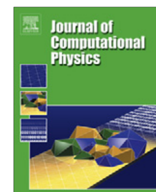
Journal of Computational Physics 231 (2012) 7932–7958

Contents lists available at SciVerse ScienceDirect

Journal of Computational Physics

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



Large-scale simulations on multiple Graphics Processing Units (GPUs) for the direct simulation Monte Carlo method

C.-C. Su [a], M.R. Smith [b], F.-A. Kuo [a,c], J.-S. Wu [a,c,*], C.-W. Hsieh [c], K.-C. Tseng [d]

[a] Department of Mechanical Engineering, National Chiao Tung University, Hsinchu, Taiwan
[b] Department of Mechanical Engineering, National Cheng Kung University, Tainan, Taiwan
[c] National Center for High-Performance Computing, National Applied Research Laboratories, Hsinchu, Taiwan
[d] National Space Organization, National Applied Research Laboratories, Hsinchu, Taiwan

ARTICLE INFO

Article history:
Received 26 November 2011
Received in revised form 21 May 2012
Accepted 24 July 2012
Available online 8 August 2012

Keywords:
Rarefied gas dynamics
Parallel direct simulation Monte Carlo
Graphics Processing Unit (GPU)
MPI-CUDA
Very large-scale simulation

ABSTRACT

In this study, the application of the two-dimensional direct simulation Monte Carlo (DSMC) method using an MPI-CUDA parallelization paradigm on Graphics Processing Units (GPUs) clusters is presented. An all-device (i.e. GPU) computational approach is adopted where the entire computation is performed on the GPU device, leaving the CPU idle during all stages of the computation, including particle moving, indexing, particle collisions and state sampling. Communication between the GPU and host is only performed to enable multiple-GPU computation. Results show that the computational expense can be reduced by 15 and 185 times when using a single GPU and 16 GPUs respectively when compared to a single core of an Intel Xeon X5670 CPU. The demonstrated parallel efficiency is 75% when using 16 GPUs as compared to a single GPU for simulations using 30 million simulated particles. Finally, several very large-scale simulations in the near-continuum regime are employed to demonstrate the excellent capability of the current parallel DSMC method.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

The direct simulation Monte Carlo (DSMC) method [1] is a computational tool for simulating flows in which effects at the molecular scale become significant. The Boltzmann equation [2], which is appropriate for modeling these rarefied flows, is extremely difficult to solve numerically due to its high dimensionality and the complexity of the collision term. DSMC provides a particle based alternative for obtaining realistic numerical solutions. In DSMC, the movement and collision behavior of a large number of representative "simulation particles" within the flow field are decoupled over a time step which is a small fraction of the local mean collision time. The computational domain itself is divided into either a structured or unstructured grid of cells which are then used to select particles for collisions on a probabilistic basis and also are used for sampling the macroscopic flow properties. The method has been shown to provide a solution to the Boltzmann equation statistically when the number of simulated particles is large enough [3]. However, high computational cost has hindered further applications of DSMC to some practical problems, especially in the near-continuum (collision-dominated) regime, while the non-equilibrium effect may be important. Hence, it is important to increase the computational efficiency to extend the applicability of the DSMC method.

The general wisdom for accelerating the DMSC computation is to parallelize the code using the massage passing interface (MPI) protocol running on clusters with large numbers of processors [4–7]. Such implementations rely upon the multiple

* Corresponding author at: Department of Mechanical Engineering, National Chiao Tung University, Hsinchu, Taiwan. Tel.: +886 3 573 1693; fax: +886 3 611 0023.
E-mail address: chongsin@faculty.nctu.edu.tw (J.-S. Wu).

0021-9991/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jcp.2012.07.038

instructions on multiple data (MIMD) parallelization philosophy. Recently, Graphics Processing Units (GPUs) have become an alternative platform for parallelization, employing a single instruction on multiple data sets (SIMD) parallelization philosophy. The resulting parallelization is much more efficient at the cost of flexibility – as a result, the computational time of several scientific computations, especially those which are optimally applied to vectorized computation strategies, have been demonstrated to reduce significantly, together with power and equipment costs. The DSMC method is essentially a highly local particle method; however, reports on DSMC using GPU computing were very limited. Especially, there is no related study in hybrid MPI-CUDA implementation for DSMC method. A parallel DSMC-accelerated in SIMD architecture had been presented by Gladkov et al. [8]. Although they obtained speedup of 65 times on a single GPU by simulating several 3D small-scale simulations (0.1–1 million particles), they have not demonstrated its capability by simulating some challenging large-scale real flow problems. In addition, several important features, such as particle removal on a single GPU and realistic boundary conditions (fully diffusive wall and inlet/outlet boundary conditions), were not included in the proposed algorithm. Each component of DSMC-accelerated in SIMD architecture and interaction between GPU computing and dimensionless parameters in gas dynamics were not discussed in detail. Recently, we had presented a parallel DSMC method in SIMD on a single GPU [9]. We demonstrated a modest speedup of 3–10 times in the simulation of supersonic flow over a flat plate and a supersonic lid-cavity flow problem with 1–10 million particles caused by a low speedup due to smaller problems and worse GPU machines.

Graphics Processing Units (GPUs) are co-processors originally designed to assist in the computations required for displaying graphics on monitor. In recent years, the GPU has been demonstrated as an effective tool in the computation of scientific and engineering problems. In the past, researchers performing General Purpose computing using Graphics Processing Units (GPGPU) [10] were forced to use graphics-related API's (like OpenGL, and later OpenCL) until CUDA (Compute Unified Device Architecture) [11] was introduced in 2007 by NVIDIA Corp. NVIDIA's CUDA is a general-purpose parallel computing architecture with a new parallel programming model and instruction set architecture. Nowadays, we can employ GPU co-processors to accelerate our computation using CUDA. And also CUDA has been developed to work in many popular computing languages such as C/C++, FORTRAN and even Java.

Thus, in this paper we intend to study the parallel performance of the DSMC method using a hybrid MPI-CUDA parallelization paradigm and demonstrate its capability in simulating several large-scale near-continuum gas flow problems.

This paper is organized as follows. The governing equation and the standard DSMC method is first introduced. Then, the DSMC method using hybrid MPI-CUDA parallelization is described in detail. The speedup and parallel performance on multiple GPUs using CUDA is investigated through subsonic/supersonic lid-driven cavity benchmarks. Following this, we apply the parallel DMSC implementation to several very large-scale simulations in near-continuum flows. Finally, the paper is summarized with some important findings.

2. Numerical method

2.1. The Boltzmann Equation

The Boltzmann equation [2] is employed to describe molecular transport phenomena based on gas-kinetics theory and statistical mechanics. It is an integral-differential equation which is valid for all flow regimes, which ranges from highly rarefied to continuum flows. The Boltzmann equation based on the assumption of binary collision is written as

$$\frac{\partial(nf)}{\partial t} + \mathbf{c}\cdot\frac{\partial(nf)}{\partial \mathbf{r}} + \mathbf{F}\cdot\frac{\partial(nf)}{\partial \mathbf{c}} = \int_{-\infty}^{\infty}\int_{0}^{4\pi} n^2(f^*f_1^* - ff_1)c_r\sigma d\Omega d\mathbf{c}_1 \tag{1}$$

where f and n are the velocity distribution function and number density, respectively. $\mathbf{c}$ and c_r are the molecular velocity and the relative molecular speed between two particles, respectively. And $\mathbf{F}$ is an external force, t is the time, σ is the collision cross section, and $d\Omega$ is the solid angle. f and f_1 denote two different types of molecules, and the superscript $*$ denotes the post-collision quantities.

In general, it is very difficult to solve the Boltzmann equation directly using conventional numerical method because the difficulty of correctly and efficiently modeling the integral collision term (right-hand term), in addition to the high numbers of dimensionality of the Boltzmann equation (configuration and velocity spaces). Instead, the DSMC method, developed by Bird during 1960s [1], has been widely used to solve the Boltzmann equation when the flow is rarefied. Until recently, the DSMC method was shown statistically to equivalent to solving the Boltzmann equation as the number of particles is large [3].

2.2. The standard DSMC method

The DSMC method is a particle-based method for solving the Boltzmann equation and it is widely used to simulate gas flows in the rarefied gas regime. The central idea of DSMC is to reproduce real flow properties with no more than collision mechanics of gas molecules through a large number of pseudo particles that are used to represent real gas molecules. Each pseudo particle represents a fairly large number of real gas molecules. In the DSMC simulation, there is an important feature that the motions and the collisions of pseudo particles are uncoupled over a time interval. The time interval (timestep) should be kept much smaller than the mean collision time.

The general procedures of the standard DSMC method [1] are divided into several steps, including initialization, particle movement, indexing, collision, and sampling. In the first step, the initialization is to set up the geometry and flow conditions. A physical space is divided into a network of cells and the domain boundaries have to be assigned according to the flow conditions. The size of computational cells should be smaller than the local mean free path. In general, each particle is assigned the velocities in x-, y-, and z-direction based on the Maxwell–Boltzmann distribution according to number density and temperature of flow conditions. Position of each particle is randomly distributed within a cell. In the step of particle movement, each particle directly moves through a time step if it does not collide with a solid surface. When the particle collides with a solid surface, the final position and velocities are determined by the boundary type. After all particles are moved, the relation between locations of particles and cells is re-indexed for the purpose of easier selection of collision partner in a cell. In the step of collision, particles in a cell are randomly selected for collision based on the real collision rate. When the flow reaches a steady state after several time steps of simulation, flow properties are then sampled in each cell.

2.3. Hybrid MPI-CUDA parallelization paradigm

Conventionally, parallel DSMC method is implemented on distributed-memory machines using MPI protocol. The computational loading is distributed on each processor (CPU) to reduce simulation time, communicate and synchronize among all processors using MPI protocol. In fact, the MPI is collection of specifications of message passing interface libraries. In contrast, the CUDA, which is a parallel computing architecture on a single GPU, is employed to accelerate computation on the DSMC-related components such as particle movement, indexing, collision and sampling. Therefore, the difference between MPI and hybrid MPI-CUDA approach is that the latter combines the MPI among different nodes of CPU processors and the CUDA on each GPU to make the most of the computing resources on a multiple GPU machines distributed around different nodes. The detail of a hybrid MPI-CUDA parallelization paradigm is described in next.

A hybrid MPI-CUDA parallelization paradigm is presented in Fig. 1. A spatial domain decomposition method was adopted in this approach. In this method, the MPI protocol is used to exchange data from memory of all MPI processors and synchronize. CUDA is used to put the DSMC-related simulation components on GPU and data transfer between CPU (host) memory and the GPU (device) global memory [12]. We used the CUDA API function *cudaGetDeviceCount()* [12] to get the number of GPU devices available in each node and *cudaSetDevice()* to assign a GPU to each individual MPI process [12]. For example, as shown in Fig. 1, a simple computing cluster with 4 nodes and 4 GPU devices per node using MPI can use *cudaGetDeviceCount(&GPUNum)* to correctly identify the number of valid GPU's on each node, and then *cudaSetDevice(MPIMyID%GPUNum)* to assign each GPU to each MPI processor. For data exchange between global memory of different GPU devices (for example, from device-A to device-B) we use the CUDA API function *cudaMemcpy()* [12] to transfer data from device-A to host-A (as shown by the red line in Fig. 1). Next, the data is transferred from host-A to host-B (as shown by the blue line in Fig. 1) using the MPI protocol with *MPI_Send()* and *MPI_Recv()*. Finally, we use *cudaMemcpy()* to transfer data from host-B to device-B. This flexible method allows a single subroutine to manage communications without concern for (i) the device type, or (ii) if the device is located on the same physical node. Note the data transfer, including particle and computational cell information, are copied between the host and the device only in the initialization phase and the end of a simulation. Therefore, there is no need to employ CUDA streams the current implementation.

2.4. Parallel Implementation of DSMC on multiple GPUs

In this study, an all-device (GPU) computational approach is adopted where the computational components of the DSMC method (including particle moving, indexing, particle collisions and sampling) are performed entirely on the GPU device.

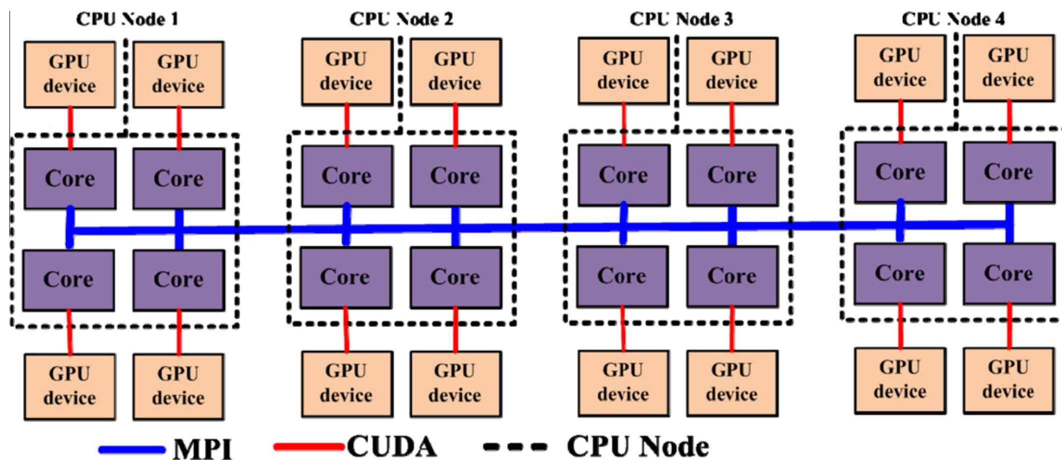


Fig. 1. Multiple GPU devices across multiple nodes MPI-CUDA paradigm.

