniform grid from the viewpoint of computational resources and calculation time. In this work, we address this issue based on two approaches, one is the development of an adaptive mesh refinement (AMR) method for the LBM, and the other is optimization of the AMR-LBM on the latest Pascal GPU architecture.

The AMR method was proposed to overcome this kind of problem [8, 9]. Since the AMR method arranges fine grids only in a necessary region, we can realize a high-resolution multi-scale analysis covering global simulation areas. AMR algorithms for the LBM have been proposed, and they have achieved successful results [10, 11].

Recently, GPU based simulations have been emerging as an effective technique to accelerate many important classes of scientific applications including CFD applications [12–14]. Studies on LBM have also been reported on implementation of GPU [15, 16]. Since there are not many examples of AMR-based applications on the latest GPU architectures, there is a room for research and development of such advanced applications. In this work, we implement an AMR-based LBM code to solve multi-scale air flows. The code is developed on the GPU-rich supercomputer TSUBAME3.0 at the Tokyo Institute of Technology, and the GPU kernel functions are tuned to realize a real-time simulation of the environmental dynamics of radioactive substances.

This paper reports implementation strategies of the AMR-LBM on the latest Pascal GPU architectures and its performance results. The code is written in CUDA 8.0 and CUDA-aware MPI. The Host/Device memory is managed by using Unified memory, and the GPU/CPU buffers are directly passed to a MPI function. We demonstrate the performance of both CPU and GPU on the TSUBAME3.0. A single GPU process (a single NVIDIA TESLA P100 processor) achieves 383.3 mega-lattice update per second (MLUPS) when leaf size equals to 4^3 in single precision. The performance is about 16 times higher than that of a single CPU process (two Broadwell-EP processors, 14×2 cores, 2.4 GHz). Regarding the weak scalability results, the AMR-LBM code achieves 22535 MLUPS using 36 GPU nodes, which is 85% efficiency compared with the performance on a single GPU node.

2 Lattice Boltzmann Method

The LBM solves the discrete Boltzmann equation to simulate the flow of a weakly compressible fluid. The flow field is expressed by a limited number of pseudo particles, which evolve through streaming and collision processes. The configuration space is discretized by uniform grids. Since pseudo particles move onto the neighbor lattice points after one time step in the streaming process, this process is completed without any error. The macroscopic diffusion and the pressure gradient are expressed by the local collisional process. The time evolution of the discretized velocity function is

$$f_i(x + c_i\Delta t, t + \Delta t) = f_i(x,t) + \Omega_i(x,t). \tag{1}$$

Here, Δt is the time interval, c_i is the lattice vectors of pseudo particles, and Ω_i is the collision operator.

It is important to choose a proper lattice velocity (vector) model by taking account of the tradeoff between efficiency and accuracy. Since their low computational cost and high efficiency, the D3Q15 and D3Q19 models are popular. Recently, it was pointed out that these velocity models do not have enough accuracy at high Reynolds number with complex geometries [17]. On the other hand, the D3Q27 model is suitable model for a weakly compressible flow at high Reynolds number.

Figure 1 shows schematic figures of the above velocity vector models. Since airflows in urban cities are turbulent with high Reynolds number, we adapt the D3Q27 model. The components of the velocity vector are defined as

$$c_i = \begin{cases} (0,0,0) & i = 0 \\ (\pm c,0,0),(0,\pm c,0),(0,0,\pm c) & i = 1-6 \\ (\pm c,\pm c,0),(0,\pm c,\pm c),(\pm c,0,\pm c) & i = 7-18 \\ (\pm c,\pm c,\pm c) & i = 19-26 \end{cases} \tag{2}$$

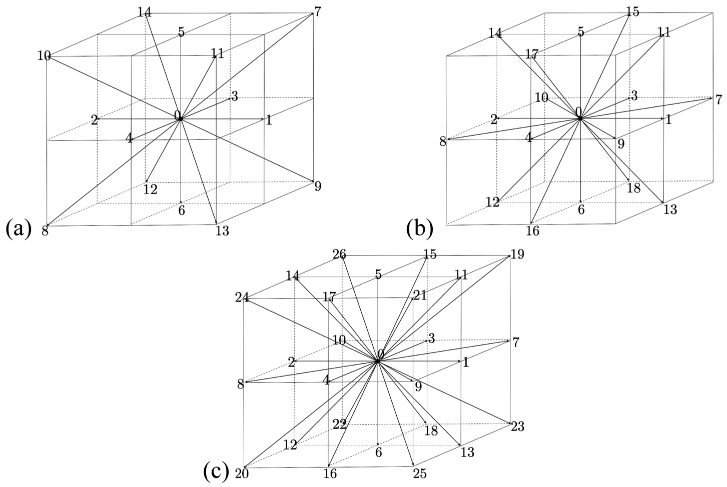


Fig. 1. Components of the velocity vector of (a) D3Q15, (b) D3Q19, and (c) D3Q27 models.

Here, c is sound speed, and is normalized as $c = 1$. Each velocity refers the predetermined upwind quantity. Since memory accesses are simple and continuous, the streaming process is suitable for high performance computing.

