Hindawi
Scientific Programming
Volume 2020, Article ID 8862123, 15 pages
https://doi.org/10.1155/2020/8862123

Research Article

Hybrid MPI and CUDA Parallelization for CFD Applications on Multi-GPU HPC Clusters

Jianqi Lai, Hang Yu, Zhengyu Tian, and Hua Li

College of Aerospace Science and Engineering, National University of Defense Technology, Changsha 410073, China

Correspondence should be addressed to Jianqi Lai; laijianqi_kd@nudt.edu.cn

Received 17 July 2020; Revised 18 August 2020; Accepted 11 September 2020; Published 25 September 2020

Academic Editor: Antonio J. Peña

Copyright © 2020 Jianqi Lai et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Graphics processing units (GPUs) have a strong floating-point capability and a high memory bandwidth in data parallelism and have been widely used in high-performance computing (HPC). Compute unified device architecture (CUDA) is used as a parallel computing platform and programming model for the GPU to reduce the complexity of programming. The programmable GPUs are becoming popular in computational fluid dynamics (CFD) applications. In this work, we propose a hybrid parallel algorithm of the message passing interface and CUDA for CFD applications on multi-GPU HPC clusters. The AUSM + UP upwind scheme and the three-step Runge–Kutta method are used for spatial discretization and time discretization, respectively. The turbulent solution is solved by the $K-\omega$ SST two-equation model. The CPU only manages the execution of the GPU and communication, and the GPU is responsible for data processing. Parallel execution and memory access optimizations are used to optimize the GPU-based CFD codes. We propose a nonblocking communication method to fully overlap GPU computing, CPU_CPU communication, and CPU_GPU data transfer by creating two CUDA streams. Furthermore, the one-dimensional domain decomposition method is used to balance the workload among GPUs. Finally, we evaluate the hybrid parallel algorithm with the compressible turbulent flow over a flat plate. The performance of a single GPU implementation and the scalability of multi-GPU clusters are discussed. Performance measurements show that multi-GPU parallelization can achieve a speedup of more than 36 times with respect to CPU-based parallel computing, and the parallel algorithm has good scalability.

1. Introduction

The developments of computer technology and numerical schemes over the past few decades have made computational fluid dynamics (CFD) become an important tool in optimal design of aircraft and analysis of a complex flow mechanism [1, 2]. A large number of CFD applications can reduce development costs and provide technical support for research on aircraft. The scope and complexity of flow problems in CFD simulation is constantly expanding, and the grid size required for simulation is increasing. The rapid increase in grid size raises the challenge in processing these huge data on processors in engineering activities and scientific research. Traditionally, multi-CPU parallelization has been used to accelerate computation. The low parallelism degree and power inefficiency may limit the parallel performance of the cluster. Furthermore, the computing time is largely dependent on the CPU update. In recent years, the development of the CPU has been a bottleneck due to limitations in power consumption and heat dissipation prevention [3, 4].

In CFD applications, a large amount of computing resources are required for complex flow problems, such as turbulent flow, reactive flow, and multiphase flow. High-performance computing (HPC) platforms, such as graphics processing unit (GPU), many integrated core (MIC), and field programmable gate array (FPGA), exhibit a more efficient performance in parallel data processing than the CPU [5–8]. Faster and better numerical solutions can be obtained by executing CFD codes on these heterogeneous accelerators. In this paper, we discuss GPU parallelization in CFD applications.

GPU has a strong floating-point capability and a high memory bandwidth in data parallelism. The latest Volta

architecture Tesla V100 GPU has single-precision and double-precision floating-point operations up to 14 and 7 TFLOP/s, respectively, which are much higher than the computing performance of the CPU. In 2006, NVIDIA introduced the compute unified device architecture (CUDA), which reduces the complexity of programming [9]. The programmable GPU has evolved into a highly parallel, multithreaded, many-core processor. Therefore, GPU acceleration is becoming popular in general-purpose computing areas such as molecular dynamics (MD), direct simulation Monte Carlo (DSMC), CFD, artificial intelligence (AI), and deep learning (DL) [10–13].

In this work, we focus on the design and optimizations of a hybrid parallel algorithm of the message passing interface (MPI) and CUDA for CFD applications on multi-GPU HPC clusters. The compressible Navier–Stokes equations are discretized based on the cell-centered finite volume method. The AUSM + UP upwind scheme and the three-step Runge–Kutta method are used for spatial discretization and time discretization, respectively. Moreover, the turbulent solution is solved by the $K - \omega$ shear stress transport (SST) two-equation model. In some previous work, the CPU was designed to perform data processing together with the GPU [14, 15]. However, for the latest Pascal or Volta Architecture GPU, the computing ability of the GPU far exceeds that of the CPU. In our design of the hybrid parallel algorithm, the CPU only manages the execution of the GPU and communication, and the GPU is responsible for data processing. Performance optimization involves three basic strategies: maximizing parallel execution to achieve maximum utilization, optimizing memory usage to achieve maximum memory throughput, and optimizing instruction usage to achieve maximum instruction throughput. In this study, parallel execution and memory access optimizations are investigated.

In a multi-GPU HPC cluster, ghost and singularity data are exchanged between GPUs. Some scholars use CUDA-aware MPI technology to accelerate the speed of data exchange [13, 16–18]. This technology not only makes it easier to work with a CUDA + MPI application, but also makes acceleration technologies like GPUDirect be utilized by the MPI library. However, the current hardware and software configurations we used in this paper do not support this technology. Moreover, the use of the $K - \omega$ SST two-equation turbulence model increases the complexity of multi-GPU parallel programming. In our design of the hybrid parallel algorithm, we need to stage GPU buffers through host memory. Two kinds of communication approaches are considered: blocking and nonblocking communication methods. The first approach uses blocking functions *MPI_Bsend* and *MPI_Recv* without overlapping communication and computations. The second approach uses nonblocking functions *MPI_Isend* and *MPI_Irecv* with fully overlapping GPU computations, CPU_CPU communication, and CPU_GPU data transfer. The nonblocking communication method can improve computational efficiency to some extent.

Multi-GPU parallelization can achieve the maximum performance by balancing the workload among GPUs based on domain decomposition [3, 19]. The one-dimensional (1D), two-dimensional (2D), or three-dimensional (3D) domain decomposition method is commonly used for GPU implementation. In this study, we design a 1D domain decomposition algorithm based on the idea of dichotomy to load each GPU with approximately the same grid scale. Though the 1D method needs to transfer more data, the 2D or 3D method cannot achieve coalesced memory access in the global memory, which results in considerable performance loss when performing CFD applications on multi-GPU HPC clusters.

In this paper, the design and optimizations of the parallel algorithm are closely related to the hardware configurations, numerical schemes, and computational grids to obtain the optimal parallel performance on multi-GPU HPC clusters. The main contributions of this work are summarized as follows:

(i) A hybrid parallel algorithm of MPI and CUDA for CFD applications implemented on multi-GPU HPC clusters is proposed, and optimization methods are adopted to improve the computational efficiency

(ii) Considering the CFD numerical schemes the nonblocking communication mode is proposed to fully overlap GPU computing, CPU_CPU communication, and CPU_GPU data transfer by creating two CUDA streams

(iii) 1D domain decomposition method based on the idea of dichotomy is used to distribute the problem among GPUs to balance workload

(iv) The proposed algorithm is evaluated with the flat plate flow application, and the parallel performance has been analyzed in detail

The remainder of this paper is organized as follows. Section 2 discusses the related work on GPU-based parallelization and optimizations. Section 3 introduces the governing equations and numerical schemes. Section 4 describes the hybrid parallel algorithm and the optimizations in detail. Section 5 presents the performance evaluation results with the compressible turbulent flow over a flat plate. Section 6 provides the conclusion of this work and a plan for future work.

2. Related Work

In the field of CFD, GPU parallelization for CFD applications has achieved numerous remarkable results. Brandvik and Pullan [20, 21] developed 2D and 3D GPU solvers for the compressible, inviscid Euler equations. This was the first CFD application to use CUDA for the 2D and 3D solutions. Ren et al. [22] and Tran et al. [23] proposed a GPU-accelerated solver for turbulent flow based on the lattice Boltzmann method, and the solver can achieve a good acceleration performance. Khajeh-Saeed and Blair Perot [24] and Salvadore et al. [25] accomplished direct numerical simulation (DNS) of turbulence using GPU-accelerated supercomputers which demonstrated that scientific problems could benefit significantly from advanced hardware. Ma et al. [26] and Zhang et al. [27] performed GPU

computing of compressible flow problems by a meshless method. Xu et al. [14] and Cao et al. [28] described hybrid OpenMP + MPI + CUDA in parallel computing of CFD codes. The results showed that the GPU-accelerated algorithm had sustainably improved efficiency and scalability. Liu et al. [29] proposed a hybrid solution method for CFD applications on CPU + GPU platforms, and a domain decomposition method based on the functional performance model was used to guarantee a balanced workload.

The optimization techniques are used to enhance the performance of GPU acceleration. Memory access and communication are the most critical parts of performance optimization. A review of optimization techniques and the specific improvement factors for each technique is shown in [4].

