Cluster Comput (2014) 17:255–270
DOI 10.1007/s10586-013-0332-1

CPU/GPU computing for a multi-block structured grid based high-order flow solver on a large heterogeneous system

Wei Cao · Chuan-fu Xu · Zheng-hua Wang · Lu Yao · Hua-yong Liu

Received: 14 February 2013 / Revised: 31 August 2013 / Accepted: 3 November 2013 / Published online: 27 November 2013

Abstract The high-order schemes have attracted more and more attention in computational fluid dynamics (CFD) simulations. As a kind of high-order schemes, weighted compact nonlinear schemes (WCNSs) have been widely applied in large eddy simulations, direct numerical simulations etc. However, due to the computational complexity, WCNSs require high-performance platforms. In recent years, the highly parallel graphics processing unit (GPU) is rapidly gaining maturity as a powerful engine for high performance computer. In this paper, we present a high-order double-precision solver of the three-dimensional, compressible viscous flow using multi-block structured grids on GPU clusters. The solver utilizes the high-order WCNS scheme for space discretization and Jacobi iteration method for time discretization. In order to utilize the computational capability of CPU and GPU for the solver, we present a workload balancing model for distributing workload among CPUs and GPUs. And we design two strategies to overlap computations with communications. The performance analyses show that the single-GPU solver achieves about 8× speed-ups relative to a serial computation on a CPU core. The performance results validate the workload distribution scheme. The strong and weak scaling analyses show that GPU clusters offer a significant advantage in performance.

Keywords CPU/GPU computing · Hybrid MPI-OpenMP-CUDA · CFD · WCNS

W. Cao (✉) · C.-f. Xu · Z.-h. Wang · L. Yao
School of Computer, National University of Defense Technology, Changsha 410073, China
e-mail: caowei193@gmail.com

C.-f. Xu
e-mail: xuchuanfu@nudt.edu.cn

Z.-h. Wang
e-mail: zhhwang188@sina.com

L. Yao
e-mail: shaoeq@163.com

H.-y. Liu
State Key Laboratory of Aerodynamics, Mianyang 621000, China
e-mail: hyliu3@sohu.com

1 Introduction

Computational fluid dynamics (CFD) has undergone great development as a discipline for 50 years. Although low-order accurate schemes are widely used for engineering applications, they are insufficient for physically-complex flows. Compared with low-order accurate schemes, the high-order accurate schemes yield lower numerical dissipation and dispersion, and are capable of producing much more accurate results. As such, the high-order schemes have attracted more and more attention in CFD simulations.

The high-order schemes were first designed in 1970s and 1980s. Harten introduced the concept of Total Variation Diminishing (TVD) difference schemes in 1983 [1]. Van Leer proposed MUSCL scheme [2], which was satisfied the TVD condition by applying the limiter. Although TVD schemes succeed in CFD simulations, the TVD schemes are second-order accurate and only first-order accurate at local extrema. In order to improve the accuracy, Harten introduced the Essentially Non-Oscillatory (ENO) schemes [3]. By using a stencil-weighting approach, Liu proposed the Weighted Essentially Non-Oscillatory (WENO) schemes [4] in order to simplify the ENO procedures. In recent years, high-order compact schemes (CS) [5], which compute the derivatives simultaneously along an entire line in a coupled fashion, were developed. As a kind of combination schemes of WENO and CS, a series of WCNSs [6] were proposed by Deng et al. based on weighted technique. The WCNSs include both implicit and explicit schemes. As one of the explicit WCNS, WCNS-E-5 is of fifth-order accuracy in smooth region and third-order accuracy in the vicinity of discontinuities. The experimental results showed that WCNS-E-5 captures the discontinuities robustly and precisely, and can be applied for solving complex flow problems [7]. Nowadays, WCNS have been widely applied in large eddy simulations (LES) [8], direct numerical simulations (DNS) [9], etc.

Although the significant computational requirement becomes a barrier for three dimensional high-order CFD applications, these applications in conjunction with high performance computers make it possible. Recently, in order to improve the performance of computers, system architects are moving away from traditional clusters of homogeneous nodes to clusters of heterogeneous nodes which are augmented with latest GPUs. As an example, the number one system on the November 2010 top-500 list was Tianhe-1A [10] which was composed of 14336 CPUs and 7168 GPUs. And novel uses of GPU with the NVIDIA's Compute Unified Device Architecture (CUDA) have revealed the potential of general-purpose GPU (GPGPU) computing. Following this trend, researchers accelerated scientific applications, including high-order accurate CFD applications in the heterogeneous environment. Depending on the numerical methods used and the ease of parallelization, the high-order accurate CFD applications ported to the GPU have acquired different speedups.

Hai P. Le et al. [11] reported an implementation of a numerical solver which incorporated high-order finite volume methods for solving the fluid dynamical equations coupled with stiff source terms on the GPU. The solver used high-order shock capturing schemes MP5 and ADERWENO. Considering only the fluid dynamics, the speedup factors obtained were respectively 30 and 55. For the chemical kinetics, the speedup ranged from 30 to 40.

In [12], a sixth-order compact finite difference scheme for solving the 2D advection equations and the 3D Navier-Stokes equations solution was implemented on GPU. Speedups between 9 and 16.5 were achieved on GPU compared to CPU computations.

Vahid Esfahanian et al. [13] studied the application of high-order shock-capturing WENO schemes to some hyperbolic equations using GPU. The comparison of the speedups showed the obtained speedups for the WENO schemes were more than that of the first-order upwind method (FTBS) scheme. The results also showed the speedups were even considerable for GT8500 GPU and could reach to several hundred for GTX480 GPU.

M. Geveler et al. [14] described FE-gMG, a geometric multigrid combined with high-order finite element approach for problems relying on unstructured grids. For a Poisson problem and computational grids in 2D and 3D, they achieved a speedup of an average of 5 on a single GPU over a multithreaded CPU code in their benchmark.

Furthermore, the CFD applications were extended to scale to large GPU clusters. In 2011, D. Jacobsen et al. [15] showed the implementation of the Navier-Stokes equations to simulate buoyancy-driven incompressible fluid flows using Jacobi iterative solver on multi-GPU Clusters. By using dual-level and tri-level parallelism, the weak scale efficiency of the implementations on 128 GPUs only achieved 17 % and 19 %, respectively. In [16], the work was extended to scale to 256 GPUs, and the parallel efficiency of the implementation achieved 94 % in one dimensional scaling case and 17 % in three dimensional scaling case. This was the largest scale GPU parallel computing platform as we know that was used to accelerate CFD applications. However, they only focused on low order accurate schemes.

L.H. Han et al. [17] presented a wavelet-based multi-resolution solver designed for solving two-dimensional compressible Euler equations on heterogeneous parallel architecture. With 6th-order WENO scheme, the simulation using 2 GPUs obtained an about 32 times speedup compared to using single CPU.

Antoniou et al. [18] presented a single-node, 4-GPU implementation based on WENO scheme for solving the Favre-averaged Navier-Stokes equations. The numerical code included options for 5th, 7th and 9th order accurate discretizations of the inviscid fluxes and used third order accurate TVD Runge-Kutta method for time marching. The application ran in single precision, and the parallel run achieved a 53× speedup on the average for several mesh sizes with 4 GPUs in comparison with the sequential run.

J. Appleyard et al. [19] carried out High-Order Upstream Central (HOUC) scheme to solve the level set equation on two different computational systems: a traditional CPU cluster and a system of 4 GPU controlled by a single quad-core CPU. 3rd, 5th, 7th and 9th-order HOUC schemes were employed together with a 3rd-order TVD Runge-Kutta time integration. The increase in performance of two orders of magnitude was seen when comparing a single CPU core to a single GPU and was found to be much greater at higher orders of accuracy.

Peng Wang et al. [20] mapped high resolution shock capturing schemes (HRSC) for hyperbolic conservation laws to GPU using CUDA. The framework as applied to the equations of inviscid compressible hydrodynamics on single GPU, could achieve approximately 10 times faster execution on one graphics card as compared to a single core on the host computer. And it achieved very good, close to ideal speedup for up to four GPUs.

Michael Griebel et al. [21] showed the first application of multi-node/multi-GPU computations in a high-order engineering targeted solver for the full non-stationary incompressible Navier-Stokes equations with fifth order WENO scheme. Overall speedup of 8-GPUs accelerated fluid solver compared to one CPU core for a grid resolution of 300^3 was 69.6.

However, on one hand, the above studies only considered computational capacity of GPU, while the computational capacity of CPU was ignored. Lu et al. [22] performed a long-wave radiation simulation by exploiting the computational capacities of both CPUs and GPUs in the Tianhe-1A supercomputer. They used the application speedup on GPU over CPU to distribute the workload. But their methods did not take the communicational cost between GPU and CPU into consideration. On the other hand, most high-order accurate simulations on GPU clusters concentrated on single-block mesh or incompressible flow or inviscid flow. While multi-node/multi-GPU computations for high-order, compressible viscous flow solver on multi-block structured mesh are really rare.

In this paper, we develop a tri-level parallelization of a three-dimensional, high-order, compressible viscous flow solver for multi-block structured grids on GPU clusters by using message passing interface (MPI), OpenMP and CUDA. This implementation incorporates an efficient management of CPU and GPU resources for computation and mechanisms to overlap computation with communication.

