A graphics processing unit-accelerated meshless method for two-dimensional compressible flows

Jiale Zhang, Hongquan Chen and Cheng Cao

College of Aerospace Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing, China

ABSTRACT
A graphics processing unit (GPU)-accelerated meshless method is presented for solving two-dimensional compressible flows over aerodynamic bodies. The Compute Unified Device Architecture (CUDA) Fortran programming model is employed to port the meshless method from central processing unit to GPU as a way of achieving efficiency, which involves implementation of CUDA kernels and management of data storage structure and thread hierarchy. The CUDA kernel subroutines are designed to meet with the point-based computing of the meshless method. The corresponding point-based data structure and thread hierarchy are constructed or manipulated in the paper by presenting two specific GPU implementations of the meshless method, which are developed for solving Navier–Stokes equations. The Jameson–Schmidt–Turkel scheme is used to estimate the flux terms of the Navier–Stokes equations and an explicit four-stage Runge–Kutta scheme is applied to update the solution at time level. After tuning the performances of the resulting two GPU-accelerated meshless solvers by changing the number of threads in a block, a set of typical flows over aerodynamic bodies are simulated for validation. Numerical results are shown in a comparison with available experimental data or computational values that appear in extant literature with an analysis of code performance. This reveals that the cost of computing time of the presented test cases is significantly reduced for both solvers without losing accuracy, while impressive speedups up to 64 times are achieved due to careful management of memory access.

ARTICLE HISTORY
Received 19 July 2016
Accepted 5 April 2017

KEYWORDS
GPU parallel computing; meshless method; Navier–Stokes equations; CUDA Fortran

1. Introduction
In recent years, a novel graphics processing unit (GPU)-based programming technique has provided scientists and engineers with an alternative means to develop computational fluid dynamics (CFD) software with high GPU-accelerated algorithms instead of traditional central processing unit (CPU)-based programming. Modern GPUs can perform orders of magnitude faster in memory bandwidth and float-point operations than standard CPUs can, since hundreds of processor cores are contained in a GPU for specialized graphics rendering in parallel. As illustrated in Figure 1, the growing gap in peak performance, measured in floating point operations per second (FLOPS), between NVIDIA GPU and Intel CPU has increased rapidly over the last 10 years (NVIDIA, 2015). The peak value of the GPU performance is tens of times faster than that of the corresponding CPU, suggesting that the computational efficiency can be improved greatly when implementing the CFD algorithms on GPU.

In early years, due to the shortage of unified programming models, carrying out parallel computing on GPUs was a complicated exercise that needed a good command of low-level graphics programming. Recently, the development of unified programming models, including Compute Unified Device Architecture (CUDA), OpenCL and OpenAcc, etc., has allowed programmers to use high-level programming languages in their GPU applications. The CFD community has been quick to recognize the potential benefits, and many successful works on GPU acceleration have emerged. Among these works, the numerical methods that have succeeded in using GPU parallel computing include finite difference methods (Elsen, Legresley, & Darve, 2008), finite volume methods (Brandvik & Pullan, 2008; Luo, Edwards, Luo, & Mueller, 2014), discontinuous Galerkin methods (Xia, Luo, Luo, & Edwards, 2014), Lattice Boltzmann methods (Guzel & Akgun, 2014; Löhner, Corrigan, Wichmann, & Wall, 2014) and so on. It can be learned from these articles that speedups of dozens of times can usually be
achieved with GPU architecture as compared with a serial CPU implementation. It can be also noted that most of these procedures are carried out using mesh-based methods. The performance of meshless methods on GPU will be investigated in this paper.

Compared with mesh-based methods, meshless methods, such as the least-squares fit-based meshless method (Batina, 1993), radius function-based meshless method (Roque, Cunha, Shu, & Ferreira, 2011) and conservative meshless method (Chiu, Wang, Hu, & Jameson, 2012), only need clouds of points, instead of mesh, to discretize the computational domain, which provides great flexibility to accommodate aerodynamic bodies with complicated geometries. Recently, in order to improve accuracy and efficiency, adaptive meshless methods based on point adding/removing (Onate, Idelsohn, Zienkiewicz, & Taylor, 1996) and point moving (Ma, Chen, & Zhou, 2008; Zhou & Xu, 2010), implicit meshless methods (Chen & Shu, 2005; Singh, Ramesh, & Balakrishnan, 2015) and multi-cloud meshless methods (Katz & Jameson, 2009) have been reported on, with successful applications. However, their computational efficiencies remain largely uncompetitive with mesh-based methods. Hence, it is necessary to develop a GPU-based meshless method to achieve further accelerations. Recently, a GPU-based meshless method (Ma, Wang, & Pu, 2014) was presented and successfully applied to accelerating simulations of compressible inviscid flows, which was achieved by applying a Hilbert space-filling curve to renumber the points to enhance performance of the global memory address. However, few attempts to simulate viscous flows and the direct management of kernel algorithm to achieve appropriate global memory address are known. Therefore, further researches related to solving more complex aerodynamic problems is needed for GPU-based meshless methods.

In view of the abovementioned flexibility of meshless methods, and the excellent efficiency of GPU computing, a GPU-based meshless method is developed in this paper to accelerate the solution of Navier–Stokes equations. The Jameson–Schmidt–Turkel (JST) scheme is used to estimate the flux terms of the Navier–Stokes equations, and an explicit Runge–Kutta time-marching scheme, which meets the requirement of data parallelization for GPU computing, is adopted within the meshless method. The CPU-based meshless solver is ported to the GPU using the CUDA Fortran programming model as a way of achieving efficiency. Some key implementation techniques, such as CUDA kernel subroutines, and the corresponding structures of data storage and thread hierarchy are addressed to fit the point-based computing meshless method. In view of the data race condition, the performance of the present point-based computing approach is analyzed based on a comparison of the existing edge-based one. The performance of the resulting GPU-accelerated meshless method is first tested by changing the number of threads in a block to obtain an appropriate size of thread block used for simulations of a set of typical validating flows over aerodynamic bodies. The numerical results are presented in comparison with available experimental data or computational values that appear in extant literature with an analysis of code performance.

2. Numerical model of the meshless method

Before implementation the meshless method on GPU, we provide a brief description of the original meshless method (Löhner, Sacco, Onate, & Idelsohn, 2002) for solving the Navier–Stokes equations.

