Journal of Computational Physics 278 (2014) 275–297

Contents lists available at ScienceDirect

Journal of Computational Physics

www.elsevier.com/locate/jcp

Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer

Chuanfu Xu [b,*], Xiaogang Deng [b], Lilun Zhang [b], Jianbin Fang [c], Guangxue Wang [a], Yi Jiang [a], Wei Cao [b], Yonggang Che [b], Yongxian Wang [b], Zhenghua Wang [b], Wei Liu [b], Xinghua Cheng [b]

[a] State Key Laboratory of Aerodynamics, P.O. Box 211, Mianyang 621000, PR China
[b] College of Computer Science, National University of Defense Technology, Changsha 410073, PR China
[c] Parallel and Distributed Systems Group, Delft University of Technology, Delft 2628CD, The Netherlands

ARTICLE INFO

Article history:
Received 23 September 2013
Received in revised form 13 August 2014
Accepted 17 August 2014
Available online 22 August 2014

Keywords:
GPU parallelization
CFD
CPU–GPU collaboration
High-order finite difference scheme
Multi-block structured grid

ABSTRACT

Programming and optimizing complex, real-world CFD codes on current many-core accelerated HPC systems is very challenging, especially when collaborating CPUs and accelerators to fully tap the potential of heterogeneous systems. In this paper, with a tri-level hybrid and heterogeneous programming model using MPI + OpenMP + CUDA, we port and optimize our high-order multi-block structured CFD software HOSTA on the GPU-accelerated TianHe-1A supercomputer. HOSTA adopts two self-developed high-order compact definite difference schemes WCNS and HDCS that can simulate flows with complex geometries. We present a dual-level parallelization scheme for efficient multi-block computation on GPUs and perform particular kernel optimizations for high-order CFD schemes. The GPU-only approach achieves a speedup of about 1.3 when comparing one Tesla M2050 GPU with two Xeon X5670 CPUs. To achieve a greater speedup, we *collaborate* CPU and GPU for HOSTA instead of using a naive *GPU-only* approach. We present a novel scheme to balance the loads between the *store-poor* GPU and the *store-rich* CPU. Taking CPU and GPU load balance into account, we improve the maximum simulation problem size per TianHe-1A node for HOSTA by 2.3×, meanwhile the collaborative approach can improve the performance by around 45% compared to the GPU-only approach. Further, to scale HOSTA on TianHe-1A, we propose a gather/scatter optimization to minimize PCI-e data transfer times for ghost and singularity data of 3D grid blocks, and overlap the collaborative computation and communication as far as possible using some advanced CUDA and MPI features. Scalability tests show that HOSTA can achieve a parallel efficiency of above 60% on 1024 TianHe-1A nodes. With our method, we have successfully simulated an EET high-lift airfoil configuration containing 800M cells and China's large civil airplane configuration containing 150M cells. To our best knowledge, those are the largest-scale CPU–GPU collaborative simulations that solve realistic CFD problems with both complex configurations and high-order schemes.

* Corresponding author.
E-mail address: xuchuanfu@nudt.edu.cn (C. Xu).

http://dx.doi.org/10.1016/j.jcp.2014.08.024
0021-9991/

1. Introduction

The solution of the Navier–Stokes equations for CFD (Computational Fluid Dynamics) involves reasonably complex numerical methods and algorithms which are used in computational science today. In the past several decades, low-order (order of accuracy ≤ 2) CFD schemes have been widely used in engineering applications, but they are insufficient in simulation resolution and fidelity for complex flows (e.g., turbulence and many viscosity dominant flows such as boundary layer flows, vortical flows, shock–boundary layer interactions, heat flux transfers, etc.) containing sharp gradients and small disturbances. An effective approach to overcome the obstacle of accurate numerical simulations is to employ high-order methods [47]. Over the past 20 to 30 years, there have been many studies in developing and applying various kinds of high-order and high accurate schemes in CFD. Among them, *compact high-order finite difference schemes* are one of the most promising schemes [36]. Based on a given compact stencil, compact schemes can achieve relative higher order of accuracy with good spectral resolution and flexibility over traditional explicit finite difference schemes. Recently, to capture discontinuities in flows with shock wave, nonlinear formulation is added and various non-linear compact schemes such as Weighted Essentially Non-Oscillatory (WENO) [33] and Weighted Compact Non-linear Scheme (WCNS) [23] were developed. They have been extensively studied and widely used for incompressible, compressible and hypersonic flows, and are also very attractive for multi-scale flows (e.g., computational aeroacoustics, computational electromagnetics and turbulence).

Although many researchers claim that they have successfully applied high-order schemes in complex geometries, the grids in their tests are fairly simple compared to those in low-order scheme applications. Problems such as robustness and grid-quality sensitivity [47,24] still hinder the wide application of these schemes on complex meshes. Recently, we have shown that these problems can be largely mitigated by treating the Geometric Conservation Law (GCL) [43] and block-interface conditions carefully when employing high-order schemes on complex configurations. To this end, we have developed a Symmetrical Conservative Metric Method (SCMM) [22] to fulfill GCL by computing grid derivatives in a conservative form, with the same scheme used for fluxes. Further, we have employed a Characteristic-Based Interface Condition (CBIC) [19] to fulfill high-order multi-block computing by directly exchanging the spatial derivatives (i.e., RHS computed on each block) on each side of an interface. Based on SCMM and CBIC, we have successfully simulated a wide range of flow problems with complex grids using WCNS [23] and recently developed a new Hybrid cell-edge and cell-node Dissipative Compact Scheme (HDCS) [18]. The WCNS and HDCS high-order schemes, along with SCMM and CBIC, have been implemented in our in-house high-order CFD software HOSTA (High-Order SimulaTor for Aerodynamics) for 3D multi-block structured grids.

Some recent applications of HOSTA can be found in [25,19,21,20]. Meanwhile, running HOSTA with a fairly large grid on a local machine often takes several weeks or even months. Thus, it is essential and practical to port it onto modern supercomputers, often featuring many-core accelerators/co-processors (GPUs [26], MIC [29], or specialized ones [4]). These heterogeneous processors can dramatically enhance the overall performance of HPC systems with remarkably low total cost of ownership and power consumption, but the development and optimization of large-scale applications are also becoming exceptionally difficult. Researchers often need to use various programming models/tools (e.g., CUDA [1] for NVIDIA's GPUs, OpenMP and MPI for intra-/inter-node parallelization), and map a hierarchy of parallelism in problem domains to heterogeneous devices with different processing capabilities, memory availability, and communication latencies. Hence, for simplicity developers often choose a naive GPU-only approach in a GPU-accelerated systems: letting GPUs compute efficiently and CPUs only manage GPU computation and communication. This is obviously a vast waste of CPU capacity, especially for top supercomputers like TianHe-1A [49] equipped with powerful multi-core CPUs. Our previous GPU-only implementation [48] shows that on one TianHe-1A node (containing two Intel Xeon X5670 CPUs and one NVIDIA Tesla M2050 GPU) the achievable performance of HOSTA on X5670s is around 80% of that on a M2050. In other words, we can potentially obtain an 80% performance improvement by using both the CPUs and the GPU. Therefore, to tap the full potential of heterogeneous systems and maximize application performance, it is immensely important to collaborate CPU and GPU for a *CPU–GPU collaborative simulation* instead of a GPU-only simulation.

Nevertheless, making an efficient and large-scale collaboration for real-world complex CFD applications on heterogeneous supercomputers has been described as a problem as hard as any that computer science has faced. Besides the aforementioned programming challenge, we need to balance the use of unbalanced hardware resources: GPUs are relatively rich in compute capacity but poor in memory capacity and the opposite holds for CPUs; this is generally true for most current heterogeneous supercomputers. On each TianHe-1A node the M2050 contributes about 515 GFlops with less than 3 GB device memory, while the X5670s, with 32 GB shared memory, contribute only 140 GFlops. Our previous results [48] show that HOSTA achieves a speedup of about 1.3 when comparing one M2050 with two hexa-core X5670s, but it can hold a maximum of 2M grid cells on the M2050. Thus, taking CPU–GPU load balance into account, the maximum amount of cells on one TianHe-1A node is around 3.5M, with 2M cells on the GPU and 1.5M ($\approx \frac{2M}{1.3}$) cells on the CPUs. Therefore, HOSTA's simulation capacity on a single TianHe-1A node is bounded by the relatively small device memory on GPUs. For large-scale grids, we can either use more compute nodes to keep the amount of cells per node below 3.5M, or perform an inefficient and unbalanced collaboration. Both cases severely limit the efficiency of heterogeneous systems. Hence, achieving load balance is one of the most serious challenges we face when collaborating *store-poor* GPU with *store-rich* CPU for HOSTA on TianHe-1A. Another practical challenge is that HOSTA performs more extensive data exchanges (specifically four exchanges in each iteration, see 2.2) among neighboring grid blocks than traditional low order scheme applications to ensure the ro-

bustness of high-order simulations. Thus, we also need to design an effective mechanism for data transfers among different parallel levels of TianHe-1A when scaling HOSTA.

In this work, with MPI + OpenMP + CUDA, we collaborate CPU and GPU on TianHe-1A for massively parallel high-order CFD simulations. The aforementioned challenges are tackled by implementing a novel balancing scheme for intra-node CPU–GPU collaboration and overlapping CPU–GPU computation with communication. Specifically, our key contributions include:

- We present a dual-level parallelization scheme for efficient multi-block CFD computation on GPUs. We use multiple CUDA streams (*multi-stream*) to exploit the coarse-grained parallelism among grid blocks and CUDA thread blocks to exploit the fine-grained parallelism among cells within a data block. Further, with a kernel decomposition strategy, we dramatically optimize the time-consuming CUDA kernels calculating high-order flow fluxes. The GPU-only approach achieves a speedup of about 1.3 when comparing one Tesla M2050 with two hexa-core Xeon X5670s.
- On a single TianHe-1A node we collaborate the CPUs and the GPU using nested OpenMP and CUDA. The key idea is to surmount the GPU memory limit by a dynamic device memory use policy for selected flow variables, coupled with grouping and pipelining GPU-computed grid blocks using CUDA streams. The balanced collaborative approach can top up HOSTA's maximum simulation problem size on a single TianHe-1A node from 3.5M cells to 8M cells (2.3×), and meanwhile has up to 45% performance advantage over the GPU-only approach.
- To scale HOSTA on large-scale compute nodes, we propose a gather/scatter optimization to minimize PCI-e data transfer times for ghost and singularity data of 3D grid blocks which are discontinuously stored in the device memory. Further, we use non-blocking MPI, CUDA events and CUDA streams to perform inter-block data exchanges as early as possible and overlap multiple levels of computation and communication. Despite of the extensive data exchanges required in HOSTA, we achieve a parallel efficiency of about 60% on 1024 nodes.
- As a case study, we have successfully collaborated hundreds of TianHe-1A nodes to simulate the high-lift airfoil configuration *30p30n* containing about 800M cells and China's large civil airplane configuration *C919* containing about 150M cells. To our best knowledge, those are the **largest-scale CPU–GPU collaborative simulations** that solve **realistic CFD problems** with both **complex configurations** and **high-order schemes**.

The remainder of the paper is organized as follows. Section 2 briefly describes numerical methods and implementation in HOSTA. Section 3 presents the overall consideration for MPI + OpenMP + CUDA parallelization. Section 4 introduces the GPU implementation and optimization. In Section 5, we detail the intra-node CPU–GPU collaboration and the balancing scheme. Section 6 scales HOSTA to a large number of TianHe-1A nodes. Section 7 presents performance results and two test cases are shown in Section 8. Finally in Section 9, we introduce some related work and conclude in Section 10.

2. Numerical methods and HOSTA implementation

In this section, we briefly introduce some numerical methods used in this work and then give the implementation of HOSTA.

2.1. Numerical methods

In Cartesian coordinates the governing equations (Euler or Navier–Stokes) in strong conservative form are

$$\frac{\partial Q}{\partial t}+\frac{\partial F}{\partial x}+\frac{\partial G}{\partial y}+\frac{\partial H}{\partial z}=0 \tag{1}$$

the equations are transformed into curvilinear coordinates by introducing the transformation $(x, y, z, t) \to (\xi, \eta, \zeta, \tau)$

$$\frac{\partial \tilde{Q}}{\partial \tau}+\frac{\partial \tilde{F}}{\partial \xi}+\frac{\partial \tilde{G}}{\partial \eta}+\frac{\partial \tilde{H}}{\partial \zeta}=0 \tag{2}$$

where

$$\tilde{F}=\tilde{\xi}_t Q+\tilde{\xi}_x F+\tilde{\xi}_y G+\tilde{\xi}_z H, \quad \tilde{\xi}_x=J^{-1}\xi_x \tag{3}$$

and with similar relations for the other terms.

