A GPU-Accelerated Discontinuous Galerkin Method for Solving Two-Dimensional Laminar Flows

GAO Huanqin[1], *CHEN Hongquan*[1*], *ZHANG Jiale*[1], *XU Shengguan*[1], *GAO Yukun*[2]

1. Key Laboratory of Non-steady Aerodynamics and Flow Control of MIIT, College of Aerospace Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, P. R. China;
2. School of Mechanical Engineering, Anhui University of Technology, Maanshan 243002, P. R. China

(Received 11 January 2021; revised 6 March 2021; accepted 24 August 2021)

Abstract: A graphics processing unit (GPU)-accelerated discontinuous Galerkin (DG) method is presented for solving two-dimensional laminar flows. The DG method is ported from central processing unit to GPU in a way of achieving GPU speedup through programming under the compute unified device architecture (CUDA) model. The CUDA kernel subroutines are designed to meet with the requirement of high order computing of DG method. The corresponding data structures are constructed in component-wised manners and the thread hierarchy is manipulated in cell-wised or edge-wised manners associated with related integrals involved in solving laminar Navier-Stokes equations, in which the inviscid and viscous flux terms are computed by the local lax-Friedrichs scheme and the second scheme of Bassi & Rebay, respectively. A strong stability preserving Runge-Kutta scheme is then used for time marching of numerical solutions. The resulting GPU-accelerated DG method is first validated by the traditional Couette flow problems with different mesh sizes associated with different orders of approximation, which shows that the orders of convergence, as expected, can be achieved. The numerical simulations of the typical flows over a circular cylinder or a NACA 0012 airfoil are then carried out, and the results are further compared with the analytical solutions or available experimental and numerical values reported in the literature, as well as with a performance analysis of the developed code in terms of GPU speedups. This shows that the costs of computing time of the presented test cases are significantly reduced without losing accuracy, while impressive speedups up to 69.7 times are achieved by the present method in comparison to its CPU counterpart.

Key words: discontinuous Galerkin; GPU; compute unified device architecture (CUDA); Navier-Stokes equation; laminar flows

CLC number: V211.3 **Document code:** A **Article ID:** 1005-1120(2022)04-0450-17

0 Introduction

Discontinuous Galerkin (DG) method, proposed in the early 1970s[1], has become a popular high order method in recent years due to attractive features including stability, conservation and convergence etc[2]. Various achievements can be noted in many research fields like computational fluid dynamics (CFD)[2-4], computational acoustics[5] and computational magneto-hydrodynamics[6]. It is reported that the computational consuming time grows rapidly with the order of approximation of the high order DG method in comparison to low order methods, such as traditional the finite volume method (FVM) and the finite element method (FEM), which restrains the further application of the DG method in engineering[7]. Therefore, many research activities have focused on improving the efficiency of the DG method through modification and parallelization. In view of modification, some influential schemes, like the implicit time marching scheme[3], the multigrid scheme[4] etc., were successfully implemented

*Corresponding author, E-mail address: hqchenam@nuaa.edu.cn.

How to cite this article: GAO Huanqin, CHEN Hongquan, ZHANG Jiale, et al. A GPU-accelerated discontinuous Galerkin method for solving two-dimensional laminar flows[J]. Transactions of Nanjing University of Aeronautics and Astronautics, 2022, 39(4): 450-466.

http://dx.doi.org/10.16356/j.1005-1120.2022.04.007

in the DG method. Speedups of the DG method through parallelization can also be noted in many literatures. Generally, most speedups are achieved on the central processing unit (CPU) with multiple processors[2,7-8], few on the modern graphics processing unit (GPU) architecture[9-11], and hence GPU acceleration of the DG method is worthwhile to investigate in view of modern computers equipped mostly with multi-GPUs.

Modern GPUs have hundreds of processing cores for specialized graphics rending in parallel, which leads to orders of magnitude higher in memory bandwidth and faster in float-point operations in comparison to those of traditional CPUs. As illustrated in Fig.1, the peak values of NVIDIA GPU performance are tens of times faster than those of the contemporaneous Intel CPU, and the gap between the NVIDIA GPU and the Intel CPU in peak performance, measured in floating point operations per second (FLOP/s), has increased over the last ten years. The fact that hardware of GPU outperforms CPU indicates the methods with GPU implementation have the potentiality to achieve high efficiency.

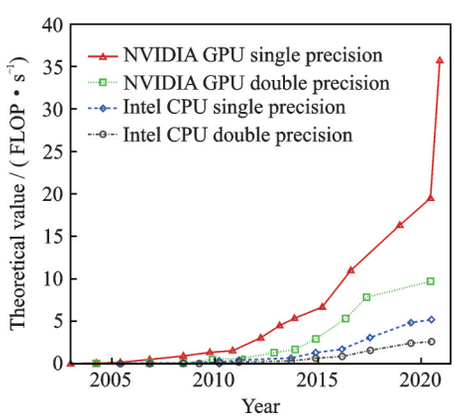


Fig.1 Floating point operations per second for CPU and GPU

However, in early years, parallel computing on GPU was a complicated exercise and mainly depended on low-level graphics programming. Recently, researchers have been allowed to use high-level programming languages with the development of unified programming models, including OpenCL, OpenACC and the compute unified device architecture (CUDA). Among them, CUDA has been recognized as the most popular programming model by many research communities. In CFD community, many low order CFD methods like the finite difference method (FDM)[12], FVM[13], FEM[14] and the meshless method[15] have been ported from CPU to GPU under the CUDA model to achieve speedups of dozens of times.

Research activities can also be noted for GPU implementations of the high order DG method. For instance, Klöeckner et al.[16] developed a 3D unstructured linear nodal DG code to solve the Maxwell equations. Siebenborn et al.[9-10] developed an explicit modal DG code to solve 3D Euler equations. It was later modified by Karakus et al.[11] for solving incompressible laminar flows. It can be noted that the developed DG codes are in two types, nodal and modal. In general, a nodal DG scheme is often adopted together with an integral-free approach, which perfectly fits into the GPU programming model. However, the integral-free approach is not sufficient for nonlinear partial differential equations (PDEs). Therefore, one has to resort to the modal type of the DG scheme for solving nonlinear PDEs like problems to be solved in this paper. However, the computations of the modal DG solver are usually complicated in comparison to the nodal counterpart due to extra numerical integrations at each time step, which often results in the waste of partial GPU threads. Therefore, in order to achieve GPU speedups efficiently, the thread hierarchy with related data structure to be appropriately designed for different DG schemes with different approximate orders is still one of the major research problems.

In this paper, efforts are made to develop a modal type of GPU-accelerated DG method for solving two-dimensional laminar flows. Following the work of Fuhry et al.[17], which is developed for solving two-dimensional Euler equations, the programming models of thread-per-cell and thread-per-edge are applied for solving two-dimensional Navier-Stokes equations, in which the inviscid and viscous flux terms are computed by the local lax-Friedrichs (LLF) scheme[18] and the second scheme of Bassi & Rebay (BR2)[3,19], respectively. A strong stability preserving (SSP) Runge-Kutta scheme[20] is then used for time marching of numerical solutions. Thus, the thread hierarchy with related data struc-

ture can be conveniently managed, which gives the great flexibility for accommodating different DG schemes with different approximate orders. The most important is that the DG scheme can be implemented in a unified manner without varying with change of approximation orders. The orders of convergence of the resulting GPU-accelerated DG method are first validated by the traditional Couette flow problems with different mesh sizes associated with different orders of approximation. The numerical simulations of the typical flows over a circular cylinder or a NACA 0012 airfoil are then carried out, and the obtained results are compared with analytical solutions or available experimental and numerical values reported in the literature, as well as with a performance analysis of the developed code in terms of GPU speedups.

1 Numerical Method

Before implementing the DG method on GPU, we provide a brief description of the traditional DG method[3,19] for laminar flow simulations.

