



**Manchester
Metropolitan
University**

Zhang, J and Ma, Z and Chen, H and Cao, C (2018) *A GPU-accelerated implicit meshless method for compressible flows*. *Journal of Computational Physics*, 360. pp. 39-56. ISSN 0021-9991

Downloaded from: <http://e-space.mmu.ac.uk/619928/>

Publisher: Elsevier

DOI: <https://doi.org/10.1016/j.jcp.2018.01.037>

Usage rights: Creative Commons: Attribution-Noncommercial-No Derivative Works 4.0

Please cite the published version

<https://e-space.mmu.ac.uk>

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

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

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

30

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

32

33

34 1. Introduction

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

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

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

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

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

67 This paper presents an effort to develop a recently proposed GPU based two-dimensional
68 explicit meshless method for compressible flows reported by Ma et al. [23]. An efficient
69 parallel LU-SGS implicit algorithm is devised and utilized to further improve the
70 computational efficiency. The capability of the original 2D meshless code is extended to deal
71 with 3D complex problems. To resolve the inherent data dependency of the standard LU-SGS
72 method, which causes thread-racing conditions destabilizing numerical solution, a robust
73 rainbow coloring method is presented and applied to organize the computational points into
74 separate independent groups by painting neighboring points with different colors. The original
75 serial LU-SGS method is modified and parallelized accordingly to perform calculations for all
76 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**

98 2D problems. It has not been addressed by these researchers whether this method could
 99 deal with complex 3D problems. In the present work we aim at solving three-dimensional
 100 compressible flows governed by the Euler equations, of which the differential form can be
 101 expressed as

$$102 \quad \frac{\partial \mathbf{W}}{\partial t} + \nabla \cdot \mathbf{F} = 0 \quad (1)$$

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

$$105 \quad \mathbf{W} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{pmatrix} \quad \mathbf{F} = \begin{pmatrix} \rho u \\ \rho uu + p \\ \rho uv \\ \rho uw \\ u(\rho E + p) \end{pmatrix} \mathbf{i} + \begin{pmatrix} \rho v \\ \rho uv \\ \rho vv + p \\ \rho vw \\ v(\rho E + p) \end{pmatrix} \mathbf{j} + \begin{pmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho ww + p \\ w(\rho E + p) \end{pmatrix} \mathbf{k} \quad (2)$$

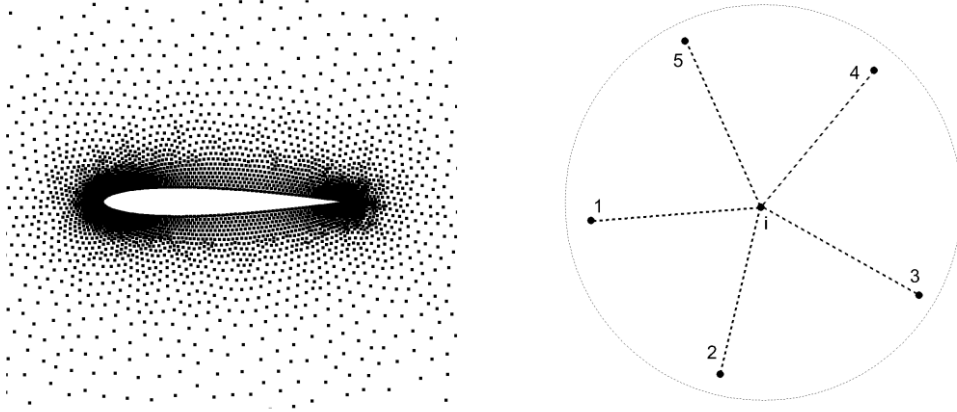
106 where ρ is the density, p is the pressure, u , v and w are the velocity components along
 107 x , y and z axes, respectively. The total energy per unit mass E is given by

$$108 \quad E = \frac{1}{\gamma - 1} \frac{p}{\rho} + \frac{1}{2} (u^2 + v^2 + w^2) \quad (3)$$

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

110 2.2 Least-square curve fit based meshless discretization

111 In meshless discretization [18-24] of the partial differential equations for CFD like
 112 Equation (1), the physical domain of the problem should be firstly discretized with scattered
 113 points. For each point in the domain as shown in Fig. 1, several surrounding points are chosen
 114 to form a local cloud of points, where the surrounding points are called as the satellites of the
 115 central point. The spatial derivatives in governing equation (1) are approximated in the
 116 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

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

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

$$A_i \bar{a}_{ij} = B_{ij} \quad (5)$$

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

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

132 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
 133 between the center point i and satellite k , $\mathbf{a}_{ij} = [\alpha_{ij} \ \beta_{ij} \ \gamma_{ij}]^T$ is the vector of derivative
 134 weight coefficients. To emphasize the contribution of certain points in the cloud, a weighting
 135 function ω is adopted, which usually takes the inverse square of its distance to the central
 136 point, with $w_{ij} = |\Delta \mathbf{r}_{ij}|^{-2}$. It can be noted that the derivative weight coefficients only depend on
 137 the nodal positions. Therefore, they are pre-computed and stored in the memory before other
 138 calculations.

139 2.3 Evaluation of the convective flux

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

$$142 \quad \nabla \cdot \mathbf{F}_i = \sum_{j \in C_i} \mathbf{F}_{ij} \cdot \mathbf{a}_{ij} \quad (7)$$

143 To estimate the convective flux $\mathbf{F}_{ij} = \mathbf{F}_{ij} \cdot \mathbf{a}_{ij}$ on the virtual edge $i-j$, the JST scheme
 144 [29] is employed, which can be expressed as

$$145 \quad \mathbf{F}_{ij} = \frac{1}{2} (\mathbf{F}(\mathbf{W}_i) + \mathbf{F}(\mathbf{W}_j)) \cdot \mathbf{a}_{ij} - \mathbf{D}_{ij} \quad (8)$$

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

$$148 \quad \mathbf{D}_i = \varepsilon_{ij}^{(2)} \lambda_{ij} (\mathbf{W}_j - \mathbf{W}_i) - \varepsilon_{ij}^{(4)} \lambda_{ij} (\nabla^2 \mathbf{W}_j - \nabla^2 \mathbf{W}_i) \quad (9)$$

149 where $\varepsilon^{(2)}$ and $\varepsilon^{(4)}$ denote the second- and forth-order adaptive coefficients, respectively.