The fifth-order WCNS scheme WCNS-E-5 and the seventh-order HDCS scheme HDCS-E8T7 were used in this work. Here we only consider the discretization of the inviscid flux derivative along the ξ direction. The discretization along the other directions can be dealt in a similar manner. The interior scheme of WCNS-E-5 can be expressed as

$$\frac{\partial \tilde{F}_i}{\partial \xi}=\frac{75}{64h}(\tilde{F}_{i+1/2}-\tilde{F}_{i-1/2})-\frac{25}{384h}(\tilde{F}_{i+3/2}-\tilde{F}_{i-3/2})+\frac{3}{640h}(\tilde{F}_{i+5/2}-\tilde{F}_{i-5/2}) \tag{4}$$

and the interior scheme of HDCS-E8T7 can be expressed as

$$\frac{\partial \tilde{F}_i}{\partial \xi}=\frac{256}{175h}(\tilde{F}_{i+1/2}-\tilde{F}_{i-1/2})-\frac{1}{4h}(\tilde{F}_{i+1}-\tilde{F}_{i-1})+\frac{1}{100h}(\tilde{F}_{i+2}-\tilde{F}_{i-2})-\frac{1}{2100h}(\tilde{F}_{i+3}-\tilde{F}_{i-3}) \tag{5}$$

where h is the grid size, and $\tilde{F}_{i+1/2}=\tilde{F}(U^L_{i+1/2},U^R_{i+1/2},\tilde{\xi}_{x,i+1/2},\tilde{\xi}_{y,i+1/2},\tilde{\xi}_{z,i+1/2})$ is the cell-edge flux computed by some flux-splitting methods that can be found in [23]. $U_i=U(\xi_i,t)$ is the flow variables. $U^L_{i+1/2}$ and $U^R_{i+1/2}$ are the left-hand and right-hand cell-edge variables. For WCNS-E-5, they can be obtained by a high-order nonlinear weighted interpolation of cell-node variables. For HDCS-E8T7, we use the following seventh-order dissipative compact interpolation

$$\begin{aligned}&\frac{5}{14}(1-\alpha)U^L_{i-1/2}+U^L_{i+1/2}+\frac{5}{14}(1+\alpha)U^L_{i+3/2}\\&=\frac{25}{32}(U_{i+1}+U_i)+\frac{5}{64}(U_{i+2}+U_{i-1})-\frac{1}{448}(U_{i+3}+U_{i-2})\\&\quad+\alpha\left[\frac{25}{64}(U_{i+1}-U_i)+\frac{15}{128}(U_{i+2}-U_{i-1})-\frac{5}{896}(U_{i+3}-U_{i-2})\right]\end{aligned}\tag{6}$$

where $\alpha<0$ is the dissipative parameter to control numerical dissipation. $U^R_{i+1/2}$ can be obtained by setting $\alpha>0$. The boundary and near-boundary schemes of WCNS-E-5 and HDCS-E8T7 can be found in [23] and [18] respectively. For the viscous fluxes, we use the same sixth order central difference scheme as [23].

The GCL includes the volume conservation law (VCL) and the surface conservation law (SCL). For stationary grids, VCL naturally holds

$$I_t=(1/J)_\tau+(\tilde{\xi}_t)_\xi+(\tilde{\eta}_t)_\eta+(\tilde{\zeta}_t)_\zeta=0\tag{7}$$

if SCL is satisfied, then

$$\begin{aligned}I_x&=(\tilde{\xi}_x)_\xi+(\tilde{\eta}_x)_\eta+(\tilde{\zeta}_x)_\zeta=0\\I_y&=(\tilde{\xi}_y)_\xi+(\tilde{\eta}_y)_\eta+(\tilde{\zeta}_y)_\zeta=0\\I_z&=(\tilde{\xi}_z)_\xi+(\tilde{\eta}_z)_\eta+(\tilde{\zeta}_z)_\zeta=0\end{aligned}\tag{8}$$

High-order schemes with their low dissipative property usually bear more risk from the SCL-related errors than that of low-order schemes. The SCMM ensures the SCL in high-order difference schemes at the following two aspects:

- First, metrics are acquired through the "conservative forms"

$$\begin{aligned}\tilde{\xi}_x&=\frac{1}{2}\left[(zy_\eta)_\zeta+(yz_\zeta)_\eta-(zy_\zeta)_\eta-(yz_\eta)_\zeta\right]\\\tilde{\eta}_x&=\frac{1}{2}\left[(zy_\zeta)_\xi+(yz_\xi)_\zeta-(zy_\xi)_\zeta-(yz_\zeta)_\xi\right]\\\tilde{\zeta}_x&=\frac{1}{2}\left[(zy_\xi)_\eta+(yz_\eta)_\xi-(zy_\eta)_\xi-(yz_\xi)_\eta\right]\end{aligned}\tag{9}$$

  and with similar relations for the remain items.
- Second, on each grid direction, the algorithm of the derivatives in Eq. (9) shall be identical to that of flow fluxes where the metrics are re-discretized in combination with the flow fluxes. For example, the re-discretization scheme of WCNS-E-5 is

$$\frac{\partial a_i}{\partial\xi}=\frac{75}{64h}(a_{i+1/2}-a_{i-1/2})-\frac{25}{384h}(a_{i+3/2}-a_{i-3/2})+\frac{3}{640h}(a_{i+5/2}-a_{i-5/2})\tag{10}$$

  where the cell-edge values can be computed by high-order linear interpolation.

Finally, we briefly introduce the CBIC. Suppose RHS be the right-hand-side of the n-th time step

$$RHS=-J\left[\left(\frac{\partial\tilde{F}}{\partial\xi}+\frac{\partial\tilde{G}}{\partial\eta}+\frac{\partial\tilde{H}}{\partial\zeta}\right)\right]^n\tag{11}$$

define the transformation matrix P_{QV_C} in terms of conservative variables Q and characteristic variables V_C

$$P_{QV_C}=\frac{\partial Q}{\partial V_C}\tag{12}$$

the CBIC are

$$\begin{aligned}\left.\frac{\partial Q}{\partial t}\right|_L&=\left(A_s^+\right)\big|_L(RHS)\big|_L+\left(A_s^-\right)\big|_L(RHS)\big|_R\\\left.\frac{\partial Q}{\partial t}\right|_R&=\left(A_s^+\right)\big|_R(RHS)\big|_R+\left(A_s^-\right)\big|_R(RHS)\big|_L\end{aligned}\tag{13}$$

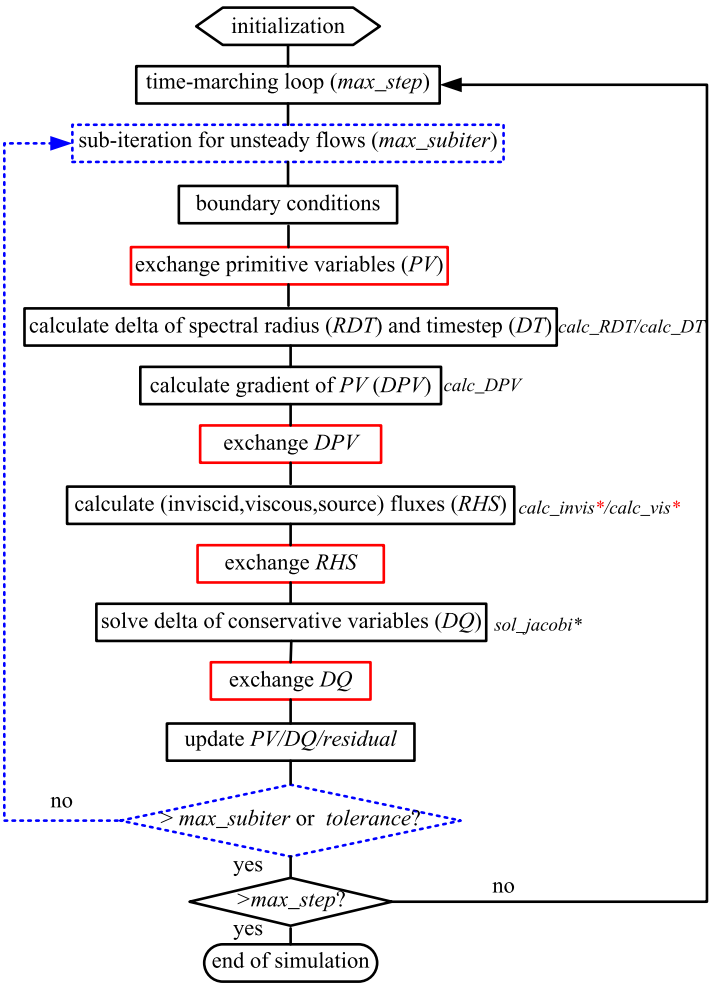


Fig. 1. Flowchart of HOSTA. Some subroutines implementing the steps are also shown. Time consuming subroutines are indicated by asterisk signs. HDCS and WCNS are implemented in `calc_vis` and `calc_invis`, indicated by red asterisk signs. Note that each iteration has four exchanges (denoted by red rectangular) for ghost/singularity data to ensure the robustness of high-order schemes.

where $(A_s^+) = P_{QV_C}\,\mathrm{diag}[(1+sign(\lambda_i))/2]P_{QV_C}^{-1}$, $(A_s^-) = P_{QV_C}\,\mathrm{diag}[(1-sign(\lambda_i))/2]P_{QV_C}^{-1}$. $(A_s^+)|_{L,R}$ are calculated by variables on the interface, and $(RHS)|_{L,R}$ can be calculated by one-side differencing or other methods. Here, the subscripts "L" and "R" denote the variables and terms on an interface but belong to the left-hand-side sub-block and the right-hand-side sub-block, respectively. One can verify that $\frac{\partial Q}{\partial t}|_L = \frac{\partial Q}{\partial t}|_R$. Please refer to [19] for more details about the CBIC.

2.2. HOSTA implementation

HOSTA is implemented in Fortran 90 and we present its main flowchart in Fig. 1. At the beginning of each iteration (a time step or a sub-iteration of unsteady simulations), boundary conditions are applied. Then data exchange for primitive variables of ghost/singularity cells is performed. Singularity cells are those cells that belong to several grid blocks. They are often unavoidable in complex multi-block structured grids. HOSTA calculates the delta of spectral radius (`calc_RDT`) and time-step before it calculates and exchanges the gradient of primitive variables (`calc_DPV`). HDCS and WCNS are implemented when calculating viscous (`calc_vis`) and inviscid (`calc_invis`) fluxes for RHS. The RHS is also exchanged. Implicit methods including LU-SGS, PR-SGS, Jacobi and explicit Runge–Kutta method are implemented for time marching. In this paper, we use the Jacobi iterative method (`sol_jacobi`) on GPUs. After HOSTA exchanges the delta of conservative variables, it updates the primitive variables and residuals and then completes the iteration. Flow variables are declared as global variables and thus can be used in several subroutines conveniently. Those variables are allocated and initialized prior to the main time-marching loop, and are deallocated at the end of the simulation. In the MPI implementation, generally a large multi-block complex 3D grid is repartitioned, along three grid directions, into many grid blocks and those blocks are then grouped and distributed among MPI processes according to a load balance model [50]. Non-blocking MPI is used to overlap message passing and computation.

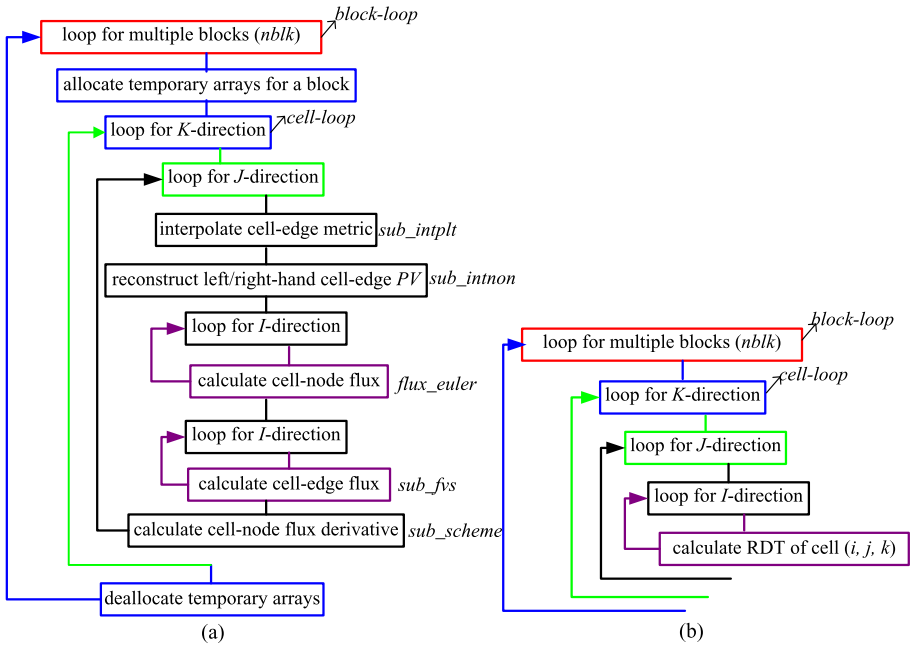


Fig. 2. A closer look of typical loop skeletons in HOSTA. (a) The *I*-direction loop skeleton in `calc_invis`. It clearly shows how high-order schemes such as HDCS and WCNS are implemented for calculating inviscid fluxes. Some steps are encapsulated in Fortran subroutines. (b) The loop skeleton in `calc_RDT`.

