A GPU-accelerated implicit meshless method for compressible flows

Jia-Le Zhang[a], Zhi-Hua Ma[b], Hong-Quan Chen[a,*], Cheng Cao[a]

[a] Department of Aerodynamics, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China.

[b] School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University, Manchester M1 5GD, UK.

[*] **Correspondence information:**

Corresponding author name: Hong-Quan Chen

Post address: Department of Aerodynamics
Nanjing University of Aeronautics and Astronautics
29 Yudao Street, Nanjing 210016, China

Email: hqchenam@nuaa.edu.cn

Tel: +86 25 84895919

A GPU-accelerated implicit meshless method for compressible flows

Jia-Le Zhang[a], Zhi-Hua Ma[b], Hong-Quan Chen[a,*], Cheng Cao[a]

[a] Department of Aerodynamics, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China

[b] School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University, Manchester M1 5GD, UK

Abstract

This paper develops a recently proposed GPU based two-dimensional explicit meshless method (Ma et al., 2014) by devising and implementing an efficient parallel LU-SGS implicit algorithm to further improve the computational efficiency. The capability of the original 2D meshless code is extended to deal with 3D complex compressible flow problems. To resolve the inherent data dependency of the standard LU-SGS method, which causes thread-racing conditions destabilizing numerical computation, a generic rainbow coloring method is presented and applied to organize the computational points into different groups by painting neighboring points with different colors. The original LU-SGS method is modified and parallelized accordingly to perform calculations in a color-by-color manner. The CUDA Fortran programming model is employed to develop the key kernel functions to apply boundary conditions, calculate time steps, evaluate residuals as well as advance and update the solution in

the temporal space. A series of two- and three-dimensional test cases including compressible flows over single- and multi-element airfoils and a M6 wing are carried out to verify the developed code. The obtained solutions agree well with experimental data and other computational results reported in the literature. Detailed analysis on the performance of the developed code reveals that the developed CPU based implicit meshless method is at least four to eight times faster than its explicit counterpart. The computational efficiency of the implicit method could be further improved by ten to fifteen times on the GPU.

Keywords: Implicit meshless; GPU computing; LU-SGS; Rainbow coloring; Euler equations

1. Introduction

In recent years, graphics processing unit (GPU) computing technology has become increasingly popular in scientific research and engineering applications due to its rapidly growing performance and memory bandwidth. The fast development of this new technology provides tremendous computing power with Tera-scale floating operations per second to computational fluid dynamics (CFD), which requires intensive calculation for complex flow problems such as the fine-scale turbulence simulation of a complete fixed-wing aircraft [1], the aero-elasticity and stability of rotorcraft [2]and the hydrodynamic response of ships and offshore floating platforms subjected to extreme wave loadings [3].

In early days, programming on GPUs used to be a complicated exercise involving the use of low-level languages/techniques. This has been much improved with the development of high-level programming languages such as CUDA [4], OpenCL [5] and OpenACC [6]. With the emerge of these languages, more and more researchers in CFD have started to pay attention to GPU computing. Some important works, which successfully accelerate mesh based numerical methods including finite difference [7, 8], finite volume [9-13], finite element [14] and discontinuous Galerkin [15-17], have been reported in the literature.

Compared to the vast amount of effort that has been made to port mesh based methods for compressible flows from CPU to GPU, the attention paid to the implementation of meshless methods on GPUs for solving high-speed flows is still limited. Meshless methods, in contrast to mesh methods using strictly closed grid elements, only utilize clouds of points to discretize the computational domain. This provides much greater flexibility to accommodate complex

aerodynamic configurations [18-22]. Parallelization of these new methods on many-core graphics processors to calculate complex compressible flows more efficiently will undoubtedly be beneficial to scientific research and engineering applications. Recently some researchers have attempted to implement explicit meshless methods on GPUs to calculate 2D compressible flows [23, 24]. However, it remains obscure whether implicit meshless methods, which converge much faster than explicit meshless methods on CPUs, would be able to be ported to GPUs to achieve further acceleration.

One of the biggest challenges in realizing implicit methods on the GPU is these methods' inherent data dependency characteristics, which will inevitably cause thread-racing conditions that could corrupt the data on the computer [24]. It is relatively easy to modify explicit algorithms to avoid thread-racing conditions, but it is much harder to achieve the same objective for implicit methods.

This paper presents an effort to develop a recently proposed GPU based two-dimensional explicit meshless method for compressible flows reported by Ma et al. [23]. An efficient parallel LU-SGS implicit algorithm is devised and utilized to further improve the computational efficiency. The capability of the original 2D meshless code is extended to deal with 3D complex problems. To resolve the inherent data dependency of the standard LU-SGS method, which causes thread-racing conditions destabilizing numerical solution, a robust rainbow coloring method is presented and applied to organize the computational points into separate independent groups by painting neighboring points with different colors. The original serial LU-SGS method is modified and parallelized accordingly to perform calculations for all the computational points in a color-by-color independent manner. This method can deal with

77 both regularly and irregularly distributed points. It is more generic than the hyper-plane and
78 pipeline methods [25, 26], which are only applicable to structured grids. The CUDA Fortran
79 programming model [27] is employed to develop the important GPU kernels to apply boundary
80 conditions, calculate time steps, evaluate residuals as well as advance and update the solution in
81 temporal space.

82 The rest of the paper is organized as follows. The numerical model, including governing
83 equations and least-square curve fit based meshless discretization, is described in Section 2.
84 The rainbow coloring method and the corresponding parallel LU-SGS algorithm, which are
85 developed to avoid the data dependency of implicit methods, are addressed in Section 3. Key
86 aspects of GPU implementation of the parallel algorithm including the development of
87 computational kernels and the management of device memory are discussed in Section 4. The
88 resulting GPU-based implicit meshless algorithm is firstly validated with typical
89 two-dimensional flows over single- and multi-element airfoils and then used to accelerate the
90 simulations of more complex three-dimensional flows in Section 5 to demonstrate the
91 capability and performance of the algorithm. Finally, conclusions are drawn in Section 6.

92 **2. Spatial discretization**

93 In this section, a brief description of the numerical model, including the governing
94 equations for inviscid compressible flows and the least-square meshless discretization, is
95 presented for the sake of completeness.

96 2.1 Governing equations

97 The explicit GPU meshless method developed by Ma et al. [23] was only used to deal with

2D problems. It has not been addressed by these researchers whether this method could deal with complex 3D problems. In the present work we aim at solving three-dimensional compressible flows governed by the Euler equations, of which the differential form can be expressed as

$$\frac{\partial \mathbf{W}}{\partial t} + \nabla \cdot \overset{\mathrm{r}}{\mathbf{F}} = 0 \tag{1}$$

where $\mathbf{W}$ and $\overset{\mathrm{r}}{\mathbf{F}}$ are the vector of conservative variables and the convective flux terms, respectively. The definitions of them are given by

$$\mathbf{W} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{pmatrix} \quad \overset{\mathrm{r}}{\mathbf{F}} = \begin{pmatrix} \rho u \\ \rho uu + p \\ \rho uv \\ \rho uw \\ u(\rho E + p) \end{pmatrix} \overset{\mathrm{r}}{\boldsymbol{i}} + \begin{pmatrix} \rho v \\ \rho uv \\ \rho vv + p \\ \rho vw \\ v(\rho E + p) \end{pmatrix} \overset{\mathrm{r}}{\boldsymbol{j}} + \begin{pmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho ww + p \\ w(\rho E + p) \end{pmatrix} \overset{\mathrm{r}}{\boldsymbol{k}} \tag{2}$$

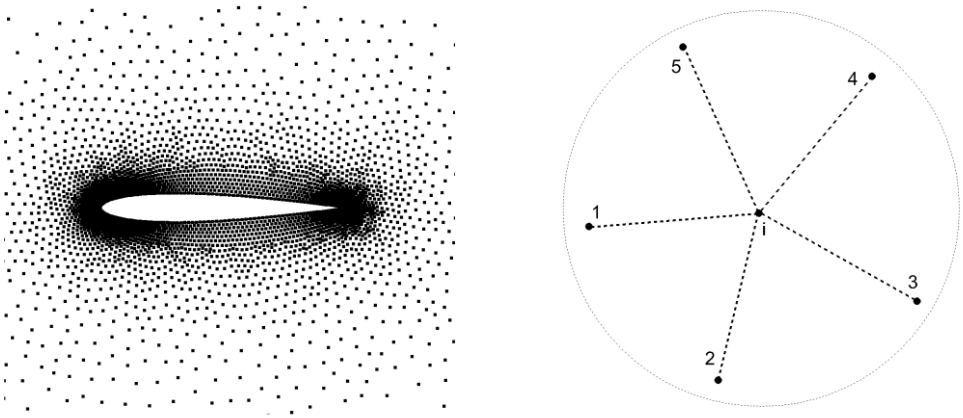
where ρ is the density, p is the pressure, u, v and w are the velocity components along x, y and z axes, respectively. The total energy per unit mass E is given by

$$E = \frac{1}{\gamma - 1}\frac{p}{\rho} + \frac{1}{2}(u^2 + v^2 + w^2) \tag{3}$$

where γ is the ratio of specific heat coefficients and $\gamma = 1.4$ for air.

2.2 Least-square curve fit based meshless discretization

In meshless discretization [18-24] of the partial differential equations for CFD like Equation (1), the physical domain of the problem should be firstly discretized with scattered points. For each point in the domain as shown in Fig. 1, several surrounding points are chosen to form a local cloud of points, where the surrounding points are called as the satellites of the central point. The spatial derivatives in governing equation (1) are approximated in the meshless clouds of points.



(a) scattered points around an airfoil (b) local cloud of point

Fig. 1. Meshless discretization of a computational domain.