This required some changes to the original DSMC method in order to allow efficient all-device computation. Fig. 2 shows the flowchart of DSMC computation using hybrid MPI-CUDA approach. During the initialization stage, input data is loaded into the memory of the primary cluster node and finally distributed to all other processors using the MPI protocol. The initial states are computed on each host CPU in parallel. We used the CUDA API function *cudaMalloc()* [12] to allocate the required global memory for each GPU (device) on every delegated CPU (host). Finally, computationally relevant information (including particle and computational cell information) is transferred to the global memory of device from the host memory. Following this initialization period the unsteady phase of the DSMC simulation is performed - particle moving, indexing, particle selection (and consequent collisions) and sampling are executed on each GPU. When the simulation is nearing completion (i.e. the flow has reached steady state and the sampled data is sufficient to remove undesired statistical scatter) we move the sampled data from GPU (device) to CPU (host). Finally, calculation of the macroscopic properties is performed by each MPI processor (host) and the data is written to file for further analysis.

A flowchart demonstrating the particle movement phase algorithm in our proposed hybrid MPI-CUDA DSMC scheme is shown in Fig. 3, involving:

- A GPU-centred function, known as a kernel (see Algorithm 1) is used to calculate the positions of all particles over a time step Δt. Each particle's deterministic motion is handled by a CUDA thread, using data held entirely in global memory.
- Particles determined to have left the current GPU's simulation domain are placed into a buffer (in the device, and finally on the host) in preparation for migration to other GPU devices.
- Introduce new particles based on inlet boundary conditions.
- Send and receive from buffers of all MPI processors (host) with the MPI API protocol *MPI_Send()* and *MPI_Recv()*.
- Reallocate particles from buffers into their newly allocated GPU's global memory.

When a particle arrives at an inlet/outlet boundary, or moves into a computational domain managed by a different GPU, it must be relocated. Traditionally, the particle information in the target and source memory is directly manipulated – this method is not well suited to the SIMD architecture employed by the GPU. Thus, we have developed a new algorithm to process such particles, as shown in Figs. 4 and 5. First, each particle requires a record indicating its current state – this is contained within the array *d_ParticleIn* (see Algorithm 1), where a value of one (1) indicates the particle is to remain in the same GPU and zero (0) indicating that (i) a transition to a different GPU, or (ii) inlet/outlet boundary treatment is required. A second array, *d_ParticleInMPI* (see Algorithm 1), is defined such that a value of one (1) confirms the particle for transition to another GPU. Following this, we use the optimized parallel prefix sum (scan) method [13] to efficiently obtain the memory

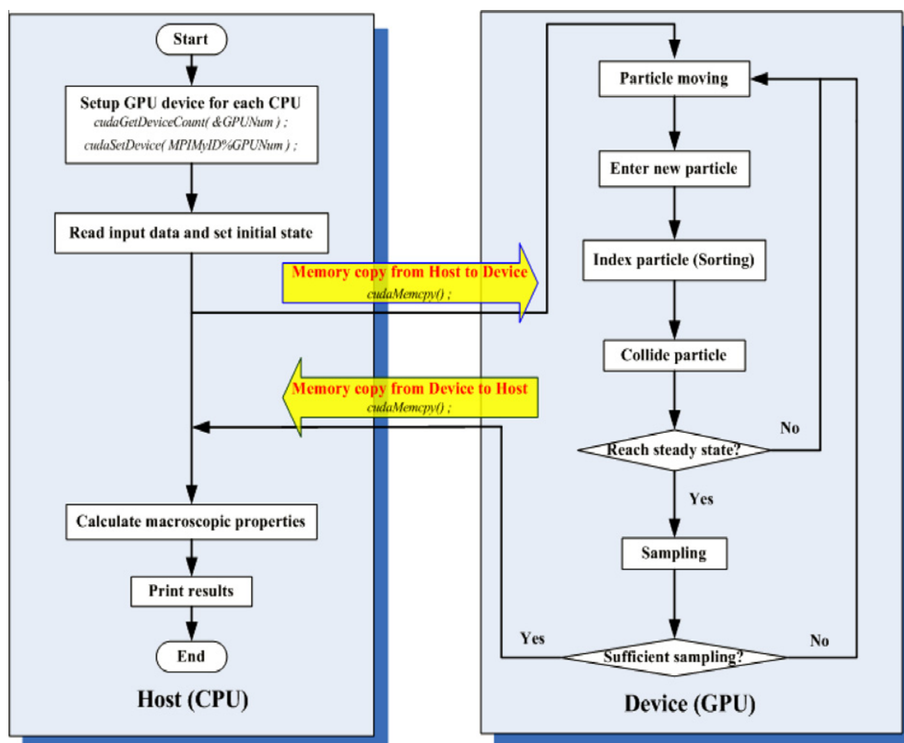


Fig. 2. Flowchart describing the application of DSMC to GPU-accelerated computation.

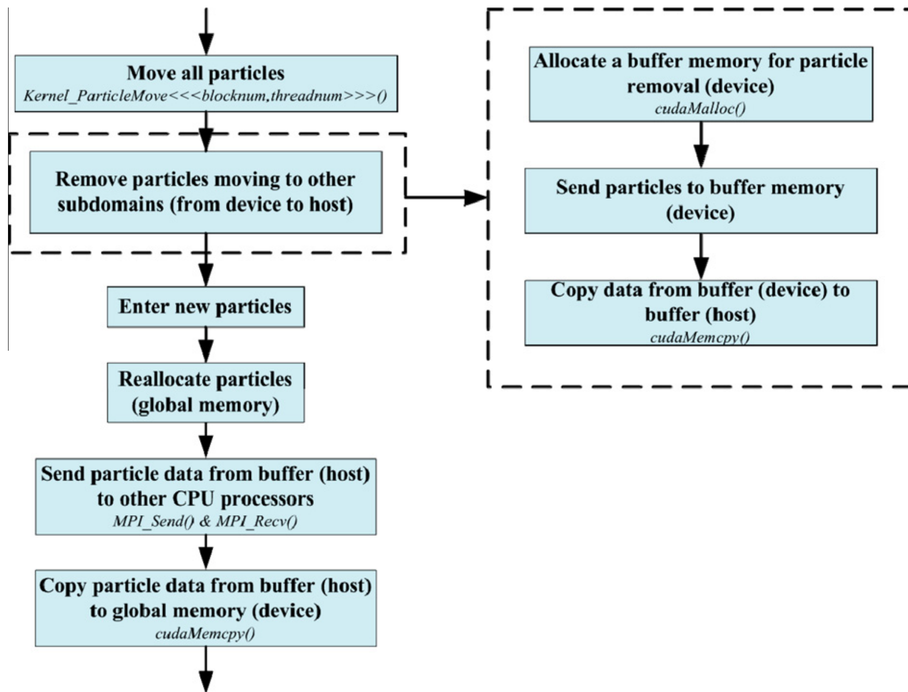


Fig. 3. Flowchart describing the particle movement phase of DSMC computation using MPI-CUDA.

location of particles (*d_ParticleWrite* and *d_ParticleWriteMPI*), including those designated for transport to other GPU's. This makes it possible to efficiently manage the movement of particles between GPU devices.

The implementation of particle indexing is similar to that of Bird's DSMC implementation [1]. Fig. 6 shows the particle indexing of DSMC with CUDA, involving the following steps:

- Counting the number of particles in each cell (Algorithm 2).
- Perform the parallel prefix sum (scan) [13].
- Indexing the relationship between particles and cells (Algorithm 3).

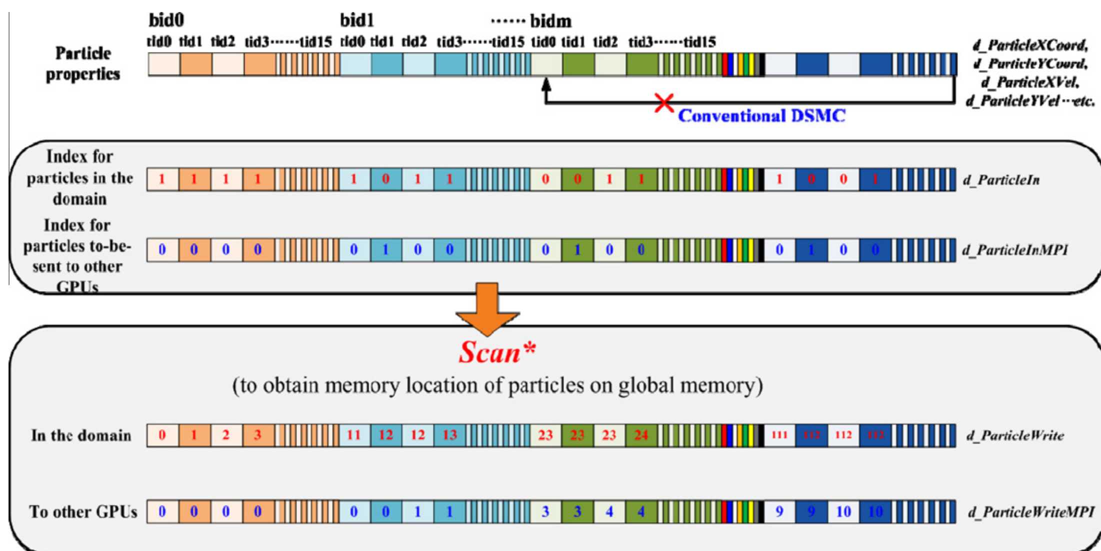


Fig. 4. Sketch demonstrating the location in memory of particles on global memory.

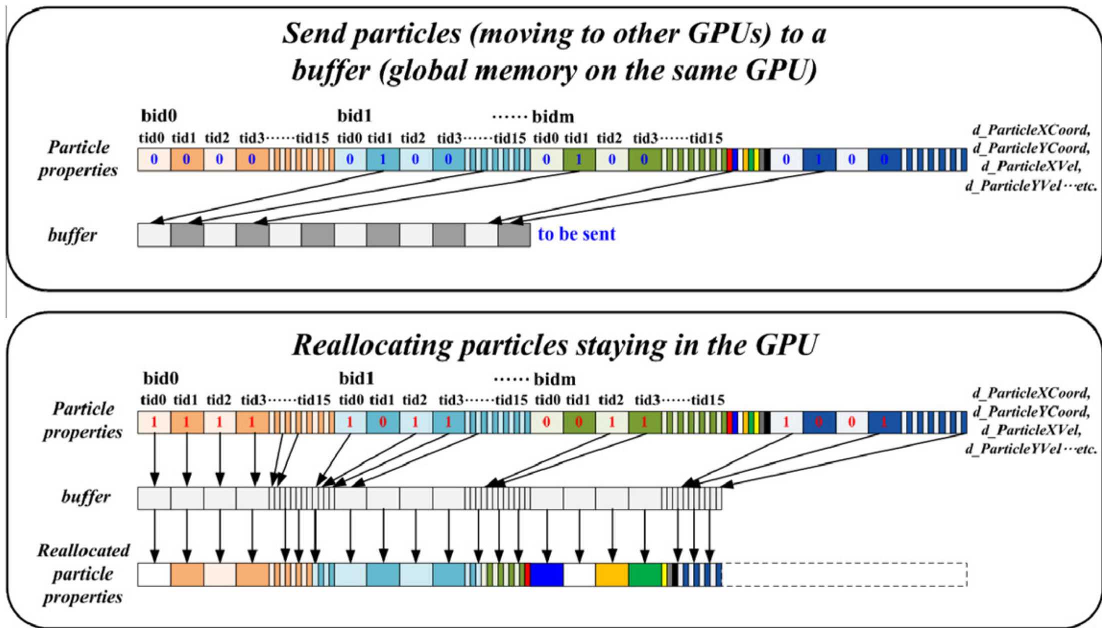


Fig. 5. Sketch demonstrating particle relocation (with in the GPU) and inter-GPU transported particles (to other GPUs) to a buffer.

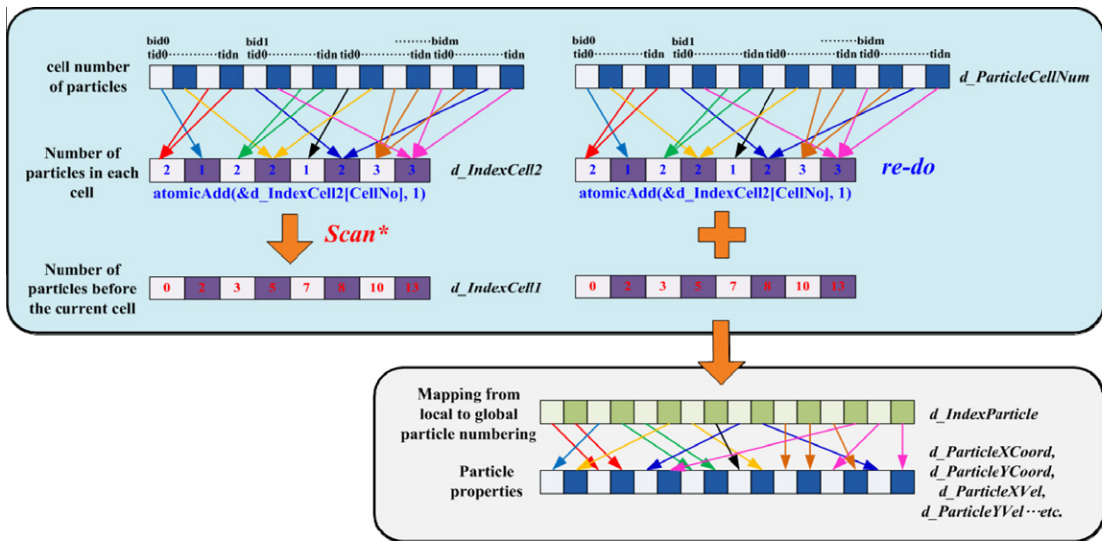


Fig. 6. The indexing method used by DSMC with CUDA.