2.1 Single Relaxation Time Model

The macroscopic diffusion and the pressure gradient are expressed by the collisional process. The lattice BGK model [18] is widely used in most of the previous studies because of its simplicity. A collision operator of a single relaxation time (SRT) model are defined as

$$\Omega_{i(x,t)} = -\frac{1}{\tau}\left(f_i(x,t) - f_i^{eq}(x,t)\right), \tag{3}$$

where τ is relaxation time, and f_i^{eq} is a local equilibrium distribution function. The relaxation time in the collisional process is determined using the dynamic viscosity and the sound speed

$$\tau = \frac{1}{2} + \frac{3\nu}{c^2\Delta t}. \tag{4}$$

In this wind simulation, since the Mach number is less than 0.3, the flow can be regarded as incompressible. The equilibrium distribution function f_i^{eq} of incompressible model is given as

$$f_i^{eq}(x,t) = \omega_i\left(1 + \frac{3c_i \cdot \vec{u}}{c^2} + \frac{9(c_i \cdot \vec{u})^2}{2c^4} - \frac{3\vec{u}^2}{2c^2}\right). \tag{5}$$

Here, ρ is the density and $\vec{u}$ is the macroscopic velocity vector. The collision operator is equivalent to the viscous term in the Navier-Stokes equation. The corresponding weighting factors of the D3Q27 model are given by

$$\omega_i = \begin{cases} 64/216 & i = 0 \\ 16/216 & i = 1-6 \\ 4/216 & i = 7-18 \\ 1/216 & i = 19-26 \end{cases}. \tag{6}$$

Since the SRT model is unstable at high Reynolds number, a Large-Eddy Simulation (LES) model has to be used to solve the LBM equation. The dynamic Smagorinsky model [19, 20] is often used, but it requires an averaging process over a wide area to determine the model constant. This is a huge overhead for large-scale computations, and it will negate the simplicity of the SRT model.

2.2 Cumulant Relaxation Time Model

The cumulant relaxation time model [21, 22] is a promising approach to solve the above problems. This model realizes turbulent simulation without LES model, and we can determine the equilibrium distribution function locally. Unlike the SRT model, the collisional process is not determined in the momentum space. We redefine physical quantities in the following. We take the two-sided Laplace transform of distribution function as

$$F\left(\vec{\Xi}\right) = \mathcal{L}\left\{f\left(\vec{\xi}\right)\right\} = \int_{-\infty}^{\infty} f\left(\vec{\xi}\right) e^{-\vec{\Xi}\cdot\overrightarrow{\xi}} d\vec{\xi}. \tag{7}$$

Here, $\vec{\Xi}$ is the velocity frequency variable. $\vec{\xi} = (\xi, \upsilon, \zeta)$ are the microscopic velocities. The coefficients of the series as countable cumulants $c_{\alpha\beta\gamma}$ are written as

$$c_{\alpha\beta\gamma} = c^{-\alpha-\beta-\gamma} \frac{\partial^\alpha \partial^\beta \partial^\gamma}{\partial^\alpha \Xi \partial^\beta \Upsilon \partial^\gamma Z} \ln(F(\Xi, \Upsilon, Z)). \tag{8}$$

Here, the subscripts α, β, and γ are indices of the cumulant. All decay processes are computed by

$$c^*_{\alpha\beta\gamma} = c^{eq}_{\alpha\beta\gamma} + (1 - \omega_{\alpha\beta\gamma}) c_{\alpha\beta\gamma}. \tag{9}$$

The asterisk $*$ is the post collision cumulant, and $\omega_{\alpha\beta\gamma}$ is the relaxation frequency. The Maxwellian equilibrium is expressed as a finite Taylor expansion.

$$\ln(F^{eq}(\Xi, \Upsilon, Z)) = \ln\left(\frac{\rho}{\rho_0}\right) - \Xi u - \Upsilon v - Zw + \frac{c^2\theta}{2}\left(\Xi^2 + \Upsilon^2 + Z^2\right)). \tag{10}$$

The velocities u, v, and w are the components of macroscopic velocity vector $\vec{u}$, and θ is a parameter. Cumulants are calculated by using local quantities as discretized velocity function f_i and macroscopic velocities $\vec{u}$. Since this model is a computationally intensive algorithm with local memory access, it should be well suited to achieve high efficiency for GPU computing.

2.3 Boundary Treatment

The LBM is suitable for modeling boundary conditions with complex shapes. The bounce-back (BB) scheme and the interpolated bounce-back (IBB) scheme make it easy to implement the no-slip velocity condition. Immersed boundary methods (IBM) [23, 24] are also able to handle complex boundary conditions by adding external forces in the LBM.

In this work, we applied the IBB scheme [25, 26] because of their flexibility and compute efficiency. Figure 2 shows schematic figures of the IBB scheme. The IBB scheme directly applies the following conditions to the velocity distribution function depending on a distance function Δ

$$f^*_{i,(-1)}(x,t) = \begin{cases} 2\Delta f_{i,(+1)}(x,t) + (1-2\Delta) f_{i,(+1)}(x - c_i\Delta t, t) + F_{i,(-1)} & \Delta < \frac{1}{2} \\ \frac{1}{2\Delta} f_{i,(+1)}(x,t) + \frac{(2\Delta-1)}{2\Delta} f_{i,(-1)}(x,t) + \frac{1}{2\Delta} F_{i,(-1)} & \Delta \geq \frac{1}{2} \end{cases}, \tag{11}$$

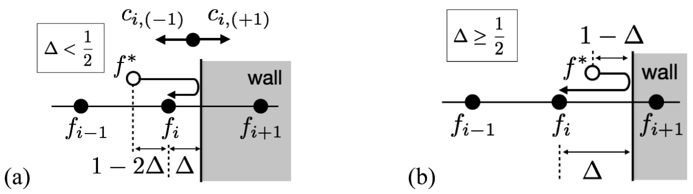


Fig. 2. Interpolated bounce-back boundary conditions of (a) $\Delta < \frac{1}{2}$ and (b) $\Delta \geq \frac{1}{2}$. The velocity distribution function f^* is computed by a linear interpolation in the upwind cell.

where subscript (± 1) is the direction of each velocity component, and F_i is force on the solid boundary given as

$$F_{i,(-1)} = -3\omega_i \rho \frac{c_i \cdot \overrightarrow{u_b}}{c^2}. \tag{12}$$

Here $\overrightarrow{u_b}$ is a velocity vector of the boundary. Since each velocity function refers the predetermined neighbor upwind and downwind quantities, it is more suitable for high performance computing than the IBM [23, 24].