For a given cloud of point C_i, the spatial derivatives of a sufficiently differentiable function $\phi(x,y,z)$ located at the central point i can be approximated by

$$\left.\frac{\partial\phi}{\partial x}\right|_i = \sum_{j\in C_i}\alpha_{ij}\phi_{ij} \quad \left.\frac{\partial\phi}{\partial y}\right|_i = \sum_{j\in C_i}\beta_{ij}\phi_{ij} \quad \left.\frac{\partial\phi}{\partial z}\right|_i = \sum_{j\in C_i}\gamma_{ij}\phi_{ij} \tag{4}$$

where ϕ_{ij} is estimated at the midpoint of the virtual edge $i-j$, and the condition $j\in C_i$ indicates that the summation should traverse all the satellites in C_i. The derivative weight coefficients α_{ij}, β_{ij} and γ_{ij} can be determined by various kinds of meshless treatments like least-square curve fit [18], radius basis functions [19], conservative meshless schemes [20]. In the present work, a weighted least-square curve fit based meshless method [28] is applied and the spatial derivative coefficients can be obtained by solving the following linear system

$$A_i \mathbf{a}_{ij} = B_{ij} \tag{5}$$

where the 3×3 matrix A_i and 3×1 matrix B_{ij} are given by

$$A_i = \begin{bmatrix} \sum_{k\in C_i}\omega_{ik}\Delta x_{ik}\Delta x_{ik} & \sum_{k\in C_i}\omega_{ik}\Delta x_{ik}\Delta y_{ik} & \sum_{k\in C_i}\omega_{ik}\Delta x_{ik}\Delta z_{ik} \\ \sum_{k\in C_i}\omega_{ik}\Delta y_{ik}\Delta x_{ik} & \sum_{k\in C_i}\omega_{ik}\Delta y_{ik}\Delta y_{ik} & \sum_{k\in C_i}\omega_{ik}\Delta y_{ik}\Delta z_{ik} \\ \sum_{k\in C_i}\omega_{ik}\Delta z_{ik}\Delta x_{ik} & \sum_{k\in C_i}\omega_{ik}\Delta z_{ik}\Delta y_{ik} & \sum_{k\in C_i}\omega_{ik}\Delta z_{ik}\Delta z_{ik} \end{bmatrix} \quad B_{ij} = \begin{bmatrix} \omega_{ij}\Delta x_{ij} \\ \omega_{ij}\Delta y_{ij} \\ \omega_{ij}\Delta z_{ij} \end{bmatrix} \tag{6}$$

in which $\Delta x_{ik} = x_k - x_i$, $\Delta y_{ik} = y_k - y_i$ and $\Delta z_{ik} = z_k - z_i$ are the coordinate differences between the center point i and satellite k, $\vec{a}_{ij} = \begin{bmatrix} \alpha_{ij} & \beta_{ij} & \gamma_{ij} \end{bmatrix}^T$ is the vector of derivative weight coefficients. To emphasize the contribution of certain points in the cloud, a weighting function ω is adopted, which usually takes the inverse square of its distance to the central point, with $w_{ij} = \left|\Delta \vec{r}_{ij}\right|^{-2}$. It can be noted that the derivative weight coefficients only depend on the nodal positions. Therefore, they are pre-computed and stored in the memory before other calculations.

2.3 Evaluation of the convective flux

Using the above mentioned derivative weight coefficients, the spatial derivative term in Equation (1) can be discretized in an arbitrary cloud C_i as

$$\nabla \cdot \vec{\mathbf{F}}_i = \sum_{j \in C_i} \vec{\mathbf{F}}_{ij} \cdot \vec{a}_{ij} \tag{7}$$

To estimate the convective flux $\mathbf{F}_{ij} = \vec{\mathbf{F}}_{ij} \cdot \vec{a}_{ij}$ on the virtual edge $i - j$, the JST scheme [29] is employed, which can be expressed as

$$\mathbf{F}_{ij} = \frac{1}{2}\left(\vec{\mathbf{F}}(\mathbf{W}_i) + \vec{\mathbf{F}}(\mathbf{W}_j)\right) \cdot \vec{a}_{ij} - \mathbf{D}_{ij} \tag{8}$$

where $\mathbf{D}_{ij}$ is the artificial dissipation consisting of a second-order and a fourth-order terms, and can be expressed as

$$\mathbf{D}_i = \varepsilon_{ij}^{(2)} \lambda_{ij} \left(\mathbf{W}_j - \mathbf{W}_i\right) - \varepsilon_{ij}^{(4)} \lambda_{ij} \left(\nabla^2 \mathbf{W}_j - \nabla^2 \mathbf{W}_i\right) \tag{9}$$

where $\varepsilon^{(2)}$ and $\varepsilon^{(4)}$ denote the second- and forth-order adaptive coefficients, respectively. ∇^2 is the Laplace operator. The spectral radius λ is also based on the meshless derivative weight coefficients, and given by

$$\lambda = \left|u\alpha + v\beta + w\gamma\right| + \sqrt{(\alpha^2 + \beta^2 + \gamma^2) \cdot \gamma\, p / \rho} \tag{10}$$

Additionally, the slip condition is enforced on all the solid wall boundaries, which means

that the normal velocity of the boundary points should be equal to zero. At the far field boundary, the non-reflecting condition is adopted to adjust the flow variables for all the boundary points. For more details on the parameters $\varepsilon^{(2)}$ and $\varepsilon^{(4)}$ and the far field boundary condition, readers can refer to the article [30].

3. Temporal discretization

3.1 Implicit LU-SGS scheme

The meshless method is used to evaluate the flux term given in Equation (8). By splitting the problem into the spatial and temporal spaces, Equation (1) can be re-written into a semi-discrete form for a meshless cloud C_i as

$$\frac{d\mathbf{W}_i}{dt} = -\sum_{j\in C_i} \mathbf{F}_{ij} \tag{11}$$

With a simple backward differential operator for $d\mathbf{W}$ and a first-order Taylor expansion for $\mathbf{F}$, the implicit form of Equation (11) can be expressed as [31]

$$\begin{aligned}\frac{\Delta\mathbf{W}_i^n}{\Delta t} &= -\sum_{j\in C_i} \mathbf{F}_{ij}^{n+1} = -\sum_{j\in C_i} \mathbf{F}(\mathbf{W}_i^{n+1}, \mathbf{W}_j^{n+1}) \\ &= -\sum_{j\in C_i}\left[\mathbf{F}(\mathbf{W}_i^n, \mathbf{W}_j^n) + \frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_i}\Delta\mathbf{W}_i^n + \frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_j}\Delta\mathbf{W}_j^n\right] \\ &= -\sum_{j\in C_i} \mathbf{F}_{ij}^n - \sum_{j\in C_i}\frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_i}\Delta\mathbf{W}_i^n - \sum_{j\in C_i}\frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_j}\Delta\mathbf{W}_j^n\end{aligned} \tag{12}$$

where $\Delta\mathbf{W}^n = \mathbf{W}^{n+1} - \mathbf{W}^n$ is the increment of the conservative variables, and Δt denotes the time step. The superscript n and $n+1$ denote the current and the next time steps, respectively. $\frac{\partial \mathbf{F}}{\partial \mathbf{W}}$ is the Jacobian matrix with respect to the conservative variables for each local cloud of points. After moving the Jacobian matrix terms to the left side, the above equation can be written as

$$\left(\frac{1}{\Delta t}\boldsymbol{I}+\sum_{j\in C_i}\frac{\partial \mathbf{F}_{ij}^{n}}{\partial \mathbf{W}_i}\right)\Delta\mathbf{W}_i^{n}+\sum_{j\in C_i}\frac{\partial \mathbf{F}_{ij}^{n}}{\partial \mathbf{W}_j}\Delta\mathbf{W}_j^{n}=-\sum_{j\in C_i}\mathbf{F}_{ij}^{n} \tag{13}$$

Applying Equation (13) to all of the clouds of points in the domain and assembling these equations, we will obtain a system of block matrix equations given by

$$\mathbf{A}(\mathbf{W}^{n})\Delta\mathbf{W}^{n}=-\mathbf{R}^{n} \tag{14}$$

in which,

$$\mathbf{A}=\left[\mathbf{A}_{ij}\right]\quad \mathbf{A}_{ij}=\begin{cases}\dfrac{1}{\Delta t}\mathbf{I}+\displaystyle\sum_{k\in C_i}\frac{\partial \mathbf{F}_{ik}^{n}}{\partial \mathbf{W}_i} & i=j\\ \dfrac{\partial \mathbf{F}_{ij}^{n}}{\partial \mathbf{W}_j} & i\neq j\end{cases}\qquad \Delta\mathbf{W}^{n}=\begin{bmatrix}\Delta\mathbf{W}_1^{n}\\ \Delta\mathbf{W}_1^{n}\\ \mathrm{M}\\ \Delta\mathbf{W}_N^{n}\end{bmatrix}\qquad \mathbf{R}^{n}=\begin{bmatrix}\displaystyle\sum_{j\in C_1}\mathbf{F}_{1j}^{n}\\ \displaystyle\sum_{j\in C_2}\mathbf{F}_{2j}^{n}\\ \mathrm{M}\\ \displaystyle\sum_{j\in C_N}\mathbf{F}_{Nj}^{n}\end{bmatrix} \tag{15}$$

The linear system of Equation (14) encapsulates the implicit iteration schemes, and it can be solved iteratively to converge to a steady state. The standard LU-SGS scheme consists of a forward iteration and a backward iteration sweeping through all the computational points in a sequential order [31], which can be written as

$$\begin{aligned}&\mathit{Forward}:\Delta\mathbf{W}_i^{*}=-\mathbf{A}_{ii}^{-1}\left[\mathbf{R}_i^{n}+\sum_{j\in C_i}^{j<i}\mathbf{A}_{ij}\Delta\mathbf{W}_j^{*}\right]\quad i=1,2,...,N-1,N\\ &\mathit{Backward}:\Delta\mathbf{W}_i^{n}=\Delta\mathbf{W}_i^{*}-\mathbf{A}_{ii}^{-1}\sum_{j\in C_i}^{j>i}\mathbf{A}_{ij}\Delta\mathbf{W}_j^{n}\quad i=N,N-1,...,2,1\end{aligned} \tag{16}$$

In the forward step of Equation (16), it can be seen that $\Delta\mathbf{W}_j^{*}$ on the right side should be calculated and prepared before computing the increment $\Delta\mathbf{W}_i^{*}$. The similar situation occurs in the backward step. The ordered forward and backward sweep of the standard LU-SGS scheme works well in serial computation. However, it is not applicable to multi- and many-core parallel computation. Because a computational point could be accessed simultaneously by several threads with conflicting writing operations, which could lead to an unstable solution that is

neither predictable nor reproducible. Therefore, the standard LU-SGS scheme cannot be directly used in GPU computing.