In a multi-GPU HPC cluster, GPUs cannot communicate directly for data exchange. Meanwhile, the blocking communication mode is simple to implement but has low efficiency. Several researchers have designed the nonblocking communication mode for GPU parallelization to overlap computation and communication. Mininni et al. [30] used the nonblocking communication method to realize the overlap between GPU computation and CPU_CPU communication. Thibault and Senocak [31], Jacobsen et al. [32], Castonguay et al. [33], and Ma et al. [34] performed the nonblocking communication method with CUDA streams to fully overlap GPU computation, CPU_CPU communication, and CPU_GPU data transfer. Their results showed that the full coverage between computation and communication is the most efficient.

Domain decomposition is a commonly used method in parallel computing of CFD simulations to balance the workload in each processor. Jacobsen et al. [32] used 1D domain decomposition to decompose 3D structured meshes into a 1D layer. Wang et al. [35] studied the HPC of atmospheric general circulation models (ACGMS) in Earth science research on multi-CPU cores. They indicated an ACGMS model with 1D domain decomposition can only run in dozens of CPU cores. Therefore, they proposed a 2D domain decomposition parallel algorithm for this large-scale problem. Baghapour et al. [36] executed CFD codes on heterogeneous platforms, with 16 Tesla C2075 GPUs, where the solver works up to 190 times faster than a single core of a Xeon E5645 processor. They pointed out that 3D domain decomposition performs best in bandwidth-bound communication and not in latency-bound communication, in which 1D domain decomposition is preferred. Given that the GPU computing can execute many threads simultaneously and the communication between the CPU and the GPU becomes a source of high latency with highly noncontiguous data transfer, the 1D domain decomposition method is the most suitable for balancing the workload on GPUs.

3. Governing Equations and Numerical Schemes

The simulation of a compressible turbulent flow is considered. All volume sources are ignored due to body forces and volumetric heating, and the integral form of the 3D Navier–Stokes equations for a compressible, viscous, heat-conduction gas can be expressed as follows [37]:

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{W}\,\mathrm{d}\Omega + \oint_{\partial\Omega}\left(\vec{F}_c - \vec{F}_v\right)\mathrm{d}S = 0, \tag{1}$$

where $\vec{W}$ is the vector of conservative variables, $\vec{F}_c$ is the vector of convective fluxes, and $\vec{F}_v$ is the vector of viscous fluxes:

$$\begin{gathered}
\vec{W} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{bmatrix}, \\
\vec{F}_c = \begin{bmatrix} \rho V \\ \rho u V + n_x p \\ \rho v V + n_y p \\ \rho w V + n_z p \\ \rho H V \end{bmatrix}, \\
\vec{F}_v = \begin{bmatrix} 0 \\ n_x\tau_{xx} + n_y\tau_{xy} + n_z\tau_{xz} \\ n_x\tau_{yx} + n_y\tau_{yy} + n_z\tau_{yz} \\ n_x\tau_{zx} + n_y\tau_{zy} + n_z\tau_{zz} \\ n_x\Theta_x + n_y\Theta_y + n_z\Theta_z \end{bmatrix}, \\
\Theta_x = u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + k\frac{\partial T}{\partial x}, \\
\Theta_y = u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + k\frac{\partial T}{\partial y}, \\
\Theta_z = u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + k\frac{\partial T}{\partial z},
\end{gathered} \tag{2}$$

where ρ is the density, (u, v, w) are the local Cartesian velocity components, p is the static pressure, E is the total energy, H is the total enthalpy, (n_x, n_y, n_z) are the unit normal vectors of the cell surface, V is the velocity normal to the surface element dS, and Θ_i stands for the work of the viscous stresses and of the heat conduction, respectively.

The $K-\omega$ shear stress transport (SST) turbulence model [38] merges the Wilcox's $K-\omega$ model [39] with a high Reynolds number $K-\varepsilon$ model [40]. The $K-\omega$ SST two-equation turbulence model can be written in integral form as follows:

$$\frac{\partial}{\partial t}\int_{\Omega}\overrightarrow{W}_T d\Omega + \oint_{\partial\Omega}\left(\overrightarrow{F}_{c,T} - \overrightarrow{F}_{v,T}\right)dS = \int_{\Omega}\overrightarrow{Q}_T d\Omega, \quad (3)$$

where $\overrightarrow{W}_T$ is the vector of conservative variables, $\overrightarrow{F}_{c,T}$ and $\overrightarrow{F}_{v,T}$ represent the convective fluxes and viscous fluxes, respectively, and $\overrightarrow{Q}_T$ is the source term:

$$\begin{aligned}
\overrightarrow{W}_T &= \begin{bmatrix} \rho K \\ \rho\omega \end{bmatrix}, \\
\overrightarrow{F}_{c,T} &= \begin{bmatrix} \rho K V \\ \rho\omega V \end{bmatrix}, \\
\overrightarrow{F}_{v,T} &= \begin{bmatrix} n_x\tau_{xx}^K + n_y\tau_{yy}^K + n_z\tau_{zz}^K \\ n_x\tau_{xx}^\omega + n_y\tau_{yy}^\omega + n_z\tau_{zz}^\omega \end{bmatrix}, \\
\overrightarrow{Q}_T &= \begin{bmatrix} P_K - \beta^*\rho\omega K \\ P_\omega - \beta\rho\omega^2 + (1 - F_1)D_{K\omega} \end{bmatrix},
\end{aligned} \quad (4)$$

where K is the turbulent kinetic energy and ω is the specific dissipation rate. The components of the source term and the model constant are introduced in [37].

The spatial discretization of equation (1) on structured meshes is based on the cell-centered finite volume method. The upwind AUSM + UP scheme [41], which has a high resolution and computational efficiency for all speeds, is used to compute the convective fluxes. Second-order accuracy is achieved through the monotone upstream-centered schemes for conservation law (MUSCL) [42] with Van Albada et al. limiter function [43]. The viscous fluxes are solved by the central scheme, and turbulence is modeled with the $K - \omega$ SST two-equation model.

The solution of equation (1) employs a separate discretization in space and in time, so that space and time integration can be treated separately. For the control volume $\Omega_{I,J,K}$, equation (1) is written in the following form:

$$\Omega_{I,J,K}\frac{d\overrightarrow{W}_{I,J,K}}{dt} = \overrightarrow{R}_{I,J,K}. \quad (5)$$

The three-step Runge–Kutta method [44] with third-order accuracy has good data parallelism and lower storage overhead, which is used for temporal discretization of equation (5):

$$\begin{aligned}
\overrightarrow{W}_{I,J,K}^{(0)} &= \overrightarrow{W}_{I,J,K}^{(n)}, \\
\overrightarrow{W}_{I,J,K}^{(1)} &= \overrightarrow{W}_{I,J,K}^{(0)} + \frac{\Delta t_{I,J,K}}{\Omega_{I,J,K}}\overrightarrow{R}_{I,J,K}^{(0)}, \\
\overrightarrow{W}_{I,J,K}^{(2)} &= \frac{3}{4}\overrightarrow{W}_{I,J,K}^{(0)} + \frac{1}{4}\overrightarrow{W}_{I,J,K}^{(1)} + \frac{1}{4}\frac{\Delta t_{I,J,K}}{\Omega_{I,J,K}}\overrightarrow{R}_{I,J,K}^{(1)}, \\
\overrightarrow{W}_{I,J,K}^{(3)} &= \frac{1}{3}\overrightarrow{W}_{I,J,K}^{(0)} + \frac{2}{3}\overrightarrow{W}_{I,J,K}^{(1)} + \frac{2}{3}\frac{\Delta t_{I,J,K}}{\Omega_{I,J,K}}\overrightarrow{R}_{I,J,K}^{(1)}, \\
\overrightarrow{W}_{I,J,K}^{(n+1)} &= \overrightarrow{W}_{I,J,K}^{(3)},
\end{aligned} \quad (6)$$

where $\Delta t_{I,J,K}$ is the time step of control volume $\Omega_{I,J,K}$ and $\overrightarrow{R}_{I,J,K}$ stands for the residual.

4. Parallel Algorithm

4.1. Algorithm Description. The GPU implementation uses CUDA. CUDA is a general-purpose parallel computing platform and programming model for GPUs. CUDA provides a programming environment for high-level languages, such as C/C++, Fortran, and Python. For NVIDIA GPUs, CUDA has wider universality than other general programming models, such as OpenCL and OpenACC [9, 45–47]. In this study, we choose CUDA as the heterogeneous model to design GPU-accelerated parallel codes for CFD on GTX 1070 and Tesla V100 GPU. A complete GPU code consists of seven parts, namely, getting the device, allocating memory, data transfer from the host to the device, kernel execution, data transfer from the device to the host, free memory space, and resetting the device. In the CUDA programming framework, the execution of a GPU code can be divided into host codes and device codes, which are executed on the CPU and the GPU, respectively. The code on the device side calls the kernel functions to execute on the GPU. The kernel corresponds to a thread grid, which consists of several thread blocks. One thread block contains multiple threads, and a thread is the smallest execution unit of a kernel. Threads within a block can cooperate by sharing data through shared memory. Although the same instructions are executed on the threads, the processed data are different; this mode of execution is called the single instruction multiple thread (SIMT).