2.1. Governing equations

In this study, compressible flows are governed by the two-dimensional laminar Navier–Stokes equations, which can be expressed in differential form as

\[
\frac{\partial \mathbf{W}}{\partial t} + \nabla \cdot (\mathbf{F} - \mathbf{F}_v) = 0
\]

where \( \mathbf{W} \) is the vector of conserved variables, and \( \mathbf{F} \) and \( \mathbf{F}_v \) are the convective and viscous flux terms, respectively, with

\[
\mathbf{W} = \begin{pmatrix}
\rho \\
\rho u \\
\rho v \\
\rho E
\end{pmatrix}, \quad \mathbf{F} = \begin{pmatrix}
\rho u \\
\rho uu + p \\
\rho uv \\
\rho E + p
\end{pmatrix}, \quad \mathbf{F}_v = \begin{pmatrix}
0 \\
0 \\
\tau_{xx} \\
\tau_{xy} - \rho_s
\end{pmatrix}
\]
In the viscous flux terms, the components of shear stresses and heat conduction are defined by

\[ \tau_{xx} = \frac{2}{3} \mu \left( \frac{\partial u}{\partial x} - \frac{\partial v}{\partial y} \right), \quad \tau_{yy} = \frac{2}{3} \mu \left( \frac{\partial v}{\partial y} - \frac{\partial u}{\partial x} \right), \]

\[ \tau_{xy} = \tau_{yx} = \mu \left( \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \]

(3)

\[ q_x = -\gamma \frac{(\gamma - 1)}{(\gamma - 1)} \Pr \frac{\mu}{\rho} \frac{\partial}{\partial x} \left( \frac{p}{\rho} \right), \]

\[ q_y = -\gamma \frac{(\gamma - 1)}{(\gamma - 1)} \Pr \frac{\mu}{\rho} \frac{\partial}{\partial y} \left( \frac{p}{\rho} \right) \]

(4)

In these equations, \( \rho \) is the density, \( p \) is the pressure, and \( u \) and \( v \) are the velocity components along the \( x \) and \( y \) axes, respectively. \( E \) is the total energy, and satisfies the following equation for idea gas:

\[ E = \frac{1}{\gamma - 1} \frac{p}{\rho} + \frac{1}{2} (u^2 + v^2) \]

(5)

\( \gamma \) is the ratio of specific heat coefficient, \( \Pr \) is the laminar Prandtl number. For air, \( \gamma = 1.4 \) and \( \Pr = 0.72 \). Additionally, the viscosity coefficient, \( \mu \), can be calculated using the Sutherland formula (Sutherland, 1893).

### 2.2. Least-squares fit-based meshless method

In meshless methods (Batina, 1993; Chen & Shu, 2005; Katz & Jameson, 2010), scattered points are first distributed in the physical domain to be solved, as shown in Figure 2(a). For each node in the domain, several nearest neighbors are selected to form a local cloud of points, establishing connectivity throughout the domain. Figure 2(b) shows a typical local cloud of points, in which the node \( i \) is named the center and the surrounding nodes are called the satellites. In order to evaluate the spatial derivatives in the partial differential equations of CFD (like Equations (1)), local Taylor series expansions and least-squares fitting are implemented onto each local cloud (Katz & Jameson, 2010; Löhner et al., 2002). For a sufficiently differentiable function \( \phi(x, y) \), the first-order Taylor series on a given cloud of point \( C(i) \) can be expressed as

\[ \phi_k \cong \phi_i + \Delta \vec{r}_{ik} \cdot \nabla \phi_i \quad (k = 1, \ldots, n) \]

(6)

where \( n \) denotes the number of the satellites, \( \Delta \vec{r}_{ik} = \vec{r}_k - \vec{r}_i \) is the distance vector from node \( i \) to node \( k \). \( \nabla \phi_i \) is the estimate of the spatial derivatives on the node \( i \) for which we seek. Hence, the total error between the exact and approximate values can be estimated by the minimum of the function

\[ f(\nabla \phi_i) = \sum_{k=1}^{n} \omega_{ik}(\Delta \vec{r}_{ik} \cdot \nabla \phi_i - \Delta \phi_{ik})^2 \]

(7)

where \( \Delta \phi_{ik} = \phi_k - \phi_i \) is the value difference of function \( \phi \) between node \( i \) and node \( k \). The weighting function \( \omega_{ik} \) used to emphasize the contribution of certain satellites in the local cloud usually takes the inverse square of its distance to node \( i \), with \( \omega_{ik} = |\Delta \vec{r}_{ik}|^{-2} \).

Minimization of the function \( f(\nabla \phi_i) \) results in linear equations, which can be expressed as

\[ \mathbf{A} \nabla \phi_i = \mathbf{B} \Delta \phi \]

(8)

where \( \Delta \phi = [\Delta \phi_{i1} \hspace{1cm} \Delta \phi_{i2} \hspace{1cm} \cdots \hspace{1cm} \Delta \phi_{in}]^T \) is the known vector of undivided differences. For the two-dimensional linear least-squares approximation, the \( 2 \times 2 \) matrix
The inviscid flux terms of the discretized equations mentioned are evaluated using the central scheme with artificial dissipative terms, which can be expressed as

\[ \sum_{k=1}^{n} \tilde{F}_{ik} \cdot \tilde{a}_{ik} = \sum_{k=1}^{n} \frac{1}{2} (\tilde{F}(W_i) + \tilde{F}(W_k)) \cdot \tilde{a}_{ik} - D_i \]  

(15)

where the artificial dissipation term, \( D_i \), is given by

\[ D_i = \sum_{k=1}^{n} \lambda_{ik} [\varepsilon(2)_{ik}(W_k - W_i) - \varepsilon(4)_{ik}(\nabla^2 W_k - \nabla^2 W_i)] \]  

(16)

Here, \( \varepsilon(2) \) and \( \varepsilon(4) \) are adaptive coefficients (Jameson, Schmidt, & Turkel, 1981), and \( \lambda \) denotes the spectral radius of the convective flux Jacobian, also based on the derivative weight coefficients associated with the cloud of points, \( C(i) \), given by

\[ \lambda = |u\alpha + v\beta| + \sqrt{\frac{\rho^2}{\gamma} (\alpha^2 + \beta^2)} \]  

(17)

The viscous flux terms in Equation (14) are evaluated directly using the average of their values at the midpoint of the edge, \( i - k \), by

\[ \tilde{F}_{v,ik} = \frac{1}{2} (\tilde{F}_v(W_i) + \tilde{F}_v(W_k)) \]  

(18)

Additionally, in order to obtain the steady solution, an explicit four-stage Runge–Kutta time-marching scheme (Jameson et al., 1981) is adopted:

\[ \begin{align*}
W_i^{(0)} &= W_i^n \\
W_i^{(m)} &= W_i^{(0)} - a_m\Delta t_i R_i^{(m-1)} & (m = 1, 2, 3, 4) \\
W_i^{n+1} &= W_i^{(4)}
\end{align*} \]  