150 ∇^2 is the Laplace operator. The spectral radius λ is also based on the meshless derivative
 151 weight coefficients, and given by

$$152 \quad \lambda = |u\alpha + v\beta + w\gamma| + \sqrt{(\alpha^2 + \beta^2 + \gamma^2) \cdot \gamma p / \rho} \quad (10)$$

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

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

158 3. Temporal discretization

159 3.1 Implicit LU-SGS scheme

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

$$163 \quad \frac{d\mathbf{W}_i}{dt} = - \sum_{j \in C_i} \mathbf{F}_{ij} \quad (11)$$

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

$$166 \quad \begin{aligned} \frac{\Delta \mathbf{W}_i^n}{\Delta t} &= - \sum_{j \in C_i} \mathbf{F}_{ij}^{n+1} = - \sum_{j \in C_i} \mathbf{F}(\mathbf{W}_i^{n+1}, \mathbf{W}_j^{n+1}) \\ &= - \sum_{j \in C_i} \left[\mathbf{F}(\mathbf{W}_i^n, \mathbf{W}_j^n) + \frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_i} \Delta \mathbf{W}_i^n + \frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_j} \Delta \mathbf{W}_j^n \right] \\ &= - \sum_{j \in C_i} \mathbf{F}_{ij}^n - \sum_{j \in C_i} \frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_i} \Delta \mathbf{W}_i^n - \sum_{j \in C_i} \frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_j} \Delta \mathbf{W}_j^n \end{aligned} \quad (12)$$

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

172
$$\left(\frac{1}{\Delta t} \mathbf{I} + \sum_{j \in C_i} \frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_i} \right) \Delta \mathbf{W}_i^n + \sum_{j \in C_i} \frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_j} \Delta \mathbf{W}_j^n = - \sum_{j \in C_i} \mathbf{F}_{ij}^n \quad (13)$$

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

175
$$\mathbf{A}(\mathbf{W}^n) \Delta \mathbf{W}^n = -\mathbf{R}^n \quad (14)$$

176 in which,

177
$$\mathbf{A} = [\mathbf{A}_{ij}] \quad \mathbf{A}_{ij} = \begin{cases} \frac{1}{\Delta t} \mathbf{I} + \sum_{k \in C_i} \frac{\partial \mathbf{F}_{ik}^n}{\partial \mathbf{W}_i} & i = j \\ \frac{\partial \mathbf{F}_{ij}^n}{\partial \mathbf{W}_j} & i \neq j \end{cases} \quad \Delta \mathbf{W}^n = \begin{bmatrix} \Delta \mathbf{W}_1^n \\ \Delta \mathbf{W}_1^n \\ \mathbf{M} \\ \Delta \mathbf{W}_N^n \end{bmatrix} \quad \mathbf{R}^n = \begin{bmatrix} \sum_{j \in C_1} \mathbf{F}_{1j}^n \\ \sum_{j \in C_2} \mathbf{F}_{2j}^n \\ \mathbf{M} \\ \sum_{j \in C_N} \mathbf{F}_{Nj}^n \end{bmatrix} \quad (15)$$

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

182
$$\begin{aligned} \text{Forward} : \Delta \mathbf{W}_i^* &= -\mathbf{A}_{ii}^{-1} \left[\mathbf{R}_i^n + \sum_{j \in C_i}^{j < i} \mathbf{A}_{ij} \Delta \mathbf{W}_j^* \right] \quad i = 1, 2, \dots, N-1, N \\ \text{Backward} : \Delta \mathbf{W}_i^n &= \Delta \mathbf{W}_i^* - \mathbf{A}_{ii}^{-1} \sum_{j \in C_i}^{j > i} \mathbf{A}_{ij} \Delta \mathbf{W}_j^n \quad i = N, N-1, \dots, 2, 1 \end{aligned} \quad (16)$$

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

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

191 3.2 Rainbow coloring method

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

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

210 meshless clouds of point into independent groups for GPU computing. The whole procedure to
 211 paint all the computational points is described in Algorithm 1. The essential criterion of this
 212 coloring algorithm is that any two neighboring points are decorated with different colors. The
 213 central point must not have the same color with any of its satellite. In the computer program, we
 214 use integer numbers to represent different colors. For example, the red color is represented by
 215 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 $color(\cdot)$ and total number of colors N_{color} .

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

216
 217 The painting procedure given in Algorithm 1 is initialized by choosing a start point v_0 in
 218 the computational domain. Once the start point is selected, the corresponding color graph will
 219 be determined accordingly. In order to know whether different choices of the start point will
 220 have significant effect on the overall computational efficiency, we have tried choosing a start
 221 point randomly and found out that its influence is almost negligible. Therefore, in the present
 222 work the first point in the global array is always selected as the start node for the sake of
 223 convenience. Examples of the generated color graphs for both regularly and irregularly
 224 distributed meshless clouds are illustrated in Fig. 2. The dashed lines in the figure are not used
 225 in calculation, they are only used here to present a clear view of neighboring points. As shown
 226 in Fig. 2(a), a simple unique graph with two colors is obtained by using Algorithm 1 for

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

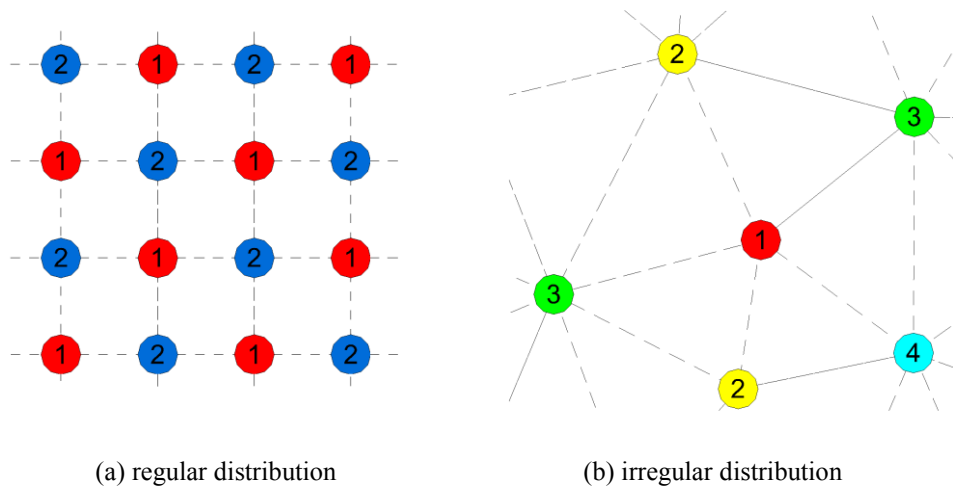


Fig. 2. Examples of color graphs.