The GPU is rich in computing power but poor in memory capacity, whereas the CPU is the opposite. To run CFD applications on a GPU, we must fully utilize the CPU and the GPU. Thus, the GPU is responsible for executing kernel functions, and the CPU only manages the execution of the GPU and communication. In a GPU parallel computing program, a thread is the smallest unit for kernel execution. Threads are executed concurrently in the streaming multiprocessor (SM) as a warp, and a warp contains 32 threads. Therefore, the optimal number of threads in each thread block is an integer multiple of 32. For current devices, the maximum number of threads on a thread block is 1024. The greater the number of threads on each thread block, the smaller the number of thread blocks an SM can call. This condition will diminish the advantage of the GPU in utilizing multithreading to hide the delays in memory acquisition and instruction execution due to the issue of thread synchronization. Too few threads in the thread block will result in idle threads and thereby insufficient device utilization. In this study, a thread block size of 256 is used. The number of thread blocks is determined by the scale of workload to ensure that each thread is loaded with the computation of a grid cell.

MPI and OpenMP are two application programming interfaces that are widely used for running parallel codes on a multi-GPU platform. OpenMP can be utilized within a node for fine-grained parallelization using shared memory, and MPI works on shared and distributed memories and is widely used for massive parallel computations. In general, MPI exhibits performance loss compared with OpenMP but is relatively easy to perform on different types of hardware

and has good scalability [48, 49]. In this work, MPI-based communication for shared or distributed memory is hybridized with CUDA to implement large-scale computation on multi-GPU HPC clusters. The parallel algorithm for CFD on multi-GPU clusters based on MPI and CUDA is shown in Algorithm 1. It is noted that “Block_size” and “Thread_size” stand for the number of thread blocks and the thread block size, respectively. At the beginning of the calculation, the *MPI_Init* function is called to enter the MPI environment. The *Device_Query* function is called to run the GPU. Then, data are transferred from the CPU to the GPU using the *cudaMemcpyHostToDevice* function, and a series of kernel functions, including the processing of boundary, the computing of local time step, the calculation of gradient of primitive variables, the calculation of fluxes, and the update of primitive variables, are executed on the GPU. Among these kernel functions, the calculation of gradients and fluxes consumes the largest amount of time. For multi-GPU parallel computing, data exchange is required between GPUs, and the *Primitive_Variables_Exchange* and *Grad_Primitive_Variables_Exchange* functions are used to exchange the primitive variables and their gradients, respectively. After the kernel iteration ends, the *cudaMemcpyDeviceToHost* function is called to transfer the data from the GPU to the CPU for postprocessing. Finally, the *MPI_Finalize* function is called to exit the MPI environment.

The data transfer is essential in multi-GPU parallel computing. In this paper, the exchange of data is done by the CPUs controlling the GPUs. The data transfer process between GPUs is shown in Figure 1. First, we call the *cudaMemcpyDeviceToHost* function to transfer data from the device to the relevant host through the PCI-e bus. Then, the data are transferred between CPUs with the MPI. Finally, we call the *cudaMemcpyHostToDevice* function to transfer the data from the host to the target device through the PCI-e bus.

4.2. Algorithm Optimizations. The performance of the GPU parallel program can be optimized using two main methods: parallel execution optimization for the highest degree of parallelism utilization and memory access optimization for maximum memory throughput.

4.2.1. Parallel Execution Optimization. CUDA programs have two execution modes: synchronous mode and asynchronous mode. Synchronous mode means that control does not return to the host until the current kernel function is executed. Asynchronous mode means that control returns to the host immediately once the kernel function is started. Therefore, the host can start new kernel functions and perform data exchange simultaneously. Streams are sequential sequences of commands that can be managed by the CUDA program to control device-level parallelism. Commands in one stream are executed in order, but streams from different commands can be executed in parallel. Thus, the concurrent execution of multiple kernel functions, which is called asynchronous concurrent execution, can be implemented via streams. For the massive parallel computing of CFD on a GPU, a series of kernel functions needs to be executed. The concurrent execution of these kernel functions can be implemented via streams. CUDA creates and destroys streams through the *cudaStreamCreate* and *cudaStreamDestroy* functions, and synchronization between threads can be achieved through the *cudaDeviceSynchronize* function. The implementation of the asynchronous concurrent execution algorithm of CFD on the GPU is shown in Algorithm 2. The initialization of the gradients and residuals, the boundary condition processing, and the time step calculation are concurrently executed by creating a stream. In addition, the calculation of the inviscid fluxes can also be concurrently executed with the calculation of the gradients of primitive variables.

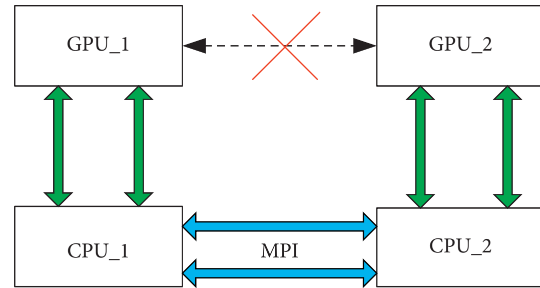

Figure 1: The data transfer process between GPUs.

4.2.2. Memory Access Optimization. The data transfer between the host and the device is realized via the PCI-e bus. The transmission speed is considerably lower than the GPU bandwidth. Therefore, the application should reduce the data exchange between the host and the device as much as possible. In this work, the kernel iteration is completely performed on the GPU, and data transmission only occurs at the beginning and end of the kernel iteration. Intermediate variables can be created in the data memory and released after the calculation is completed. However, for multi-GPU parallel computing, data transfer is inevitable between the host and the device due to the features of communication between GPUs.

The GPU provides different levels of memory structure: global memory, texture memory, shared memory, and the registers. This storage mode ensures that the GPU can reduce the data transfer between the global memory and the device. The global memory locates in the video memory with a large access delay. Therefore, the maximum bandwidth can only be obtained by employing the coalesced memory access. Texture memory is also part of the video memory. Compared with the global memory, the texture memory can use the cache to improve the data access speed and obtain a high bandwidth without strictly observing the conditions of coalesced memory access. The shared memory has a bandwidth much higher than that of the global and texture memories. Data sharing among threads in the SM can be realized by storing the frequently used data in the shared memory. The register is the exclusive storage type of the thread, which stores the variables declared in the kernel to accelerate data processing. The GPU computing efficiency

```
(1)  MPI_Init (&argc, &argv);
(2)  Device_Query ( );
(3)  cudaMemcpy (d_a, h_a, sizeof(float)*n, cudaMemcpyHostToDevice);
(4)  //Kernel execution start
(5)  for i = 0; i < max_step; i++
(6)     Boundary_Processing_GPU<<<Block_size, Thread_size>>> ( );
(7)     Time_Step_GPU<<<Block_size, Thread_size>>> ();
(8)     Primitive_Variables_Exchange ();
(9)     Grad_Primitive_Variables_GPU<<<Block_size, Thread_size>>> ( );
(10)    Grad_primitive_Variables_Exchange ( );
(11)    Flux_GPU<<<Block_size, Thread_size>>> ( );
(12)    Primitive_Variables_Update_GPU<<<Block_size, Thread_size>>> ( );
(13) end for
(14) //kernel execution end
(15) cudaMemcpy (h_a, d_a, sizeof(float)*n. cudaMemcpyDeviceToHost);
(16) Flow_post-processing ( );
(17) MPI_finalize ( );
```

Algorithm 1: Parallel algorithm for CFD on multi-GPU HPC clusters.

```
(1)  cudaStreamCreate (&stream[j]);
(2)     Boundary_Processing_GPU<<<Block_size, Thread_size, stream[0]>>>( );
(3)     Time_Step_GPU<<<Block_size, Thread_size, stream[1]>>>( );
(4)     Grad_Initial<<<Block_Size, Thread_Size, stream[2]>>>( );
(5)     RHS_Initial<<<Block_Size, Thread_Size, stream[3]>>>( );
(6)     cudaDeviceSynchronize ( );
(7)     Grad_Primitive_Variables_GPU<<<Block_size, Thread_size, stream[0]>>>( );
(8)     Convective_Flux_GPU<<<Block_size, Thread_size, stream[1]>>>( );
(9)     cudaDeviceSynchronize ( );
(10)    Viscous_Flux_GPU<<<Block_size, Thread_size, stream[0]>>>( );
(11)    RHS_GPU<<<Block_size, Thread_size, stream[0]>>>( );
(12)    Primitive_Variables_Update_GPU<<<Block_size, Thread_size, stream[0]>>>( );
(13)    cudaDeviceSynchronize ();
(14) cudaStreamDestroy (stream[j]);
```

Algorithm 2: Asynchronous concurrent execution algorithm of CFD on the GPU.

can be improved by properly using the texture memory, the shared memory, and the registers and reducing the number of accesses to the global memory.

4.3. Nonblocking Communication Mode. When performing the parallel computing of CFD on multi-GPU HPC clusters, the kernel iteration process needs to exchange data on the boundary, including primitive variables and their gradients. The primitive variables $\overrightarrow{U} = [\rho \ u \ v \ w \ p \ K \ \omega]^T$ are chosen to ensure that as small data as possible are exchanged. In this process, CPU_CPU communication and CPU_GPU data transfer exist. Optimizing the communication between GPUs significantly affects the performance of multi-GPU parallel systems.