1.1 Governing equations

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

$$\frac{\partial \boldsymbol{U}}{\partial t}+\nabla\cdot\left(\boldsymbol{F}_i(\boldsymbol{U})-\boldsymbol{F}_v(\boldsymbol{U},\nabla\boldsymbol{U})\right)=0 \tag{1}$$

where $\boldsymbol{U}=(\rho,\rho u,\rho v,\rho E)^{\mathrm{T}}$ is the vector of conserved variables; and $\boldsymbol{F}_i(\boldsymbol{U}),\boldsymbol{F}_v(\boldsymbol{U},\nabla\boldsymbol{U})$ are the inviscid and viscous flux terms, respectively, with

$$\begin{cases}\boldsymbol{F}_i(\boldsymbol{U})=\begin{pmatrix}\rho u & \rho v\\ \rho u^2+p & \rho vu\\ \rho uv & \rho v^2+p\\ \rho u\left(E+\dfrac{p}{\rho}\right) & \rho v\left(E+\dfrac{p}{\rho}\right)\end{pmatrix}\\ \boldsymbol{F}_v(\boldsymbol{U},\nabla\boldsymbol{U})=\begin{pmatrix}0 & 0\\ \tau_{xx} & \tau_{yx}\\ \tau_{xy} & \tau_{yy}\\ u\tau_{xx}+v\tau_{xy}-q_x & u\tau_{yx}+v\tau_{yy}-q_y\end{pmatrix}\end{cases} \tag{2}$$

where ρ is the fluid density; u, v are the velocity components along x and y axes, respectively; E is the total energy per unit mass and p the pressure. Assume that the fluid is perfect gas, and the equation $E=\dfrac{1}{\gamma-1}\dfrac{p}{\rho}+\dfrac{1}{2}(u^2+v^2+w^2)$ is satisfied, where γ indicates the ratio of specific heat coefficient, for air, $\gamma=1.4$. In the viscous flux term, the components of shear stress and heat conduction are defined as

$$\begin{cases}\tau_{ij}=\mu\left(\dfrac{\partial u_i}{\partial x_j}+\dfrac{\partial u_j}{\partial x_i}\right)-\dfrac{2}{3}\mu\dfrac{\partial u_k}{\partial x_k}\delta_{ij}\\ q_j=-\dfrac{\mu C_{\mathrm{p}}}{Pr}\dfrac{\partial T}{\partial x_j}\end{cases} \tag{3}$$

where the subscripts i and j indicate x and y directions, respectively; Pr is the Prandtl number and taken as 0.72 for laminar flows; C_{p} the specific heat capacity at a constant pressure. The viscosity coefficient μ is calculated by the Sutherland formulation

$$\frac{\mu}{\mu_0}=\left(\frac{T}{T_0}\right)^{\frac{3}{2}}\frac{T_0+S}{T+S} \tag{4}$$

where $T_0=288.15\,\mathrm{K}$, $\mu_0=1.716\times10^{-5}\,\mathrm{kg/ms}$, $S=110.4\,\mathrm{K}$ for air.

1.2 Discontinuous Galerkin method

The DG method used in this paper is the well-known BR2 scheme[3,19]. To apply the BR2 scheme, Eq.(1) needs to be reformulated by replacing the gradient of the solution $\nabla\boldsymbol{U}$ with an additional independent unknown $\boldsymbol{\Theta}$. Thus, Eq.(1) can then be reshaped as

$$\boldsymbol{\Theta}-\nabla\boldsymbol{U}=0 \tag{5}$$

$$\frac{\partial \boldsymbol{U}}{\partial t}+\nabla\cdot\boldsymbol{F}_i(\boldsymbol{U})-\nabla\cdot\boldsymbol{F}_v(\boldsymbol{U},\boldsymbol{\Theta})=0 \tag{6}$$

Then the traditional DG approach is applied to Eqs.(5) and (6), resulting in Eq.(7) and (8) for cell Ω_e

$$\int_{\Omega_e}v_k\boldsymbol{\Theta}_h\mathrm{d}\Omega_e-\overbrace{\int_{\partial\Omega_e}v_k\boldsymbol{U}_h\cdot\boldsymbol{n}\,\mathrm{d}\sigma}^{\text{Edge integral ①}}+\int_{\Omega_e}\nabla v_k\cdot\boldsymbol{U}_h\mathrm{d}\Omega_e=0 \tag{7}$$

$$\int_{\Omega_e}v_k\frac{\partial \boldsymbol{U}_h}{\partial t}\mathrm{d}\Omega+\underbrace{\overbrace{\int_{\partial\Omega_e}v_k\boldsymbol{F}_i(\boldsymbol{U}_h)\cdot\boldsymbol{n}\mathrm{d}\sigma}^{\text{Edge integral ②}}-\int_{\Omega_e}\nabla v_k\cdot\boldsymbol{F}_i(\boldsymbol{U}_h)\mathrm{d}\Omega}_{\text{Convective part}}-\underbrace{\overbrace{\int_{\partial\Omega_e}v_k\boldsymbol{F}_v(\boldsymbol{U}_h,\boldsymbol{\Theta})\cdot\boldsymbol{n}\mathrm{d}\sigma}^{\text{Edge integral ③}}+\int_{\Omega_e}\nabla v_k\cdot\boldsymbol{F}_v(\boldsymbol{U}_h,\boldsymbol{\Theta})\mathrm{d}\Omega}_{\text{Diffusion part}}=0 \tag{8}$$

where $\boldsymbol{U}_h$ is approximated to the numerical solution $\boldsymbol{U}_k$; p the approximate order. $\boldsymbol{U}_h$ can be written as

$$\boldsymbol{U}_h=\sum_{k=1}^{N_p}\boldsymbol{c}_k(t)v_k(\boldsymbol{x}) \tag{9}$$

where $v_k(\boldsymbol{x})$, the so-called basis or test functions, is a base of the polynomial space P^k and $N_p=(p+1)(p+2)/2$ the total number of these functions. The Tayler basis functions[20] are chosen in present article for their simplicity.

As there is no global continuity requirement for $\boldsymbol{U}_h$ in the DG method, numerical fluxes $\boldsymbol{H}_{\text{aux}}, \boldsymbol{H}_i, \boldsymbol{H}_v$ in edge integrals ①, ② and ③ need to be defined to handle this discontinuity

$$\int_{\partial\Omega_e}v_k\boldsymbol{U}_h\cdot\boldsymbol{n}\mathrm{d}\sigma=\int_{\partial\Omega_e}v_k\boldsymbol{H}_{\text{aux}}(\boldsymbol{U}^+,\boldsymbol{U}^-)\cdot\boldsymbol{n}\mathrm{d}\sigma \tag{10}$$

$$\int_{\partial\Omega_e}v_k\boldsymbol{F}_i(\boldsymbol{U}_h)\cdot\boldsymbol{n}\mathrm{d}\sigma=\int_{\partial\Omega_e}v_k\boldsymbol{H}_i(\boldsymbol{U}^+,\boldsymbol{U}^-)\cdot\boldsymbol{n}\mathrm{d}\sigma \tag{11}$$

$$\int_{\partial\Omega_e}v_k\boldsymbol{F}_v(\boldsymbol{U}_h)\cdot\boldsymbol{n}\mathrm{d}\sigma=\int_{\partial\Omega_e}v_k\boldsymbol{H}_v(\boldsymbol{U}^+,\boldsymbol{\Theta}^+,\boldsymbol{U}^-,\boldsymbol{\Theta}^-)\cdot\boldsymbol{n}\mathrm{d}\sigma \tag{12}$$

where the $(\cdot)^+$ and $(\cdot)^-$ are notated to indicate the traces from the exterior and the interior of the cell, respectively. The inviscid numerical flux $\boldsymbol{H}_i$ is taken as the widely used LLF[18] scheme in this paper for its stability and simplicity. The auxiliary and viscous numerical fluxes, $\boldsymbol{H}_{\text{aux}}$ and $\boldsymbol{H}_v$, proposed by Bassi and Rebay in their BR2 scheme are written as

$$\boldsymbol{H}_{\text{aux}}=\frac{1}{2}(\boldsymbol{U}^++\boldsymbol{U}^-) \tag{13}$$

$$\boldsymbol{H}_v=\frac{1}{2}(\boldsymbol{F}(\boldsymbol{U}^+,\boldsymbol{\Theta}^+)+\boldsymbol{F}(\boldsymbol{U}^-,\boldsymbol{\Theta}^-)) \tag{14}$$

After inserting the centered flux H_{aux} and integrating by parts for integrals ①, Eq.(7) can then be transformed as

$$\int_{\Omega_e}v_k\boldsymbol{\Theta}_h\mathrm{d}\Omega_e=\int_{\Omega_e}v_k\nabla\boldsymbol{U}_h\mathrm{d}\Omega_e+\int_{\partial\Omega_e}v_k(\boldsymbol{H}_{\text{aux}}-\boldsymbol{U}^-)\cdot\boldsymbol{n}\mathrm{d}\sigma \tag{15}$$

It can be noted that the auxiliary variable $\boldsymbol{\Theta}$ is a sum of the solution gradient $\nabla\boldsymbol{U}$ and a global lifting operator $\boldsymbol{R}$, $\boldsymbol{\Theta}=\nabla\boldsymbol{U}+\boldsymbol{R}$, where $\boldsymbol{R}=\sum_{e\in\partial\Omega_e}\boldsymbol{r}^e$ and $\boldsymbol{r}^e$ are local lifting operators only related to the edge e

$$\int_e v_k\boldsymbol{r}^e\mathrm{d}\Omega_e=\int_e v_k(\boldsymbol{H}_{aux}-\boldsymbol{U}^-)\cdot\boldsymbol{n}\mathrm{d}\sigma \tag{16}$$