240 3.3 Parallel LU-SGS method

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

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

Algorithm 2 The procedure of parallel LU-SGS method

1: *Forward updating:*

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

3: 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;

4: *Backward updating:*

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

6: 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;

251

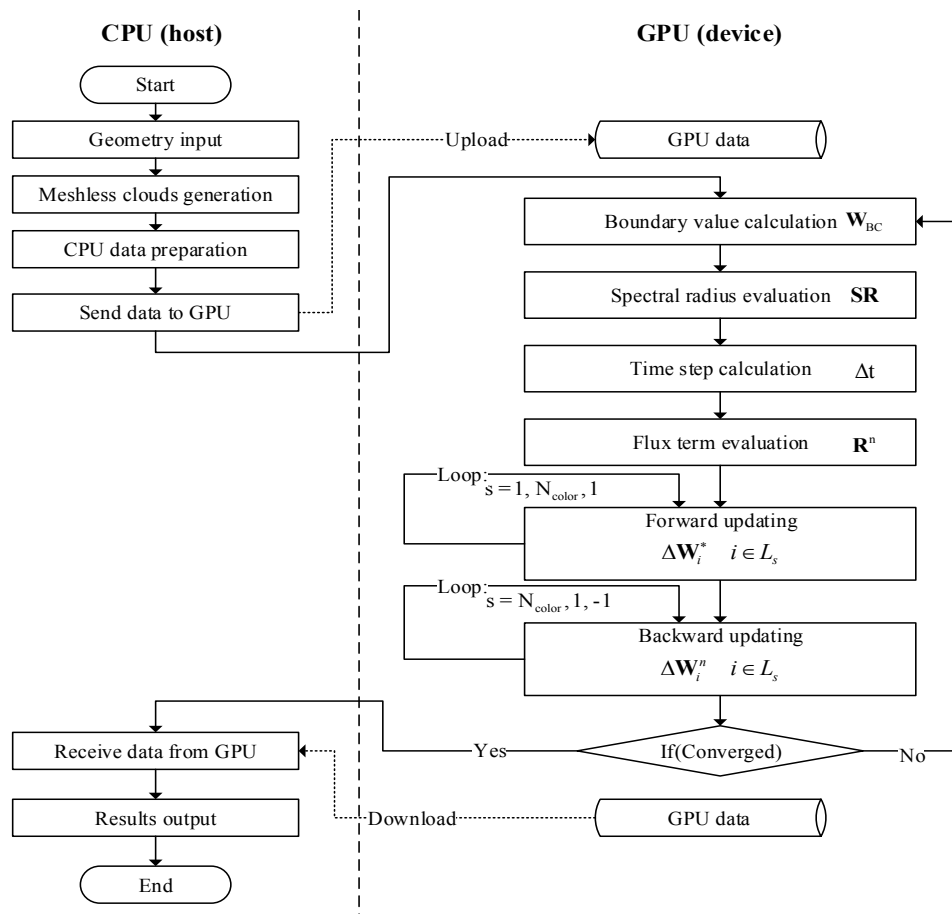
252 4. GPU implementation

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

257 4.1 Program framework

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

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



270
271

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

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

278 4.2 CUDA kernel functions

279 In the present work, the CUDA functions developed for the time marching procedure are
 280 categorized into three types including **internal**, **boundary** and **update kernels** according to the
 281 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

```

282

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

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

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

291 **Boundary kernels** are designed to enforce boundary conditions including no-penetration
 292 wall, symmetric plane and non-reflective far field in the present work. We noted that if the
 293 near-boundary points are treated differently with the field points, the efficiency of the related
 294 kernels will be excessively degraded due to the divergence of thread branch. In the present work,
 295 similar treatment of both near-boundary and field points is adopted to avoid the branch
 296 divergence by introducing ghost points to implement boundary conditions, which is carried out
 297 by a specific kernel. An example code of the boundary kernel is given in Listing 2, in which
 298 each thread evaluates the boundary values for one ghost point. The variable nBC is the total
 299 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

```

300

301

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

302

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

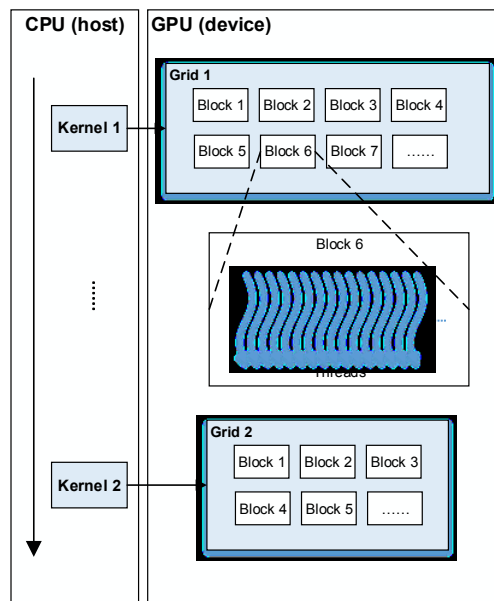


Fig. 4. The thread hierarchy of CUDA kernels.

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

324
$$gridDim = (nTotalThread + blockDim - 1) / blockDim \quad (17)$$

325 where $nTotalThread$ represents the total number of threads.