3 Adaptive Mesh Refinement (AMR) Method

3.1 Block-Structured AMR Method

Since a lot of buildings and complex structures make the air flow turbulent in large urban areas, it is necessary to carry out multi-scale CFD simulations. However, it is difficult to perform such a multi-scale analysis with uniform grids from the viewpoint of computational resources and calculation time. The AMR method [8, 27] is a grid generation method, which can arrange high-resolution grids only in a necessary region. In the AMR methods based on a forest-of-octrees approach [16, 28], one domain named a leaf is subdivided into four leaves in two dimensions (quadtree) and eight leaves in three dimensions (octree). Since the leaf is recursively subdivided into half, it is easy to implement the algorithm for parallel computing, and the same number of leaves are assigned to each process.

The block-structured AMR method [29, 30] is an efficient method suitable for multithread computation. Since a leaf contains N^3 grid points and these memory accesses are continuous, it is suitable for GPU computation. Figure 3(a) shows a schematic figure of computational leaves at the interface of leaves at different levels, where each level needs the halo region across the interface. In such halo leaves, data is constructed from data on another level. Figure 3(b) shows an example of the leaf arrangement in 2D case, where the calculation region at each level is surrounded by the halo region, which is constructed from the data on leaves at the next level. Therefore, only one level difference is allowed at the interface of leaves at different levels.

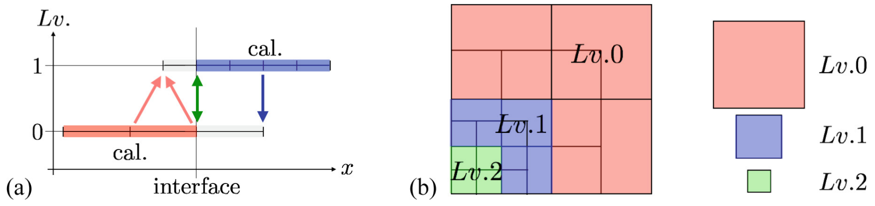


Fig. 3. Schematic figures of computational leaves: (a) Interpolating operations of (red) linear interpolation, (green) exchange values between coarse and fine grids, (blue) copy values from fine to coarse grid in 1D case. (b) An example of leaf arrangement in 2D case. Calculation region is surrounded by the halo (boundary) region of the same refined level. (Color figure online)

The AMR method is applied to resolve the boundary layer near the buildings. The octree is initialized at the beginning of the simulation and does not dynamically change the mesh during the time step.

3.2 LBM with AMR

The LBM is a dimensionless method in time and space. It is necessary to arrange these parameters according to the resolution of AMR grids [5]. The kinematic viscosity, defined in the LBM, depends on the time step size with

$$\nu = \frac{1}{3}\left(\tau - \frac{1}{2}\right)c^2\Delta t \tag{13}$$

To keep a constant viscosity on coarse and fine grids, the relaxation time τ satisfies the following expression

$$\left(\tau_f - \frac{1}{2}\right) = m\left(\tau_c - \frac{1}{2}\right). \tag{14}$$

Here the super- and sub-scripts c and f denote the value of the coarse and fine grids, respectively. The coefficient m is the refinement factor. The time step is also redefined for each resolution as $\Delta t_f = (\Delta t_c)/m$. To take account of the continuity of hydrodynamic variables and their derivatives on the interface between two resolutions, the distribution functions satisfy the following equations

$$f_i^c = f_i^{eq,f} + m\frac{\tau_c - 1}{\tau_f - 1}\left(f_i^f - f_i^{eq,f}\right), \tag{15}$$

$$f_i^f = f_i^{eq,c} + \frac{1}{m}\frac{\tau_f - 1}{\tau_c - 1}\left(f_i^c - f_i^{eq,c}\right). \tag{16}$$

The refinement factor m is set to 2 for stability and simplicity reasons.

Figure 4 illustrates the flowchart of the computational procedure on coarse grid and fine grid. At first, streaming and collision terms are calculated on each grid. Before the fine grid calculation starts at time $t + \Delta t/2$, boundary values around the fine grid are interpolated from the coarse grid. MPI communications are executed after the computational procedures ① and ③. Temporal and spatial interpolations in halo region are executed at ② and ④.

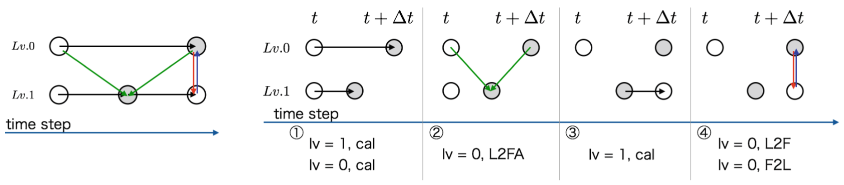


Fig. 4. Flowchart of the computational procedure on coarse grid $(Lv.0)$ and fine grid: $(Lv.1)$ ① Streaming and collision on each grid, ② time and space interpolation, ③ streaming and collision on fine grid, and ④ space interpolation on each level. Processes ② and ④ are executed in halo region.

