A parallel meshless dynamic cloud method on graphic processing units for unsteady compressible flows past moving boundaries

Z.H. Ma, H. Wang, S.H. Pu

PII: S0045-7825(14)00435-6
DOI: http://dx.doi.org/10.1016/j.cma.2014.11.010
Reference: CMA 10459

To appear in: *Comput. Methods Appl. Mech. Engrg.*

Received date: 12 February 2014
Revised date: 5 November 2014
Accepted date: 7 November 2014

A parallel meshless dynamic cloud method on graphic processing units for unsteady compressible flows past moving boundaries

Z. H. Ma[a,*], H. Wang[b], S. H. Pu[c,d]

[a]Centre for Mathematical Modelling and Flow Analysis, School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University, Manchester M1 5GD, United Kingdom
[b]Department of Marine Technology, Norwegian University of Science and Technology, Trondheim, NO-7491, Norway
[c]Chengdu Aircraft Design & Research Institute, Chengdu 610041, P.R. China
[d]Department of Aerodynamics, Nanjing University of Aeronautics & Astronautics, Nanjing 210016, P.R. China

Abstract

This paper presents an effort to implement a recently proposed meshless dynamic cloud method [Hong Wang et al. A study of gridless method with dynamic clouds of points for solving unsteady CFD problems in aerodynamics, *Int. J. Numer. Meth. Fluids* 2010; 64: 98-118] on modern high-performance graphic processing units (GPUs) with the compute unified device architecture (CUDA) programming model. Within the framework of the meshless method, clouds of points used as basic computational stencils are distributed in the whole flow domain. The spatial derivatives of the governing equations are discretised by the moving-least square scheme on every cloud of points. Roe's approximate Riemann solver is adopted to compute the convective flux. A dual-time stepping approach, which iterates in physical and pseudo temporal spaces, is employed to obtain the time-accurate solution. Simulation of steady compressible flows over a fixed aerofoil is firstly carried out to verify the GPU implementation of the method. Then it is extended to compute unsteady flows past oscillatory aerofoils. Numerical outcomes are compared with experimental and/or other reference results to validate the method. Significant performance speedup of more than an order of magnitude is verified by the numerical results. Systematic analysis shows that GPU is more energy efficient than CPU for solving aerodynamic problems. This demonstrates the potential of the proposed method to solve fluid-structure interaction problems.

Keywords: aerodynamics, oscillatory aerofoil, CUDA, graph mapping

*Corresponding author. Tel: +44 (0)161-247-1574
Email addresses: z.ma@mmu.ac.uk (Z. H. Ma), hong.wang@ntnu.no (H. Wang), nuaapu@yahoo.com.cn (S. H. Pu)

1. Introduction

Unsteady flows over moving boundaries are frequently encountered in many scientific fields such as aerodynamics, hydrodynamics and biological fluid dynamics etc. These complicated problems play an important role in fundamental research and industrial applications, however they have proved extremely challenging to theoretical, experimental and numerical investigations. When dealing with them by a computational fluid dynamics (CFD) method, the motion of boundaries need to be handled appropriately with a robust numerical algorithm. Meanwhile, the simulation itself is very time-consuming due to intensive computing. A persistent objective of CFD is to devise accurate and efficient numerical methods to solve these complicated flow problems.

During the past several decades, a new kind of numerical algorithm named meshless (or gridless, meshfree, particle) method has gradually attracted more and more attentions of researchers in CFD. A distinctive feature of meshless methods is that connectivities between points are not necessary to be considered, since they do not adopt traditional structured/unstructured mesh topologies but employs flexible clouds of points, which are basically composed of a centre point and several satellites, to discretise the flow domain. The derivatives of a mathematical function in a cloud of points can be computed by the least-square curve fit, radial basis function or other effective strategies. In the area of aerodynamics, meshless methods have been successfully applied to solve steady compressible flows [1, 2, 3, 4, 5, 6, 7]. Considering unsteady flows, Wang et al. [8] proposed a meshless dynamic cloud method to deal with moving boundaries. A very simple but effective algebraic mapping strategy was used in their work to adjust the distribution of meshless points. Solid boundary penetration induced by other numerical methods was avoided by the dynamic cloud method even for cases with relatively large displacements, such as a 30° pitch motion of an aerofoil (see figures 3 and 4 of [8]). This method has also been extended to drag reduction design for an aerofoil with active flow control [9].

Until now, these aforementioned research works of meshless methods for steady and unsteady flows past fixed solid bodies have mostly been carried out with serial computing on a single core of the CPU. On the other hand, Ortega et al. [10, 11] paid attention to the parallelisation of the finite point method on multi-core CPUs with the OpenMP programming model. They observed unsatisfactory scalability problems and pointed out that attainable speedups on multi-core CPUs will drop once the number of processor cores is over 4 due to the high cache miss rate and limited memory bandwidth of CPU. Therefore, they suggested to use much higher-performance hardware platforms [11].

Nowadays, computer science is embracing a new and fast developing territorial namely GPU computing technology, in which the graphic hardware can deliver Tera-scale single- and double-precision floating-point operations per second in very recent years. This provides tremendous power to scientific computing and it is extremely attractive to the CFD community, in which high efficiency/performance is always a requirement of numerical methods for many complicated problems. For important GPU implementations of mesh methods, readers may refer to the works of Karatarakis et al. [12] and Papadrakakis et al. [13] for solid mechanics problems; Bard and Dorelli [14], Liang et al. [15], Corrigan et al. [16], Asouti et al. [17] and Kampolis et al. [18] for fluid mechanics problems. In these works, the strategies to utilise the GPU to solve complicated problems in solid or fluid mechanics are explained in detail. Specific techniques to prevent thread race conditions or to improve memory performance are also provided. All of them reported impressive speedups of the fundamental mesh based numerical solvers, this triggered off our intention to investigate the possibility of realising the meshless method for CFD on GPUs in the first place. Initial success of such kind of attempt to solve steady compressible flows with a meshless method on GPUs was reported in our recent work [19]. These inspiring works for flows over fixed objects encourage us to further develop GPU based numerical methods to solve more challenging unsteady compressible flows past moving boundaries. This study exhibits such kind of an effort to accelerate the meshless dynamic method, which enjoys the robustness to deal with rigid and/or flexible boundaries, on modern graphic hardware.

The rest of the paper is organised as follows. Key aspects of the numerical method including the governing equations, meshless discretisation, dual time stepping scheme and dynamic cloud technique are described in Section 2. The implementation of the meshless dynamic cloud method on the GPU is discussed in Section 3. Numerical examples of steady and unsteady flows are given in Section 4. The obtained results are compared to the experiment and/or other available reference solutions to verify the accuracy of the present method. Systematic performance benchmarks of the method on CPU and GPU with up to one million points are also carried out. Not only the running time costs are compared but also the energy consumptions are investigated.

The major contributions of the work may contain the following phases:

- To the best of our knowledge, we are the first to present a GPU based numerical method for simulating unsteady compressible flows past moving boundaries.

- The performance of meshless dynamic cloud method is successfully improved by more than an order of magnitude, and the GPU based computing method is more energy efficient than the CPU.
- This work demonstrates the potential of the present GPU based algorithm for solving more complicated fluid-structure interaction problems.

2. Numerical method

2.1. Governing equations

In a two-dimensional Cartesian coordinate system, the Euler equations in an arbitrary Lagrangian and Eulerian form can be expressed as

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{E}}{\partial x} + \frac{\partial \mathbf{F}}{\partial y} = 0 \tag{1}$$

where **U** is a vector of conservative variables, **E** and **F** are the flux terms, they are defined as

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho e_{\mathrm{t}} \end{bmatrix}, \quad \mathbf{E} = \begin{bmatrix} \rho\,(u - x_t) \\ \rho u\,(u - x_t) + p \\ \rho v\,(u - x_t) \\ \rho e_{\mathrm{t}}\,(u - x_t) + pu \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \rho\,(v - y_t) \\ \rho u\,(v - y_t) \\ \rho v\,(v - y_t) + p \\ \rho e_{\mathrm{t}}\,(v - y_t) + pv \end{bmatrix} \tag{2}$$

in which, ρ is the density, p is the pressure, u and v are the components of (fluid) velocity vector $\vec{V}$ along x and y axes respectively; x_t and y_t represent the components of velocity vector $\vec{V}_t$ along x and y axes of discrete points. The total energy per volume ρe_{t} is given by

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

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

2.2. Spatial and temporal discretisation

For any cloud C_i in the flow domain, the Euler equations (1) are required to be satisfied

$$\left.\frac{\partial \mathbf{U}}{\partial t}\right|_{C_i} + \left(\frac{\partial \mathbf{E}}{\partial x} + \frac{\partial \mathbf{F}}{\partial y}\right)_{C_i} = 0 \tag{4}$$

For simplicity, we use the subscript i to represent the cloud C_i in the following. With a moving least square curve fit [1, 3, 5], Eq. (4) can be written as

$$\frac{\partial \mathbf{U}_i}{\partial t} + \sum_{j=1}^{M_i} \left[\left(\alpha_{ij} \mathbf{E}_{ij} + \beta_{ij} \mathbf{F}_{ij} \right) - \left(\alpha_{ij} \mathbf{E}_i + \beta_{ij} \mathbf{F}_i \right) \right] = 0 \tag{5}$$

where the subscript ij indicates the midpoint between the centre i and a satellite j. Introducing a parameter $\lambda = \sqrt{\alpha^2 + \beta^2}$ and a vector $\vec{\eta} = (\alpha/\lambda, \beta/\lambda)$, Eq. (5) can be expressed as

$$\frac{\partial \mathbf{U}_i}{\partial t} + \sum_{j=1}^{M_i} (\mathbf{G}_{ij} - \mathbf{G}_i) \lambda_{ij} = 0 \tag{6}$$