3.2 Rainbow coloring method

As mentioned before, data dependency impedes the parallel implementation of the standard LU-SGS algorithm. Some special strategies have been proposed in the past to undertake parallel computation on structured grids, which include the alternating direction implicit method [11], red-black ordering method [12], hyper-plane/hyper-line method [25] and pipeline methods [26]. Unfortunately, the application of these methods is limited to structured meshes only so that they are not suitable to other methods using irregularly distributed points and/or grids. Despite this limitation, a careful comparison of these methods gives us a hint that data independency for irregularly distributed meshless points and/or mesh cells can still be achieved if a proper treatment is used to separate them into several different groups. It is expected that all the points in the same group could be manipulated simultaneously by parallel threads without interfering each other. In addition, the underlying numerical algorithm needs to be modified properly to assure that write operations will be carried out in a group-by-group manner. These two conditions will guarantee that there will be no conflicting operations at a computational point at any time. Some researchers proposed a reordering method to paint unstructured meshes cells with different colors [32]. However, this technique has only been tested on multi-core CPUs so far and whether it could be applied to GPU computing remains unknown.

In the current work, we develop and present a rainbow coloring method to organize

meshless clouds of point into independent groups for GPU computing. The whole procedure to paint all the computational points is described in Algorithm 1. The essential criterion of this coloring algorithm is that any two neighboring points are decorated with different colors. The central point must not have the same color with any of its satellite. In the computer program, we use integer numbers to represent different colors. For example, the red color is represented by index 1 and the blue color can be illustrated by index 2.

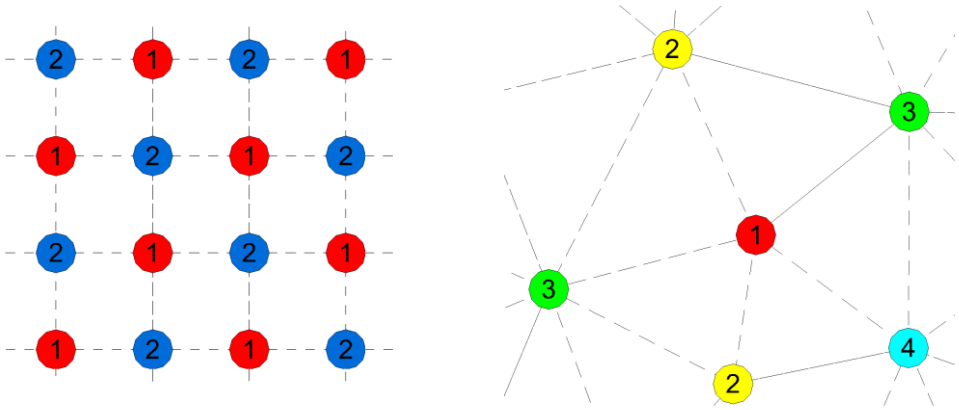
Algorithm 1 The procedure of rainbow coloring method

Input: The original meshless clouds of points $\{C_i | i \in \Omega\}$ and a start point v_0.
Output: The array of point colors $color(:)$ and total number of colors N_{color}.

initialize $color(:) = 0$;
choose a point v_0 as the start point of the traversal, and set $color(v_0) = 1$;
repeat
  For each colored point $\{v \mid color(v) > 0\}$**do**
    For each uncolored point $\{w \mid w \in C_v\}$**do**
      paint $color(w) = min\{k > 0 \mid k \neq color(j)\ \forall\ j \in C_w\}$;
until all points are painted

The painting procedure given in Algorithm 1 is initialized by choosing a start point v_0 in the computational domain. Once the start point is selected, the corresponding color graph will be determined accordingly. In order to know whether different choices of the start point will have significant effect on the overall computational efficiency, we have tried choosing a start point randomly and found out that its influence is almost negligible. Therefore, in the present work the first point in the global array is always selected as the start node for the sake of convenience. Examples of the generated color graphs for both regularly and irregularly distributed meshless clouds are illustrated in Fig. 2. The dashed lines in the figure are not used in calculation, they are only used here to present a clear view of neighboring points. As shown in Fig. 2(a), a simple unique graph with two colors is obtained by using Algorithm 1 for

regularly distributed meshless. It can be seen that the implicit computing (see Equation (16)) of each red point (with color index 1) depends only on itself and the surrounding black points (with color index 2) in its local cloud, while the implicit computing of each black point only relies on itself and the surrounding red points. Therefore, algebraic operations at the points with the same color are independent with each other and they can be easily parallelized. Irregularly distributed meshless points can be treated in the same way, but more colors may be needed to paint these points due to the complex distribution as shown in Fig. 2(b). Obviously, the rainbow coloring method can deal with different types of point distribution, so it is more general than the ADI, red-black, hyper-plane and pipe-line methods, which can only be applied to regularly distributed points.



(a) regular distribution (b) irregular distribution

Fig. 2. Examples of color graphs.

3.3 Parallel LU-SGS method

The standard LU-SGS algorithm sweeps all the computational points in a sequential order, unfortunately this is not applicable to parallel computing. Here we modify it by using the rainbow coloring strategy so that the new algorithm traverses all the data points in a

group-by-group manner from the first color to the last color in the forward updating step, then it moves across the points from the last color to the first color in the backward iteration. The detailed procedure of the parallel LU-SGS method is presented in Algorithm 2, where the variable N_{color} indicates the total number of colors and L_s is a one-dimensional array storing all the colors used to paint the computational points. The data dependency issue can be successfully avoided by using this method. In the next section, we will discuss the implementation of the proposed parallel algorithm on the GPU.

Algorithm 2 The procedure of parallel LU-SGS method

Forward updating:

for $(s = 1\ to\ N_{color})$ **do**

  compute $\Delta \mathbf{W}_i^* = -\mathbf{A}_{ii}^{-1} \left\{ \sum_{j \in C_i}^{L(j)<s} \mathbf{A}_{ij} \Delta \mathbf{W}_j^* + \mathbf{R}_i^n \right\}$ for $i \in L_s$ in parallel;

Backward updating:

for $(s = N_{color}\ to\ 1)$ **do**

  compute $\Delta \mathbf{W}_i^n = \Delta \mathbf{W}_i^* - \mathbf{A}_{ii}^{-1} \sum_{j \in C_i}^{L(j)>s} \mathbf{A}_{ij} \Delta \mathbf{W}_j^n$ for $i \in L_s$ in parallel;

4. GPU implementation

CUDA, OpenCL and OpenACC are three major programming models used to develop accelerator codes. The comparison of these models' advantages and disadvantages is beyond the scope of the present work. Here we choose the CUDA Fortran language [27] to develop the parallel implicit meshless program on the GPU.

4.1 Program framework

In practical programming, the time-consuming parts are usually parallelized on the GPU while the other parts are kept on the CPU. For the implicit meshless method mentioned before,

the works related to the I/O operation and the generation of meshless clouds are kept on the CPU side since the former needs to deal with external storages like hard drives and the latter is calculated only once before other computations. The functions related to the implicit time marching are the most computing intensive parts. Hence these works need to be accelerated on the GPU. The implicit time marching procedure in each time step involves boundary condition enforcement, spectral radius calculation, time step estimation, flux term evaluation and solution update. For every single small task, a corresponding GPU kernel function is developed accordingly by using the CUDA Fortran language. The framework of the whole computer program is illustrated in Fig. 3, in which different tasks are assigned to the CPU and GPU, respectively.