The paper is organized as follows. In Sect. 2, the numerical approach for solving dynamic fluids, including the formulation of the WCNS, is described briefly. Section 3 presents details of the GPU implementation. Section 4 extends our solver to CPU/GPU heterogeneous environment with a mixed OpenMP-CUDA implementation. Section 5 extends our solver to CPU/GPU clusters with a tri-level parallelization. In Sect. 6, the implementation is tested with several aerodynamic configurations to demonstrate the efficiency of the approach. Conclusions and possible future work are summarized in Sect. 7.

2 Governing equations and numerical approach

2.1 Governing equations

We consider the dimensionless Navier-Stokes equations for viscid, compressible flow in generalized coordinate,

$$\frac{\partial \hat{Q}}{\partial t} + \left(\frac{\partial \hat{F}_c}{\partial \xi} + \frac{\partial \hat{G}_c}{\partial \eta} + \frac{\partial \hat{H}_c}{\partial \zeta}\right) - \frac{1}{Re_{ref}}\left(\frac{\partial \hat{F}_v}{\partial \xi} + \frac{\partial \hat{G}_v}{\partial \eta} + \frac{\partial \hat{H}_v}{\partial \zeta}\right) = 0 \tag{1}$$

In Eq. (1), Q is the vector of conserved variables, density ρ, momentum u, v, w, and total energy per unit volume e, such that $\hat{Q} = \frac{Q}{J} = \frac{1}{J}(\rho, \rho u, \rho v, \rho w, \rho e)^T$, and $\hat{F}_c$, $\hat{G}_c$, $\hat{H}_c$ are inviscid flux terms, and $\hat{F}_v$, $\hat{G}_v$, $\hat{H}_v$ are viscous flux terms. Re_{ref} is the reference Reynolds number. The variable J represents the Jacobian of the transformation between the Cartesian variables (x, y, z) and the generalized coordinated (ξ, η, ζ).

2.2 Numerical approach

In order to solve the governing equations of the fluid motion, a common solution is to discretize and linearize the equations. For spatial discretization, the numerical algorithm uses a semi-discrete cell-centered finite difference scheme in this paper. Viscous flux terms are discretized by using high-order central difference scheme. Inviscid flux terms are discretized by using high-order WCNSs which are much more complicated than central difference scheme, and will be discussed in more detail.

The WCNSs procedure consists of three components: nonlinear interpolation, flux evaluation and linear differencing. Let us only consider the $(2r-1)$th order discretization of the inviscid flux derivative along the η direction. In the first component of procedure, $\widetilde{Q}^L_{j+1/2}$ and $\widetilde{Q}^R_{j+1/2}$, which are the left-hand and right-hand cell-interface flow variables, respectively, are obtained by a high-order nonlinear weighted interpolation. The interpolations are usually approximated in the characteristic fields. First, conservative variable Q_k is transformed to characteristic variables $\mathcal{Q}_k$:

$$\mathcal{Q}_{k,m} = l_{j,m} Q_k \quad (k = j - r + 1, \dots, j + r - 1) \tag{2}$$

where $\mathcal{Q}_{k,m}$ denotes the mth characteristic variable and $l_{j,m}$ denotes the mth left eigenvector of the matrix $A = \partial G/\partial Q$ on the jth mesh point. Hereafter the construction of $\widetilde{\mathcal{Q}}^L_{j+1/2}$ is only noted, while $\widetilde{\mathcal{Q}}^R_{j+1/2}$ can be computed symmetrically. Second, the general rth-order interpolation of $\widetilde{\mathcal{Q}}^L_{j+1/2}$ in $j + 1/2$ can be written as:

$$\mathcal{Q}^L_{j+\frac{1}{2},k,m} = \mathcal{Q}_{j,m} + \sum_{n=1}^{r-1}\left(\frac{1}{n!}\right)\left(\frac{h}{2}\right)^n \mathcal{Q}^{(n)}_{j,k,m} \tag{3}$$

where $\mathcal{Q}^{(n)}_{j,k,m}$ is the nth derivative for the mth characteristic variable using the kth $(k = 1, 2, \dots, r)$ stencil. Then, combining these values together with the weights, we get:

$$\mathcal{Q}^L_{j+\frac{1}{2},m} = \sum_{l=1}^{r} w_k \mathcal{Q}^L_{j+\frac{1}{2},k,m} \tag{4}$$

Finally, the variables at cell interfaces can be obtained as:

$$\widetilde{Q}^L_{j+\frac{1}{2}} = \sum_m \mathcal{Q}^L_{j+\frac{1}{2},m} r^m_j \tag{5}$$

In the second component of the WCNS procedure, a WCNS numerical flux is evaluated based on:

$$\widetilde{G}_{j+\frac{1}{2}} = f\left(\widetilde{Q}^L_{j+\frac{1}{2}}, \widetilde{Q}^R_{j+\frac{1}{2}}\right) \tag{6}$$

There are lots of flux evaluation methods that can be applied in WCNS, e.g. Steger-Warming or Van Leer flux vector split (FVS) scheme, Roe's flux difference splitting method (FDS), and AUSMPW+ scheme. The numerical flux obtained here is an approximation of the exact flux at the cell interface, as follows:

$$\widetilde{G}_{j+\frac{1}{2}} = G_{j+\frac{1}{2}} + O\left(h^{2r-1}\right) \tag{7}$$

The third component computes the derivative of flux with a high-order linear scheme which can be implicit and explicit. The fifth-order explicit WCNS-E-5 can be expressed as:

$$\begin{aligned}\frac{\partial \widetilde{G}_j}{\partial \eta} &= \frac{75}{64h}(\widetilde{G}_{j+\frac{1}{2}} - \widetilde{G}_{j-\frac{1}{2}}) - \frac{25}{384h}(\widetilde{G}_{j+\frac{3}{2}} - \widetilde{G}_{j-\frac{3}{2}}) \\ &\quad + \frac{3}{640h}(\widetilde{G}_{j+\frac{5}{2}} - \widetilde{G}_{j-\frac{5}{2}}) \end{aligned} \tag{8}$$

And the implicit WCNS-5 can be expressed as:

$$\begin{aligned}&\frac{9}{62}\frac{\partial \widetilde{G}_{j+1}}{\partial \eta} + \frac{\partial \widetilde{G}_j}{\partial \eta} + \frac{9}{62}\frac{\partial \widetilde{G}_{j-1}}{\partial \eta} \\ &\quad = \frac{63}{62h}(\widetilde{G}_{j+\frac{1}{2}} - \widetilde{G}_{j-\frac{1}{2}}) - \frac{17}{186h}(\widetilde{G}_{j+\frac{3}{2}} - \widetilde{G}_{j-\frac{3}{2}}) \end{aligned} \tag{9}$$

After spatial discretization, Eq. (1) can be written as:

$$\frac{\partial \hat{Q}}{\partial t} = R(J, Q) \tag{10}$$

where $R(J, Q)$ is called right-hand side (RHS). The solution is advanced in time with implicit methods, such as Runge-Kutta method, or explicit methods, such as Jacobi method and LU-SGS method.

2.3 Specification of the CFD

According to the computational procedure, a CFD numerical simulation can be divided into 3 steps: preprocessing step, solving step, and post-processing step. In the preprocessing step, the simulator inputs the mesh and defines the setup of the simulation case. In this paper, we employ the one-to-one blocking mesh, which means that the faces shared by two mesh blocks are exactly the same. For boundary cells, additional buffer zones (often referred as the "ghost zones") for finite-difference stencils are added around each block to store the values of cells in the neighboring blocks. As a simple illustration of one-to-one blocking mesh and ghost zones, consider Fig. 1. The black dots represent ghost cells, the white dots represent the corresponding boundary cells, and the grey dots represent the interface cells. The application of very high-order schemes in this context lead to an increased depth of the ghost zones required. For example, 5th-order WCNS require 5 ghost cells on each side of computational domain. The data at ghost

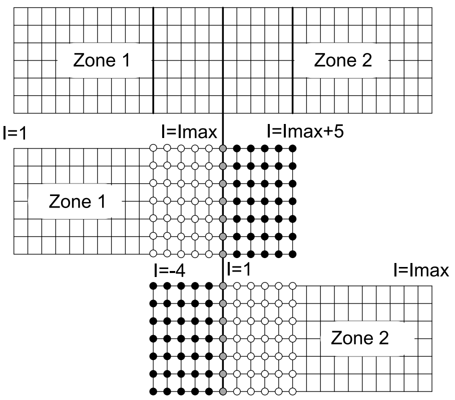


Fig. 1 The one-to-one blocking mesh. The *black cells* mean ghost cells, and the white cells mean boundary cells, the *grey cells* mean the interface meshes

cells need to be exchanged between blocks. For a N^3 cells per block, the total number of boundary cells which should be sent/received by each block utilizing a 5th-order scheme is $\sim 30N^2$.

In the solving step, approximate numerical solution for the flow is obtained by solving the governing equations. The solving step is the performance-critical section of simulator. The procedure of the solving step is illustrated as Fig. 2. In each step, the program loops over each mesh block to calculate values, and then take the next step. In order to maintain the accuracy of high-order scheme, the values of primitive variables, gradients of primitive variables, RHS, and increment of conserved variables at block boundaries must be exchanged with its neighbors, and be averaged with its original values.

After the solving step, the post-processing analyzes the results and calculates derived quantities e.g. force, heat, and outputs the results to files.

3 Parallel algorithm on GPU

According to Amdahl's law, the parallel algorithm for the solver, which is the performance critical section of the simulator, is studied in this paper. We investigate the memory access patterns and data dependencies, and design different GPU parallel algorithms for different patterns. The implementation and optimization of these algorithms to harness the compute-power of GPUs will see in Sect. 6.

