Hindawi
Scientific Programming
Volume 2017, Article ID 4610138, 16 pages
https://doi.org/10.1155/2017/4610138

Research Article

Parallelization of an Unsteady ALE Solver with Deforming Mesh Using OpenACC

Wenpeng Ma,[1] Zhonghua Lu,[2] Wu Yuan,[2] and Xiaodong Hu[2]

[1]*College of Computer and Information Technology, Xinyang Normal University, Henan 464200, China*
[2]*Supercomputing Center of Chinese Academy of Sciences, Beijing 100190, China*

Correspondence should be addressed to Wenpeng Ma; markwinpe@163.com

Received 28 November 2016; Revised 19 May 2017; Accepted 11 June 2017; Published 16 July 2017

Academic Editor: Can Özturan

This paper presents a parallel, GPU-based, deforming mesh-enabled unsteady numerical solver for solving moving body problems by using OpenACC. Both the 2D and 3D parallel algorithms based on spring-like deforming mesh methods are proposed and then implemented through OpenACC programming model. Furthermore, these algorithms are coupled with an unstructured mesh based, implicit time scheme integrated numerical solver, which makes the full GPU version of the solver capable of handling unsteady calculations with deforming mesh. Experiments results show that the proposed parallel deforming mesh algorithm can achieve over 2.5x speedup on K20 GPU card in comparison with 20 OpenMP threads on Intel E5-2658 V2 CPU cores. And both 2D and 3D cases are conducted to validate the efficiency, correctness, and accuracy of the present solver.

1. Introduction

Many unsteady fluid flow simulations, such as fluid-structure interaction (FSI) [1–3] and wing-store separation [4], involve slight deformation of wall boundaries or relative motion between components in Computational Fluid Dynamics (CFD). This kind of problems is termed as dynamic mesh problems in unsteady calculations. So far various kinds of dynamic mesh approaches [3, 5–8] have been proposed and validated.

According to the present literatures, the dynamic mesh approaches can be divided into two categories. The first class refers to the overset grid technique [7, 9–12] in which mesh movements can be modeled independently for the involved boundary that is supposed to be moved. The meshes of different moving parts overlap with each other to form the complete overset grid system. This technique, applied in both structured [10, 11] and unstructured [7] grids, alleviates the complexity of mesh generation and improves the mesh quality of local domains. However, it adds the difficulty of the mesh preprocessing work [7, 9, 11, 13] in which the interpolated information of each pair of overlapped mesh-block needs to be determined from each one. An alternative category, termed as deforming mesh technique, is about using single, consistent mesh and deforming it to conform the new geometric shape without changing the connectivity of the whole computational mesh. In this category, the node connectivity-based methods consider each two neighboring mesh nodes and treat the whole mesh domain as a spring network. The representative methods involve 2D spring-analogy [14], 2D torsional spring [15], 3D ball-vertex [16], and 3D torsional spring [17, 18] approaches. Contrary to the node connectivity-based methods, the nonconnectivity methods move the mesh nodes point-by-point by calculating the relative position with respect to the given reference position. RBF- (radial basis functions-) based algorithm [19] and DGM (Delaunay grid mapping) [20] algorithm are two popular approaches in nonconnectivity methods.

Parallel algorithm design and implementation (high performance computing, HPC) for CFD applications have become popular and necessary in recent decades due to the ever-growing computational scale. And the flow fluid simulations based on the traditional parallel techniques, such as MPI (message passing interface) [21] and OpenMP [22], have been studied in [3, 4, 6, 9, 11]. Since Nvidia's CUDA (Compute Unified Device Architecture) [23] was

published in 2007, GPU computing has attracted more and more attention due to its high memory bandwidth and light-weighted threads of parallelism. CUDA also provides both hardware configurations and software supports for one to develop programs in parallel by using C/C++ language. And recent years have witnessed a lot of numerical solvers [24–30] that involve various CFD areas on the GPU device.

However, the GPU-based developed solvers are most of the steady ones, which do not involve moving mesh. Even in unsteady calculations [31–33] on GPU device, either no mesh deformation or dynamic overset grid technique is studied. Furthermore, the developed GPU-based solvers are implemented by CUDA programming model in which the core functions of the source code require the programmers to redesign in CUDA-kernel subroutines. This code rewriting procedure is a time-consuming job that leads to low efficiency of GPU code development. With this in mind, the objective of this paper is to develop an accelerated, unstructured mesh based, unsteady solver that is capable of handling deforming mesh for moving body problems on the GPU platform by employing the new directive-based parallel programming interface-OpenACC. The proposed solver requires minor code revision effort but can show well parallel performance on the GPU device. The present paper is organized as follows: Section 2 gives a brief introduction of the OpenACC programming model; Section 3 states the unsteady numerical algorithms of ALE formulation. Both 2D and 3D parallel deforming mesh algorithms (PDMA) and OpenACC implementation are proposed in Section 4. The implementation of an unsteady solver that is coupled with the proposed algorithm for moving body problems is presented in Section 5; Section 6 reports the experiments result of the ALE unsteady solver with deforming mesh; and the last section is the summary of the work.

2. OpenACC Programming Model

The past several years have witnessed the development of heterogeneous architectures, such as DSPs (digital signal processors), FPGA (Field-Programmable Gate Array), and GPU (Graphics Processing Unit), capable of providing great computational power as an auxiliary accelerator for various applications in scientific areas. The programming interfaces, meanwhile, including OpenCL [34], CUDA [23], and C++AMP [35], emerge for developers to design the corresponding algorithm on each architecture. However, in order to explore the potential of computing of theses architectures thoroughly, the programming models require users to understand the hardware-level framework well and put large amount of effort on rewriting the codes, which makes the code migration work with low efficiency. OpenACC [36], published in late 2011, is designed as a programming model that allows researchers to accelerate their scientific codes on different processors and platforms by using high-level compiler directives. The design concept of OpenACC resembles OpenMP [22]: one needs to firstly identify the areas that should be parallelized and then accelerate them by declaring appropriate directives or calling runtime library routines. This directive-based programming model not only makes one parallelize application codes with minor effort but also enable developers to run a single source code on different accelerating platforms, such as many-core CPUs, graphics processing units (GPU). The OpenACC standard is currently supported by four commercial compiler vendors, CAPS [37], Cray [38], PGI [39], and Nvidia [40]. It is quickly becoming a popular, effective, attractive programming model that offers scientists and researchers an efficient way to develop platform independent, portable and parallel scientific codes. Anyone who is familiar with scientific language, such as C, C++, and Fortran, is capable of porting the codes on an accelerator. From the point of view of architecture, the OpenACC provides an abstract model by defining host and accelerator device with a hierarchy of memories [41]. The accelerator device could be the same device as a CPU core, a GPU card, or a future accelerator device. The interactions of different devices, such as memory management, data transfers, and asynchronous operations, can be achieved by compiler and runtime library routines.

3. ALE Formulation for Unsteady Simulation

The numerical model for unsteady flows with moving or deforming mesh employed in present paper is Arbitrary Lagrangian Eulerian (ALE) formulation for Navier Stokes equations, which can be written in integral form as

$$\frac{\partial}{\partial t}\int_{\Omega}\mathbf{W}d\Omega+\oint_{\partial\Omega}\left(\mathbf{F}_c^M-\mathbf{F}_v\right)dS=0, \tag{1}$$

where Ω is control volume, dS is the integral surface element, $\mathbf{W}=[\rho,\rho u,\rho v,\rho w,\rho E]^T$ denotes five conservative variables in three dimensions, and $\mathbf{F}_c^M=\mathbf{F}_c^{\text{inv}}-\mathbf{F}_c^{\text{ALE}}$, $\mathbf{F}_v$ represent the convective fluxes on moving or deforming mesh and viscous fluxes, respectively. These two vectors are given by

$$\mathbf{F}_c^M=\begin{pmatrix}\rho V_r\\ \rho\mathbf{U}V_r+\mathbf{N}p\\ (\rho E+p)V_r+V_t p\end{pmatrix},\qquad \mathbf{F}_v=\begin{pmatrix}0\\ \boldsymbol{\tau}\mathbf{N}\\ \boldsymbol{\Theta}\cdot\mathbf{N}\end{pmatrix}, \tag{2}$$

where ρ, p denote the density and pressure, respectively, $\mathbf{U}=[u,v,w]^T$ represents the velocity components in each direction, $\mathbf{N}=[n_x,n_y,n_z]^T$ stands for the unit normal vector of the integral surface, and $\tau=(\tau)_{ij}$ $(i,j=x,y,z)$ denotes the viscous stress tensor. Two velocity vectors, $\mathbf{V}_t$ and $\mathbf{V}_r$, satisfy $V_t=\mathbf{U}_t\cdot\mathbf{N}$ and $V_r=\mathbf{U}\cdot\mathbf{N}-V_t$, where the vector $\mathbf{U}_t=[u_t,v_t,w_t]^T$ is the velocity of the mesh. The component of $\boldsymbol{\Theta}$, denoted as $\Theta_i=u\tau_{ix}+v\tau_{iy}+w\tau_{iz}+k(\partial T/\partial i)$ $(i=x,y,z)$, stands for the work of viscous stresses and the heat conduction. By introducing two additional equations, $p=\rho RT$ and $p=(\gamma-1)\rho(E-(u^2+v^2+w^2)/2)$, the complete ALE equations contain seven unknown variables to be solved.

In the case of unsteady simulations involving deforming or moving mesh, Geometric Conservation Law (GCL) [42]

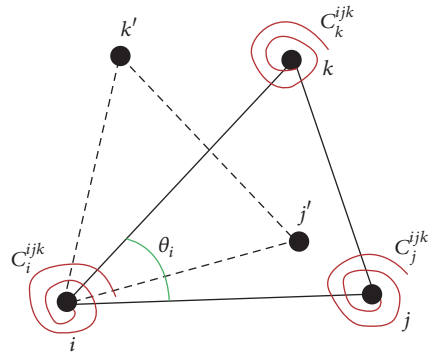

Figure 1: Two-dimensional torsional spring model.

is generally satisfied to avoid errors induced by the deformed control volumes. Written as integral form, GCL reads

$$\frac{\partial}{\partial t}\int_{\Omega} d\Omega - \oint_{\partial\Omega} V_t dS = 0. \tag{3}$$

By moving the second term to the right side of (3) and performing second-order time discretization on the left term of (3), the variation of control volumes in different time levels can be expressed as

$$\Omega^{n+1} = \frac{4}{3}\Omega^n - \frac{1}{3}\Omega^{n-1} + \frac{2}{3}\triangle t \sum_{m=1}^{N_f} (V_t)_m \triangle S_m. \tag{4}$$

4. Deforming Mesh Algorithm on the GPU Platform

4.1. Two-Dimensional Parallel Deforming Mesh Algorithm. The spring-like deforming mesh algorithms [14, 15] have been applied widely due to its robustness in very complex configurations. The main idea of these approaches is to view the unstructured grid of the computational domain as a network of springs which are constructed by the connectivity of nodes. The equilibrium equations of the spring network are satisfied when the grid is static and the equations are satisfied again after the grid movement by providing the displacements of moving nodes as boundary conditions. Then the displacements of inner nodes can be determined through solving a linear system of equations. For the convenience of the following statement, the classical two-dimensional torsional spring deforming mesh approach [15] is reviewed first.