The flux function $\mathbf{G}$ is evaluated by Roe's approximate Riemann solver [8, 20]

$$\mathbf{G} = \frac{1}{2} \left[\mathbf{G}(\mathbf{U}_{\mathrm{L}}) + \mathbf{G}(\mathbf{U}_{\mathrm{R}}) - |\mathbf{A}|(\mathbf{U}_{\mathrm{R}} - \mathbf{U}_{\mathrm{L}}) \right] \tag{7}$$

In order to improve the accuracy, the data is reconstructed by a piecewise linear interpolation scheme and van Leer's limiter is used to prevent spurious oscillations caused by the interpolation [19]. The semi-discrete form of Eq. (6) can be written as

$$\frac{\mathrm{d}\mathbf{U_i}}{\mathrm{d}t} + \mathbf{R_i} = 0 \tag{8}$$

where $\mathbf{R}$ represents the residual vector. In order to obtain the solution, a second-order time differential scheme is used

$$\frac{3\mathbf{U}_i^{n+1} - 4\mathbf{U}_i^{n} + \mathbf{U}_i^{n-1}}{2\Delta t} + \mathbf{R}_i \left(\mathbf{U}_i^{n+1} \right) = 0 \tag{9}$$

A dual time-stepping approach [21] is employed to solve Eq. (9), the derivative of pseudo time is denoted as τ

$$\frac{\mathrm{d}\mathbf{U}_i^{n+1}}{\mathrm{d}\tau} + \frac{3\mathbf{U}_i^{n+1} - 4\mathbf{U}_i^{n} + \mathbf{U}_i^{n-1}}{2\Delta t} + \mathbf{R}_i \left(\mathbf{U}_i^{n+1} \right) = 0 \tag{10}$$

using $\mathbf{U}^*$ as the approximation for $\mathbf{U}^{n+1}$, the unsteady residual is defined as

$$\mathbf{R}_i^* \left(\mathbf{U}_i^* \right) = \frac{3\mathbf{U}_i^{*} - 4\mathbf{U}_i^{n} + \mathbf{U}_i^{n-1}}{2\Delta t} + \mathbf{R}_i \left(\mathbf{U}_i^* \right) \tag{11}$$

The solution to Eq. (10) is the steady state of pseudo time $\Delta\tau$

$$\frac{\mathrm{d}\mathbf{U}_i^*}{\mathrm{d}\tau} + \mathbf{R}_i^* \left(\mathbf{U}_i^* \right) = 0 \tag{12}$$

An explicit multi-stage Runge-Kutta scheme [21] is applied to march Eq. (12) from pseudo time level $n\Delta\tau$ to level $(n+1)\Delta\tau$,

$$\mathbf{U}_i^{(0)} = (\mathbf{U}_i^*)^n \tag{13a}$$

$$\mathbf{U}_i^{(1)} = \mathbf{U}_i^{(0)} - \alpha_1 \Delta\tau_i \mathbf{R}_i^* \left(\mathbf{U}_i^{(0)}\right) \tag{13b}$$

$$\vdots \tag{13c}$$

$$\mathbf{U}_i^{(m)} = \mathbf{U}_i^{(m-1)} - \alpha_m \Delta\tau_i \mathbf{R}_i^* \left(\mathbf{U}_i^{(m-1)}\right) \tag{13d}$$

$$\vdots \tag{13e}$$

$$\mathbf{U}_i^{(p)} = \mathbf{U}_i^{(p-1)} - \alpha_p \Delta\tau_i \mathbf{R}_i^* \left(\mathbf{U}_i^{(p-1)}\right) \tag{13f}$$

$$(\mathbf{U}_i^*)^{n+1} = \mathbf{U}_i^{(p)} \tag{13g}$$

More details of the dual time-stepping method can be found in the work of Jameson [21] .

2.3. *Dynamic cloud technique*

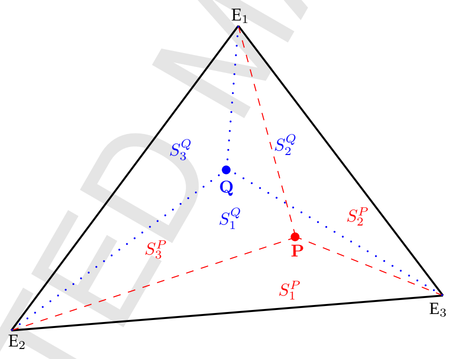


Figure 1: Determination of the mapping coefficients ($a_i^P = S_i^P/S$, $a_i^Q = S_i^Q/S$).

As is known, the computational domain is usually defined by physical boundaries (e.g. solid walls) and artificial boundaries (e.g. far-field boundaries for external flows). In order to adjust the distribution of discrete points to accommodate the motion of moving boundaries, a simple Delaunay graph mapping approach proposed by Liu et al. [22] is employed in the present work.

In order to generate the Delaunay graph to overlay the whole flow domain, we first need to select some representative (or all) boundary points. For this given set of boundary points, there exists a unique triangulation known as the Delaunay criterion [22]. Since the Delaunay graph covers the whole solution domain, every discrete point can be located in a triangle element of the graph. Such triangle is named the host element for the point, and it can be used to redistribute the nodes inside it. The essential idea to manipulate the position of a single point is shown in Figure 1, where a point P lies inside a Delaunay graph element T with three vertices notated as E_1, E_2 and E_3. The vertices E_1, E_2 and E_3 are basically chosen from the boundary points $\{E^{\mathrm{B}}\} = \{E^w\} \bigcup \{E^f\}$ (the superscript w stands for solid wall and f indicates far field boundary) in a computational domain. The coordinates of P can be expressed as

$$x_P = \sum_{i=1}^{3} a_i^P x_{E_i}, \quad y_P = \sum_{i=1}^{3} a_i^P y_{E_i} \tag{14}$$

where (x_{E_i}, y_{E_i}) are the Cartesian coordinates of vertex E_i. If S is the area of T, and $S_i (i = 1, 2, 3)$ are the areas of the sub-triangles shown in Figure 1, then $a_i = S_i/S (i = 1, 2, 3)$. For point P, the areas are given by

$$S_1 = \frac{1}{2}\begin{vmatrix} x_p & y_p & 1 \\ x_{E_2} & y_{E_2} & 1 \\ x_{E_3} & y_{E_3} & 1 \end{vmatrix}, \; S_2 = \frac{1}{2}\begin{vmatrix} x_p & y_p & 1 \\ x_{E_3} & y_{E_3} & 1 \\ x_{E_1} & y_{E_1} & 1 \end{vmatrix}, \; S_3 = \frac{1}{2}\begin{vmatrix} x_p & y_p & 1 \\ x_{E_1} & y_{E_1} & 1 \\ x_{E_2} & y_{E_2} & 1 \end{vmatrix}, \; S = \frac{1}{2}\begin{vmatrix} x_{E_1} & y_{E_1} & 1 \\ x_{E_2} & y_{E_2} & 1 \\ x_{E_3} & y_{E_3} & 1 \end{vmatrix} \tag{15}$$

In the generated Delaunay graph, the far field boundary points may stay stationary, while the solid wall points representing the geometrical configuration are allowed to move in the flow field. Hence, the Delaunay graph will also move/deform. After moving/deforming the graph, a new set of coordinates is obtained for the vertices of each graph element. It is requested that the distribution of the point in a graph element keeps the area ratio coefficients a_i as constants during graph movement [22]. Therefore, the new coordinates of point P can be determined as

$$x'_P = \sum_{i=1}^{3} a_i^P x'_{E_i}, \quad y'_P = \sum_{i=1}^{3} a_i^P y'_{E_i} \tag{16}$$

where (x'_{E_i}, y'_{E_i}) are the new coordinates for graph element nodal points. In other words, the area ratio coefficients a_i can be used to relocate the point P in the domain [22].

Such kind of procedure is illustrated in Figure 2, where point P is mapped to point P' after the movement of graph element. Since the distribution of these points is controlled by the constant area ratio coefficients throughout the graph movement, this is very useful to keep the relative position of a point between its each surrounding node. Figure 3 shows the relocation of five neighbouring nodes in three adjacent graph elements. The relative positions between P_1 and its surrounding points can be maintained if the amplitude of graph movement is not extremely large. Consequently, if these five nodes form a meshless point cloud initially, it can be used throughout the graph movement without frequently changing its member nodes.

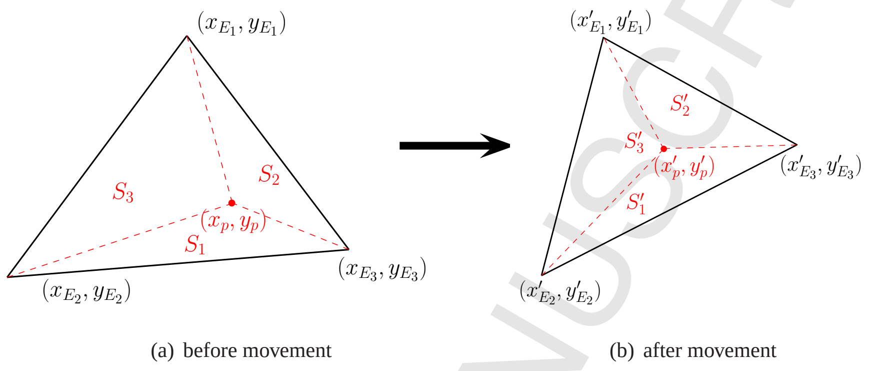


Figure 2: Relocation of a point P in a Delaunay element during the graph movement. The new area ratio coefficients $a'_i = S'_i/S'$ are equal to the original values $a_i = S_i/S$.