In indexing of particles, we create three arrays for selecting potential particle collision pairs using the following concept:

Selecting a particle in ith-cell = *d_IndexCell1[i]* + *d_IndexCell2[i]*∗*Rand [0−1]*

Relating to the global particle number = *d_IndexParticle[selected particle]*where *d_IndexCell1[i]* and *d_IndexCell2[i]* contain the (i) cumulative total number of particles in cells $[0,1,2\ldots(i-1)]$ and (ii) the number of particles in the ith cell respectively. The mapping of particle indexes from local (single GPU) to global (over all GPU's) is contained within the array *d_IndexParticle[]*.

To avoid race conditions, Algorithms 2 and 3 both employ *atomic addition* [12]. This essentially ensures that threads cannot concurrently access critical areas of memory, at the sacrifice of parallel efficiency. During the collision phase all particle collisions within any given cell are handled by a dedicated CUDA thread, as demonstrated in Algorithm 4. Here, we employ the classical No Time Counter (NTC) method together with a variable hard sphere (VHS) model for particle collisions [1]. The nature of DSMC collisions requires a random number generator, both for the collision outcome and collision partner

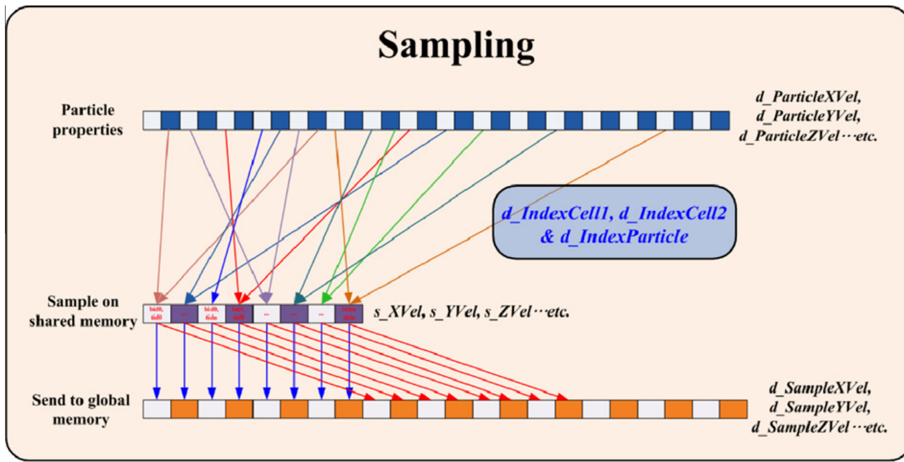


Fig. 7. The sampling method used by DSMC with CUDA.

selection. The algorithm for a uniformly distributed random number generator, based on the method employed by Bird and centred on the GPU device is presented in Code Snippet 1.

Fig. 7 shows the schematic diagram for the particle sampling phase of our proposed MPI-CUDA DSMC implementation. Particle sampling within each cell is handled by a dedicated CUDA thread on GPU, as shown in Algorithm 5, and all particles within a cell are sampled using shared memory [12], which results in much faster access than global memory. When sampling within each cell is completed, the sampled data is transferred from shared memory to global memory.

Table 1
Simulation conditions for subsonic/supersonic lid-driven cavity problem.

	Kn	Cell number	Total particle number
Case 1	0.01	200 × 200	10 M
Case 2	0.01	200 × 200	20 M
Case 3	0.01	200 × 200	30 M
Case 4	0.005	400 × 400	10 M
Case 5	0.005	400 × 400	20 M
Case 6	0.005	400 × 400	30 M
Case 7	0.002	1000 × 1000	10 M
Case 8	0.002	1000 × 1000	20 M
Case 9	0.002	1000 × 1000	30 M

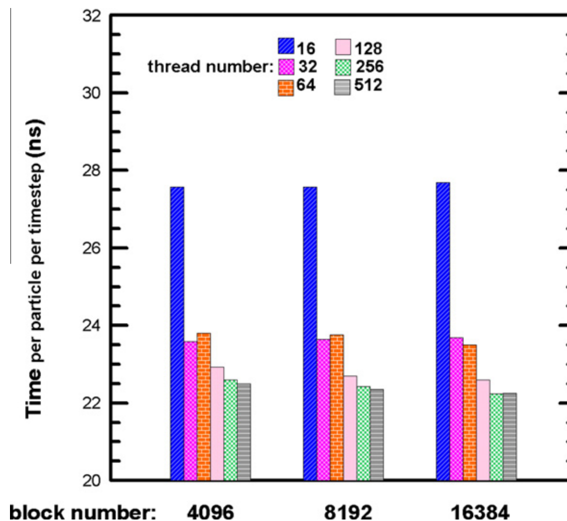


Fig. 8. Computational time per particle and per timestep when using different combinations of block and threads per block numbers for Case 5 in the subsonic lid-driven cavity problem.

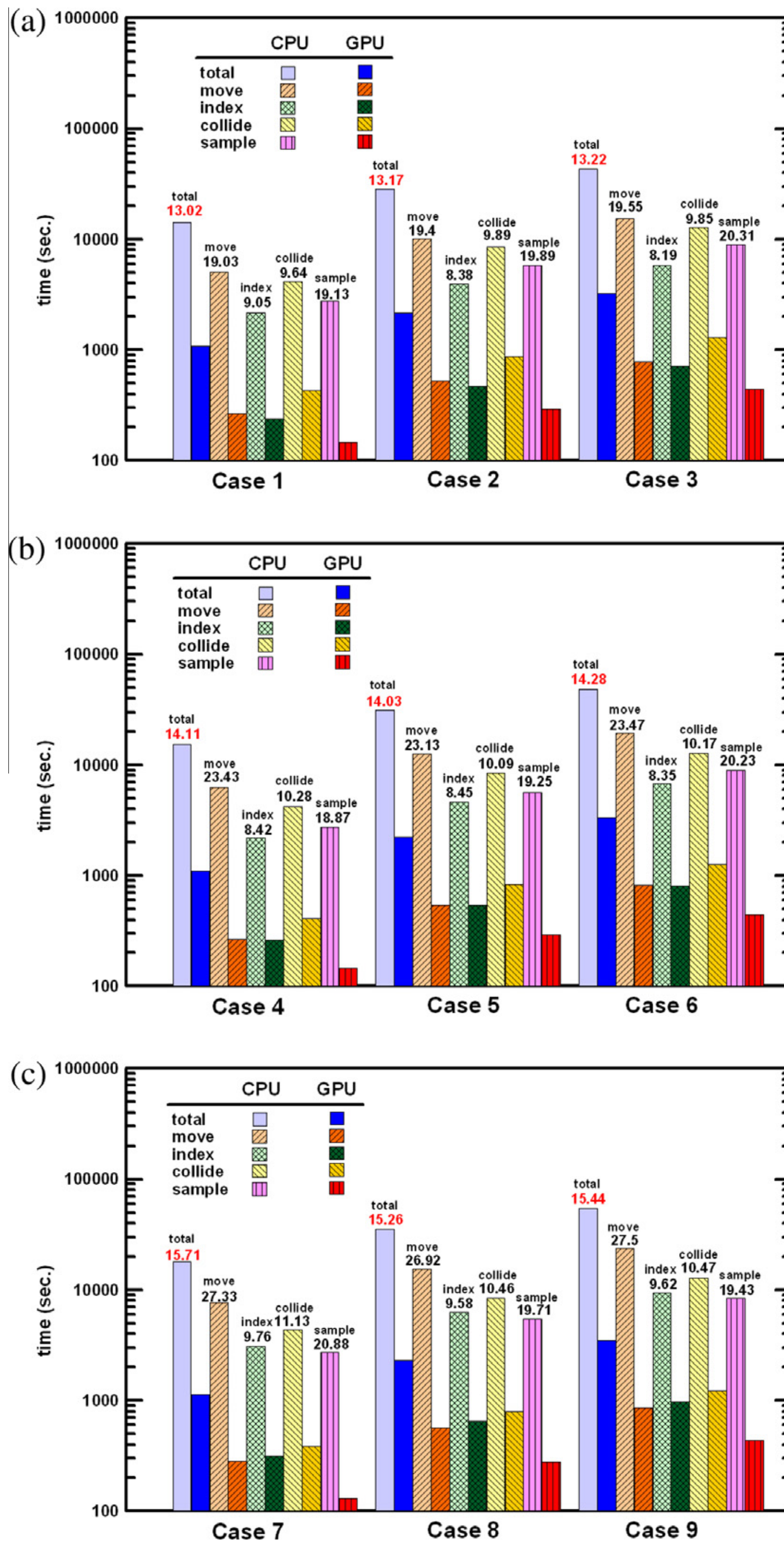


Fig. 9. Computational time and speedup$_{CPU/GPU}$ ratio using a core of the CPU (Intel Xeon X5670) and a GPU (NVIDIA Tesla M2070) for each component of DSMC in subsonic lid-cavity problem: (a) Kn = 0.01, (b) Kn = 0.005, and (c) Kn = 0.002.

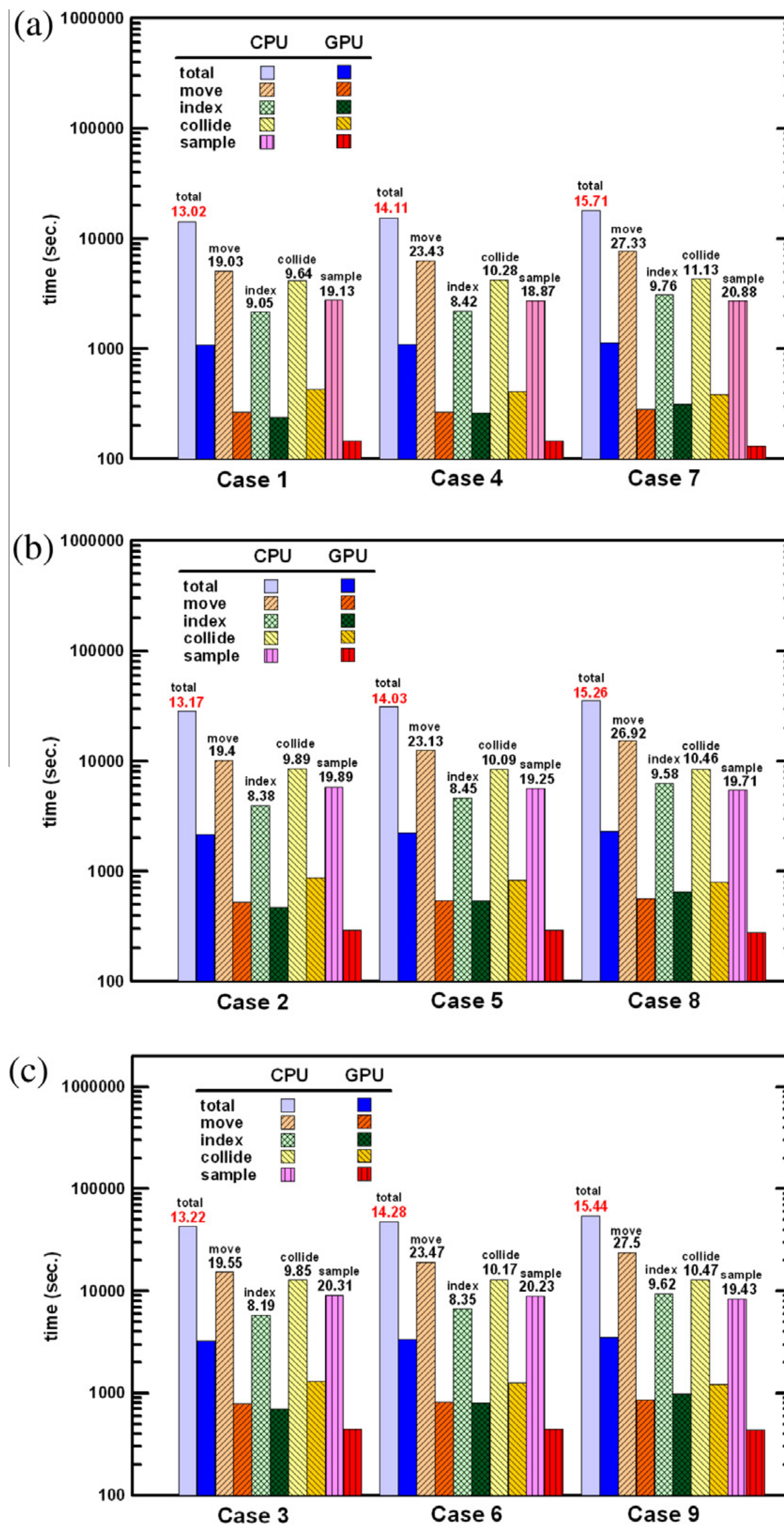


Fig. 10. Computational time and speedup$_{CPU/GPU}$ ratio using a core of the CPU (Intel Xeon X5670) and a GPU (NVIDIA Tesla M2070) for each component of DSMC in subsonic lid-cavity problem: (a) 10 M particles, (b) 20 M particles, and (c) 30 M particles.