As illustrated in Figure 1, the torsional stiffness C_m^{ijk} is defined as $1/\sin^2\theta_i^{ijk}$, which is used as collapse control parameters to avoid grid crossover problems during the grid movement. Then the moments, expressed as a vector $\mathbf{M^{ijk}}$, can be written as $\mathbf{M^{ijk}} = \mathbf{C^{ijk}}\Delta\boldsymbol{\theta}^{\mathbf{ijk}}$, where $\mathbf{C^{ijk}}$ is a 3×3 diagonal matrix with $C_i^{ijk}, C_j^{ijk}, C_k^{ijk}$ in diagonal positions, respectively, and $\Delta\boldsymbol{\theta}^{ijk} = [\Delta\theta_i, \Delta\theta_j, \Delta\theta_k]^T = \mathbf{R}^{ijk}\mathbf{q}^{ijk}$ represents the rotation increment vector with a 3×6 matrix $\mathbf{R}^{ijk}$ and the displacements of i, j, k nodes in horizontal and vertical directions. Finally, the effect of torsional springs in triangle T_{ijk} can be calculated by using discrete forces as

$$\mathbf{F}_{\text{torsional}}^{ijk} = \mathbf{R}^{ijk^T}\mathbf{C}^{ijk}\mathbf{R}^{ijk}\mathbf{q}^{ijk} = \mathbf{K}_{\text{torsional}}^{ijk}\mathbf{q}^{ijk}, \tag{5}$$

where

$$\mathbf{R}^{ijk} = \begin{bmatrix} b_{ik}-b_{ij} & a_{ij}-a_{ik} & b_{ij} & -a_{ij} & -b_{ik} & a_{ik} \\ -b_{ji} & a_{ji} & b_{ji}-b_{jk} & a_{jk}-a_{ji} & b_{jk} & -a_{jk} \\ b_{ki} & -a_{ki} & -b_{kj} & a_{kj} & b_{kj}-b_{ki} & a_{ki}-a_{kj} \end{bmatrix} \tag{6}$$

$a_{st} = (x_t - x_s)/((x_t - x_s)^2 + (y_t - y_s)^2)$, $b_{st} = (y_t - y_s)/((x_t - x_s)^2 + (y_t - y_s)^2)$ $(s \neq t, s,\ t = i, j, k)$.

The above linear system of equations is generally solved by employing traditional iterative methods, such as Jacobi, Gauss-Seidel, SOR, and SSOR. These methods do not require explicit matrix format and work with lower memory cost. However, they generally require a lot of iterations to convergence. The spring-like mesh deformation algorithms produce symmetric torsional stiffness matrices, but the matrices are not always positive definite. Therefore, a fast Krylov subspace based preconditioned BiCGSTAB [43] (P-BiCGSTAB) method is employed in this paper. And a parallel coefficient matrix assembly (PCMA) algorithm is proposed to generate explicit format of the sparse matrix required by P-BiCGSTAB method.

As listed in Algorithm 1, 2DPCMA process fills the CSR (Compressed Sparse Row) format of the spring coefficient matrix which is constructed by a row array *row_arr*, a column array *col_arr*, and a value array *val_arr* through launching many threads concurrently. Each thread calculates two rows of the sparse matrix independently by accumulating the corresponding values in torsional coefficient matrix. Specifically, given an inner node i, each triangle that contains i contributes to $v_{2i,2i}$ (value at $2i^{\text{th}}$ row, $2i^{\text{th}}$ column of the torsional coefficient matrix with nonsparse format), $v_{2i,2i+1}$, $v_{2i+1,2i}$, $v_{2i+1,2i+1}$. This step is accomplished by calculating each torsional coefficient K_t^{ijk} of T_{ijk} and accumulating four values of the top left corner of K_t^{ijk}. To avoid redundant computing, the values in K_t^{ijk} except the four values of the top left corner do not have to be computed because this step does not involve these values. While calculating these values, (5) and (6) provide explicit form of each part of K_t^{ijk} that relates to the vertical and horizontal coordinates of i, j, k. And the four values of K_t^{ijk} used to fill *val_arr* are explicitly expressed as

$$K[0][0] = C_i^{ijk}\left(b_{ik} - b_{ij}\right)^2 + C_j^{ijk}b_{ji}^2 + C_k^{ijk}b_{ki}^2,$$

$$K[0][1] = C_i^{ijk}\left(b_{ik} - b_{ij}\right)\left(a_{ij} - a_{ik}\right) + C_j^{ijk}b_{ij}a_{ji} + C_k^{ijk}b_{ik}a_{ki},$$

Input: (1) Unstructured grid metrics (UGM) including the coordinates of the mesh nodes (**X**, **Y**) and the indices of the nodes that form the elements; adjacent data structure of grid
(2) Row array, *row_arr*, which stores the starting positions of rows.
(3) Inner node identifier i and the GPU thread identifier *tid*.

Output:
(1) Column array, *col_arr*, which stores the positions of the no-zero values in each row.
(2) Value array, *val_arr*, which stores the no-zero values in each row.
(3) Right hand vector of the linear system of equations, **b**.

Procedure: Thread *tid* is responsible for the following tasks
(1) initialize an integer counter: $nn = 0$;
(2) initialize the elements of vector **b** that *tid* is to calculate: $\mathbf{b}[2i] = \mathbf{b}[2i+1] = 0$;
(3) **For** each triangle T_{ijk} that contains i **Do**
(4) Calculate torsional coefficients of $K_t^{ijk}[0:1][0:1] = f(\mathbf{X}_{ijk}, \mathbf{Y}_{ijk})$;
(5) $val\_arr[row\_arr[2i]+s] += K_t^{ijk}[0][s]; (s \in \{0,1\})$;
(6) $val\_arr[row\_arr[2i+1]+s] += K_t^{ijk}[1][s]; (s \in \{0,1\})$;
(7) **End For**
(8) **For** each neighbor j of node i **Do**
(9) **IF** j is a moving boundary node **Then**
(10) **For** each triangle T_{ijm} that contains ij edge **Do**
(11) $\mathbf{b}[2i+r] -= K_t^{ijm}[r][2]p_j^x + K_t^{ijm}[r][3]p_j^y \ (r \in \{0,1\})$;
(12) **End For**
(13) **Else**
(14) $nn++$;
(15) $col\_arr[row\_arr[2i]+2nn] = col\_arr[row\_arr[2i+1]+2nn] = 2j$;
(16) $col\_arr[row\_arr[2i]+2nn+1] = col\_arr[row\_arr[2i+1]+2nn+1] = 2j+1$;
(17) **For** each triangle T_{ijm} that contains ij edge **Do**
(18) $val\_arr[row\_arr[2i+r]+2nn] += K_t^{ijm}[r][2] \ (r \in \{0,1\})$;
(19) $val\_arr[row\_arr[2i+r]+2nn+1] += K_t^{ijm}[r][3] \ (r \in \{0,1\})$;
(20) **End For**
(21) **End IF**
(22) **End For**

Algorithm 1: Two-dimensional parallel coefficient matrix assembly (2DPCMA) algorithm.

$$
\begin{aligned}
K[1][0] &= C_i^{ijk}\left(b_{ik}-b_{ij}\right)\left(a_{ij}-a_{ik}\right) + C_j^{ijk}b_{ij}a_{ji} \\
&\quad + C_k^{ijk}b_{ik}a_{ki}, \\
K[1][1] &= C_i^{ijk}\left(a_{ij}-a_{ik}\right)^2 + C_j^{ijk}a_{ji}^2 + C_k^{ijk}a_{ki}^2.
\end{aligned} \tag{7}
$$

The subsequent lines in Algorithm 1 fill the remaining positions of the $2i^{\text{th}}$ and $2i+1^{\text{th}}$ rows of the torsional coefficient matrix and the nonzero positions of vector **b**. This filling process is accomplished by looping all neighbors of node i. When node i encounters a boundary node j that is tagged as a moving point, then the torsional coefficients of the triangles which contain ij edge contribute to $2i^{\text{th}}$ position of vector **b**; otherwise, they are accumulated to $v_{2i,2j}$, $v_{2i,2j+1}$, $v_{2i+1,2j}$, $v_{2i+1,2j+1}$. To explain this, Figure 2 shows an example of unstructured mesh that contains 7 nodes, 6 triangles, and two boundary nodes. Figures 3 and 4 explain how to fill data for an inner neighbor (d) and a boundary neighbor (c) of a given inner node a in Figure 2. Both T_{adc} and T_{ade} contribute to four nonzero columns regarding the inner node (d); this is accomplished by computing the corresponding element-based torsional matrix of K_t^{adc} and K_t^{ade} and accumulating the values that are associated with edge ad, that is, $K_t^{ad[c,e]}{}_{02}$, $K_t^{ad[c,e]}{}_{03}$, $K_t^{ad[c,e]}{}_{12}$, $K_t^{ad[c,e]}{}_{13}$, to the global CSR (or non-CSR (Figure 4)) format coefficient matrix, respectively. Similarly, two associated local torsional matrices are calculated for a boundary neighbor (c) of (a), but the column number is not recorded in *col_arr* since the movement displacements of node (c) are provided by the motion equation of boundaries at each motion step. Therefore, the known displacements of node (c) (i.e., p_c^x, p_c^y) can be excluded from vector **p** and this part can be transferred to the right hand side of the linear equations by multiplying the corresponding two group values (i.e., $K_t^{ac[b,d]}{}_{02}$, $K_t^{ac[b,d]}{}_{03}$ and $K_t^{ac[b,d]}{}_{12}$, $K_t^{ac[b,d]}{}_{13}$). Same step is performed for the following boundary neighbors of (a) during the neighbor loop and all the transferred values are accumulated to b_{2a} and b_{2a+1}. During the process, the variable of nn is to record the current number of inner nodes of (a) and use the number to make *col_arr* and *val_arr* record the column and the value of nonzero position one by one. It is shown in Algorithm 1 that *col_arr* and *val_arr* are updated synchronously, which indicates that value in *col_arr* and

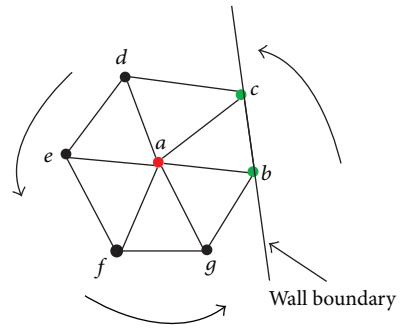

Figure 2: Data arrangement for 2DPCMA algorithm.