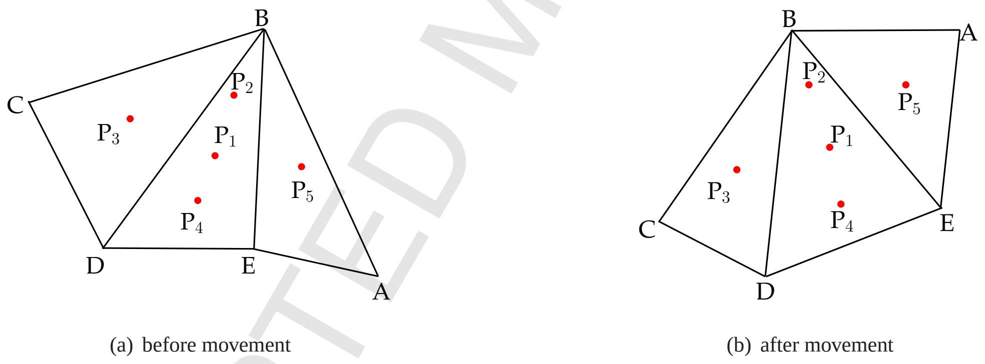


Figure 3: Relocation of five points in neighbouring Delaunay elements during the graph movement.

The basic steps of the dynamic cloud method are listed as follows

1. Input the cloud of points.
2. Generate a Delaunay graph $G = \bigcup_{k=1}^{K} T_k$ for the boundary points $\{E^{\mathrm{B}}\}$.
3. For each internal field point P, search the host element T^P inside which it lies.
4. Compute the mapping coefficients a_i^P for point P.
5. Moving the Delaunay graph.

6. Relocating the points in the graph.

The information for step 2, 3 and 4 only needs to be computed once and stored in the computer memory before the flow simulation starts.

Examples are given here to illustrate the procedure of implementing dynamic cloud for a NACA0012 aerofoil and a NACA64A010 aerofoil with pitch motions, respectively. We first need to input the cloud of points for the aerofoil, then generate a Delaunay graph G of boundary points $\{E^B\}$ as shown in Figure 4 and 6. If we rotate the aerofoil about its quarter for 30^o, the coordinates of aerofoil surface points are updated as $(x_{E^w_{new}}, y_{E^w_{new}})$. Substitute the new coordinates to Eq (14), then the coordinates of all the internal field points will be updated as illustrated in Figure 5 and 7. The advantage of this approach is that intensive iterations requested by other methods like spring-analogy technique are avoided since it only needs very simple linear algebraic operations. Moreover, it can effectively handle cases with relatively large displacements without penetrating solid boundaries as shown in Figure 5 and 7. Meanwhile, solid boundary penetration may occur when other strategies such as spring analogy are utilised to adjust the clouds of points for these cases (see Figure 3 of [8]).

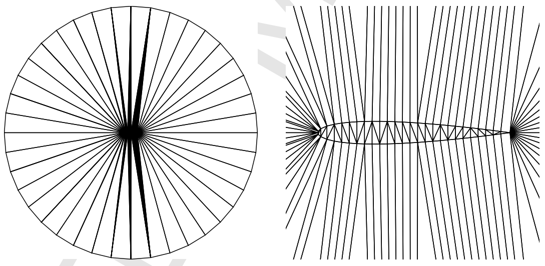
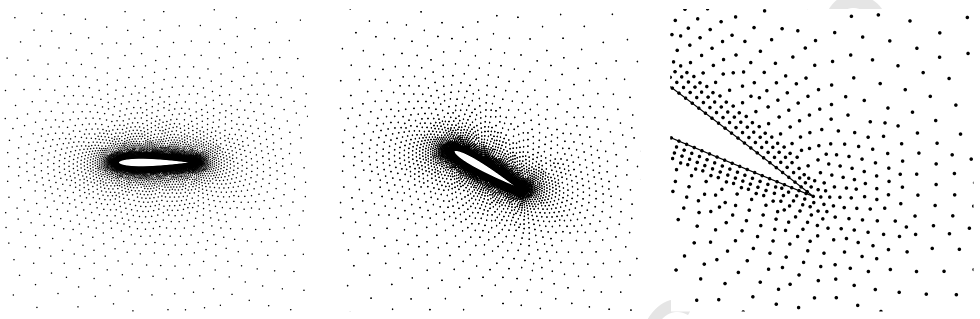


Figure 4: Global and close-up views of a Delaunay graph for a single NACA0012 aerofoil.

3. Implementation on the GPU

3.1. The procedure

As is well known, a GPU computing program usually needs the CPU to input the information from the hard drive (or elsewhere) and pre-process the data. The data is then sent from the CPU to the GPU. The complete or partial computing task is off-loaded to the GPU. Once the task is finished, the result is transferred back to the CPU. This general procedure for a GPU based CFD program is illustrated in Figure 8.



(a) Initial cloud (b) Mapped cloud (30^o pitch) (c) Trailing edge close-up view

Figure 5: Dynamic cloud for a single NACA0012 aerofoil with pitch motion.

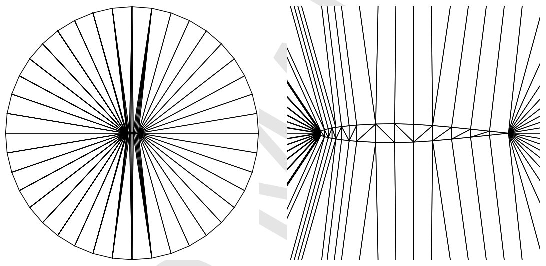
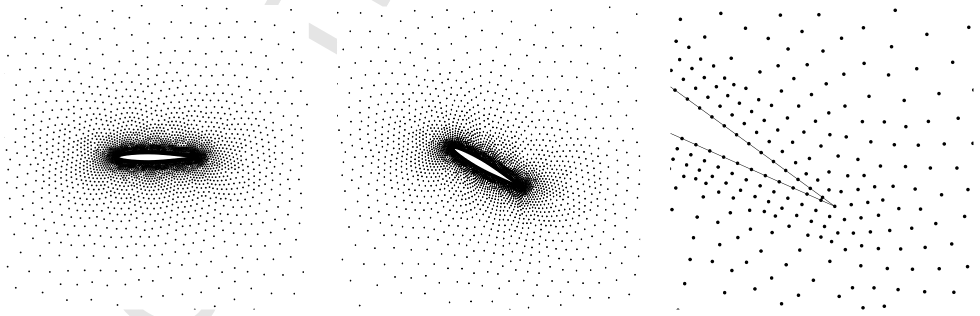


Figure 6: Global and close-up views of a Delaunay graph for a single NACA64A010 aerofoil.



(a) Initial cloud (b) Mapped cloud (30^o pitch) (c) Trailing edge close-up view

Figure 7: Dynamic cloud for a single NACA64A010 aerofoil with pitch motion.

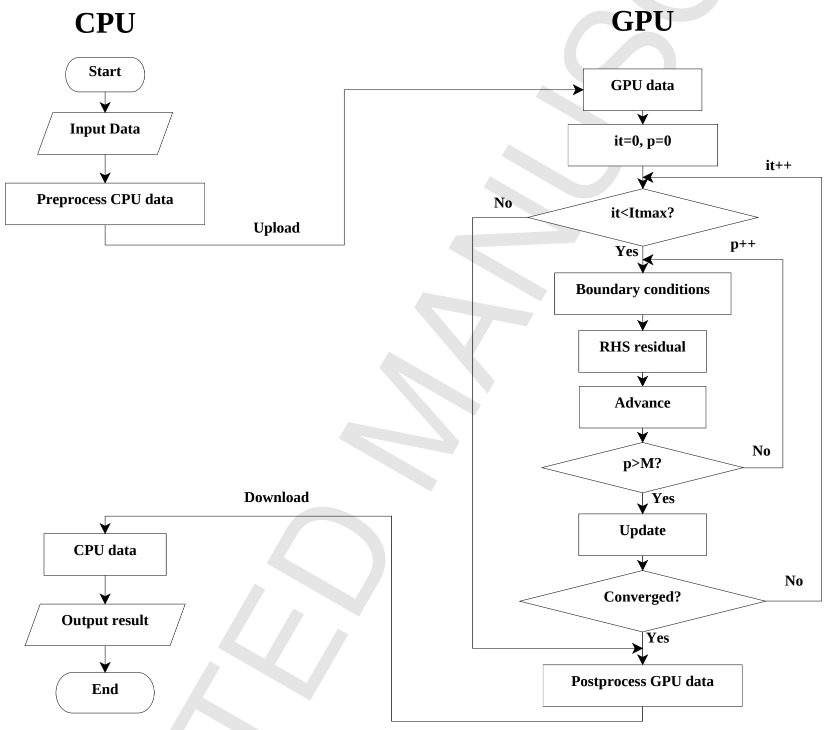


Figure 8: A general procedure of GPU computing program. The CPU is responsible to input/output the data, the GPU is recruited to tackle the computing intensive task.

Our original single-core CPU based meshless solver for unsteady flows was exclusively coded in Fortran 90. It does not adopt any third-party numerical libraries. The main program written in Fortran 90 is presented in Listing 1. The most time-consuming portion is the flow solver (the function ALE_solver), whose major steps are shown in Listing 2. We use CUDA C [23] to re-program this part of the code. Other portions of the solver including data input/output and flow field initialisation are kept the same as shown in Figure 8 and Listing 1. The CUDA code for the flow solver is presented in Listing 3.