A stable factor η is also introduced in the local lifting operator and usually taken as the number of edges in each cell. Thus, Eq.(8) can then be written as

$$\begin{aligned}&\int_{\Omega_e}v_k\frac{\partial\boldsymbol{U}_h}{\partial t}\mathrm{d}\Omega_e+\int_{\partial\Omega_e}v_k\boldsymbol{H}_i(\boldsymbol{U}^+,\boldsymbol{U}^-)\cdot\boldsymbol{n}\mathrm{d}\sigma-\\&\int_{\Omega_e}\nabla v_k\cdot\boldsymbol{F}_i(\boldsymbol{U}_h)\mathrm{d}\Omega-\sum_{e\in\partial\Omega_e}\int_e v_k\boldsymbol{H}_v(\boldsymbol{U}^+,\nabla\boldsymbol{U}^++\\&\eta\boldsymbol{r}_e^+,\boldsymbol{U}^-,\nabla\boldsymbol{U}^-+\eta\boldsymbol{r}_e^-)\cdot\boldsymbol{n}\mathrm{d}\sigma+\\&\int_{\Omega_e}\nabla v_k\cdot\boldsymbol{F}_v(\boldsymbol{U}_h,\nabla\boldsymbol{U}_h+R)\mathrm{d}\Omega=0\end{aligned} \tag{17}$$

After several mathematical conversions, the final formulation of the semi-discrete of the DG method can be written as

$$\begin{aligned}\frac{\partial\boldsymbol{U}_h}{\partial t}=\boldsymbol{M}^{-1}\Bigg(&-\int_{\partial\Omega_e}v_k\boldsymbol{H}_i(\boldsymbol{U}^+,\boldsymbol{U}^-)\cdot\boldsymbol{n}\mathrm{d}\sigma+\\&\int_{\Omega_e}\nabla v_k\cdot\boldsymbol{F}_i(\boldsymbol{U}_h)\mathrm{d}\Omega+\sum_{e\in\partial\Omega_e}\int_e v_k\boldsymbol{H}_v(\boldsymbol{U}^+,\nabla\boldsymbol{U}^++\\&\eta\boldsymbol{r}_e^+,\boldsymbol{U}^-,\nabla\boldsymbol{U}^-+\eta\boldsymbol{r}_e^-\Bigg)\cdot\boldsymbol{n}\mathrm{d}\sigma-\\&\int_{\Omega_e}\nabla v_k\cdot\boldsymbol{F}_v(\boldsymbol{U}_h,\nabla\boldsymbol{U}_h+R)\mathrm{d}\Omega)\end{aligned} \tag{18}$$

where $\boldsymbol{M}$ is the mass matrix with elements $M_{i,j}=\int_{\Omega_e}v_iv_j\mathrm{d}\Omega_e$.

To simplify the calculations, all integrals are solved using the Gaussian quadrature rules and computed in canonical cells[17] by establishing map functions between arbitrary cells in the mesh and the canonical triangle $\Gamma=\{-1\leqslant r,s\leqslant 1, r+s\leqslant -1\}$ or canonical quadrangle $\Gamma=\{-1\leqslant r\leqslant 1,\ -1\leqslant s\leqslant 1\}$ on which the Gaussian quadrature rules are known. These map functions Ψ_e: $\Omega_e\rightarrow\Gamma$ are created to connect the physical coordinates $\boldsymbol{x}=(x,y)$ and the ones in the canonical cell $\boldsymbol{r}=(r,s)$ for each cell Ω_e. Thus, integrations for any functions in the physical space Ω_e are evaluated as

$$\int_{\Omega_e}f g\mathrm{d}\Omega=\int_{\Gamma=\Psi_e(\Omega_e)}f(\Psi_e)g(\Psi_e)\ |\det(D\Psi_e)|\mathrm{d}\Gamma \tag{19}$$

where $D\Psi_e=\begin{pmatrix}r_x & r_y\\ s_x & s_y\end{pmatrix}$; $J=|D\Psi_e|$ is a Jacobi number and stays positive when edges of the cell are counterclockwise. This method is suitable for cells with straight edges, but for cells with curved edges, special treatment is usually needed. A strategy for curved cells appeared in the work of Lübon et al.[21] is adopted for treating curved triangle and quadrangle cells involved in the present work.

The time integration of the semi discrete system in Eq.(18) is accomplished by the SSP Runge-Kutta scheme[22], which is stable for a Courant num-

ber less or equal to $1/(2p+1)$. In the present work, this scheme is adopted and Courant number is fixed to be $1/(2p+1)$ for all the test cases.

2 GPU Implementation of DG Method

Among the available unified programming models, CUDA is recognized as the most popular one since it provides researchers with a high-level programming language for GPU computing. In the present work, the CUDA C programming model is used in GPU implementation of the DG method. Some techniques involved in GPU implementation, including CUDA subroutine, data structure and thread hierarchy, will be discussed in this section after a brief introduction of the CUDA C programming model.

2.1 CUDA C programming model

In the present work, a NVIDIA GTX TITAN GPU together with the CUDA C programming model[22] is employed for developing the GPU codes. The GTX TITAN GPU is based on the Kepler architecture, which contains a total of 14 streaming multiprocessors (SM) and each SM has 192 CUDA cores. In the CUDA programming model, when a parallel task (usually a kernel function) is executed on GPU, a double-layer-based thread hierarchy is created: All threads are organized into a set of thread blocks, and all of these thread blocks are then gathered into a thread grid, as shown in Fig.2. For a specified grid, each block has the same number of threads, which will be sent to different cores in the same SM and be executed in parallel.

The efficiency of GPU parallelization is related to the usage of available memories existed on the GPU architecture. The global memory, the register and constant memory are required to be used in a way of careful management due to different access speeds. The slow-access global memory located outside of the chip has the capacity to store major data and may introduce large latencies. Therefore, the management of global memory is realized by minimizing access times to the memory. In specific,

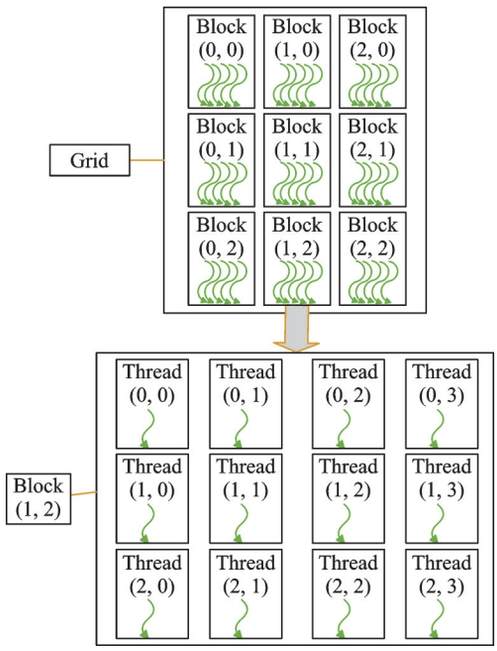


Fig.2 Description of thread hierarchy

global transactions are coalesced by instructing the nearby threads to access the same blocks of memory. Each thread has its own private registers located in the fast-access GPU chips as shown in Fig.3. Although registers available in each thread are limited, their capacities are used as much as possible thanks to their fast-access features. As usual, the low capacity constant memory, which is reported to be almost as fast as registers when all threads in a warp access the same location[23], is used to store a small number of constant data accessed frequently.

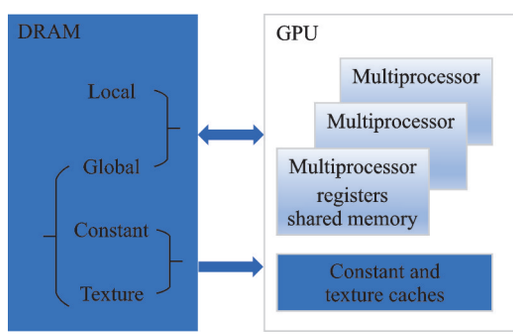


Fig.3 Types of memories and their locations

2.2 CUDA C-based GPU implementation

As mentioned above, GPU is ideally suited for computations that can be run on numerous data simultaneously in parallel, and performs poorly when dealing with logical judgements and branch struc-

tures. Therefore, the computing tasks to be executed on GPU should be carefully selected. For the DG method described in Section 1.2, the computing tasks associated with the Runge-Kutta time-marching procedure are the most time-consuming part, and hence are ported to GPU with the use of CUDA C; while the other parts related to pre- or post-processing are still kept on CPU. As shown in Fig. 4, according to the Runge-Kutta scheme and the specific DG discretization in space, the time-marching procedure of the DG method has been split into a set of sub-procedures, which could be calculated by kernel subroutines in GPU implementation. In specific, the subroutine, namely DT, is assigned to update the value of time step; the subroutine ConVars is designed to compute the conserved variables appeared in Eq. (9); the subroutine Grad is developed to compute the gradients of conservative variables based on the BR2 scheme; the subroutine Flux is assigned to compute the numerical fluxes, including the LLF flux and viscous numerical flux appeared in Eq. (14); the subroutine RHS is designed to compute the righthand side part in Eq. (18); the subroutine UP is programmed to update the solution based on the Runge-Kutta scheme; and the subroutine Res is used to evaluate the computational residuals.