The traditional method is the blocking communication mode, as shown in Figure 2. Algorithm 3 shows the algorithm of the blocking communication mode. The blocking communication mode calls the *MPI_Bsend* and *MPI_Recv* functions for data transmission and reception, respectively. In the blocking communication mode, GPU computing, CPU_CPU communication, and CPU_GPU data transfer are completely separated. The communication time in this communication mode is a pure overhead, which seriously reduces the efficiency of the parallel system.

The nonblocking communication mode shields the communication time with the computing time by overlapping the computation and the communication, as shown in Figure 3. Algorithm 4 shows the algorithm of the nonblocking communication mode. The overlaps among GPU computing, CPU_CPU communication, and CPU_GPU data transfer are achieved by creating two CUDA streams. When exchanging primitive variables, stream 0 is used for CPU_GPU data transfer and stream 1 is used for boundary condition processing and time step calculation. When the gradients of the primitive variables are exchanged, stream 1 is used to calculate the inviscid fluxes. This can be done

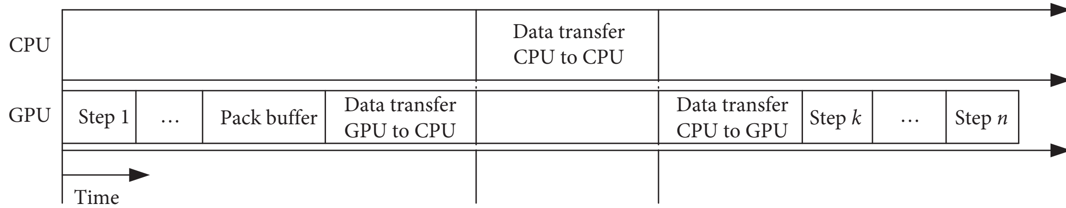

Figure 2: Blocking communication mode.

```
(1) if device_count>1 then
(2)    cudaMemory (h_a, d_a, sizeof(float)*n, cudaMemcpyDeviceToHost);
(3)    MPI_Bsend (*buf, int count, MPI_Datatype, int dest, int tag,MPI_COMM_WORLD);
(4)    MPI_Recv (*buf, int count, MPI_Datatype, int source, int tag,MPI_COMM_WORLD, MPI_Status *n,
       cudaMemcpyHostToDevice);
(5)    cudaMemcpy (d_a, h_a, sizeof(float)*n, cudaMemcpyHostToDevice);
(6) end if
```

Algorithm 3: Blocking communication mode algorithm.

because the computations of the inviscid fluxes do not depend on the values being transferred among GPUs. The exchange of the gradients to the host can start as soon as the data are packaged by using stream 0. Then, the data transmission between CPUs is implemented with MPI. At the same time, the *Convective_Flux_GPU* function is executed by using stream 1. Finally, the target device can receive the data from the host with stream 0. Therefore, the overlaps among GPU computing, CPU_CPU communication, and CPU_GPU data transfer can be realized. In this work, the nonblocking communication mode based on multistream computing is used to optimize the communication of GPU parallel programs. The nonblocking communication mode calls the *MPI_Isend* and *MPI_Irecv* functions for data transmission and reception, respectively, and the *MPI_Waitall* function to await the communication completion and query the completion status. The *cudaMemcpyAsync* function is used for asynchronous data transmission.

4.4. Domain Decomposition and Load Balancing. In the multi-GPU parallel computing, the computational grid needs to be partitioned. Considering the load balancing problem, this work uses the 1D domain decomposition method to load each GPU with approximately the same number of computational grid. The 1D domain decomposition is shown in Figure 4. This method adopts the concept of the bisection method, which is simple to implement and facilitates load balancing. The dichotomy algorithm is shown in Algorithm 5.

Meanwhile, the coalesced memory access is easy to implement due to its boundary data alignment for effectively improving the efficiency of boundary data communication. When the nonblocking communication mode is used, the data in the GPU are divided into three parts (top, middle, and bottom). The top and bottom parts of the data need to be exchanged with other devices. The data transfer of the top and bottom parts can occur simultaneously with the computation of the middle portion.

5. Results and Discussion

5.1. Test Case: Flat Plate Flow. The supersonic flow over a flat plate is a well-known benchmark problem for compressible turbulent flow of CFD applications [50, 51]. This test case has been studied by many researchers and is widely used to verify and validate CFD codes.

The free stream Ma number is 4.5, the Reynolds number based on the unit length is 6.43×10^6, the static temperature is 61.1 K, and the angle of attack is 0°. No-slip boundary condition is applied at the stationary flat plate surface, which is also assumed to be adiabatic.

The supersonic flat plate boundary layer problem is solved on various meshes, namely, mesh 1 (0.72 million), mesh 2 (1.44 million), mesh 3 (2.88 million), mesh 4 (5.76 million), mesh 5 (11.52 million), mesh 6 (23.04 million), mesh 7 (46.08 million), and mesh 8 (92.16 million).

5.2. Hardware Environment and Benchmarking. In this work, two types of devices, namely, GTX 1070 GPU and Tesla V100 GPU, are introduced. These two devices were introduced by NVIDIA Corporation. Table 1 shows the main performance parameters of GPUs. For these types of devices, the single-precision floating-point operations far exceed double-precision ones. Therefore, the single-precision data for GPU parallelization are used. The performance of the latest Turing architecture Tesla V100 GPU is greatly improved compared with the previous architecture GPU.

In this study, we use CUDA version 10.1, Visual Studio 2017 for C code and MPICH2 1.4.1 for MPI communication.

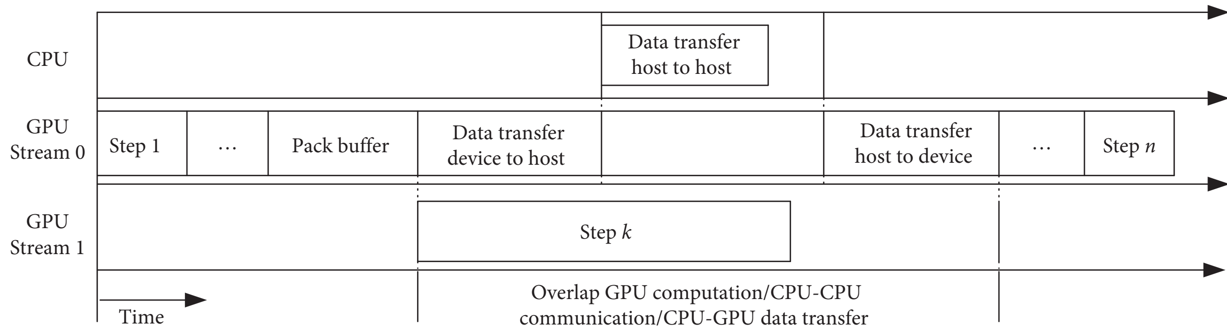

Figure 3: Nonblocking communication mode.

```
(1)  if device_count>1 then
(2)     cudaMemcpyAsync (h_a, d_a, sizeof(float)*n, cudaMemcpyDeviceToHost, stream[0]);
(3)     MPI_Isend (*buf, int count, MPI_Datatype, int dest, int tag, MPI_COMM_WORLD, MPI_Request *request);
(4)     //Primitive_Variables_Exchange;
(5)     Boundary_Processing_GPU<<<Block_size, Thread_size, stream[1]>>> ( );
(6)     Time_Step_GPU<<<Block_size, Thread_size, stream[1]>>> ( );
(7)     //Grad_Primitive_Variables_Exchange;
(8)     Convective_Flux_GPU<<<Block_size, Thread_size, stream[1]>>> ( );
(9)     MPI_Irecv (*buf, int count, MPI_Datatype, int source, int tag, MPI_COMM_WORLD, MPI_Status *status, MPI_Request
      *request);
(10)    MPI_Waitall ( );
(11)    cudaMemcpyAsync (d_a, h_a, sizeof(float)*n, cudaMemcpyHostToDevice, stream[0]);
(12) end if
```

Algorithm 4: Nonblocking communication mode algorithm.

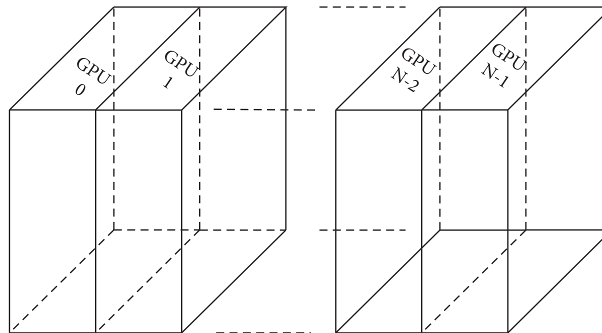

Figure 4: 1D domain decomposition method.

```
(1)  for i = 1; i < partition_number; i++
(2)     a = min_XYZ; b = max_XYZ; xyz_current = 0.5*(min_XYZ + max_XYZ);
(3)     while abs (dis) > 1.0exp-10 do
(4)        dis=(current_count-average_count)/average_count;
(5)        if dis>0.0 then
(6)           b = xyz_current;
(7)        else
(8)           a = xyz_current;
(9)        end if
(10)       xyz_current = 0.5*(a + b);
(11)    end while
(12) end for
```

Algorithm 5: The dichotomy algorithm.