Listing 1: The Fortran main program

```fortran
program main
  call InputData(CPU_data)
  call FlowInit(CPU_data)
  call DelaunayGraphGen(CPU_data)

  ! solve the ALE equation on GPU
  call ALE_solver_GPU(CPU_data)

  call ResultOutput(CPU_data)
end program main
```

Listing 2: Fortran code for the ALE solver

```fortran
subroutine ALE_solver_CPU(CPU_data)
  physical_time: do i=0,TotalTimeStep
    call MoveSolidBound(CPU_data)
    call RelocatePointInGraph(CPU_data)

    ! pseudo time iteration
    pseudo_time: do it=1,InerItMax
      do im=1,NRK
        call residual(CPU_data)
        call advance(CPU_data)
      enddo

      call FlowUpdate(CPU_data)
    enddo pseudo_time
  enddo physical_time
end subroutine ALE_solver()
```

Listing 3: CUDA code for the ALE solver

```c
#ifdef __cplusplus
extern "C"
#endif
```

```
void ALE_solver_GPU(CPU_data)
{
  //upload data from CPU to GPU;
  cudaMemcpy(CPU_data, GPU_data, data_size, cudaMemcpyHostToDevice);

  //physical time
  for(int i=0;i<TotalTimeStep;i++){
      Cuda_MoveSolidBound<<<block,grid>>>(GPU_data);
      Cuda_RelocatePointInGraph<<<block,grid>>>(GPU_data);

      //pseudo time iteration
      for(int it=1;it<=InerItMax;it++){
          for(int im=1;im<=NRK;im++){//Runge-Kutta stepping
              CUDA_residual<<<block,grid>>(GPU_data);
              CUDA_advance<<<block,grid>>>(GPU_data);
          }
          CUDA_FlowUpdate<<<block,grid>>>(GPU_data)
      }
  }

  //offload data from GPU to CPU;
  cudaMemcpy(GPU_data, CPU_data, data_size, cudaMemcpyDeviceToHost);
}
```

3.2. Hierarchy of CUDA thread and memory

Different with MPI or OpenMP, which provides coarse-grain parallelism, CUDA offers fine-grain parallelism as thousands (or even many more) of lightweight threads can be launched on the graphic hardware. Each thread can be properly used to deal with a computational stencil (a mesh cell for structured/unstructured grid methods, or a cloud of points for meshless methods). It can access the data stored in the memory and carry out algebraic operations on the data, etc. CUDA uses grid and blocks to manage these threads, and every thread has a unique index.

Figure 9 presents a simple layout of the CUDA thread and memory hierarchy. The CPU is usually considered as the host, and the GPU is called the device. The data stored in the host memory is firstly copied to the global memory of the device. On the device, every thread can access the global memory. All the threads in the same block can access the shared memory belonging to this block, each thread can have its own private registers. Proper use of shared memory will greatly benefit the program performance especially when there are a lot of data reuse [23]. It works well for structured grid methods, which can address memory with regular patterns.

However, for indirect addressing model based applications such as unstructured grid methods, it is difficult to utilise the shared memory due to the irregular memory access pattern. As the meshless solver indirectly addresses the data, we do not adopt the shared memory in the current work. Using constant memory provided on GPUs may reduce the required memory bandwidth [23]. In the present work, we use constant memory to store important parameters such as the ratio of specific heat coefficients γ. Adequate threads can be created on the device to accomplish a one-to-one mapping of the CFD grid. A CUDA grid can have up to three dimensions to manage the thread blocks and map the corresponding CFD grid.

In the present work, Both the thread block and grid are set to be one-dimensional. The number of threads N_t in a block is usually set as times of 32, which is the size of a warp, according to the CUDA C programming guide [23]. If the total number of meshless clouds of points is N_c in the whole flow field, then the number of thread blocks N_b can be chosen as an integer number no less than N_c/N_t. Accordingly, the blocksize is set as N_t and the gridsize is N_b for the CUDA kernel functions shown in Listing 3. The parameter N_t can be tuned in order to obtain the optimal performance.

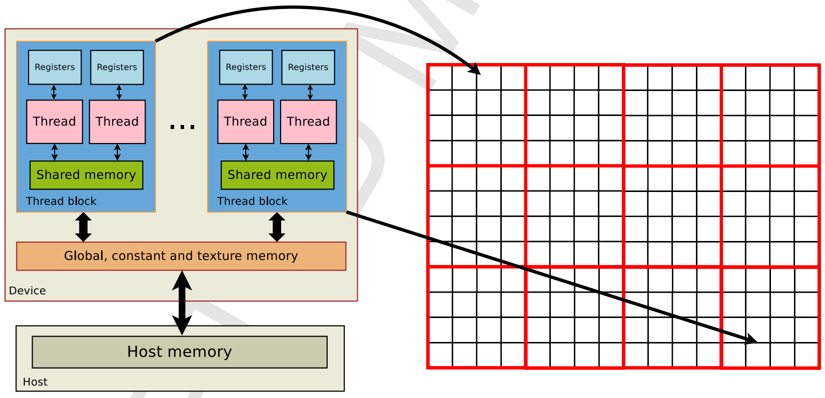


Figure 9: CUDA memory and thread hierarchy. A one-to-one mapping of the CFD grid can be established if adequate threads are created on the CUDA device.

3.3. *Data structure*

Fortran derived types were used in our original program to encapsulate data associated with the same cloud of points. This made the program quite concise and readable. Although structure of arrays (SoA) is more coalesced in computer memory than arrays of structure (AoS), completely substituting AoS with SoA brings penalties to code debugging and development of an existing large program composed of thousands of lines or even more. In case researchers prefer not to re-design their programs from scratches but just want to obtain satisfactory performance speedup ($> 10\times$) by using GPUs, AoS can still be used if this does not harm the performance too much.

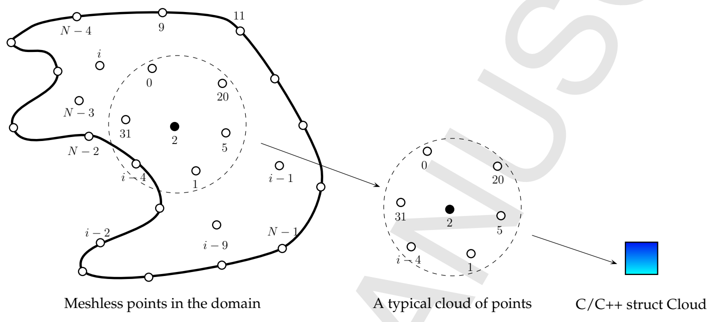
For the current study, the strategy to manage the data on GPUs is shown in Figure 10(a). Arrays of C/C++ structure are used to store the information of each cloud of points, which include the number of points in the cloud, serial index and geometric scalar coefficients of every satellite. Flow variables and/or their gradients can be encapsulated in a C/C++ structure of point. Attentions need to be paid to these C/C++ structures, they should be compatible with the Fortran derived types, which means the variables in a C/C++ structure must be in the same sequence as those in a Fortran derived type. Otherwise, the data passed from a Fortran subroutine to a C/C++ function will possibly be corrupted by reading/writing a false address in the memory.

Figure 10(b) presents a mapping of the CUDA threads to all the computational stencils, in which every CUDA thread is responsible to deal with a corresponding cloud of points. To access a piece of data stored in the GPU global memory from a single thread in a block, the global index of the thread needs to computed (page 9 of [23]). In most cases, the number of threads in a block needs to be tuned to optimise the performance of a GPU program.

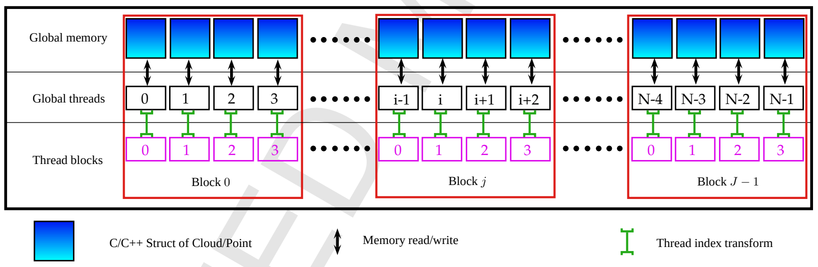
3.4. CUDA thread race conditions

Attention needs to be paid to the underlying numerical method when we try to port the CPU functions to the GPU. Some well-founded computer algorithms can not be directly converted to CUDA kernel functions if they are not inherent parallel. A typical problem is the racing of CUDA threads, in which no less than two threads attempt to access the same memory location concurrently and at least one access is write. This may produce unexpected results [24].

To reveal this kind of problem, here we choose the Laplace equation as an example. If the Laplace equation is solved on a uniform structured grid with the Gauss-Seidel iteration method, the value at a CFD grid point P_C can be obtained through averaging its four closest neighbours as shown in Figure 11 (left part). However, direct converting this function to CUDA is not favourable. When a thread is trying to update the value at P_c with a write operation, other four threads may read this piece of memory at the same time. Consequently, there is a conflict between the write and read operations, and this will lead to an unpredictable result. Hence, the



(a) Encapsulation of clouds of points in C/C++ structure



(b) GPU global memory and thread hierarchy

Figure 10: Data arrangement for clouds of points in the GPU memory. To read/write the GPU global memory, a global index needs to be calculated by using the local thread index in a block and the number of threads in a block.

algorithm needs to be replaced with a chequerboard Gauss-Seidel iteration (right part of Figure 11) or other methods. The example is given here to emphasise the importance of the concept of parallelism for GPU computing, which we should bear in mind throughout the work.

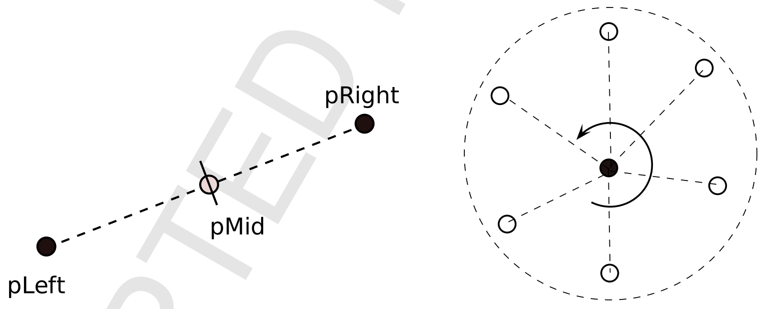
Serial CPU function

```
void GS(float *A, const int I,
            const int J)
{
  for(int j=0;j<J;j++)
    for(int i=0;i<I;i++){
      int C = j*I+i; //Center point
      int L = C-1; //Left point
      int R = C+1; //Right point
      int D = C-I; //Lower point
      int U = C+I; //Upper point
      A[C]=(A[L]+A[R]+A[D]+A[U])/4;
    }
}
```

CUDA kernel function

```
__global__ void GS_RED_CUDA(float *A,
 const int I, const int J, const int RED)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  int j=blockIdx.y*blockDim.y+threadIdx.y;
  int C = j*I+i; //Center point
  int L = C-1; //Left point
  int R = C+1; //Right point
  int D = C-I; //Lower point
  int U = C+I; //Upper point
  if(i<I-1&&j<J-1&&(i+j)%2==RED)
    A[C]=(A[L]+A[R]+A[D]+A[U])/4;
}
```

Figure 11: A five-point Gauss-Seidel iteration method, serial Vs parallel. Chequerboard GS iteration is used on the GPU to prevent thread racing problem, which will cause conflict read/write operations.