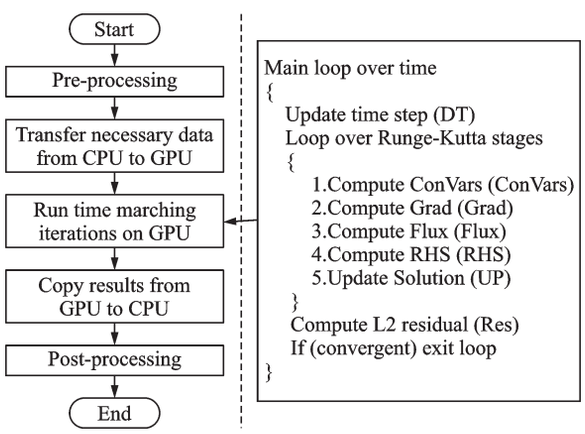


Fig.4 A general procedure of GPU-based DG solver

As discussed in Section 2.1, registers are limited in each thread block, thus, the subroutines mentioned are further split to low level kernels. Corresponding cell-based and edge-based kernels are then designed to compute the quantities related to cells and edges. For instance, the subroutine RHS consists of three kernels, Vol_Rhs_Kernel, Surf_Rhs_Kernel and MulMtx_Rhs_Kernel. Specifically, Vol_Rhs_Kernel is cell-based and used to compute the cell integrals appeared in Eq. (18); Surf_Rhs_Kernel is edge-based and developed to evaluate edge integrals in Eq. (18). The cell and edge integrals are then combined and multiplied by the inverse of mass matrix $\boldsymbol{M}$ in a cell-based kernel MulMtx_Rhs_Kernel. Listing 1 gives the corresponding code snippet of subroutine RHS.

$$\boldsymbol{C}[i,j,m]=\left|C^1_{1,1}\right|C^1_{2,1}\left|\cdots\right|C^1_{N,1}\left|C^1_{1,2}\right|C^1_{2,2}\left|\cdots\right|C^1_{N,2}\left|\cdots\right|C^1_{N,N}\left|C^2_{1,1}\right|\cdots\left|C^2_{N,N}\right|\cdots\left|C^M_{N,N}\right|$$

Fig.5 Resulting storage pattern of array $\boldsymbol{C}$

Listing 1 Code snippet of subroutine RHS

```
(1) void RHS (…)
(2) {…
(3)     Vol_Rhs_Kernel <<<>>> (…);
(4)     Surf_Rhs_Kernel <<<>>> (…);
(5)     MulMtx_Rhs_Kernel <<<>>>
        (…);
(6) …
(7) }
```

The kernels are also developed in remaining subroutines like ConVars mentioned above. In order to make it clear, the main subroutine of the Runge-Kutta time-marching procedure is given in Listing 2.

Listing 2 Code snippet of subroutine SSP_RKDG

```
(1) void SSP_RKDG (…)
(2) {
(3)     ! evaluate the time step
(4)     DT(…);
(5)
(6)     ! loop over SSP Runge-Kutta iter-
        ation
(7)     while (j<nRK)
(8)     {
(9)         ConVars(…);
(10)        Grad(…);
(11)        Flux(…);
(12)        RHS(…);
(13)        UP(…);
(14)    }
```

(15)
(16) Res(…);
(17) if (isConvergent) break;
(18) }

It can be noted that the whole procedure of the program mainly depends on the clear modules of these computational subroutines, which results in great flexibility for possible extensions associated with schemes updating in view of modules replacement.

2.2.1 Data structure and memory management

As mentioned above, the coalesced memory access is helpful for reducing access times to the global memory. In order to maximize coalesced memory access, data structures are designed in component-wise manners. Thus, data related to the same component can be stored continuously. For example, the coefficients with the same order, which are required to be fetched simultaneously, can now be placed side by side in the solution coefficient array $\boldsymbol{C}$ (Eq.(9)), as shown in Fig.5. The element $C_{i,j}^{m}$ of array $\boldsymbol{C}$ denotes the coefficient of cell i, corresponding to the basis function v_j of the mth conservative variables. Therefore, the nearby threads can access nearby addresses, as illustrated in Fig.6.

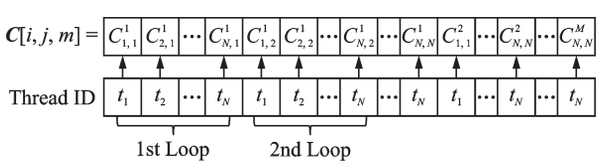


Fig.6 Thread access pattern of array $\boldsymbol{C}$

Besides, constant data that are used frequently are stored in the constant memory to improve memory throughput. In specific, the Tayler basis functions v_k and their gradients ∇v_k together with the coefficients corresponding to the Gaussian quadrature rules in canonical cells are such kind constant data in the present GPU implementation, and they are all stored in constant memory. With the data structure developed and memory management mentioned, the related quantities like numerical fluxes and residuals are computed by specific GPU kernels, as listed in Listing 1, which will be addressed in next section.

2.2.2 Key kernels

In order to have a deep understanding of kernels constructed in the Section 2.2, three representative key kernels are described in details. Locations of related variables in different memories are listed in Table 1.

Table 1 Notations of variables

Notation	Description
${}^{g}A$	Superscript g: Variables located in global memory
${}^{t}A$	Superscript t: Variables located in thread private memory
${}^{c}A$	Superscript c: Variables located in constant memory

The first kernel, namely Flux_Kernel, is edge-based and used to calculate the numerical fluxes at the edge integral points. As shown in Algorithm 1, the thread index, id, is calculated in line 2, the upper bound is checked in line 3 to avoid possible mistakes caused by computations beyond the bound. After that, the loop is carried out over each integral point into four steps. In Step 1, necessary data in the global memory are fetched and assigned to the thread private registers. The boundary conditions are enforced in Step 2. Notice that branch structures are existed with different boundary treatments. In order to reduce low-efficient branches among neighboring threads from the same thread warp (strictly executing the same instructions synchronously, as the technique details in Ref.[23] describes), a simple sorting of edges is performed during the pre-processing to gather the same type of boundary edges together, which reduces the amount of low-efficient branches to no more than N_{type}. N_{type} is the number of boundary types in the computational domain. In Step 3, the inviscid and viscous numerical fluxes are then calculated by the specific schemes. Finally, the obtained values are sent to the global memory in Step 4.

Algorithm 1 __global__ void FluxKernel()
Input Global memory: Conserved variables at the edge integral points W_f;
Gradients of conserved variables at the edge integral points $\mathrm{d}x W_f$, $\mathrm{d}y W_f$;

Connectivity matrices $\boldsymbol{L}, \boldsymbol{R}, \boldsymbol{L}_{\text{point}}, \boldsymbol{R}_{\text{point}}$;
Norm vectors $\boldsymbol{n}_x, \boldsymbol{n}_y$.

Output Flux at the edge integral points.

(1)
(2)id=threadIdx.x+blockIdx.x×blockDim.x
(3)if id<NL then
(4) cl=$\boldsymbol{L}$[id]
(5) cr=R[id]
(6) for each edge integral point i do
(7)
(8) Step 1 Load data from global memory
(9) ${}^{\text{t}}\boldsymbol{n}_x \leftarrow {}^{\text{g}}\boldsymbol{n}_x, {}^{\text{t}}\boldsymbol{n}_y \leftarrow {}^{\text{g}}\boldsymbol{n}_y$
(10) ${}^{\text{t}}W_L \leftarrow {}^{\text{g}}W_f$, ${}^{\text{t}}\text{d}x W_L \leftarrow {}^{\text{g}}\text{d}x W_f$, ${}^{\text{t}}\text{d}y W_L \leftarrow {}^{\text{g}}\text{d}y W_f$
(11) if cr>0 then
(12) ${}^{\text{t}}W_R \leftarrow {}^{\text{g}}W_f$, ${}^{\text{t}}\text{d}x W_R \leftarrow {}^{\text{g}}\text{d}x W_f$, ${}^{\text{t}}\text{d}y W_R \leftarrow {}^{\text{g}}\text{d}y W_f$
(13) else
(14) ${}^{\text{t}}W_R \leftarrow {}^{\text{t}}W_L$
(15)
(16) Step 2 Enforce boundary condition if cr<0
(17) if cr==WALL then
(18) WallBoundary(${}^{\text{t}}W_R$)
(19) else if cr==FAR then
(20) FarBoundary(${}^{\text{t}}W_R$)
(21) else if cr==SYM then
(22) SymBoundary(${}^{\text{t}}W_R$)
(23)
(24) Step 3 Calculate flux at ecah intergral point
(25) if cr>0 then
(26) ${}^{\text{t}}$flux=LLFflux(${}^{\text{t}}W_L$, ${}^{\text{t}}W_R$, ${}^{\text{t}}\boldsymbol{n}_x$, ${}^{\text{t}}\boldsymbol{n}_y$) ${}^{\text{t}}$flux+=NumericalVisflux(${}^{\text{t}}W_L$, ${}^{\text{t}}W_R$, ${}^{\text{t}}\text{d}x W_L$ ${}^{\text{t}}\text{d}x W_R$, ${}^{\text{t}}\text{d}y W_L$, ${}^{\text{t}}\text{d}y W_R$, ${}^{\text{t}}\boldsymbol{n}_x$, ${}^{\text{t}}\boldsymbol{n}_y$)
(27) else
(28) ${}^{\text{t}}$flux=LLFflux(${}^{\text{t}}W_L$, ${}^{\text{t}}W_R$, ${}^{\text{t}}\boldsymbol{n}_x$, ${}^{\text{t}}\boldsymbol{n}_y$) ${}^{\text{t}}$flux+=NumericalVisflux(${}^{\text{t}}W_L$, ${}^{\text{t}}W_R$, ${}^{\text{t}}\text{d}x W_L$, ${}^{\text{t}}\text{d}x W_L$, ${}^{\text{t}}\text{d}y W_L$, ${}^{\text{t}}\text{d}y W_L$, ${}^{\text{t}}\boldsymbol{n}_x$, ${}^{\text{t}}\boldsymbol{n}_y$)
(29)
(30) Step 4 Send data to global memory
(31) ${}^{\text{g}}$flux$\leftarrow {}^{\text{t}}$flux