4 Implementation and Optimization

4.1 CPU and GPU Implementation

In this section, we describe implementation of wind simulation code. The code is written in CUDA 8.0. We adopted the Array of Structures (AoS) memory layout to optimize multi-threaded performance. Each array is allocated by using the CUDA runtime API "cudaMallocManaged" which defines CPU and GPU memory space in the same address space. The CUDA system software automatically migrates data between CPU and GPU, so that it keeps the portability.

Figure 5 shows pseudocodes for stencil computation on CPU and GPU. The calculation code consists of a calling function (Fig. 5 top), loop functions (Fig. 5 middle), and a kernel function (Fig. 5 bottom). The calling function and the kernel function are shared by CPU and GPU. The loop functions generate indices for multi-threaded computation. CUDA threads are assigned to grid points in the leaf, and thread blocks are assigned to leaves.

The code is parallelized by the MPI library. OpenMPI 2.1.1 is CUDA-aware MPI that enables to send and receive CUDA device memory directly. OpenMPI 2.1.1 also supports Unified Memory, and the GPU/CPU buffers can be directly passed to a MPI function. MPI communications are executed in each leaf unit, and the leaf unit is transferred by one-sided communication of "MPI_Put" function implemented by MPI-2.

```
void launch_func() {
#ifdef CPU_CAL_
  func_cpu(num_leaves, nx_leaf, arguments);
#elif  GPU_CAL_
  dim3 grid(thread_x_max, thread_y_max, thread_z_max);/*less than nx_leaf*/
  dim3 block(num_leaves, 1, 1);

  func_gpu <<<grid, block>>> (num_leaves, nx_leaf, arguments);
  cudaDeviceSynchronize();
#endif
}
```

```
void  func_cpu(num_leaves,  nx_leaf,  argu-
ments) {
#pragma omp parallel for collapse(4)
  for (int l=0; l<num_leaves; l++) {
    for (int k=0; k<nx_leaf; k++) {
    for (int j=0; j<nx_leaf; j++) {
    for (int i=0; i<nx_leaf; i++) {
      kernel(i, j, k, l, arguments);
    }
    }
    }
}
```

```
__global__    void    func_gpu(num_leaves,
nx_leaf, arguments) {
  int l  = blockIdx.x;
  int mx = nx_leaf / blockDim.x;
  int my = nx_leaf / blockDim.y;
  int mz = nx_leaf / blockDim.z;
  for (int _k=0; _k<mz; _k++) {
  for (int _j=0; _j<my; _j++) {
  for (int _i=0; _i<mx; _i++) {
    int i = threadIdx.x + blockDim.x*_i;
    int j = threadIdx.y + blockDim.y*_j;
    int k = threadIdx.z + blockDim.z*_k;

    kernel(i, j, k, l, arguments);
  }
  }
  }
}
```

```
inline __host__ __device__  void kernel(i, j, k, l, arguments) {
  /* do something on CPU or GPU */
}
```

Fig. 5. Pseudocodes for stencil computation as (top) function to call CPU or GPU instruction, (middle left) function executed on the CPU, (middle right) function executed on the GPU, and (bottom) common function of both CPU and GPU.

4.2 Optimization for GPU Computation

In our GPU implementation, the streaming and collision processes are fused to reduce global memory accesses. In order to achieve high performance, it is also necessary to use thousands of cores in GPUs. The upper limit of the number of threads is limited by the usage of registers per streaming multiprocessor (SM), and it is determined at compile time. For example, according to the GP100 Pa whitepaper of NVIDIA [32], the Pascal GP100 provides 65536 32-bit registers on each SM. If one thread requires 128 registers, only 512 threads are executed on SM simultaneously. On the other hand, if one thread requires 32 registers, 2048 threads are executed and that is the upper limit of the Pascal GP100. Since the D3Q27 model and its cumulant collision operator need a lot of register memories on GPUs, the number of threads executed is limited by the lack of registers.

As a simple solution to reduce the amount of registers, it is effective to create a kernel function for each conditional branch. The main conditional branch of the streaming and collision function is the boundary condition on the object. The IBB scheme (Eq. (13)) requires a level-set function and velocity vector of boundary, and this branch requests more memory read/write and registers. In this research, since the boundary objects are fixed, optimal kernel functions are created at the beginning of calculation. We show the PTX information generated by NVIDIA CUDA Compiler 8.0.61 in single precision.

```
 ・Func1: stream_collision_without_boundary_condtion
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info: Used 88 registers, 108 bytes smem, 364 bytes cmem[0],
260 bytes cmem[2]
 ・Func2: stream_collision_with_boundary_condition
328 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info: Used 93 registers, 108 bytes smem, 396 bytes cmem[0],
264 bytes cmem[2]
```

As described above, the function without boundary conditions (Func1) can reduce the number of registers compared to the original function (Func2). By executing two functions asynchronously, it is possible to use more threads than the original calculation. Details of computational performance are discussed in Sect. 6.1 below.

5 Numerical Verification and Validation

5.1 Lid-Driven Cavity Flow

The validity of the adopted local grid refinement was verified by simulating the classical problem of lid-driven cavity flow in two-dimensions [33]. The computational domain is surrounded by walls, and its top boundary wall moves in the horizontal direction (left to right). Table 1 shows the discretization parameters. The whole computational domain is divided into 8 $\times$ 8 sub-domains. The coarse-resolution leaves are located in 6 $\times$ 6 sub-domains of the center part, and the middle-resolution leaves are located around coarse-resolution leaves, and the fine-resolution leaves are located near the walls. Each leaf contains 8 $\times$ 8 grid points. The total number of grid points in 2D-surface is 20992. It is equivalent to 32% grid points compared to the finest uniform grids in the whole domain.