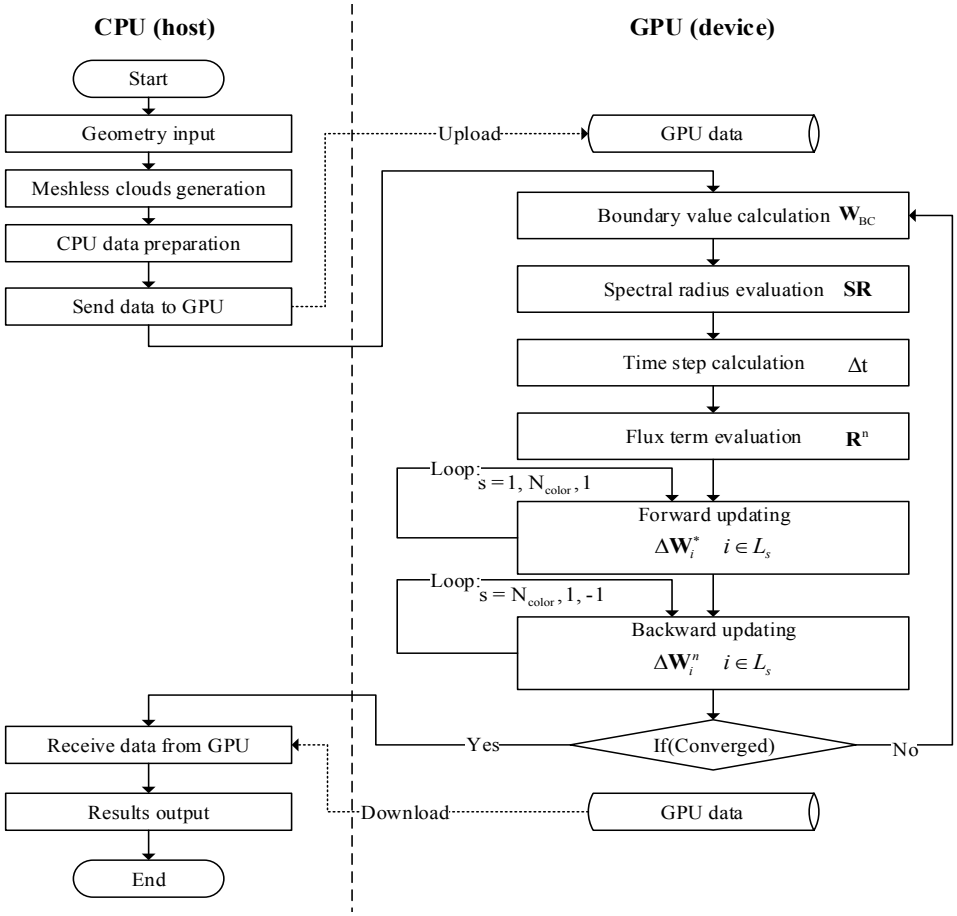


Fig. 3. The general program procedure of GPU-based implicit meshless approach

As shown in Fig. 3, the program starts from the CPU side with the pre-processing tasks including geometry input, meshless clouds generation and necessary data initialization, which should be executed before invoking the GPU kernel functions. Once the computing tasks on the GPU are finished, the results are sent back to the CPU for post-processing. A key to the success of GPU programming lies in the development of kernel functions and careful management of the device memory.

4.2 CUDA kernel functions

In the present work, the CUDA functions developed for the time marching procedure are categorized into three types including **internal**, **boundary** and **update kernels** according to the actual tasks assigned to them.

```
1   attributes(global) subroutine kernel_TimeStep()
2
3     !!get the thread index
4     i=(blockIdx%x-1)*blockDim%x + threadIdx%x
5
6     if(i <= N) then !!judge to avoid out of bounds
7       dti = 0.0
8       do j=1,nSate(i) !!sum through all satellites
9         dti = dti + SR(j,i) !!SR is the spectral radius
10      endDo
11      DT(i) = CFL/dti
12    endIf
13
14  endSubroutine
```

Listing 1. An example of internal kernel for time step calculation

Internal kernels are used to calculate the spectral radius, time step and flux term for internal field meshless clouds of points. For every meshless cloud of points, a CUDA thread is launched on the device to undertake important tasks. The total number of threads created the CUDA device should be no less than the number of points in the domain. An example of the internal kernel function for time step calculation is presented in Listing 1, in which every thread

deals with one local cloud. The variable *N* in the example code is the total number of points in the computational domain.

Boundary kernels are designed to enforce boundary conditions including no-penetration wall, symmetric plane and non-reflective far field in the present work. We noted that if the near-boundary points are treated differently with the field points, the efficiency of the related kernels will be excessively degraded due to the divergence of thread branch. In the present work, similar treatment of both near-boundary and field points is adopted to avoid the branch divergence by introducing ghost points to implement boundary conditions, which is carried out by a specific kernel. An example code of the boundary kernel is given in Listing 2, in which each thread evaluates the boundary values for one ghost point. The variable *nBC* is the total number of ghost points.

```
1   attributes(global) subroutine kernel_Boundary()
2
3     !!get the thread index
4     i=(blockIdx%x-1)*blockDim%x + threadIdx%x
5
6     if(i <= nBC) then !!judge to avoid out of bounds
7       nodeID      = iNode_BC(i)       !!get left node index
8       ww_left(:) = ww(nodeID,:)       !!get left values
9       normal(:)   = iBCNormal(i,:)    !!get normal vector
10
11      select (BCType(i))
12      case WALL:        !!wall boundary
13        call BC_wall(ww_left, normal, wwFi)
14      case SYMM:        !!symmetry boundary
15        call BC_symm(ww_left, normal, wwFi)
16      case FAR:         !!far field boundary
17        call BC_far(ww_left, ww_far, normal, wwFi)
18      endSelect
19
20      wwBC(i,:) = wwFi(:)  !!store the boundary values
21    endIf
22
23  endSubroutine
```

Listing 2. The kernel for boundary value evaluation of ghost points

Update kernels are developed to advance the solution in the temporal space. Two kernels

303 namely *LUSGS_Lower* and *LUSGS_Upper* are designed to execute the forward and backward
304 updating steps as described in Algorithm 2, respectively. Example code of the kernel
305 *LUSGS_Lower* is illustrated in Listing 3, where s is the index of color group and
306 $nPoin\_clor(s)$ is the total number of points in that group.

```
1   attributes(global) subroutine kernel_LUSGS_Lower(in s)
2     !!s is color layer index
3
4     !!get the thread index
5     i=(blockIdx%x-1)*blockDim%x + threadIdx%x
6
7     !! get the point index for current thread
8     pID = i + beginID_clor(s)
9
10    if(i <= nPoin_clor(s)) then !!judge to avoid out of bounds
11      R(:) = 0.0
12
13      do j=1,nSateL(pID) !! sum through all Lower satellites
14        sateID = iSateL(pID,j) !!get sateID
15
16        R(:) = R(:) + Aij(pID,j,:,:)*(ww(sateID,:) - wwOld(sateID,:))
17      endDo
18
19      !!update solution variables
20      ww(pID,:) = ww_old(pID,:) - Aii_inv(pID,:,:) * (R(:) + Res(pID,:))
21    endIf
22
23  endSubroutine
```

307

308 **Listing 3.** The update kernel for forward marching of LU-SGS

```
1   subroutine timeMarching_LUSGS()
2
3     call kernel_Boundary<<<nBlock_BC, 64>>>()
4
5     call kernel_SpectralRadius<<<nBlock_Node, 64>>>()
6
7     call kernel_TimeStep<<<nBlock_Node, 64>>>()
8
9     call kernel_Flux<<<nBlock_Node, 64>>>()
10
11    do s=1,nClor,1
12      call kernel_LUSGS_Lower<<<nBlock_Clor(s), 64>>>(s)
13    endDo
14
15    do s=nClor,1,-1
16      call kernel_LUSGS_Upper<<<nBlock_Clor(s), 64>>>(s)
17    endDo
18
19  endSubroutine
```

309

310 **Listing 4.** The host fuction for launching GPU kernels

311 Listing 4 shows the executing order of the GPU kernels, which is controlled by the CPU
312 function *timeMarching_LUSGS*. For every kernel, a two-layer hierarchy is used to manage the
313 CUDA threads launched on the device. As shown in Fig. 4, all threads in a kernel are organized
314 into a set of thread blocks to form a CUDA grid, and each thread block contains the same
315 number of threads. Depending on the underlying numerical method, the CUDA grid and thread

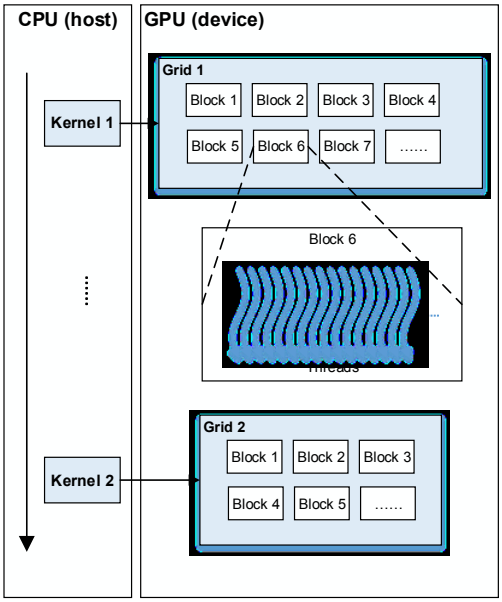


Fig. 4. The thread hierarchy of CUDA kernels.

316 block can be one-dimensional or multi-dimensional. Two parameters, *gridDim* and *blockDim*,
317 are usually used to control the needed dimensions when calling a GPU kernel. In the present
318 work, we set both the CUDA grid and thread block to be one-dimensional, which means
319 *gridDim* is equal to the number of blocks and *blockDim* is equal to the number of threads per
320 block. In order to optimize the GPU performance, the number of threads per block for each
321 kernel should be carefully tuned. According to our recently reported work [33], 64 threads per
322 block is a reasonable choice for the CUDA kernels. Thus the total number of thread blocks
323 could be determined by

$$gridDim = (nTotalThread + blockDim - 1) / blockDim \quad (17)$$

where $nTotalThread$ represents the total number of threads.

4.3 Device memory management

The performance of a GPU kernel function is heavily influenced by various types of memories, among which global memory, shared memory and register are three major types of memories that could be used and controlled by programmers. In order to enhance the overall performance of the program, efforts should be made to achieve an optimal use of the device memeory.

In this paper, the thread index is used to build the mapping relationships between the threads of the kernels and the corresponding computing data stored on the graphics card for memory addressing. As presented in Listings 1, 2 and 3, three build-in variables, *blockDim*, *blockIdx* and *threadIdx*, related to the thread hierarchy are used to compute the thread index. The utilizing of these important variables can be found in article [4] for details. When fetching data from or writing them to the global memory, coalesced memory access is the ideal pattern [34]. This pattern is adopted in the present work so that all the threads in a half wrap map/access the global memory simultaneously with respect to the center of a meshless cloud. In reality, this means consecutive thread access consecutive memory addresses [33, 34].

The low-latency shared memory, which is usually used in structured grid based regular computation for sharing data between sibling threads in the same block, is not utilized in the present work due to the unpredictable irregular memory access pattern of the meshless method with respect to satellite points in a cloud. Instead, the shared memory is used as an extension to

the registers to store local variables of each thread. For each local variable stored in the shared memory, a memory space with size of *blockDim* is allocated for each thread block and the variable *threadIdx* is used to search the corresponding value for each thread.