Fig. 2 presents a closer look of two typical loop skeletons to further illustrate multi-block calculating procedures in HOSTA. Generally, HOSTA organizes its calculation in a four-level loop: an outer *block-loop* specifying block index and a three-level inner *cell-loop* specifying cell index in the block along the three directions. For complex kernels like `calc_vis` and `calc_invis`, we implement three different four-level loops for the three directions. Fig. 2(a) shows the *I*-direction loop skeleton in `calc_invis`. Firstly, the cell-edge metrics are interpolated and the left-hand and right-hand cell-edge primitive variables are reconstructed. Then, the cell-node fluxes are calculated and the cell-edge fluxes are also calculated using flux-splitting methods such as Roe, Steger-warming and etc. Finally, the cell-node flux derivatives are calculated. The whole calculation procedure is very complex, involving the algorithms and methods described in Section 2.1. HOSTA uses temporary arrays to hold intermediate results in the procedure, and those arrays are allocated and deallocated dynamically for each grid block in each iteration. Data dependence only exists in the inner *I*-loop, i.e., the calculation of cell (i, j, k) depends on its neighboring cells $(i \pm m, j, k)$, where m is the width of the stencil. The complexity of the loop skeletons in `calc_vis` and `calc_invis` requires our extra implementation and optimization efforts when porting HOSTA to GPUs. Fig. 2(b) shows the loop skeleton in `calc_RDT`. It is relatively simple: each cell uses flow variables on itself and there is no data dependence. To summarize, we can see that in HOSTA there exists a two-level parallelism, i.e., a coarse-grained parallelism (corresponding to the block-loop) among multiple grid blocks and a fine-grained parallelism (corresponding to the cell-loop) among cells in a grid block.

3. Overall MPI + OpenMP + CUDA parallelization

In this section, we briefly describe the TianHe-1A supercomputer and the overall MPI + OpenMP + CUDA parallelization of HOSTA on TianHe-1A.

3.1. The TianHe-1A supercomputer

TianHe-1A [49] is a GPU-accelerated supercomputer developed by National University of Defense Technology (NUDT), China, and was ranked No. 1 in the 36th Top500 list [5] for HPC systems in Nov., 2010. Each TianHe-1A node (see Fig. 3) contains two Intel hexa-core Xeon X5670s sharing a 32 GB main memory and one NVIDIA Tesla M2050 with a 3 GB device memory. Each GPU contains 14 Streaming Multiprocessors (SMs) and each SM has 32 CUDA cores. The theoretical double-precision performance of the two X5670s is about 140 GFlops, which is about 27% of the M2050's 515 GFlops. However, the memory bandwidth of the two X5670s is about 64 GB/s, which is about 43% of the M2050's 148 GB/s. Since CFD is memory-bounded, the difference of memory bandwidth rather than that of the computation is expected to be the dominant factor of the achievable performance difference between CPU and GPU. Nodes are connected via a self-developed fat-tree interconnection network, with a latency of about 1.57 μs and a bidirectional bandwidth of about 20 GB/s.

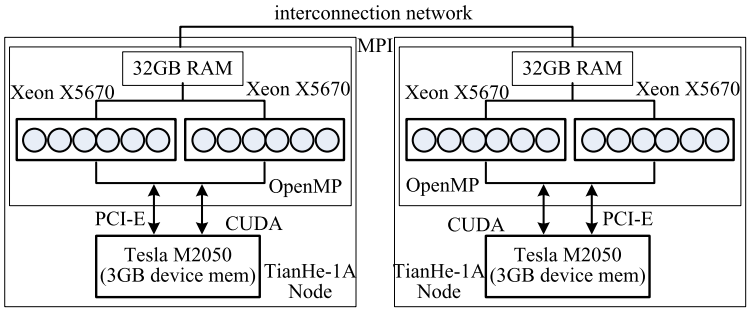


Fig. 3. Compute nodes and heterogeneous programming in TianHe-1A. Each TianHe-1A node has two hexa-core Xeon X5670 CPUs and one Tesla M2050 GPU, contributing totally about 656 GFlops theoretical performance in double-precision.

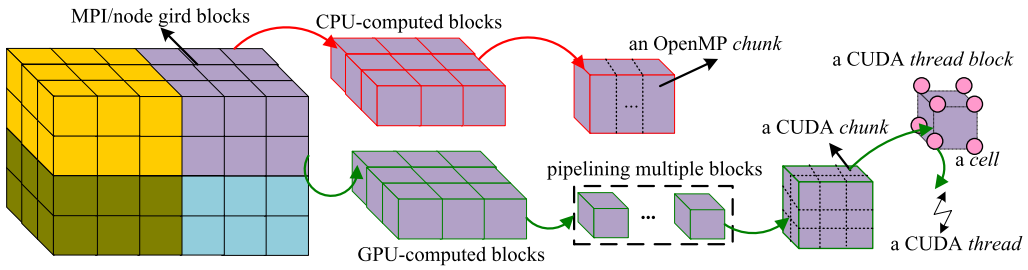


Fig. 4. Overall domain decomposition for MPI + OpenMP + CUDA parallelization. Grid blocks are first distributed among MPI processes/nodes and then on each node are further allocated between CPU and GPU.

With regard to programming models, we use CUDA on GPUs, OpenMP on CPUs. We also use MPI to enable inter-node communication. CUDA functions that run on GPUs are called *kernels*. A kernel is implemented using the SPMD (Single Program Multiple Data) model and executed in parallel by many GPU threads on CUDA cores using the SIMD (Single Instruction Multiple Data) model. GPU threads are grouped into *thread blocks* which are scheduled to a multiprocessor. All thread blocks compose a *thread grid*. Before the kernel is launched, developers configure the number of threads in a thread block and the number of thread blocks in a thread grid. Each CUDA thread can use the built-in *dim3* type variables *threadIdx* and *blockIdx* to specify its index in the local thread block and the global thread grid. Applications could manage concurrency through CUDA *streams*. A CUDA stream is simply a sequence of operations that are performed in order on the device. Operations from different streams can be interleaved on devices that are capable of concurrent copy and execute, and it is also possible to overlap host computation with asynchronous data transfers and device computations.

3.2. Overall MPI + OpenMP + CUDA parallelization

Fig. 4 shows the overall domain decomposition for MPI + OpenMP + CUDA parallelization. The domain decomposition for MPI parallelization is same as described in Section 2.2. We create one MPI process per node (of TianHe-1A) to manage the two CPUs and the GPU. The *nblk* grid blocks on a MPI process/node is further allocated between the intra-node CPUs and GPU (see Section 5). The OpenMP directives are added over the outer-most cell-loop, i.e., each CPU-computed grid block is logically partitioned to many *OpenMP chunks* and each chunk is calculated by an OpenMP thread. GPU-computed grid blocks are issued simultaneously by multiple CUDA streams, and each grid block is logically partitioned into many *CUDA chunks*; each CUDA chunk is updated by a CUDA thread block and finally, each cell in the chunk is calculated by a CUDA thread.

4. GPU-only parallelization and optimization

Before we detail the collaborative approach, in this section we parallelize and optimize HOSTA on a single GPU.

4.1. Dual-level GPU parallelization

As described in Section 2.2, in HOSTA a multi-block loop can be viewed as two-level: block-loop and cell-loop. This two-level parallelism is mapped to two CUDA constructs, i.e., CUDA stream and kernel, in our dual-level GPU parallelization (see Fig. 5). Each cell-loop is implemented as a CUDA kernel dedicated to the calculation in a grid block. The block-loop is replaced with a CUDA *stream-loop* in which a stream ID is used in a CUDA kernel to index a specific grid block.

For different CUDA kernels, we employ different CUDA implementations and configurations according to data dependence in cell-loops. If there is no data dependence between cells (e.g., `sol_jacobi` and `calc_RDT`), we adopt a 3D CUDA kernel configuration. For example, consider a 3D gird block of size (NI, NJ, NK) (i.e., there are NI, NJ and NK cells on I, J and

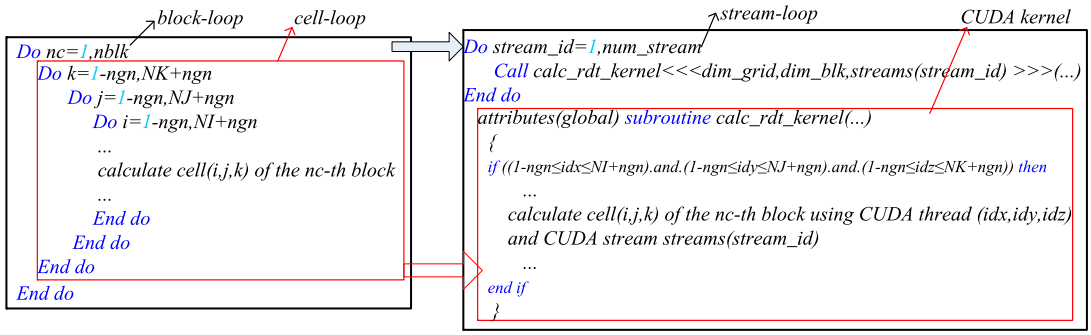


Fig. 5. Dual-level GPU parallelization for multi-block calculation. The block-loop and cell-loop are parallelized with CUDA streams and CUDA kernels respectively. Since HOSTA is implemented in Fortran, for simplicity and clarification in this paper we use Fortran and CUDA Fortran [2] features to describe code skeletons or pseudocodes.

K directions respectively) with ngn ghost cells on each direction, we use (x_blk, y_blk, z_blk) to specify a 3D CUDA thread block and define two CUDA built-in type `dim3` variables `dim_blk` and `dim_grid` for CUDA thread block and thread grid configuration respectively

```
dim_blk=dim3(x_blk,y_blk,z_blk)
```
$$\texttt{dim\_grid=dim3}(\lceil NI + 2 \times ngn/x\_blk \rceil, \lceil NJ + 2 \times ngn/y\_blk \rceil, \lceil NK + 2 \times ngn/z\_blk \rceil)$$

where $\lceil x \rceil$ is the minimum integer no smaller than x. For a 3D kernel configuration, 3D grid cells on I, J and K directions are mapped to GPU threads on X, Y and Z dimensions correspondingly. The 3D CUDA kernel is implemented as an elemental function. Each GPU thread executes its own copy of the kernel code, uses its thread ID (idx, idy, idz) to index a cell and calculates the cell independently, where

$$idx = threadIdx.x + blockDim.x \times (blockIdx.x) + 1 - ngn$$
$$idy = threadIdx.y + blockDim.y \times (blockIdx.y) + 1 - ngn$$
$$idz = threadIdx.z + blockDim.z \times (blockIdx.z) + 1 - ngn$$

However, as illustrated in Fig. 2, in HOSTA there also exist more complex cell-loops (e.g., `calc_invis` and `calc_vis`) with data dependence and subroutine calling. As explained in Section 2.2, HOSTA decomposes those complex high-order flux calculation kernels along individual directions, thus data dependence and subroutine calling only exists in one direction, i.e., the inner-most loop. Since CUDA has no global synchronization, our initial GPU implementation uses a 2D kernel configuration to satisfy the data dependence and takes subroutines in the cell-loop as CUDA device kernels. Take the I direction cell-loop of `calc_invis` as an example, the 2D CUDA kernel configuration is

```
dim_blk=dim3(1,y_blk,z_blk)
```
$$\texttt{dim\_grid=dim3}(1, \lceil NJ + 2 \times ngn/y\_blk \rceil, \lceil NK + 2 \times ngn/z\_blk \rceil)$$

Thus, each CUDA thread calculates all cells on the I direction.

Since there is no data dependence in a block-loop (i.e., among grid blocks), it can be parallelized with CUDA streams. Fig. 5 shows a simple multi-stream implementation. We create the same number of CUDA streams as that of grid blocks before the time loop and destroy them at the end of the simulation. Thus, the block-loop is converted into the stream-loop. In the stream-loop, we bind each grid block with a CUDA stream and issue all the streams to the GPU. All the GPU operations of the grid block including kernel execution, host to device (H2D) and device to host (D2H) data copy are performed with the associated CUDA stream. Fig. 7(a) shows the timeline for multi-stream when streaming 4 grid blocks. The multi-stream scheme can eliminate the serial execution semantics between grid blocks in the block-loop, which is implicitly added by programming languages such as Fortran and C, and better exploit the parallel potential of modern GPU architecture. For example, when a stream is accessing the global memory, the kernel engine can schedule and execute GPU warps from other streams to hide memory access latency. More importantly, the kernel execution and PCI-e data transfers of different streams/blocks can be substantially overlapped, especially for GPUs with separate H2D and D2H copy engines such as the Tesla M2050 in TianHe-1A. Furthermore, as we will describe in the following sections, the multi-stream scheme can be extended for better load balance in a collaborative simulation and overlapping CPU/GPU computation and communication.

4.2. Kernel decomposition

Generally, a 3D kernel configuration can ensure more GPU threads to run simultaneously and may allow threads to access the global memory in a coalesced manner, and thus improves performance. In HOSTA, high-order flux calculation kernels

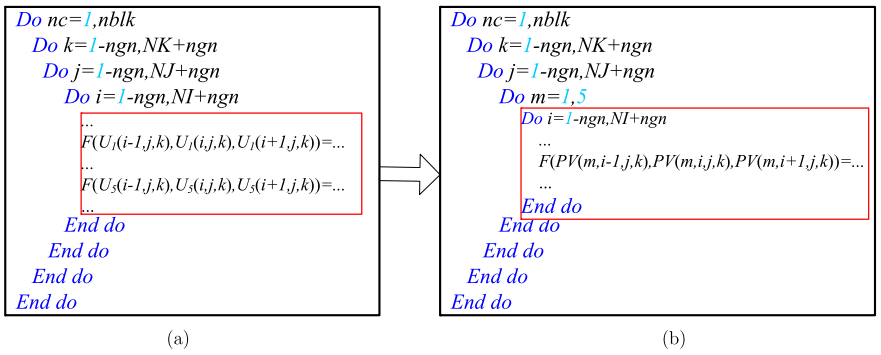


Fig. 6. Code transformation of data structures and inner-loops in HDCS-E8T7.