(19)

where \( a_1 = 1/4, a_2 = 1/3, a_3 = 1/2 \) and \( a_4 = 1 \). The time step, \( \Delta t_i \), is calculated by the local time technique as follows:

\[ \Delta t_i = \frac{CFL}{\sum_{k=1}^{n} \lambda_{ik}} \]  

(20)

### 3. GPU implementation of the meshless method

Regarding the underlying unified programming models for GPU computing, CUDA is the most widely used model since it provides developers with a complete software and hardware architecture for parallel computing on their GPUs. In present work, we adopted CUDA Fortran (Rutjes & Fatica, 2014) as our main programming model to develop GPU-based meshless algorithms. Some key techniques within the implementation of GPU-based meshless algorithms – such as CUDA kernel subroutines, the data storage structure and thread hierarchy – will be discussed in this section after a brief review of CUDA.
3.1. CUDA programming model

In present work, a NVIDIA GTX TITAN GPU was used as the computing platform and the CUDA programming model (NVIDIA, 2015) was employed for developing the GPU codes by using the CUDA Fortran programming language (Rüetsch & Fatica, 2014). The GTX TITAN GPU is based on Kepler architecture (NVIDIA, 2014), which contains a total of 14 streaming multiprocessors (SM) and each SM contains 192 CUDA cores, as shown in Figure 3(a). When executing a parallel task on a GPU (called a kernel), a double-layer-based thread hierarchy must be assigned: all threads are organized into a set of thread blocks, and all blocks are then gathered into a thread grid, as shown in Figure 3(b). Each block in a specified grid contains the same number of threads, which will be sent to different cores in the same SM and executed in parallel, while different blocks can also be executed on different SMs in parallel. The memory hierarchy, which includes variant memories and their features, should also be considered for GPU computing. Each thread has its own private registers and local memory, threads in same block have shared memory for sharing data, and the major computing data of all threads from different blocks should be stored in global memory. The scarce registers and shared memory are located in the fast-access GPU chip, which is many times faster than the slow-access global memory located outside of the chip, as also shown in Figure 3(a). It should be emphasized that when using GPU for parallel computing, the challenge is not only parallelization of the code, but also optimization of the access to device memory by making the best use of registers and shared memory (Corrigan, Camelli, Löhner, & Wallin, 2011; Julien & Inanc, 2009). Meanwhile, it should be pointed out that the shared memory is difficult to use for an irregular meshless structure of clouds of points due to the unpredictable memory access pattern (see Ma et al. (2014) for details); therefore, the present work is mainly focused on the best use of registers by searching for an appropriate size of thread block used for the meshless solver; this will be discussed in section 4.2.2.

3.2. CUDA implementation

3.2.1. Summary of GPU-based meshless solver

For a general GPU-based solver, not all computing tasks should be executed on the GPU side: the most time-consuming part (such as time-marching procedure) is posted to the GPU and executed in parallel, while the other parts related to pre- or post-processing are kept on
Figure 4. A general procedure of GPU-based meshless solver.

the CPU side. This is because numerous arithmetic operations are performed well by the GPU, while it performs poorly for pre- or post-processing tasks that require a great deal of logical judgment and branch structures. In our implementation, the Runge–Kutta time-marching scheme is adopted; this is the most time-consuming part of the whole program and therefore it is ported to the GPU with the use of CUDA Fortran, while the other parts of the code are kept on the CPU. We will now take a look at the time-marching procedure of the Runge–Kutta scheme, which involves calculations of time step, boundary conditions, derivatives of flow variables, flux terms, residuals and solution updating. Data communication between the CPU and GPU only occurs when the $L_2$ norms of the residuals are sent back to the host to monitor the convergence of the solver. The whole procedure of the GPU-based meshless solver related to the calculations mentioned can be described as shown in Figure 4.

3.2.2. Point-based algorithms

As mentioned above, Ma et al. (2014) renumbered the points of meshless clouds to improve the performance of the global memory address; here, we manage the kernel algorithm directly by developing a point-based algorithm to achieve an appropriate global memory address. In this way, the GPU kernels can own the same thread structure, and the new measurement can therefore be taken by combining part of the kernels to enable further improvement of the performance.

In order to take different measures for conveniently coding specific GPU kernels, it is necessary to classify the calculations involved (as described in Figure 4) into three categories based on their locations of memory access. The first category, namely point-based calculation, is the calculation with a loop structure over all scattered points, where only the current values of flow variables at the point are needed. In the present work, the calculations of $L_2$ residuals and solution updating are of this type. The second category, namely edge-based calculation, is the calculation with a loop structure now over all internal virtual edges (here, edges refer to the lines between the points, as shown in Figure 2(b)), where the metric coefficients of the edge (see Equation (6)) and flow information of the related points are needed. In the present implementation, most of the calculations, such as those of derivatives, flux terms and local time steps, are of this type. The last category, namely boundary-based calculation, is the calculation with a loop structure over the boundaries. The boundary treatment and the second category calculations mentioned on the boundaries are of this type.

The corresponding kernels of the first and third categories mentioned can be derived directly by unrolling the loop structures over points or boundaries, respectively, while more attention should be paid to the second category of the edge-based calculations due to its more complete location of memory access. The flux terms evaluation is a typical example of the edge-based calculation, and the general edge-based kernel can be listed as follows.

In the edge-based kernel (Listing 1), the subroutine kernel_flux is defined as a CUDA kernel with a function qualifier attributes(global). The thread ID of the kernel is captured in line 3, and the arithmetic operators of the corresponding thread are defined in the remaining lines. It can be noted that after the flux terms are evaluated (see lines 10–11 in Listing 1), they should be added to the two endpoints of the edge using a specified function atomicadd() (see lines 14–15 in Listing 1). The use of atomicadd() can avoid the data race condition that occurs when two threads concurrently access a variable stored in the identical memory location; however, it is much more time-consuming than normal memory access because each thread must wait until all other threads finish the operation for the same variable. In order to avoid the use of atomicadd() and to improve the efficiency of memory access, a modified point-based kernel is developed to replace the edge-based one. The code snippet is shown in Listing 2.