(a) P_{mid} and the pair: P_{left}–P_{right} (b) a solution point and its satellites

Figure 12: Two strategies to loop over all the solutions points in the flow field. Left: loop over every point pair; Right: loop over every solution point, aided with a small loop over its surrounding nodes (satellites).

Listing 4: A Fortran subroutine to compute spatial derivatives $\partial\rho/\partial x$. Point-pair loop, please refer to Figure 12(a).

```
subroutine FlowDerivativeCPU(CPU_data)
  integer::pmid,pLeft,pRight

  do pmid=1,PmidTotal
```

```
    pLeft=Left(pmid)
    pRight=Right(pmid)
    dRhodx(pLeft)=dRhodx(pLeft)+alphaLeft(pmid)*rho(Right)
    dRhodx(pRight)=dRhodx(pRight)+alphaRight(pmid)*rho(Left)
  end do
end subroutine FlowDerivativeCPU
```

Listing 5: A thread-racing GPU kernel function for $\partial\rho/\partial x$. Point-pair loop, please refer to Figure 12(a).

```
__global__ void FlowDerivativeGPU_A(GPU_data)
{
  //index midpoint
  int pmid=blockDim.x*blockIdx.x+threadIdx.x;

  //derivative of density
  pLeft=Left[pmid]
  pRight=Right[pmid]
  dRhodx[pLeft]+=alphaLeft[pmid]*rho[pRight];
  dRhodx[pRight]+=alphaRight[pmid]*rho[pLeft];
  }
}
```

Listing 6: A thread-racing-free GPU kernel function for $\partial\rho/\partial x$. Hierarchy loop, please refer to Figure 12(b)

```
__global__ void FlowDerivativeGPU_B(GPU_data) { //index meshless cloud int
  i=blockDim.x*blockIdx.x+threadIdx.x;

  //derivative of density
  for(int j=0;j<M;j++){
    dRhodx[i]+=alpha[j]*Rho[C[i][j]];
  }
}
```

Similarly, all the GPU kernel functions developed for the meshless dynamic cloud method must preclude thread race conditions. When computing spatial derivatives of a mathematical function or the convective fluxes on the CPU, we can loop over every pair of points P_{left} and P_{right} as shown in Figure 12(a). This method can be simply named point-pair loop (PPL). The corresponding Fortran code is shown in Listing 4. Directly porting this code to the GPU will lead to a thread-racing kernel function as shown in Listing 5.

To prevent race conditions, we may choose to have a hierarchy loop (HL), which has an outer loop for every solution point and an inner small loop for its surrounding nodes within the same point cloud. The hierarchy loop shown in Figure 12(b) can successfully prohibit race conditions and is suitable for parallel computing. The corresponding thread-racing-free GPU kernel function is shown in Listing 6. The hierarchy loop used in the present work is very similar to the redundant computation technique proposed by Corrigan et al. (Section 3.2 of [16]).

On the CPU, the performance of PPL is much better than HL. This can be seen from Table 1, which lists the running time costs of the HL-based and PPL-based functions for computing the derivative $\partial\rho/\partial x$ on a single CPU core. Compared to PPL, HL needs extra 39.2% ~ 52.3% CPU time. However, HL is parallel friendly while PPL will cause race conditions on the GPU.

Table 1: CPU run time costs of the functions for computing $\partial\rho/\partial x$ with hierarchy loop (HL) and point-pair loop (PPL). Both functions are executed 50000 times on a single CPU core. Compared to PPL, HL needs extra 39.2% ~ 52.3% CPU time.

Case	Number of points	HL cost (s)	PPL cost (s)	HL/PPL
1	3142	3.62	2.60	139.2%
2	5557	6.55	4.62	141.8%
3	8993	11.19	7.59	147.4%
4	15198	17.64	11.58	152.3%

3.5. Hardware and software platform

All the following numerical simulations presented in this paper are performed on a Linux workstation equipped with a Intel Xeon E5645 CPU (12M cache, 2.40 GHz, 6 cores) and 24GB RAM. The maximum power consumption of the CPU is 80w, so this is roughly 13.33w for a single core. Two NVIDIA graphics cards Quadro 2000 and Tesla C2075 are installed on the workstation. The specifications of the two graphic cards are listed in Table 2. The operating system is Ubuntu 10.10 64-bit. We use PGI Fortran and NVCC to compile Fortran and CUDA C codes respectively. The optimisation level for each compiler is set to `-O3` without debugging and profiling options. Two libraries *lstdc++* (C++ run time library) and *libcudart* (CUDA run time library) need to be linked to the object files in the final assembling stage in order to guarantee the executable program be generated successfully.

4. Numerical results

In order to verify our method on the GPU, we start with compressible steady flows over a fixed NACA0012 aerofoil. Then we extend the method to unsteady flows past oscillatory NACA0012 and NACA64A010 aerofoils. No-penetration condition is adopted on the aerofoil surface and non-reflection condition is applied on the far-field boundary. Two important parameters *speedup* and *energy consumption ratio* will be used in the following sections to indicate the performance of GPUs regarding the computing speed and energy efficiency. *Speedup* is defined as the ratio of CPU running time to GPU running time. *Energy consumption ratio* is calculated by dividing the GPU energy consumption with the CPU energy consumption. We need to point out that the energy consumption is obtained through multiplying the processor's maximum power consumption with its running time. Throughout our work, the CPU program is executed on a single core, therefore the actual CPU energy consumption is divided by the number of cores inside it .

4.1. *Steady flows over a NACA0012 aerofoil*

The flow condition for this case is $M_\infty = 0.8$ with angle of attack $\alpha = 1.25^\circ$. The number of points distributed in the domain is $5,557$. This is a classical test to benchmark the numerical method's capability to capture shock waves correctly regarding the position and strength. On the upper surface of the aerofoil, a strong shock appears near 0.6 chord length. On the lower surface, a weak shock forms around 0.375 chord length. These shock waves are clearly shown in the right part of Figure 13. The pressure coefficients around the aerofoil surface are depicted in the left part of the figure, in which the solid line is the present work computed on Tesla C2075, the square dot is Pulliam and Steger's result [25]. A cell-centred finite volume method with the JST (Jameson-Schmidt-Turkel) scheme [26] is also utilised to solve this problem and the solution is represented by the cross. Obviously, the present result agrees well with the other solutions. The CPU and GPU running time costs for this case are presented in Table 3. It is clearly shown that we achieve speedups of 10.86 and 32.92 on Quadro 2000 and Tesla C2075 cards respectively. At the same time, it is easy to find that Quadro and Tesla are about twice energy efficient of a Intel Xeon E5645 core as indicated in Table 3.

In order to investigate the performance of these two GPUs when different number of points are used, we carry out a systematic benchmark of the meshless solver. The number of points distributed in the domain varies from two thousands to one million. The meshless cloud with one million points occupies about 220MB memory on the GPUs. Therefore, the Quadro 2000 card has the capacity to handle about four million points and the Tesla C2075 card can handle about 25 million points. For cases with even more points, multi-GPU strategy needs to be considered. However this is beyond the scope of the current paper. We are planning to investigate this issue in our future work.

Figure 14 shows the running time speedups of the two GPUs. Quadro 2000 gives a good speedup rising from 9.5 to more than 13 when the number of points is increased, its energy consumption ratio decreases from 48.78% to 34.84% (the lowest value is 33.78%). On Tesla C2075, at the same time, the running time speedup gradually rises from 26 to 56 with a corresponding energy consumption ratio dropping from 65.79% to 30.49%.

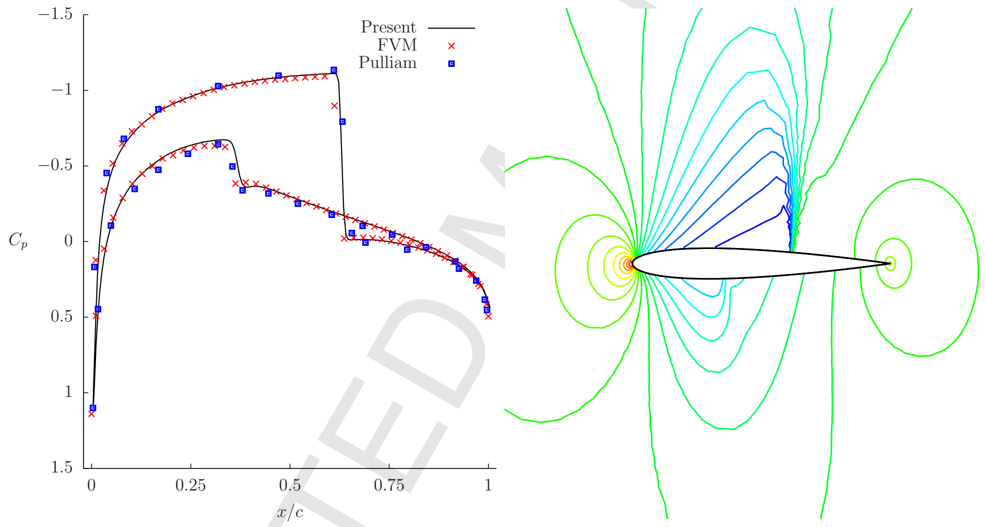


Figure 13: Transonic steady flows over the NACA0012 aerofoil for $M_\infty = 0.8$, $\alpha = 1.25°$. Left: the pressure coefficients around the aerofoil computed by the meshless solver on Tesla C2075 GPU, finite volume method (with the JST scheme [26]) and Pulliam and Steger's result [25]. Right: pressure contours in the flow field computed by the present method. The number of points distributed in the domain is 5, 557.