The registers, which have the lowest latency compared to other types of GPU memory, are used to store local variables for each thread. It should be noted that the number of registers provided by the hardware is very limited. A careful and delicate management is needed to ease the pressure on this scarce resource. Proper reusing of non-conflicting local variables and tuning the number of threads in a block are helpful to reduce the register pressure and to achieve the optimal performance [33].

5. Numerical results and analysis

Table 1 Specifications of the Intel core i5-3450 CPU and NVIDIA GTX TITAN GPU.

		Intel i5-3450	NVIDIA GTX TITAN
Processor	Total number of cores	4	2688
	Clock rate	3.10 GHz	837 MHz
Memory	Global memory	16GB	6GB
	Shared memory	-	64KB
	Registers per block	-	49152
Theoretical performance	Single-precision FLOP	198.4 GFLOP/s	4500 GFLOP/s
	Double-precision FLOP	99.2 GFLOP/s	1500 GFLOP/s
	Memory bandwidth	25.6 GB/s	288 GB/s

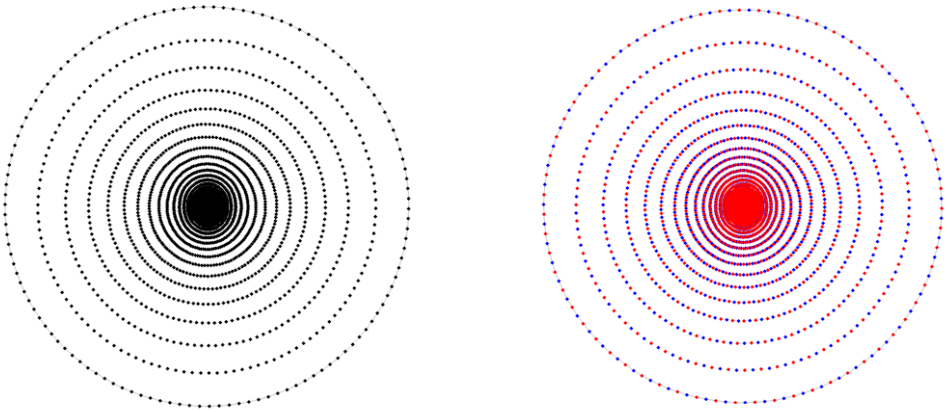
A set of 2D and 3D inviscid compressible flows over aerodynamic bodies, for which regularly or irregularly distributed meshless clouds of pointed are used, have been carried out to verify the developed code. To evaluate the overall computing performance, we have

359 programmed and benchmarked four suits of CFD codes: 1) CPU based explicit code (CE), 2)
360 CPU based implicit code (CI), 3) GPU based explicit code (GE) and 4) GPU based implicit
361 code (GI) in the present work. Both the explicit and implicit CPU codes are executed in the
362 serial mode using only one core. All the codes run in the double-precision mode. Wall time is
363 recorded for all the codes to make direct comparisons. The hardware employed in the present
364 work is a desktop workstation equipped with an Intel I5-3450 CPU and a NVIDIA GTX TITAN
365 GPU, of which the specifications are presented in Table 1.

366 5.1 Transonic flow past a NACA0012 airfoil

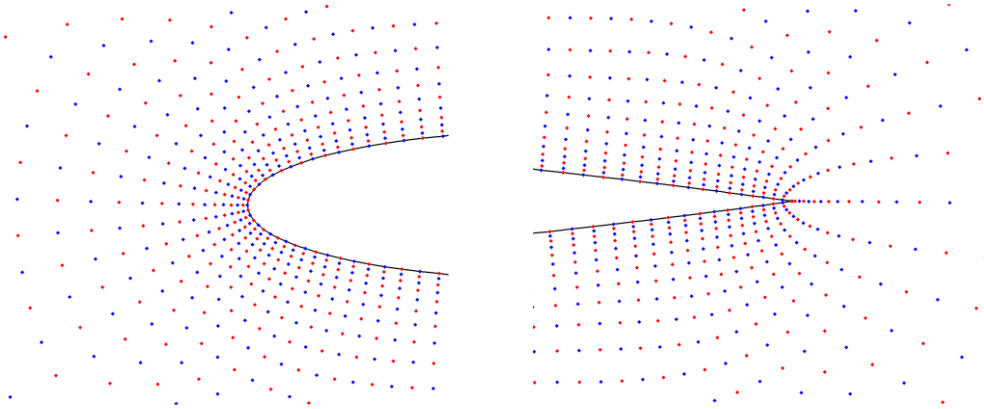
367 Two-dimensional inviscid compressible flow over a NACA0012 airfoil is firstly simulated
368 to validate the numerical method. In the computation, the freestream conditions are assigned
369 with Mach number $M_\infty = 0.8$ and angle of attack $\alpha = 1.25^o$. The computational domain is
370 discretized with 128×40 points regularly distributed as shown in Fig. 5(a). Each internal cloud
371 of points is composed of one central point and four surrounding satellite points. Fig. 5(b) shows
372 the corresponding color graph obtained by using Algorithm 1. Close views of the graph at the
373 leading and trailing edges of the airfoil are presented in Fig. 6. It can be seen that the red and
374 blue points appear alternately in the graph, and hence total 2560 red points and 2560 blue points
375 are painted respectively.

376 The computed results including Mach number contours and pressure coefficients are
377 depicted in Fig. 7. Experimental data and reference numerical results published in the literature
378 [18, 35] are also presented here to facilitate a direct comparison. It can be seen that the present
379 solution agrees well with these reference experimental and numerical results.



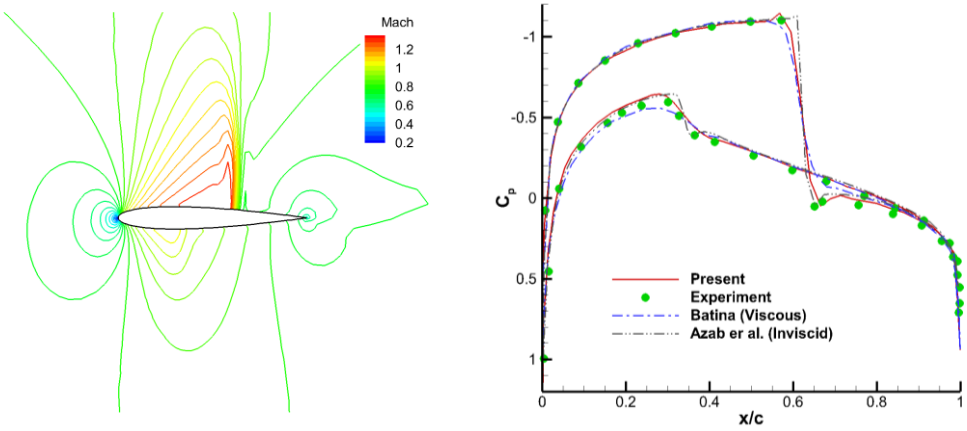
(a) meshless cloud (b) color graph

Fig. 5. The whole meshless cloud and color graph around the NACA0012 airfoil.



(a) the leading edge (b) the trailing edge

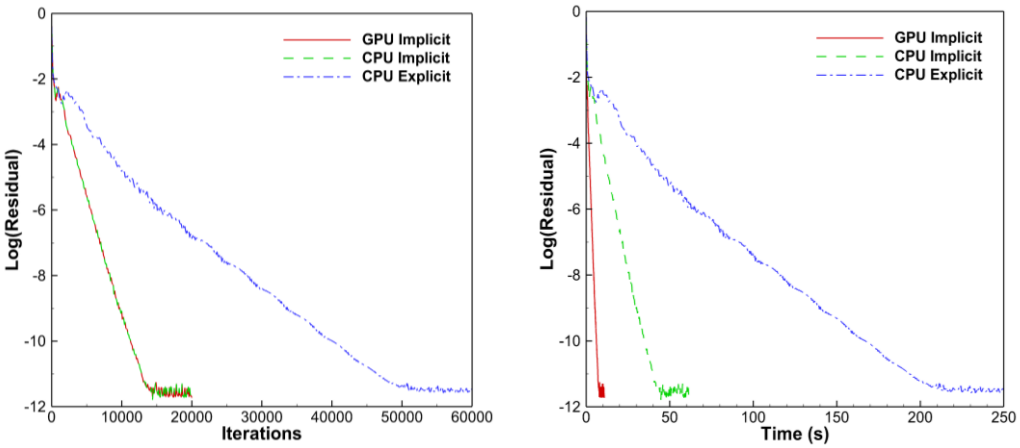
Fig. 6. The detailed color graphs around the NACA0012 airfoil.



(a) contours of Mach number (b) plots of pressure coefficient

Fig. 7. Computed results for transonic flow past the NACA0012 airfoil.

The histories of residual convergence with respect to iteration and wall time are shown in Fig. 8. It can be noted that the numbers of iteration of the implicit algorithms used to achieve the convergence are only a quarter of the explicit method. The implicit methods on the CPU and GPU have the same convergence rate per iteration. Compared to the large amount of computing time spent by the CPU based explicit method, the CPU implicit algorithm could reduce it effectively. The time cost could be further cut by the GPU implicit code.



(a) residual against iteration (b) residual against time

Fig. 8. Convergence histories for transonic flow past the NACA0012 airfoil.

5.2 Subsonic flow past a three-element airfoil

Two-dimensional inviscid compressible flow past a three-element airfoil with $M_\infty = 0.2$ and $\alpha = 1.25^o$ is then simulated to test the performance of the algorithm using irregularly distributed meshless clouds of points. There are 9592 points irregularly distributed in the computational domain as shown in Fig. 9. By adopting Algorithm 1, six colors are requested to paint all the points. The detailed color graphs at the leading and trailing gaps are presented in Fig. 10. Specifically, the numbers of points in each of the six color groups are 2600, 2525, 2314, 1818, 332 and 3, respectively.