Table 1: Main performance parameters of GPUs.

	GTX 1070	Tesla V100
Date introduced	June 2016	January 2018
Architecture	Pascal	Volta
Computation capability	6.1	7.0
Device memory (GB)	8	32
Streaming multiprocessors	15	80
Stream processors	1,920	5,120
Single precision (TFLOP/S)	6	14
Double precision (TFLOP/S)	0.2	7
Memory bandwidth (GB/s)	256	900

In addition, a node contains one Intel Xeon E5-2670 CPU at 2.6 GHz with eight cores and four GPUs.

5.3. Performance Analysis. Speedup (SP) and parallel efficiency (PE) are important parameters for measuring the performance of a hybrid parallel algorithm. Speedup and parallel efficiency are defined as follows [9, 52]:

$$\begin{aligned} \mathrm{SP} &= \frac{t_{\mathrm{CPU}}}{t_{\mathrm{GPU}}}, \\ \mathrm{PE} &= \frac{(W_2/N_2)}{(W_1/N_1)} \times \frac{T_1}{T_2}, \end{aligned} \tag{7}$$

where t_{CPU} is the runtime of one iteration step for one CPU with eight cores, t_{GPU} is the runtime of one iteration step for GPUs, W_1 and W_2 are the problem sizes, N_1 and N_2 are the number of GPUs, and T_1 and T_2 are the computation times. If $W_2 = W_1$, then the problem size remains constant when the number of GPUs is increased. The parallel efficiency of strong scalability is (N_1T_1/N_2T_2). If $(W_2/N_2) = (W_1/N_1)$, then the problem size grows proportionally with the number of GPUs; thus, the problem size remains constant in each GPU. The parallel efficiency of weak scalability is (T_1/T_2). Here, subscript 1 denotes a single GPU and subscript 2 stands for multi-GPU HPC clusters. The runtime of one iteration step is achieved by averaging the execution time of ten thousand time steps.

5.3.1. Single GPU Implementation. For a single GPU implementation, the time required for the calculation of one iteration step is provided in Table 2 (time is given in milliseconds). Figure 5 shows that the speedup of a single GPU increases with the increase in grid size. For the Tesla V100 GPU, the speedup reaches 135.39 for mesh 1 and 182.11 for mesh 8. Thus, the Tesla V100 GPU has a considerably greater speedup than the GTX 1070 GPU. GPU parallelization can greatly improve the computational efficiency compared with CPU-based parallel computing. For meshes 7 and 8, the GTX 1070 GPU cannot afford such a large amount of calculations because of the limitation of device memory. For our GPU codes, 1 GB of device memory can load approximately 3 million grid cells. As the grid size is increased, the speedup of the GPU code increases gradually. The reason is that, as the grid size increases, the proportion of kernel execution increases with those of data arrangement and communication. Meanwhile, the growth rate of speedup is gradually decreasing because of the limitation of the number of SMs and CUDA cores.

5.3.2. Scalability. In this section, the blocking communication mode is used to study the scalability of GPU codes. Here, the performance of GTX 1070 and Tesla V100 multi-GPU HPC clusters is discussed.

Strong scaling tests are performed for meshes 1 to 8 on multi-GPU HPC clusters. Tables 3 and 4 provide the time required for the calculation of one iteration step for multi-GPU implementation (time is given in milliseconds). Figures 6 and 7 show the strong speedup and parallel efficiency, respectively. In Figure 6, the strong speedup is shown for different grid sizes. Evidently, a large grid size can reach a high speedup. For GTX 1070 and Tesla V100 multi-GPU clusters, the speedups of four GPUs reach 172.59 and 576.49, respectively. These values are far greater than the speedups achieved by a single GPU. Thus, a high degree of strong scaling performance is maintained. Multi-GPU parallelization can considerably improve the computational efficiency with the increase in grid size. However, multi-GPU parallel computing does not show obvious advantages when the grid size is small. This can be explained by the fact that the relative weight of data exchange is inversely proportional to the grid size. GPU is specialized for compute-intensive, highly parallel computation. In Figure 7, the strong parallel efficiency is shown for different grid sizes. This result is consistent with the change law of speedup; that is, the parallel efficiency increases with the increase in grid size, and the strong parallel efficiency performance of GTX 1070 multi-clusters is slightly better than that of Tesla V100 GPUs. For mesh 8, the strong parallel efficiency of four Tesla V100 GPUs is close to 80%. In addition, a larger number of GPUs indicate a lower parallel efficiency because of the increase in the amount of data transfer. Figure 8 shows the amount of memory communications for parallel computing with four GPUs. As the grid size increases, the amount of memory communication increases proportionally.

The weak scaling tests for parallel efficiency are shown in Figure 9, and the grid size loaded on each block remains constant. As expected, the parallel efficiency decreases as the number of GPUs increases because the amount of data exchange increases with the increase in the number of GPUs. For the grid size of mesh 6 on each GTX 1070 GPU and Tesla V100 GPU, the weak parallel efficiency of four GPUs can

Table 2: The runtime for one CPU and a single GPU.

No.	CPU (ms)	GTX 1070 (ms)	Tesla V100 (ms)
Mesh 1	567.29	15.69	4.19
Mesh 2	1,170.6	30.61	8.27
Mesh 3	2,619.41	62.75	16.9
Mesh 4	5,605.29	120.25	33.18
Mesh 5	11,258.58	236.29	64.88
Mesh 6	22,850.69	476.12	128.98
Mesh 7	46,211.38	—	256.72
Mesh 8	93,322.76	—	512.44

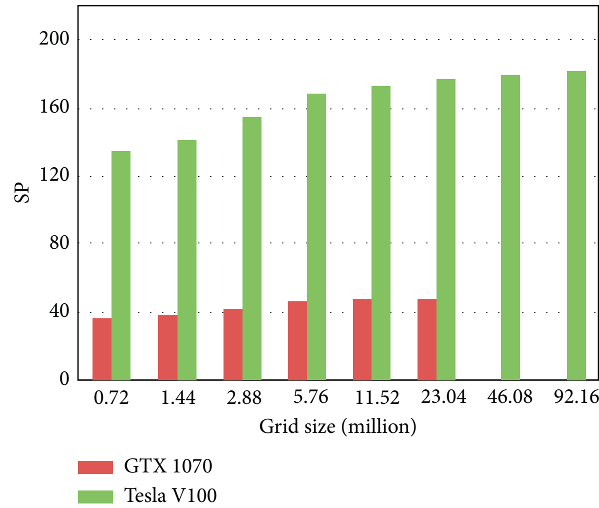

Figure 5: The speedup of a single GPU for different number of grid cells.

Table 3: The runtime for GTX 1070 multi-GPU clusters.

No.	Two GPUs (ms)	Three GPUs (ms)	Four GPUs (ms)
Mesh 1	12.62	11.27	10.38
Mesh 2	20.86	18.19	16.01
Mesh 3	35.56	28.19	23.38
Mesh 4	67.55	52.43	42.33
Mesh 5	126.93	96.24	77.15
Mesh 6	252.01	187.84	144.43
Mesh 7	499.02	369.68	276.86
Mesh 8	—	—	540.73

Table 4: The runtime for Tesla V100 multi-GPU clusters.

No.	Two GPUs (ms)	Three GPUs (ms)	Four GPUs (ms)
Mesh 1	3.85	3.41	3.01
Mesh 2	7.16	6.58	5.86
Mesh 3	13.78	11.82	9.31
Mesh 4	23.99	20.43	15.8
Mesh 5	41.59	32.22	25.71
Mesh 6	78.69	58.81	45.16
Mesh 7	151.78	108.85	83.92
Mesh 8	295.13	208.72	161.88