326 4.3 Device memory management

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

332 In this paper, the thread index is used to build the mapping relationships between the
333 threads of the kernels and the corresponding computing data stored on the graphics card for
334 memory addressing. As presented in Listings 1, 2 and 3, three build-in variables, $blockDim$,
335 $blockIdx$ and $threadIdx$, related to the thread hierarchy are used to compute the thread index.
336 The utilizing of these important variables can be found in article [4] for details. When fetching
337 data from or writing them to the global memory, coalesced memory access is the ideal pattern
338 [34]. This pattern is adopted in the present work so that all the threads in a half wrap map/access
339 the global memory simultaneously with respect to the center of a meshless cloud. In reality, this
340 means consecutive thread access consecutive memory addresses [33, 34].

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

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

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

354 5. Numerical results and analysis

355 **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

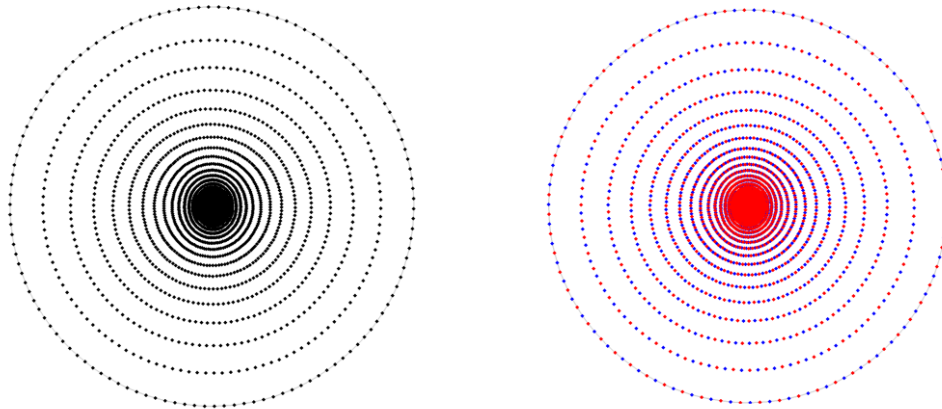
356 A set of 2D and 3D inviscid compressible flows over aerodynamic bodies, for which
 357 regularly or irregularly distributed meshless clouds of pointed are used, have been carried out to
 358 verify the developed code. To evaluate the overall computing performance, we have

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 $\alpha = 1.25^\circ$. 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.