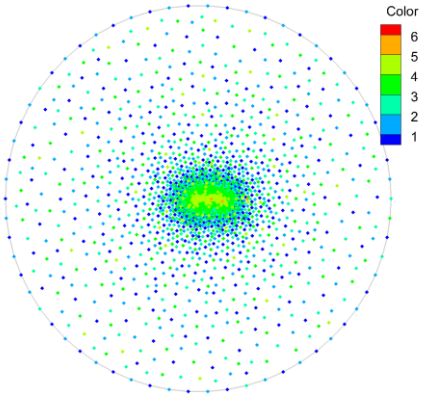
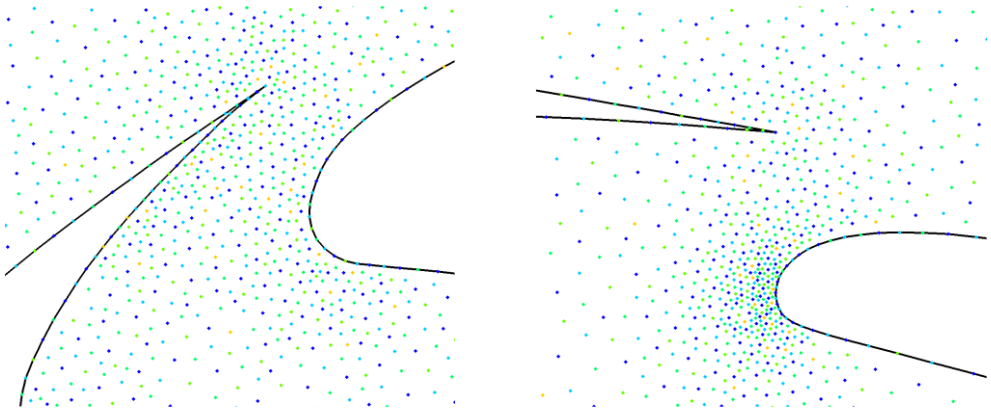


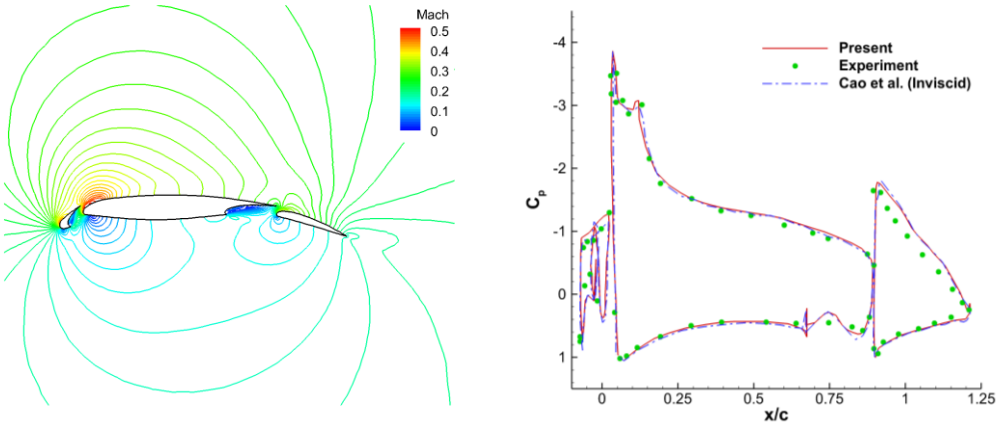
Fig. 9. The whole meshless cloud and color graph around the three-element airfoil.



(a) the leading gap (b) the trailing gap

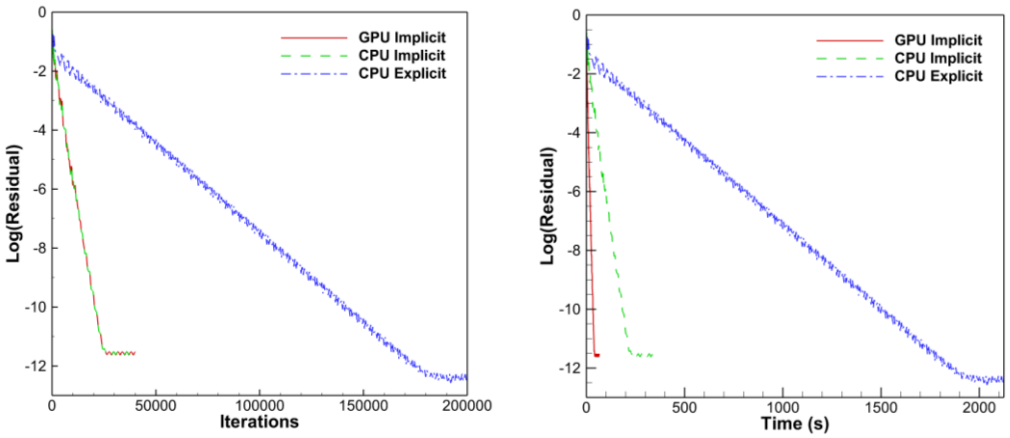
Fig. 10. The detailed color graphs around the three-element airfoil.

Fig. 11 shows the computed results including the Mach number contours and the pressure coefficient plots, which are close to the experimental data and other numerical results reported in the literature [36]. The histories of convergence in terms of iteration and time are presented in Fig. 12. It can be seen from Fig. 12(a) that the numbers of iterations needed to achieve the convergence for implicit algorithms are only about one-eighth of the explicit method. Once again, we can notice that the implicit methods could effectively reduce the computing time compared to the explicit method.



(a) contours of Mach number (b) plots of pressure coefficient

Fig. 11. Computed results for subsonic flow past the three-element airfoil.



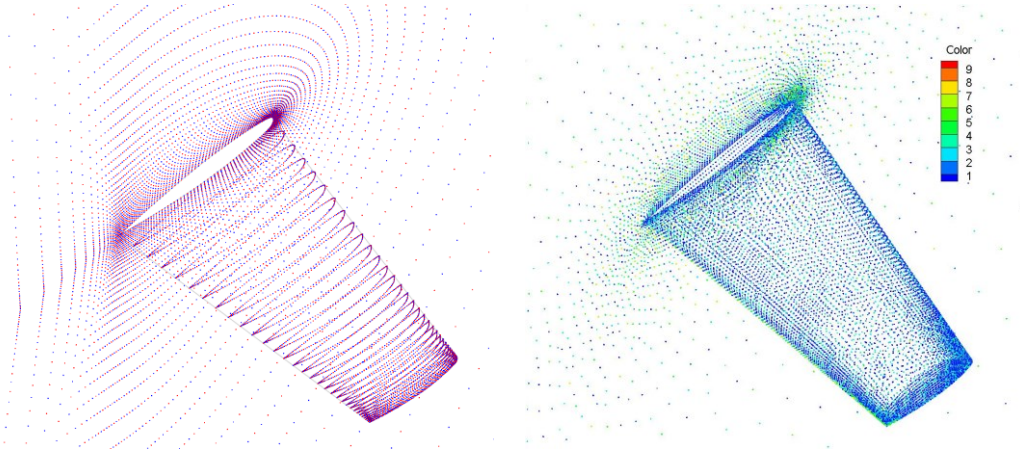
(a) residual against iteration (b) residual against time

Fig. 12. Convergence histories for subsonic flow past the three-element airfoil.

5.3 Transonic flow past a M6 wing

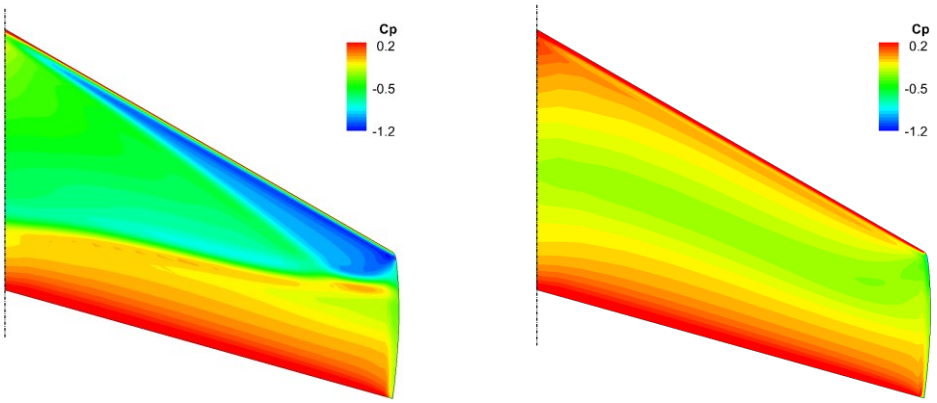
After testing two-dimensional problems, the develop code is used to accelerate the simulation of complex flows over three-dimensional aerodynamic bodies. Here, a typical transonic flow problem for the ONERA M6 wing with the Mach number $M_\infty = 0.84$ and the angle of attack $\alpha = 3.06^o$ is tested with regularly and irregularly distributed points. Fig. 13 shows the points distributed on the wing surface and the symmetric plane. It can be noted that only two colors are used for the regular distribution while nine colors are needed to paint the

irregularly distributed points. The numerical results computed for the two sets of points are very close to each other, hence for convenience we only present the flow obtained on the first set of points.



(a) points distributed regularly (b) points distributed irregularly

Fig. 13. The whole meshless cloud and color graph around the M6 wing.



(a) the upper surface (b) the lower surface

Fig. 14. The contours of pressure coefficient at the surface of M6 wing.

Fig. 14 shows the pressure coefficient contours on the upper and lower surfaces of the wing. It can be noted that the characteristic lambda shock on the upper surface of the wing is clearly captured. Pressure coefficients computed at several span-wise sections of the wing are presented in Fig. 15, where experimental data [37] and other numerical results published in the articles [38, 39] are also plotted. It can be noted that the present solution agrees well with these

reference data.