Traditionally, the multi-block CFD parallel computing on CPUs is based on domain decomposition, in which each CPU handles one or more mesh blocks separately. GPUs can run hundreds of thousands of threads concurrently and hide the memory access latency by quickly switching from one thread to another. So it is not suitable for GPU parallelization to adopt the method for CPU parallelization. Considering CUDA programming features and data dependencies of

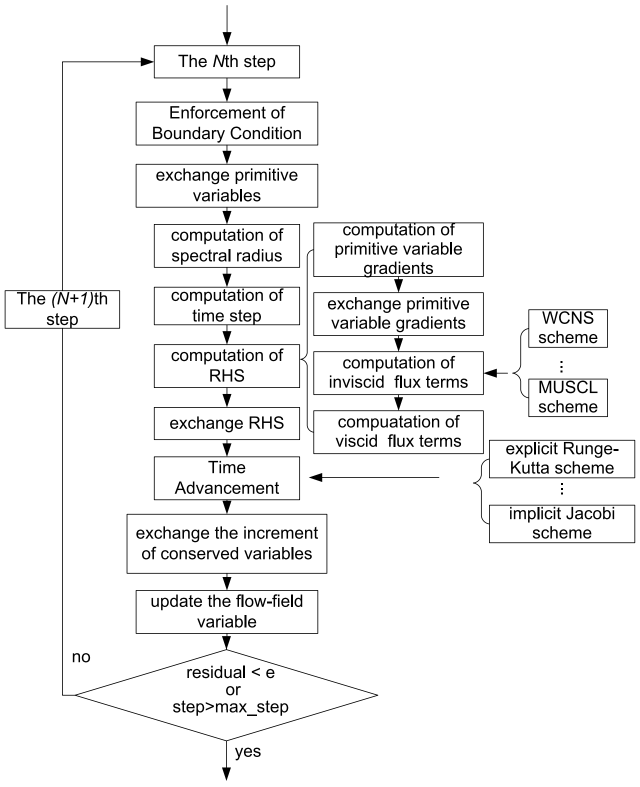

The *N*th step
Enforcement of Boundary Condition
exchange primitive variables
computation of spectral radius
computation of time step
computation of RHS
exchange RHS
Time Advancement
exchange the increment of conserved variables
update the flow-field variable
residual < e or step>max_step
no
yes
The *(N+1)*th step
computation of primitive variable gradients
exchange primitive variable gradients
computation of inviscid flux terms
compuatation of viscid flux terms
WCNS scheme
MUSCL scheme
explicit Runge-Kutta scheme
implicit Jacobi scheme

Fig. 2 Flow chart of the solver. In exchange procedure, the adjacent blocks exchange the values at boundaries, and in other procedures, the solver loops over all blocks to compute the values

the CFD computing, we design the fine grain data parallel algorithm based on cells. Take the computation of spectral radius and the computation of inviscid flux terms as examples, we discuss GPU parallel algorithms in detail.

The computation of spectral radius computes the spectral radius of Jacobian matrix. Take spectral radius of Jacobian matrix corresponding to inviscid flux $\hat{F}_c$ as an example, it is calculated by:

$$\lambda = \theta + a\sqrt{\xi_x^2 + \xi_y^2 + \xi_z^2} \tag{11}$$

where

$$\theta = \xi_x u + \xi_y v + \xi_z w \tag{12}$$

where (u, v, w) is velocity vector, ξ_x, ξ_y, ξ_z are coordinate metrics, and a denotes the speed of sound. The spectral radius in a cell is only data-dependent on the values of variables in this cell, and is not data-dependent on the values of variables in other cells. So the spectral radius computation can be parallelized on a per-cell basis, with one thread per cell. In order to associate to the 3D structured mesh, the threads are organized into 3D blocks, and then the blocks are organized into a 3D grid, as illustrated in Fig. 3.

The computation of inviscid flux terms computes five values: the coordinate derivatives at the cell interfaces, the primitive variables at cell interfaces, fluxes in the interior cells, fluxes at the cell interfaces, and the inviscid flux derivatives, in each cell along 3 dimensions. Except the coordinate derivative, each variable in a cell is dependent on the values of previous variable in its neighbor cells. Take the 5th order WCNS scheme as an example, the inviscid flux derivative in the cell i along the ξ dimensional is dependent on the fluxes at the cell $i \pm 1/2$, $i \pm 3/2$, $i \pm 5/2$ interfaces. If we assign to a thread all the computation of inviscid flux terms associated to a cell, then these threads may belong to different thread blocks (Fig. 4(a)), so there is a need of global synchronization across all threads to force the data coherence. Since the CUDA model is lack of efficient global synchronization mechanism, we consider two different ways to support the fine-grain parallelization. One is the redundant computation method and the other one is kernel decomposition method.

The redundant computation method means to compute values in the cells that are associated to the thread block boundaries twice, by the two neighbored thread blocks (Fig. 4(b)). It's easy to ensure that writing thread has write the data before the other one read it by synchronization among threads in the same thread block. Although the redundant computation method decreases the cost of kernel launch, it brings more extra calculations, especially when the size of thread block becomes smaller and the computing accuracy gets higher. And the redundant computation brings more branches for dealing with computation at the thread block boundaries, which can decrease the performance of the kernel. Also, the redundant computation method increases the source code size, while the resource that a kernel requires is proportional to source code size. If there is insufficient register space to hold the variable, automatic variables are likely to be placed in local memory. However, the cost of local memory access is as expensive as the cost of global memory access, which is much cheaper than the cost of register access. And as the resource that a kernel requires increases, the maximum number of active threads that we can schedule for the kernel is not sufficient to hide the memory access latency.

Considering the above features, kernel decomposition method which splits computation into several kernels is the better way (Fig. 4(c)). For the computation of inviscid flux terms, we decompose the computation into three parts based on the calculation along three dimensions. Then according to data dependency, each part is split into several kernels to ensure that there is no data dependence in each kernel. Between two kernels, an explicit synchronization is needed. With the kernel decomposition method, the values can be computed as the computation of spectral radius.

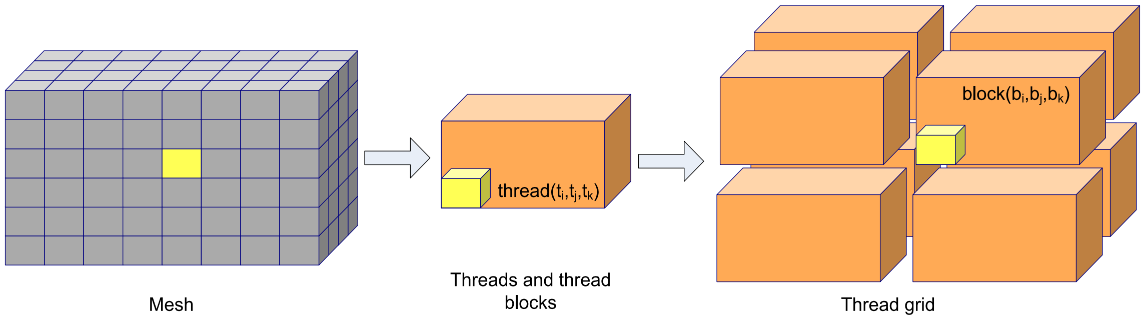
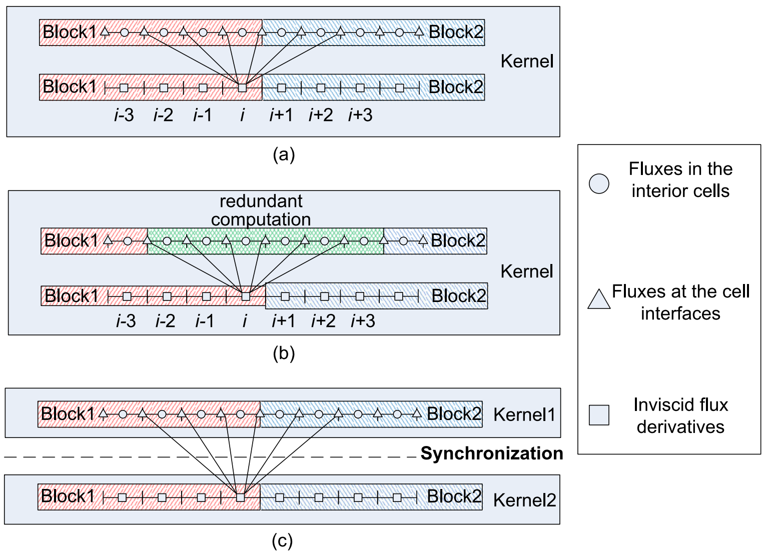


Fig. 3 The GPU threads organization on a per-cell basis

Fig. 4 The nonlinear interpolation of WCNS on GPU. (**a**) The organization on a per-cell basis; (**b**) The redundant computation method; (**c**) The kernel decomposition method



4 Parallel algorithm on GPU/CPU

Computing resources in GPU clusters include multi-core CPUs and many-core GPUs. Usually, CPUs and GPUs cooperate as the way that CPUs prepare and transfer data and GPUs perform arithmetic operations. However, the computational capability of CPUs is ignored in this way. While in most high performance systems, the number of GPUs is less than the number of cores of CPUs. Hence, we can use one CPU core to prepare and transfer data for one GPU, and use the rest CPU cores to perform arithmetic operations.