4.2. *Unsteady flows over an oscillatory NACA0012 aerofoil*

A standard AGARD test case of an oscillating NACA0012 aerofoil is considered here. For this case, the aerofoil rotates about its quarter chord with the instantaneous angle of attack given

Table 2: Specifications of Intel Xeon E5645 CPU, NVIDIA Quadro 2000 and Tesla C2075 graphic cards. Throughout our work, the running time and energy consumption of the CPU refer to a single core.

	Intel Xeon E5645	Quadro 2000	Tesla C2075
Clock Rate	2.4 GHz	1.25 GHz	1.15 GHz
Global memory	24 GB	1 GB	6 GB
Shared memory	–	48 KB	48 KB
Registers per block	–	32768	32768
Number of multiprocessor	1	4	14
Cores per multiprocessor	6	48	32
Total number of cores	6	192	448
Compute capability	–	2.1	2.0
Max power consumption	80 w	62 w	225w
Max power consumption per core	13.33 w	0.32 w	0.50 w

Table 3: CPU and GPU running time costs of the meshless solver for steady flows over the NACA0012 aerofoil for $M_\infty = 0.8$, $\alpha = 1.25°$. The running time and energy consumption of the CPU refer to a single core. (The number of points distributed in the domain is $5,557$. The number of Runge-Kutta iterations is fixed to $10,000$ for comparison purpose.)

Device	Intel Xeon E5645	Nvidia Quadro 2000	Nvidia Tesla C2075
Wall time(s)	465.80	42.89	14.15
Speedup	–	10.86	32.92
Energy consumption	100%	42.74%	51.28%

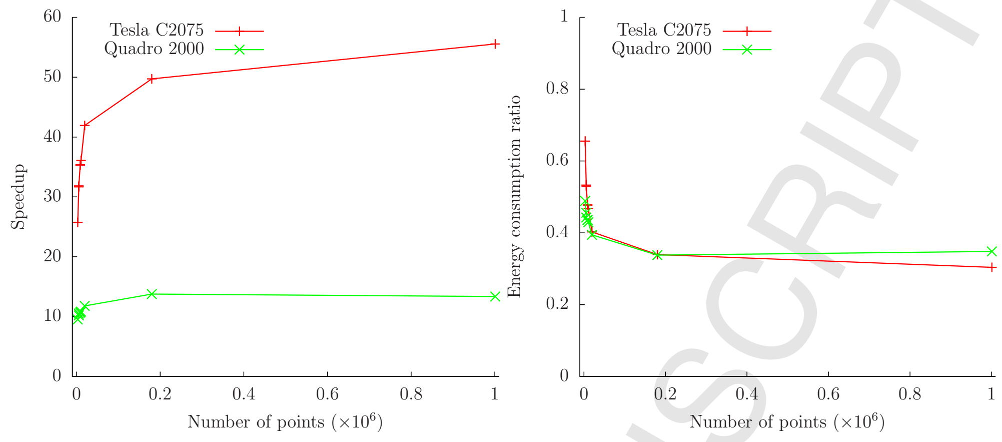


Figure 14: Systematic benchmark of the meshless solver for steady flows on two CUDA supported graphic cards. Left: running time analysis; Right: energy consumption analysis. Speedup is defined as the ratio of CPU running time to GPU running time. The running time and energy consumption of the CPU refer to a single core. The running time cost for numerical simulations can be dramatically reduced by more than an order of magnitude on GPUs, which are also more energy efficient.

by

$$\alpha(t) = \alpha_m + \alpha_0 \sin(\omega t) \tag{17}$$

where α_m is the mean angle of attack, α_0 is the pitching range and ω is the angular frequency. The angular frequency ω is related to the reduced frequency given by

$$\kappa = \omega c / 2U_\infty \tag{18}$$

where c is the chord length of the aerofoil and U_∞ is the free-stream speed of the flow. The case is solved with the following conditions: $M_\infty = 0.755$, $\alpha_m = 0.016°$, $\alpha_0 = 2.51°$, $\kappa = 0.0814$. The computational domain is discretised by $5,557$ points, among which 337 nodes are distributed on the aerofoil. Prior to performing the unsteady simulation, a steady flow solution is firstly computed with the specified flow conditions $M_\infty = 0.755$ and $\alpha_m = 0.016°$. The simulation of the unsteady flow field is initiated once the steady solution converges. The unsteady computation is carried out using 64 real-time steps in every oscillation period. Within each real-time step, it takes about 500 to 600 iterations to reduce the residual by more than four orders of magnitude. For this case, we compute ten oscillation periods in total.

Instantaneous lift coefficient C_L and moment coefficient C_M versus angle of attack during the oscillatory motion are presented in Figure 15, and they agree well with Landon's experiment

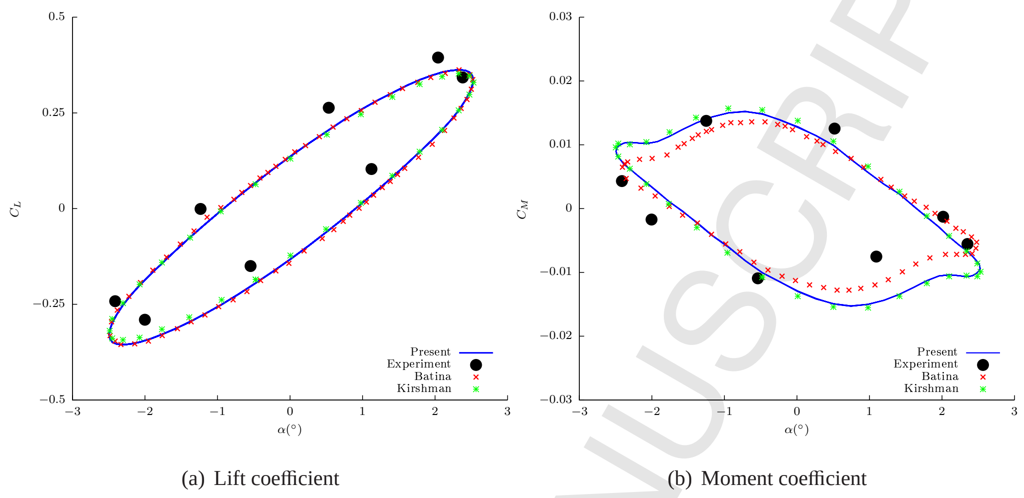


Figure 15: Comparison of lift and moment coefficients with Landon's experiment [27] (black dot), Batina's numerical result [28] (red cross) and Kirshman's computation [29] (green star) for the oscillatory NACA0012 aerofoil.

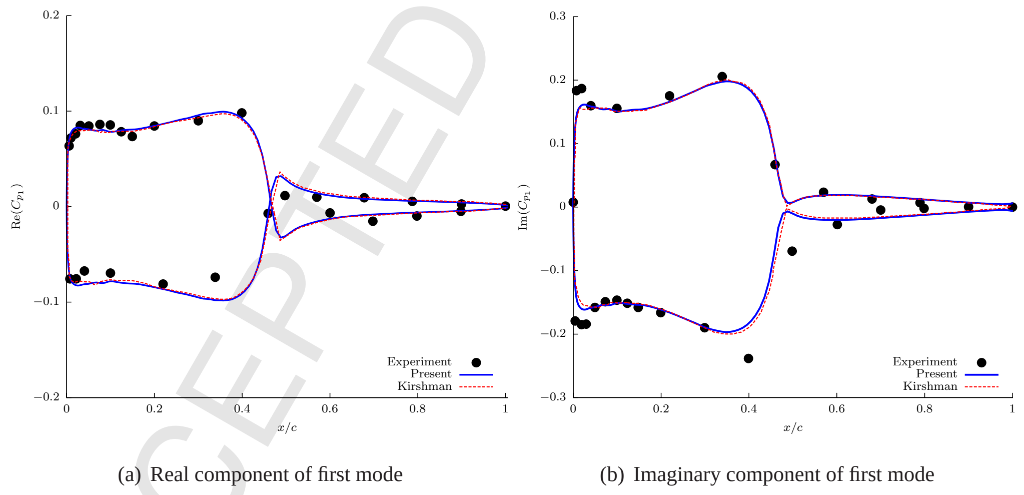
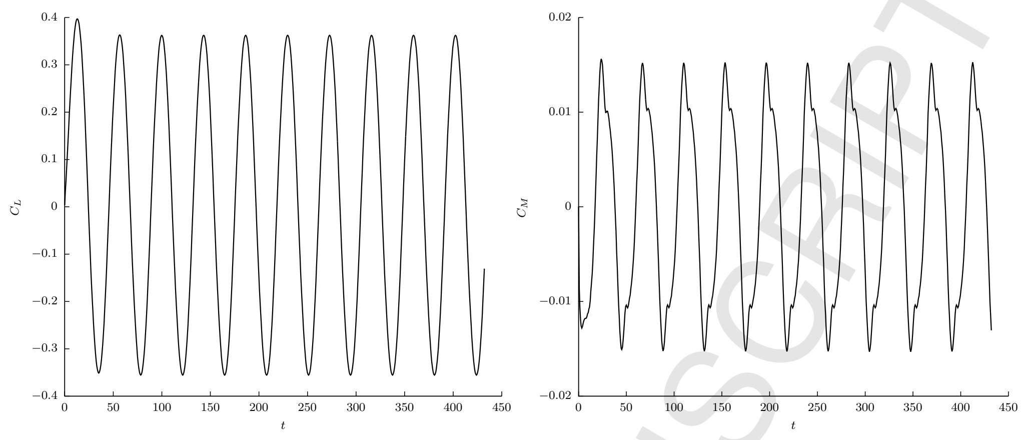


Figure 16: Fourier decomposition of the surface pressure coefficients for an oscillatory NACA0012 aerofoil (black dot is Landon's result [27], dashed line is Kirshman's solution [29]).