3. Results and discussion

Here we discuss the performance of our proposed MPI-CUDA DSMC implementation on multiple GPUs using two test cases: a subsonic and a supersonic lid-driven cavity flow under various rarefied conditions (Kn = 0.01–0.002). Following this we demonstrate the capability of the proposed parallel DSMC method with MPI-CUDA using different numbers of GPU devices in several large-scale simulations in near-continuum flow. For the purpose of this analysis, all DSMC simulations herein (in Section 3.1. and Section 3.2.) are based on the execution of 5000 time steps using 16 NVIDIA Tesla M2070 devices (448 CUDA cores @ 1.15 GHz, 6 GB DDR5 RAM, ECC support) across 4 nodes. Each node has dual Intel Xeon X5670 (2.93 GHz, 12 M Cache, 6.4 GT/s Intel QPI) with 48 GB RDIMM RAM @ 1333 MHz. All nodes are connected through an infiniband fabric. With regards to the DSMC algorithm itself, the VHS collision model and NTC methods are applied in simulations unless otherwise specified.

3.1. Subsonic Lid-Driven Cavity Flow

A two-dimensional subsonic lid driven cavity problem is simulated here using our proposed hybrid DSMC method. The simulation domain is a square cavity (1 m $\times$ 1 m) with diffusely reflecting walls of fixed temperature (300 K). All walls are

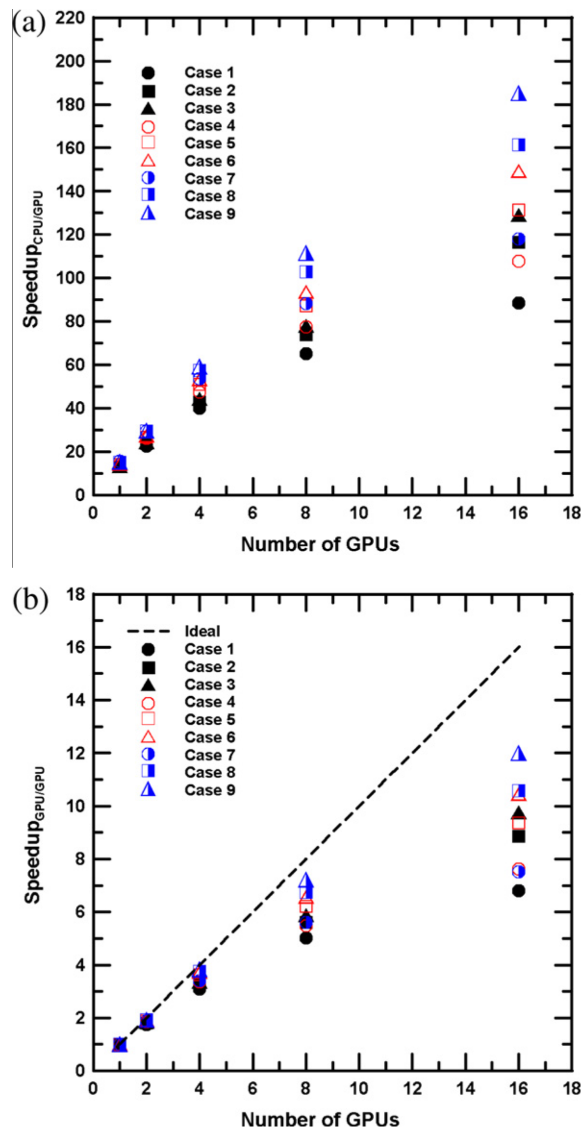


Fig. 11. Speedup ratio as a function of GPU (M2070) number in strong scaling for subsonic lid-driven cavity problem: (a) compared to a core of the CPU (Intel Xeon X5670), and (b) compared to a single GPU (M2070).

stationary except the upper wall which is moving (with positive velocity) at Mach number $M = 0.2$. The gas (ideal argon, $\gamma = 5/3$) is initially at rest with a temperature of 300 K. Various Knudsen numbers are investigated through the manipulation of gas density, computed using a characteristic length equal to the wall length. The cell size is uniform and approximately 1/2 of mean free path. Simulation conditions of all cases are summarized in Table 1. There are nine cases ranging from Kn = 0.01 to Kn = 0.002 with various number of particles (10 M–30 M). Note the Kn is the Knudsen number defined as the ratio of the mean free path to the characteristic flow length.

Fig. 8 shows the computational time per particle per timestep (i.e. the *specific time*) for different combinations of block and thread numbers for Case 5 as shown in Table 1. For this test case, it seems the optimal number of blocks and threads per block is 16,384 and 256 respectively. Future analysis and simulations of all test cases employ the same configuration of block and thread numbers.

3.1.1. Timing breakdown for a single GPU

Figs. 9 and 10 show the computational time required to execute 5000 timesteps and $speedup_{CPU/GPU}$, i.e. the ratio of computational time when using a single core of Intel Xeon X5670 and a single GPU (NVIDIA Tesla M2070). The former is presented by the same Knudsen number with different numbers of particles in each figure and the latter is presented by the same total number of particles with different Knudsen numbers in each figure. The results show that a speedup of 13–15 times can be obtained using a single GPU for the current simulation conditions. The results in Figs. 9 and 10 demonstrate that the speedup ratio depends slightly on the Knudsen number – we found that when the Knudsen number is smaller (i.e. nearer to continuum) the DSMC algorithm is consistently better accelerated using GPU, as shown in Fig. 10. This is caused by the longer runtime for the denser cases on CPU due to increasing contribution from both particle movement and indexing phases. However, the computational time on GPU scales almost linearly with number of particles no matter

Table 2
Non-uniformity (UN (%)) and speedup ratio ($S_{C/G}$ is $speedup_{CPU/GPU}$ (compared to a core of the Intel Xeon X5670 CPU), and $S_{G/G}$ is $speedup_{GPU/GPU}$ (compared to a single Tesla M2070 GPU)) in strong scaling for subsonic lid-driven cavity problem.

	GPUs	$S_{C/G}$	$S_{G/G}$	UN (%)		GPUs	$S_{C/G}$	$S_{G/G}$	UN (%)		GPUs	$S_{C/G}$	$S_{G/G}$	UN (%)
Case 1	1	13.02	1.00	0.00	Case 4	1	14.11	1.00	0.00	Case 7	1	15.71	1.00	0.00
	2	22.67	1.74	0.30		2	26.13	1.85	0.35		2	29.63	1.89	0.23
	4	40.13	3.08	0.27		4	47.36	3.36	0.95		4	53.57	3.41	0.48
	8	65.29	5.01	0.45		8	77.42	5.49	1.33		8	88.18	5.61	0.80
	16	88.65	6.81	8.69		16	107.69	7.63	1.18		16	118.06	7.52	2.54
Case 2	1	13.17	1.00	0.00	Case 5	1	14.03	1.00	0.00	Case 8	1	15.26	1.00	0.00
	2	23.82	1.81	0.21		2	26.69	1.90	0.38		2	29.45	1.93	0.21
	4	42.98	3.26	0.30		4	51.01	3.64	0.86		4	57.37	3.76	0.34
	8	73.84	5.61	0.48		8	87.25	6.22	1.27		8	102.91	6.75	0.70
	16	116.48	8.85	8.38		16	131.44	9.37	1.29		16	161.43	10.58	2.59
Case 3	1	13.22	1.00	0.00	Case 6	1	14.28	1.00	0.00	Case 9	1	15.44	1.00	0.00
	2	24.29	1.84	0.27		2	27.43	1.92	0.35		2	29.88	1.94	0.25
	4	44.47	3.36	0.33		4	53.45	3.74	0.87		4	59.38	3.85	0.39
	8	77.92	5.89	0.39		8	93.77	6.57	1.24		8	111.68	7.23	0.71
	16	129.03	9.76	8.53		16	149.28	10.45	1.23		16	185.28	12.00	2.79

Table 3
Simulation conditions and non-uniformity (UN (%)) in weak scaling for subsonic lid-driven cavity problem.

	Kn	GPUs	Total particle number	UN (%)		Kn	GPUs	Total particle number	UN (%)		Kn	GPUs	Total particle number	UN (%)
Case I	0.01	1	5 M	0.00	Case IV	0.005	1	5 M	0.00	Case VII	0.002	1	5 M	0.00
		2	10 M	0.18			2	10 M	0.46			2	10 M	0.15
		4	20 M	0.34			4	20 M	0.82			4	20 M	0.35
		8	40 M	0.39			8	40 M	1.14			8	40 M	0.73
		16	80 M	8.43			16	80 M	1.21			16	80 M	2.67
Case II	0.01	1	10 M	0.00	Case V	0.005	1	10 M	0.00	Case VIII	0.002	1	10 M	0.00
		2	20 M	0.20			2	20 M	0.36			2	20 M	0.24
		4	40 M	0.38			4	40 M	0.91			4	40 M	0.42
		8	80 M	0.43			8	80 M	1.13			8	80 M	0.67
		16	160 M	8.47			16	160 M	1.17			16	160 M	2.62
Case III	0.01	1	15 M	0.00	Case VI	0.005	1	15 M	0.00	Case IX	0.002	1	15 M	0.00
		2	30 M	0.22			2	30 M	0.35			2	30 M	0.22
		4	60 M	0.33			4	60 M	0.85			4	60 M	0.37
		8	120 M	0.40			8	120 M	1.11			8	120 M	0.74
		16	240 M	8.45			16	240 M	1.16			16	240 M	2.52

what the rarefaction is. In addition, the collision phase becomes the most time-consuming part of the DSMC method on GPU in contrast with the movement phase does on CPU. In general, the particle movement and the sampling phases are better accelerated with GPU than the other components of DSMC, including the indexing phase and particle collision phase, which is explained next.

In the movement phase, it is faster mainly because accessing to global memory for obtaining particle properties (velocities and positions) for all threads is a typical coalesced reading, although particle removal needs some special care (see Algorithm 1). In the sampling phase, although accessing to global memory for all threads, based on cell number mapping, is a non-coalesced reading, a faster speed is obtained because much faster shared memory is used to temporarily store sampling results. When sampling is completed for several cells, the sampled data is transferred from shared memory to global memory (see Algorithm 5). By doing so, there are two immediate advantages: (1) reduced number of global memory access and written, and (2) no synchronization required. In the indexing phase, it is slower because of the use of *atomicAdd* (see Algorithms 2 and 3) [12] which, in the worse case scenario, essentially a sequential function used in during the indexing phase. During the collision phase, access to global memory is non-coalesced for all threads due to the random nature of particle selection.

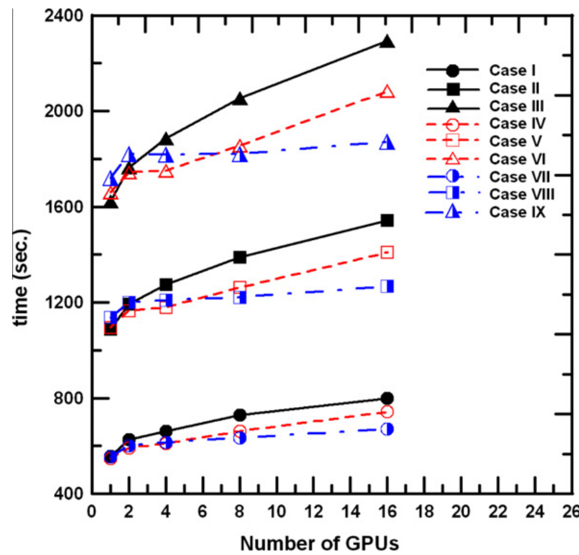


Fig. 12. Computational time (unit: second) as a function of the number of GPU devices for weak scaling in the subsonic lid-driven cavity problem.

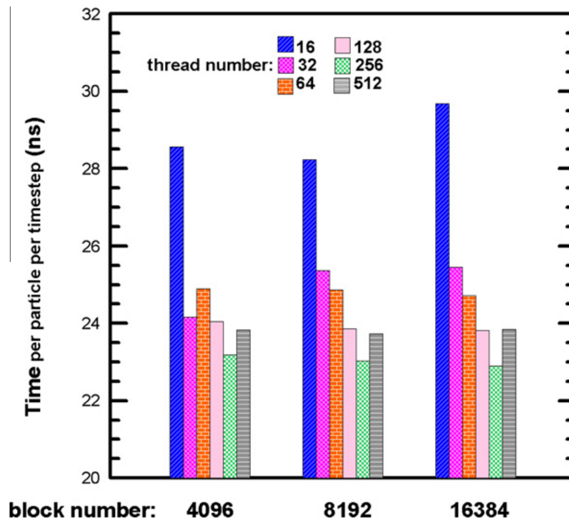


Fig. 13. Computational time per particle and per timestep when using different combinations of block and threads per block numbers for Case 5 in the supersonic lid-driven cavity problem.