For multi-block mesh, CFD computation within each block can be computed independently, providing coarse-grain parallelism. Fine-grain loop-level parallelism can be exploited within each block. Considering these, we propose a two-level parallelization approach. The outer parallelization exploits the coarse-grain parallelism across mesh blocks. And the inner parallelization exploits the fine-grain parallelism inside mesh block.

Assume there are N_g GPUs and N_c CPUs in a node. Each CPU is consisted of n cores, and N_g is less than $N_c \times n$. The process spawns $N_g + 1$ threads by employing OpenMP to exploit coarse-grain parallelism. Among them, N_g threads which run on N_g CPU cores are used to cooperate with GPUs, and 1 thread is used to cooperate with the rest CPU cores. In the inner parallelization, the GPU exploits the fine-grain parallelism by using CUDA, and the CPU spawns $(N_c \times n - N_g)$ threads to exploit the fine-grain parallelism by using nested OpenMP. The CPU/GPU cooperative model is illustrated in Fig. 5.

In order to utilize the two-level parallelization efficiently, it is important to distribute the proper workload to processors according to the computational capability of each computing unit. Assuming there are W mesh blocks to distribute, and all mesh blocks have the same size. Let n_x, n_y and n_z be the number of computational cells in the ξ, η and ζ directions for a mesh block, respectively. Let P_{cpu} denote the computing power of each CPU core, and P_{gpu} denote the computing power of GPU. W_{cpu} is the number of blocks that are distributed to CPU and W_{gpu} is the number of blocks that are distributed to a GPU. The wall time for CPU/GPU computing is:

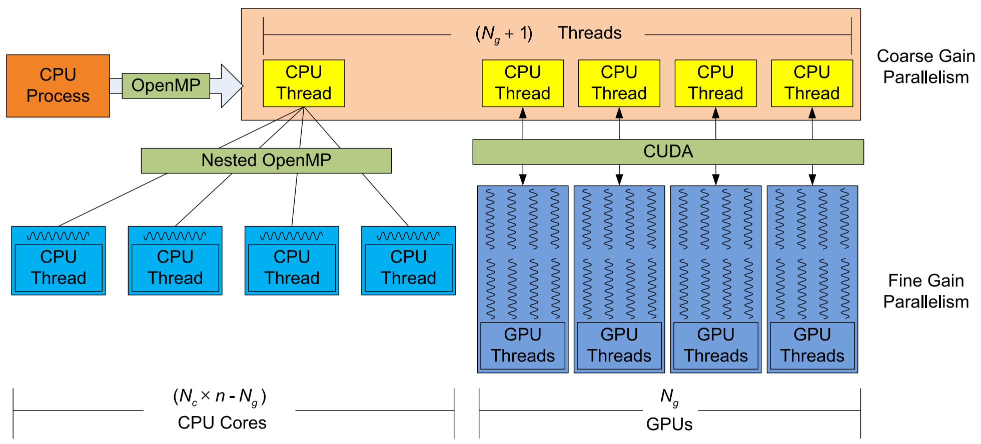


Fig. 5 The CPU/GPU cooperative model

$$T_{cpu} = W_{cpu} \cdot (n_x \cdot n_y \cdot n_z)/P_{cpu} \cdot \left[1 - f + f/(N_c \cdot n - N_g)\right] \tag{13}$$

$$T_{gpu} = W_{gpu} \cdot (n_x \cdot n_y \cdot n_z)/P_{gpu} + T_{comm} \cdot W_{gpu} \tag{14}$$

$$W = W_{cpu} + W_{gpu} \cdot N_g \tag{15}$$

where T_{comm} denotes the communicational time between GPU and CPU. In order to maintain the accuracy of the high-order scheme, the values of primitive variables, gradients of primitive variables, RHS, and increment of conserved variables at the mesh block boundaries must be exchanged with its adjacent blocks. But usually, the data at ξ–ζ and η–ζ planes are stored in non-continuous mode. As illustrated in Fig. 6, to transfer these data, we allocate a device memory buffer and a host memory buffer for each boundary. The sender packs the data at boundary cells into the buffer, and transfers it to the receiver. The receiver unpacks the data and stores them in boundary cells.

So we have:

$$T_{comm} = T_{htod} + T_{dtoh} \tag{16}$$

$$T_{htod} = S/Bw + L_{htod} \tag{17}$$

$$T_{dtoh} = S/Bw + L_{dtoh} \tag{18}$$

where T_{htod} is the communicational time for data transfer from host to device. T_{dtoh} is the communicational time for data transfer from device to host. S is the size of the exchanged data. Bw is the bandwidth between host and device. For PCIE2.0-x16, the bi-directional bandwidth is bounded at 20 GB/s. L_{htod} denotes packing/unpacking time for the data from host to device, and L_{dtoh} denotes packing/unpacking time for the data from device to host.

And we have:

$$S = 2 \cdot (n_x \cdot n_y + n_y \cdot n_z + n_x \cdot n_z) \times n_{gh} \cdot n_{var} \cdot m \tag{19}$$

$$L_{htod} = \left[2 \cdot (n_y \cdot n_z + n_x \cdot n_z)/P_{cpu\_pack} + 2 \cdot (n_y \cdot n_z + n_x \cdot n_z)/P_{gpu\_unpack}\right] \times n_{gh} \cdot n_{var} \tag{20}$$

$$L_{dtoh} = \left[2 \cdot (n_y \cdot n_z + n_x \cdot n_z)/P_{gpu\_pack} + 2 \cdot (n_y \cdot n_z + n_x \cdot n_z)/P_{cpu\_unpack}\right] \times n_{gh} \cdot n_{var} \tag{21}$$

where n_{gh} denotes the layer of ghost cells, and n_{var} denotes the number of variables for exchanging in each cell. m denotes the memory requirement for each flow variable, and for double precision computing, m is 8. P_{cpu_pack}, P_{gpu_pack} mean the speed of data packing on CPU and GPU, respectively. P_{cpu_unpack}, P_{gpu_unpack} mean the speed of data unpacking on CPU and GPU, respectively.

When T_{cpu} equals T_{gpu}, we get the most efficient workload distribution. Hence, we get the workload distribution:

$$S_{block} = n_x \cdot n_y \cdot n_z \tag{22}$$

$$SP_{par} = 1 - f + f/(N_c \cdot n - N_g) \tag{23}$$

$$T'_{comm} = T_{comm} \cdot P_{cpu} \cdot P_{gpu}/S_{block} \tag{24}$$

$$W_{gpu} = W \cdot P_{gpu} \cdot SP_{par} /\left(P_{cpu} + N_g \cdot P_{gpu} \cdot SP_{par} + T'_{comm}\right) \tag{25}$$

$$W_{cpu} = W \cdot \left(P_{cpu} + T'_{comm}\right) /\left(P_{cpu} + N_g \cdot P_{gpu} \cdot SP_{par} + T'_{comm}\right) \tag{26}$$

Since the device memory in most graphic cards is fixed and much smaller than host memory, so if we assume that the memory requirement is proportional to workload and the communication cost is ignored, when the workload is balanced between CPU and GPU, we have:

$$W_{cpu} = \frac{1}{\alpha} M_{cpu} \tag{27}$$

$$W_{gpu} = \frac{1}{\alpha} M_{gpu} \tag{28}$$

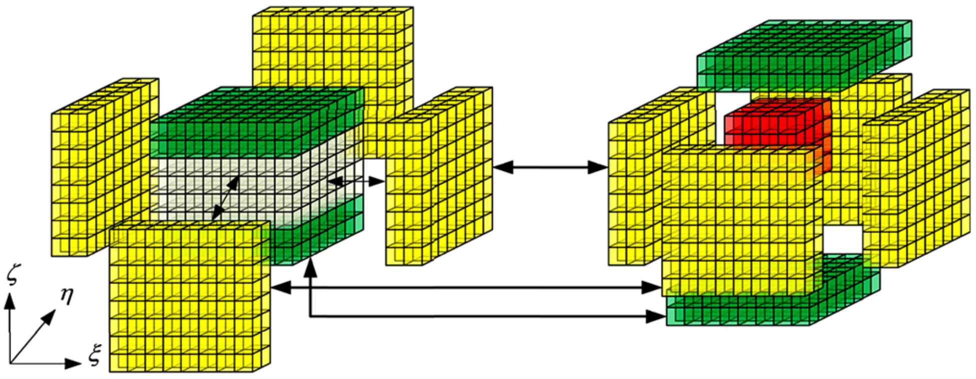


Fig. 6 The data exchange between GPU and CPU. The data at ξ–η plane can exchange directly, while the data at ξ–ζ and η–ζ planes need buffers

$$M_{cpu} = M_{gpu}\frac{P_{cpu}}{P_{gpu} \cdot SP_{par}} \tag{29}$$

$$W_{cpu} = \frac{1}{\alpha}M_{gpu}\frac{P_{cpu}}{P_{gpu} \cdot SP_{par}} \tag{30}$$

Where M_{gpu}, M_{cpu} denote the memory requirement on GPU and on CPU, respectively. α is the proportionality factor. According to Eqs. (27) and (29), the workload on CPU and the workload on GPU are limited by the device memory. In order to run bigger simulations, we propose out-of-core method to increase the simulation size on single node, and use MPI to increase the simulation size on multi-nodes.