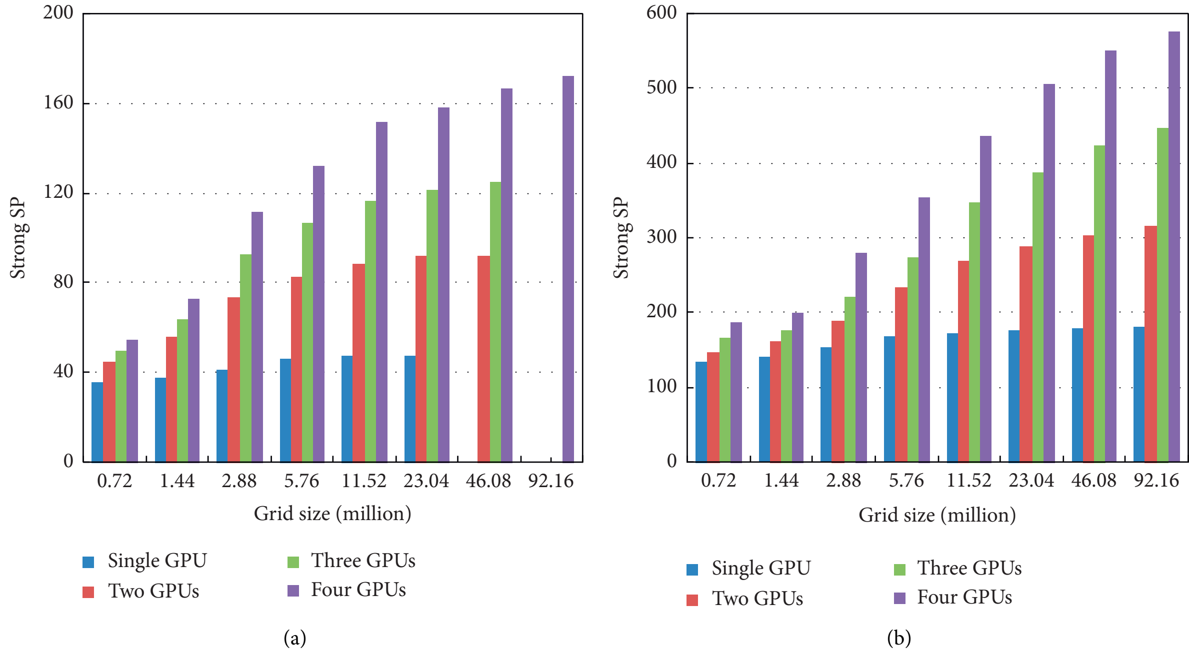

Figure 6: The strong speedup of multi-GPU clusters. (a) GTX 1070. (b) Tesla V100.

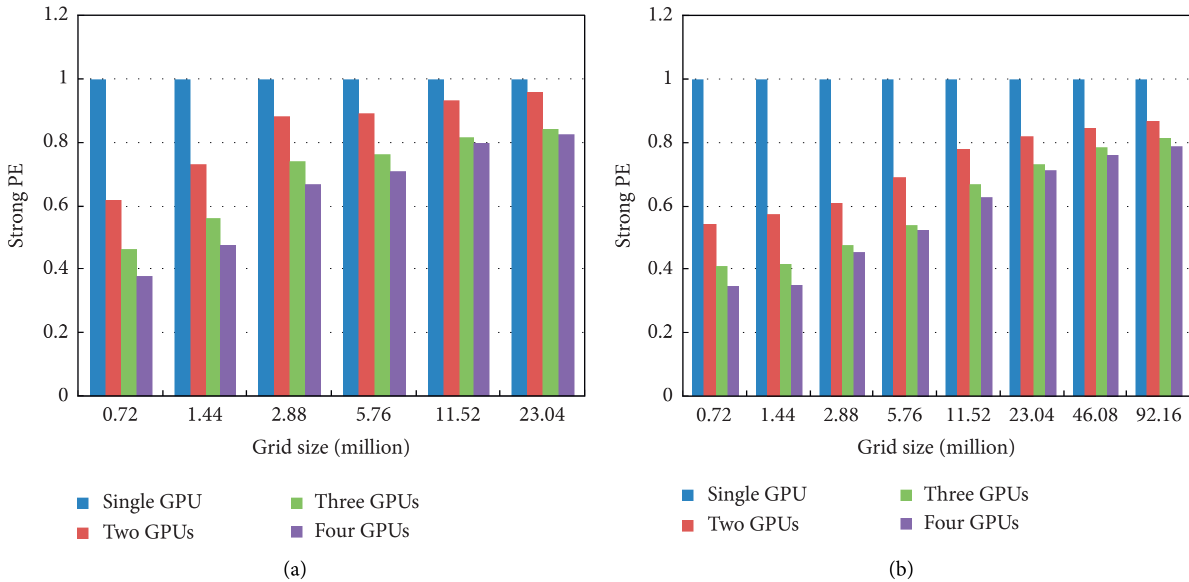

Figure 7: The strong parallel efficiency of multi-GPU clusters. (a) GTX 1070. (b) Tesla V100.

reach 88.05% and 79.68%, respectively. Thus, a high degree of weak scaling performance is maintained.

5.3.3. Performance of Nonblocking Mode between GPUs. In this section, the nonblocking communication mode is used to study the performance of GTX 1070 GPUs and Tesla V100 multi-GPU HPC clusters. As an example, mesh 6 is investigated on one node with four GPUs to compare the performance of the two communication modes.

Figures 10 and 11 show the strong scalability performance of the nonblocking communication mode compared with the blocking communication mode. Figure 12 shows the weak parallel efficiency of the two methods. The results show that the nonblocking communication mode can shield the communication time with the computing time by overlapping the communication and the computation. For four GPUs, the strong speedups of GTX 1070 and Tesla V100 multi-GPUs increase by 15.15% and 19.63%, respectively. In addition, the weak parallel efficiency remains above 87%

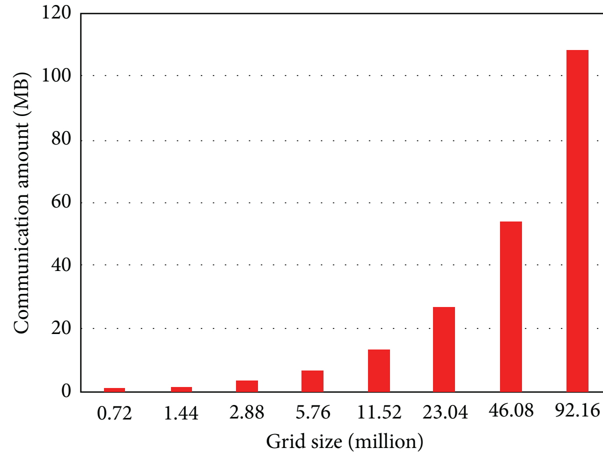

Figure 8: The amount of memory communications.

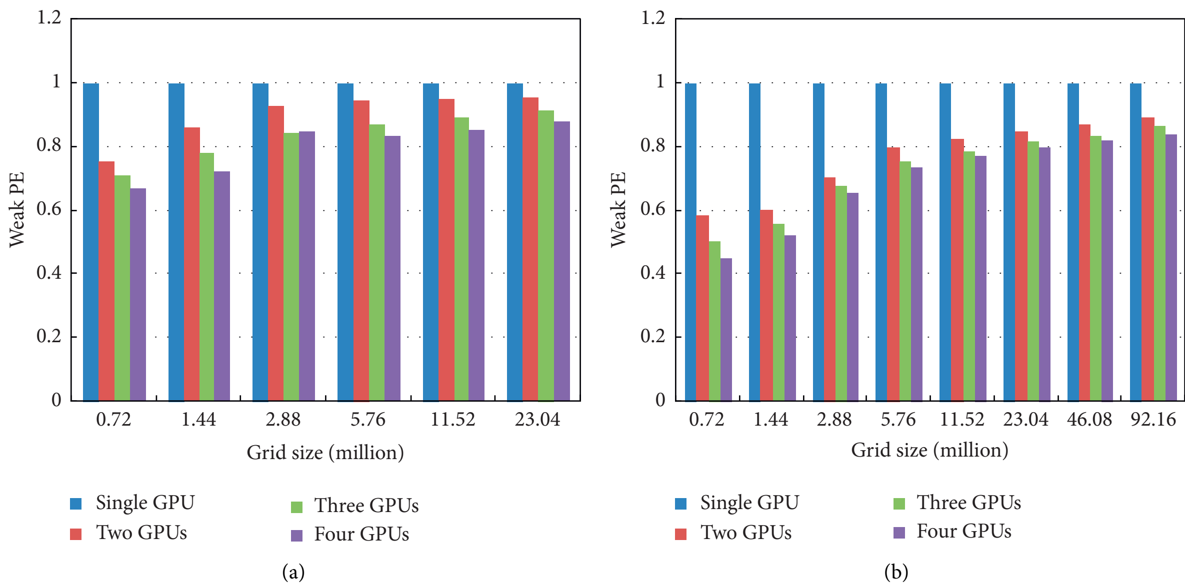

Figure 9: The weak parallel efficiency of multi-GPU clusters. (a) GTX 1070. (b) Tesla V100.

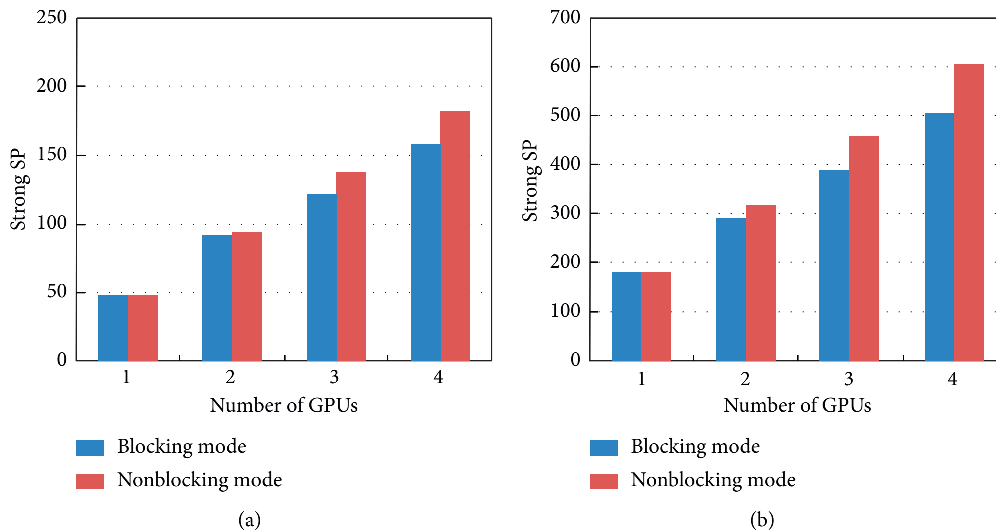

Figure 10: The strong speedup of multi-GPU clusters with different communication modes. (a) GTX 1070. (b) Tesla V100.