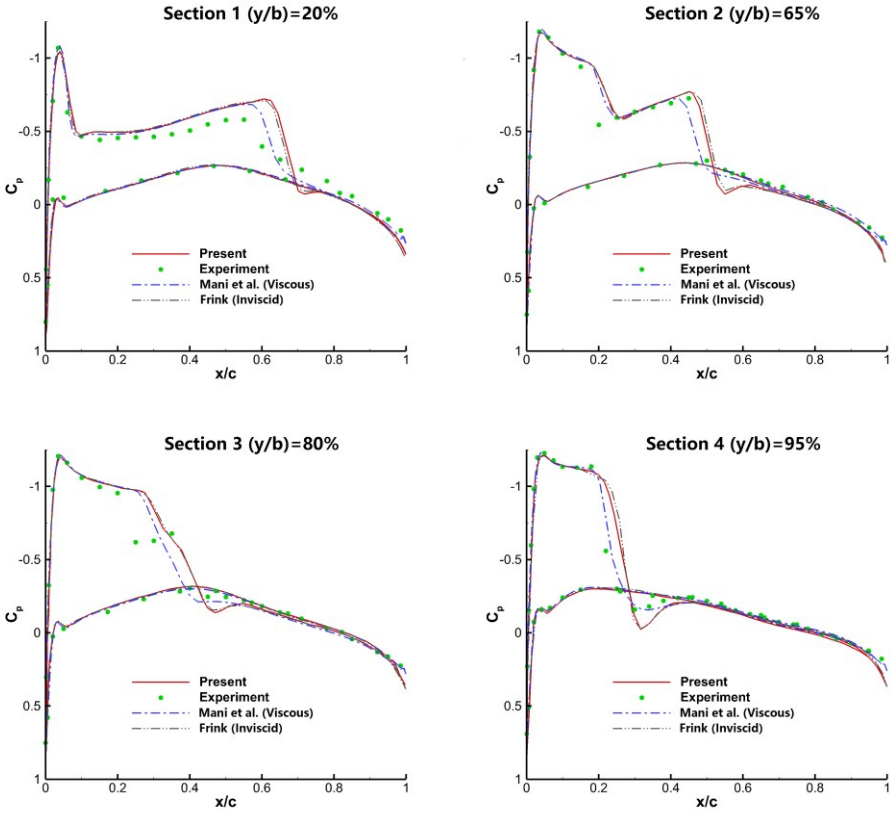
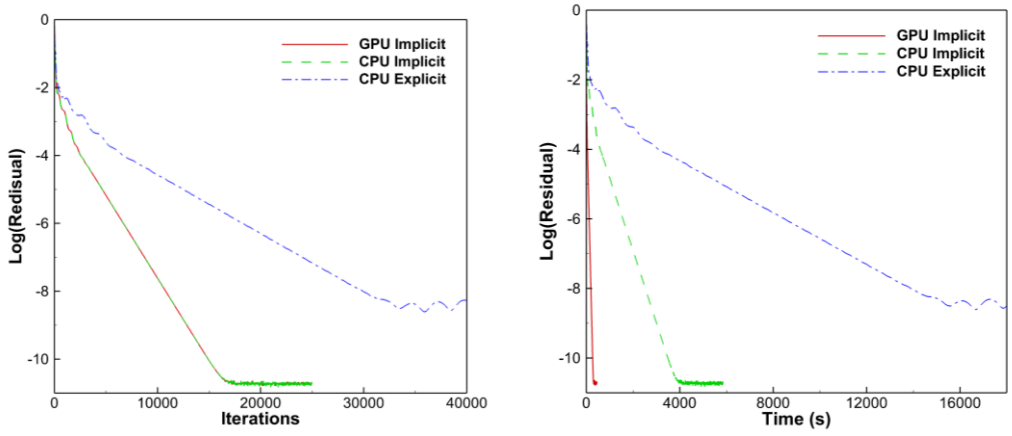


Fig. 15. The plots of pressure coefficient at four inboard sections of the M6 wing.



(a) residual against iteration (b) residual against time

Fig. 16. The comparation of convergence histories for transonic flow past the M6 wing.

Fig. 16 shows the histories of convergence obtained by the CE, CI and GI codes. It can be seen from Fig. 16(a) that for achieving the convergence, the numbers of iterations used by

implicit codes are only one-third of the explicit code. The saving in time offered by the GI code is very significant as illustrated in Fig. 16 (b).

5.4 Performance analysis

To have a quantitative comparison of the performance for all the codes used in the present work, we set 10^{-8} as the convergence criteria for all the test cases. The actual costs of computing (wall) time for all the four codes are listed in Table 2. For the M6 wing (Case 3), the explicit CPU code needs nearly 3.9 hours to bring down the residual by 8 orders of magnitude, the implicit CPU code requires about 42 minutes, the explicit GPU code spends 9.5 minutes, while the implicit GPU code only asks for 3.3 minutes. This achievement is impressive and especially useful to engineers who need to conduct a quick and accurate analysis on the aerodynamic performance of aircraft. Multiple 3D simulations could be completed in a relative short time to assist engineers to identify and optimize the key parameters to improve the performance of aircraft such as the ratio of lift to drag.

Table 3 presents the speedup, which compares the time costs of (any) two codes from the four. On the CPU, the implicit code offers a speedup from 4.46 to 8.11 compared to the explicit code. If accelerating the explicit code on the GPU, we can gain a speedup from 7.20 to 24.34. If the implicit code is parallelized on the GPU, we can get a speedup from 5.78 to 12.50. Comparing the GPU based implicit code to the explicit GPU program, we can have a speedup from 2.86 to 4.20, which is less than the speedup on the CPU side with respect to the ratio of CI to CE. The drop in the speedup of implicit method over explicit algorithm on the GPU side is due to the overhead of executing multiple colored small LU-SGS kernel functions. Launching a

kernel on the device is not free in terms of time, it actually causes overhead, which is usually more expensive than calling a similar function on the CPU. This phenomenon is consistent with the general idea in the high performance computing community that the parallelization of implicit codes is usually much more difficult than explicit programs. Nevertheless, the outcomes here demonstrate that the present work is of value that parallelizing the implicit code on the GPU could further cut computing time cost effectively compared to the explicit GPU code.

Table 2 Computing time cost.

		Computing time (seconds)			
Case	Number of points	CPU explict	CPU implicit	GPU explicit	GPU implicit
1	5120	1.16×10^{2}	2.60×10^{1}	1.61×10^{1}	4.50×10^{0}
2	9592	1.14×10^{3}	1.40×10^{2}	1.05×10^{2}	2.50×10^{1}
3	306577	1.39×10^{4}	2.50×10^{3}	5.71×10^{2}	2.00×10^{2}

Table 3 Speedup. CE: CPU explicit; CI: CPU implicit; GE: GPU explicit; GI: GPU implicit.

		Speedup				
Case	Number of points	CI/CE	GE/CE	GI/CE	GI/CI	GI/GE
1	5120	4.46	7.20	25.78	5.78	3.58
2	9592	8.11	10.86	45.60	5.60	4.20
3	306577	5.56	24.34	69.50	12.50	2.86

5.5 Size effect

For the first and second 2D cases, we only obtain a relatively small speedup in the range of 5 to 6 with respect to GI/CI. For the 3D case, the speedup rises to 12.50. The similar situation occurs for the explicit code on GPU with respect to GE/CE. In fact, the numbers of points used for the first and second cases are less than 10,000, which are not large enough to keep the GPU busy. In general, the GPU likes the programmer to feed it as much data as possible. Heavier the

491 better is a principle in GPU computing towards achieving the full potential of many-core
492 processors.

493 To investigate the size effect on the speedup, here we carry out extra tests of the implicit
494 CPU and GPU codes by continually increasing the number of points used for the computation.
495 The obtained computer time as well as the speedup are listed in Table 4. It is interesting to note
496 that a relatively stable speedup around 15 could be accomplished by providing large number of
497 data points (over 15 thousand) for the regular distribution case. For large number of irregularly
498 distributed points, we can achieve a speedup of 10 in average.

499 **Table 4** Size effect on the computing time and speedup. CI: CPU Implicit; GI: GPU Implicit.

		Computing time per iteration (seconds)		Speedup
Case	Number of points	CI	GI	GI/CI
Regular distribution	155680	1.2287×10^{-1}	8.4870×10^{-3}	14.5
	306577	2.3524×10^{-1}	1.6226×10^{-2}	14.5
	601408	4.6477×10^{-1}	3.0579×10^{-2}	15.2
	1193504	8.9608×10^{-1}	6.0897×10^{-2}	14.7
Irregular distribution	164160	2.7640×10^{-1}	3.2305×10^{-2}	8.5
	319168	5.6693×10^{-1}	6.0300×10^{-2}	9.4
	617104	1.1279×10^{0}	1.0400×10^{-1}	10.8
	1228880	2.1539×10^{0}	1.9511×10^{-1}	11.0

500 We can also notice that the time required by the regular distribution case is much less than
501 the irregular distribution case, the former is around a quarter or half of the latter. The difference
502 in computer time could be caused by several reasons. First is the number of satellite points. A
503 regular meshless cloud has less satellites compared to an irregular cloud, the difference could
504 be 8 to 20 in a general 3D scenario. Having more satellites in a cloud means more work per
505 cloud. Second is the number of colors used to paint the points. Usually regular distribution only

needs two colors to organize all the points into independent groups. While irregular distribution needs more colors e.g. 9 as shown in Fig. 13 (b). More colors will request more kernels to be launched, and more kernels will cause heavier overhead cost. Of course, this could also be influenced by the data locality issue [24]. These problems will be further investigated and addressed in our future work.

6. Conclusions

A parallel LU-SGS implicit meshless method has been developed to solve complex 3D compressible flow problems on many-core GPUs. A rainbow coloring method has been proposed to organize computational points into independent groups and to parallelize the LU-SGS algorithm. A series of two- and three-dimensional test cases including compressible flows over single- and multi-element airfoils and a M6 wing have been carried out to verify the developed code. The obtained solutions agree well with experimental data and other computational results reported in the literature. Detailed analysis on the performance of the computer programs reveals that the developed implicit GPU code can achieve up to 70× speedups compared to the CPU based explicit meshless method for the 3D computation of compressible flows over a M6 wing. This demonstrates the potential of the method to be applied to solve more complex and time-consuming problems. In future, we will further develop the method to deal with challenging fluid-structure-interaction problems such as the aero-elasticity calculation of fixed-wing aircraft and rotorcraft.