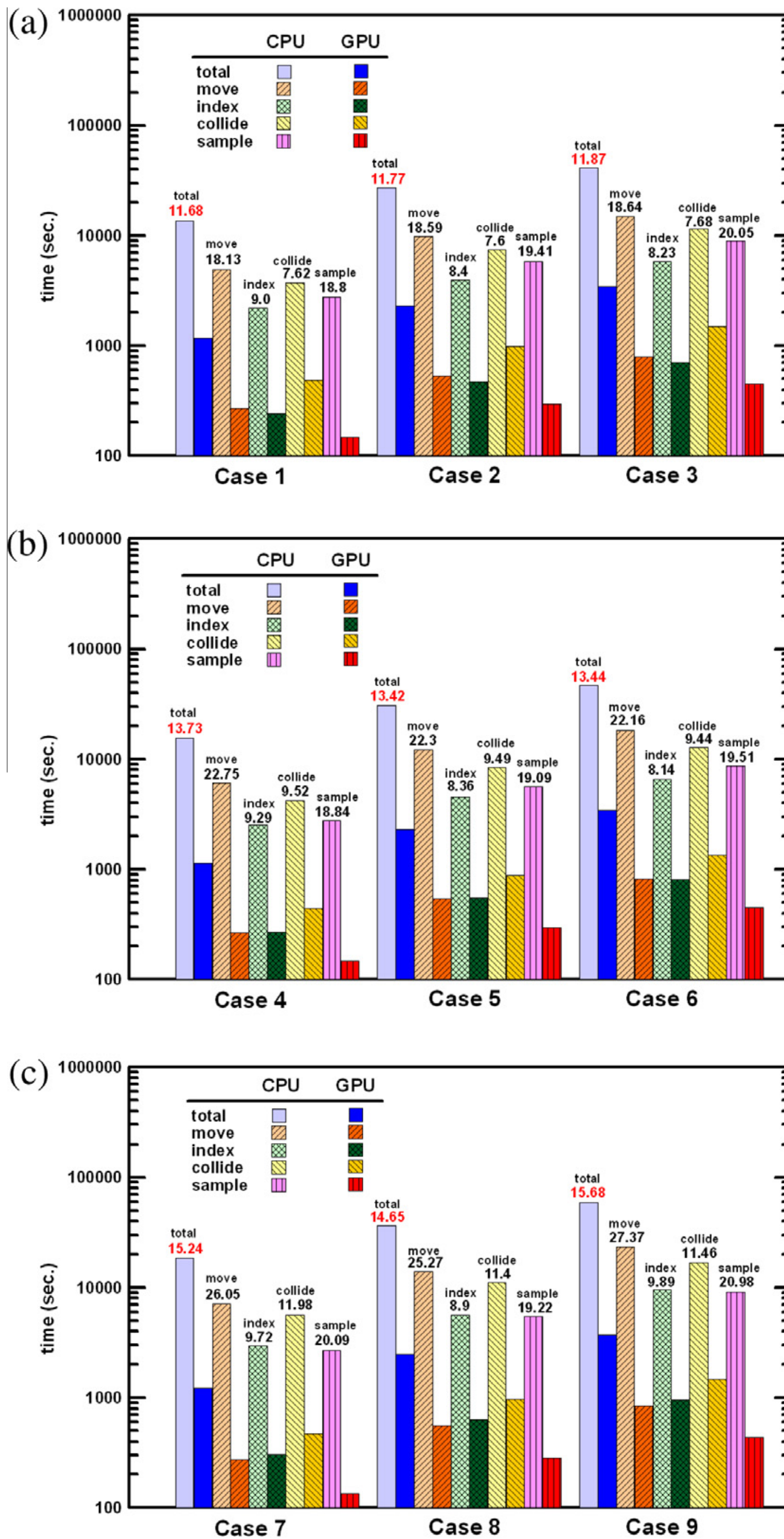


Fig. 14. Computational time and speedup$_{CPU/GPU}$ ratio using a core of the CPU (Intel Xeon X5670) and a GPU (NVIDIA Tesla M2070) for each component of DSMC in supersonic lid-driven cavity problem: (a) Kn = 0.01, (b) Kn = 0.005, and (c) Kn = 0.002.

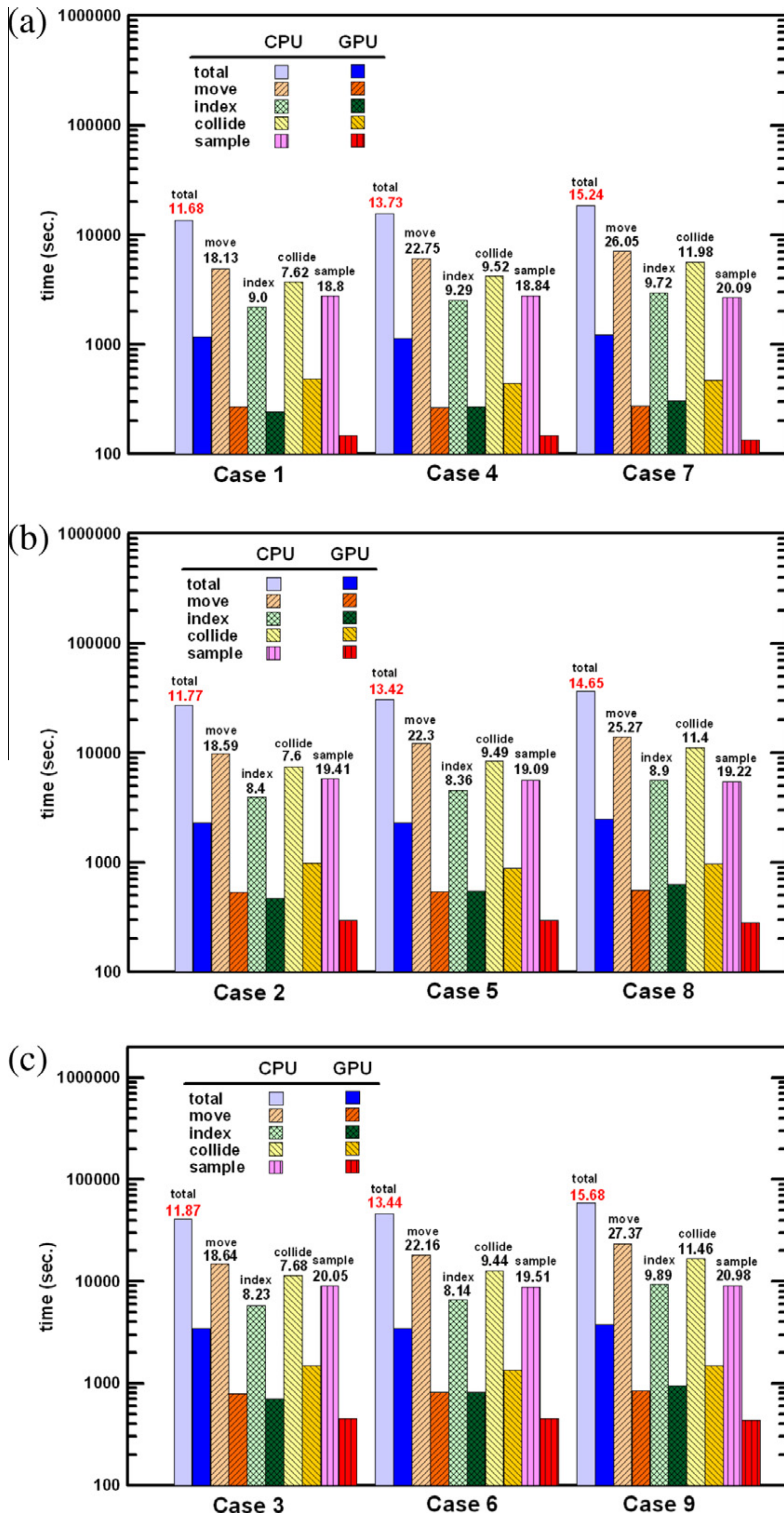


Fig. 15. Computational time and speedup$_{CPU/GPU}$ ratio using a core of the CPU (Intel Xeon X5670) and a GPU (NVIDIA Tesla M2070) for each component of DSMC in supersonic lid-driven cavity problem: (a) 10 M particles, (b) 20 M particles, and (c) 30 M particles.

3.1.2. Parallel performance using multiple GPUs in strong scaling and weak scaling

Strong scaling is defined as how the simulation time varies with the number of processors for a fixed total problem size. Weak scaling is defined as how the simulation time varies with the number of processors for a fixed problem size per processor. In the case of strong scaling, the parallel performance is presented for the subsonic lid-driven cavity with the simulation conditions shown Table 1. Fig. 11 shows the variation in speedup ratio of the fixed total problems with increasing numbers of GPU devices. In Fig. 11 (a), the result shows that up to 185.28 times of speedup can be reached using 16 GPUs (NVIDIA Tesla M2070) for the densest case with 30M particles as compared to a single core of Intel Xeon X5670 CPU. When compared to the performance using a single M2070 GPU, the ratio of speedup of 12 times (see Fig. 11 (b)) demonstrates approximately 75% parallelization efficiency. Again, larger speedup is obtained with decreasing rarefaction. We would expect a much better speedup at 16 GPUs if a much larger problem is simulated.

The balance of computational load between GPU's on different nodes strongly influences the performance of our proposed hybrid method. It is shown earlier in Figs. 9 and 10 that the simulation time of the DSMC method is proportional to the total number of the particles in a single GPU. Thus, we define a non-uniformity (UN) parameter to describe situation of computing load balance as

$$UN = \frac{P_{\max} - P_{\min}}{P_{\text{average}}} \times 100\% \tag{2}$$

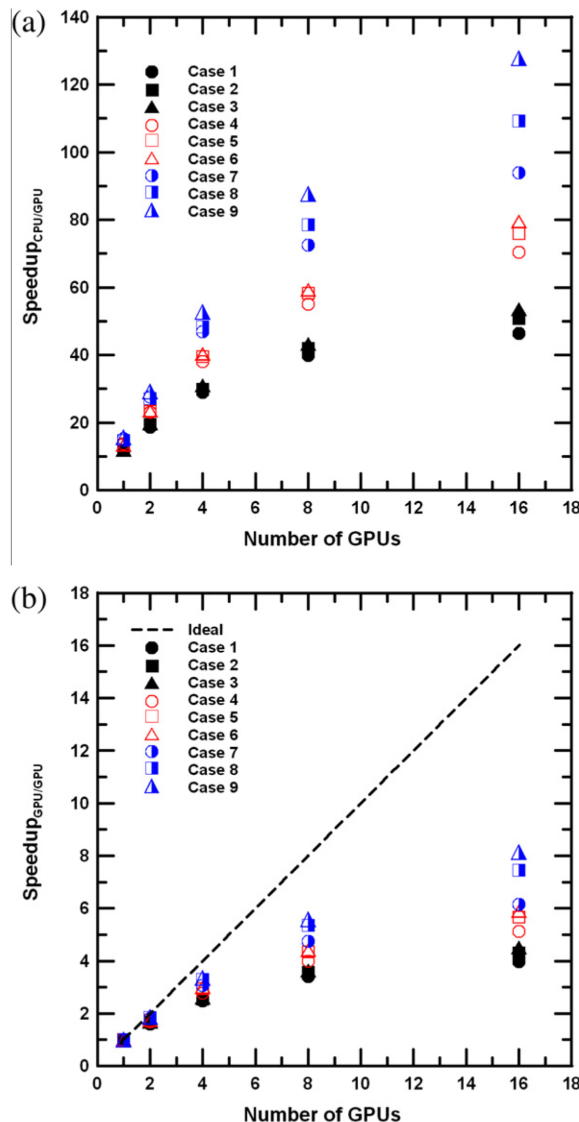


Fig. 16. Speedup ratio as a function of GPU (M2070) number in strong scaling for supersonic lid-driven cavity problem: (a) compared to a core of the CPU (Intel Xeon X5670), and (b) compared to a single GPU (M2070).

where P_{max} and P_{min} are the maximum and minimum number of particles across the GPU devices, respectively, and $P_{average}$ is the average number of particles (= total number of particles/number of GPUs). Therefore, when the value of *UN* is close to "zero", there is a lower variation in the number of particles from GPU to GPU. The computed non-uniformity (UN) and speed-up ratio are presented in Table 2, showing that the computational load across the GPU devices is relatively even.

For weak scaling, we present the parallel performance for a fixed number of particles per processor as shown in Table 3. In this test, there are nine cases ranging from Kn = 0.01 (Case I–III) to Kn = 0.002 (Case VII–IX) with various number of particles (5 M, 10 M, 15 M) per processor at some specific Kn; therefore, there are 80 million to 240 million particles when 16 GPUs are used. The computational time for all cases in the case of weak scaling is shown in Fig. 12. The results show that parallel performance is better (or more scalable) when nearing continuum conditions (i.e. Kn = 0.002) than in more rarefied conditions (Kn = 0.01 and Kn = 0.005). We also demonstrate relatively uniform loading across GPU devices (see Table 3).

3.2. Supersonic Lid-Driven Cavity Flow

This test case is similar to the previously investigated lid-driven cavity problem, with the exception of the speed of the upper moving wall ($M = 2$). The purpose is to understand how the load imbalance will influence the parallel performance. Fig. 13 shows computational time per particle and per timestep with different combinations of block and thread per block numbers using a single GPU (M2070) for case 5. As per the previous (subsonic) investigation, the optimal number of blocks and threads per block is 16,384 and 256 respectively. This configuration is thus employed for all of the test cases presented next.

3.2.1. Timing breakdown for a single GPU

Figs. 14 and 15 show the computational time required to execute 5000 timesteps and the corresponding speedup$_{CPU/GPU}$ ratio using a core of the CPU (Intel Xeon 5670) and a GPU (NVIDIA Tesla M2070) for each case. In addition to previous analysis, we compute the computational expense required for each component of the DSMC simulation. The results show that similar speedups are obtained as compared to those of subsonic cases, although slightly lower speedups are obtained for more rarefied cases. In addition, they again show that our proposed DSMC simulation with CUDA has two key bottlenecks, being the indexing phase and collision phase. The discussion about the bottlenecks was presented in Section 3.1.1.