(a) Lift coefficient (b) Moment coefficient

Figure 17: Time history of the computed lift and moment coefficients for NACA0012 aerofoil.

[27], Batina's computation [28] and Kirshman's simulation [29]. Figure 16 illustrates the first Fourier mode of the surface pressure coefficient, where the real component is depicted in the left part and the imaginary component is shown in the part. Apparently, the present result is in a good agreement with the experiment [27] and Kirshman's computation [29]. Figure 17 shows the time history of the lift and moment coefficients for ten oscillation cycles. The periodic phenomenon is well established from the second cycle as shown in the figure.

We list the running time costs for different compute hardwares in Table 4. It takes the CPU 30.41 seconds to compute a real time step. This is shortened to 3.22 seconds by the Quadro 2000 graphic card with a speedup of 9.44. The Tesla GPU achieves a speedup of 29.81 as it only spends 1.02 seconds. More than fives hours' CPU time can be dramatically reduced to less than eleven minutes by Tesla C2075 for the total ten cycles as shown in Figure 21. The energy consumption ratio of Quadro card is 49.26% and it is 56.50% for Tesla card.

4.3. *Unsteady flows over an oscillatory NACA64A010 aerofoil*

Another standard AGARD test case of an oscillating NACA64A010 aerofoil is considered. For this test, the aerofoil rotates about its quarter chord with the instantaneous angle of attack indicated by the same equation (17). The angular frequency ω is related to the reduced frequency κ defined by Eq. (18). This case is simulated with the following conditions: $M_\infty = 0.796$, $\alpha_m = 0.0°$, $\alpha_0 = 1.01°$, $\kappa = 0.202$. There are 4006 points in the flow domain and 200 nodes are

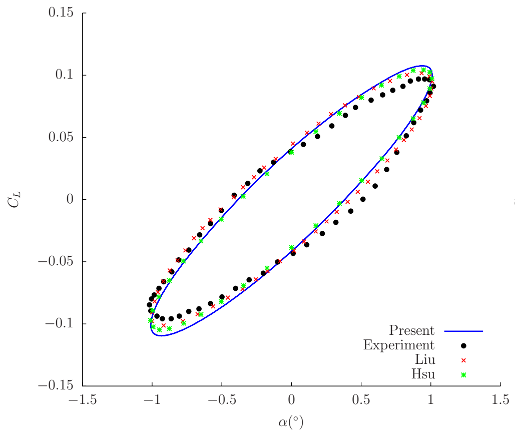
Table 4: Pseudo time iteration costs of the meshless solver for unsteady flows over the NACA0012 aerofoil. The running time and energy consumption of the CPU refer to a single core. (The number of points distributed in the domain is $5,557$. The maximum number of sub iterations is 1000.)

Device	Intel Xeon E5645	Nvidia Quadro 2000	Nvidia Tesla C2075
Wall time(s)	30.41	3.22	1.02
Speedup	–	9.44	29.81
Energy consumption	100%	49.26%	56.50%

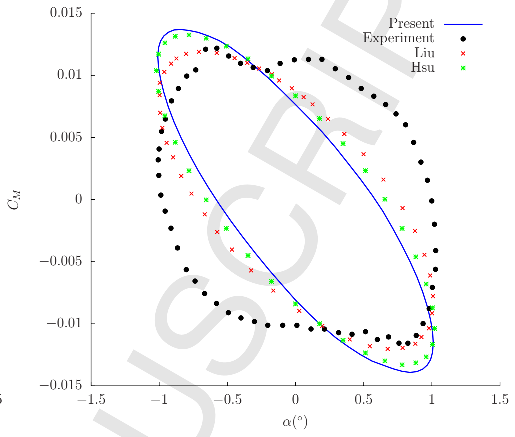
distributed on the aerofoil surface. Once the steady solution converges for M_∞ = 0.796 and α_m = 0.0°, the unsteady computation is started and kept for ten oscillation periods. Each period is divided by 64 chunks. For this case, it takes about 150 pseudo-time iterations for each real-time step to reduce the residual by four orders of magnitude.

Instantaneous lift coefficient C_L and moment coefficient C_M versus angle of attack during the oscillatory motion are presented in Figure 18. The present computed lift coefficient agrees well with Davis' experiment [30], Hsu & Jameson's inviscid solution [31, 32] and Liu & and Ji's viscous result [33]. Inspecting the moment coefficient, it's not difficult to find that there is a relative big discrepancy between the experiment and all the numerical computations. Our result is in a good agreement with Hsu & Jameson's solution, but apparently both of the inviscid solutions over predict the amplitude of moment coefficient. While Liu & Ji's viscous result is closer to the experiment regarding the minimum and maximum moment coefficients. The real and imaginary components of the first Fourier mode for the surface pressure coefficients are depicted in Figure 19. Apparently, our computation is in a satisfactory agreement with the experiment and other numerical solutions. Time history of the lift and moment coefficients is shown in Figure 20, the periodic phenomenon is well established from the third cycle.

The pseudo-time iteration costs are listed in 5. Quadro 2000 performs relatively well as it spends 0.588 seconds to compute a real-time step, which gives a speedup of 8.48 compared to the CPU. While it is very interesting to note that Tesla C2075 provides a 23.97$\times$ speedup, it takes this device only 0.208 seconds to complete a real-time step. Almost one hour's CPU work can be finished within three minutes as shown in Figure 21. Both Quadro 2000 and Tesla C2075 are more energy efficient than the CPU for this case.

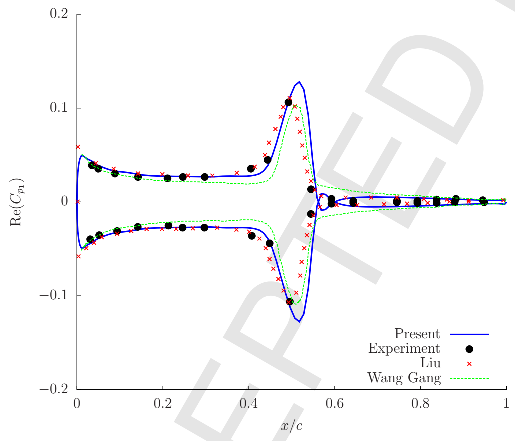


(a) Lift coefficient

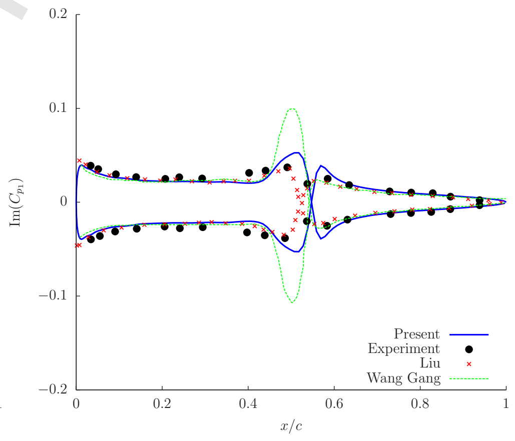


(b) Moment coefficient

Figure 18: Comparison of lift and moment coefficients with Davis's experiment [30] (black dot), Liu & Ji's numerical result [33] (red cross) and Hsu & Jameson's computation [31, 32] (green star) for the oscillatory NACA64A010 aerofoil.

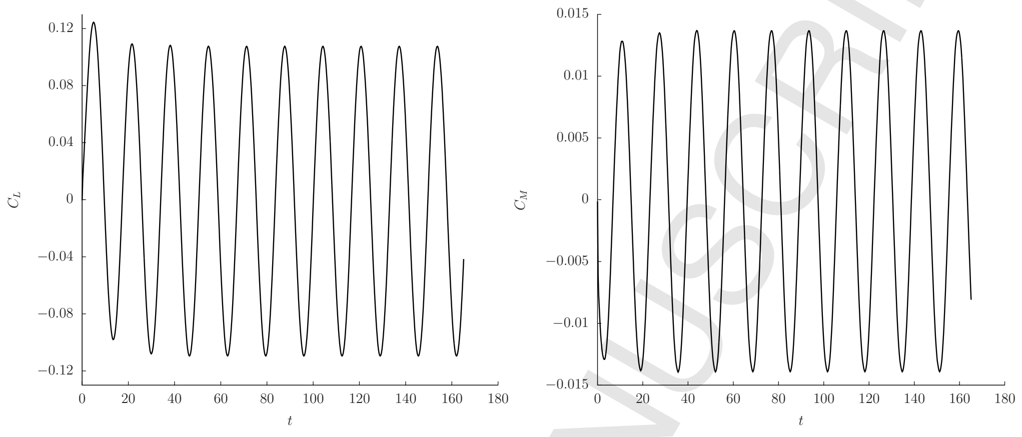


(a) Real component of first mode



(b) Imaginary component of first mode

Figure 19: Fourier decomposition of the surface pressure coefficients for an oscillatory NACA64A010 aerofoil (black dot is Davis's experiment [30], red cross is Liu and Ji's numerical result [33], green dashed line is Wang et al's computation [34]).



(a) Lift coefficient (b) Moment coefficient

Figure 20: Time history of the computed lift and moment coefficients for NACA64A010 aerofoil.

Table 5: Pseudo time iteration costs of the meshless solver for unsteady flows over the NACA64A010 aerofoil. The running time and energy consumption of the CPU refer to a single core. (The number of points distributed in the domain is $4,006$. The maximum number of sub iterations is 1000.)