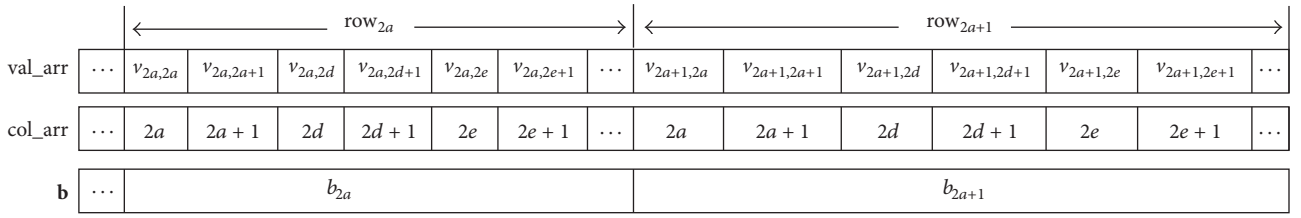

Figure 3: Fill the corresponding segments of *val_arr*, *col_arr*, and *b*, respectively, for the a^{th} inner node.

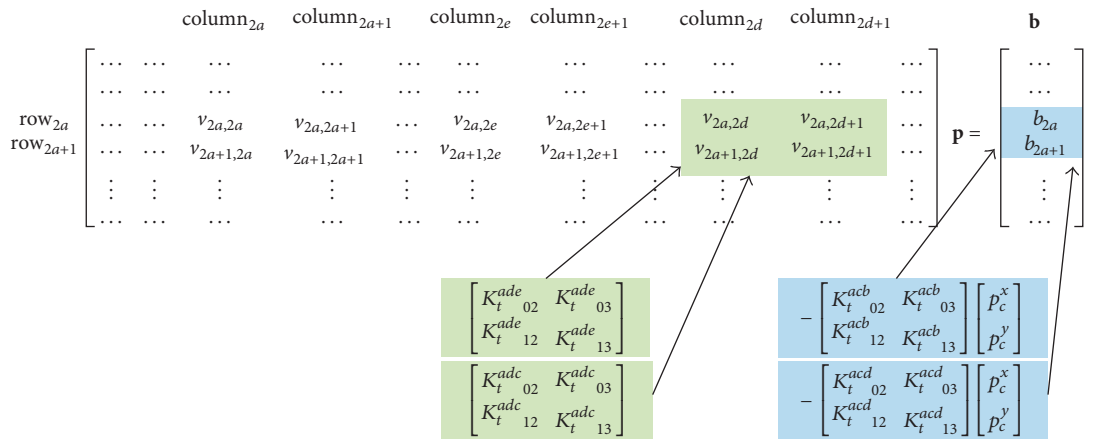

Figure 4: The process of filling data for an inner neighbor (d) and a boundary neighbor (c) of a, respectively.

val_arr is one-to-one mapped and the sequence of neighbor looping has no impact on the content of CSR format matrix.

To implement the 2DPCMA algorithm efficiently, basic grid data structure has to be arranged optimally. Figure 2 shows the anticlockwise neighbor storage strategy in the implementation that is capable of containing all neighboring information implicitly. As depicted in Figure 2, for example, the storing of the neighbors of node (a) starts with node (b) and ends with node (g); then it is easy to visit the previous and subsequent nodes of (d) (c and e, resp.) when the data information of triangles that contain edge ad is required in line (10) of Algorithm 1.

The implementation of 2DPCMA by using OpenACC directives on the GPU accelerator is shown as Listing 1. This implementation consists of two core sections. One part is responsible for arranging memory on GPU via employing *#pragma acc data* clause. The first three lines list two groups of variables with different data attributes. The memories of *val_arr* and *col_arr* variables are created by using *creat* clause while the memories of the remaining variables are both created and initialized with the corresponding values on CPU through *copyin* clause. Another segment of Listing 1 is a function in which the workload can be distributed automatically to independent thread launched through *#pragma acc parallel loop* directive. The data used in this function is allocated and initialized in previous step; thus the *present* keyword is used to indicate that the variables declared in the following brackets are already created on the GPU accelerator. During the *OpenAcc_2DPCMA* procedure, the workload which is related to an inner node i is implicitly distributed to a specific GPU thread *tid* to execute and these two variables are treated as input parameters in 2DPCMA listed in Algorithm 1.

```
//Allocates memory on GPU or copy data from CPU to allocated GPU memory
(1) #pragma acc data create(val_arr[0 : len], col_arr[0 : len])
(2) #pragma acc data copyin(row_arr[0 : rowlen])
(3) #pragma acc data copyin(xcor[0 : nnodes], ycor[0 : nnodes], neighbors[0 : neilen])
Void OpenACC_2DPCMA(len, rowlen, nnodes, niNodes, neilen, val_arr, col_arr,
 xcor, ycor, neighbors)
//Distribute workload to each thread by using "parallel loop" directive
(1) #pragma acc parallel loop present(val_arr[0 : len], col_arr[0 : len], row_arr[0 : rowlen],
                                      xcor[0 : nnodes], ycor[0 : nnodes], neighbors[0 : neilen])
(2)         for(inode = 0; inode < ninNodes; inode + +){
(3)           //each thread fills two rows of the CSR format coefficient matrix
(4)           //The 2i^th row: from row_arr[2i] to row_arr[2i + 1] − 1
(5)           //And the 2i + 1^th row: from row_arr[2i + 1] to row_arr[2i + 2] − 1
(6)           ...
(7)           codes in 2DPCMA algorithm: line (1) to (22)
(8)       }
```

Listing 1: OpenACC implementation for 2DPCMA algorithm on GPU.

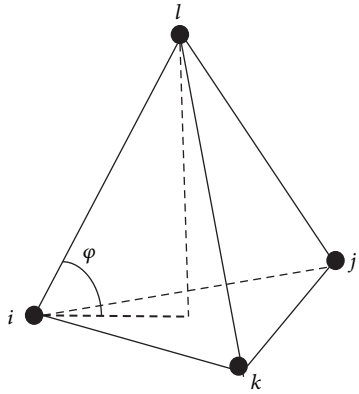

Figure 5: Diagram of 3D torsional spring in a tetrahedron.

4.2. Three-Dimensional Parallel Deforming Mesh Algorithm. Degand and Farhat's work [17] extended the two-dimensional torsional springs deforming mesh algorithm to three-dimensional ones. The main idea of their work is to split the quality constraints of a tetrahedron into the quality constraints of several triangles and then transform each 2D 6×6 stiffness matrix to 3D 9×9 matrix via the coordinate transformation formula. Burg [18] proposed a universal formulation of the torsional spring idea where two-dimensional formulation can be directly extendible to three-dimensional case by analyzing the equations of volume and area for a mesh cell. As the three-dimensional formulation is more straightforward in Clarence's work, the parallel algorithm of this formulation is considered as follows.

As demonstrated in Figure 5, considering face ijk and the opposite point l of an tetrahedron cell T_{ijkl}, the stiffness matrix is expressed as $K_t^{ijk,l} = R_t^{ijk,l} C_t^{ijk,l} R_t^{ijk,l^T}$, where $C_t^{ijk,l} = 1/\sin^2\varphi$ and $R_t^{ijk,l} = [\partial\varphi/\partial\xi]^T$ (1×12 matrix, $\xi = x_u, y_u, z_u$, $u = i, j, k, l$). The complete stiffness matrix with respect to corner i can be obtained by accumulating the stiffness matrix formed from every face containing point i. Thus, all inner points (N_{inner}) can produce $12N_{\text{inner}} \times 12N_{\text{inner}}$ matrix, but there are only $3N_{\text{inner}}$ unknown variables that need to be solved, which indicates that the $12N_{\text{inner}}$ row of the matrix is not linearly independent. In fact, the first three rows of the stiffness matrix with respect to node i and cell T_{ijkl}, denoted as K_t^{i-jkl}, are formed by looping every face that contains i while the remaining rows are the same as the first three rows of $K_t^{j-ikl}, K_t^{k-ijl}, K_t^{l-ijk}$, respectively, when the same cell T_{ijkl} and j, k, l node are considered.

Algorithm 2 lists the pseudocode for 3DPCMA algorithm that is capable of assembling the final stiffness matrix in parallel. Much like 2DPCMA algorithm, the notion of 3DPCMA algorithm is to split the task of filling the CSR format matrix into each subpart of three rows and distribute the subtask to independent GPU thread. Nevertheless, unlike 2DPCMA algorithm, to complete filling the values of the neighbors of node i may require more than one accumulation while the same work is finished in one loop process (lines (17)–(20) in Algorithm 1) in 2DPCMA. This is to increase cache space utilization rate because a lot of variables such as $\mathbf{X}_{ijkl}, \mathbf{Y}_{ijkl}, \mathbf{Z}_{ijkl}$ need to read when an element T_{ijkl} that contains i starts to be visited (line (4) in Algorithm 2) and these variables could be reused to compute as many related data as possible. However, this causes a problem that the position of each neighbor of node i which is already operated could be visited again in the next loop. So extra data structure is required to be constructed to record the relative address of each neighbor with respect to node i for possible revisiting. Moreover, more complicated than 2DPCMA, 3DPCMA procedure requires the adjacent data to provide the information with which three points along with node i make up a tetrahedron because the linear arrangement of the neighbors, unlike the strategy illustrated in Figure 2, can not embrace the configuration of a tetrahedron implicitly.