Unlike the edge-based kernel (Listing 1), each thread of the point-based kernel (Listing 2) manages the evaluations of flux terms at all edges from the corresponding cloud of points. A comparison of computing-time consumption of the edge-based and point-based kernels is carried out in relation to a series of successively refined clouds of points. As shown in Table 1, three groups of
kernels are tested to calculate the flux terms (see ‘kernel_Flux’ column), local time steps (see ‘kernel_DT’ column) and spatial derivatives (see ‘kernel_Grad’ column), respectively. It can be noted that the computing-time consumption of the point-based kernel for the evaluations of flux terms is significantly reduced (by nearly 25%) as compared with the edge-based kernel. This is because only one thread of the point-based kernel is required to finish the summation of the flux on each edge to the central node, the data race condition is now nonexistent and the efficiency of memory access can be improved. Similarly, consumption times for computing local time steps and spatial derivatives are also reduced (by about 10% ~ 20%) with the use of point-based kernels. Therefore, the point-based kernels are used for all calculations of the second category. Generally, all the calculations in the solver can be realized using two types of kernels: point-based kernels for the first and second categories and boundary-based kernels for the third category mentioned.

Meanwhile, the point-based data storage structure and thread hierarchy are also specified with respect to the

---

**Listing 1.** Code snippet of the edge-based kernel for flux evaluation.

```c
attributes(global) subroutine kernel_flux(ww_d,flux_d)
    !get the thread ID
    i = (blockIdx%x - 1) * blockDim%x + threadIdx%x
    !get variables of point i and point j
    ww_left(:) = ww_d(:, leftID)
    ww_right(:) = ww_d(:,rightID)
    !compute the fluxes of edge i-j and j-i
    flux_ij(:) = f(ww_left,ww_right)
    Flux_ji(:) = f(ww_right,ww_left)
    !add flux to the points i and j
    istat = atomicadd(flux_d(:, leftID),flux_ij(:))
    istat = atomicadd(flux_d(:,rightID),flux_ji(:))
endsubroutine
```

**Listing 2.** Code snippet of the point-based kernel for flux evaluation.

```c
attributes(global) subroutine kernel_flux(ww_d,flux_d)
    !get the thread ID
    i = (blockIdx%x - 1) * blockDim%x + threadIdx%x
    !get variables of left point i
    ww_left(:) = ww_d(:,i)
    Flux_i(:) = 0.0
    !loop over all satellites in cloud of point i
    do iNum = 1,Ni
        !get right point ID j
        j = sateID_d(iNum,i)
        !get variables of right point j
        ww_right(:) = ww_d(:,j)
        !compute the flux of edge i-j
        Flux_ij(:) = f(ww_left,ww_right)
        !add flux_ij to flux_i
        Flux_i(:) = flux_i(:) + flux_ij(:)
    enddo
    !save flux term to GPU data on global memory
    flux_d(:,i) = flux_i(:)
endsubroutine
```
Table 1. Consumption times of one single execution for the edge-based and point-based kernels.

| Number of points | kernel_Flux | kernel_DT | kernel_Grad |
|------------------|-------------|-----------|-------------|
|                  | Edge-based  | Point-based| Edge-based  | Point-based| Edge-based  | Point-based|
| 192 x 64         | 2.44E-05    | 1.75E-05  | 2.20E-05    | 1.81E-05  | 3.08E-05    | 2.08E-05   |
| 384 x 128        | 7.39E-05    | 5.31E-05  | 6.05E-05    | 5.31E-05  | 9.87E-05    | 8.16E-05   |
| 768 x 256        | 2.43E-04    | 1.84E-04  | 2.00E-04    | 1.82E-04  | 3.50E-04    | 2.66E-04   |
| 1536 x 512       | 9.37E-04    | 7.22E-04  | 7.62E-04    | 7.06E-04  | 1.34E-03    | 1.04E-03   |
| 3072 x 1024      | 3.71E-03    | 2.84E-03  | 3.03E-03    | 2.77E-03  | 5.35E-03    | 4.39E-03   |

Figure 5. The one-to-one correlation among clouds of points, data structure and thread hierarchy.

developed kernels to meet the flexible structure of meshless clouds of points, as shown in Figure 5. Since most of the flow variables are stored on the scattered points, a structure of one-dimensional array with a length of Np (referring to the number of points) is used to store each of them, and a similar one-dimensional structure is adopted in the thread hierarchy of the point-based kernels. Highly efficient global memory access can be expected due to use of the above one-to-one correlation among the nodes, the data storage structure and thread hierarchy, as illustrated in Figure 5. Additionally, the size of thread block Nt (thread number per block) is fixed to 128 for all kernels of the Euler solver, and 64 for the N-S solver from experience; the size of thread grid, Nb (number of blocks), can be calculated using the following equation: Nb = (nPoin + Nt – 1)/Nt.

3.2.3. GPU-based meshless solver: GFlow

Using the abovementioned implementation techniques of CUDA, a modularized solver, namely ‘GFlow’, is developed with the purpose of flexible expansion for further engineering applications. The computational kernels of the solver are grouped and packed into a serial of modules, such as Grad (used to define the calculations of derivatives of flow variables), Flux (used to define the calculations of convective and viscous flux terms), BC (used to define the calculations of boundaries), DT (used to define the calculations of time steps), Res (used to define the calculations of residuals) and Upd (used to update solution at time level) as shown in Figure 6. In each module, several kernels are used to define the corresponding calculations with different memory locations or different schemes, and a host subroutine is contained for managing those kernels. For instance, five point-based or

Figure 6. Typical modules of the program.
boundary-based kernels, which begin with ‘kernel_’, are contained in the module of Flux (see Figure 6): one point-based kernel is used to define the calculations of convective flux terms at internal edges with a central scheme, another point-based kernel is used to define the same calculations with an upwind scheme, one boundary-based kernel is used to define the calculations of convective flux terms at boundary points, and another two kernels are used to define the calculations of viscous flux terms at internal edges and boundary points, respectively. The host subroutine calKernel_Flux used to manage those kernels is shown in Listing 3, where branch structures are used to specify which kernel should be used. A similar status occurs in the module of BC, where only one major kernel launched from the CPU is contained, while a set of sub-kernels that begin with ‘subKernel_’ are used for different kinds of boundaries and launched by the major kernel. As a result, the main subroutine of the Runge–Kutta time-marching procedure can be determined as shown in Listing 4. The use of those computational modules results in a clear program architecture and can be flexibly expanded for complex engineering problems by adding new schemes or new boundary types in the corresponding modules, without modifying other parts of the solver.