Device	Intel Xeon E5645	Nvidia Quadro 2000	Nvidia Tesla C2075
Wall time(s)	4.985	0.588	0.208
Speedup	–	8.48	23.97
Energy consumption	100%	54.94%	70.42%

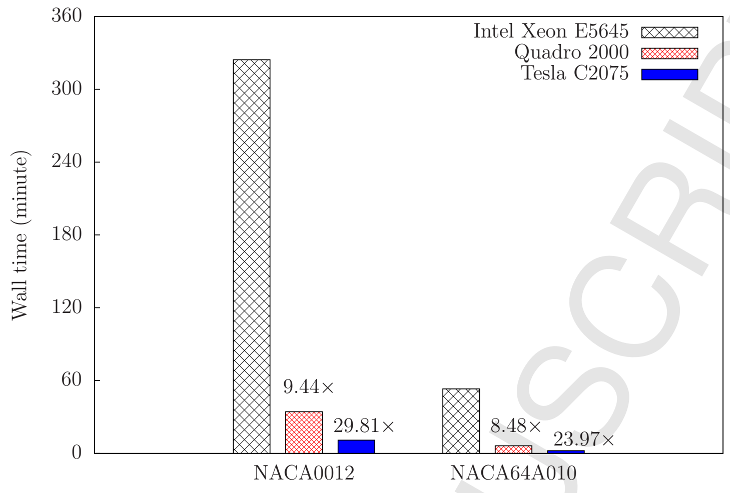


Figure 21: Total running time cost of ten oscillation cycles for NACA0012 and NACA64A010 aerofoils. The running time of the CPU refers to a single core.

5. Conclusions

The original single-core CPU based meshless dynamic cloud method is successfully ported to many-core programmable CUDA supported GPUs. C/C++ structures compatible with Fortran derived types are utilised to enclose data for meshless clouds of points, which are stored in the global memory of GPUs. Numerical simulation of steady compressible flows is firstly conducted to verify the underlying method. It is further extended to compute unsteady compressible flows over oscillatory aerofoils. The results are validated through detailed comparison with experiments and other reference solutions. Systematic analysis reveals that the meshless dynamic method is successfully accelerated by more than an order of magnitude, and it takes the GPU less energy to complete the same task compared to the CPU. Our next step's work will focus on fluid-structure interaction problems such as aerofoil/wing flutter prediction. We will also try to solve multi-objective optimisation problems for various real-word design problems by the present method coupled with evolutionary algorithms.

References

[1] J. T. Batina, A gridless Euler/Navier-Stokes solution algorithm for complex-aircraft applications, in: 31st Aerospace Sciences Meeting & Exhibit, 1993, AIAA Paper 93-0333.

[2] R. Löhner, C. Sacco, E. Oñate, S. Idelsohn, A finite point method for compressible flow, International Journal for Numerical Methods in Engineering 53 (8) (2002) 1765 – 1779. doi:10.1002/nme.334.

[3] Z. Ma, H. Chen, C. Zhou, A study of point moving adaptivity in gridless method, Computer Methods in Applied Mechanics and Engineering 197 (21-24) (2008) 1926–1937. doi:10.1016/j.cma.2007.12.012.

[4] K. Morinishi, An implicit gridless type solver for the Navier-Stokes equations, Computational Fluid Dynamics Journal Special Issue (2001) 551–560.

[5] N. Munikrishna, N. Balakrishnan, Turbulent flow computations on a hybrid cartesian point distribution using meshless solver LSFD-U, Computers & Fluids 40 (2011) 118–138. doi:10.1016/j.compfluid.2010.08.017.

[6] D. Sridar, N. Balakrishnan, An upwind finite difference scheme for meshless solvers, Journal of Computational Physics 189 (1) (2003) 1–29. doi:10.1016/S0021-9991(03)00197-9.

[7] E. Ortega, E. Oñate, S. Idelsohn, A finite point method for adaptive three-dimensional compressible flow calculations, International Journal for Numerical Methods in Fluids 60 (9) (2009) 937 – 971. doi:10.1002/fld.1892.

[8] H. Wang, H.-Q. Chen, J. Periaux, A study of gridless method with dynamic clouds of points for solving unsteady CFD problems in aerodynamics, International Journal for Numerical Methods in Fluids 64 (1) (2010) 98–118. doi:10.1002/fld.2145.

[9] H. Wang, J. Leskinen, D.-S. Lee, J. Periaux, Active flow control of airfoil using mesh/meshless methods coupled to hierarchical genetic algorithms for drag reduction design, Engineering Computations 30 (4) (2013) 562 – 580. doi:10.1108/02644401311329370.

[10] E. Ortega, E. Oñate, S. Idelsohn, R. Flores, A meshless finite point method for three-dimensional analysis of compressible flow problems involving moving boundaries and adaptivity, International Journal for Numerical Methods in Fluids 73 (2013) 323–343. doi:10.1002/fld.3799.

[11] E. Ortega, E. Oate, S. Idelsohn, R. Flores, Comparative accuracy and performance assessment of the finite point method in compressible flow problems, Computers & Fluids 89 (0) (2014) 53 – 65. doi:http://dx.doi.org/10.1016/j.compfluid.2013.10.024.

[12] A. Karatarakis, P. Karakitsios, M. Papadrakakis, GPU accelerated computation of the isogeometric analysis stiffness matrix, Computer Methods in Applied Mechanics and Engineering 269 (2014) 334–355. doi:10.1016/j.cma.2013.11.008.

[13] M. Papadrakakis, G. Stavroulakis, A. Karatarakis, A new era in scientific computing: Domain decomposition methods in hybrid CPU–gpu architectures, Computer Methods in Applied Mechanics and Engineering 200 (13-16) (2011) 1490–1508. doi:10.1016/j.cma.2011.01.013.

[14] C. M. Bard, J. C. Dorelli, A simple GPU–accelerated two-dimensional muscl–hancock solver for ideal magnetohydrodynamics, Journal of Computational Physics 259 (2014) 444–460. doi:10.1016/j.jcp.2013.12.006.

[15] S. Liang, W. Liu, L. Yuan, Solving seven-equation model for compressible two-phase flow using multiple GPUs, Computers & Fluids(in press). doi:10.1016/j.compfluid.2014.04.021.

[16] A. Corrigan, F. F. Camelli, R. Lhner, J. Wallin, Running unstructured grid-based CFD solvers on modern graphics hardware, International Journal for Numerical Methods in Fluids 66 (2) (2011) 221–229. doi:10.1002/fld.2254.

[17] V. G. Asouti, X. S. Trompoukis, I. C. Kampolis, K. C. Giannakoglou, Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on Graphics Processing Units, International Journal for Numerical

Methods in Fluids 67 (2) (2011) 232–246. doi:10.1002/fld.2352.

[18] I. Kampolis, X. Trompoukis, V. Asouti, K. Giannakoglou, CFD-based analysis and two-level aerodynamic optimization on graphics processing units, Computer Methods in Applied Mechanics and Engineering 199 (2010) 712 – 722. doi:10.1016/j.cma.2009.11.001.

[19] Z. Ma, H. Wang, S. Pu, GPU computing of compressible flow problems by a meshless method with space-filling curves, Journal of Computational Physics 263 (2014) 113–135. doi:10.1016/j.jcp.2014.01.023.

[20] P. Roe, Approximate Riemann solvers, parameter vectors, and difference schemes, Journal of Computational Physics 43 (2) (1981) 357 – 372. doi:10.1016/0021-9991(81)90128-5.

[21] A. Jameson, Time dependent calculations using multigrid, with applications to unsteady flows past airfoils and wings, in: AIAA 10th Computational Fluid Dyanmics Conference, 1991, AIAA 91-1956.

[22] X. Liu, N. Qin, H. Xia, Fast dynamic grid deformation based on delaunay graph mapping, Journal of Computational Physics 211 (2) (2006) 405–423. doi:10.1016/j.jcp.2005.05.025.

[23] NVIDIA, CUDA C programming guide (2012).
URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[24] J. Balfour, CUDA threads and atomics (2011).
URL http://mc.stanford.edu/cgi-bin/images/3/34/Darve_cme343_cuda_3.pdf

[25] T. Pulliam, J. Steger, Recent improvements in efficiency, accuracy, and convergence for implicit approximate factorization algorithms, in: AlAA 23rd Aerospace Sciences Meeting, Vol. 85, 1985, p. 0360.

[26] A. Jameson, W. Schmidt, E. Turkel, et al., Numerical solutions of the Euler equations by finite volume methods using Runge-Kutta time-stepping schemes, AIAA paper 81 (1981) 1259.

[27] R. Landon, NACA0012 oscillatory and transient pitching, Tech. rep., AGARD Report 702 (1982).

[28] J. T. Batina, Unsteady euler airfoil solutions using unstructured dynamic meshes, AIAA Journal 28 (1990) 1381–1388.

[29] D. Kirshman, F. Liu, Flutter prediction by an euler method on non-moving cartesian grids with gridless boundary conditions, Computers & Fluids 35 (6) (2006) 571 – 586. doi:10.1016/j.compfluid.2005.04.004.

[30] S. Davis, NACA 64A010 (NASA AMES Model) oscillatory pitiching, Tech. rep., AGARD-R-702 (1982).

[31] J. M. Hsu, A. Jameson, An implicit-explicit hybrid scheme for calculating complex unsteady flows, in: 40th AIAA Aerospace Sciences Meeting and exhibit, 2002.

[32] J. M.-J. Hsu, An implicit-explicit flow solver for complex unsteady flows, Ph.D. thesis, STANFORD UNIVERSITY (2004).

[33] F. Liu, S. Ji, Unsteady flow calculations with a multigrid Navier-Stokes method, AIAA JOURNAL 34 (1996) 2047–2053.

[34] G. WANG, Y. dan SUN, Z. yin YE, Gridless solution method for two-dimensional unsteady flow, Chinese Journal of Aeronautics 18 (1) (2005) 8 – 14. doi:http://dx.doi.org/10.1016/S1000-9361(11)60275-6.