It should be noted that a temporary matrix-variable K is declared in Algorithm 2 to store the 3×3 submatrix of

Input: (1) Unstructured grid metrics (UGM), including the coordinates of the mesh nodes (**X**, **Y**) and the indices of the nodes that form the elements; adjacent data structure of grid
(2) Row array, *row_arr*, which stores the starting positions of rows.
(3) Inner node identifier i and the GPU thread identifier *tid*.
Output:
(1) Column array, *col_arr*, which stores the positions of the no-zero values in each row.
(2) Value array, *val_arr*, which stores the no-zero values in each row.
(3) Right hand vector of the linear system of equations, **b**.
Procedure: Thread *tid* is responsible for the following tasks
(1) initialize vector **b** that *tid* is to calculate: $\mathbf{b}[3i] = \mathbf{b}[3i+1] = \mathbf{b}[3i+2] = 0$;
(2) fetch the global and local indices of the neighbors of node i: nb_{global}, nb_{local}
(3) fill *col_arr*: $col\_arr[row\_arr[3i+r] + 3nb_{local} + s] = 3nb_{global} + s; (r, s \in \{0,1,2\})$
(4) **For** each tetrahedron $T_{ijkl} \ni i$ **Do**
(5) consider each face that contains node i, face ijk for example:
(6) $K[0:2][0:2] = K_t^{ijk,l}[0:2][0:2] = g(\mathbf{X}_{ijkl}, \mathbf{Y}_{ijkl}, \mathbf{Z}_{ijkl}, V_{ijkl})$
(7) $val\_arr[row\_arr[3i+r]+s] += K[r][s]; (r, s \in \{0,1,2\})$
(8) $K[0:2][0:2] = K_t^{ijk,l}[3:5][3:5]$
(9) **IF** (j is a boundary node) **then**
(10) $\mathbf{b}[3i+r] -= K[r][0]d_j^x + K[r][1]p_j^y + K[r][2]p_j^z; (r \in \{0,1,2\})$
(11) **ELSE**
(12) $val\_arr[row\_arr[3i+r] + 3j_{local} + s] += K[r][s]; (r, s \in \{0,1,2\})$
(13) **End IF**
(14) $K[0:2][0:2] = K_t^{ijk,l}[6:8][6:8]$
(15) similar steps are performed like previous (9)–(13) lines for node k
(16) $K[0:2][0:2] = K_t^{ijk,l}[9:11][9:11]$
(17) similar steps are performed like previous (9)–(13) lines for node l
(18) **End For**

Algorithm 2: Three-dimensional parallel coefficient matrix assembly (3DPCMA) algorithm.

$K_t^{ijk,l}$. In fact, the temporary variable K can be substituted by a single variable to store each value of $K_t^{ijk,l}$ and be used to update the corresponding position of *val_arr* in turn because the value in each position of $K_t^{ijk,l}$ can be explicitly expressed and calculated by $R_t^{ijk,l}$. The implementation that contains less temporary variables is preferable as the registers are costful hardware resource of GPU. However, this low-level consideration, termed as register usage optimization, is less popular but very helpful for efficiency improvement in directive-based OpenACC implementation.

5. Unsteady Solver Parallelized by OpenACC

In this section, the proposed algorithms are integrated into an unstructured, vertex-centered finite volume method based numerical solver to form a full GPU version of an ALE solver that is able to simulate unsteady flows with deforming mesh. By demonstrating Figure 6, the vertex-centered finite volume Ω_p around a node P on two-dimensional triangle meshes, the basic numerical schemes employed in this solver, are structured as follows:

(i) Roe scheme, in which the convective fluxes are computed at a face of the control volume from the left and right state by solving the Rieman problem, is employed for convective flux discretization.

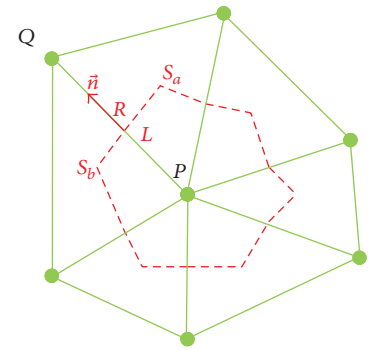

Figure 6: Vertex-centered finite volume.

The mathematical formulation for Roe scheme is expressed as

$$\mathbf{F}_c^{\text{inv,PQ}} = \frac{1}{2}\left[\mathbf{F}_c^{\text{inv}}(\mathbf{W}_R) + \mathbf{F}_c^{\text{inv}}(\mathbf{W}_L) - \left|\mathbf{A}_{\text{Roe}}^{\text{PQ}}\right|(\mathbf{W}_R - \mathbf{W}_L)\right], \tag{8}$$

where the right and left state, $\mathbf{W}_R, \mathbf{W}_L$, are constructed by using *Venkatakrishnan* limiter [44] and the gradients of convective variables.

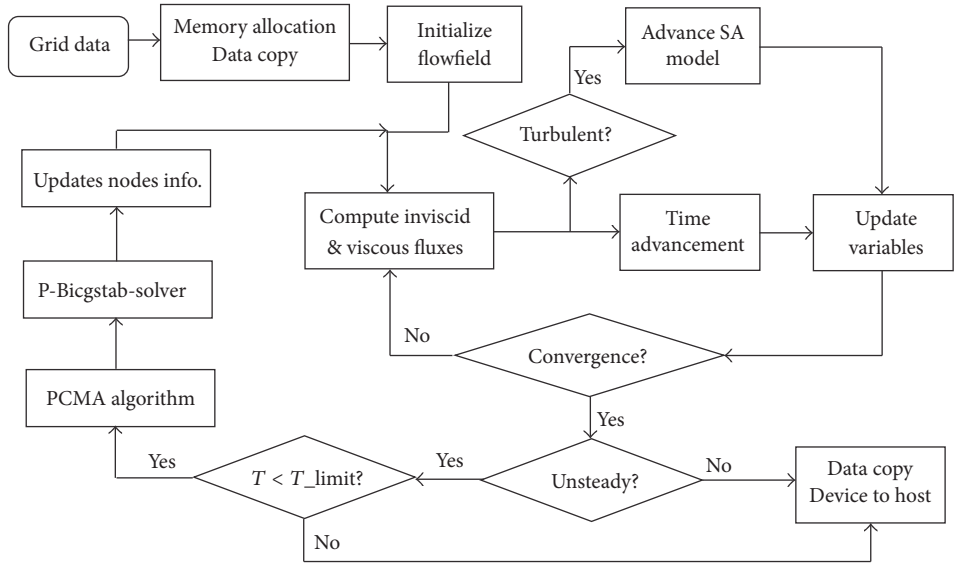

Figure 7: Flowchart of the unsteady GPU solver.

(ii) The central scheme which evaluates the flow quantities at a face of the control volume by averaging the quantities of left and right volumes is applied to discretize both ALE flux $\mathbf{F}_c^{\mathrm{ALE}}$ and viscous flux $\mathbf{F}_v$ in (1).

(iii) The implicit schemes of dual time-stepping based on the second-order time [42] accuracy is employed to discrete (1) in time. By introducing a pseudo-time variable t^* and unsteady residual variable R^*, the dual time-stepping scheme can be formulated as

$$\left[\left(\frac{1}{\triangle t^*}+\frac{3}{2\triangle t}\right)\Omega^{n+1}+\frac{\partial \mathbf{R}}{\partial \mathbf{W}}\right]\triangle \mathbf{W}^* = -\left(R^*\right)^l, \quad (9)$$

where $\mathbf{R}^* = \mathbf{R}(\mathbf{W}^*)+(3/2\triangle t)\Omega^{n+1}-(2/\triangle t)\Omega^n\mathbf{W}^n-(1/2\triangle t)\Omega^{n-1}\mathbf{W}^{n-1}$.

(iv) The Spalart-Allmaras (SA) one-equation turbulence model [45] is adopted. It can be written in integral form as

$$\frac{\partial}{\partial t}\int_\Omega \widetilde{\nu} d\Omega + \oint_{\partial\Omega}\left(F_{c,T}-F_{v,T}\right)dS = \int_\Omega Q_T d\Omega. \quad (10)$$

The convective flux and viscous flux are given by $F_{c,T} = \widetilde{\nu}V$ and $F_{v,T} = \vec{N}\cdot\vec{\tau}$ with V being the contravariant velocity, $\vec{N}$ being the outward facing unit normal vector of the control surface, and $\vec{\tau}$ being the normal viscous stresses. The source term Q_T is introduced in [45]. The first-order upwind scheme and central scheme are adopted to discretize the convective and viscous flux and the implicit time scheme [42] is employed for advancing SA equation in time.

The detailed steps of the solver are demonstrated as Figure 7 in which the procedures of *PCMA algorithm*, *P-Bicgstab-solver*, *updates nodes info.* are to be executed between each two physical steps. Listing 2 gives an overview of the OpenACC implementation of the unsteady solver. It starts with *alloc_mem_init* subroutine to allocate and initialize variables on the host. Then the variables which are used as intermediate variables for computation are explicitly created on GPU memory by using *#pragma acc data create* clause. And the variables which are created and initialized by *alloc_mem_init* are copied from CPU to GPU by using *copyin* keyword. The previous two steps of data arrangement operation are followed by a pair of curly braces where the main part of GPU computation is placed. This data transfer consideration avoids extra data exchange between the host and the device during the whole computational process. By considering (1), (10), GCL law (3), and mesh deformation equations, there are $8N$ unknown variables (ρ, u, v, w, E, p, T) to be solved in (1) and (10) at each pseudo-time step and $4N$ unknown variables (GCL volume, movements of each node in x, y, z directions) between each two physical time steps, where N denotes the number of mesh nodes.

In the main computation section, each step is encapsulated as a subroutine that accepts necessary parameters. These functions have no difference with the serial implementation. At this point in the OpenACC implementation, it requires no effort except two directive-based data declarations out of the curly braces. Even in each function, the modification of the CPU code is concentrated only in *for* loop regions and data segments where read-write conflicts exist. Take *Set_Time_Step* function, for example (shown in Listing 2), *#pragma acc parallel loop* clause, is declared to make the *for* loop parallel and this clause is followed by *present* keyword to declare that the global arrays used in the following *for* loop are already present on GPU memory. The variables which are not mentioned in the data statement are treated as private variables during the loop. Edge-based data structure for parallel implementation will encounter read-write conflict inevitably because the quantities of a common node shared

```
//Allocates memory on CPU and initialization
(1) alloc_mem_init(data_pointer1,data_pointer2, ...);
(2) #pragma acc data create(gvar_arr_1, gvar_arr_2, ...)
(3) #pragma acc data copyin(cgvar_arr_1, cgvar_arr_2, ....)
(4) {
(5) ...
(6) Set_Time_Step(nNds,nEgs,lambda_inv,lambda_visc, ...);
(7) Compute_Grident_Limiter(nNds,nEgs,edgeLn,edgeRn,den,denU,denV,denW,denE, ...);
(8) Upwind_Residual(...);Viscous_Residual(...); Time_Advance(...);
(9) }
(10) #pragma acc data update(cgvar_arr_1,cgvar_arr_2, ...)
// example of one function
void Set_Time_Step(nNds,nEgs,lambda_inv,lambda_visc, ...) {
(1) #pragma acc parallel loop present(edgeLn[0 : nEdges], edgeRn[0 : nEdges], ... lambda_inv[0 : nNodes])
(2)         for (iEdge = 0; iEdge < nEdges; iEdge + +){
(3)           iPoint=edgeLn[iEdge];
(4)           jPoint=edgeRn[iEdge];
(5)           ...
(6)           #pragma acc atomic
(7)           lambda_inv[iPoint]+=lambda;
(8)           #pargma acc atomic
(9)           lambda_inv[jPoint]+=lambda;
(10)           ...
(11)        }
```

Listing 2: OpenACC implementation of the unsteady solver.