3.2.4. GPU-based meshless solver: GFlow-Ex

In order to further fully utilize the massively parallel GPU architecture, another GPU-based meshless solver, namely ‘GFlow-Ex’ is developed. GFlow-Ex is a series of standalone solvers used for different flow types. This means that each flow type is realized separately with a whole set of private kernels to avoid the negative influence on efficiency from the branch structures, which is frequently used in GFlow for specifying which kernel should be launched for different flow types or schemes (as per Listing 3). Without the limit of branch structure, the calculations and the memory access of the solver can be improved by combining some of the computational kernels. These premerger kernels should have the same structure in terms of thread hierarchy, either point-based one or boundary-based one, and they can be merged together when one of the following two conditions is
Listing 5. Two kinds of premerger and merged kernels.

```
attributes(global) subroutine kernelA(in A(:,), out B(:,))
  B(i) = f1(A(i))
endsubroutine

attributes(global) subroutine kernelB(in A(:,), out C(:,))
  C(i) = f2(A(i))
endsubroutine

attributes(global) subroutine kernel_combined(in A(:,), out B(:,), out C(:,))
  B(i) = f1(A(i))
  C(i) = f2(A(i))
endsubroutine
```

```
attributes(global) subroutine kernelA(in A(:,), out B(:,))
  B(i) = f1(A(i))
endsubroutine

attributes(global) subroutine kernelB(in B(:,), out C(:,))
  C(i) = f2(B(i))
endsubroutine

attributes(global) subroutine kernel_combined(in A(:,), out C(:,))
  C(i) = f2(f1(A(i)))
endsubroutine
```

Listing 5. Two kinds of premerger and merged kernels.

satisfied. First, when the kernels contain the same input variables and arbitrary orders of execution of them are feasible, they can be merged together (see Listing 5(a)). Second, when the output variables of one kernel are just the input variables of another kernel, they can also be merged together (see Listing 5(b)). The actual condition in our implementation is much more complex, since three or more kernels can be merged together and the two conditions mentioned often occur at the same time. Take the implementation of the steady Euler solver, for example; the same input of the conservative variables are used in four point-based kernels for calculating the spectral radiuses (Equation (17)), local time steps (Equation (20)), artificial dissipative terms (Equation (16)) and derivatives of flow variables (Equation (13)) at all scattered points, respectively, and therefore these kernels are considered to be merged together. The kernels for the similar calculations at boundaries can also be merged together. Furthermore, the variables of flux terms are just used for updating the solution after evaluation; thus, the kernels of flux term evaluation (Equations (15) and (18)) and solution updating (Equation (19)) can be merged together. It is emphasized that the amount of memory access is significantly decreased due to careful management of memory access via the above merging of the computational kernels, which will result in further acceleration (see performance analysis in the section 4).

4. Numerical results and analysis

All the following numerical simulations presented were performed on a Windows desktop platform equipped with an Intel core i5-3450 CPU and a NVIDIA GTX TITAN GPU. As illustrated in Table 2, the peak performances of the present GPU, such as the peak single-precision GFLOPS (Giga floating point operations per second) and memory bandwidth GB/S (Gigabytes per second), are tens of times faster than that of a CPU. In addition to the raw computational cost and power, metrics such as price/performance and power/performance are presented in the table to assess the potential of new computing hardware. It can be noted that the price of present GPU is reduced to nearly one quarter and the power consuming is reduced to about one sixth as compared with the CPU, while achieving the same peak performance. Additionally, PGI Fortran is used to compile the CUDA Fortran codes. The resulting GPU parallel solvers are executed in single-precision, as well as the CPU serial solver. Having presented the results of various simulations of inviscid or viscous flows over aerodynamic
Table 2. Key performance indicators of Intel core i5-3450 CPU and NVIDIA GTX TITAN GPU.

|                      | Intel core i5-3450 | NVIDIA GTX TITAN |
|----------------------|--------------------|------------------|
| Processing Units     | 4                  | 2688             |
| Frequency (GHz)      | 3.1                | 837              |
| Peak single-precision (GFLOPS) | 198.4           | 4500             |
| Memory (GB)          | 16                 | 6                |
| Memory Bandwidth (GB/S) | 25.6             | 288              |
| Power (Watt)         | 77                 | 250              |
| Street Price ($)     | 195                | 999              |
| Price/Performance    | 0.89               | 0.22             |
| Power/Performance    | 0.35               | 0.056            |

or academic bodies for validation, the performance analysis is addressed in the subsequent section with a comparison of present GPU-based meshless solvers.

4.1. Test cases

In order to facilitate comparisons of our meshless results with those of mesh-based methods, the structured or unstructured grids are directly used for constructing the clouds of points to discretize the computational domains of the following test cases. For the inviscid test cases, a condition of flow tangency is applied at the solid boundaries by setting normal velocity to be zero, while no-slip boundary conditions are enforced by setting zero velocity at the solid walls for the viscous test cases used. At the far field, characteristic nonreflecting boundary conditions are enforced (for details, see Blazek, 2001). The size of clouds of points used in present calculations can be found in the Table 4 (see section 4.2).

4.1.1. Inviscid flow over RAE2822 airfoil

In this case, an inviscid flow around a RAE2822 airfoil for a free-stream Mach number of 0.729 and an angle of attack of 2.31° is simulated. This case is often selected to test the ability to capture the flow features of shock and its location in transonic flows. Regularly distributed clouds of points obtained from structured grids for the sake of convenience are employed in this simulation (see Figure 7). The resulting contours of Mach number and the plot of pressure coefficients are presented in Figure 8(a) and 8(b), respectively. The corresponding experimental data (as cited in Cook, McDonald, & Firmin, 1979) and other available numerical results of mesh-based methods (Ahmed, Ahmed, Affan, & Larik, 2014; Cook et al., 1979) are also depicted in the figure for comparison. Due to our ignoring the viscous effect, the differences of pressure coefficient can also be observed, particularly at the trailing edge of the airfoil. However, in general, the numerical results clearly bring out the fact that the jump across the top shock, the location and wall pressure distributions are all captured accurately by the present method.

4.1.2. Inviscid flow over a three-element airfoil

In this case, an inviscid flow around a three-element airfoil for a free-stream Mach number of 0.197 and angle of attack of 4.01° is simulated to test the ability to capture flow features at a low Mach number. Irregularly distributed clouds of points that obtained from unstructured grids are used for simulation (see Figure 9). The resulting Mach number contours and the \( C_p \) plot are presented in Figure 10(a) and 10(b), respectively. It can be noted that the present results for both GFlow and GFlow-Ex GPU-solvers are all in good agreement with available experimental data (as cited in Cao & Chen, 2015) and computational values (Cao & Chen, 2015), particularly on the surface of the main part of the three-element airfoil.