Table 1. Discretization parameters for 2D lid-driven cavity flow.

AMR lv.	Δleaf	Δx	# of leaves	# of grid points
0	L/8	L/64	$36 = 6^2$	2304
1	L/16	L/128	$52 = 14^2 - 12^2$	3328
2	L/32	L/256	$240 = 32^2 - 28^2$	15360
Total	–	–	328	20992

Figure 6 shows velocity profiles of velocities along a vertical line and a horizontal line passing through the center of the cavity at (a) Re = 1000, (b) Re = 3200, (c) Re = 5000, and (d) Re = 10000. Calculation results are in good agreement with the reference results. If we used the SRT model, calculation was diverged at a high Reynolds number such as 3200. We conclude that our simulation is robust against high Reynolds number, and physical phenomena can be reproduced with few grid points.

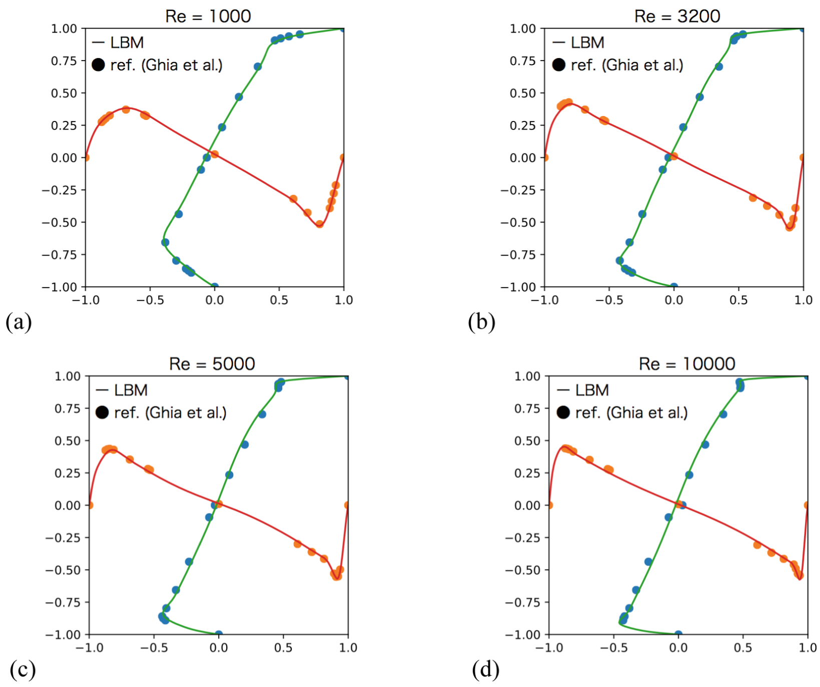


Fig. 6. Velocity profiles of u along a vertical line (green solid line) and v along a horizontal line (orange solid line) passing through the center of the cavity at (a) Re = 1000, (b) Re = 3200, (c) Re = 5000, and (d) Re = 10000. Each axis is normalized by the half-length of computational domain and the velocity of the moving wall. (Color figure online)

5.2 Wind Tunnel Test

The code is validated against a wind tunnel test, which was released from the National Institute of Advanced Industrial Science and Technology (AIST) in Japan [34]. Figure 7 shows schematic figures of a wind tunnel test. A cube is placed on the center of the floor. Inflow and outflow boundary conditions are applied in the streamwise direction. Periodic boundary conditions are assumed in the spanwise direction. A non-slip condition is imposed on the ground, and a moving boundary condition is given on the top in the vertical direction. The inlet velocity is set to be

$$u(z) = u_s \left(\frac{z}{z_s} \right)^{\frac{1}{7}}, \tag{17}$$

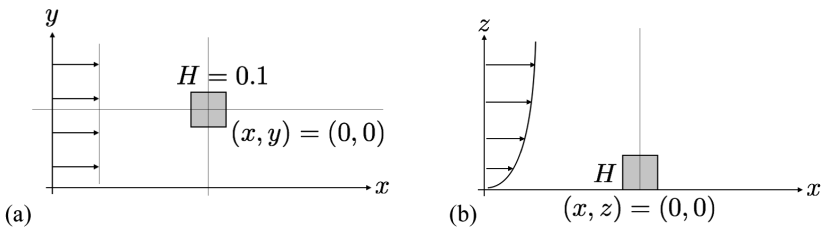


Fig. 7. Schematic figures of the wind tunnel test: (a) top view and (b) side view. A cube is placed on the center of the floor.

where the ground roughness is z_s = 0.5 m and wind velocity coefficient is u_s = 2.14 m/s. The Reynolds number, which is evaluated from the inlet velocity and physical properties of the air, is about 14000 at the top of the cube (z = 0.1 m).

Table 2 shows the discretization parameters for wind tunnel test. The computational domain size is from (−19.2, −1.2, −0.2) to (19.2, 1.2, 2.2) corresponding to the streamwise, spanwize and vertical direction, respectively. The bottom boundary condition is given at z = 0.0, and the top boundary condition is given at z = 2.0.

Table 2. Discretization parameters for wind tunnel test.