Table 1: Configuration parameters of K20 GPU.

Compute capability	3.5	Stream multiprocessor	14
Architecture	Kepler	Processors	2496
Processor clock	706 MHZ	Registers/SM	65536
Memory clock	2.6 G	Max registers/thread	255
Memory size	320-bit GDDR5 5 GB	Share memory config	16 K/32 K/48 K

by two edges may be updated simultaneously in two different GPU threads. This problem can be solved by employing atomic operation in OpenACC.

6. Experiments

The following test cases were conducted on a heterogeneous platform YUAN at supercomputing center of Chinese Academy of Sciences. YUAN consists of 270 blades nodes, 30 GPU nodes, and 40 MIC (Intel Many Integrated Core Architecture) nodes. Each GPU node has two Intel E5-2658 V2 (Ivy Bridge, 2.8 GHz) CPUs and two Nvidia Tesla K20 GPU cards. The configuration parameters of K20 GPU are list in Table 1.

6.1. Performance of Parallel Mesh Deformation. In this part, both triangular and tetrahedral meshes with variety of computational scales are available to capture the performance of the parallel deforming mesh algorithm on the GPU device. The evaluation of performance is speedup which is defined as the ratio of wall clock times running on Intel E5-2658 V2 CPUs and K20 Card. In order to make fair comparison, the serial version of deforming mesh code is compiled by using GCC compiler with highest optimization option -O3.

Table 2: Four mesh configurations for test.

Mesh number	Type	Points	Elements
1	Triangular	21329	41636
2	Triangular	288268	574232
3	Tetrahedral	108396	582752
4	Tetrahedral	275561	1453849

Table 2 lists two 2D and two 3D mesh configurations performed in this test. Table 3 gives the CPU times, GPU times for each part of mesh deforming procedure, and the speedup performances. The fourth column in Table 3 shows the speedup for matrix assembly running on K20 GPU card against Intel E5-2658 CPUs. It is shown that higher speedup can be achieved on the first two 2D meshes compared to that on the following two 3D meshes. This can be explained by the fact that the workload each GPU thread undertakes varies owing to neighbor sweeping procedure in Algorithms

Table 3: CPU times, GPU times, and speedups on four test meshes.

Mesh	CTMA(s)	GTMA(s)	SPMA	CTB(s)	GTB(s)	SPB	OSP
1	0.0379	0.0021	18.04x	0.2476	0.0304	8.14x	8.78x
2	0.4902	0.0219	22.38x	3.6404	0.3023	12.04x	12.74x
3	0.3805	0.0244	15.59x	2.5815	0.2392	10.79x	11.23x
4	0.8923	0.0537	16.62x	5.1088	0.3582	14.26x	14.56x

CTMA: CPU time of matrix assembly; GTMA: GPU time of matrix assembly; SPMA: speedup for matrix assembly; CTB: CPU time of p-bicgstab solver; GTB: GPU time of p-bicgstab solver; SPB: speedup for p-bicgstab solver; OSP: overall speedup.

1 and 2 and a 3D tetrahedral node has a larger disparity in the number of neighbors than that of a 2D triangle node, which leads to more load imbalance of workload on GPU threads for 3D meshes. The fourth column data also indicates that larger mesh size results in higher speedup for the same mesh type. This is to be expected, because larger mesh size is more conductive to make best use of the GPU resources and this trend can be sustained before the saturation of the available GPU resources is achieved.

The fifth and sixth column of Table 3 list the CPU time and GPU time of the sparse linear system of equations obtained by PCMA algorithm, respectively. Six iterations are performed in the P-BiCGSTAB procedure and all of the four cases attain an accuracy of 1.0×10^{-8} within these 6 iterations. In order to make the GPU implementation of P-BiCGSTAB solver more efficient, cuBLAS and cuSPARSE [46], the optimized, parallel GPU-accelerated libraries that provide efficient matrix-vector, sparse matrix-vector, and vector-vector multiplications, are introduced. However, the API functions in cuBLAS or cuSPARSE require explicit data pointers in the GPU memory while the general OpenACC model uses unified data pointers and switches on CPU and GPU platform implicitly. Fortunately, OpenACC model provides a way to invoke the third-party CUDA library via OpenACC directive declaration with *#pragma acc host_data use_device (pointerA, pointerB)*, which returns the corresponding device pointers for the following CUDA-based API calls. *cusparseDcsrilu0()* was used to compute the incomplete-LU factorization as the preconditioner and *cusparseDcsrmv()* was called to conduct SpMV during the process of P-BiCGSTAB. The last column lists the overall speedups by combining the wall clock times of the previous two subprocedures. And about 8.78x~14.56x speedup can be obtained for both 2D and 3D meshes.

Figures 8 and 9 show the speedup trend of parallel deforming mesh algorithm for 2D mesh and 3D mesh, respectively, with the size of mesh nodes increases. Nine meshes whose numbers of mesh nodes were within $[2.7 \times 10^5, 56.7 \times 10^5]$ interval were generated by *Triangle* [47] for 2D test. And seven meshes whose numbers of mesh nodes were within interval $[3.2 \times 10^5, 41.2 \times 10^5]$ were generated by *TetGen* [48] for 3D PDMA test. An average of 10 iterations were performed for P-BiCGSTAB process. The results indicate that larger mesh size produces higher speedup. That is expected, because larger meshes make better use of the available GPU resources, such as GPU memory, stream processors, and registers. However, the increment of speedup is not proportional to the increment of mesh size as the mesh size increases; this can be explained by the fact

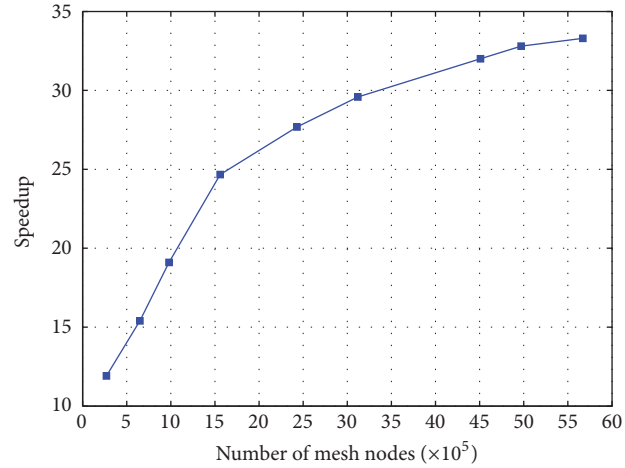

Figure 8: 2D speedup trend for PDMA corresponds to mesh size.

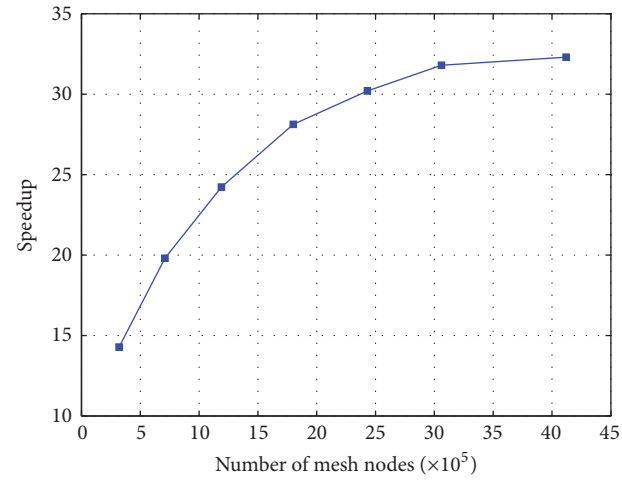

Figure 9: 3D speedup trend for PDMA corresponds to mesh size.

that the available resources of GPU are gradually saturated as the processing data of the mesh increases. Figures 8 and 9 demonstrate that the speedups of both 2D and 3D case become stable at around 30x.

The performance of the present algorithms was also investigated by OpenMP directives which is a popular shared-memory programming mode. The 2D test mesh was configured by 5672268 nodes and 11114646 triangles, and the 3D test mesh consisted of 4127686 nodes and 24023132 tetrahedrons. As the test node has two Intel E5-2658 V2 CPUs

Table 4: Performance comparisons of multicore CPU and GPU results on mesh deformation algorithm.

2D case		Single CPU core	GPU	20 OpenMP threads
5672268 nodes	Matrix assembly	9.8 s	0.23 s	0.7 s
11117646 triangles	P-BiCGSTAB	52 s	1.63 s	4.19 s
3D case		Single CPU core	GPU	20 OpenMP threads
4127686 nodes	Matrix assembly	12.6 s	0.33 s	0.78 s
24023132 tetrahedrons	P-BiCGSTAB	58.1 s	1.84 s	4.83 s

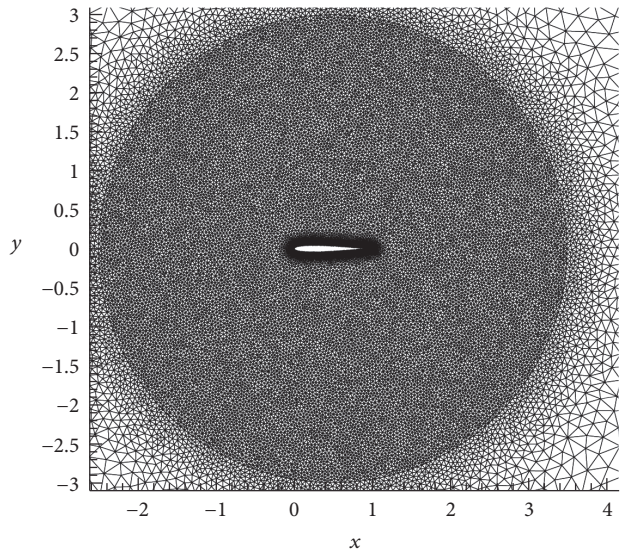

Figure 10: Unstructured grids around NACA0012 airfoil.