380

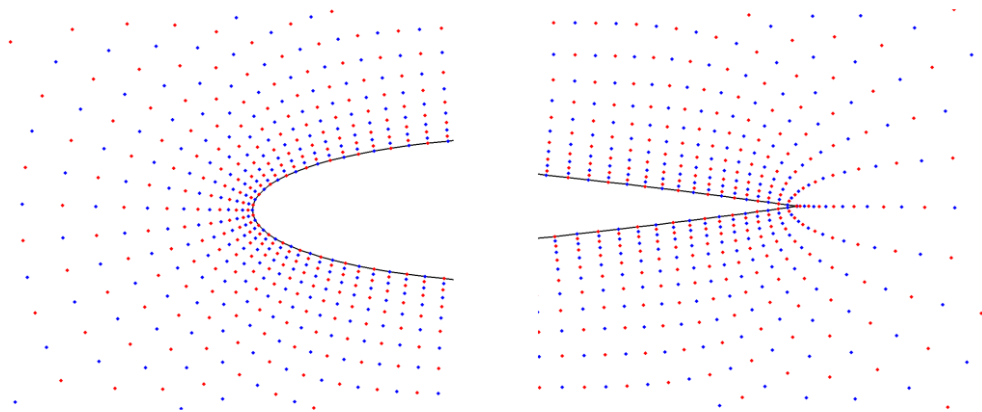
381

(a) meshless cloud

(b) color graph

382

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



383

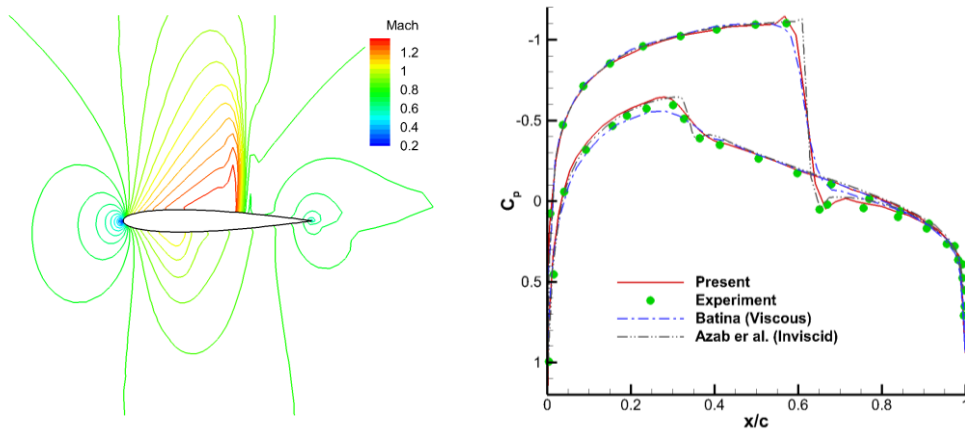
384

(a) the leading edge

(b) the trailing edge

385

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



386

387

(a) contours of Mach number

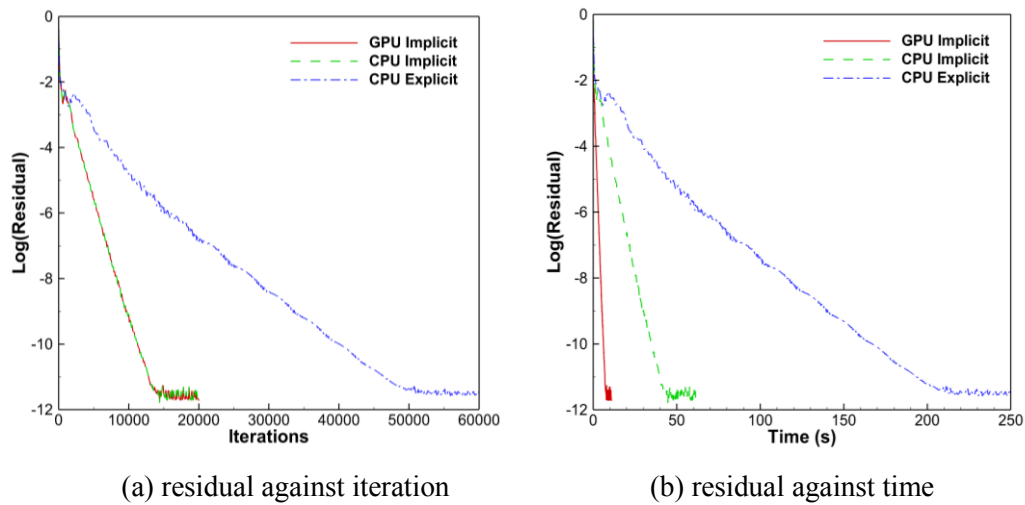
(b) plots of pressure coefficient

388

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

389

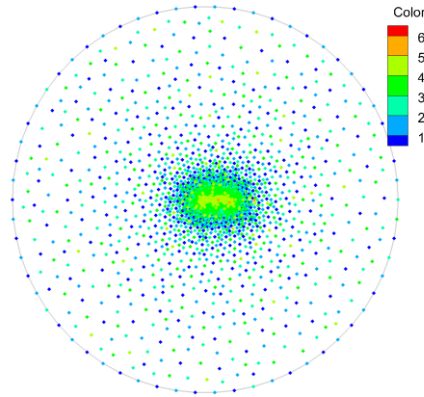
390 The histories of residual convergence with respect to iteration and wall time are shown in
 391 Fig. 8. It can be noted that the numbers of iteration of the implicit algorithms used to achieve the
 392 convergence are only a quarter of the explicit method. The implicit methods on the CPU and
 393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



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

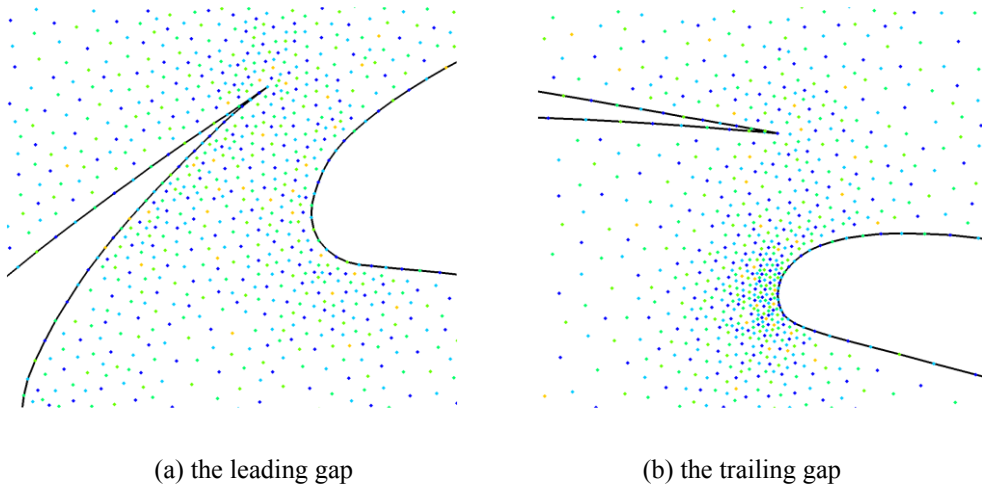
399 5.2 Subsonic flow past a three-element airfoil

400 Two-dimensional inviscid compressible flow past a three-element airfoil with $M_\infty = 0.2$
 401 and $\alpha = 1.25^\circ$ is then simulated to test the performance of the algorithm using irregularly
 402 distributed meshless clouds of points. There are 9592 points irregularly distributed in the
 403 computational domain as shown in Fig. 9. By adopting Algorithm 1, six colors are requested to
 404 paint all the points. The detailed color graphs at the leading and trailing gaps are presented in
 405 Fig. 10. Specifically, the numbers of points in each of the six color groups are 2600, 2525, 2314,
 406 1818, 332 and 3, respectively.



407
408

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



409
410

(a) the leading gap

(b) the trailing gap

411

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

412

Fig. 11 shows the computed results including the Mach number contours and the pressure

413

coefficient plots, which are close to the experimental data and other numerical results reported

414

in the literature [36]. The histories of convergence in terms of iteration and time are presented in

415

Fig. 12. It can be seen from Fig. 12(a) that the numbers of iterations needed to achieve the

416

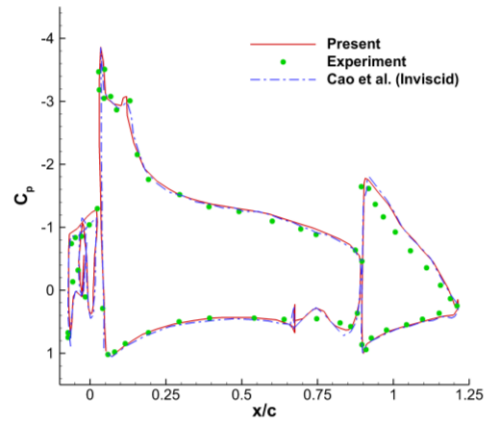
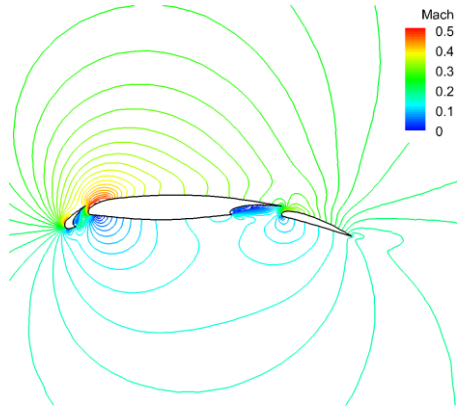
convergence for implicit algorithms are only about one-eighth of the explicit method. Once

417

again, we can notice that the implicit methods could effectively reduce the computing time

418

compared to the explicit method.