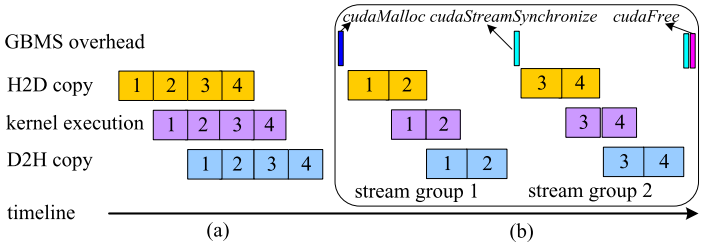


Fig. 7. Different schemes when pipelining 4 grid blocks: (a) multi-stream uses 4 streams; (b) GBMS uses 2 stream groups and each group contains 2 streams. It calls `cudaStreamSynchronize` to synchronize streams in each group and also calls `cudaMalloc`/`cudaFree` (and may also perform H2D/D2H copy) for selected flow variables.

(i.e., kernels in `calc_invis` and `calc_vis`) are the most time-consuming ones. But they are initially implemented with a 2D configuration, due to data dependence. In this paper, we optimize those kernels with a functional decomposition strategy. By carefully analyzing the data dependence and inlining device kernels, a large complex 2D kernel can be broken into several small 3D kernels. For example, the initial 2D kernel `calc_I_invis` calculating the I direction inviscid fluxes based on WCNS-E-5 is decomposed to four 3D kernels: `calc_I_invis_1` calculating the cell-edge metrics, `calc_I_invis_2` reconstructing the left-hand and right-hand cell-edge primitives, `calc_I_invis_3` calculating the cell-edge and cell-node inviscid fluxes and `calc_I_invis_4` calculating the cell-node derivative of inviscid fluxes.

For HDCS-E8T7, we use a *semi-3D* kernel configuration for `calc_I_invis_2` because its interpolation scheme requires solving a system of tri-diagonal equations (see Eq. (6)). The semi-3D kernel is implemented based on a code transformation of data structures and inner-loops. Fig. 6(a) shows the initial code snippet which has five separated primitive variables U_m $(1 \le m \le 5)$. Fig. 6(b) shows the new code snippet with a merged primitive variable PV replacing the five and forming a new inner loop for m. Then we exchange loop m and loop i, and an independent "M" direction is available to be parallelized on the GPU. Thus, a semi-3D kernel configuration of a 3D thread block (5,*y_blk*,*z_blk*) and a 2D GPU grid $(1, \lceil NJ + 2 \times ngn/y\_blk \rceil, \lceil NK + 2 \times ngn/z\_blk \rceil)$ can be used in `calc_I_invis_2`.

5. Collaborating intra-node CPU and GPU

5.1. Intra-node collaborative programming

Since most OpenMP compilers support nested parallelism, we use this feature to coordinate the OpenMP parallelized CPU code and the CUDA parallelized GPU code. Fig. 8 illustrates the collaborative programming as well as the balancing scheme (see Section 5.2) for calculating the inviscid fluxes (`calc_invis`). The other calculating procedures can be implemented in a similar manner. To implement a collaborative calculation, we develop two different versions of `calc_invis`: the GPU version `calc_invis_gpu` wrapping CUDA kernels and the CPU version `calc_invis_cpu` parallelized with OpenMP. We create two OpenMP threads at the first-level parallel region. One thread calls `calc_invis_gpu` to launch CUDA kernels, dealing with the GPU-computed grid blocks. The other thread calls `calc_invis_cpu`, dealing with the CPU-computed grid blocks. In `calc_invis_cpu`, for each block we fork a second-level nested OpenMP threads (11 threads) on the multi-core CPUs. We call `cudaDeviceSynchronize` to perform a CPU–GPU synchronization before exchanging ghost/singularity data. During collaboration, heterogeneous devices are concurrently working on their own grid blocks.

We employ a static task/load partition for the intra-node CPUs and GPU: on each side all kernels always calculate the same data objects (specifically grid blocks in HOSTA) until the end of a simulation. This is based on an observation: in HOSTA (and also many other heterogeneous applications) the GPU code employs a static but efficient memory use policy similar to the CPU code. All flow variables of GPU-computed grid blocks are allocated and initialized before the time-marching loop,

```
streams(:) : CUDA stream arrays
gpu_blk=ceiling(gpu_ratio*nblk)
 ...
!$omp parallel num_threads(2)
 If (omp_get_thread_num().eq.0) then
    If  GBMS is defined  then    !GBMS implementation
        allocate device mem of num_grpstream blocks for mb_var
        Do stream_grp_id=1,num_grp
            Do  stream_id=1,num_grpstream
               nc=(stream_grp_id-1)*num_grpstream+stream_id
               If (nc<gpu_blk) then
                  H2D copy the nc-th mb_var using streams(stream_id)
                  Call calc_invis_gpu(…) using streams(stream_id)
                  D2H copy the nc-th mb_var using streams(stream_id)
               End If
            End Do
            synchronize streams
        End Do
        deallocate mb_var
    Else   !multi-stream implementation
        Do  stream_id=1,gpu_blk
            Call calc_invis_gpu(…) using streams(stream_id)
         End Do
    End If
 End If
 If (omp_get_thread_num().eq.1) then
       Call calc_invis_cpu(…)
 End If
!$omp end  parallel
 ...
```

Fig. 8. Pseudocode illustrating collaborative programming and GBMS in `calc_invis`. `mb_var` is a selected flow variable. The host and device kernels are encapsulated in `calc_invis_cpu` and `calc_invis_gpu` respectively.

and deallocated until the end of the loop (see Section 2.2). This can avoid unnecessary PCI-e data transfers during iterations, and we only need to transfer four flow variables (i.e., `RHS`, `DQ`, `PV` and `DPV`) when their ghost/singularity data exchanges are needed. The partition is implemented at the granularity of grid blocks. We use a parameter *gpu_ratio* to adjust the number of grid blocks simulated by each side

$$gpu\_blk = ceiling(gpu\_ratio \times nblk) \tag{14}$$

The GPU calculates $[1, gpu\_blk]$ blocks and the others are calculated on the CPUs. *gpu_ratio* is configured by profiling HOSTA's sustainable performance of one iteration on different devices. We set $gpu\_ratio = 0$ to perform a CPU-only simulation and then set $gpu\_ratio = 1.0$ to perform a GPU-only simulation. Taking the CPU performance as the baseline, *gpu_ratio* is evaluated as $\frac{SP_{GPU}}{(1.0+SP_{GPU})}$ (suppose the achieved speedup of GPU to CPU is SP_{GPU}).

Note that it is possible to implement a more fine-grained or dynamic partition like the approaches presented in [35,7]. For example, in HOSTA we can precisely partition specific number of cells to different devices, or even assign different number of cells for different kernels according to their execution characteristics on different devices. Those approaches generally require developers to reconstruct data structures, and meanwhile carefully manage data dependence among kernels and data transfers among devices. However, HOSTA has so many data objects/kernels implemented by different programming models on different devices that the changes will definitely make programming more difficult and may result in uncertain overall performance benefits. Therefore, the approaches in [35,7] are often tested and applied in heterogeneous code implemented using a portable and unified language such as OpenCL [3] on different devices. Our partition approach in Eq. (14) is fairly simple but generic for multi-block CFD applications. Furthermore, we can adjust the partition granularity (e.g., grid block size) by independent mesh repartition tools [46].

5.2. Balancing intra-node CPU and GPU

Our results show that HOSTA runs about 0.3 times faster on the M2050 than on the two X5670s ($gpu\_ratio = \frac{1.3}{2.3} \approx 0.57$), and thus we tend to distribute more grid blocks on the GPU. Suppose we have a large grid and each node is expected to simulate 6M cells, ideally we let the GPU simulate $6M \times 0.57 \approx 3.4M$ cells and the CPUs simulate 2.6M ($= 6M - 3.4M$) cells according to their achievable performance. However, the M2050 can only hold 2M cells and the other 4M cells have to be distributed to the X5670s. Consequently, without a load-balancing scheme, for this case there is a significant imbalance of resource utilization: the GPU stays idle for most of the time while the CPUs are busy. Therefore, HOSTA's performance on a single TianHe-1A node is bounded by the relatively small device memory on GPUs.

To mitigate the limit, we need to reduce the maximum device memory requirement of GPU-computed grid blocks. For this end, we extend the multi-stream scheme [48] in two steps. As a first step, we group multiple CUDA streams in multi-stream to form a Group-Based Multiple Streams (GBMS) scheme. In multi-stream, we use *gpu_blk* streams to issue

the same number of grid blocks (see Fig. 5 and Fig. 7(a)). In GBMS, we divide *gpu_blk* streams into *num_grp* stream groups and each group contains *num_grpstream* streams (see Fig. 5 and Fig. 7(b)), i.e., $gpu\_blk = num\_grp \times num\_grpstream$. As a second step, we add a dynamic device memory use policy in GBMS for particularly *selected flow variables* (e.g., `mb_var` in Fig. 5). For the GPU-computed grid blocks, `mb_var` is also stored in the host main memory. The device memory of `mb_var` is allocated/deallocated dynamically before/after a CUDA kernel referencing it. Before executing the kernel, we copy it from the host memory to the device memory. After executing the kernel, we copy it back to the host memory if it is changed in the kernel. Fig. 5 shows the pseudocode illustrating how GBMS is implemented in a collaborative simulation.

Compared with the multi-stream scheme, using GBMS can significantly reduce the requirement of device memory. Suppose for each block, `mb_var` requires D device memory, thus we need $gpu\_blk \times D$ device memory in multi-stream since those blocks are simultaneously executed on the GPU. With GBMS, the device memory requirement of `mb_var` is reduced to $num\_grpstream \times D$. Furthermore, in multi-stream the device memory of `mb_var` is persistent until the end of a simulation, while in GBMS the device memory of `mb_var` is only needed during the execution of a kernel and thus is temporary. In other words, the device memory of `mb_var` is deallocated immediately after the kernel finishes its execution; if the to-be-executed kernel do not use `mb_var`, we do not need to keep its device memory. We can use different $(num\_grp, num\_grpstream)$ configurations to further adjust the device memory requirement. For example, when pipelining 8 grid blocks on the GPU, $(num\_grp, num\_grpstream)$ can be set to $(2, 4)$ or $(4, 2)$. Obviously, the $(4, 2)$ configuration needs less device memory. For a large grid, users can adjust the configuration to check if the device memory is enough.

We specify selected flow variables based on the following principles:

- Variables used by a small number of CUDA kernels.
- Variables that have to be transferred for ghost/singularity data exchanges.
- Variables that occupy relatively large memory spaces.
- Variables used in the CUDA kernels that are far faster than their counterpart Fortran subroutines.

Those principles allow us to specify fewer selected flow variables, avoid significantly changing HOSTA code, and possibly minimize GBMS overhead (see next paragraph). The more selected flow variables we specify, the less device memory we require. In our implementation, we carefully specify 9 flow variables as selected flow variables, which accounts for about 36% of the total 25 flow variables and about 50% of the total device memory requirement. As a result, with GBMS we increase the maximum simulation problem size of HOSTA on the M2050 from 2M cells to about 4M cells.

At the same time, GBMS incurs an overhead on the GPU, due to dynamic device memory allocation/deallocation, H2D/D2H copy and stream synchronization between stream groups to ensure the completeness of asynchronous H2D/D2H copy. As Section 7.3 shows, the relative speedup of one M2050 to two X5670s drops from 1.3 to around 1.0 in GBMS. GBMS also needs additional host memory to hold selected flow variables for GPU-computed grid blocks. The overhead is reasonable, mainly attributed to the employment of CUDA streams and carefully specifying selected flow variables, both of which can hide or minimize the GBMS overhead properly. GBMS provides a solid base for more flexible task partition and better load balance in a collaborative simulation. In the previous example where each node is expected to simulate 6M cells, we can perform a well balanced collaborative simulation using GBMS: both the CPUs and the GPU simulates roughly 3M cells (i.e., $gpu\_ratio = 0.5$).

6. Scaling HOSTA on TianHe-1A

6.1. Minimizing PCI-e data transfer times

Grid blocks need to exchange ghost and singularity data with their neighboring blocks. On heterogeneous systems, when neighboring blocks reside on different compute nodes or devices, data exchanges may involve both intra-node PCI-e data transfer and inter-node MPI communication (see Fig. 9). In CUDA, `cudaMemcpy/cudaMemcpyAsync` can only copy continuous data elements at a time. Considering a 3D grid block with six ghost zones, the data for a single ghost zone is continuously stored in the device memory, while the data for different ghost zones is not. Thus, we can either call `cudaMemcpy` many times for the six ghost zones or call `cudaMemcpy` one time to transfer the whole 3D grid block rather than just ghost zones. Both solutions will result in extra costs on PCI-e data transfers. We present a gather/scatter optimization to minimize transfer times for discontinuous ghost and singularity data (see Fig. 10). Before performing D2H copy, a CUDA kernel `GPU_gather` collects all non-continuous data to a continuous device buffer `OutBuffer_D`, and then the entire buffer is copied to a host buffer `OutBuffer_H`. Correspondingly, before performing H2D copy, a Fortran subroutine `CPU_gather` packs all the updated data to a host buffer `InBuffer_H`, and then the entire buffer is transferred to a device buffer `InBuffer_D`. Finally, we use a scatter kernel `Scatter_kernel` to distribute the data elements in `InBuffer_D` to ghost and singularity cells.