The second kernel, namely Vol_Rhs_Kernel, is cell-based and used to calculate the cell integral contributions to the right-hand side of Eq.(18). As shown in Algorithm 2, the calculations are accomplished in three steps. In Step 1, a private array ${}^{\text{t}}\text{RHS}_v$ is assigned in each thread. Thus, access times to the global memory can be benefited from this private array. It means that the access times mentioned can be greatly reduced, and hence it is possible for achieving high speedups. Then in Step 2, the calculation loop of the array is performed over each integral point in two sub-steps. In specific, for each integral point within the loop mentioned, the related data, which are previously in the global memory, are transferred to the registers of the thread as written in Step 2.1. With the available data of the point, the calculations associated with the array are then carried out in Step 2.2. After that, the obtained values of array are finally stored in the global memory in Step 3.

Algorithm 2 __global__ void Vol_Rhs_Kernel()

Input Global memory: Conserved variables at the cell integral points W_v;
Gradients of conserved variables at the cell integral points $\text{d}x W_v$, $\text{d}y W_v$; geometry factors at the cell integral points: r_x, r_y, s_x, s_y, J_a.
Constant memory: Basis functions v at the cell integral points and their gradients v_r, v_s; weighted coefficient w at the cell integral points

Output Cell integral contribution

(1)
(2)id=threadIdx.x+blockIdx.x×blockDim.x
(3)if id<NC then
(4) Step 1 Allocate array ${}^{\text{t}}\text{RHS}_v$
(5) ${}^{\text{t}}\text{RHS}_v[j,k] \leftarrow 0$
(6)
(7) Step 2 Calculate contribution to ${}^{\text{t}}\text{RHS}_v$
(8) for each volume integral point i do
(9) Step 2.1 Load data from global memory
(10) ${}^{\text{t}}W_v \xleftarrow{(k)} {}^{\text{g}}W_v$, ${}^{\text{t}}\text{d}x W_v \xleftarrow{(k)} {}^{\text{g}}\text{d}x W_v$, ${}^{\text{t}}\text{d}y W_v \xleftarrow{(k)} {}^{\text{g}}\text{d}y W_v$
(11) ${}^{\text{t}}r_x \leftarrow {}^{\text{g}}r_x$, ${}^{\text{t}}r_y \leftarrow {}^{\text{g}}r_y$, ${}^{\text{t}}s_x \leftarrow {}^{\text{g}}s_x$, ${}^{\text{t}}s_y \leftarrow {}^{\text{g}}s_y$, ${}^{\text{t}}J_a \leftarrow {}^{\text{g}}J_a$

(12)

(13) Step 2.2 Calculate contribution of the point to ${}^{t}\mathrm{RHS}_{v}$

(14) Flux(${}^{t}F$, ${}^{t}G$, ${}^{t}W_{v}$, ${}^{t}\mathrm{d}xW_{v}$, ${}^{t}\mathrm{d}yW_{v}$)

(15) ${}^{t}\mathrm{RHS}_{v}[j,k] \xleftarrow[(j,k)]{} ({}^{t}F[k]({}^{c}v_{r}[i,j]$ ${}^{t}r_{x}+{}^{c}v_{s}[i,j]{}^{t}s_{x})+$

(16) ${}^{t}G[k]({}^{c}v_{r}[i,j]{}^{t}r_{y}+{}^{c}v_{s}[i,j]{}^{t}s_{y}))$ ${}^{c}w[i]{}^{t}J_{a}$

(17)

(18) Step 3 Send data to global memory

(19) ${}^{g}\mathrm{RHS}[\mathrm{id},j,k] \xleftarrow[(j,k)]{} {}^{t}\mathrm{RHS}_{v}[j,k]$

The last kernel, Surf_Rhs_Kernel, which is edge-based, is designed to compute the edge integral contributions to the right-hand side of Eq.(18). As shown in Algorithm 3, two thread private arrays, ${}^{t}\mathrm{RHS}_{L}$ and ${}^{t}\mathrm{RHS}_{R}$, are allocated in Step 1, as shown in Algorithm 3. Then in Step 2 a loop is performed to compute the values of these arrays over each integral point, which is carried out in two substeps similar to the ones as the lines 9 to 16 in Algorithm 2. Finally, the obtained values of the arrays are sent to the global memory. Notice that race condition[23] may occur in Step 3. Therefore, an atomic operator[23] atomicAdd is used in this kernel to bypass this problem.

Algorithm 3 __global__ void Surf_Rhs_Kernel()

Input Global memory: Flux at the edge integral points; geometry factors: J_{s}; Connectivity matrices $\boldsymbol{L}, \boldsymbol{R}, \boldsymbol{L}_{\mathrm{point}}, \boldsymbol{R}_{\mathrm{point}}$. Constant memory: Basis functions v at the edge integral points; weighted coefficient w at the edge integral points

Output Edge integral contribution

(1)

(2)id=threadId.x.x+blockId.x.x×blockDim.x

(3)if id<NL then

(4) Step 1 Allocate array ${}^{t}\mathrm{RHS}_{L}$, ${}^{t}\mathrm{RHS}_{R}$

(5) ${}^{t}\mathrm{RHS}_{L}[j,k] \xleftarrow[(j,k)]{} 0$, ${}^{t}\mathrm{RHS}_{R}[j,k] \xleftarrow[(j,k)]{} 0$

(6) cl=$\boldsymbol{L}$[id], cr=$\boldsymbol{R}$[id]

(7)

(8) Step 2 Calculate contribution to ${}^{t}\mathrm{RHS}_{L}$, ${}^{t}\mathrm{RHS}_{R}$

(9) for each edge integral point i do

(10) Step 2.1 Load data from global memory

(11) idl $\leftarrow {}^{g}\boldsymbol{L}_{\mathrm{point}}$, idr $\leftarrow {}^{g}\boldsymbol{R}_{\mathrm{point}}$

(12) ${}^{t}\mathrm{flux}[j] \xleftarrow[(j)]{} {}^{g}\mathrm{flux}, J_{s} \leftarrow {}^{g}J_{s}$

(13)

(14) Step 2 Calculate contribution of the point to ${}^{t}\mathrm{RHS}_{L}$, ${}^{t}\mathrm{RHS}_{R}$

(15) ${}^{t}\mathrm{RHS}_{L}[j,k] \xleftarrow[(j,k)]{} {}^{t}\mathrm{flux}[k]$ ${}^{c}v[\mathrm{idl},j]{}^{t}J_{s}{}^{c}w[i]$

(16) if cr>0 then

(17) ${}^{t}\mathrm{RHS}_{R}[j,k] \xleftarrow[(j,k)]{} {}^{t}\mathrm{flux}[k]$ ${}^{c}v[\mathrm{idr},j]{}^{t}J_{s}{}^{c}w[i]$

(18)

(19) Step 3 Send data to global memory

(20) ${}^{g}\mathrm{RHS}[\mathrm{cl},j,k] \xleftarrow[(j,k)]{}$ atomicAdd(${}^{g}\mathrm{RHS}[\mathrm{cl},j,k]$, $-{}^{t}\mathrm{RHS}_{L}[j,k]$)

(21) if cr>0 then

(22) ${}^{g}\mathrm{RHS}[\mathrm{cr},j,k] \xleftarrow[(j,k)]{}$ atomicAdd(${}^{g}\mathrm{RHS}[\mathrm{cr},j,k]$, ${}^{t}\mathrm{RHS}_{R}[j,k]$)

3 Numerical Results and Performance Analyse

The CPU or GPU codes developed in the double-precision mode with methods described have been verified through simulating a set of typical flow problems, including Couette flows and flows over a circular cylinder or an aerodynamic airfoil. All simulations are performed on a Windows desktop platform equipped with a NVIDIA GTX TITAN GPU and an Intel core i5-3450 CPU (See Table 2 for their specifications). The performances of the developed codes will be analyzed after presenting numerical results.