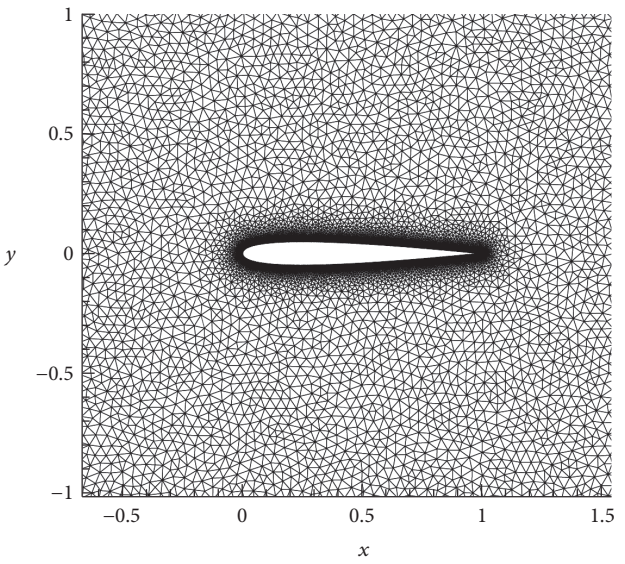

Figure 11: Close view of the unstructured grids around NACA0012 airfoil.

with each having 10 cores, total 20 threads can be launched. Table 4 shows the comparisons of multicore CPU and GPU results on the two test cases for mesh deformation algorithm. It is shown in Table 4 that a speedup of 2.6x can be obtained on K20 GPU in comparison with 20 OpenMP threads for both of 2D and 3D cases, which demonstrates the computing power of GPU through light-weighted threads.

6.2. Simulation Cases. The first simulating case used by the presented solver is about the unsteady Euler solutions induced by the NACA0012 airfoil. The varying of the angle of attack by pitching harmonically about the quarter chord with respect to physical time is given as

$$\alpha = \alpha_0 + \alpha_m \sin(kt), \tag{11}$$

where $\alpha_0 = 0.016^\circ$, $\alpha_m = 2.51^\circ$ are the initial and amplitude of angle of attack, respectively, $k = 0.0814$ denotes the reduced frequency based on semichord, and t represents the physical time. The initial Mach number is $M = 0.755$.

Figures 10 and 11 show the unstructured mesh that contains 94085 nodes and 186306 triangles in a far and close view, respectively. Steady state was first calculated at $\alpha_0 = 0.016^\circ$ and the obtained flow is used as the initial flow field for the following unsteady simulation. Figure 12 shows the Mach contours around the NACA0012 airfoil at the steady state. A physical time step of $\triangle t = 0.2$ was employed for the unsteady simulation. In the pseudo-time process, a

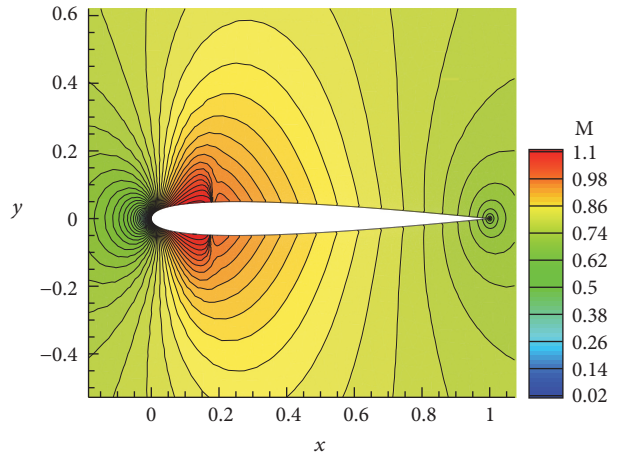

Figure 12: Mach contours around the NACA0012 airfoil at $\alpha = 0.016^\circ$.

maximum number of iterations of $Max\_Iter = 5$ and a minimum error of $Solver\_Error = 1.0 \times 10^{-6}$ are configured for the linear solver of the implicit time formulation. The parallel deforming mesh algorithm was conducted between every two adjacent physical time steps for assembling and solving a sparse matrix linear system of equations with a scale of 182018×182018.

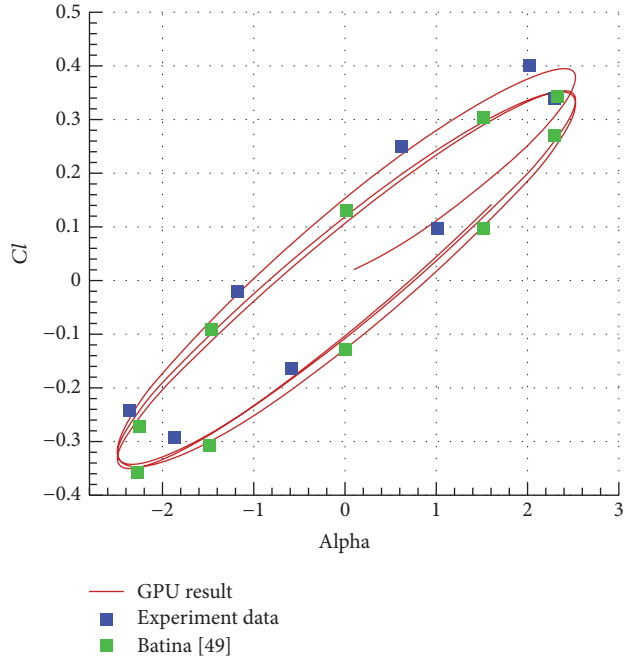

Figure 13: Lift coefficient corresponds to instantaneous angle of attack.

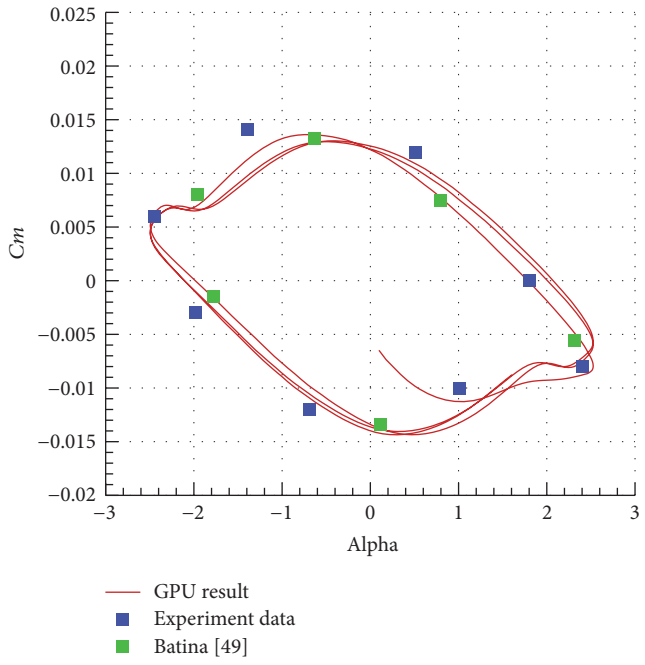

Figure 14: Moment coefficient corresponds to instantaneous angle of attack.

To obtain stable unsteady solutions, three cycles of periodic motion were computed and comparisons were conducted between the GPU results, experiment data, and Batina's results stated in [49], which are shown in Figures 13 and 14. It is noted that the calculated results show a slight deviation when comparing with experiment data but agree well with Batina's work. The validation through comparing

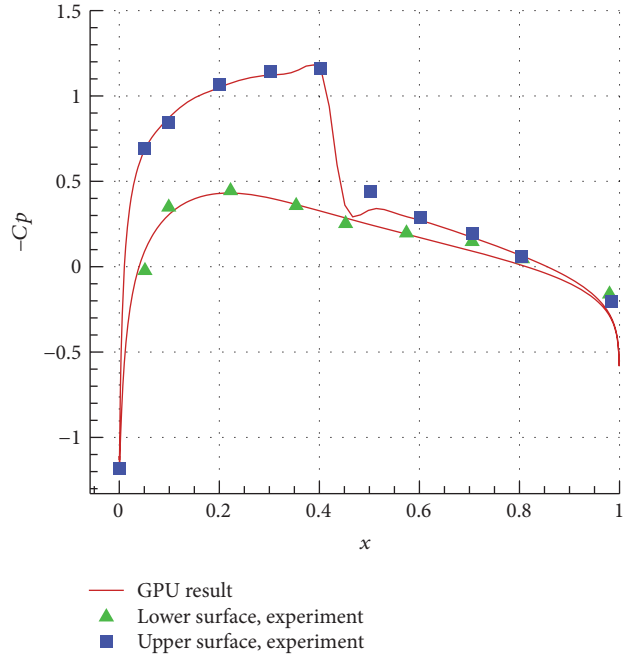

Figure 15: Instantaneous pressure coefficient comparisons at $kt = 69^\circ$ ($\alpha = 2.34^\circ$).

Table 5: Wall clock time of each procedure for the NACA0012 case.

Procedure name	CPU time (s)	GPU time (s)	Speedup
Gradient_Limiter	0.2728	0.0261	10.45x
Set_Time_Step	0.0570	0.0041	13.90x
Upwind_Residual	0.4320	0.0301	14.35x
Boundary_Condition	0.0033	0.0008	4.125x
Implicit_Time_Advance	1.2436	0.1099	11.31x

instantaneous pressure coefficient with experiment data has also been conducted at a specified angle of attack, which is shown in Figure 15 for $\alpha = 2.34^\circ$ (*i.e.* $kt = 69^\circ$).

Table 5 lists the wall clock times executing on CPU and GPU for five primary procedures of the unsteady solver. It is shown that solving linear system equations, inviscid residual computing, gradients, and limiter calculations are the most time-consuming part in the solver. In *Implicit_Time_Advance* procedure, 3 iterations in BiCGSTAB scheme make the iteration convergence to the given *solver_Error*. And All three parts can be obtained over 10x speedup on GPU with respect to Intel E5-2658 V2 CPU core. According to statistics, advancing one pseudo-step within a physical step requires 1.8876 seconds on CPU and 0.1620 seconds on GPU for this case, and conducting a deforming mesh step requires 1.2886 on CPU and 0.1208 on GPU, in which the numerical solver and mesh deforming parts are both accelerated.