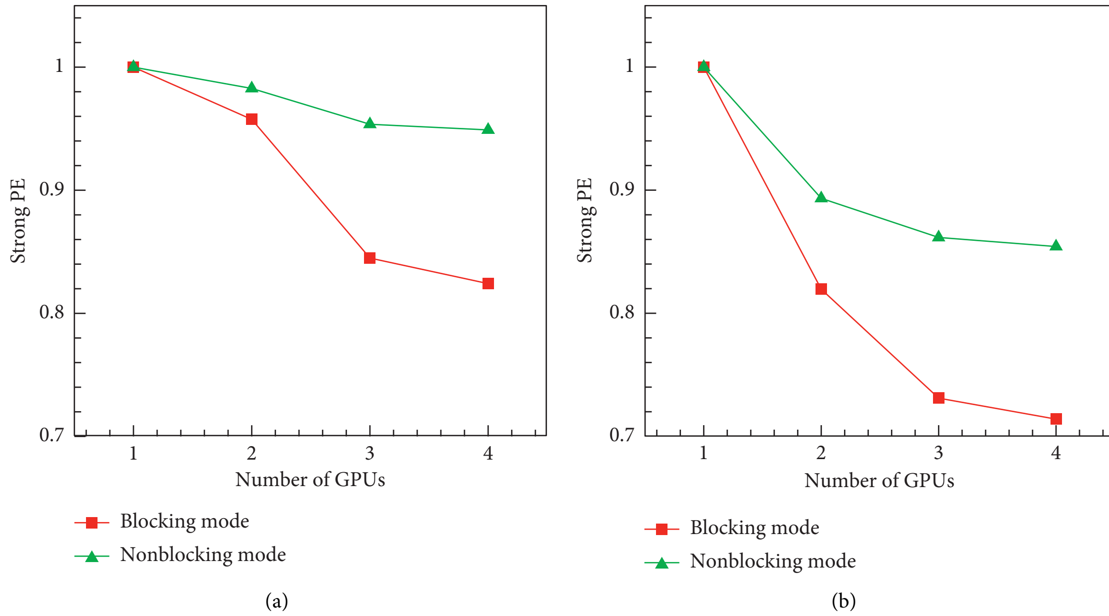

FIGURE 11: The strong parallel efficiency of multi-GPU clusters with different communication modes. (a) GTX 1070. (b) Tesla V100.

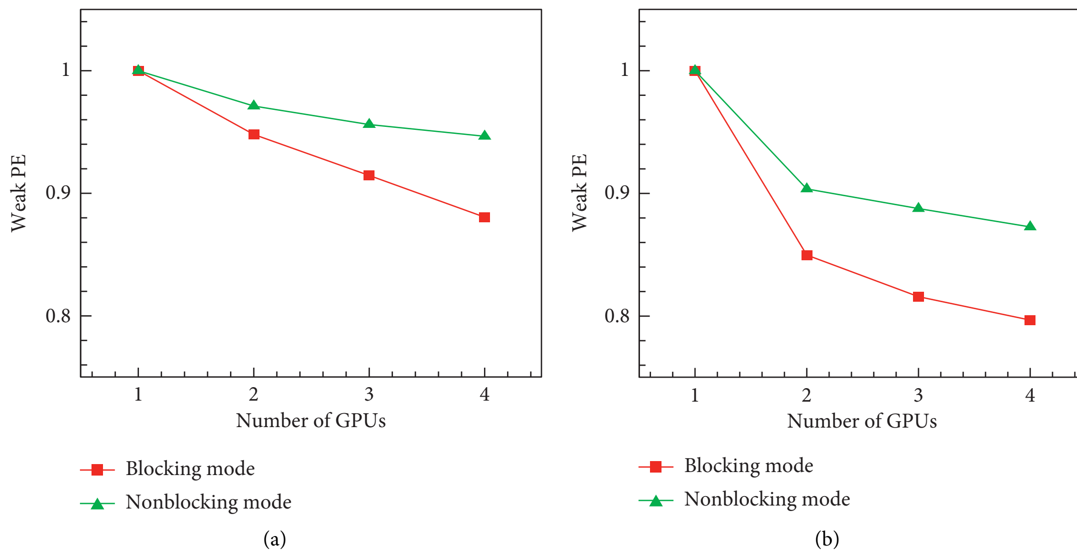

FIGURE 12: The weak parallel efficiency of multi-GPU clusters with different communication modes. (a) GTX 1070. (b) Tesla V100.

with the nonblocking communication mode. Moreover, the GPU-based parallelization with Tesla V100 GPUs can exhibit a better performance improvement than the GTX 1070 multi-GPU HPC clusters. This result is because the relative weight of the communication time of Tesla V100 GPUs is higher.

6. Conclusions

In this work, a hybrid parallel algorithm of MPI and CUDA for CFD applications on multi-GPU HPC clusters has been proposed. Optimization methods have been adopted to achieve the highest degree of parallelism utilization and maximum memory throughput. In this study, a thread block size of 256 is used. The number of thread blocks is determined by the scale of workload to ensure that each thread is loaded with the computation of a grid cell. For a single GPU implementation, two types of devices have been discussed. We obtain an acceleration ratio of more than 36 times, which indicates that GPU parallelization can greatly improve the computational efficiency compared with CPU-based parallel computing. Meanwhile, a large grid size can reach a high speedup due to the increase in the proportion of kernel executions compared with those of data arrangement and communication. The speedups of four GPUs reach 172.59 and 576.49 for GTX 1070 GPUs and Tesla V100 multi-GPU

HPC clusters, respectively. The strong and weak parallel efficiency are maintained at a high level when the grid size is at a large value. Thus, the parallel algorithm has good strong and weak scalability. Nonblocking communication mode has been proposed to fully overlap GPU computing, CPU_CPU communication, and CPU_GPU data transfer. For four GPUs, the strong speedups of GTX 1070 and Tesla V100 multi-GPUs increase by 15.15% and 19.63%, respectively. In addition, the weak parallel efficiency remains above 87% with the nonblocking communication mode.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Foundation (NNSF of China) project (no. 11472004). Also, the first author would like to thank Dr. Feng Liu.

References

[1] Z. J. Wang, K. Fidkowski, R. Abgrall et al., "High-order CFD methods: current status and perspective," *International Journal for Numerical Methods in Fluids*, vol. 72, no. 8, pp. 811–845, 2013.

[2] D. Zhang, S. Tang, and J. Che, "Concurrent subspace design optimization and analysis of hypersonic vehicles based on response surface models," *Aerospace Science and Technology*, vol. 42, pp. 39–49, 2015.

[3] A. Afzal, Z. Ansari, A. R. Faizabadi, and M. K. Ramis, "Parallelization strategies for computational fluid dynamics software: state of the art review," *Archives of Computational Methods in Engineering*, vol. 24, no. 2, pp. 337–363, 2017.

[4] K. E. Niemeyer and C.-J. Sung, "Recent progress and challenges in exploiting graphics processors in computational fluid dynamics," *The Journal of Supercomputing*, vol. 67, no. 2, pp. 528–564, 2014.

[5] A. Moreno, J. J. Rodríguez, D. Beltrán, A. Sikora, J. Jorba, and E. César, "Designing a benchmark for the performance evaluation of agent-based simulation applications on HPC," *The Journal of Supercomputing*, vol. 75, no. 3, pp. 1524–1550, 2019.

[6] M. Rodriguez and L. Brualla, "Many-integrated core (MIC) technology for accelerating Monte Carlo simulation of radiation transport: a study based on the code DPM," *Computer Physics Communications*, vol. 225, pp. 28–35, 2018.

[7] N. Cadenelli, Z. Jakšić, J. Polo, and D. Carrera, "Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads," *Future Generation Computer Systems*, vol. 94, pp. 148–159, 2019.

[8] J. Satheesh Kumar, G. Saravana Kumar, and A. Ahilan, "High performance decoding aware FPGA bit-stream compression using RG codes," *Cluster Computing*, vol. 22, no. S6, pp. 15007–15013, 2019.

[9] NVIDIA Corporation, *NVIDIA: CUDA C Programming Guide 10.2*, NVIDIA Corporation, Santa Clara, CA, USA, 2019, https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[10] M. S. Friedrichs, P. Eastman, V. Vaidyanathan et al., "Accelerating molecular dynamic simulation on graphics processing units," *Journal of Computational Chemistry*, vol. 30, no. 6, pp. 864–872, 2009.