Table 2 Specifications of Intel core i5-3450 CPU and NVIDIA GTX TITAN GPU

Component	Intel core i5-3450	NVIDIA GTX TITAN
Processing units	4	2 688
Frequency/GHz	3.1	837
Peak double-precision/(GFLOP·s^{-1})	99.2	1 300
Memory/GB	16	6
Memory bandwidth	25.6	288

3.1 Numerical results

3.1.1 Couette flows

In order to have a view of the accuracy and convergence orders of the developed DG codes, Couette flow between two parallel plates where the upper plate moves at a constant velocity U and the bottom plate is fixed, is firstly selected and simulated with arbitrary grids. Assume that the viscosity coefficient μ is a constant, the exact solution of this Couette flow can be expressed as

$$u=\frac{y}{H}U, v=0, p=p_\infty$$

$$T=T_0+\frac{y}{H}(T_1-T_0)+\frac{Pr}{2C_p}\frac{y}{H}\left(1.0-\frac{y}{H}\right)$$

$$\rho=\frac{p}{RT}$$

where H is the distance between two plates and y the distance between the field point and the bottom fixed plate. Following the work in Ref.[24], the other parameters are taken as: The temperature of the bottom plate T_0 is set to be 0.80 and the temperature of the upper plate T_1 is taken as 0.85. The Mach number of the upper wall $Ma_\infty=0.1$ and the Reynolds number $Re_\infty=100$. The computational domain ($0\leqslant x\leqslant 2H, 0\leqslant y\leqslant H, H=2$) is discretized with unstructured grids and structured grids. As shown in Fig. 7, three successive refinement are carried out for flow simulations with approximate order p from 1 to 3. The differences between the computed density and the exact density is used to measure the order of convergence with the L^2 errors, which are computed and listed in Table 3. The $(p+1)$th order of convergence of the present DG codes of approximate order p can be achieved as expected.

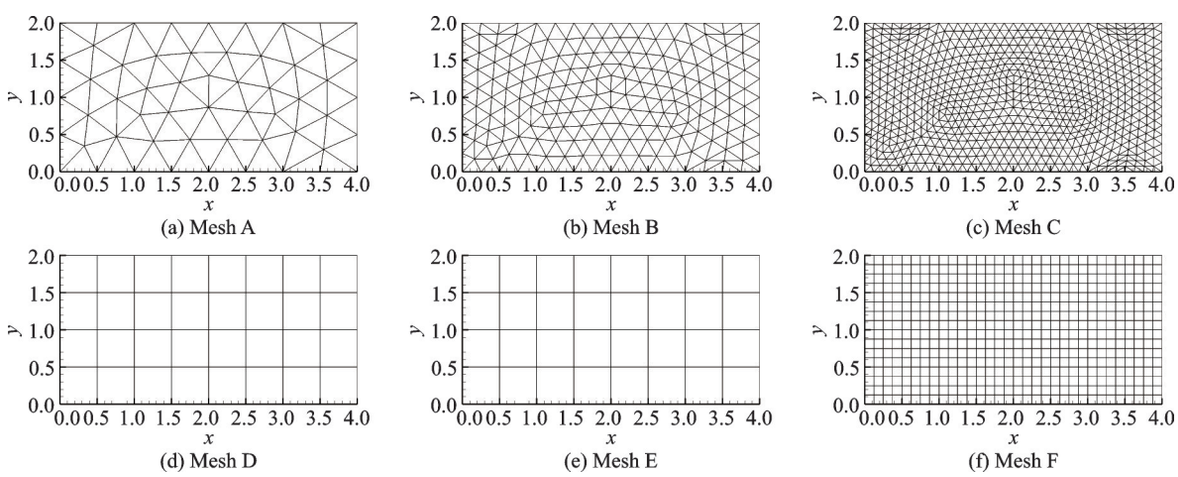


Fig.7 Successively refined meshes (Unstructured meshes:A to C; structured meshes:D to F)

Table 3 L^2 error and convergence order of Couette flow on unstructured and structured meshes

Approximate order	Unstructured			Structured		
	Mesh	L^2 error	order	Mesh	L^2 error	order
$p=1$	A	1.30E−5		D	1.66E−5	
	B	2.98E−6	2.12	E	3.62E−6	2.19
	C	7.09E−7	2.07	F	9.30E−7	1.96
$p=2$	A	7.46E−8		D	1.28E−7	
	B	1.01E−8	2.88	E	1.61E−8	2.99
	C	1.10E−9	3.19	F	2.02E−9	2.99
$p=3$	A	2.47E−10		D	4.78E−10	
	B	1.95E−11	3.66	E	3.09E−11	3.95
	C	1.46E−12	3.73	F	2.35E−12	3.71

3.1.2 Viscous flow past a circular cylinder

A compressible viscous flow past a cylinder[25-27] with conditions of $Ma_\infty=0.2$ and $Re_\infty=40$ is then selected and simulated for further validation. As shown in Fig.8, the computational

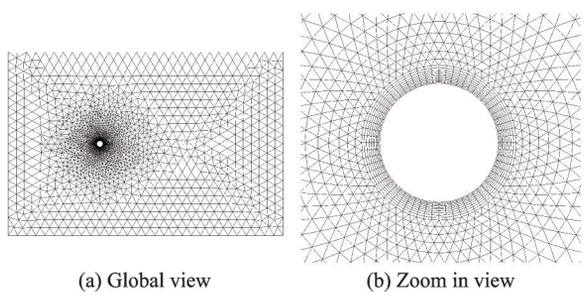
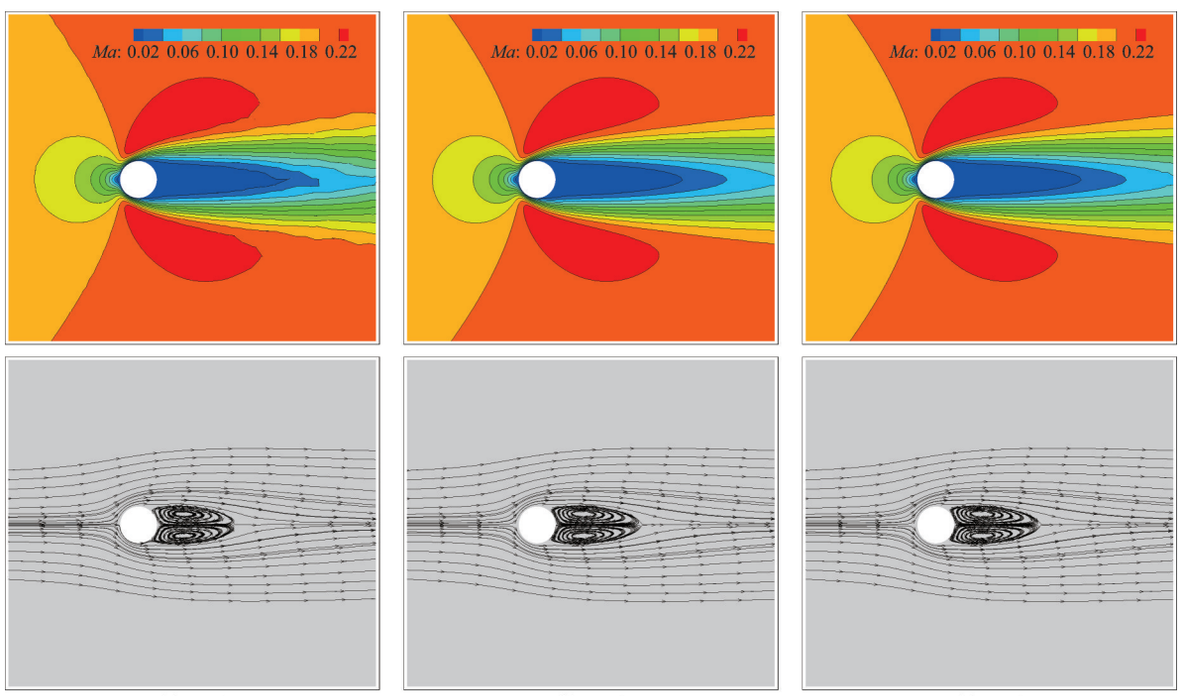


Fig.8 Computational mesh of the circular cylinder

domain Ω of $[-15, 30]\times[-15, 15]$, which is sufficiently large to eliminate the influence of far field boundary on the net drag and separation bubble length s, is discretized with 640 quadrilateral grid cells and 4 254 triangular ones for simulations. The contours of Mach number are computed as shown in Fig.9. The smoothness of the computed contours can be observed to be notably improved as the approximate order increases. Besides, the corresponding streamlines of the flow field are also presented in Fig.9 in order to have a clear view of separated bubbles. The computed total drag coefficients C_D and normalized lengths of separation bubble relative to the diameter d of the cylinder, s/d, are listed in Table 4, which are all in good agreement with established results[25-28].