6.2. Overlapping computation and communication

HOSTA has already implemented the overlapping of CPU computation and MPI communication by non-blocking MPI. For example, after calculating `DPV`, HOSTA calls `MPI_Isend` and `MPI_Irecv` to exchange `DPV` and the non-blocking

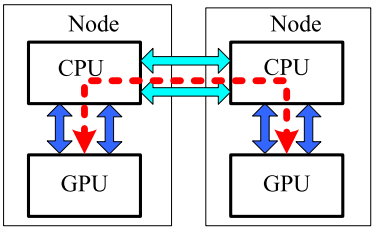


Fig. 9. Inter-node and intra-node data exchanges among heterogeneous devices.

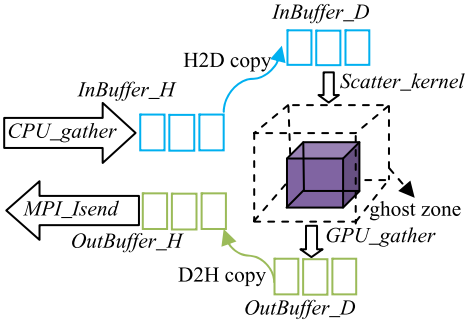


Fig. 10. Gather/scatter optimization.

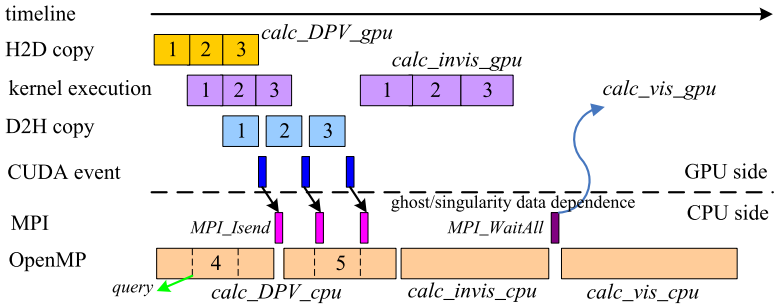


Fig. 11. Overlapping the collaborative CPU–GPU computation and communication for `calc_DPV`, `calc_invis` and `calc_vis`. Grid block *1*, *2* and *3* are calculated by the GPU. Grid block *4* and *5* are calculated by the CPUs.

MPI communication will be overlapped with the following `calc_invis`. Before using `DPV` in `calc_vis`, HOSTA calls `MPI_Waitall` to ensure that `MPI_Irecv` has finished receiving `DPV`.

In this paper, using non-blocking MPI, CUDA multiple streams and CUDA events, we implemented an *as early as possible* non-blocking communication mechanism to further overlap the collaborative computation with PCI-e data transfers and MPI communication. Fig. 11 illustrates the overlapping in `calc_DPV`, `calc_invis` and `calc_vis`. On the GPU side, when a stream finishes its operations of the associated grid block for `calc_DPV`, a CUDA event with the same stream ID is recorded to represent the data dependence between MPI communication and ghost/singularity data calculated by the stream. On the CPU side, we enable a counter in `calc_DPV`. During the execution of `calc_DPV`, for each CPU-computed gird block we query CUDA events several times (e.g., 3 times) to check if there are any GPU-computed grid blocks that have finished their calculations and intra-node data transfers, and also check the state of CPU-computed grid blocks. Then we call `MPI_Isend` or `MPI_Irecv` for ready grid blocks on both sides. Hence, we perform non-blocking communication for each grid block as early as possible, and this helps us better overlap the collaborative computation and the extensive high-order inter-block communication.

7. Performance results

7.1. HOSTA implementation on TianHe-1A

The GPU code is implemented using CUDA C. All CPU subroutines in the time-marching loop except those implementing MPI communications are rewritten as CUDA C kernels. Those kernels are wrapped in C functions to be called from Fortran code. We extend the main program loop to glue the GPU code with the CPU code for collaborative calculations. Since HOSTA is often used for high-resolution flow simulations, we always use a double precision implementation with ECC on. In total,

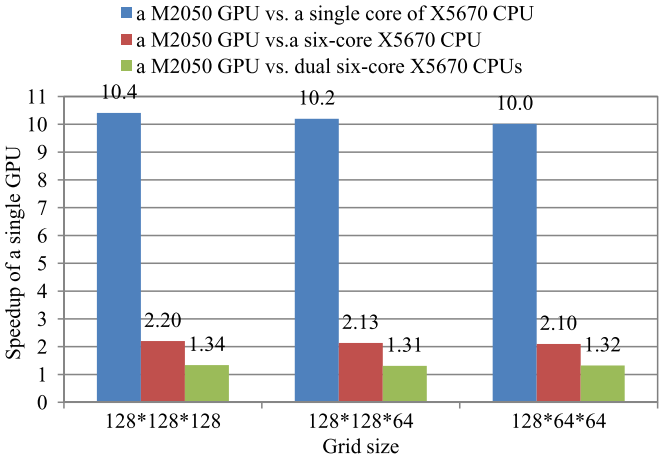


Fig. 12. Single GPU speedup for different grid sizes.

we (4 full-time researchers and several part-time researchers) spent roughly 8 months for the MPI–CUDA version and another 3 months for the collaborative version. As we have mentioned, it is because we need to use diverse programming models/tools. Another important reason is that HOSTA is rather than a *toy* project, but a large-scale software containing more than 25 000 lines of code. Totally we have added more than 16 000 lines of CUDA C/C code and Fortran wrapper code for the final collaborative version. We believe it is also due to the time spent on finding and reducing bugs. By direct comparison of corresponding data fields, we make sure that the GPU-only version and the collaborative version always produces the same results as the CPU version up to machine accuracy. Debugging tends to be harder when various components/routines are tightly coupled, as changes in one may cause bugs to emerge in another. Therefore, making parallelization and optimization on heterogeneous supercomputers is still an exceptionally time-consuming and challenging job.

7.2. Experimental setup and metrics

We use CUDA of version 5.0, Intel icc11.1 for C code and Intel ifort11.1 for Fortran code. All code is compiled with the -O3 option. We use MPICH2-GLEX for MPI communication. We use automatically generated 3D airfoil configurations to facilitate performance evaluation. For each airfoil grid, users can specify the numbers of cells and partition parameters on three grid directions and obtain grid blocks of equal size. This can save a lot of grid generation time for complex geometries. We also present detailed timing results of *C919*. The total double precision floating point operation count on X5670s is measured by Intel Vtune amplifier XE 2011. Since we have no tools to count the GPU operations, we use the CPU operation count as a reference to estimate the GPU and the CPU + GPU machine efficiencies. To collect performance data, we simulate 10 time steps ($max\_step = 10$), with each having 50 sub-iterations ($max\_subiter = 50$), and average the execution time. We define the collaborative efficiency (CE) to evaluate the efficiency loss in CPU–GPU collaboration

$$CE = \frac{SP_{CPU+GPU}}{SP_{CPU} + SP_{GPU}} \times 100\% \tag{15}$$

$SP_{CPU+GPU}$ is the achieved collaborative speedup, taking the performance of the two X5670s as the baseline. For example, if $SP_{CPU+GPU} = 1.8$ and $SP_{GPU} = 1.3$, then CE is $\frac{1.8}{1.0+1.3} \times 100\% \approx 78.3\%$, which represents an efficiency loss of around 22% in the collaboration.

Since we achieve a speedup of about 1.3/1.0 when comparing the M2050 to the X5670s without/with GBMS, *gpu_ratio* in Eq. (14) is set to 0.57 and 0.5 respectively. For simplicity, in default we fix the number of grid blocks (*#block*) on each node to be 8.[1] Correspondingly, we set $(num\_grp, num\_grpstream) = (2, 2)$ in GMBS.[2]

7.3. Node-level performance

In this subsection, we present performance results on one TianHe-1A node. Fig. 12 presents the speedup of a M2050 GPU over a single core of X5670 CPU using 1 OpenMP thread, a six-core X5670 CPU using 6 OpenMP threads and dual six-core X5670 CPUs using 12 OpenMP threads respectively. We evaluated grids of three different problem sizes. We can see that our GPU code achieves a speedup of about 10 over the serial CPU code and the speedup is about 2.1 when comparing a M2050 to an X5670. Note that the price of a M2050 is similar to that of an X5670, and the results are comparable to the speedup

[1] I.e., without GBMS, 3 blocks are calculated on the CPUs and the other 5 are calculated on the GPU; with GBMS, each side calculates 4 blocks.

[2] In practice, different ($num\_grp, num\_grpstream$), *gpu_ratio* and *#block* may lead to different performance, but the maximum achieved performance is always within ±5% of the performance obtained with the default setting. Thus, we only present the results of the default setting.

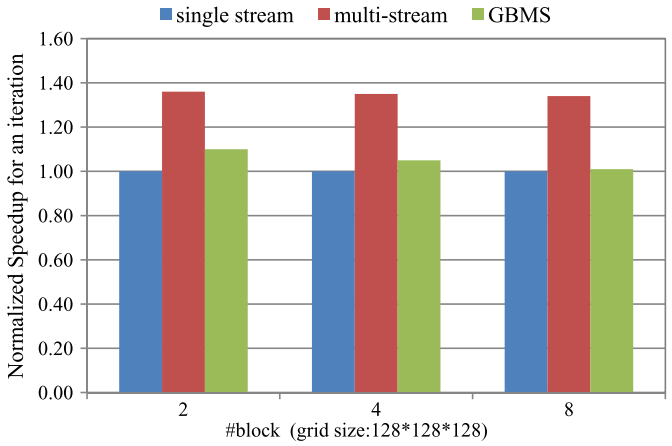


Fig. 13. Performance comparison of different CUDA stream implementation.

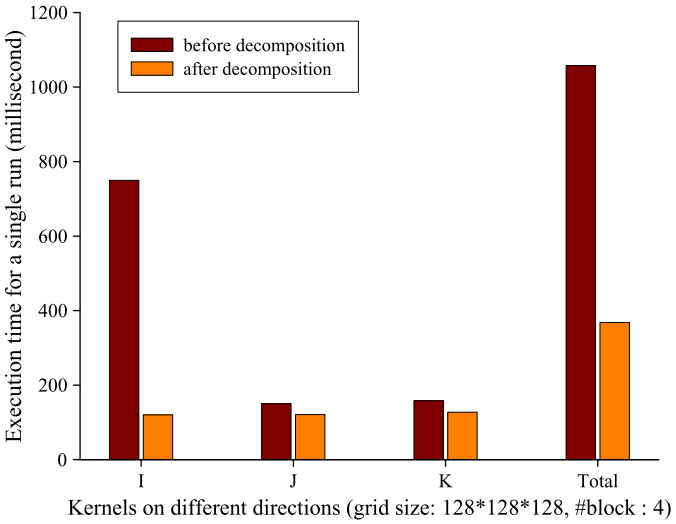


Fig. 14. Performance for `calc_invis` before/after kernel decomposition.

in paper [51] as far as similar-priced comparison of CPU and GPU is concerned. Also note that the power consumption of a M2050 is roughly equal to that of two X5670s and the speedup is about 1.3 when comparing a M2050 to dual X5670s, which further validates GPU's cost-effectiveness. This also motivates us for CPU–GPU collaboration to maximize HOSTA's performance at the node-level, since the sustainable performance of HOSTA on the two CPUs is not negligible (about 80% of the GPU). We get better performance on larger problem sizes, because higher workloads can better overlap the GPU computation and device memory accesses.

Fig. 13 shows the performance comparison of different streaming implementation for one iteration. The grid size is fixed to $128 \times 128 \times 128$ with the number of grid blocks varied from 2 to 8. We take the single CUDA stream performance as the baseline. We see a 25% to 30% performance enhancement when using CUDA streams to simultaneously execute grid blocks on GPU in multi-stream. The overhead of GBMS is indeed a little significant and we lose about 28% performance from multi-stream, due to CPU–GPU synchronization and PCI-e data transfers for selected flow variables. However, the compensation of this overhead is also substantial: we improve HOSTA's maximum simulation capacity on a M2050 from 2M cells to 4M cells. With GBMS, we successfully trade rich compute capacity for poor memory footprint in a GPU.

Fig. 14 shows the performance comparison between before and after the kernel decomposition in `calc_invis`. We reduce about 80% execution time for the I direction inviscid fluxes calculation when using decomposed kernels, but for the J and K directions, the performance improvements are both around 15%. This is because the decomposition in the I direction can ensure more GPU threads to access the global memory in a coalesced manner.