3.2.2. Parallel performance using multiple GPUs in strong scaling and weak scaling

In strong scaling, the parallel performance is presented by the above test case (supersonic lid-driven cavity, $M = 2$) and simulation conditions summarized in Table 1. Fig. 16 shows the variation in speedup ratio of the fixed problems with increasing numbers of GPU devices. The result shows that speedup of 127.8 times can be reached using 16 GPUs (NVIDIA Tesla M2070) as compared to single core of Intel Xeon X5670 CPU in Fig. 16 (a). In this instance, the parallel performance when using 16 GPU devices is only 51% (speedup of 8.15 times when compared to a single GPU), as shown in Fig. 16(b). A possible explanation for the low level of performance is the level of imbalance of particles across the GPU devices – this is sensible since the supersonic lid-driven cavity results in large density variations across the simulation domain – thus demonstrating the weaknesses associated with a static domain decomposition as proposed here, which requires further improvement in the future. The speedup ratio and non-uniformity (UN) in this test are presented in Table 4. Results (UN, in Table 4) demonstrate that the parallel performance on multiple GPUs in this case is dominated by computing load for each GPU. Fig. 17 shows the computational expense for the supersonic lid-driven cavity problem with a fixed number of particles

Table 4
Non-uniformity (UN (%)) and speedup ratio ($S_{C/G}$ is speedup$_{CPU/GPU}$ (compared to a core of the Intel Xeon X5670 CPU), and $S_{G/G}$ is speedup$_{GPU/GPU}$ (compared to a single Tesla M2070 GPU)) in strong scaling for supersonic lid-driven cavity problem.

	GPUs	$S_{C/G}$	$S_{G/G}$	UN (%)		GPUs	$S_{C/G}$	$S_{G/G}$	UN (%)		GPUs	$S_{C/G}$	$S_{G/G}$	UN (%)
Case 1	1	11.68	1.00	0.00	Case 4	1	13.73	1.00	0.00	Case 7	1	15.24	1.00	0.00
	2	18.70	1.60	23.44		2	23.08	1.68	18.75		2	27.65	1.81	6.51
	4	28.88	2.47	32.96		4	37.99	2.77	23.91		4	46.85	3.07	19.97
	8	39.78	3.41	39.19		8	54.97	4.00	31.12		8	72.46	4.75	32.18
	16	46.38	3.97	49.92		16	70.45	5.13	35.20		16	93.96	6.16	42.01
Case 2	1	11.77	1.00	0.00	Case 5	1	13.42	1.00	0.00	Case 8	1	14.65	1.00	0.00
	2	19.37	1.65	23.48		2	23.34	1.74	18.61		2	27.18	1.85	6.48
	4	29.89	2.54	32.82		4	39.50	2.94	23.74		4	48.37	3.30	19.32
	8	42.02	3.57	39.42		8	58.34	4.35	30.93		8	78.56	5.36	32.12
	16	50.73	4.31	50.01		16	75.93	5.66	34.97		16	109.30	7.46	42.30
Case 3	1	11.87	1.00	0.00	Case 6	1	13.44	1.00	0.00	Case 9	1	15.68	1.00	0.00
	2	19.84	1.67	23.46		2	23.44	1.74	18.59		2	29.22	1.86	6.49
	4	30.97	2.61	32.89		4	40.28	3.00	23.87		4	52.83	3.37	19.05
	8	43.25	3.64	39.31		8	59.30	4.41	30.99		8	87.69	5.59	31.96
	16	53.60	4.51	50.04		16	79.44	5.91	35.28		16	127.80	8.15	42.25

per processor (i.e. weak scaling) with the results specifically listed in Table 5 that is similar to Table 3 for the subsonic case. As in the strong scaling tests, we clearly see the poor level of load balancing across multiple GPU devices, resulting in relatively low levels of performance as some GPU devices are forced to wait for other heavily loaded GPU devices to complete their computation cycles.

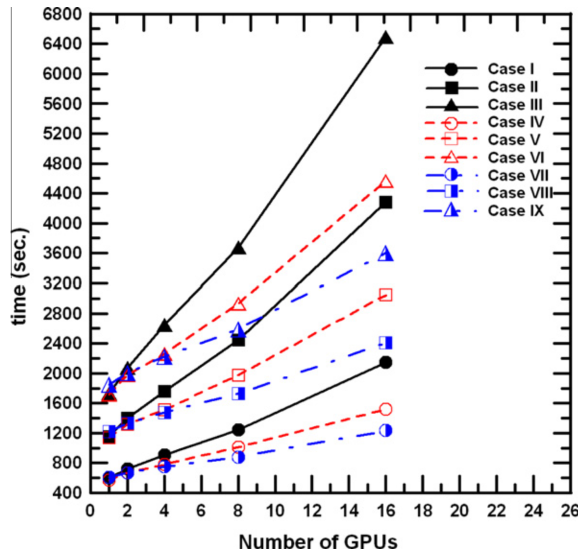


Fig. 17. Computational time (units: second) as a function of the number of GPU devices for weak scaling for the supersonic lid-driven cavity problem.

Table 5
Simulation conditions and non-uniformity (UN (%)) in weak scaling for supersonic lid-driven cavity problem.

	Kn	GPUs	Total particle number	UN (%)		Kn	GPUs	Total particle number	UN (%)		Kn	GPUs	Total particle number	UN (%)
Case I	0.01	1	5 M	0.00	Case IV	0.005	1	5 M	0.00	Case VII	0.002	1	5 M	0.00
		2	10 M	23.49			2	10 M	18.37			2	10 M	6.55
		4	20 M	32.88			4	20 M	23.75			4	20 M	19.08
		8	40 M	39.33			8	40 M	30.87			8	40 M	32.03
		16	80 M	50.04			16	80 M	35.22			16	80 M	42.42
Case II	0.01	1	10 M	0.00	Case V	0.005	1	10 M	0.00	Case VIII	0.002	1	10 M	0.00
		2	20 M	23.53			2	20 M	18.57			2	20 M	9.81
		4	40 M	32.89			4	40 M	23.79			4	40 M	18.87
		8	80 M	39.27			8	80 M	30.85			8	80 M	31.94
		16	160 M	50.05			16	160 M	35.13			16	160 M	42.19
Case III	0.01	1	15 M	0.00	Case VI	0.005	1	15 M	0.00	Case IX	0.002	1	15 M	0.00
		2	30 M	23.50			2	30 M	18.44			2	30 M	6.54
		4	60 M	32.76			4	60 M	23.76			4	60 M	18.86
		8	120 M	39.27			8	120 M	30.98			8	120 M	31.96
		16	240 M	49.94			16	240 M	35.20			16	240 M	42.27

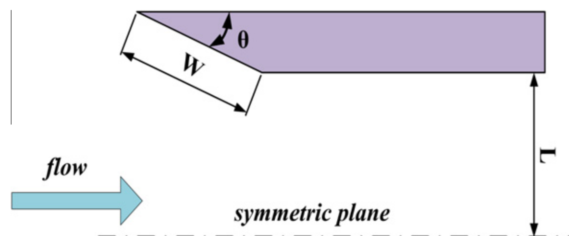


Fig. 18. Sketch of the Mach reflection problem.

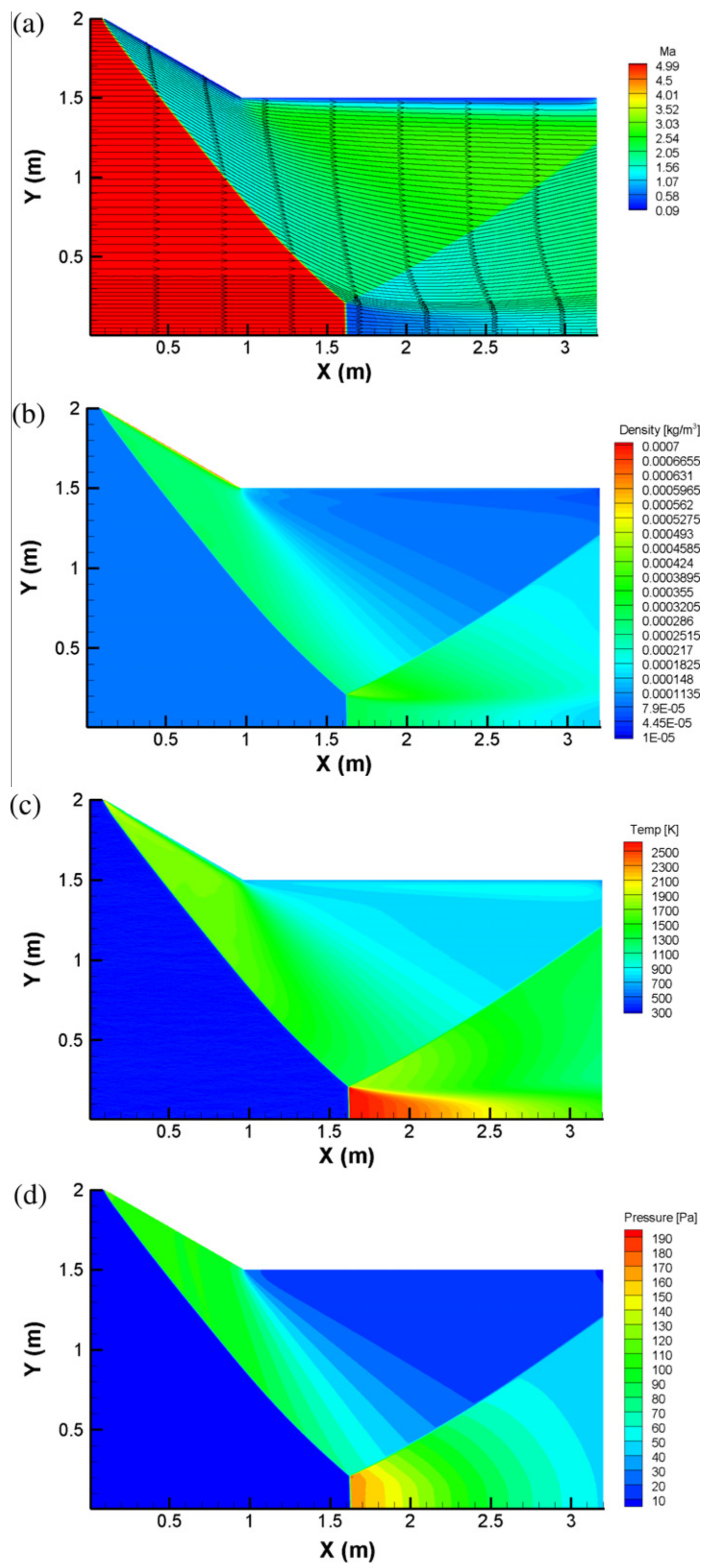


Fig. 19. Contours of properties of Mach reflection problem: (a) Mach number and streamline, (b) density, (c) temperature, and (d) pressure.

3.3. Large-scale simulations in near-continuum flow

To demonstrate the capability of the current parallel DSMC method using MPI-CUDA parallelization paradigm using multiple GPU devices, we have selected several very large-scale simulations in near-continuum flow. These include: (i) Mach reflection problem, (ii) hypersonic flow over a wedge problem, and (iii) supersonic over a square block problem. These simulations are performed across several different kinds and generations of GPU devices including NVIDIA Tesla M2070, NVIDIA Telsa C1060 (240 CUDA cores) and low-cost GeForce GTX 590 (2 GPU chips on board, 1024 CUDA cores in total) to demonstrate flexibility of the current implementation.

3.3.1. Mach reflection problem

The Mach reflection problem involves the high-speed flow ($M = 5.0$) between two symmetrical wedges with angle θ (=30 degree) and L/W (=1.5) in Fig. 18. Since the problem is symmetric, we only simulate the upper half of the problem. The initial conditions are argon gas at temperature 300 K with a uniform flow speed of 1612.45 m/s. The initial number density is 1.2944×10^{21} particles/m^3 and the wall temperature is fixed at 600 K – these conditions correspond to a free-stream

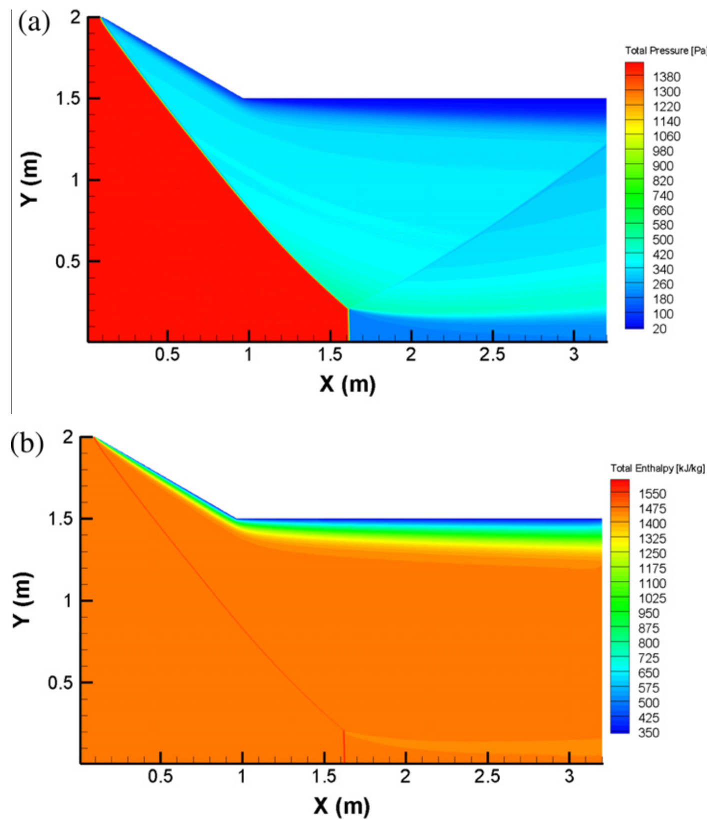


Fig. 20. Contours of properties of Mach reflection problem: (a) total pressure, and (b) total enthalpy.