(a) $p=1$ (b) $p=2$ (c) $p=3$

Fig.9 Mach number contours and streamlines with different approximate orders

Table 4 Comparison of computed drag coefficients and separation lengths for laminar flows over a circular cylinder

Approach	Method	C_D	s/d
The proposed	BR2-P1	1.563	2.09
	BR2-P2	1.554	2.32
	BR2-P3	1.553	2.32
Yang et al.[25]	DDG-P1	1.537	2.31
	DDG-P2	1.537	2.31
Ye et al.[26]	FVM	1.52	2.27
Xiao et al.[27]	BR2-P2		2.28
Visbal et al.[28]	Experiment	1.48	2.1

3.1.3 Viscous flow past NACA0012 airfoil

The last test case presented is a viscous flow past a NACA0012 airfoil. The conditions of the free-stream are Mach number $Ma_\infty = 0.5$, the angle of attack $\alpha = 0^\circ$ and $Re_c = 5\,000$, in which subscript c denotes the chord length of the airfoil used as the reference length of the Reynolds number. The adiabatic wall boundary condition is considered in this case. The computational domain is discretized with 170×30 quadrilateral cells (Fig.10) for simulations. The computed Mach number contours of the 5th order results ($p=4$) are presented in Fig.11 along with the streamlines near trailing edge of the airfoil in order to have a clear view of separated flow patterns. It can be noted that flow separations occur near the trailing edge of the airfoil, leading to the generation of two recirculation bubbles in the wake region. The corresponding pressure and skin friction coefficients, C_p and C_f, are also presented in Fig.12, which are all in good agreement with the result appeared in the open literature[29]. Additionally, in order to have a view of the effect of different ap-

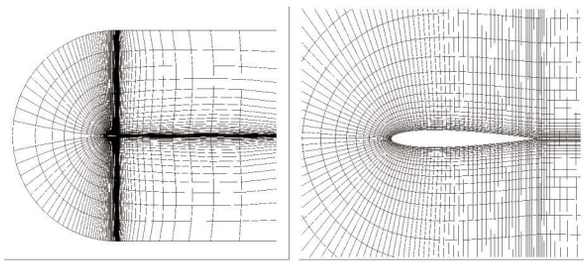


Fig.10 Computational mesh of NACA0012 airfoil

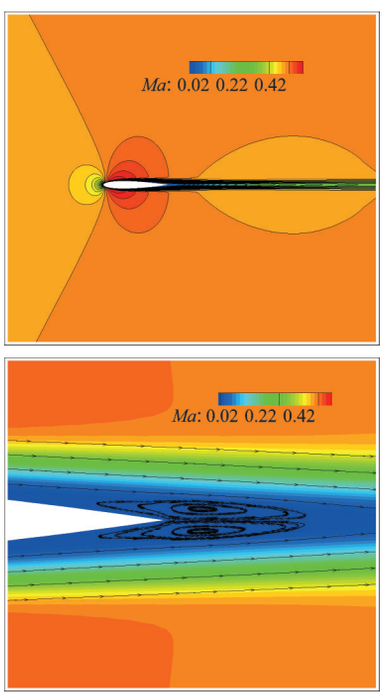


Fig.11 Mach number contours and streamlines near the trailing edge with approximate order $p=4$

proximate order, the drag coefficients due to pressure, namely C_{Dp}, are further computed and listed in Table 5 together with the drag coefficients due to viscous stress C_{Dv} and the total drag coefficient C_D. In the column Method, postfixes, P1, P2, P3 and P4, indicate the method approximated by order of 1 ,2, 3 and 4, respectively. It can be noted that the computed C_D increases as the approximate order increases like reference data[25,29] listed. In specific, the drag coefficients similar to the results of BR1-P3 and DDG-P2 can be obtained by the BR2-P3 and BR2-P4 schemes presented, and their differences of C_{Dp}, C_{Dv}, C_D are within 3% ,1% and 2%, respectively.

3.2 Performance analysis

3.2.1 Performance indicators

To have a quantitative comparison of the performances of the developed DG codes, an absolute indicator-unified time T_k is firstly defined as

$$T_k = \sigma \times \frac{T_{\text{iter}}}{N_{\text{DOF}}} \tag{20}$$

where σ is a constant for scaling the magnitude of the unified time index and is taken as 10^6 in this paper; T_{iter} the computational time of a single iteration for the DG solver; N_{DOF} the number of degrees of freedom (DOF), which depends on number of cells N_{cells}, number of conservative variables $N_{\text{variables}}$ and number of basis functions N_p. Thus, $N_{\text{DOF}} = N_{\text{cells}} \times$

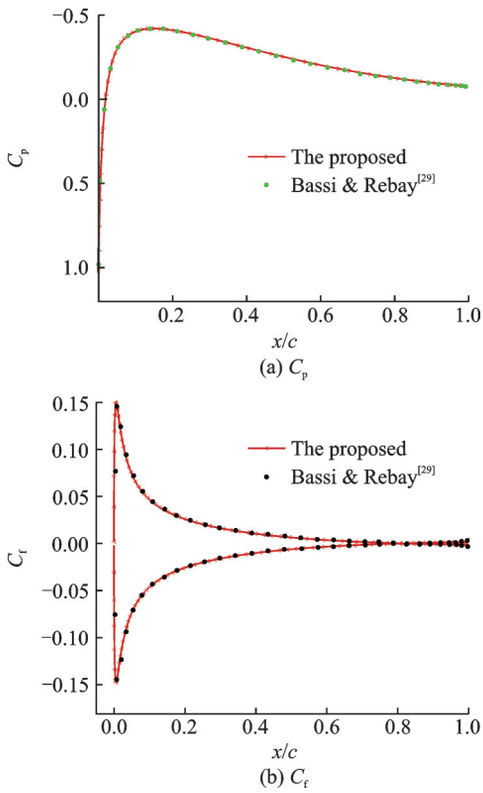


Fig.12 Distributions of pressure and skin friction coefficients with $p=4$

Table 5 Comparison of drag coefficients with different approximate orders

Approach	Method	C_{Dp}	C_{Dv}	C_D
The proposed	BR2-P1	0.016 15	0.037 49	0.053 64
	BR2-P2	0.021 77	0.034 00	0.055 77
	BR2-P3	0.022 73	0.033 23	0.055 96
	BR2-P4	0.022 87	0.033 13	0.056 00
Bassi &Rebay[29]	BR1-P1	0.019 63	0.030 51	0.050 14
	BR1-P2	0.019 91	0.033 61	0.053 52
	BR1-P3	0.022 08	0.033 01	0.055 09
Yang et al.[25]	DDG-P1	0.022 46	0.032 20	0.054 67
	DDG-P2	0.022 52	0.032 90	0.055 42

$N_{\text{variables}} \times N_{\text{p}}$. As a result, T_k can be interpreted as the time consumed in a single iteration for one million DOFs. Then, the performance of the GPU codes compared with its CPU counterparts can be quantified by a relative indicator with speedup ratio (SR)

$$\text{SR} = \frac{T_{\text{CPU}}}{T_{\text{GPU}}} \tag{21}$$

where T_{CPU} is the value associated with CPU-based simulations and T_{GPU} the value associated with GPU-based simulations. It is well known that the key kernel subroutines dominate the whole performance of the program. Therefore, the benchmark tests with related performance analysis will be firstly carried out for the key kernel subroutines before presenting the performances of the whole program.

3.2.2 Performances of key kernel subroutines

For an intuitive impression of the effect of GPU parallelization, the tests of key kernel subroutines are carried out by gradually moving the key procedures of the CPU serial code onto the GPU with computational kernels. Without loss of generality, the Couette flows presented in Section 3.1.1 are selected for these tests. The computational domain is uniformly covered by 768×256 quadrangle grid cells, which is much greater than the number of processor cores of the GPU used. Thus, the computations can make full load of the massive parallel GPU architecture. As illustrated in Fig.13, key kernel subroutines of seven procedures, namely DT, ConVars, Grad, Flux, RHS, Update, and Res, are successively analyzed. To test the DG method with approximate order $p=1$, the code is first executed on the CPU (The first-row ribbon, "all CPU", in Fig.13(a)), and the DT procedure is then moved onto the GPU using corresponding kernel subroutines, while the other procedures are kept on the CPU (The second-row ribbon, "+DT", in Fig.13(a)). After that, the remaining procedures are incrementally moved one by one onto the GPU using CUDA C kernels until all the procedures of the CPU serial code are executed on the GPU (The last row ribbon in Fig.13(a)). The tests of the DG method with other approximate orders are also carried out as illustrated in Figs.13(b—d), respectively. It can be noted that gradual acceleration can be achieved as more of the procedures are moved onto the GPU. The detailed time costs, T_k, of both CPU- and GPU-based procedures are listed in Table 6, as well as the corresponding speedup ratio SR. It can also be noted that each procedure can be significantly accelerated on GPU (Rows T_{CPU} and T_{GPU}), appeared to be dozens of times of speedup ratios (Row SR in Table 6).