Figure 7. Point distribution around RAE2822 airfoil. (a) Points in the computational domain. (b) Points near the leading-edge.
Figure 8. The inviscid results for RAE2822 airfoil at $M_\infty = 0.729$ and AoA = 2.31°. (a) Contours of Mach number. (b) Plots of pressure coefficient.

Figure 9. Point distribution around a three-element airfoil. (a) Points in the computational domain. (b) Points near the leading-edge.

Figure 10. The inviscid results for a three-element airfoil at $M_\infty = 0.197$ and AoA = 4.01°. (a) Contours of Mach number. (b) Plots of pressure coefficient.
4.1.3. Viscous flow over NACA0012 airfoil
A viscous flow around NACA0012 airfoil is simulated to test the ability to capture the features of viscous flows. The flow conditions for this case are a Mach number of 0.5, an angle of attack of 0, and a Reynolds number of 5,000. Regularly distributed clouds of points are also employed and are shown in Figure 11. The resulting Mach number contours and the $C_p$ plot are presented in Figure 12(a) and 12(b), respectively. Compared with available numerical results (Caraeni & Fuchs, 2002; Chassaing, Khelladi, & Nogueira, 2013; Lyra & Morgan, 2002), reasonably accurate results are obtained using the present GPU-based meshless solvers.

4.1.4. Viscous flow past a cylinder
The last case presented here is the unsteady flow ($M_\infty = 0.002$, $Re_\infty = 200$) past a cylinder based on irregularly distributed clouds of points (see Figure 13). In this case, the laminar Navier–Stokes equations with preconditioning (Zhang & Chen, 2016) are employed to deal with the stiffness of the low-speed problems. The contours of Mach number and of vorticity are shown in Figures 14(a) and 14(b), respectively. It can be seen that the vortexes are formed alternately at the top or bottom half of the cylinder. The values of lift and drag coefficients for the periodic state are represented in Table 3, as well as the Strouhal number. In addition, the results from several other calculations (Lecointe & Piquet, 1984; Rogers & Kwak, 1990; Rosenfeld, Kwak, & Vinokur, 1988) are listed in the same table for comparison. It can be noted that the present results are in good agreement with other data listed in the table.

4.2. Performance analysis
4.2.1. Performance indexes
In order to analyze the running performance of the solver or key kernels in the solver developed, a unified time
Table 3. Lift and grad coefficients and Strouhal numbers for circular cylinder flow at Re = 200.

|                  | $C_D$       | $C_L$       | $St$       |
|------------------|-------------|-------------|------------|
| Present          | 1.28 ± 0.05 | ±0.63       | 0.198      |
| Rogers and Kwak  | 1.23 ± 0.05 | ±0.65       | 0.185      |
| Rosenfeld        | 1.46 ± 0.05 | ±0.69       | 0.211      |
| Lecointe and Piquet | 1.46 ± 0.04 | ±0.70       | 0.227      |

An index, $T_k$, is constructed, which reads

$$T_k = \sigma \times \frac{T_{\text{iter}}}{N_p} \quad (21)$$

where $T_{\text{iter}}$ is the running time of a single iteration of the solver, $N_p$ is the total number of meshless points in

Table 4. The unified time index, $T_k$, and the speedups of CPU and GPU implementations.

| Test case         | Number of points | CPU (ms) | GFlow (ms) | GFlow-Ex (ms) | GFlow (ms) | GFlow-Ex (ms) | Speedup |
|-------------------|------------------|----------|------------|---------------|------------|---------------|---------|
| RAE2822           | 192 × 32         | 973.8    | 205.6      | 109.8         | 4.74×      | 8.87×         |         |
|                   | 384 × 64         | 923.3    | 72.0       | 37.1          | 12.83×     | 24.89×        |         |
|                   | 768 × 128        | 898.2    | 42.1       | 20.5          | 21.35×     | 43.79×        |         |
|                   | 1536 × 256       | 894.9    | 31.9       | 15.2          | 28.04×     | 58.69×        |         |
|                   | 3072 × 512       | 880.1    | 30.1       | 13.7          | 29.27×     | 64.09×        |         |
|                   | 6144 × 1024      | 887.1    | 30.1       | 14.3          | 29.45×     | 62.23×        |         |
| Three-element airfoil | 5865         | 1153.5   | 240.7      | 139.6         | 4.79×      | 8.26×         |         |
|                   | 23181           | 1203.8   | 97.6       | 59.0          | 12.33×     | 20.40×        |         |
|                   | 97062           | 1265.2   | 63.2       | 40.5          | 20.02×     | 31.22×        |         |
|                   | 387022          | 1381.6   | 52.1       | 34.3          | 26.50×     | 40.28×        |         |
|                   | 1548088         | 1368.5   | 53.2       | 33.7          | 25.71×     | 40.57×        |         |
|                   | 6192352         | 1373.8   | 53.8       | 32.3          | 25.55×     | 42.52×        |         |
| NACA0012          | 192 × 64        | 1731.6   | 179.8      | 93.6          | 9.63×      | 18.49×        |         |
|                   | 384 × 128       | 1733.1   | 105.4      | 53.9          | 16.44×     | 32.18×        |         |
|                   | 768 × 256       | 1834.7   | 79.0       | 40.3          | 23.21×     | 45.51×        |         |
|                   | 1536 × 512      | 1784.8   | 73.6       | 36.9          | 24.26×     | 48.37×        |         |
|                   | 3072 × 1024     | 1796.7   | 76.3       | 39.2          | 23.56×     | 45.88×        |         |
| 2-D cylinder      | 17677           | 2207.2   | 205.3      | 117.5         | 10.75×     | 18.78×        |         |
|                   | 56686           | 2311.0   | 148.4      | 85.9          | 15.57×     | 26.92×        |         |
|                   | 207401          | 2523.0   | 127.3      | 73.7          | 19.82×     | 34.23×        |         |
|                   | 802531          | 2666.6   | 126.3      | 74.7          | 21.12×     | 35.72×        |         |
|                   | 3210124         | 2654.4   | 119.0      | 77.2          | 22.30×     | 34.37×        |         |
the computational domain, $\sigma$ is a constant for scaling the magnitude of the unified time index, and $\sigma = 10^6$ is used in this paper according to the present test environment of GPU platform, yielding the throughput, $T_k$, in seconds per million point iteration, where subscript $k$ refers CPU or GPU. Thus, the speedup of the solver developed can be computed as

$$\text{Speedup} = \frac{T_{\text{CPU}}}{T_{\text{GPU}}} \tag{22}$$

The performance of the solver is mainly dependent on the performances of key kernels adopted in the solver; therefore, we will first analyze the performances of key kernels, which could help us to further modify the program developed.