The general idea of out-of-core method is to divide the workload distributed to GPU into several groups, and compute the solutions group by group. Only the device memory that is required by a group of mesh blocks is allocated. Before each kernel launches, the data which are needed by the kernel are copy to GPU as the input of the kernel. After the kernel finished, the data which are produced by the kernel are copy out from device memory to host memory, even the data that are produced by a kernel will be consumed by the next kernel. In this way, the workloads that can be distributed to GPU and CPU respectively are:

$$W_{gpu} = \alpha M_{gpu} \cdot N_{group} \tag{31}$$

$$W_{cpu} = \alpha M_{gpu} \cdot N_{group}\frac{P_{cpu}}{P_{gpu} \cdot SP_{par}} \tag{32}$$

However, the out-of-core method introduces more data transfer overhead. In order to reduce the overhead, on devices that are capable of “concurrent copy and execute”, we use the asynchronous transfers and streams on GPU to overlap the computation and communication between host and device. The asynchronous transfers require pinned host memory, and the data transfers and kernels must use different no-default streams. By using asynchronous transfers and streams, the workload which is consisted of several mesh blocks is transferred in multiple stages, launching multiple kernels to operate on each block as it arrives. Figure 7 illustrates the time lines of data transfers and kernel executions for workload which is consisted of 3 groups, and each group is consisted of 4 mesh blocks. With the out-of-core method, the total communicational time between CPU and GPU is proportional to the number of groups. And we have the wall time for CPU/GPU computation:

$$T_{cpu} = W_{cpu} \cdot S_{block}/P_{cpu} \cdot SP_{par} \tag{33}$$

$$T_{gpu} = W_{gpu} \cdot S_{block}/P_{gpu} + 2 \cdot T_{comm} \cdot N_{group} \tag{34}$$

$$S = (n_x + 2 \cdot n_{gh})(n_y + 2 \cdot n_{gh})(n_z + 2 \cdot n_{gh}) \cdot n_{var} \cdot m \tag{35}$$

$$L_{htod} = 0 \tag{36}$$

$$L_{dtoh} = 0 \tag{37}$$

when T_{cpu} equals T_{gpu}, we get the workload distribution with out-of-core method:

$$T''_{comm} = 2 \cdot T_{comm} \cdot N_{group} \cdot P_{cpu}/(S_{block} \cdot SP_{par}) \tag{38}$$

$$W_{gpu} = \frac{W - T''_{comm}}{P_{cpu}/(P_{gpu} \cdot SP_{par}) + N_g} \tag{39}$$

$$W_{gpu} = \frac{W \cdot P_{cpu}/(P_{gpu} \cdot SP_{par}) + N_g \cdot T''_{comm}}{P_{cpu}/(P_{gpu} \cdot SP_{par}) + N_g} \tag{40}$$

The more groups that the workload is divided into, the bigger simulation can run on GPU. However, the more groups, the more time that is spent for filling and draining pipeline, and the throughput of GPU decreases. Our goal is to maximize the performance for a fixed problem. So we first distribute the workload according Eqs. (24) and (25), and we calculate the device memory required by this workload distribution, and set the number of groups to 1. If the device memory requirement is larger than the device memory size, then the number of groups increases and workload is distributed according Eqs. (38) and (39). The pseudo code is illustrated as Algorithm 1.

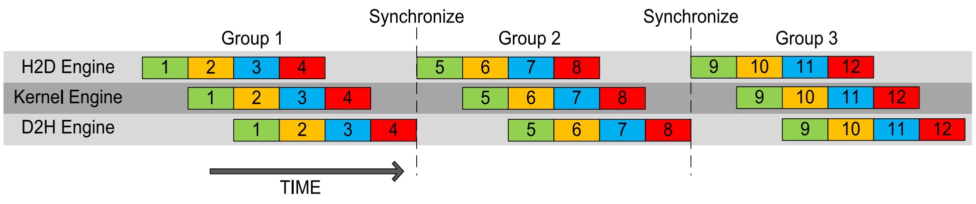


Fig. 7 Timeline view execution on GPU with out-of-core method

Algorithm 1 The workload distribution, and the out-of-core execution on GPU

```
1:  procedure FUNC_OUT_OF_CORE
2:  $W_{gpu}$ ←Get_workload()                         ▷Workload distribution according Eqs. (25) and (26)
3:  $num\_groups$ ← 1
4:  $num\_streams$ ← $W_{gpu}$
5:  $M_{gpu}$ ← Get_req_ memory($W_{gpu}$)             ▷Calculate the required device memory of $W_{gpu}$ workload
6:  while num_groups≤ $W_{gpu}$ and $M_{gpu}$>MEM$_{gpu}$ do
7:      num_groups num_gpoups+1
8:      $W_{gpu}$ ←Get_workload_MS()                  ▷Workload distribution according Eqs. (39) and (40)
9:      num_streams← ⌈$W_{gpu}$/num_groups ⌉
10:     $M_{gpu}$ ←Get_req_memory(⌈$W_{gpu}$/num_groups⌉)
11: end while
12: if num_grpus >$W_{gpu}$ then                      ▷The mesh size is too large to handle by single compute node
13:     Print("The block size is too big to simulate")
14:     Exit
15: end if
16: Func_alloc_Mem($M_{gpu}$)
17: for $i$ ← 1, num_ groups do
18:     for $j$ ← 1, num_ streams do
19:         cudaMemcpyAsync(device($i$*num_groups + $j$), host($i$*num_groups + $j$), mem_size, stream($j$))
                                                     ▷Asynchronous data transfer from CPU to GPU
20:         Kernel<<<block, grid, 0, stream($j$)>>>()  ▷Kernel execution
21:         cudaMemcpyAsync(host($i$*num_groups + $j$), device($i$*num_groups + $j$), mem_size, stream($j$))
                                                     ▷Asynchronous data transfer from GPU to CPU
22:     end for
23: cudaSynchronize()                                 ▷Waiting for the finish of calculation of the group of workload
24: end for
25: end procedure
```

5 Parallel algorithm on GPU clusters

Although the usage of single node systems that consisted of CPUs and GPUs makes it possible to satisfy the performance requirements, the simulation size is limited to the memory of single node. We use MPI to increase the simulation size on multi-nodes. However, the Fermi GPUs does not support direct peer-to-peer communication across multiple devices in different nodes. The data transfer needs CPU-side data buffering, and then exchange by using MPI.

For simulations that do not use out-of-device memory method, we only buffer the mesh block boundaries data (BBD). Usually, the data at ξ–ζ and η–ζ planes are stored in non-continuous mode. In order to exchange these data efficiently, we pack them into device buffer. Take sending data as an example, the data at boundary cells are packed into the buffer on GPU, and then transferred to host buffer. The data are transferred to other nodes at last, as illustrated in Fig. 8.

For simulations that use out-of-core method, we have buffered all data (BAD) of a mesh block after the kernel finishes. Also considering the data at ξ–ζ and η–ζ planes

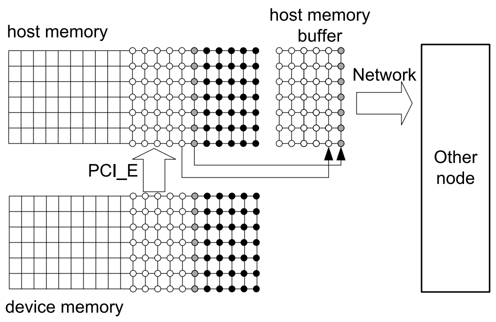


Fig. 8 Data exchanging with the buffering the boundaries data method

which are stored in non-continuous mode, we use CPU to pack/unpack the data at boundary cells, and exchange them with other nodes, as illustrated in Fig. 9.

There are potential bottlenecks which are the communication bandwidth on the PCI bus between the CPU and GPU, and the communication bandwidth on network among compute nodes. Although the bi-directional bandwidth on the PCI bus and the bi-directional bandwidth on the network

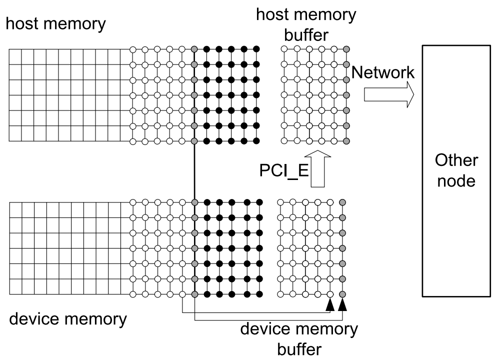


Fig. 9 Data exchanging with the buffering all data method

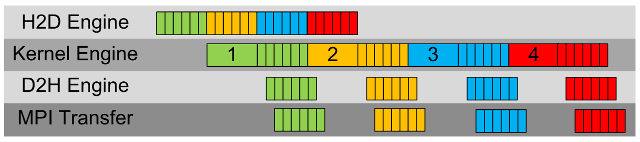


Fig. 10 Transfer and computation in unit of boundaries strategy

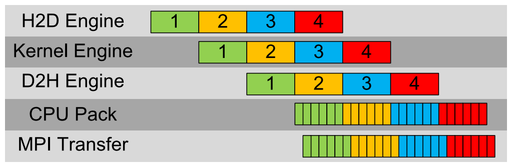


Fig. 11 Transfer and computation in unit of block strategy

can be as high as 20 GB/s, these are much lower than the bandwidth between CPU and main memory and the bandwidth between GPU and its own main memory. We design two different transfer strategies for the two transfer methods by using streams, the asynchronous data transfer between CPU and GPU, and the non-blocking MPI communication to overlap the computation and communication. Hereafter the data sending is only noted, while we can receive data symmetrically.

For the BBD transfer method, we develop the data transfer and computation in unit of boundaries (TCBO) strategy, as illustrated in Fig. 10. The TCBO strategy is a software pipeline which is consisted of 4 stages: the data at boundaries cells transfer from CPU to GPU, the kernel execution and the data packing on GPU, the data at boundaries cells transfer from GPU to CPU, and the data sending to other nodes via network.