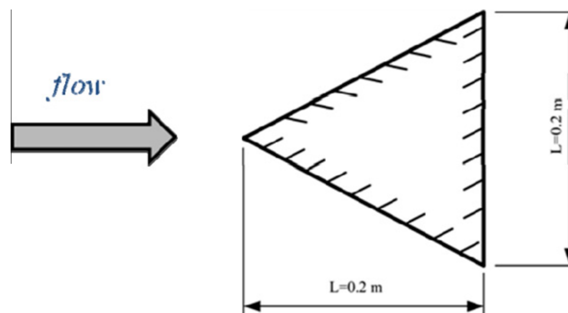


Fig. 21. Sketch of the hypersonic flow over a wedge.

Knudsen number of 0.001 based on the free-stream mean free path (λ_∞ = 0.001 m) and the length (W = 1 m). Results show that approximately 21.81 hours are required for 120,000 simulation time steps with (after 120,000 timesteps) approximately 255 million particles and 6.4 million cells using 16 GPU devices (NVIDIA Tesla M2070). Note the timestep is 3.74×10^{-7} s and the grid is uniform with cell size of roughly the free-stream mean free path.

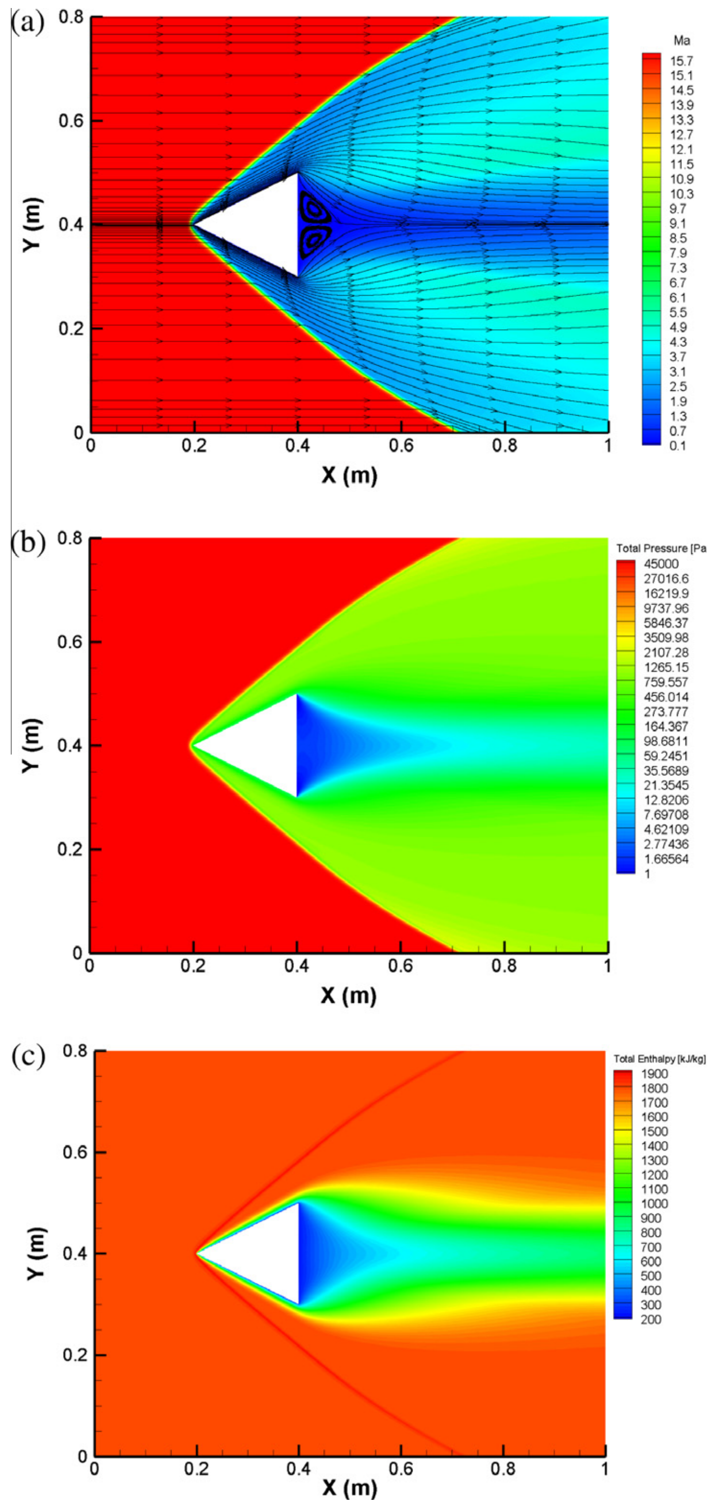


Fig. 22. Contours of properties of hypersonic flow over a wedge: (a) Mach number and streamline, (b) total pressure, and (c) total enthalpy.

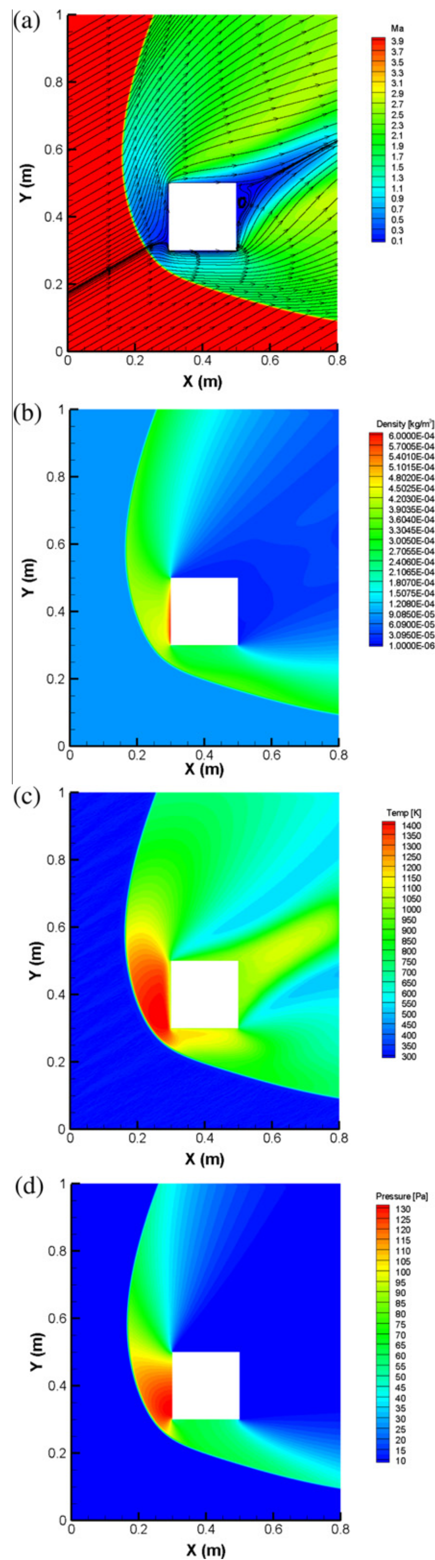


Fig. 23. Contours of properties of supersonic flow over a block: (a) Mach number and streamline, (b) density, (c) temperature, and (d) pressure.

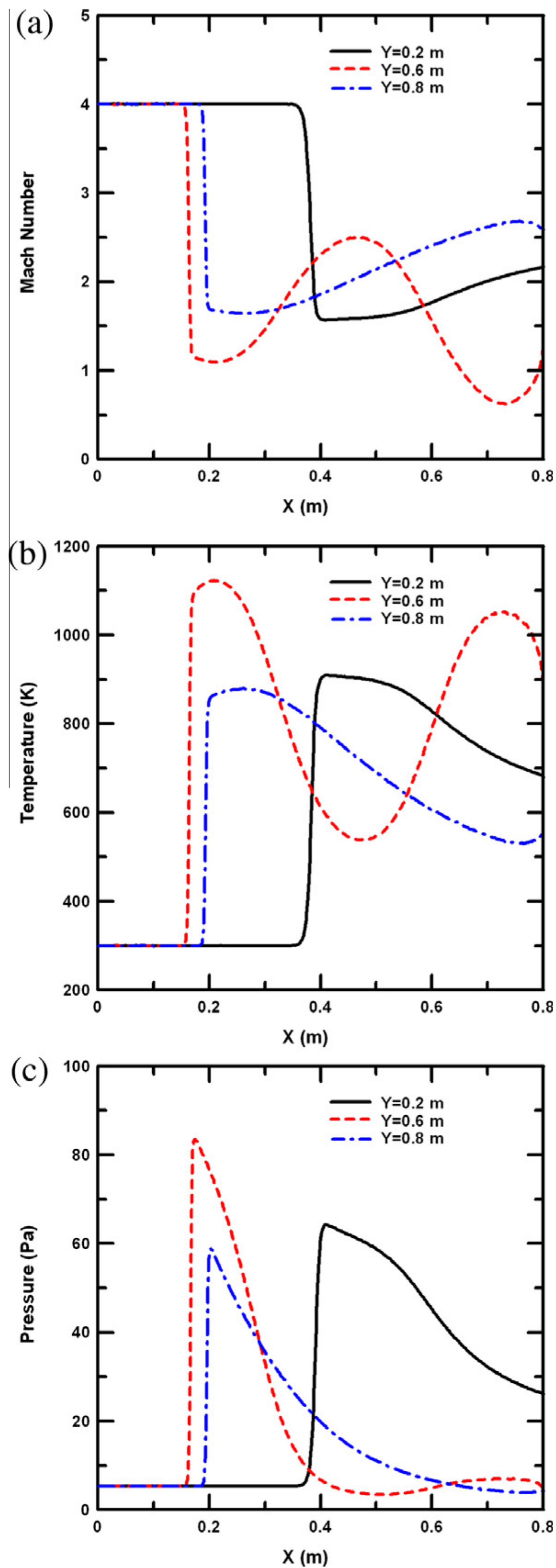


Fig. 24. Property distributions along horizontal lines (y = 0.2, 0.6 and 0.8 m): (a) Mach number, (b) temperature, and (c) pressure.

Table 6
Comparison of the performance and the computational time using various classes of GPU devices for the supersonic flow past a square block (80,000 time steps).

	Time (sec.)	speedup	Increased efficiency (%)
Tesla C1060	25,276	1	0
Tesla M2070	19,625	1.29	22
GeForce GTX 590	16,812	1.5	33.5

Fig. 19 shows the contours of Mach number, streamlines, density and temperature after a flowtime of 0.04488 s. The results show the expected trends – an oblique shock is formed on the leading edge of the wedge which, after contact with the plane of symmetry, forms a normal shock wave, forming a triple point. Conditions immediately behind the normal shock wave are subsonic, which quickly accelerate to supersonic conditions as the flow moves downstream. Thus, the resulting flow behaves like a "virtual nozzle". Fig. 20 shows contours of total pressure and total enthalpy, showing the energy drop past the incident shock wave and in the boundary layer region. On the other hand, we can observed that the overall structure of Mach reflection is captured without any doubt, although the cell size in this simulation is one mean free path based on the free-stream. In addition, we have found that the grid resolution for the shock (e.g., close to the triple point) is about 5–7 cells because of rarefaction, which is good enough for resolving the shock structure.

3.3.2. Hypersonic flow over a wedge

Fig. 21 shows the layout of a hypersonic flow past a wedge used in the simulations discussed below. The free-stream flow conditions consist of argon gas moving with a speed of 1884.12 m/s, a temperature of 40 K and a number density of 1.2944×10^{21} particles/m^3. These correspond to a Mach number of 16 and a free-stream Knudsen number of 0.005 based on the free-stream mean free path ($\lambda_\infty = 0.001$ m) and the length (L = 0.2 m). The wall temperature of the wedge is held constant at 300 K. In this test, the timestep size is 2.2×10^{-7} s and the grid is uniform with cell size $\sim$1/2 of the free-stream mean free path. Results show that only approximately 7.24 hours are required for 50,000 time steps of simulation with using (after 50,000 steps) 170 million particles across 3.2 million cells when executed on 16 NVIDIA Tesla C1060 GPU devices across 4 nodes. Each node has dual Intel Xeon X5472 (3 GHz, 12 M Cache, 1600 MHz FSB), 4 GPU devices of NVIDIA Tesla C1060 (240 microprocessors @ 1.3 GHz, 4 GB DDR3 memory, without ECC support) and 32GB RDIMM system memory. The interconnection between all nodes is InfiniBand.

Fig. 22 shows contours of Mach number, streamlines, total pressure and total enthalpy. As expected, leading oblique shock waves are formed from the leading edges of the wedge, with a boundary layer formed on its surface post-shock. Separation of flow occurs behind the wedge (see Fig. 22 (a)). The energy drops across the flow field for this problem are reflected in Fig. 22 (b) and Fig. 22 (c).