### 4.2.2. Performances of key computational kernels

For the sake of convenience, inviscid or viscous internal flows in a rectangular tunnel (length: 3, width: 1) are simulated for testing the performances of key computational kernels of the 2D Euler or N-S solver GFlow. The domain is uniformly covered by $1536 \times 512$ points, which is much greater than the number of processor cores of the GPU presented; thus, point-based meshless calculation is large enough to fully utilize the massively parallel GPU architecture. The tests of key computational kernels are carried out by incrementally moving more of the key procedures of the CPU serial code onto the GPU using CUDA kernels.

As illustrated in Figure 15, key computational kernels of seven procedures, namely Grad, Flux, BC, DT, Upd, Res (refer to subsection 3.2.3 for details) and Oth (other calculations), are successively analyzed. To test the Euler solver (Figure 15(a)), the program is first executed on the CPU (the first columnar ribbon, ‘CPU Only’, in Figure 15(a)), and the ‘Flux’ procedure is then moved onto the GPU using corresponding kernels, while the other procedures are kept on the CPU (the second columnar ribbon, ‘+ Flux’, in Figure 15(a)). The test is carried out by incrementally moving more of the key procedures of the CPU serial code onto the GPU using CUDA kernels.

Figure 15. Accelerated progress of key computational kernels, conducted incrementally by adding more GPU support to the serial CPU implementation. (a) Euler. (b) N-S.

Figure 16. Effect of the size of thread block for $T_k$. (a) Euler. (b) N-S.
is conducted by moving the remaining procedures one by one onto the GPU until all of them are executed on the GPU (corresponding to the last columnar ribbon in Figure 15(a)). It can be seen that the simulation is gradually accelerated as more of the procedures are moved onto the GPU. A similar feature can be obtained from the N-S test (see Figure 15(b)).

The performances of the unified times, $T_k$, of those key procedures with both serial CPU and parallel GPU implementations are also listed at the bottom of Figure 15, as well as the computational speedups. The key kernels of linear calculations (such as Grad, Flux, BC, DT and Upd in Figure 15) take more than 95% of the total running time of the solver, and can perform remarkable speedups of dozens of times (from $11 \times$ to $53 \times$). However, the kernel of reduction operation for monitoring the convergence (see Res in Figure 15) performs almost no speedup (less than $3 \times$). It should be emphasized that different speedups achieved by the kernels mentioned leaves room for further modification of the program developed. We can see that the final speedups for the Euler and N-S solver are $29 \times$ and $21 \times$, respectively.

Additionally, an appropriate size of thread block used for flow simulations is obtained by testing the GPU-based meshless solver as a way of changing the number of threads in a block. A typical set of numbers, 8, 16, 32, 64, 128, 256 and 512, is chosen to configure the sizes of thread block for both Euler and N-S test cases. The values of computing-time consumption variation with the number of threads in a block are illustrated in Figures 16(a) and 16(b) for the developed Euler and N-S solver, respectively. In those figures, different columnar ribbons denote different numbers of threads in a block, and the vertical height of each columnar ribbon denotes the consumption time of one iteration. Parabolic distributions can be observed, and therefore appropriate sizes of thread block, 128 for inviscid and 64 for viscous, can be selected for the following related simulations.

### 4.2.3. Performances of whole programs

The cases outlined in section 4.1 are used here for testing the performances of two developed solvers GFlow and GFlow-Ex. In order to obtain a view of the relationship between the performance of GPU and the size of the problem, four successively refined clouds of points are employed in each case, as illustrated in Table 2. The performance of unified time, $T_k$ (see Equation (21)), with serial CPU and parallel GPU implementations are presented in Table 4, as well as the computational speedups. It can be seen that the unified time of the serial CPU implementation in each test case is kept almost unchanged with the increasing size of points (see the third column of Table 4), while the $T_k$ of both GFlow and GFlow-Ex obviously decrease (see the fourth and fifth columns of Table 2). Due to the further modification of key computational kernels (see section 3.2.4), the unified time of the solver GFlow-Ex drops to approximately one-half to one-third of that of GFlow.

Figure 17 presents the computational speedups with respect to different sizes of points. Depending on the problem size, the performance of GFlow varies from $4 \times$ to $29 \times$ relative to the serial CPU implementation (Figure 17(a)), while that of GFlow-Ex varies from $8 \times$ to $64 \times$ (Figure 17(b)). It can be observed that continual increase of speedups related to the growing number of points can be achieved until the total computing effort exceeds the capacity of GPU architecture allowed. It can also be clearly seen that GFlow-Ex can perform two to three times faster speedup than GFlow. It is emphasized that the computational speeds are impressive for large problem sizes, because the arithmetic intensity can fully utilize the massively parallel GPU architecture, which
exhibits the potential of the GPU-accelerated meshless method for more complicated flow problems.

5. Conclusions

A GPU-accelerated meshless method is presented in this paper for solving two-dimensional compressible flows over a set of typical aerodynamic bodies, such as single- or multi-element airfoils. The GPU implementations of the meshless method are successfully carried out using the CUDA Fortran programming model. The use of CUDA kernels, the point-based data storage structure and thread hierarchy results in two GPU-accelerated solvers, namely GFlow and GFlow-Ex. Numerical results of the test cases presented for both solvers are in good agreement with available experimental data or computational values that appear in extant literature. It can be seen that the performances of both solvers are greatly improved with a speedup of $29 \times$ at most for GFlow and $64 \times$ for GFlow-Ex. It can also be noted that the speedup of GFlow-Ex was nearly two times faster than that of GFlow due to careful management of the memory access. The present GPU-based meshless method entails point-based flexibility; hence, further extension for more complex flow problems can be considered. In addition, the speedup can be further enhanced by using techniques such as data reordering, and by adopting more challenging implicit schemes, such as LU-SGS, which will be addressed in future research.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was supported by National Natural Science Foundation of China: [Grant Number 11172134]; Founding of Jiangsu Innovation Program for Graduate Education: [Grant Number CXLX13_132].

References

Ahmed, F., Ahmed, F., Affan, M., & Larik, J. R. (2014). Numerical solution of 2D Euler equations for the transonic flow past over NACA0012 and RAE2822 airfoils using high order accurate Runge-Kutta discontinuous galerkin method. Proceedings of 11th international Bhurban conference on applied sciences & technology, Islamabad, Pakistan.

Batina, J. (1993). A Gridless Euler/Navier–stokes solution algorithm for complex-aircraft applications. AIAA Paper 93-0333.

Blazek, J. (2001). Computational fluid dynamics: Principles and applications (1st ed.). Amsterdam: Elsevier Science Ltd.