For the BAD transfer method, we develop the transfer and computation in unit of mesh blocks (TCBL) strategy, as illustrated in Fig. 11. The TCBL strategy is a software pipeline which consists of 5 stages: the whole data of a block mesh transfer from CPU to GPU, the kernel execution on GPU, the data of a mesh block transfer from GPU to CPU, the data packing on CPU, and the data sending to other nodes via network.

6 Implementation and experimental results

In this section, we first implement the solver on Tianhe-1A supercomputer. And then we perform a numerical experiment to demonstrate the correctness of the GPU algorithm. Next the performance is evaluated with the aforementioned techniques.

6.1 Implementation

The solver which is 5th order accurate and advances in time with Jacobi method, is carried on Tianhe-1A supercomputer. The Tianhe-1A includes 7168 compute nodes connected via optic-electronic hybrid fat-tree structure network with the bi-directional bandwidth of 20 GB/s. Each compute node has two Intel E5670 (2.93 GHz, six-core) processors with 48 GB of host memory, and one NVIDIA Fermi M2050 GPU with 3GB device memory. The compiler system supports C, C + +, FORTRAN and CUDA languages, as well as MPI and OpenMP parallel program model. The hardware and software are illustrated as Table 1. The GPU and CPU codes are compiled with the optimization flag -O3. Double precision is used in all computations.

We use FORTRAN language interoperate with the CUDA C language to implement the simulation on GPU. The host code adopts the FORTRAN language, and the kernels are wrapped in functions coded by C language. A FORTRAN procedure references to a C function by calling C-code wrapper functions which call CUDA C code running on GPU.

In order to increase the performance of application on GPU, it is necessary to best utilize the GPU's memory hierarchy. We utilize arrays to store the variables of all mesh blocks. Unlike the CPU implementations of the CFD applications which usually adopt an array of structures (AoS) data layouts to increase cache utilization, CUDA implementations benefit from structure of arrays (SoA) data layouts, in which the individual components are stored contiguously for one dimension. So we use SoA data layout to store data for coalescing global memory accesses. For CFD simulations, it is difficult to use shared memory because of the data dependency and global synchronization. So we configure the on-chip memory as 16 KB of shared memory and 48 KB of L1 cache to increase cache hit ratio.

The overlap of computation and communication is devised by using nested OpenMP parallelization and massage passing interface. Thread 0 spawns 2 threads and thread 1 cooperate with the GPU. Computation on CPU is distributed to 11 threads spawned by thread 2. The asynchronous communication among nodes is implemented by nonblocking send routine (MPI_Isend) and nonblocking receive routine (MPI_Isend).

Table 1 Experimental environment

	GPU	CPU
Units per node	1	2
Cores per node	448	12
Type	NVIDIA M2050	Intel Xeon X5670
Memory	about 3 GB	48 GB
Interconnect	PCIE 2.0-x16	Fat tree network
Price	1350.00 $	(1128.75 × 2) $
Software stack	CUDA 4.0	
	MPICH2-GLEX [23]	
	Intel C++/Fortran Compiler 11.1 Release	
	Kylin Operating System	

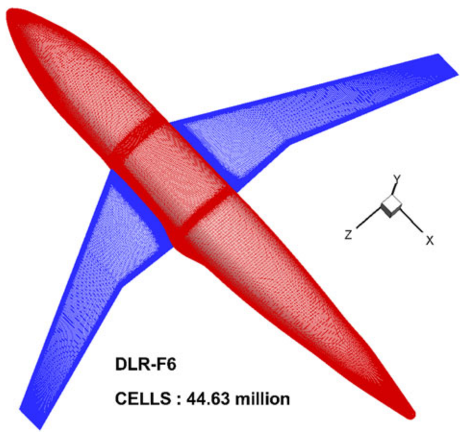


Fig. 12 DLR-F6 surface mesh

6.2 Numerical experiments

A test case is the solution for the steady flow around DLR-F6 aircraft at a Mach number of 0.75, a Reynolds number of $3e^6$, and angle of attack of $-3{-}1.5^\circ$. Three meshes resolution in this case are 4.36 million, 44.63 million and 108.99 million, respectively. As an example, the surface of the mesh with 44.63 million cells is shown in Fig. 12. The comparison physical experiment was carried out in wind tunnel.

The solver use WCNS-E-5 together with SST two-equation turbulence model. The aerodynamic coefficient compared with physical experiment is illustrated in Fig. 13. The lift coefficient and drag coefficient is closed to experiment, and lift curve slope is consistent with experiment. At the same lift coefficient, the computation result is 10–15 drag unit less than physical experiment result. The pressure coefficient at Mach number 0.75 and angle of attack of 0.49° is illustrated in Fig. 14, and is closed to physical experiment result. Especial the computation with the mesh of 44.63 million cells provides a more accurate position of shock wave.

6.3 Single GPU

The three-dimensional flow over elliptic cylinder with an axis ratio of 1/6 at a Reynolds number of $1.7e^6$ and with the angle of attack of $\alpha = 5.5^\circ$ is chosen for performance measurements. The sub-iteration of Jacobi iteration is set to be 3. The benchmark simulations use 3 kinds of mesh. The size of the first one is 128 × 64 × 64, and the meshes are divided to 1 block and 2 blocks, respectively. The size of the second one is 128 × 128 × 64 and the meshes are divided to 1 block, 2 and 4 blocks, respectively. The size of the third one is 128 × 128 × 96 and the meshes are divided to 1 block, 2, 4, 8 blocks, respectively. Figure 15 shows the running time (s) per iteration in double-precision for variety of meshes. The result shows that the computational performance on M2050 is about 8× faster than on single Intel Xeon E5670 core, and is better than the performance on two Intel Xeon CPUs. Compared with the price/performance ratio of CPU, the price/performance ratio of GPU is average about 1.85× better.

6.4 Single node

Also, the three-dimensional flow over elliptic cylinder with an axis ratio of 1/6 at a Reynolds number of $1.7e^6$ is chosen for performance measurements in this case, the angle of attack is set to $\alpha = 5.5^\circ$. The benchmark simulations use 2 kinds of mesh. The size of the first one is 128 × 128 × 64, and the mesh is divided to 4 blocks. The size of the second one is 128 × 128 × 128 and the meshes are divided to 4 and 8 blocks, respectively. Table 2 presents the parameters for workload distribution scheme. According to the workload distribution scheme, the best workload that are distributed to GPU for the 3 meshes are 2.17 blocks, 2.18 blocks and 4.36 blocks, respectively. Figure 16 shows the runtime per iteration for the 3 meshes while the workload on GPU varying. The performance result shows when the workload on GPU is balanced with the workload on CPU, the minimal runtime is achieved. And experimental results show that the

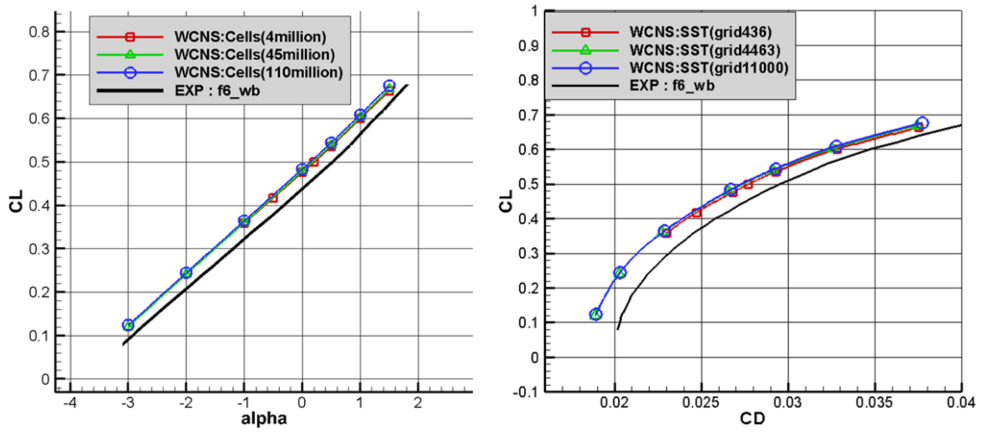


Fig. 13 The aerodynamic coefficient of DLR-F6 ($Ma = 0.75$, $\alpha = -3-1.5^\circ$)

Table 2 The parameter for workload distribution Scheme

Parameter	Value
N_c	6
n	2
N_g	1
P_{cpu}	5.69e–4 cells/s
P_{gpu}	4.73e–5 cells/s
Bw	5.5 GB/s
n_{gh}	5
n_{var}	20
$(1/P_{cpu\_pack} + 1/P_{gpu\_unpack})$	1.0e–9 cells/s
$(1/P_{gpu\_pack} + 1/P_{cpu\_unpack})$	1.0e–9 cells/s
f	0.94

best workloads distributed on GPU are approximately equal to theoretical results, which validate our workload distribution scheme. The computational performance on a single node are 1.48×, 1.61×, 1.55× speedup relative to 12 Intel Xeon CPU cores, respectively.

Through the earlier test, we can get the memory requirement per cell is about 1500 Byte. So the GPU with 3GB memory can only handle about maximum 2 million cells, and the CPU and GPU cooperation can handle about maximum 4 million cells. We test the performance on the mesh with 256 × 128 × 128 cells with out-of-core method on a single node. The mesh is consisted of 16 blocks. Table 3 shows the runtime per iteration for the workload on GPU varying from 6 to 10 for different groups. The performance results show that the more groups that the workload are divided into, the lower performance we get. When 9 blocks of mesh are distributed to GPU, the performance of overall system is the best. Table 4 illustrates the performance for several mesh resolutions. The results also validate our workload distribution scheme with out-of-core method.