419

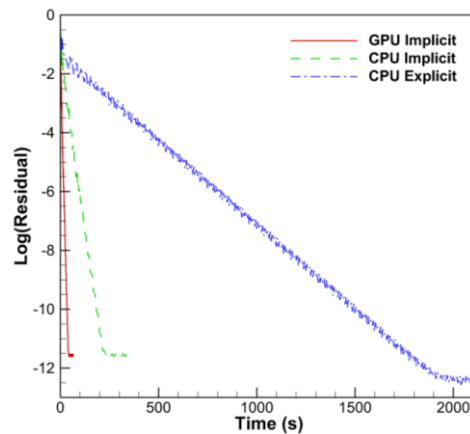
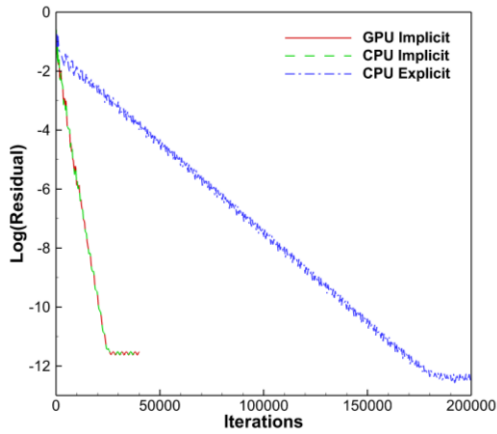
(a) contours of Mach number

(b) plots of pressure coefficient

420

421

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



422

(a) residual against iteration

(b) residual against time

423

424

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

425 5.3 Transonic flow past a M6 wing

426 After testing two-dimensional problems, the develop code is used to accelerate the

427 simulation of complex flows over three-dimensional aerodynamic bodies. Here, a typical

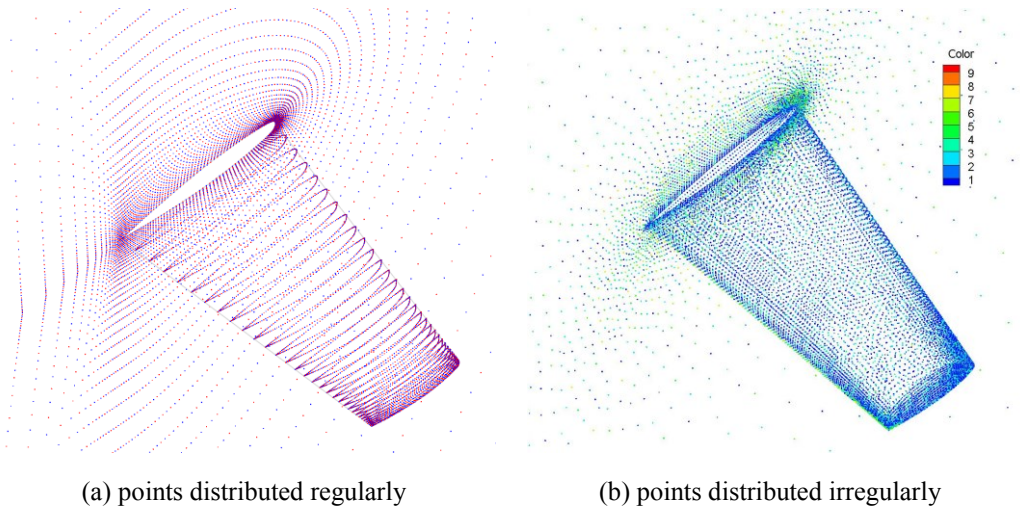
428 transonic flow problem for the ONERA M6 wing with the Mach number $M_\infty = 0.84$ and the

429 angle of attack $\alpha = 3.06^\circ$ is tested with regularly and irregularly distributed points. Fig. 13

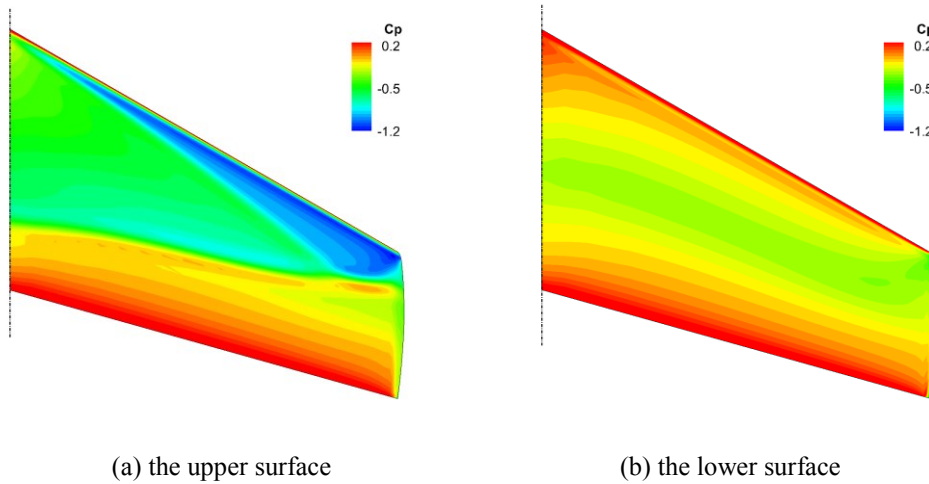
430 shows the points distributed on the wing surface and the symmetric plane. It can be noted that

431 only two colors are used for the regular distribution while nine colors are needed to paint the