Brandvik, T., & Pullan, G. (2008). Acceleration of a 3D Euler solver using commodity graphics hardware. AIAA Paper 2008-607.

Cao, C., & Chen, H. Q. (2015). A preconditioned gridless method for solving Euler equations at Low mach numbers. Transactions of Nanjing University of Aeronautics and Astronautics, 32(4), 399–407.

Caraeni, D., & Fuchs, L. (2002). Compact third-order multidimensional upwind scheme for Navier–stokes simulations. Theoretical & Computational Fluid Dynamics, 15(6), 373–401.

Chassaing, J. C., Khelladi, S., & Nogueira, X. (2013). Accuracy assessment of a high-order moving least squares finite volume method for compressible flows. Computers & Fluids, 71(2), 41–53.

Chen, H. Q., & Shu, C. (2005). An efficient implicit mesh-free method to solve Two-dimensional compressible euler equations. International Journal of Modern Physics C, 16(3), 439–454.

Chiu, K-Y.E., Wang, Q. Q., Hu, R., & Jameson, A. (2012). A conservative mesh-free scheme and generalized framework for conservation laws. Siam Journal on Scientific Computing, 34(6), A2896–A2916.

Cook, P. H., McDonald, M. A., & Firmin, M. C. P. (1979). Aerofoil RAE2822 - Pressure distributions, and boundary layer and wake measurements. Experimental Data Base for Computer Program Assessment, AGARD Report AR 138.

Corrigan, A., Camelli, F. F., Löhner, R., & Wallin, J. (2011). Running unstructured grid-based CFD solvers on modern graphics hardware. International Journal for Numerical Methods in Fluids, 66(2), 221–229.

Elsen, E., Legresley, P., & Darve, E. (2008). Large calculation of the flow over a hypersonic vehicle using a GPU. Journal of Computational Physics, 227(24), 10148–10161.

Guzel, G., & Akgun, T. (2014). Accelerating finite-volume based Lattice Boltzmann flow solutions on a graphics processing unit using CUDA. AIAA Paper 2014–3082.

Jameson, A., Schmidt, A., & Turkel, E. (1981). Numerical solutions of the Euler equations by Finite volume methods using Runge–Kutta time-stepping schemes. AIAA Paper 81-1259.

Julien, C. T., & Inanc, S. (2009). CUDA implementation of a Navier–stokes solver on multi-GPU desktop platforms for incompressible flows. AIAA Paper 2009-758.

Katz, A., & Jameson, A. (2009). Multicloud: Multigrid convergence with a meshless operator. Journal of Computational Physics, 228(14), 5237–5250.

Katz, A., & Jameson, A. (2010). Meshless scheme based on alignment constraints. AIAA Journal, 48(11), 2501–2511.

Lecointe, Y., & Piquet, J. (1984). On the Use of several compact methods for the study of unsteady incompressible viscous flow round a circular cylinder. Computer and Fluids, 12(4), 255–280.

Löhner, R., Corrigan, A., Wichmann, K. R., & Wall, W. (2014). Comparison of Lattice-Boltzmann and Finite Difference Solvers. AIAA Paper 2014-1439.

Löhner, R., Sacco, C., Oñate, E., & Idelsohn, S. (2002). A finite point method for compressible flow. International Journal for Numerical Methods in Engineering, 53(8), 1765–1779.

Luo, L. X., Edwards, J. R., Luo, H., & Mueller, F. (2014). GPU port of a parallel incompressible Navier–stokes solver based on openACC and MVAPICH2. AIAA Paper 2014-3083.

Lyra, P. R. M., & Morgan, K. (2002). A review and comparative study of upwind biased schemes for compressible flow computation. Part III: Multidimensional extension on
unstructured grids. *Archives of Computational Methods in Engineering*, 9(3), 207–256.

Ma, Z. H., Chen, H. Q., & Zhou, C. H. (2008). A study of point moving adaptivity in gridless method. *Computer Methods in Applied Mechanics and Engineering*, 197(21–24), 1926–1937.

Ma, Z. H., Wang, H., & Pu, S. H. (2014). GPU computing of compressible flow problems by a Meshless method with space-filling curves. *Journal of Computational Physics*, 263(C), 113–135.

NVIDIA. (2014). NVIDIA’s next generation CUDA compute architecture: Kepler GK110/210. Retrieved from [http://www.nvidia.com](http://www.nvidia.com).

NVIDIA. (2015). CUDA C programming guide, version 7.5. Retrieved from [http://docs.nvidia.com](http://docs.nvidia.com).

Oñate, E., Idelsohn, S., Zienkiewicz, O. C., & Taylor, R. L. (1996). A finite point method in computational mechanics. *Applications to Convective Transport and Fluid Flow. International Journal for Numerical Methods in Engineering*, 39(22), 3839–3866.

Rogers, S. E., & Kwak, D. (1990). Upwind differencing scheme for the time-accurate incompressible Navier–stokes equations. *AIAA Journal*, 28(2), 253–262.

Roque, C. M. C., Cunha, D., Shu, C., & Ferreira, A. J. M. (2011). A local radial basis functions—Finite differences technique for the analysis of composite plates. *Engineering Analysis with Boundary Elements*, 35(3), 363–374.

Rosenfeld, M., Kwak, D., & Vinokur, M. (1988). A solution method for the unsteady and incompressible Navier–stokes equations in generalized coordinate systems. AIAA paper 88-0718.

Ruetsch, G., & Fatica, M. (2014). *CUDA fortran for scientists and engineers: Best practices for efficient CUDA fortran programming*. Amsterdam: Elsevier Science Ltd.

Singh, M. K., Ramesh, V., & Balakrishnan, N. (2015). Implicit scheme for meshless compressible Euler solver. *Engineering Applications of Computational Fluid Mechanics*, 9(1), 382–398.

Sutherland, W. (1893). LII. The viscosity of gases and molecular force. *Philosophical Magazine Series 5*, 36(223), 507–531.

Xia, Y. D., Luo, J. L., Luo, L. X., Luo, H., & Edwards, J. R. (2014). On the multi-GPU computing of a reconstructed discontinuous Galerkin method for compressible flows on 3D hybrid grids. AIAA Paper 2014-3081.

Zhang, J. L., & Chen, H. Q. (2016). GPU implementation of preconditioning method for Low-speed flows. *International Journal of Modern Physics: Conference Series*, 42, 1660165, 1–18.

Zhou, X., & Xu, H. Q. (2010). Gridless method for unsteady flows involving moving discrete points and its applications. *Engineering Applications of Computational Fluid Mechanics*, 4(2), 276–286.