AMR lv.	Δx$(H = 0.1m)$	Domain size $(X_{min,max}/Y_{min,max}/Z_{min,max})$	# of leaves	# of grid points $(\times 10^6)$
0	$H/4$	−1.5, 1.5/−0.5, 0.5/−0.2, 0.75	24048	12.31
1	$H/8$	−4.0, 4.0/−1.0, 1.0/−0.2, 1.5	25800	13.21
2	$H/16$	−19.2, 19.2/−1.2, 1.2/−0.2, 2.2	24000	12.29
Total	–	–	73848	37.81

We compute a simulation with three refinement levels. Fine-resolution leaves are located near the cube, and middle-resolution leaves are surrounding the fine-resolution leaves, and coarse-resolution leaves are used in the outer region. The total number of grid points is 3.78×10^7, which corresponds to 4.2% compared to the finest uniform grids in the whole domain.

Figure 8 shows mean velocity profiles in the stream wise direction. Red solid lines show calculation results and blue dots show experimental data. Figure 8(a) shows mean velocity profiles horizontal plane at the center of the cube (z = H/2). Calculation results are smooth around a cube and in good agreement with the reference results. Figure 8(b) shows mean velocity profiles in vertical plane at the center of the cube (y = 0). The flow behind the cube is captured well, and calculation results are also in good agreement with the reference results. We conclude that our simulation can reproduce the wind tunnel experiment with an optimal number of grid points.

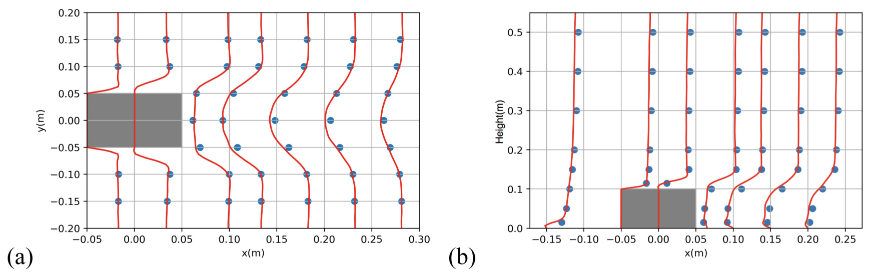


Fig. 8. Mean velocity profiles (m/s) in stream wise direction: (a) in horizontal plane at the center of the cube (z = 1/2H), and (b) in vertical plane at the center of the cube (y = 0). Red solid lines show calculation results and blue dots show experiment data as $u_{plot} = 0.02u_{mean} + x_{line}$. Simulation and experiment data have been measured along the lines: $x_{line} = (-50, 0, 65, 100, 150, 200, 250mm)$. (Color figure online)

6 Performance on the TSUBAME 3 Supercomputer

The TSUBAME 3.0 supercomputer at the Tokyo Institute of Technology is equipped with more than 2,000 GPUs (NVIDIA TESLA P100). The peak performance is 12.15/24.3 PFLOPS in double/single precision, respectively, and has achieved 8.125 PFLOPS on the Linpack benchmark. Table 3 shows the specification of TSUBAME 3.0. A compute node consists of two Intel Xeon E5-2680 V4 Processor (Broadwell-EP, 14 cores, 2.4 GHz) and four NVIDIA TESLA P100 processors. We measured the performance of our LBM code on TSUBAME 3.0.

Table 3. TSUBAME 3.0 specification of a node.

	Architecture	Bandwidth/node (GB/s)
CPU	Intel Xeon E5-2680 V4 (14 cores) $\times$ 2	153.6 (76.8 $\times$ 2)
GPU	NVIDIA TESLA P100 (16 GB, SXM2) $\times$ 4	2928 (732 $\times$ 4)
Network	Intel Omni-Path HFI 100 Gbps $\times$ 4	50 (12.5 $\times$ 4)
Memory	DDR4-2400 DIMM 256 GB	–
PCI Express	PCI Express Gen3 $\times$ 16	–

6.1 Performance on a Single Process

We show the performance results of the application on a single process by comparing three versions as follows. A CPU version is the original code parallelized by using OpenMP library, and executed on a single node (two CPU sockets). A GPU version is written in CUDA, and executed on a single GPU. An Optimal GPU version is optimized by using a boundary separate technique described Sect. 4.2 above. CPU and GPU codes are compiled with the NVIDIA CUDA Compiler 8.0.61 (-O3 -use_-fast_math -restrict -Xcompiler fopenmp –gpu-architecture = sm_60 -std = C++ 11). As for OpenMP parallelization, we use 28 threads on two Intel Xeon E5-2680 V4 Processor, while for GPU computation, the number of threads is set to $min(N_{Leaf}, 256)$.

Table 4 shows the benchmark parameters and the single process performance on TSUBAME 3.0. Here, the single process performance is estimated by subtracting the communication cost from the total cost. We scan the number of grid points in a leaf (Nleaf), while the total number of grid point are set to be equal. The performances in mega-lattice update per second (MLUPS) are measured in single precision. Table 4 shows the performances of the GPU version are about 10 times higher than those of the CPU version under various leaf size. It is unclear why the GPU performance is much higher than the ratio of GPU and CPU memory bandwidth. We estimate that the main kernel is compute intensive, and the NVIDIA CUDA compiler may not generate the SIMD-optimized CPU code. There is a possibility that the Intel compiler can generate faster CPU code.

Table 4. Performance on a single process in a single node of TSUBAME 3.0.