Fig. 16 presents the intra-node speedup and CE with/without GBMS. The GPU-only results are only available for the four small grids, due to the device memory limit. For $256 \times 128 \times 128$ (roughly 4M), we have to use GBMS and SP_{GPU} drops from 1.3 to 1.05. Although losing about 25% performance due to the GBMS overhead, one M2050 is still comparable to two X5670s. For those four small problems, we need not use GBMS in collaboration, and the average $SP_{CPU+GPU}$ and CE are 1.81

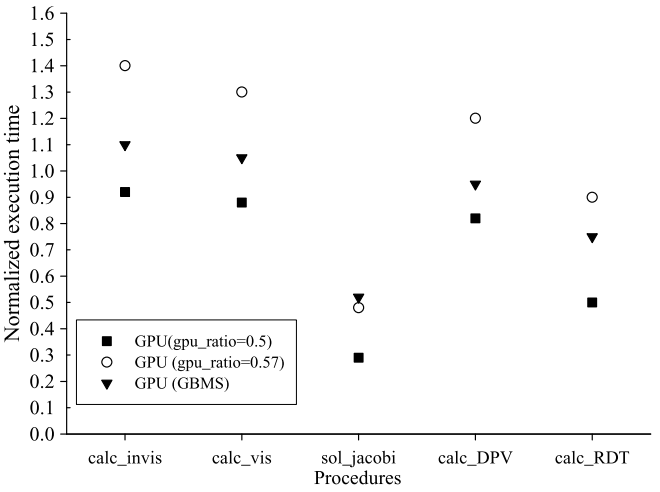


Fig. 15. Normalized execution times of different CPU/GPU procedures.

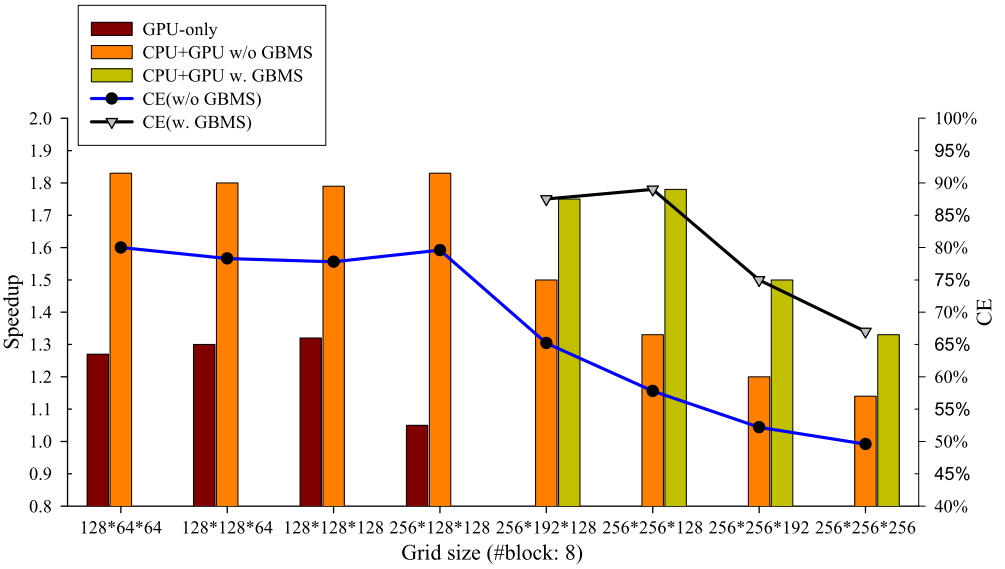


Fig. 16. Intra-node collaborative speedup and efficiency.

and 79% respectively. The collaborative approach has up to 45% performance advantage over the GPU-only approach. For larger problem sizes ($\geq$4M), GBMS plays a very important role for improving both the collaborative speedup and efficiency. Taking $256 \times 256 \times 128$ (roughly 8M) as an example, without GBMS the GPU can only simulates 2M cells and the other 6M cells have to be simulated on the CPUs. Due to the severe load imbalance, $SP_{CPU+GPU}$ is only about 1.3 and *CE* is only about 57%. With GBMS each side simulates 4M cells, $SP_{CPU+GPU}$ and *CE* are increased to about 1.79 and 89% respectively. 8M cells is HOSTA's maximum simulation capacity on one TianHe-1A node for a balanced collaboration, and this is an improvement of 2.3$\times$ compared to 3.5M cells without GBMS. As Fig. 16 shows, further increase in problem size leads to a significant decrease in the collaborative speedup and efficiency. This is because the GPU simulation workload is still limited ($\leq$4M) even using GBMS. Despite of this, the GBMS balanced collaboration always has a notable advantage over the collaborative approach without GBMS.

We note that we lose around 20% speedup from ideal speedup (we get 1.8 of 2.3) in the intra-node collaboration. This is mainly due to the difference in sustainable performance between different kernels on CPU and GPU. As we mentioned, HOSTA has various complex numerical procedures consisting of many kernels, with each implemented in two versions. Those kernels may exhibit different characteristics on different devices. Some kernels run faster on the M2050, and others perform better on the X5670s. Fig. 15 shows the normalized execution time of several time-consuming procedures in HOSTA, taking CPU procedure as the baseline. For an equal problem size on both devices (i.e., *gpu_ratio* = 0.5), the difference is very significant. The GPU version `sol_jacobi` and `calc_RDT` far outperform their CPU counterparts (around 3.5$\times$ and 2$\times$ respectively); while for `calc_invis` and `calc_vis` the two versions almost have the same performance. Even the

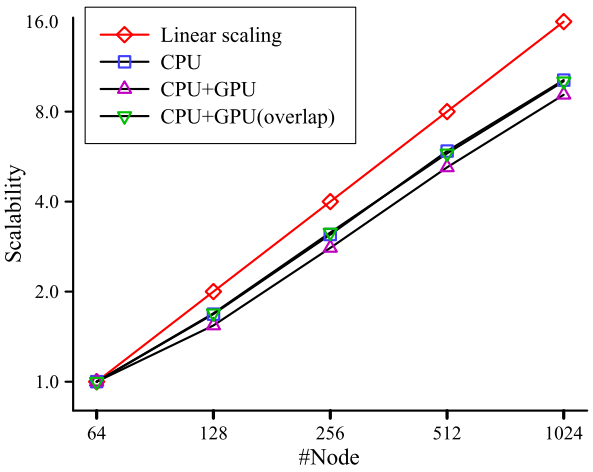


Fig. 17. Strong scalability results.

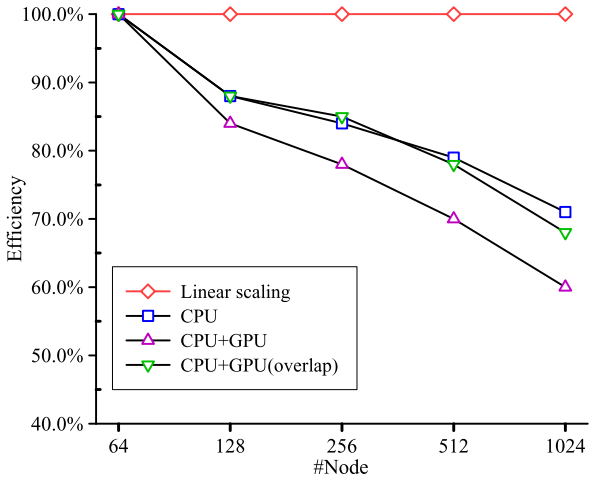


Fig. 18. Weak scalability results.

problem size is adjusted according to Eq. (14) (e.g., *gpu_ratio* = 0.57 in Fig. 15) to maximize HOSTA's performance of a whole iteration, the difference is still obvious. GBMS to some extent levels off the difference, because the GBMS overhead is carefully added to CUDA kernels (i.e., `sol_jacobi`) that run far faster than their counterpart Fortran subroutines.

The kernel characteristic difference is a practical bottleneck for efficient heterogeneous collaboration in multi-kernel applications [35], because kernels on different devices must wait for each other to finish. A possible optimization is to employ a more fine-grained (e.g., at the granularity of grid cells) partition or tailoring partition for different kernels, as described in Section 5. To level off the performance difference, we may also focus on optimizing kernels with larger performance difference on different devices, or use a larger OpenMP parallel region and put the unbalanced and independent kernels together.

7.4. Large-scale performance

In the following scalability tests, the 64 node results are used as the baseline. Fig. 17 shows strong scalability results. The problem size of the baseline test is $256 \times 128 \times 128$ per node, and the total problem size is $64 \times 256 \times 128 \times 128$ (about 268M). Grids are evenly distributed among compute nodes. We achieve a speedup of about 10 when using 1024 nodes for CPU-only results, with an efficiency of around 64%. We use "*CPU + GPU*" to denote the collaborative results without the overlapping optimization. Since the largest problem size per node (i.e., the 64 node test) is about 4M, we need not use GBMS. The collaborative speedup and efficiency decrease to about 9 and 56% respectively on 1024 nodes without the overlapping optimization. We turn on the overlapping optimization to evaluate its effectiveness. The optimized scalability results are similar to the CPU-only results. Fig. 18 presents weak scalability results. The problem size per node is fixed to $256 \times 128 \times 128$. We lose about 27% efficiency from the perfect efficiency of 100% for CPU-only simulations when scaling

Table 1
The execution times (in seconds), GFlops and machine efficiencies (E) for HOSTA when simulating *C919* on TianHe-1A.

#node	64	128	256	512
CPU	124	72	44	30
GPU	104	61	37	26
CPU + GPU (CE)	70 (77%)	43 (73%)	27 (71%)	19 (68%)
CPU GFlops (E)	780 (8.7%)	1344 (7.5%)	2186 (6.1%)	3226 (4.5%)
GPU GFlops (E)	928 (2.8%)	1586 (2.4%)	2580 (2.0%)	3710 (1.4%)
CPU + GPU GFlops (E)	1382 (3.3%)	2250 (2.7%)	3562 (2.1%)	5094 (1.5%)

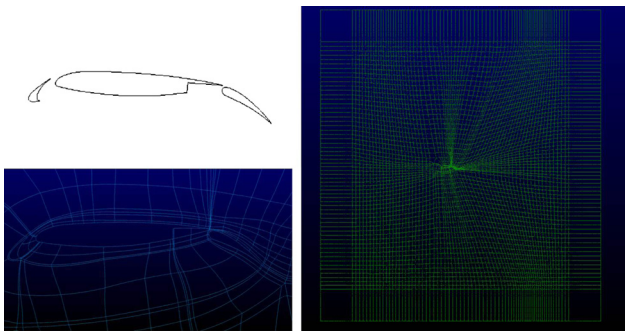


Fig. 19. The grid structure of *30p30n*.

to 1024 nodes. The efficiency loss is up to 32% (with the overlapping) and 40% (without the overlapping) for 1024 node collaboration, again demonstrating the effectiveness of the overlapping optimization.

To summarize, heterogeneous applications involve more complex and expensive data movements between devices at different levels than traditional CPU applications, which indicates that achieving a good parallel scalability and a high efficiency in large-scale systems is a tough challenge. Although the overlapping of communication and computation in HOSTA is fairly effective, we note that four data exchanges in one iteration is indeed a bottleneck in large-scale simulations. We are working on an improved implementation that can avoid some ghost/singularity communication.

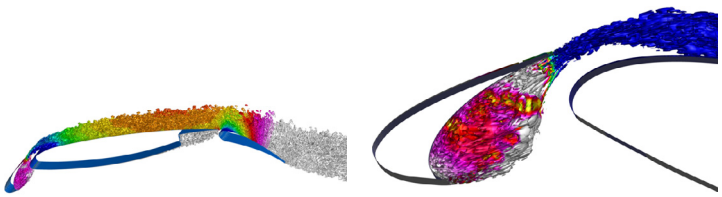
Table 1 shows the detailed timing results of *C919*. *C919* is China's large civil airplane which currently is still under development. Since the grid contains about 150M cells, the 64 node GPU-only results are obtained with GBMS. For other node scales or collaborations, we need not use GBMS. The collaborative efficiency and the parallel efficiency are lower than that of the airfoil configuration, especially for large-number of nodes. This is because *C919* has a more complex geometry which is hard to be partitioned into grid blocks with a similar size. When distributing those blocks to large amount of TianHe-1A nodes, and further allocating them between intra-node CPU and GPU, we cannot achieve as good a load balance (of grid cells) as the less complex configurations such as airfoil. Nevertheless, the benefit of collaboration when simulating large grids with fairly complex geometries like *C919* is still reasonable: generally our collaborative approach can obtain the same performance, but uses only 50% less compute nodes than the CPU-/GPU-only approach on TianHe-1A. This will significantly save the simulation cost of HOSTA.

Note that the machine efficiencies are not high: less than 10% for CPU and less than 3.0% for GPU. For HOSTA-like implicit CFD code, the flop per byte is quite low, but modern cache-based CPU architecture needs to execute large amount of operations per data item for high efficiency. This mismatch can explain the low efficiency (often less than 20% and even less than 10%) of real-world CFD applications [27]. For GPU, it is more difficult to exploit its full potential Flops and requires more tuning work to achieve a higher efficiency.