[11] S. S. Sawant, O. Tumuklu, R. Jambunathan, and D. A. Levin, "Application of adaptively refined unstructured grids in DSMC to shock wave simulations," *Computers & Fluids*, vol. 170, pp. 197–212, 2018.

[12] G. Nguyen, S. Dlugolinsky, M. Bobák et al., "Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey," *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.

[13] A. A. Awan, K. V. Manian, C. H. Chu, H. Subramoni, and D. K. Panda, "Optimized large-message broadcast for deep learning workloads: MPI, MPI+NCCL, or NCCL2?" *Parallel Computing*, vol. 85, pp. 141–152, 2009.

[14] C. Xu, X. Deng, L. Zhang et al., "Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer," *Journal of Computational Physics*, vol. 278, pp. 275–297, 2014.

[15] S. Iturriaga, S. Nesmachnow, F. Luna, and E. Alba, "A parallel local search in CPU/GPU for scheduling independent tasks on large heterogeneous computing systems," *The Journal of Supercomputing*, vol. 71, no. 2, pp. 648–672, 2015.

[16] E. Calore, A. Gabbana, J. Kraus, E. Pellegrini, S. F. Schifano, and R. Tripiccione, "Massively parallel lattice-Boltzmann codes on large GPU clusters," *Parallel Computing*, vol. 58, pp. 1–24, 2016.

[17] J. Castagna, X. Guo, M. Seaton, and A. O'Cais, "Towards extreme scale dissipative particle dynamics simulations using multiple gpgpus," *Computer Physics Communications*, vol. 251, p. 107159, 2020.

[18] J. Kraus, "An introduction to CUDA-Aware MPI," 2013, https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.71.

[19] D. Li, C. Xu, B. Cheng, M. Xiong, X. Gao, and X. Deng, "Performance modeling and optimization of parallel LU-SGS on many-core processors for 3D high-order CFD simulations," *The Journal of Supercomputing*, vol. 73, no. 6, pp. 2506–2524, 2017.

[20] T. Brandvik and G. Pullan, "Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 221, no. 12, pp. 1745–1748, 2007.

[21] T. Brandvik and G. Pullan, "Acceleration of a 3D Euler solver using commodity graphics hardware," in *Proceedings of the 46th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NEV, USA, January 2008.

[22] F. Ren, B. Song, Y. Zhang, and H. Hu, "A GPU-accelerated solver for turbulent flow and scalar transport based on the Lattice Boltzmann method," *Computers & Fluids*, vol. 173, pp. 29–36, 2018.

[23] N. Tran, M. Lee, and S. Hong, "Performance optimization of 3D lattice Boltzmann flow solver on a GPU," *Scientific Programming*, vol. 2017, Article ID 1205892, 16 pages, 2017.

[24] A. Khajeh-Saeed and J. Blair Perot, "Direct numerical simulation of turbulence using GPU accelerated supercomputers," *Journal of Computational Physics*, vol. 235, pp. 241–257, 2013.

[25] F. Salvadore, M. Bernardini, and M. Botti, "GPU accelerated flow solver for direct numerical simulation of turbulent

flows," *Journal of Computational Physics*, vol. 235, pp. 129–142, 2013.

[26] Z. H. Ma, H. Wang, and S. H. Pu, "GPU computing of compressible flow problems by a meshless method with space-filling curves," *Journal of Computational Physics*, vol. 263, pp. 113–135, 2014.

[27] J.-L. Zhang, Z.-H. Ma, H.-Q. Chen, and C. Cao, "A GPU-accelerated implicit meshless method for compressible flows," *Journal of Computational Physics*, vol. 360, pp. 39–56, 2018.

[28] W. Cao, C.-f. Xu, Z.-h. Wang, L. Yao, and H.-y. Liu, "CPU/GPU computing for a multi-block structured grid based high-order flow solver on a large heterogeneous system," *Cluster Computing*, vol. 17, no. 2, pp. 255–270, 2014.

[29] X. Liu, Z. Zhong, and K. Xu, "A hybrid solution method for CFD applications on GPU-accelerated hybrid HPC platforms," *Future Generation Computer Systems*, vol. 56, pp. 759–765, 2016.

[30] P. D. Mininni, D. Rosenberg, R. Reddy, and A. Pouquet, "A hybrid MPI-OpenMP scheme for scalable parallel pseudo-spectral computations for fluid turbulence," *Parallel Computing*, vol. 37, no. 6-7, pp. 316–326, 2011.

[31] J. C. Thibault and I. Senocak, "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows," in *Proceedings of the 47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Orlando, FL, USA, January 2009.

[32] D. A. Jacobsen, J. C. Thibault, and I. Senocak, "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters," in *Proceedings of the 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Orlando, FL, USA, January 2010.

[33] P. Castonguay, D. M. Williams, P. E. Vincent, M. Lopez, and A. Jameson, "On the development of a high-order, multi-GPU enabled, compressible viscous flow solver for mixed unstructured grids," in *Proceedings of the 20th AIAA Computational Fluid Dynamics Conference*, Honolulu, Hawaii, June 2011.

[34] W. Ma, Z. Lu, and J. Zhang, "GPU parallelization of unstructured/hybrid grid ALE multigrid unsteady solver for moving body problems," *Computers & Fluids*, vol. 110, pp. 122–135, 2015.

[35] Y. Wang, J. Jiang, H. Zhang et al., "A scalable parallel algorithm for atmospheric general circulation models on a multicore cluster," *Future Generation Computer Systems*, vol. 72, pp. 1–10, 2017.

[36] B. Baghapour, A. McCall, and C. J. Roy, "Multilevel parallelism for CFD codes on heterogeneous platforms," in *Proceedings of the 46th AIAA Fluid Dynamics Conference*, Washington, DC, USA, June 2016.

[37] J. Blazek, *Computational Fluid Dynamics: Principles and Applications*, Elsevier, Amsterdam, Netherlands, third edition, 2015.

[38] F. R. Menter, "Two-equation eddy-viscosity turbulence models for engineering applications," *AIAA Journal*, vol. 32, no. 8, pp. 1598–1605, 1994.

[39] D. C. Wilcox, *Turbulence Modeling for CFD*, DCW Industries, La Cañada Flintridge, CA, USA, third edition, 2006.

[40] W. P. Jones and B. E. Launder, "The prediction of laminarization with a two-equation model of turbulence," *International Journal of Heat and Mass Transfer*, vol. 15, no. 2, pp. 301–314, 1972.

[41] M.-S. Liou, "A sequel to AUSM, part II: AUSM+-up for all speeds," *Journal of Computational Physics*, vol. 214, no. 1, pp. 137–170, 2006.

[42] B. Van Leer, "Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method," *Journal of Computational Physics*, vol. 135, pp. 101–136, 1997.

[43] G. D. Van Albada, B. Van Leer, and W. W. Roberts, "A comparative study of computational methods in cosmic gas dynamics," *Astronomy & Astrophysics*, vol. 108, pp. 439–471, 1982.

[44] C.-W. Shu and S. Osher, "Efficient implementation of essentially non-oscillatory shock-capturing schemes," *Journal of Computational Physics*, vol. 77, no. 2, pp. 439–471, 1988.

[45] W. P. Ma, Z. H. Lu, W. Yuan, and X. D. Hu, "Parallelization of an unsteady ALE solver with deforming mesh using Open ACC," *Scientific Programming*, vol. 2017, Article ID 4610138, 16 pages, 2017.

[46] J. Q. Lai, H. Li, Z. Y. Tian, and Y. Zhang, "A multi-GPU parallel algorithm in hypersonic flow computations," *Mathematical Problems in Engineering*, vol. 2019, Article ID 2053156, 15 pages, 2019.

[47] J. Q. Lai, Z. Y. Tian, H. Yu, and H. Li, "Numerical investigation of supersonic transverse jet interaction on CPU/GPU system," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 42, 2020.

[48] P. S. Rakić, D. D. Milašinovic, Ž. Živanov, Z. Suvajdžin, M. Nikolić, and M. Hajduković, "MPI–CUDA parallelization of a finite-strip program for geometric nonlinear analysis: a hybrid approach," *Advances in Engineering Software*, vol. 42, pp. 273–285, 2011.

[49] F. Schmitt, R. Dietrich, and G. Juckeland, "Scalable critical path analysis for hybrid MPI-CUDA applications," in *Proceedings of the 2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops*, Phoenix, AZ, USA, May 2014.

[50] L. Jiang, C.-L. Chang, M. Choudhari, and C. Liu, "Instability-wave propagation in boundary-layer flows at subsonic through hypersonic mach numbers," *Mathematics and Computers in Simulation*, vol. 65, no. 4-5, pp. 469–487, 2004.

[51] L. Jiang, M. Choudhari, C. L. Chang, and C. Q. Liu, "Numerical simulations of laminar-turbulent transition in supersonic boundary layer," in *Proceedings of the 36th AIAA Fluid Dynamics Conference and Exhibit*, San Francisco, CA, USA, June 2006.

[52] J. Watkins, J. Romero, and A. Jameson, "Multi-GPU, implicit time stepping for high-order methods on unstructured grids," in *Proceedings of the 46th AIAA Fluid Dynamics Conference*, Washington, DC, USA, June 2016.