3.3.3. Supersonic flow over a block

The final test case is a two dimensional supersonic flow over a rectangular block. Ideal argon with a temperature of 300 K is initially assumed to be moving with Mach number $M = 4.0$ and with an angle of attack 30 degree over a diffusely reflecting block of fixed temperature 1000 K. The block is square with sides of length L = 0.2 m. The corresponding free-stream Knudsen number is 0.005 based on the free-stream mean free path ($\lambda_\infty = 0.001$ m) and the length ($L = 0.2$ m). Here, the timestep size is 2.12×10^{-7} s and the grid is uniform with cell size $\sim$1/2 of the free-stream mean free path. Results show that approximately 4.67 hours are required for 80,000 time steps of simulation with (after 80,000 steps) approximately 65 million particles and 3.2 million cells. We have also performed a cell-refinement study using 12.8 million cells and 200 million particles. The results are essentially the same as the case with 3.2 million cells. This simulation was executed on a very low-cost system, which includes 6x NVIDIA GeForce GTX 590's across 3 nodes. Each node has 1x AMD Phenom II X6 1090T (3.2 GHz, 6 M Cache), 2 NVIDIA GTX 590 GPU devices and 8 GB UDIMM system memory. Each GTX590 has dual GF110 chips including 1.5 GB DDR5 memory per each, totally 3 GB and 1024 CUDA cores. The interconnection between all nodes is Gigabit Ethernet. Fig. 23 shows contours of Mach number, streamline, density, temperature and pressure. The results show a clear bow shock predicted in front of the block as expected. For a future comparison of the data, Fig. 24 shows several property distributions along several horizontal lines at y = 0.2, 0.6, and 0.8 m, including Mach number, temperature and pressure.

In addition, the comparison of the performance and the computational time using various classes of GPU devices (NVIDIA Tesla C1060 and M2070) in this test case is presented in Table 6. The performance using NVIDIA GeForce GTX 590 cards is among the best, and its efficiency can increase about 15% as compared to that of NVIDIA Tesla M2070. The results show that large-scale DSMC simulation is feasible using general gaming cards in a low-cost system.

4. Conclusions

In this paper we present a parallel DSMC method using MPI-CUDA parallelization paradigm using multiple Graphics Processing Units (GPUs) capable of very large-scale simulations in near-continuum conditions. The results demonstrate that a speedup of 15 times using a single GPU (NVIDIA Tesla M2070) when compared to single core of an Intel Xeon X5670 CPU can

be obtained. Computational time scales almost linearly with number of particles on a single GPU. In the instance of multiple GPU devices and strong scaling, we have shown the computational time can be reduced by 185 times when using 16 GPU devices (NVIDIA Tesla M2070). When the scalability across multiple GPU devices was investigated, we found 75% parallelization efficiency for a problem of fixed size (i.e., strong scaling). The performance characteristics of the parallel GPU implementation are dependent on the degree of rarefaction of the flow, with increased parallel speedup encountered with a decrease in the Knudsen number. The key bottlenecks identified in this study (with regard to the DSMC computation itself) where (i) particle indexing into cells, and (ii) particle partner selection and collision phase. A significant challenge demonstrated was the particle imbalance across several GPU devices when using a domain decomposition paradigm. To demonstrate the capacity of this implementation to perform large-scale near continuum DSMC simulation, a large number of computational benchmarks were performed. We performed these tests using various GPU devices – from professional GPU computing devices (NVIDIA's Tesla series) to devices traditionally employed for computing gaming (NVIDIA GeForce series). In one instance, a benchmark simulation employing approximately 255 million particles across 6.4 million cells required approximately 21 hours to complete 120,000 time steps.

In the future, we aim to continue the optimization of the DSMC algorithm for application to the multiple GPU parallelization paradigm. Investigations into the indexing and collision phases, in addition to inclusion of the sub-cell method [14] and variable time step scheme [15] are planned in the immediate future. In order to improve the computational load balancing across multiple GPU devices we will implement a dynamic domain decomposition method [16]. Finally, we will extend our implementation to incorporate three-dimensional flows to increase the applicability to very large scale, real life flow problems in the near-continuum and possibly continuum regime. For those who are interested in this research, we have provided a source code of two-dimensional DSMC implemented on a single GPU in our group website [17].

Code Snippet 1. A GPU-based function (code) for the creation of uniformly distributed random numbers.

```
class RANDOM{
  public:
    int inext , inextp , iff , ma[56] ;
    RANDOM() ;
} ;
RANDOM::RANDOM(){
  inext = 0 ;
  inextp = 0 ;
  iff = 0 ;
  for (int i = 0 ; i<56 ; i++)
    ma[i] = 0 ;
}
__device__ float Randn(RANDOM *Rand , int index_n){
  int i , j ;
  float rf ;
  const int MBIG = 1000000000 ;
  const int MSEED = 161803398 ;
  const int MZ = 0 ;
  const float FAC = 1.E-9 ;
  int MJ , MK , II ;
  if (index_n < 0 || Rand->iff == 0){
    Rand->iff = 1 ;
    MJ = MSEED - index_n ;
    MJ = MJ%MBIG ;
    Rand->ma [55] = MJ ;
    MK = 1 ;
    for (i = 1; i<55; i++){
      II = 21*i%55 ;
      Rand->ma[II] = MK ;
      MK = MJ-MK ;
      if (MK < MZ) MK = MK + MBIG ;
      MJ = Rand->ma[II] ;
    }
    for (j = 0; j < 4; j++){
      for (i = 1; i < 56; i++){
```

(continued on next page)

```
            Rand->ma[i] = Rand->ma[i] - Rand->ma[1 + (i + 30)%55] ;
            if (Rand->ma[i] < MZ) Rand->ma[i] = Rand->ma[i] + MBIG ;
          }
        }
        Rand->inext = 0 ; Rand->inextp = 31 ;
      }
      do{
        Rand->inext = Rand->inext + 1;
        if (Rand->inext == 56) Rand->inext = 1;
        Rand->inextp = Rand->inextp+1;
        if (Rand->inextp == 56) Rand->inextp = 1;
        MJ = Rand->ma[Rand->inext] - Rand->ma[Rand->inextp];
        if (MJ < MZ) MJ = MJ + MBIG;
        Rand->ma[Rand->inext] = MJ;
        rf = MJ*FAC;
      }while (!(rf > 1.E-8 && rf < 0.99999999));
      return rf;
    }
```

Algorithm 1. Calculate positions of all particles over a time step.

Require: bidx is the index of this block (*blockIdx.x*)
Require: tidx is the index of this thread within the block (*threadIdx.x*)
Require: bdimx is the number of blocks (*gridDim.x*)
Require: tdimx is the number of threads in each block (*blockDim.x*)
Require: d_ParticleIn is initialized to “1” from 0 to $N_{particles} - 1$
Require: d_ParticleInMPI is initialized to “0”
for i = bidx $*$ tdim+tidx to $N_{particles}$-1 ; i+ = bdimx $*$ tdimx **do**
  ParticleCoordinate[i]+ = ParticleVelocity[i] $*$ timestep
  if particle collide boundary of GPU’s domain
    d_ParticleInMPI[i] = 1
    d_ParticleIn[i] = 0
    continue
  else if particle collide inlet/outlet boundary
    d_ParticleIn[i] = 0
    continue
  else if particle collide wall boundary
    particle be rebounded based on type of wall
    continue
  end if
  calculate cell number of particle
end for

Algorithm 2. Sum number of particles in each cell.

Require: bidx is the index of this block (*blockIdx.x*)
Require: tidx is the index of this thread within the block (*threadIdx.x*)
Require: bdimx is the number of blocks (*gridDim.x*)
Require: tdimx is the number of threads in each block (*blockDim.x*)
Require: d_IndexCell2 is initialized to “0”
for i = bidx $*$ tdim+tidx to $N_{particles} - 1$; i+ = bdimx $*$ tdimx **do**
  CellNo = d_ParticleCellNo[i]
  atomicAdd(&d_IndexCell2[CellNo], 1)
end for

Algorithm 3. Index the relationship between particles and cells.

Require: bidx is the index of this block (*blockIdx.x*)
Require: tidx is the index of this thread within the block (*threadIdx.x*)
Require: bdimx is the number of blocks (*gridDim.x*)
Require: tdimx is the number of threads in each block (*blockDim.x*)
Require: d_IndexCell2 is initialized to "0"
for i = bidx $*$ tdim + tidx to $N_{particles} - 1$; i+ = bdimx$*$tdimx **do**
  CellNo = d_ParticleCellNo[i]
  j = **atomicAdd**(&d_IndexCell2[CellNo], 1)
  k = d_IndexCell1[CellNo] + j
  d_IndexParicle[k] = i
end for

Algorithm 4. Collision implementation with CUDA in DSMC.

Require: bidx is the index of this block (*blockIdx.x*)
Require: tidx is the index of this thread within the block (*threadIdx.x*)
Require: bdimx is the number of blocks (*gridDim.x*)
Require: tdimx is the number of threads in each block (*blockDim.x*)
for i = bidx $*$ tdim+tidx to $N_{cells} - 1$; i + = bdimx $*$ tdimx **do**
  collision pairs is calculated by NTC method
  for j = 0 to $M_{collision\ pairs} - 1$ **do**
    select colliding particle and to calculate relative speed$*$collision cross section ($\sigma_T * c_r$)
    if Rand[0 − 1] < $(\sigma_T * c_r)/(\sigma_T * c_r)_{max}$
      calculate velocity post-collided
    end if
  end for
  maximum relative speed$*$collision cross section $(\sigma_T * c_r)_{max}$ is updated in this cell
end for

Algorithm 5. Sampling implementation with CUDA in DSMC.

Require: bidx is the index of this block (*blockIdx.x*)
Require: tidx is the index of this thread within the block (*threadIdx.x*)
Require: bdimx is the number of blocks (*gridDim.x*)
Require: tdimx is the number of threads in each block (*blockDim.x*)
Require: s_sampleproperties are stored the sampling data on shared memory
Require: d_sampleproperties are stored the sampling data on global memory
for i = bidx $*$ tdim + tidx to $N_{cells} - 1$; i+ = bdimx $*$ tdimx **do**
  s_sampleproperties[tidx] = 0
  $M_{particles}$ = d_IndexCell2[i]
  m = d_IndexCell1[i]
  for j = 0 to $M_{particles} - 1$ **do**
    k = d_IndexParticle[m + j]
    s_sampleproperties[tidx] + = d_ParticleProperties[k]
  end for
  d_SampleParticleNum[tidx] + = $M_{particles}$
  d_sampleproperties[tidx] + = s_sampleproperties[tidx]
end for

Acknowledgments

Computing resources provided by National Center for High-Performance Computing of Taiwan is highly appreciated. The authors also would like to acknowledge support from the National Science Council (NSC) of Taiwan through Grant NSC99-2221-E-009-056-MY2 and NSC100-2627-E-009-001, and the National Space Organization (NSPO) of Taiwan through Grant NSPO-S-099128.

References

[1] G.A. Bird, Molecular Gas Dynamics and the Direct Simulation of Gas Flows, Clarendon Press, Oxford, 1994.
[2] C. Cercignani, The Boltzmann Equation and Its Applications, Springer, New York, 1988.
[3] W. Wagner, A convergence proof for Bird's direct simulation Monte Carlo method for the Boltzmann equation, J. Stat. Phys. 66 (1992) 1011–1044.
[4] S. Dietrich, I.D. Boyd, Scalar and parallel optimized implementation of the direct simulation Monte Carlo method, J. Comput. Phys. 126 (1996) 328–342.
[5] M. Ivanov, G. Markelov, S. Taylor, J. Watts, Parallel DSMC strategies for 3D computations, in: Proceedings of the Parallel CFD'96, Capri, Italy, 1997, pp. 485.
[6] G.J. LeBeau, A parallel implementation of the direct simulation Monte Carlo method, Comput. Methods Appl. Mech. Eng. 174 (1999) 319–337.
[7] J.-S. Wu, Y.-Y. Lian, Parallel three-dimensional direct simulation Monte Carlo method and its applications, Comput. Fluids 32 (2003) 1133–1160.
[8] D. Gladkov, J.J. Tapia, S. Alberts, R.M. D'Souza, Graphics processing unit based direct simulation Monte Carlo, Simulation published, 26 Sept. 2011.
[9] C.-C. Su, C.-W. Hsieh, M.R. Smith, M.C. Jermy, J.-S. Wu, Parallel direct simulation Monte Carlo computation using CUDA on GPUs, in: Proceedings of the 27th International Symposium on Rarefied Gas Dynamics, Pacific Grove, California, USA, July 2010.
[10] GPGPU, http://www.gpgpu.org.
[11] NVIDIA Crop., NVIDIA Compute Unified Device Architecture Programming Guide Version 1.0, 2007.
[12] NVIDIA Crop. NVIDIA CUDA C Programming Guide Version 4.0, 2011.
[13] M. Harris, S. Sengupta, J.D. Owens, Parallel Prefix Sum (Scan) with CUDA, in: GPU Gems 3, H. Nguyen (Ed.), Addison Wesley, 2007, pp. 851–876.
[14] C.-C. Su, K.-C. Tseng, H.M. Cave, J.-S. Wu, Y.-Y. Lian, T.-C. Kuo, M.C. Jermy, Implementation of a transient adaptive sub-cell module for the parallel-DSMC code using unstructured grids, Comput. Fluids 39 (2010) 1136–1145.
[15] J.-S. Wu, K.-C. Tseng, F.-Y. Wu, Parallel three-dimensional DSMC method using mesh refinement and variable time-step scheme, Comput. Phys. Comm. 162 (2004) 166–187.
[16] C.D. Robinson, Particle simulation on parallel computers with dynamic load balancing, Ph.D. Thesis, Imperial College of Science, Technology and Medicine, UK, 1998.
[17] http://www.appltw.org/downloads.