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



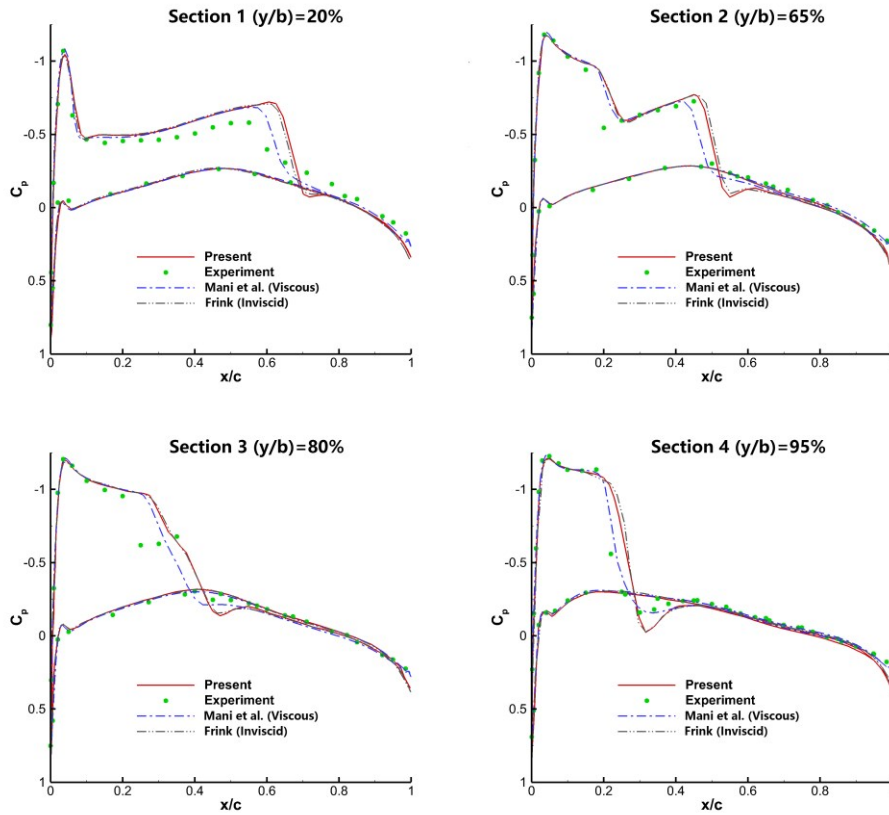
437 **Fig. 13.** The whole meshless cloud and color graph around the M6 wing.



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

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

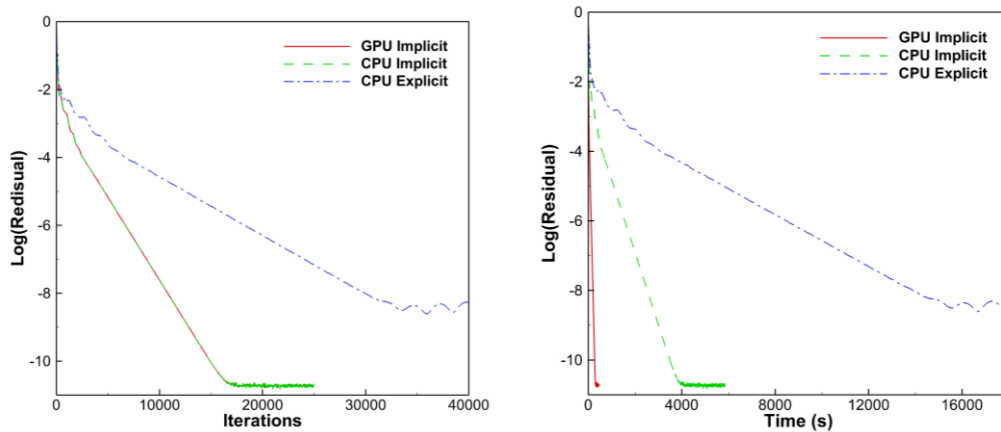
446 reference data.



447

448

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



449

450

(a) residual against iteration

(b) residual against time

451

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

452

Fig. 16 shows the histories of convergence obtained by the CE, CI and GI codes. It can be

453

seen from Fig. 16(a) that for achieving the convergence, the numbers of iterations used by

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

456 5.4 Performance analysis

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

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

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

482 **Table 2** Computing time cost.

Case	Number of points	Computing time (seconds)			
		CPU explicit	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

483
 484 **Table 3** Speedup. CE: CPU explicit; CI: CPU implicit; GE: GPU explicit; GI: GPU implicit.

Case	Number of points	Speedup				
		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

485 5.5 Size effect

486 For the first and second 2D cases, we only obtain a relatively small speedup in the range of
 487 5 to 6 with respect to GI/CI. For the 3D case, the speedup rises to 12.50. The similar situation
 488 occurs for the explicit code on GPU with respect to GE/CE. In fact, the numbers of points used
 489 for the first and second cases are less than 10,000, which are not large enough to keep the GPU
 490 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.

Case	Number of points	Computing time per iteration (seconds)		Speedup
		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

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

511 **6. Conclusions**

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

525 **Acknowledgements**

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

528 **References**

- 529 [1] R. Agarwal, Computational fluid dynamics of whole-body aircraft, Annual Review of
530 Fluid Mechanics, 31 (1999) 125-169. DOI: 10.1146/annurev.fluid.31.1.125.
- 531 [2] I. Goulos, V. Pachidis, Real-time aero-elasticity simulation of open rotors with slender
532 blades for the multidisciplinary design of rotocraft, Journal of Engineering for Gas
533 Turbines and Power, 137 (2015) 012503. DOI: 10.1115/1.4028180.
- 534 [3] S. Das, K.F. Cheung, Scattered waves and motions of marine vessels advancing in a
535 seaway, Wave Motion, 49 (2012) 181-197. DOI: 10.1016/j.wavemoti.2011.09.003.
- 536 [4] NVIDIA, CUDA C Programming Guide, version 8.0. [http://docs.nvidia.com/cuda/cuda-c-](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)
537 [programming-guide/index.html](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html), 2017 (accessed 07.05.2017).
- 538 [5] KHRONOS, OpenCL 2.1 Reference Pages. [https://www.khronos.org/registry/OpenCL](https://www.khronos.org/registry/OpenCL/sdk/2.1/docs/man/xhtml/)
539 [/sdk/2.1/docs/man/xhtml/](https://www.khronos.org/registry/OpenCL/sdk/2.1/docs/man/xhtml/), 2017 (accessed 07.05.2017).
- 540 [6] R. Farber, Parallel Programming with OpenACC, Elsevier Science Ltd., Amsterdam,
541 2017.
- 542 [7] A. Antoniou, K. Karantasis, E. Polychronopoulos, J. Ekaterinaris, Acceleration of a
543 Finite-Difference WENO Scheme for Large-Scale Simulations on Many-Core
544 Architectures, AIAA Paper 2010-2525. DOI: 10.2514/6.2010-525.
- 545 [8] R. Lohner, A.T. Corrigan, K.-R. Wichmann, W. Wall, On the Achievable Speeds of Finite
546 Difference Solvers on CPUs and GPUs, AIAA Paper 2013-2852. DOI: 10.2514/6.2013-
547 2852.
- 548 [9] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle
549 using a GPU, Journal of Computational Physics, 227 (2008) 10148-10161. DOI: 10.1016
550 /j.jcp.2008.08.023.