Acknowledgements

This work was partially supported by Natural Science Foundation of China (No.11172134).

References

[1] R. Agarwal, Computational fluid dynamics of whole-body airfcraft, Annual Review of Fluid Mechanics, 31 (1999) 125-169. DOI: 10.1146/annurev.fluid.31.1.125.

[2] I. Goulos, V. Pachidis, Real-time aero-elasticity simulation of open rotors with slender blades for the multidisciplinary design of rotocraft, Journal of Engineering for Gas Trubines and Power, 137 (2015) 012503. DOI: 10.1115/1.4028180.

[3] S. Das, K.F. Cheung, Scattered waves and motions of marine vessels advancing in a seaway, Wave Motion, 49 (2012) 181-197. DOI: 10.1016/j.wavemoti.2011.09.003.

[4] NVIDIA, CUDA C Programming Guide, version 8.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2017 (accessed 07.05.2017).

[5] KHRONOS, OpenCL 2.1 Reference Pages. https://www.khronos.org/registry/OpenCL /sdk/2.1/docs/man/xhtml/, 2017 (accessed 07.05.2017).

[6] R. Farber, Parallel Programming with OpenACC, Elsevier Science Ltd., Amsterdam, 2017.

[7] A. Antoniou, K. Karantasis, E. Polychronopoulos, J. Ekaterinaris, Acceleration of a Finite-Difference WENO Scheme for Large-Scale Simulations on Many-Core Architectures, AIAA Paper 2010-2525. DOI: 10.2514/6. 2010-525.

[8] R. Lohner, A.T. Corrigan, K.-R. Wichmann, W. Wall, On the Achievable Speeds of Finite Difference Solvers on CPUs and GPUs, AIAA Paper 2013-2852. DOI: 10.2514/6.2013-2852.

[9] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, Journal of Computational Physics, 227 (2008) 10148-10161. DOI: 10.1016 /j.jcp.2008.08.023.

[10] E.H. Phillips, Y. Zhang, R.L. Davis, J.D. Owens, Acceleration of 2-D Compressible Flow Solvers with Graphics Processing Unit Clusters, Journal of Aerospace Computing, Information, and Communication, 8 (2011) 237-249. DOI: 10.2514/1.44909.

[11] C. Stone, E. Duque, Y. Zhang, D. Car, R. Davis, J. Owens, GPGPU parallel algorithms for structured-grid CFD codes, AIAA Paper 2011-3221. DOI: 10.2514/6.2011-3221.

[12] J.T. Liu, Z.S. Ma, S.H. Li, Y. Zhao, A GPU Accelerated Red-Black SOR Algorithm for Computational Fluid Dynamics Problems, Advanced Materials Research, 320 (2011) 335-340. DOI: 10.4028/www.scientific.net/AMR.320.335.

[13] B.J. Zimmerman, B. Wie, Graphics-Processing-Unit-Accelerated Multiphase Computational Tool for Asteroid Fragmentation/Pulverization Simulation, AIAA Journal, 55 (2017) 599-609. DOI: 10.2514/1.j055163.

[14] J.F. Remacle, R. Gandham, T. Warburton, GPU accelerated spectral finite elements on all-hex meshes, Journal of Computational Physics, 324 (2016) 246-257. DOI: 10.1016/j.jcp.2016.08.005.

[15] A. Klöckner, T. Warburton, J. Bridge, J.S. Hesthaven, Nodal discontinuous Galerkin methods on graphics processors, Journal of Computational Physics, 228 (2009) 7863-7882. DOI: 10.1016/j.jcp.2009.06.041.

[16] M. Fuhry, A. Giuliani, L. Krivodonova, Discontinuous Galerkin methods on graphics processing units for nonlinear hyperbolic conservation laws, International Journal for Numerical Methods in Fluids, 76 (2014) 982-1003. DOI: 10.1002/fld.3963.

[17] Y.D. Xia, L.X. Luo, H. Luo, OpenACC-based GPU Acceleration of a 3-D Unstructured Discontinuous Galerkin Method, AIAA Paper 2014-1129. DOI: 10.2514/6.2014-1129.

[18] J.T. Batina, A gridless Euler/Navier-Stokes solution algorithm for complex-aircraft applications, AIAA Paper 93-0333. DOI: 10.2514/6.1993-333.

[19] C.M.C. Roque, D. Cunha, C. Shu, A.J.M. Ferreira, A local radial basis functions—Finite differences technique for the analysis of composite plates, Engineering Analysis with Boundary Elements, 35 (2011) 363-374. DOI: 10.1016/j.enganabound.2010.09.012.

[20] E. K.-Y. Chiu, Q.Q. Wang, R. Hu, A. Jameson, A Conservative Mesh-Free Scheme and Generalized Framework for Conservation Laws, SIAM Journal on Scientific Computing,

580 34 (2012) A2896-A2916. DOI: 10.1137/110842740.
581 [21] E. Oñate, S. Idelsohn, O.C. Zienkiewicz, R.L. Taylor, A finite point method in
582 computational mechanics. applications to convective transport and fluid flow,
583 International Journal for Numerical Methods in Engineering, 39 (1996) 3839-3866. DOI:
584 10.1002/(sici)1097-0207(19961130)39:22<3839::aid-nme27>3.0.co;2-r.
585 [22] A. Katz, A. Jameson, Multicloud: Multigrid convergence with a meshless operator,
586 Journal of Computational Physics, 228 (2009) 5237-5250. DOI: 10.1016/j.jcp.2009.04.
587 023.
588 [23] Z.H. Ma, H. Wang, S.H. Pu, GPU computing of compressible flow problems by a meshless
589 method with space-filling curves, Journal of Computational Physics, 263 (2014) 113-135.
590 DOI: 10.1016/j.jcp.2014.01.023.
591 [24] Z.H. Ma, H. Wang, S.H. Pu, A parallel meshless dynamic cloud method on graphic
592 processing units for unsteady compressible flows past moving boundaries, Computer
593 Methods in Applied Mechanics and Engineering, 285 (2015) 146-165. DOI: 10.1016/j.
594 cma.2014.11.010.
595 [25] S. Yoon, G. Jost, S. Chang, Parallelization of Lower-Upper Symmetric Gauss-Seidel
596 Method for Chemically Reacting Flow, AIAA Paper 2005-4627. DOI: 10.2514/6.2005-
597 4627.
598 [26] D.L. Li, C.F. Xu, B. Cheng, M. Xiong, X. Gao, X.G. Deng, Performance modeling and
599 optimization of parallel LU-SGS on many-core processors for 3D high-order CFD
600 simulations, The Journal of Supercomputing, 72 (2016) 1-19. DOI: 10.1007/s11227-016-
601 1943-0.
602 [27] PGI, CUDA Fortran Programming Guide and Reference. http://www.pgroup.com/doc/
603 pgicudaforug.pdf, 2017 (accessed 07.05.2017).
604 [28] A. Katz, A. Jameson, Meshless Scheme Based on Alignment Constraints, AIAA Journal,
605 48 (2010) 2501-2511. DOI: 10.2514/1.j050127.
606 [29] A. Jameson, W. Schmidt, E.L.I. Turkel, Numerical solution of the Euler equations by finite
607 volume methods using Runge Kutta time stepping schemes, AIAA Paper 81-1259. DOI:
608 10.2514/6.1981-1259.

[30] J. Blazek, Computational Fluid Dynamics : Principles and Applications, Elsevier Science Ltd., Amsterdam, 2001.

[31] S. Yoon, A. Jameson, Lower-upper Symmetric-Gauss-Seidel method for the Euler and Navier-Stokes equations, AIAA Journal, 26 (1988) 1025-1026. DOI: 10.2514/3.10007.

[32] Y. Sato, T. Hino, K. Ohashi, Parallelization of an unstructured Navier–Stokes solver using a multi-color ordering method for OpenMP, Computers & Fluids, 88 (2013) 496-509. DOI: 10.1016/j.compfluid.2013.10.008.

[33] J.L. Zhang, H.Q. Chen, C. Cao, A graphics processing unit-accelerated meshless method for two-dimensional compressible flows, Engineering Applications of Computational Fluid Mechanics, 11(2017) (accepted). DOI: 10.1080/19942060.2017.1317027.

[34] NVIDIA, CUDA C Best Practices Guide v8.0. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2017 (accessed 07.05.2017).

[35] M.B. Azab, M.I. Mustafa, Numerical solution of inviscid transonic flow using hybrid finite volume-finite difference solution technique on unstructured grid, in: International Conference on Aerospace Science and Aviation Technology, Military Technical College, Cairo, Egypt, 2011.

[36] C. Cao, H.Q. Chen, A Preconditioned Gridless Method for Solving Euler Equations at Low Mach Numbers, Transactions of Nanjing University of Aeronautics and Astronautics, 32 (2015) 399-407. DOI: 10.16356/j.1005-1120.2015.04.399.

[37] V. Schmitt, F. Charpin, Pressure Distributions on the ONERA-M6-Wing at Transonic Mach Numbers, Experimental Data Base for Computer Program Assessment. Report of the Fluid Dynamics Panel Working Group 04, AGARD AR 138 (1979).

[38] M. Mani, J.A. Ladd, A.B. Cain, R.H. Bush, An Assessment of One- and Two-Equation Turbulence Models for Internal and External Flows, AIAA Paper 97-2010. DOI: 10.2514/6.1997- 2010.

[39] N.T. Frink, Upwind scheme for solving the Euler equations on unstructured tetrahedral meshes, AIAA Journal, 30 (1992) 70-77. DOI: 10.2514/3.10884.