Nleaf	# of leaves in each level (Lv. = 0/1/2)	CPU (2 sockets) MLUPS	GPU MLUPS	Optimal GPU MLUPS
4^3	19008 /73728 /294912	23.3	231.6	383.5
8^3	2448 /9216 /36864	17.4	237.4	369.7
16^3	324 /1152 /4608	18.0	229.0	342.7
32^3	45 /144 /576	13.2	184.4	243.5

The performances of the Optimal GPU version are about 1.5 times higher than those of the GPU version under the conditions of $N_{Leaf} = (4^3, 8^3, 16^3)$. Since the benchmark is executed including the whole AMR leaves, the boundary separate technique works well under the condition with a small leaf size.

6.2 Performance on Multiple Processes in a Single Node

We show the performance results of the application on multiple processes in a single node. A communication cost of GPU based applications becomes a large overhead compared with that of CPU based ones. Table 3 shows that the memory bandwidth of GPUs is 19 times higher than that of CPUs in a single node. In other words, an impact of the communications cost on GPUs are 19 times larger than that on CPUs.

Table 5 shows the performance the Optimal GPU version with 4 MPI processes in a single node. The total number of leaves is 4 times larger than the condition used in Table 4. Although the performances in a single node is higher than those in a single GPU, the communication time occupies most of the total calculation time particularly when leaf size equals to 4^3. Since MPI communications are executed in each leaf unit, it is difficult to obtain high network bandwidth with a small message size. Unfortunately, MPI communications using Unified memory in OpenMPI 2.1.2 are slower than using Device or Host memory. This may be resolved by using GPUDirect RDMA or NVLink. We will address this issue in future work.

Table 5. Performance of GPU computation in a single node.

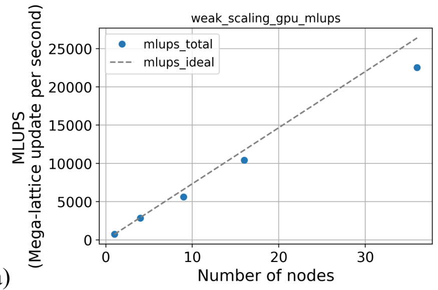
Nleaf	# of leaves in each process (Lv. = 0/1/2)	MLUPS (4 GPUs)	MPI cost %
4^3	19008/73728/294912	261.0	88.2
8^3	2448/9216/36864	729.5	65.4
16^3	324/1152/4608	840.6	48.8

(Note: OpenMPI 2.1.2 supports GPUDirect RDMA, which enables a direct P2P (Peer-to-Peer) data transfer between GPUs. However, we do not succeed in MPI communications using the GPUDirect RDMA in TSUBAME 3.0.)

6.3 Performance on Multiple Nodes

We show the performance results of the application in multiple nodes. The leaf size is set to 8^3 considering the performance and applicability to real problems. The number of leaves in a node is the same as that in Sect. 6.2 above.

Figure 9 presents weak scalabilities of CPU and GPU performances on TSUBAME 3.0. In these figures, the horizontal axis indicates the number of nodes, and the vertical axis indicates the MLUPS per step respectively.

(a)


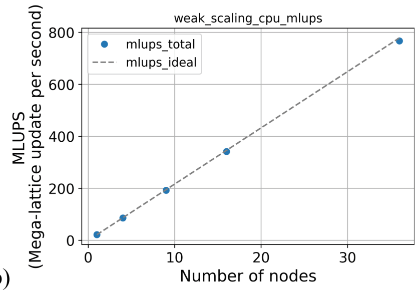
(b)


Fig. 9. Weak scaling results of the LBM simulation on (a) GPUs and (b) CPUs. 4 MPI processes are executed in each node.

In the weak scaling tests, the parallel efficiencies from 1 node to 36 nodes of CPUs and GPUs are 98% and 85%, respectively. Although CPUs show better scalability, the performance on a single GPU node (733MLUPS) is comparable to that on 36 CPU nodes (767MLUPS).

6.4 Estimation of Performance in Wind Simulation

Our final goal is to develop a real-time simulation of the environmental dynamics of radioactive substances. We estimate the minimum mesh resolution $\Delta x_{real\,time}$, at which a wind simulation can be executed in real time. The mesh resolution can be easily estimated from the Courant–Friedrichs–Lewy (CFL) condition as

$$\Delta x_{realtime} = \frac{U_{target}}{CFL_{target}} \times \Delta t_{cal}. \tag{18}$$

Here U_{target} is a wind velocity, and CFL_{target} is the CFL number at U_{target}, and Δt_{cal} is the elapse time per step.

We estimate the mesh resolution under the condition of $(U_{target}, CFL_{target}) = (5.0m/s, 0.2)$. The computational condition is based on a single GPU node case in the previous Subsect. 6.3. The fine leaves are placed near the ground surface, and the resolution changes in the height direction. The leaves are arranged with $24 \times 24 \times 17$ at Lv. 0, $48 \times 48 \times 16$ at Lv. 1, and $96 \times 96 \times 16$ at Lv. 2. The computational performance is achieved 733MLUPS using a single GPU node. The minimum mesh resolution becomes $\Delta x_{realtime} = m$ that corresponds to the whole computation domain size of $(L_x, L_y, L_z) = (2.8\,km, 2.8\,km, 3.3\,km)$. The above estimation shows that a detailed real-time wind simulation is realized by GPU computing.