8. Case study

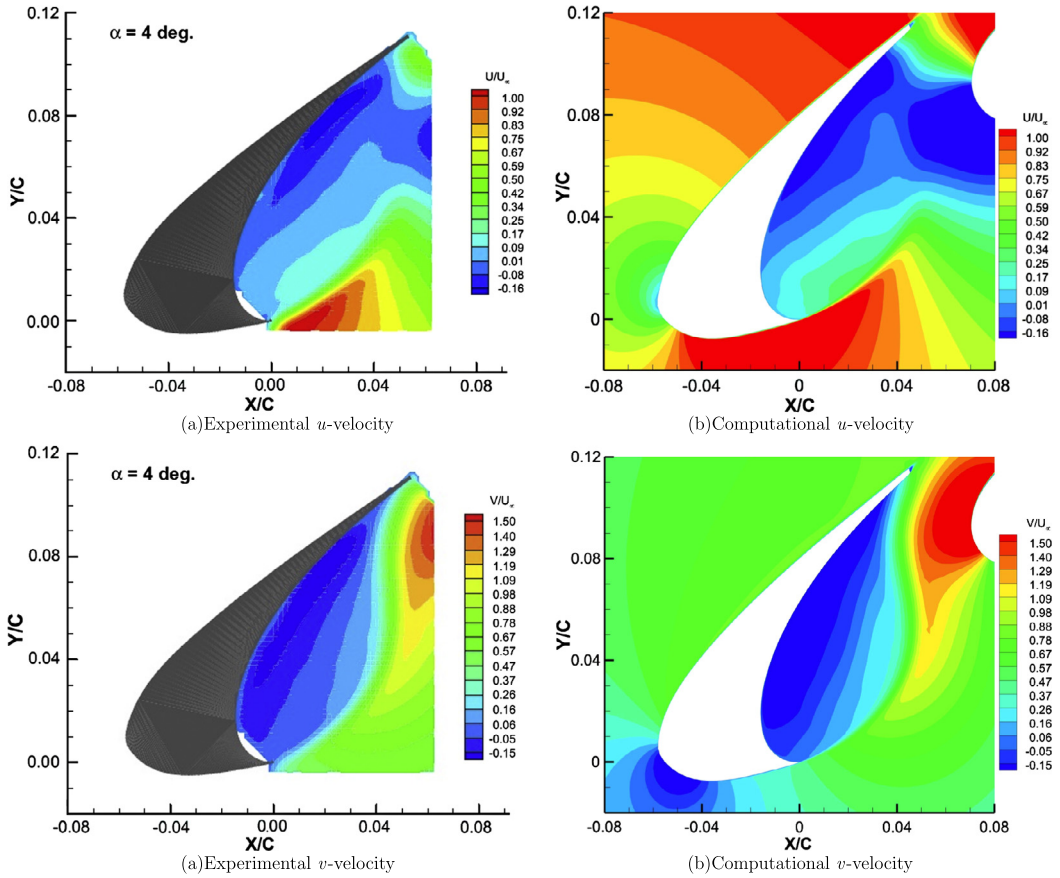
8.1. HDCS-E8T7

To validate HDCS, we perform the Large-Eddy Simulation (LES) of a compressible viscous flow over the EET high-lift airfoil configuration *30p30n* as tested in the LTPT facility at NASA LaRC [32]. The conditions are same as [32]. The incoming Mach number $Ma = 0.17$, and the chord length $c = 0.457$ m, the corresponding chord Reynolds number is 1.71×10^6, the angle of attack is 4°. Fig. 19 shows the grid structure. The *30p30n* configuration used in this work contains 13 computational domains (with approximately 800M cells and 2400 grid blocks) and highly stretched meshes are used. The spacing on the surface is 0.0002 (normalized by the reference length $L\_ref = 0.0254$ m) in the normal direction. At the outer boundary, far-field boundary conditions based on the LODI approximation [40] and sponge technique [11] are prescribed. The no-slip condition is invoked on the airfoil surface, together with fifth-order accurate approximations for an adiabatic wall and zero normal pressure gradient. At the spanwise boundaries, periodicity is applied.



(a)Whole field (b)Near the slat

Fig. 20. Iso-surfaces of the second invariant of velocity gradient tensor for instantaneous vortex structure.



(a)Experimental u-velocity (b)Computational u-velocity

(a)Experimental v-velocity (b)Computational v-velocity

Fig. 21. Contours of the velocity.

HDCS-E8T7 uses the same idea as Monotone Integrated Large-Eddy Simulation (MILES) [12] to simulate the turbulent flow, i.e., the effects of explicit LES models are imitated by the truncation error. The computation has been advanced with a time step of 2.5×10^{-6} s for total 30 000 time steps. Fig. 20 demonstrates the iso-surfaces of the second invariant colored by density contours for the instantaneous flow. The second invariant Q is:

$$Q = (\Omega_{ij}\Omega_{ij} + S_{ij}S_{ij})/2 \tag{16}$$

where $\Omega_{ij} = (u_{i,j} - u_{j,i})/2$ and $S_{ij} = (u_{i,j} + u_{j,i})/2$. Due to the fine mesh, we can see the details about the vortex structure. Fig. 21 shows the mean velocity contours near the slat comparing with the experimental PIV measurements. We can see a qualitative agreement between the computation and the experiment with all salient features in the experiment being reproduced in the computation. Fig. 22 compares the mean velocity-vector near the slat between computation and experiment. The computational mean spanwise-vorticity comparing with the experimental measurement is shown in Fig. 23. We clearly see that the shear layer separating from the leading edge is demonstrated by the computation.

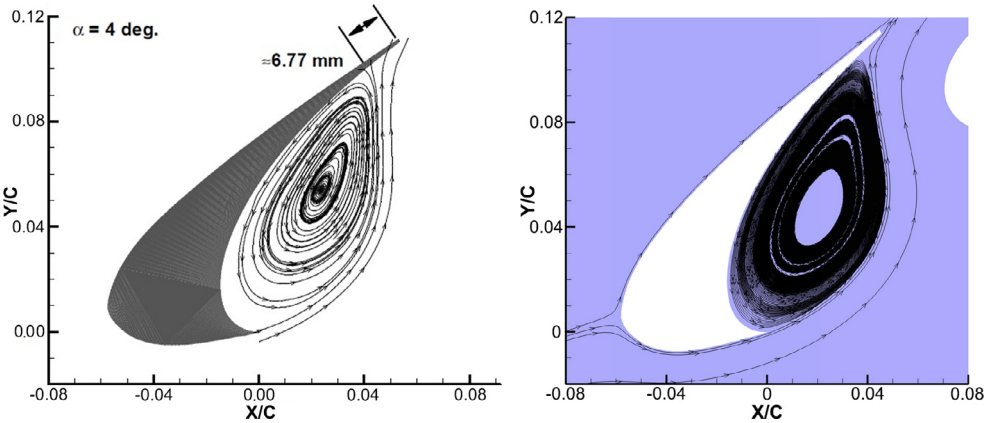


Fig. 22. The time-averaged velocity-vector distribution.

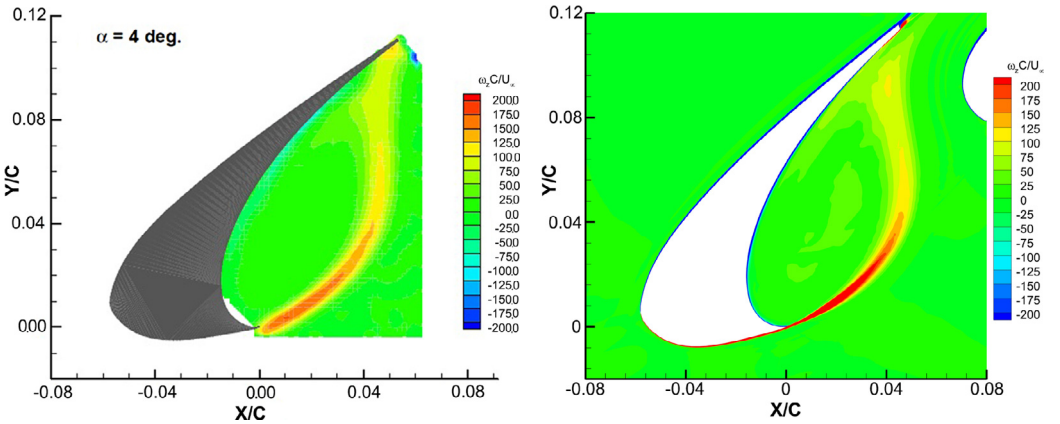


Fig. 23. The time-averaged spanwise-vorticity distribution.

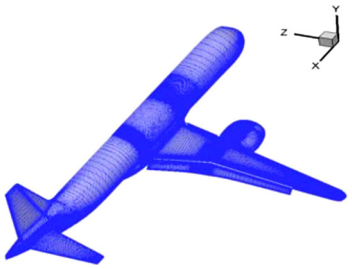


Fig. 24. The grid structure of *C919*.

8.2. WCNS-E-5

C919 airliner is China's large civil airplane which currently is still under development. We perform the high-order aerodynamic simulation using WCNS-E-5 to evaluate its takeoff state and configuration. The body surface grid of the whole airplane is shown in Fig. 24. The geometry is very complex and contains the wing, the body, the horizontal tail, the vertical fin, the pylon, the nacelle, the winglet, the leading edge slat and the trailing edge flap. The grid points near the leading and trailing wing edges, the post-wing, the horizontal and vertical tails are refined, with a 0.00001 m first-level mesh spacing. The reference area is 120 m^2 and the distance between the reference point of torque and airplane nose is 18.35 m. The incoming Mach number $Ma = 0.2$, and the chord length $c = 4.8$ m, the corresponding chord Reynolds number is 2.0×10^7, the angle of attack is 8°. We use the Roe flux-splitting scheme and the two-equation turbulence model in the simulation.

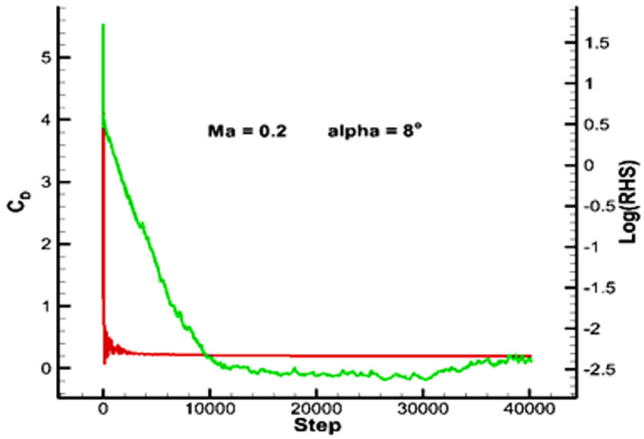


Fig. 25. The aerodynamic coefficient and the residual.

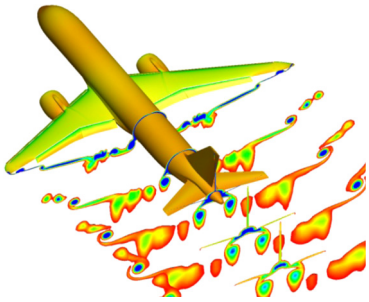


Fig. 26. The equivalent total pressure for different cross sections.

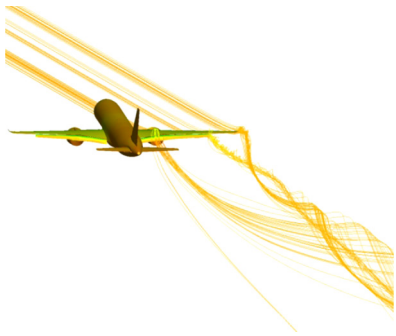


Fig. 27. Streamline of the wing tip, the side edge of wing flap and the nacelle.

Fig. 25 shows convergence curves of the aerodynamic coefficient and the residual. The residual decreases about five order of magnitude and the drag coefficient keeps stable after 40 thousands step calculations, indicating the convergence of pressure and flow field in the simulation. In Fig. 26, the nephogram for the equivalent total pressure of different cross sections is shown. Fig. 27 shows the streamline of the wing tip, the side edge of wing flap and the nacelle. We clearly see the vortex at the wing tip, the slide and the trailing edge. The results indicate a very subtle numerical simulation based on the WCNS-E-5 high-order scheme.

9. Related work

In the past twenty years, there have been many studies in developing and applying high-order compact finite difference schemes in CFD. In 1981, Harten et al. [28] developed a fourth order accurate implicit finite difference scheme for shock capturing. Lele [36] developed several central compact schemes with spectral-like resolution. Cockburn et al. [15] developed

a fourth order compact TVD scheme for shock calculations. Visbal and Gaitonde [45] use filters to prevent numerical oscillations of central compact schemes. Recently, Lele's central compact schemes have been successfully applied by Rizzetta et al. [42] in the simulation of low speed flows. Adams et al. [6] developed a compact-ENO (Essentially Non-Oscillatory) scheme. Pirozzoli [39] developed a compact-WENO (Weighted Essentially Non-Oscillatory) scheme which was further improved by Ren et al. [41]. The WCNS scheme developed by us in 2000 has been successfully applied in many complex flows. In [37], Nonomura shows the excellent freestream and vortex preservation properties of WCNS on curvilinear grids, compared with those of WENO. Recently we have shown the importance of fulfilling GCL when applying high-order finite difference schemes on complex grids. Based on the idea of SCMM, we have proposed the HDCS scheme with inherent dissipation that can conveniently fulfill the GCL.