The second example for validating the unsteady computing with deforming mesh on GPU involves the transonic simulation induced by the ONERA M6 wing. The initial conditions are configured as Mach = 0.84, Re = 21.66×10^6, the motion of wing on different spans is expressed as $\alpha(y, t) = \alpha_0 + \alpha_m(y/L)\sin(kt)$, where $\alpha_0 = 0^\circ$ and $\alpha_m = 10^\circ$ denote the

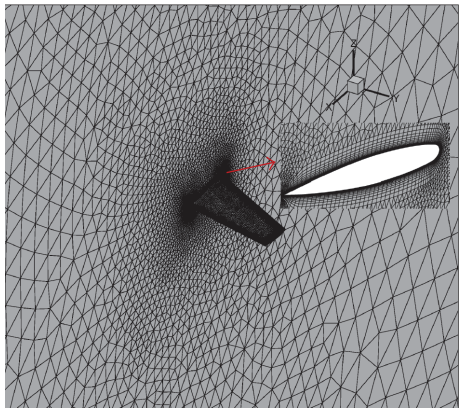

Figure 16: View of the hybrid mesh around ONERA M6 wing.

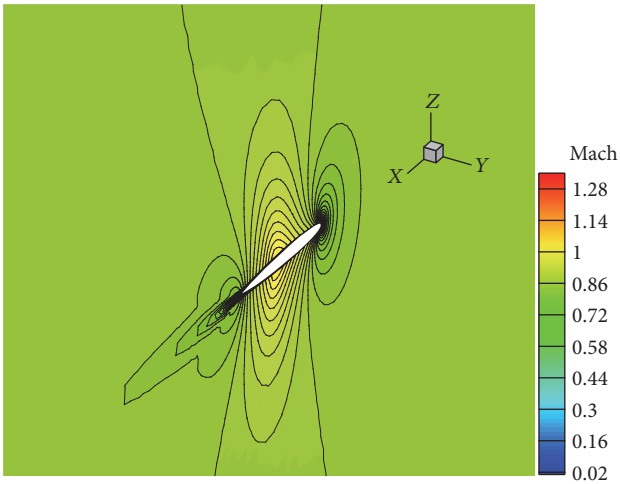

Figure 17: Steady state Mach contours at the symmetrical plane.

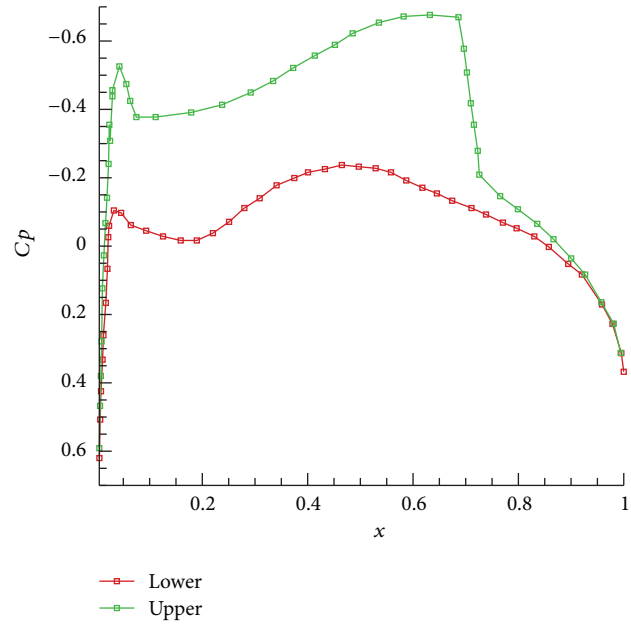

Figure 18: One-quarter of last cycle: Instantaneous pressure coefficient distributing on 20% wing span along y direction.

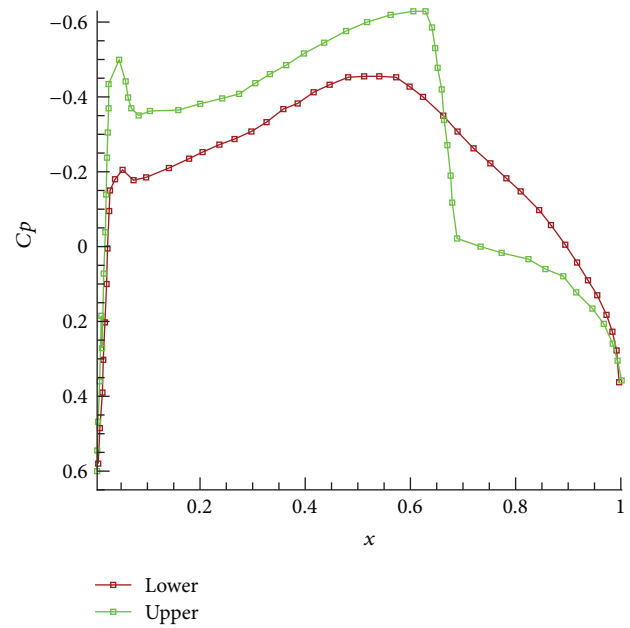

Figure 19: Half of last cycle: Instantaneous pressure coefficient distributing on 20% wing span along y direction.

average and the amplitude of angle of attack for this periodic movement, L is the wing span and y represents the specified station along the span direction, and $k = 0.27$ is the reduced frequency based on mean chord.

The hybrid mesh used for this case contains 450634 nodes, 961146 tetrahedrons, and 548688 prisms. Figure 16 shows the hybrid mesh at the symmetrical plane. Steady calculations were performed at $\alpha_0 = 0^\circ$ at first to provide initial solution of the flow for subsequent unsteady simulations. And the steady state Mach contours at the symmetrical plane are shown in Figure 17. In order to deform the hybrid mesh, Liu's method [20] was employed by providing a coarse tetrahedral mesh as a background grid. The reason why tetrahedral mesh is considered as the background mesh is that it is more robust in the treatment of complex geometries than that of Delaunay graph mesh. However, it requires more CPU time to finish mesh deforming at each step. The present parallel algorithm for deforming tetrahedral mesh was first applied in Liu's scheme and then the interpolation process [20] was conducted to deform the foreground mesh. In this case, the background mesh provided contains 108396 nodes and 582752 tetrahedrons. $\triangle t = 0.12$ was selected for the global physical time step and 5 periods of simulation were computed to lead to stable periodic results. Figures 18 and 19 demonstrate the instantaneous pressure distributing on 20% wing span along y direction in a quarter and half of the last unsteady cycle, respectively.

It took about 0.31 seconds to deform the auxiliary tetrahedral mesh in the background and 0.05 seconds to interpolate the movement of the background mesh into the computational mesh through DGM interpolation process

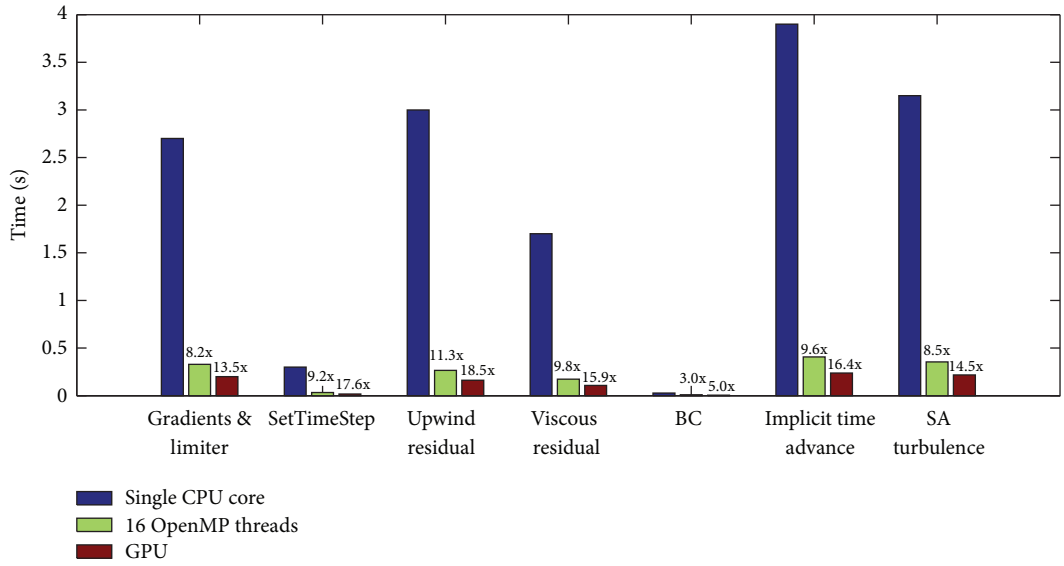

Figure 20: CPU and GPU time comparisons for each part of the solver.

[20]. And 0.19 seconds was required to update the quantities that are associated with the computational mesh on K20 GPU card. Figure 20 gives the CPU and GPU time comparisons for each part of the solver. It is shown in Figure 20 that the calculation of gradients and limiters, the computing of upwind residual, and the advancement of one-step flow solution are the three most time-consuming parts in advancing ALE (1) equations. However, the process of implicit time stepping, which is the most time-consuming procedure, does not achieve the highest speedup among the three procedures. That is mainly because an unsymmetrical linear system of equations which contains a lot of memory intensive operations such as SpMV (Sparse Matrix Multiply Vector), cross product of vectors, is required to be solved and accelerating the operations of this type on GPU is generally not as easy as that of compute intensive type. Another factor that affects the parallel efficiency of *implicit time advance* is that the GPU resources are not saturated due to relatively small-scale of the equations and there is not enough computational workload to switch during memory access. Memory intensive operations also account for the relatively low speedup obtained by 16 OpenMP threads on the procedure of *implicit time advance*. Another reason is that the OpenMP directives for solving equations were implemented within the outer loop and the frequent creating, switching, and destroying of OpenMP threads increased the extra overhead.

Overall, higher speedup was obtained on GPU platform in comparison with multicore CPU environment. This result also demonstrates that GPU has great abilities in accelerating fine-grained computational task by utilizing its light weighed threads. These threads can be launched and switched with extremely low overhead. Two procedures, involving gradient and limiter computation, achieved the lowest speedup on GPU; one of the reasons is that the two procedures contain more atomic addition (or subtraction) operations than *SetTimeStep* and *SA Turbulence* procedures. Actually, the procedure of *upwind residual* involves the maximum number of atomic operations among the procedures and these atomic operations had much impact on its acceleration. The atomic operations were eliminated by storing edge related values into global memory individually first and accumulating these values to nodes via looping the surrounding edges of these nodes. However, this strategy was not applied on the other procedures that contain atomic operations because it requires extra code revision effort and much more memory usage. The time spent on atomic operations accounts for nearly 15% of the total wall clock time of this case, which benefits from the optimized atomic operations of the K-series GPU card. And this reduces the effort on code revision and makes parallel code developing more efficient. Overall, it took about 14.8 seconds on a single CPU core, 0.9457 seconds on K20 GPU card, and 1.56955 seconds on 16 CPU cores for advancing one pseudo-step in time. About 15.6x speedup can be achieved on K20 GPU card relative to a single Intel E5-2568 V2 CPU core and about 1.66x speedup relative to 16 OpenMP threads.