-
- 551 [10] E.H. Phillips, Y. Zhang, R.L. Davis, J.D. Owens, Acceleration of 2-D Compressible Flow
552 Solvers with Graphics Processing Unit Clusters, *Journal of Aerospace Computing,*
553 *Information, and Communication*, 8 (2011) 237-249. DOI: 10.2514/1.44909.
- 554 [11] C. Stone, E. Duque, Y. Zhang, D. Car, R. Davis, J. Owens, GPGPU parallel algorithms for
555 structured-grid CFD codes, *AIAA Paper 2011-3221*. DOI: 10.2514/6.2011-3221.
- 556 [12] J.T. Liu, Z.S. Ma, S.H. Li, Y. Zhao, A GPU Accelerated Red-Black SOR Algorithm for
557 Computational Fluid Dynamics Problems, *Advanced Materials Research*, 320 (2011)
558 335-340. DOI: 10.4028/www.scientific.net/AMR.320.335.
- 559 [13] B.J. Zimmerman, B. Wie, Graphics-Processing-Unit-Accelerated Multiphase
560 Computational Tool for Asteroid Fragmentation/Pulverization Simulation, *AIAA Journal*,
561 55 (2017) 599-609. DOI: 10.2514/1.j055163.
- 562 [14] J.F. Remacle, R. Gandham, T. Warburton, GPU accelerated spectral finite elements on
563 all-hex meshes, *Journal of Computational Physics*, 324 (2016) 246-257. DOI: 10.1016/j.
564 jcp.2016.08.005.
- 565 [15] A. Klöckner, T. Warburton, J. Bridge, J.S. Hesthaven, Nodal discontinuous Galerkin
566 methods on graphics processors, *Journal of Computational Physics*, 228 (2009) 7863-7882.
567 DOI: 10.1016/j.jcp.2009.06.041.
- 568 [16] M. Fuhry, A. Giuliani, L. Krivodonova, Discontinuous Galerkin methods on graphics
569 processing units for nonlinear hyperbolic conservation laws, *International Journal for*
570 *Numerical Methods in Fluids*, 76 (2014) 982-1003. DOI: 10.1002/fld.3963.
- 571 [17] Y.D. Xia, L.X. Luo, H. Luo, OpenACC-based GPU Acceleration of a 3-D Unstructured
572 Discontinuous Galerkin Method, *AIAA Paper 2014-1129*. DOI: 10.2514/6.2014-1129.
- 573 [18] J.T. Batina, A gridless Euler/Navier-Stokes solution algorithm for complex-aircraft
574 applications, *AIAA Paper 93-0333*. DOI: 10.2514/6.1993-333.
- 575 [19] C.M.C. Roque, D. Cunha, C. Shu, A.J.M. Ferreira, A local radial basis functions—Finite
576 differences technique for the analysis of composite plates, *Engineering Analysis with*
577 *Boundary Elements*, 35 (2011) 363-374. DOI: 10.1016/j.enganabound.2010.09.012.
- 578 [20] E. K.-Y. Chiu, Q.Q. Wang, R. Hu, A. Jameson, A Conservative Mesh-Free Scheme and
579 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](http://www.pgroup.com/doc/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.

-
- 609 [30] J. Blazek, Computational Fluid Dynamics : Principles and Applications, Elsevier Science
610 Ltd., Amsterdam, 2001.
- 611 [31] S. Yoon, A. Jameson, Lower-upper Symmetric-Gauss-Seidel method for the Euler and
612 Navier-Stokes equations, AIAA Journal, 26 (1988) 1025-1026. DOI: 10.2514/3.10007.
- 613 [32] Y. Sato, T. Hino, K. Ohashi, Parallelization of an unstructured Navier–Stokes solver using
614 a multi-color ordering method for OpenMP, Computers & Fluids, 88 (2013) 496-509. DOI:
615 10.1016/j.compfluid.2013.10.008.
- 616 [33] J.L. Zhang, H.Q. Chen, C. Cao, A graphics processing unit-accelerated meshless method
617 for two-dimensional compressible flows, Engineering Applications of Computational
618 Fluid Mechanics, 11(2017) (accepted). DOI: 10.1080/19942060.2017.1317027.
- 619 [34] NVIDIA, CUDA C Best Practices Guide v8.0. [http://docs.nvidia.com/cuda/cuda-c-best-](http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html)
620 [practices-guide/index.html](http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html), 2017 (accessed 07.05.2017).
- 621 [35] M.B. Azab, M.I. Mustafa, Numerical solution of inviscid transonic flow using hybrid
622 finite volume-finite difference solution technique on unstructured grid, in: International
623 Conference on Aerospace Science and Aviation Technology, Military Technical College,
624 Cairo, Egypt, 2011.
- 625 [36] C. Cao, H.Q. Chen, A Preconditioned Gridless Method for Solving Euler Equations at Low
626 Mach Numbers, Transactions of Nanjing University of Aeronautics and Astronautics, 32
627 (2015) 399-407. DOI: 10.16356/j.1005-1120.2015.04.399.
- 628 [37] V. Schmitt, F. Charpin, Pressure Distributions on the ONERA-M6-Wing at Transonic
629 Mach Numbers, Experimental Data Base for Computer Program Assessment. Report of
630 the Fluid Dynamics Panel Working Group 04, AGARD AR 138 (1979).
- 631 [38] M. Mani, J.A. Ladd, A.B. Cain, R.H. Bush, An Assessment of One- and Two-Equation
632 Turbulence Models for Internal and External Flows, AIAA Paper 97-2010. DOI:
633 10.2514/6.1997- 2010.
- 634 [39] N.T. Frink, Upwind scheme for solving the Euler equations on unstructured tetrahedral
635 meshes, AIAA Journal, 30 (1992) 70-77. DOI: 10.2514/3.10884.
- 636