Many prior work has shown the experiences of porting CFD codes to GPUs, with impressive speed-ups. Phillips et al. [38] developed a 2D compressible Euler solver on a GPU cluster. Appa et al. [9] implemented and optimized an unstructured 3D explicit finite volume CFD code on multi-core CPUs and GPUs. Corrigan et al. [16] ported an adaptive, edge-based finite element code FEFLO to GPU clusters in a semi-automated fashion: the existing Fortran–MPI code is preserved while the translator inserts data transfer calls as required. Brandvik and Pullan [13] implemented a multi-GPU enabled Navier–Stokes solver for flows in turbomachines. They reported almost linear weak scaling using up to 16 GPUs. All the above studies were tested on small-scale GPU platforms using low-order CFD methods. For large-scale GPU clusters, Ali et al. [34] parallelized an incompressible solver for turbulence with a second-order scheme and reported scalability results on 64 GPUs. But no validation is presented. Jacobsen et al. [31] parallelized a CFD solver for incompressible fluid flows using up to 128 GPUs. They further demonstrated the large eddy simulation of a turbulent channel flow on 256 GPUs [17], but they only simulated a lid-driven cavity problem using 1D decomposition and low-order schemes. They also compare the scalability between the tri-level MPI + OpenMP + CUDA and the dual-level MPI + CUDA implementation [30], but they only use OpenMP to substitute intra-node MPI communication rather than collaborating CPU and GPU for computation as we have done in our work.

All the above studies use low-order CFD methods. Due to the complexity, high-order CFD schemes generally require extra implementation and optimization efforts on GPUs. Tutkun et al. [44] implemented a six-order compact finite difference scheme on a single GPU. Antoniou et al. [8] implemented a high-order solver on multi-GPUs for the compressible turbulence using WENO. But the solver can only run on a single node platform containing 4 GPUs for very simple domains like a 2D or 3D box. Their single-precision implementation achieve a speedup of 53 when comparing 4 Tesla C1070s with a single core of an Xeon X5450. Castonguay et al. [14] parallelized the first high-order, compressible viscous flow solver for mixed unstructured grids with MPI and CUDA, where the Vincent–Castonguay–Jameson–Huynh method is used. A flow over SD7003 airfoil and a flow over sphere is simulated using 32 GPUs. With MPI + CUDA, Appleyard et al. [10] accelerated the solution of the level set equations for interface tracking using an HOUC (High-Order Upstream Central) scheme. But they only demonstrated performance results on 4 GPUs. Zaspel et al. [51] implemented an incompressible double-precision two-phase solver on GPU clusters using a fifth-order WENO scheme. The test problem is a rising bubble of air inside a tank of water with surface tension effects, and parallel performance results are reported using up to 48 GPUs. In [48], with MPI and CUDA, we parallelize HOSTA on TianHe-1A using HDCS, but no CPU–GPU collaboration as well as heterogeneous load balance were discussed or implemented. This work was a major extension based on [48]. To summarize, the above GPU-enabled CFD simulations using high-order schemes are still preliminary as far as grid complexity, problem size and parallel scale are comprehensively concerned.

10. Conclusion and future work

The employment of high-order CFD schemes is an effective approach to obtain high-resolution and high fidelity simulation results for complex flow problems. In the past twenty years, we have developed several high-order compact finite difference schemes including WCNS and HDCS. To tackle the problems challenging the application of high-order schemes in complex grids, we have also proposed SCMM to fulfill GCL and CBIC for high-order block-interface conditions. Those high-order CFD schemes and methods have been successfully used in many complex flow problems with reasonably complex geometries, showing promising perspective for engineering applications.

Large-scale high-order CFD simulations often need to exploit powerful supercomputers to accelerate their running. In this work, with MPI + OpenMP + CUDA, we ported and optimized HOSTA, our in-house high-order CFD software for 3D multi-block structured grids, onto the GPU-accelerated supercomputer TianHe-1A. We presented some novel techniques to achieve balanced and scalable high-order CPU–GPU collaborative simulations on TianHe-1A. The approach and implementation of GPU parallelization, CPU–GPU collaboration as well as balancing scheme provide a fairly general experience of similar porting and optimizing efforts for multi-block CFD applications. As a case study, we have successfully performed CPU–GPU collaborative high-order accurate simulations with two large-scale complex grids using hundreds of TianHe-1A nodes.

For future work, besides fine tuning of the aforementioned bottlenecks in the collaborative code, we are planning to port HOSTA to China's new supercomputer TianHe-2, which was ranked No. 1 in Jun., 2013 for large-scale simulations such as direct numerical simulations of turbulence. Due to the usage of traditional programming models, we expect that the work would take us relatively less time. Since porting and optimizing CFD code often involves some common programming efforts such as restructuring kernel skeletons and data structures, in the long run we will encapsulate our experience learned in

this paper and further work into an application programming and auto-tuning infrastructure for multi-block CFD simulations on heterogeneous systems.

Acknowledgements

This paper was supported by the National Science Foundation of China under Grant No. 11272352 and No. 61379056, and the Open Research Program of China State Key Laboratory of Aerodynamics under Grant No. SKLA20130105.

References

[1] NVIDIA Corp., CUDA C Programming Guide v4.2, 2012.
[2] Cuda Fortran, available online, http://www.pgroup.com/cudafortran, 2013.
[3] Khronos OpenCL working group, http://www.khronos.org/opencl, 2013.
[4] Mont-Blanc project home page, available online, http://www.montblanc-project.eu/, 2013.
[5] Top500 cite, available online, http://top500.org/, 2013.
[6] N. Adams, K. Shariff, A high-resolution hybrid compact-ENO scheme for shock–turbulence interaction problems, J. Comput. Phys. 127 (1996).
[7] E. Albayrak, Improving application behavior on heterogeneous manycore systems through kernel mapping, Parallel Comput. (2013).
[8] A.S. Antoniou, K.I. Karantasis, E.D. Polychronopoulos, Acceleration of a finite-difference WENO scheme for large-scale simulations on many-core architectures, AIAA Paper 2010-0525, 2010.
[9] J. Appa, J. Sharpe, P. Moinier, An unstructured 3D CFD code optimised for multicore and graphics processing units, in: MRSC2010 Proceedings.
[10] J. Appleyard, D. Drikakis, Higher-order CFD and interface tracking methods on highly-parallel MPI and GPU systems, Comput. Fluids 46 (2011) 101–105.
[11] D.J. Bodony, Analysis of sponge zones for computational fluid mechanics, J. Comput. Phys. 212 (2006) 681–702.
[12] J. Boris, F. Grinstein, E. Oran, R. Kolbe, New insights into large eddy simulation, Fluid Dyn. Res. 10 (1992).
[13] T. Brandvik, G. Pullan, An accelerated 3D Navier–Stokes solver for flows in turbomachines, in: ASME Turbo Expo 2009: Power for Land, Sea and Air.
[14] P. Castonguay, D.M. Williams, P.E. Vincent, M. Lopez, A. Jameson, On the development of a high-order, multi-GPU enabled, compressible viscous flow solver for mixed unstructured grids, AIAA Paper 2011-3229, 2011.
[15] B. Cockburn, C.W. Shu, Nonlinearly stable compact schemes for shock calculations, SIAM J. Numer. Anal. 31 (1994) 607–630.
[16] A. Corrigan, R. Lohner, Porting of FEFLO to multi-GPU clusters, AIAA Paper 2011-0948, 2011.
[17] R. DeLeon, D. Jacobsen, I. Senocak, Large-eddy simulations of turbulent incompressible flows on GPU clusters, Comput. Sci. Eng. 15 (2013) 26–33.
[18] X. Deng, Y. Jiang, M. Mao, H. Liu, G. Tu, Developing hybrid cell-edge and cell-node dissipative compact scheme for complex geometry flows, in: The Ninth Asian Computational Fluid Dynamics Conference Proceedings.
[19] X. Deng, M. Mao, G. Tu, et al., Extending the fifth-order weighted compact nonlinear scheme to complex grids with characteristic-based interface conditions, AIAA J. 48 (2010) 2840–2851.
[20] X. Deng, M. Mao, G. Tu, et al., High-order and high accurate CFD methods and their applications for complex grid problems, Commun. Comput. Phys. 11 (2012) 1081–1102.
[21] X. Deng, M. Mao, G. Tu, H. Liu, H. Zhang, Geometric conservation law and applications to high-order finite difference schemes with stationary grids, J. Comput. Phys. 230 (2011) 1100–1115.
[22] X. Deng, Y. Min, M. Mao, H. Liu, G. Tu, H. Zhang, Further studies on Geometric conservation law and applications to high-order finite difference schemes with stationary grids, J. Comput. Phys. 239 (2013) 90–111.
[23] X. Deng, H. Zhang, Developing high-order weighted compact nonlinear schemes, J. Comput. Phys. 165 (2000) 22–44.
[24] J. Ekaterinaris, High-order accurate, low numerical diffusion methods for aerodynamics, Prog. Aerosp. Sci. 41 (2005) 192–300.
[25] K. Fujii, T. Nonomura, S. Tsutsumi, Toward accurate simulation and analysis of strong acoustic wave phenomena—a review from the experience of our study on rocket problems, Int. J. Numer. Methods Fluids 64 (2010) 1412–1432.
[26] GPGPU, General-purpose computation on graphics hardware, available online, http://gpgpu.org/, 2013.
[27] W.D. Gropp, D.K. Kaushik, D.E. Keyes, B.F. Smith, Towards realistic performance bounds for implicit CFD codes, in: ParCFD.
[28] A. Harten, H. Tal-Ezer, On a fourth order accurate implicit finite difference scheme for hyperbolic conservation laws, II. Five-point schemes, J. Comput. Phys. 41 (1981) 329–356.
[29] Intel, Many integrated core (MIC) architecture, available online, http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html, 2013.
[30] D.A. Jacobsen, I. Senocak, Scalability of incompressible flow computations on multi-GPU clusters using dual-level and tri-level parallelism, AIAA Paper 2011-947, 2011.
[31] D.A. Jacobsen, J.C. Thibault, I. Senocak, An MPI–CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters, AIAA Paper 2010-0522, 2010.
[32] L. Jenkins, M. Khorrami, M. Choudhari, Characterization of unsteady flow structures near leading-edge slat: Part I. PIV measurements, AIAA Paper 2004-2801, 2004.
[33] G. Jiang, C. Shu, Efficient implementation of weighted ENO schemes, J. Comput. Phys. 126 (1996) 202–228.
[34] A. Khajeh-Saeed, J.B. Perot, Computational fluid dynamics simulations using many graphics processors, Comput. Sci. Eng. (2012).
[35] K. Kofler, I. Grasso, B. Cosenza, T. Fahringer, An automatic input-sensitive approach for heterogeneous task partitioning, in: ICS.
[36] S. Lele, Compact finite difference schemes with spectral-like resolution, J. Comput. Phys. 103 (1992) 16–42.
[37] T. Nonomura, Free-stream and vortex preservation properties of high-order WENO and WCNS on curvilinear grids, Comput. Fluids 39 (2010) 197–214.
[38] E.H. Phillips, Y. Zhang, R.L. Davis, J.D. Owens, Rapid aerodynamic performance prediction on a cluster of graphics processing units, AIAA Paper 2009-565, 2009.
[39] S. Pirozzoli, Conservative hybrid compact-WENO schemes for shock–turbulence interaction, J. Comput. Phys. 179 (2002) 81–117.
[40] T. Poinsot, S.K. Lele, Boundary conditions for direct simulations of compressible viscous flows, J. Comput. Phys. 101 (1992) 104–129.
[41] Y. Ren, M. Liu, H. Zhang, A characteristic-wise hybrid compact-WENO schemes for solving hyperbolic conservations, J. Comput. Phys. 192 (2005) 365–386.
[42] D. Rizzetta, M. Visbal, P. Morgan, A high-order compact finite difference scheme for large-eddy simulation of active flow control, Prog. Aerosp. Sci. 44 (2008) 397–426.
[43] J. Trulio, K. Trigger, Numerical solution of the one-dimensional hydrodynamic equations in an arbitrary time-dependent coordinate system, Technical Report UCLR-6522, University of California, Lawrence Radiation Laboratory, 1961.
[44] B. Tutkun, F.O. Edis, A GPU application for high-order compact finite difference scheme, Comput. Fluids 55 (2012) 29–35.
[45] M. Visbal, D. Gaitonde, High-order accurate methods for complex unsteady subsonic flows, AIAA J. 37 (1999) 1231–1239.

[46] Y. Wang, L. Zhang, Y. Che, W. Liu, C. Xu, H. Liu, TH-meshsplit: a multi-block grid repartitioning tool for parallel CFD applications on heterogeneous CPU/GPU supercomputer, in: ParCFD.
[47] Z. Wang, High-order methods for the Euler and Navier–Stokes equations on unstructured grids, Prog. Aerosp. Sci. 43 (2007) 1–41.
[48] C. Xu, L. Zhang, X. Deng, Y. Jiang, W.C.J. Fang, Y. Che, Y. Wang, W. Liu, Parallelizing a high-order CFD software for 3D, multi-block, structural grids on the TianHe-1A supercomputer, in: International Supercomputing Conference.
[49] X. Yang, X. Liao, K. Lu, et al., The TianHe-1A supercomputer: its hardware and software, J. Comput. Sci. Technol. 26 (2011) 344–351.
[50] A. Ytterstrom, A tool for partitioning structured multiblock meshes for parallel computational mechanics, Int. J. High Perform. Comput. Appl. 11 (1997) 336–343.
[51] P. Zaspel, M. Griebel, Solving incompressible two-phase flows on multi-GPU clusters, Comput. Fluids (2012).