6.5 Multi nodes

In order to test the parallel performance, measurements are performed for both strong scaling where the problem size remains fixed as the number of nodes increases, and weak scaling where the problem size grows in direct proportion to the number of nodes.

Figure 17 shows the speedup of the MPI-OpenMP-CUDA implementation of our flow solver relative to single compute node. The mesh resolutions considered in this test are fixed as 512 × 512 × 256 and 512 × 512 × 512. When the workload on a node is larger than 4 million cells, we adopt out-of-core method to execute the solver and the TCBL strategy to transfer data. When the nodes increase, the amount of work to do on each node drops and we use normal CPU/GPU cooperation to solve the flow problem and the TCBO strategy to transfer data. The computational performance on 128 nodes perform 12.53× and 12.43× faster than 8 nodes, respectively.

Figure 18 indicates the parallel efficiency of the implementation for weak scaling. For the mesh resolution of 128 × 64 × 64 on each node, we use only the GPU to do the calculation and the efficiency drops from 100 % with 1 node to 51.2 % with 128 nodes. For the mesh resolution of 128 × 128 × 128 on each node, we use the CPU and GPU to do the calculation and the efficiency drops from 100 % with 1 node to 62.4 % with 128 nodes. For the mesh resolution of 256 × 128 × 128 on each node, we use the CPU and GPU with out-of-core method to do calculation and the efficiency drops from 100 % with 1 node to 72.5 % with 128 nodes.

7 Conclusions and future work

In this paper, a dual-level and tri-level parallel implementations of a three-dimensional, high-order, compressible viscous flow solver for multi-block structured grids on GPU clusters by using MPI, OpenMP and CUDA, is developed. A load balancing model is presented to effectively distribute

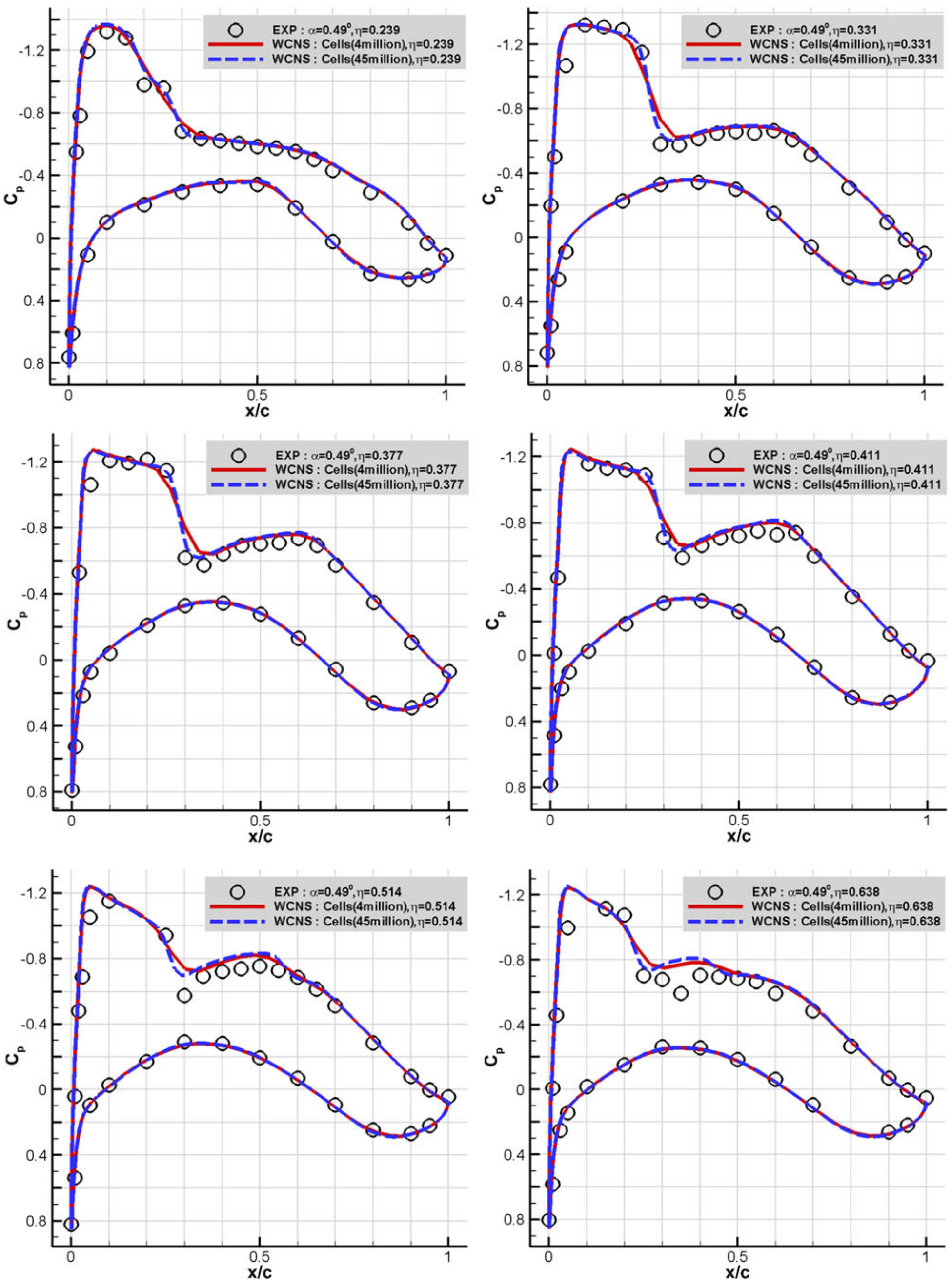


Fig. 14 The pressure coefficient compared with physical experiment ($Ma = 0.75$, $\alpha = -3-1.5°$)

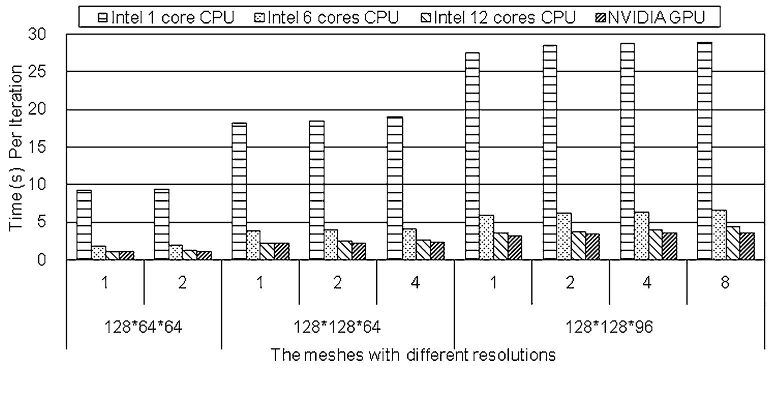


Fig. 15 Performance comparisons among different platforms

Table 3 Performance comparisons with varying groups for out-of-core method

Workload on GPU	Runtime(s) per iteration on CPU and GPU		
	2 groups	3 groups	4 groups
6	8.428 (+0.010/−0.013)	8.443 (+0.012/−0.014)	8.485 (+0.008/−0.009)
7	8.047 (+0.010/−0.011)	8.068 (+0.006/−0.013)	8.155 (+0.012/−0.015)
8	7.925 (+0.008/−0.007)	7.969 (+0.009/−0.006)	8.016 (+0.010/−0.005)
9	7.759 (+0.007/−0.013)	7.770 (+0.008/−0.009)	7.931 (+0.010/−0.014)
10	7.850 (+0.005/−0.009)	7.876 (+0.010/−0.009)	7.959 (+0.009/−0.007)

Table 4 Performance for different mesh size with out-of-core method

Mesh size	Total workload	Runtime on 1 CPU core	Runtime on 12 CPU cores	Runtime on CPU and GPU	Workload on GPU	The groups of streams	Speedup
$256 \times 192 \times 128$	6 blocks	115.668 ± 0.018 (s)	16.657 ± 0.010 (s)	10.482 ± 0.008 (s)	3	3	11.035
$256 \times 192 \times 128$	12 blocks	116.688 ± 0.019 (s)	16.821 ± 0.011 (s)	10.702 ± 0.009 (s)	6	3	10.904
$256 \times 192 \times 128$	24 blocks	118.365 ± 0.023 (s)	16.993 ± 0.013 (s)	11.942 ± 0.007 (s)	13	4	9.912
$256 \times 256 \times 128$	16 blocks	154.238 ± 0.024 (s)	22.729 ± 0.012 (s)	14.197 ± 0.010 (s)	8	4	10.864
$256 \times 256 \times 128$	32 blocks	157.498 ± 0.022 (s)	24.587 ± 0.012 (s)	15.953 ± 0.009 (s)	17	6	9.873