7. Summary and Future Work

The parallel deforming mesh algorithm is present and implemented by using directive-based programming model-OpenACC. And the proposed algorithms are integrated into an OpenACC-parallelized unstructured ALE solver which can conduct unsteady simulation with deforming mesh completely on GPU. The OpenACC implementation of the unsteady solver requires minor revision of the original serial codes, which makes code migration easier and efficient. The effort is underway to extend the present unsteady solver to CPU + GPU heterogeneous computing platform by considering combining MPI and OpenACC and the new GPU-Direct communication feature will be used to reduce overhead while transferring data from one GPU to a remote GPU.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by grants from National Natural Science Foundation of China (nos. 11502267 and 61501393). This work is also supported by the key research project of institutions of higher education of Henan province (no. 17B520034).

References

[1] P. Le Tallec and J. Mouro, "Fluid structure interaction with large structural displacements," *Computer Methods in Applied Mechanics and Engineering*, vol. 190, no. 24-25, pp. 3039–3067, 2001.

[2] E. H. van Brummelen, K. G. van der Zee, and R. de Borst, "Space/time multigrid for a fluid–structure-interaction problem," *Applied Numerical Mathematics. An IMACS Journal*, vol. 58, no. 12, pp. 1951–1971, 2008.

[3] M. Hussain, M. Abid, M. Ahmad, A. Khokhar, and A. Masud, "A parallel implementation of ALE moving mesh technique for FSI problems using OpenMP," *International Journal of Parallel Programming*, vol. 39, no. 6, pp. 717–745, 2011.

[4] N. C. Prewitt, D. M. Belk, and W. Shyy, "Parallel computing of overset grids for aerodynamic problems with moving objects," *Progress in Aerospace Sciences*, vol. 36, no. 2, pp. 117–172, 2000.

[5] A. Naderi, M. Darbandi, and M. Taeibi-Rahni, "Developing a unified FVE-ALE approach to solve unsteady fluid flow with moving boundaries," *International Journal for Numerical Methods in Fluids*, vol. 63, no. 1, pp. 40–68, 2010.

[6] T. C. Rendall and C. B. Allen, "Parallel efficient mesh motion using radial basis functions with application to multi-bladed rotors," *International Journal for Numerical Methods in Engineering*, vol. 81, no. 1, pp. 89–105, 2010.

[7] B. Roget and J. Sitaraman, "Robust and efficient overset grid assembly for partitioned unstructured meshes," *Journal of Computational Physics*, vol. 260, pp. 1–24, 2014.

[8] M. Jung and O. Kwon, "Numerical simulation of unsteady rotor flow using an unstructured overset mesh flow solver," *International Journal of Aeronautical and Space Sciences*, vol. 10, no. 1, pp. 104–111, 2009.

[9] G. Zagaris, M. T. Campbell, D. J. Bodony, E. Shaffer, and M. D. Brandyberry, "A toolkit for parallel overset grid assembly targeting large-scale moving body aerodynamic simulations," in *Proceedings of the 19th International Meshing Roundtable, IMR 2010*, pp. 385–401, usa, October 2010.

[10] W. Liao, J. Cai, and H. M. Tsai, "A multigrid overset grid flow solver with implicit hole cutting method," *Computer Methods in Applied Mechanics and Engineering*, vol. 196, no. 9-12, pp. 1701–1715, 2007.

[11] J. Cai, H. M. Tsai, and F. Liu, "A parallel viscous flow solver on multi-block overset grids," *Computers & Fluids*, vol. 35, no. 10, pp. 1290–1301, 2006.

[12] B. Landmann and M. Montagnac, "A highly automated parallel Chimera method for overset grids based on the implicit hole cutting technique," *International Journal for Numerical Methods in Fluids*, vol. 66, no. 6, pp. 778–804, 2011.

[13] S. E. Rogers, N. E. Suhs, and W. E. Dietz, "PEGASUS 5: An automated preprocessor for overset-grid computational fluid dynamics," *Journal of American Institute of Aeronautics and Astronautics*, vol. 41, no. 6, pp. 1037–1045, 2003.

[14] F. J. Blom, "Considerations on the spring analogy," *International Journal for Numerical Methods in Fluids*, vol. 32, no. 6, pp. 647–668, 2000.

[15] C. Farhat, C. Degand, B. Koobus, and M. Lesoinne, "Torsional springs for two-dimensional dynamic unstructured fluid meshes," *Computer Methods in Applied Mechanics and Engineering*, vol. 163, no. 1-4, pp. 231–245, 1998.

[16] C. L. Bottasso, D. Detomi, and R. Serra, "The ball-vertex method: a new simple spring analogy method for unstructured dynamic meshes," *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 39-41, pp. 4244–4246, 2005.

[17] C. Degand and C. Farhat, "A three-dimensional torsional spring analogy method for unstructured dynamic meshes," *Computers and Structures*, vol. 80, no. 3-4, pp. 305–316, 2002.

[18] Burg C., "A robust unstructured grid movement strategy using three-dimensional torsional springs," *Journal of American Institute of Aeronautics and Astronautics*, 2004.

[19] A. de Boer, M. S. van der Schoot, and H. Bijl, "Mesh deformation based on radial basis function interpolation," *Computers and Structures*, vol. 85, no. 11-14, pp. 784–795, 2007.

[20] X. Liu, N. Qin, and H. Xia, "Fast dynamic grid deformation based on Delaunay graph mapping," *Journal of Computational Physics*, vol. 211, no. 2, pp. 405–423, 2006.

[21] "A High Performance Message Passing Library," http://www.open-mpi.org.

[22] "The OpenMP API Specification For Parallel Programming," http://openmp.org.

[23] CUDA., http://www.nvidia.com/object/cuda_home_new.html.

[24] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin, "Running unstructured grid-based CFD solvers on modern graphics hardware," *International Journal for Numerical Methods in Fluids*, vol. 66, no. 2, pp. 221–229, 2011.

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

[26] K. Soni, D. D. Chandar, and J. Sitaraman, "Development of an overset grid computational fluid dynamics solver on graphical processing units," *Computers & Fluids*, vol. 58, pp. 1–14, 2012.

[27] S. Gohari M I, V. Esfahanian, and H. Moqtaderi, "Coalesced computations of the incompressible Navier—Stokes equations over an airfoil using graphics processing units," *Computers & Fluids*, vol. 80, no. 1, pp. 102–115, 2013.

[28] L. Luo, "GPU port of a parallel incompressible Navier-Stokes solver based on OpenACC and MVAPICH2," in *Proceedings of the 7th AIAA Theoretical Fluid Mechanics Conference*, Atlanta, GA, USA, 2014.

[29] Y. Xia, H. Luo, M. Frisbey, and R. Nourgaliev, "A set of parallel, implicit methods for a reconstructed discontinuous Galerkin method for compressible flows on 3D hybrid grids," in *Proceedings of the 7th AIAA Theoretical Fluid Mechanics Conference*, Atlanta, Ga, USA, 2014.

[30] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith et al., "Accelerating Hydrocodes with OpenACC, OpeCL and CUDA," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, pp. 465–471, IEEE Computer Society, 2012.

[31] D. D. J. Chandar, J. Sitaraman, and D. Mavriplis, "GPU parallelization of an unstructured overset grid incompressible Navier-Stokes solver for moving bodies," in *Proceedings of the 50th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2012.

[32] V. G. Asouti, X. S. Trompoukis, I. C. Kampolis, and K. C. Giannakoglou, "Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on Graphics Processing Units," *International Journal for Numerical Methods in Fluids*, vol. 67, no. 2, pp. 232–246, 2011.

[33] D. Chandar, J. Sitaraman, and D. Mavriplis, "Dynamic Overset Grid Computations for CFD Applications on Graphics Processing Units," in *Proceedings of the Seventh International Conference on Computational Fluid Dynamics*, Big Island, Hawaii, 2012.

[34] A. Munshi, B. Gaster, T. Mattson G et al., "Programming Guid," 2011.

[35] K. Gregory and A. Miller, *C++ AMP: Accelereated Massive Parallelism with Microsoft Visual C++*, 2012.

[36] "The OpenACC Standard," http://www.openacc-standard.or.

[37] CAPS, "The fastest way to many core programming," 2012, http://www.caps-entreprise.com.

[38] "OpenACC for HPC Accelerator Programming," http://www.cray.com/blog/openacc-for-hpc-accelerator-programming/.

[39] B. Lebacki, M. Wolfe, and D. Miles, *Fortran and C99 OpenACC Compliers*, 2012.

[40] https://developer.nvidia.com/openacc.

[41] OpenACC Programming and Best Practices Guide, http://www.openacc.org.

[42] J. Blazek, *Computational Fluid Dynamics: Principles and Applications*, vol. 55 of *Computational Fluid Dynamics Principles & Applications*, Chapter 6, 1, Second Edition edition, 2001, 1C4.

[43] H. A. van der Vorst, "BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.

[44] V. Venkatakrishnan, "Convergence to steady state solutions of the Euler equations on unstructured grids with limiters," *Journal of Computational Physics*, vol. 118, no. 1, pp. 120–130, 1995.

[45] P. Spalart and S. Allmaras, "A one-equation turbulence model for aerodynamic flows," *La Recherche Aerospatiale*, no. 1, pp. 5–21, 1994.

[46] CUDA Toolkit Documentation, http://docs.nvidia.com/cuda/.

[47] R. S. Jonatha and A. Triangle, *A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*, http://www.cs.cmu.edu/~quake/triangle.html.

[48] H. Si, "TetGen, a Delaunay-based quality tetrahedral mesh generator," *Association for Computing Machinery. Transactions on Mathematical Software*, vol. 41, no. 2, Art. 11, 36 pages, 2015.

[49] J. T. Batina, "Unsteady Euler airfoil solutions using unstructured dynamic meshes," *AIAA Journal*, vol. 28, no. 8, pp. 1381–1388, 1990.