7 Summary and Conclusions

This paper presented the GPU implementation of air flow simulations on the environmental dynamics of radioactive substances. We have successfully implemented the AMR-based LBM with a state-of-the-art cumulant collision operator. Our code is written in CUDA 8.0, and executed both on CPUs and GPUs by using the CUDA runtime API "cudaMallocManaged". Since the LBM kernel needs a lot of register memories on GPUs, the number of threads executed is limited by the lack of registers. We propose the effective optimization to create a kernel function for each conditional branch. This technique can reduce the number of registers compared to the original function, and the single GPU performance is accelerated by ~ 1.5 times. The performance of a single GPU process (NVIDIA TESLA P100) achieved 383.3 mega-lattice update per second (MLUPS) with the leaf size of 4^3 in single precision. The performance is about 16 times higher than that of a single CPU process (two Broadwell-EP 14 cores 2.4 GHz).

We have also discussed the weak scalability results. Regarding the weak scalability results, 36 GPU nodes achieved 22535 MLUPS with the parallel efficiency of 85% compared with a single GPU node. The present scaling studies revealed a severe performance bottleneck due to MPI communication, which will be addressed via GPUDirect RDMA or NVLink in the future work.

Finally, we estimate the minimum mesh resolution $\Delta x_{realtime}$ at which air flow simulations can be executed in real time. The above estimation shows that a detailed real-time wind simulation is realized by GPU computing. We conclude that the present scheme is one of efficient approaches to realize a real-time simulation of the environmental dynamics of radioactive substances.

Acknowledgements. This research was supported in part by the Japan Society for the Promotion of Science (KAKENHI), a Grant-in-Aid for Scientific Research (C) 17K06570 and a

Grant-in-Aid for Scientific Research (B) 17H03493 from the Ministry of Education, and "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" in Japan (Project ID: jh170031-NAH). Computations were performed on the TSUBAME 3.0 at the Tokyo Institute of Technology, and the ICEX at the Japan Atomic Energy Agency.

References

1. Nakayama, H., Takemi, T., Nagai, H.: Adv. Sci. Res. **12**, 127–133
2. Rothman, D.H., Zaleski, S.: J. Fluid Mech. **382**(01), 374–378 (1997)
3. Inamuro, T.: Fluid Dyn. Res. **44**, 024001 (2012). 21 pp.
4. Inagaki, A., Kanda, M., et al.: Boundary-Layer Meteorology, pp. 1–21 (2017)
5. Kuwata, Y., Suga, K.: J. Comp. Phys. **311** (2016)
6. Rahimian, A., Lashuk, I., et al.: In: Proceedings of the 2010 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE Computer Society (2010)
7. Rossinelli, D., Tang, Y.H., et al.: In: Proceedings of the 2015 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis, vol. 2. IEEE Computer Society (2015)
8. Berger, M.J., Oliger, J.: J. Comp. Phys. **53**(3), 484–512 (1984)
9. Zhao, Y., Liang-Shih, F.: J. Comp. Phys. **228**(17), 6456–6478 (2009)
10. Zhao, Y., Qiu, F., et al.: Proceedings of 2007 Symposium on Interactive 3D Graphics, pp. 181–188 (2007)
11. Yu, Z., Fan, L.S.: J. Comput. Phys. **228**(17), 6456–6478 (2009)
12. Wang, X., Aoki, T.: Parallel Comput. **37**(9), 521–535 (2011)
13. Shimokawabe, T., Aoki, T., et al.: In: Proceedings of the 2010 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE Computer Society (2010)
14. Shimokawabe, T., Aoki, T., et al.: In: Proceedings of the 2011 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis, vol. 3. IEEE Computer Society (2011)
15. Feichtinger, C., Habich, J., et al.: Parallel Computing **37**(9), 536–549 (2011)
16. Zabelock, S., et al.: J. Comput. Phy. **303**(15), 455–469 (2015)
17. Kang, S.K., Hassan, Y.A.: J. Comput. Phys. **232**(1), 100–117 (2013)
18. Zou, Q., He, X., et al.: Phys. Fluid **9**(6), 1591–1598 (1996)
19. Germano, M., Piomelli, U., Moin, P., Cabot, W.H.: Physics of Fluids A: Fluid Dynamics 3 (7), pp.1760–1765 (1991)
20. Lilly, D.K.: Phys. Fluids A **4**(3), 633–635 (1992)
21. Geier, M., Schonherr, M., et al.: Comput. Math. Appl. **70**(4), 507–547 (2015)
22. Geier, M., Psquali, A., et al.: J. Comput. Phys. **348**, 889–898 (2017)
23. Kim, J., Kim, D., Choi, H.: J. Comput. Phys. **171**(20), 132–150 (2001)
24. Peng, Y., Shu, C., et al.: J. Comput. Phys. **218**(2), 460–478 (2006)
25. Chun, B., Ladd, A.J.C.: Phys. Rev. E **75**(6), 066705 (2007)
26. Yin, X., Zhang, J.: J. Comput. Phys. **231**(11), 4296–4303 (2012)
27. Guzik, S.M., Weisgraber, T.H., et al.: J. Comput. Phys. **259**(15), 461–487 (2014)
28. Laurmaa, V., Picasso, M., Steiner, G.: Comput. Fluids **131**(5), 190–204 (2016)
29. Zuzio, D., Estivalezes, J.L.: Comput. Fluids **44**(1), 339–357 (2011)
30. Usui, H., Nagara, A., et al.: Proc. Comput. Sci. **29**, 2351–2359 (2014)

31. Open MPI: Running CUDA-aware Open MPI. https://www.open-mpi.org/faq/?category=runcuda
32. NVIDIA: Whitepaper, NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
33. Ghia, U., Ghia, K.N., Shin, C.T.: J. Comput. Phys. **48**, 387–411 (1982)
34. National Institute of Advanced Industrial Science and Technology, Database, (in Japanese). https://unit.aist.go.jp/emri/ja/results/db/01/db_01.html