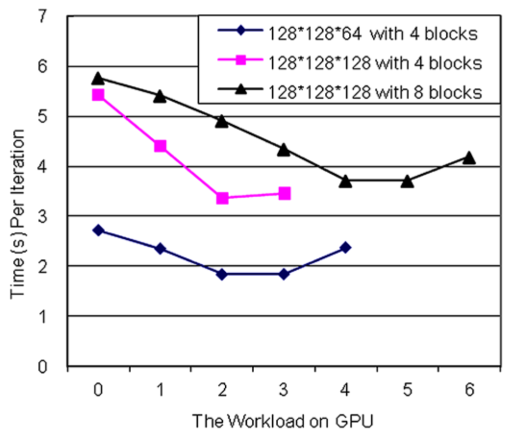


Fig. 16 Performance comparison with varying workload on GPU

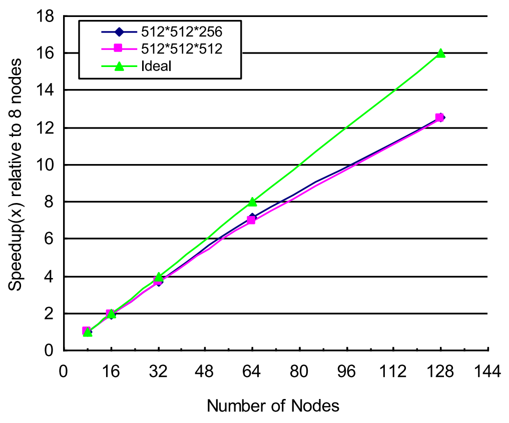


Fig. 17 Strong scalability presentation. The size of the computational meshes are $512 \times 512 \times 256$ and $512 \times 512 \times 512$. The speedup is relative to 8 nodes

workload between GPU and CPU, and utilize the CPU and GPU computational capability. We design an out-of-core method to increase the scale of simulation on a single node. And we propose two strategies to overlap the computation with communication of CPU-GPU and CPU-CPU.

Besides, we have evaluated the performance of our parallel implementation of the solver. The results demonstrate that high-order, structured Navier-Stokes solvers can achieve significant performance improvements from the use of GPUs. We observe about 8× speed up of one GPU implementation over the one CPU core counterpart. With the work distribution, we observe 1.48×−1.61× speed up of CUDA+OpenMP implementation over OpenMP counterpart. By using out-of-core method, we can process the mesh size as big as $256 \times 256 \times 128$ on a single node. The strategies to overlap the computation and communication allow the multi nodes implementation to scale well. The parallel efficiency gets 51.2–72.5 % during weak scaling analysis and the observed speedups on 128 nodes are of 12.53×

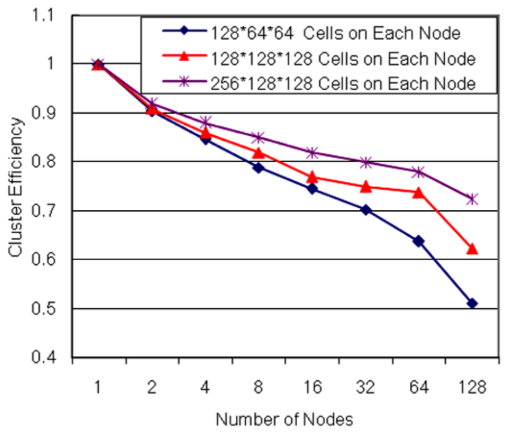


Fig. 18 Weak scalability presentation

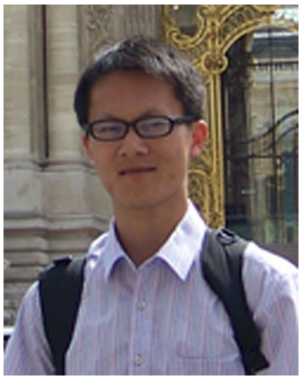
compared to 8 nodes implementation during strong scaling analysis.

In the future, we plan to exploit the usage of shared memory of GPU in solving high-order accurate schemes, and optimize it to improve the performance of the implementation on GPU. We are also planning to design a dynamic load balancing strategy to achieve a better application performance on heterogeneous parallel architecture.

References

1. Harten, A.: High resolution schemes for hyperbolic conservation laws. J. Comput. Phys. **49**(3), 357–393 (1983)
2. van Leer, B.: Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. J. Comput. Phys. **32**(1), 101–136 (1979)
3. Harten, A., Engquist, B., Osher, S., Chakravarthy, S.R.: Uniformly high order accurate essentially non-oscillatory schemes, III. J. Comput. Phys. **71**(2), 231–303 (1987)
4. Liu, X.-D., Osher, S., Chan, T.: Weighted essentially nonoscillatory schemes. J. Comput. Phys. **115**(1), 200–212 (1994)
5. Lele, S.K.: Compact finite difference schemes with spectral-like resolution. J. Comput. Phys. **103**(1), 16–42 (1992)
6. Deng, X., Zhang, H.: Developing high-order weighted compact nonlinear schemes. J. Comput. Phys. **165**(1), 22–44 (2000)
7. Deng, X., Liu, X., Mao, M., Zhang, H.: Investigation on weighted compact fifth-order nonlinear scheme and applications to complex flow. In: 17th AIAA Computational Fluid Dynamics Conference, June 2005. American Institute of Aeronautics and Astronautics (2005)
8. Ishiko, K., Ohnishi, N., Ueno, K., Sawada, K.: Implicit large eddy simulation of two-dimensional homogeneous turbulence using weighted compact nonlinear scheme. J. Fluids Eng. **131**(6), 061401 (2009)
9. Tani, H., Teramoto, S., Yamanishi, N., Okamoto, K.: A numerical study on a temporal mixing layer under transcritical conditions. Comput. Fluids (2012). doi:10.1016/j.compfluid.2012.10.022
10. Yang, X.-J., Liao, X.-K., Lu, K., Hu, Q.-F., Song, J.-Q., Su, J.-S.: The Tianhe-1a supercomputer: its hardware and software. J. Comput. Sci. Technol. **26**, 344–351 (2011)
11. Le, H., Cambier, J.L.: Development of a flow solver with complex kinetics on the graphic processing units. ArXiv e-prints (2011)
12. Tutkun, B., Edis, F.O.: A gpu application for high-order compact finite difference scheme. Comput. Fluids **55**(0), 29–35 (2012)
13. Esfahanian, V., Darian, H.M., Gohari, S.I.: Assessment of Weno schemes for numerical simulation of some hyperbolic equations using GPU. Comput. Fluids (2012). doi:10.1016/j.compfluid.2012.02.031
14. Geveler, M., Ribbrock, D., Goddeke, D., Zajac, P., Turek, S.: Towards a complete fem-based simulation toolkit on GPUs: unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses. Comput. Fluids (2012). doi:10.1016/j.compfluid.2012.01.025
15. Jacobsen, D., Thibault, J., Senocak, I.: An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In: 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, January 2010. American Institute of Aeronautics and Astronautics (2010)
16. Jacobsen, D.A., Senocak, I.: Multi-level parallelism for incompressible flow computations on GPU clusters. Parallel Comput. **39**(1), 1–20 (2013)
17. Han, L., Indinger, T., Hu, X., Adams, N.: Wavelet-based adaptive multi-resolution solver on heterogeneous parallel architecture for computational fluid dynamics. Comput. Sci. Res. Dev. **26**, 197–203 (2011)
18. Antoniou, A., Karantasis, K., Polychronopoulos, E., Ekaterinaris, J.: Acceleration of a finite-difference WENO scheme for large-scale simulations on many-core architectures. In: 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, January 2010. American Institute of Aeronautics and Astronautics (2010)
19. Appleyard, J., Drikakis, D.: Higher-order CFD and interface tracking methods on highly-parallel MPI and GPU systems. Comput. Fluids **46**(1), 101–105 (2011)
20. Wang, P., Abel, T., Kaehler, R.: Adaptive mesh fluid simulations on GPU. New Astron. **15**(7), 581–589 (2010)
21. Griebel, M., Zaspel, P.: A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier–Stokes equations. Comput. Sci. Res. Dev. **25**, 65–73 (2010)
22. Lu, F., Song, J., Cao, X., Zhu, X.: CPU/GPU computing for long-wave radiation physics on large GPU clusters. Comput. Fluids **41**(0), 47–55 (2012)
23. Xie, M., Lu, Y., Liu, L., Cao, H., Yang, X.: Implementation and evaluation of network interface and message passing services for Tianhe-1a supercomputer. In: 2011 IEEE 19th Annual Symposium on High Performance Interconnects (HOTI), pp. 78–86 (2011)

Wei Cao was born in China in 1983. He received B.S. degree at University of Science and Technology of China in 2006 and M.S. degree in computer science at the National University of Defense Technology in China in 2009. Since 2009, he has been a Ph.D. student in computer science at the National University of Defense Technology. His current research interests include CFD numerical algorithm and GPGPU computing.



Chuan-fu Xu was born in China in 1980 and obtained his Ph.D. in computer science and technology in 2011 from National University of Defense Technology, China. Currently he is an assistant professor of computer college, NUDT. His main research interests involve the parallel algorithms and implementations on large-scale HPC systems for applications in computational sciences and engineering.

Lu Yao was born in China in 1983 and received B.S. degree, M.S. degree and Ph.D. degree in computer science at the National University of Defense Technology in China in 2006, 2008 and 2013 respectively. His current research interests include parallel algorithms and scientific computations.

Zheng-hua Wang was born in China in 1962 and received his B.S. degree, M.S. degree and Ph.D. degree at the National University of Defense Technology in China in 1983, 1986, and 1991 respectively. He has been a professor of computer science at the National University of Defense Technology since 2000. His research interests are in computer systems performance evaluation and compiler optimization.

Hua-yong Liu received Ph.D. degree in aerodynamics at the National University of Defense Technology in China in 1999. His current research interests include high-order numerical methods and complex flow mechanism.