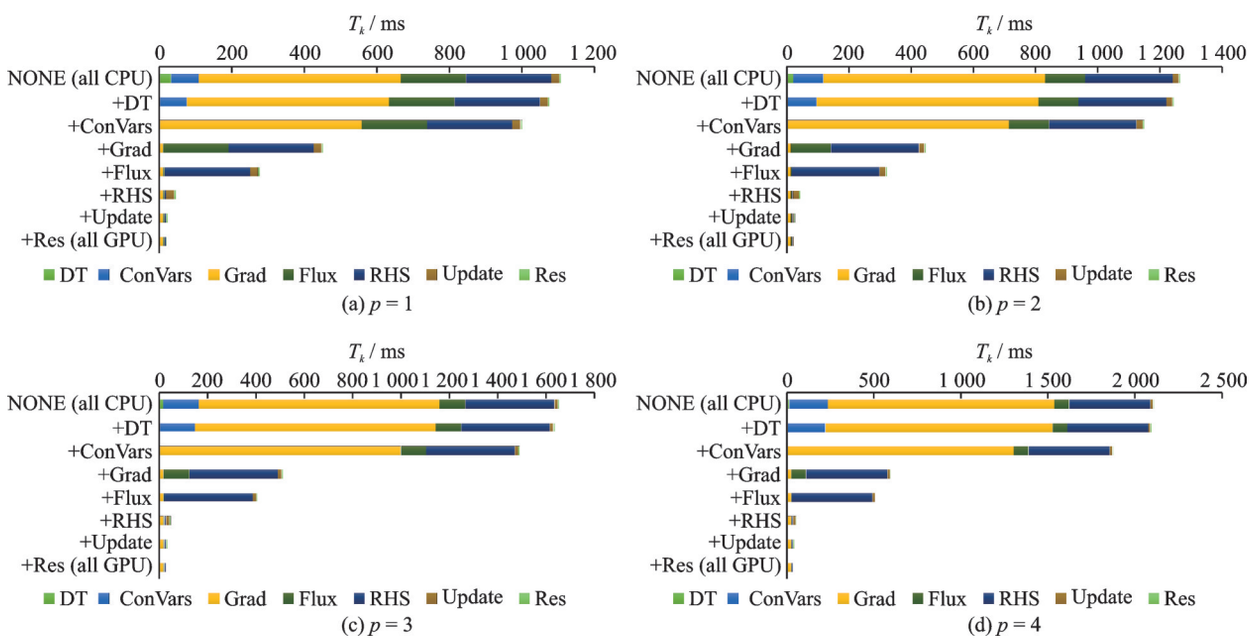


Fig.13 Accelerated progress of key computational kernels, conducted incrementally by adding more GPU to support the serial CPU implementation

Table 6 Unified time of key subroutines on CPU and GPU together with corresponding speedup ratios

Order	Parameter	DT	ConVars	Grad	Flux	RHS	Update	Res	Total
$p=1$	T_{CPU}	32.46	76.24	556.73	180.23	235.25	22.33	4.37	1107.60
	T_{GPU}	0.46	0.63	9.50	4.16	4.03	0.51	0.56	19.86
	SR	70.81	120.34	58.59	43.28	58.43	43.89	7.74	55.78
$p=2$	T_{CPU}	21.02	96.24	713.04	128.87	282.51	18.77	5.02	1265.47
	T_{GPU}	0.28	0.63	11.89	2.87	4.79	0.54	0.29	21.29
	SR	75.52	152.19	59.97	44.95	59.02	34.64	17.34	59.45
$p=3$	T_{CPU}	16.38	146.81	996.12	106.99	366.23	14.70	5.50	1652.73
	T_{GPU}	0.22	0.96	16.39	2.23	6.88	0.45	0.26	27.39
	SR	73.80	152.75	60.79	47.90	53.21	32.63	21.46	60.34
$p=4$	T_{CPU}	12.05	223.13	1301.9	86.75	464.97	12.56	5.43	2106.79
	T_{GPU}	0.15	1.42	24.16	1.46	9.64	0.44	0.23	37.49
	SR	80.33	157.69	53.89	59.42	48.23	28.55	23.61	56.19

Additionally, a benchmark test of size effect is also performed by changing the number of threads in a block. A typical set of numbers, 8, 16, 32, 64, 128 and 256, are selected to configure the sizes of thread blocks. The values of computing-time costs, varying with the size of thread block, are illustrated in Figs. 14 (a—d) for the present GPU implementation of DG methods with different approximate orders. In Fig. 14, different sizes of thread blocks and corresponding unified times are denoted by columnar ribbons and their vertical heights, respectively. It can be learned that an appropriate size of thread block, 128, can be selected due to obvious parabolic distributions (Fig.14). This number will be fixed in the following simulations.

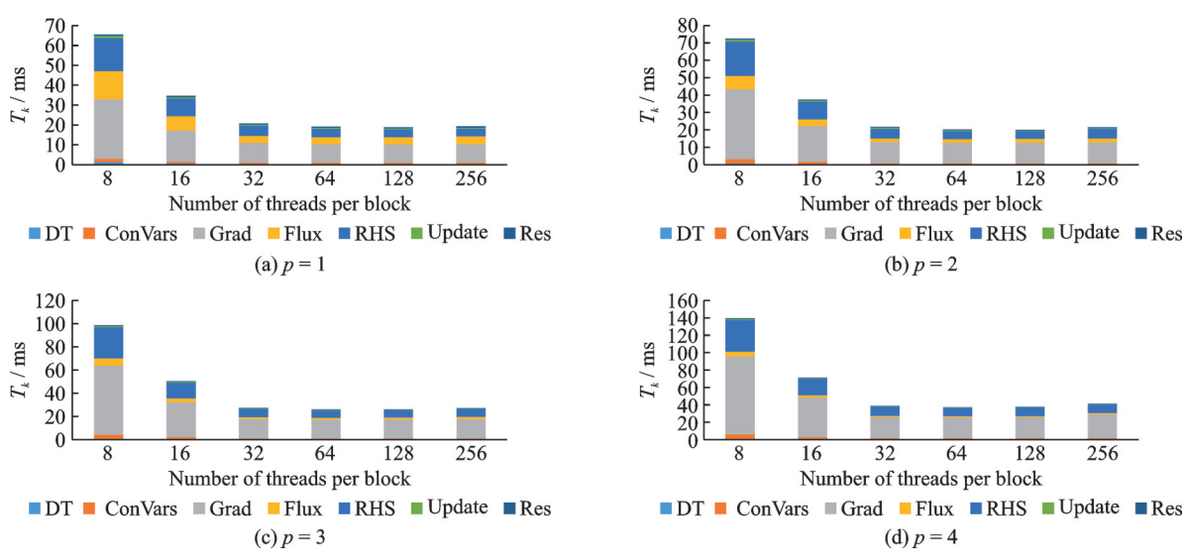


Fig.14 Size effect of threadblock

3.2.3 Performances of the proposed approach

The two unstructured and structured cases, respectively outlined in Section 3.1.2 and Section 3.1.3, are used here for testing the performances of the whole approach. For both cases, successively refined grids are employed for all related CPU and GPU computations in order to have a view of the effects of mesh scales. The corresponding unified time and speedup ratios are listed in Table 7. It can be seen that dozens of GPU speedups, between 14.68 and 69.70 times, are achieved. It can also be observed that the unified time of the serial CPU implementations in each test case are kept almost unchanged (the fourth column in Table 7) with the increasing size of mesh, while the T_k of GPU implementations obviously decrease (the fifth column of Table 7). Fig.15 presents the computational speedup ratios with respect to different mesh scales. It can

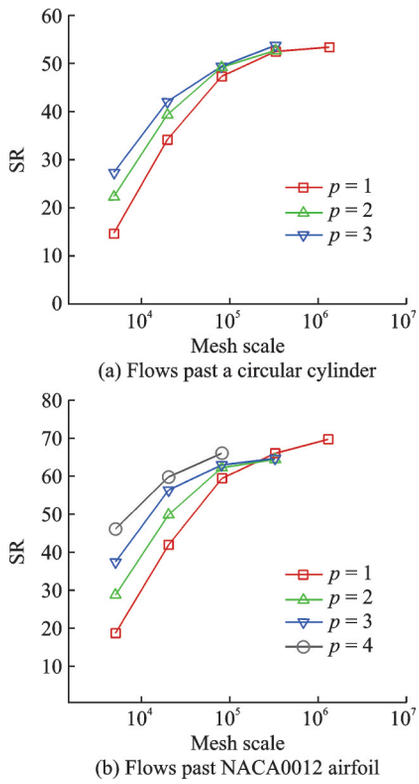
Table 7 Unified time of CPU and GPU implementations together with corresponding speedup ratio

Problem	Order	Mesh scale	T_{CPU}	T_{GPU}	SR
Flows past a circular cylinder	p=1	64 × 10 + 4 254	1 055.99	71.89	14.68
		128 × 20 + 17 230	1 153.53	33.81	34.12
		256 × 40 + 70 972	1 155.22	24.44	47.27
		512 × 80 + 291 372	1 169.51	22.30	52.44
		1 024 × 160 + 1 182 960	1 178.81	22.08	53.35
	p=2	64 × 10 + 4 254	1 200.45	53.85	22.29
		128 × 20 + 17 230	1 216.94	30.95	39.32
		256 × 40 + 70 972	1 214.26	24.70	49.15
		512 × 80 + 291 372	1 211.66	23.09	52.67
	p=3	64 × 10 + 4 254	1 527.38	55.96	27.30
		128 × 20 + 17 230	1 522.23	36.21	42.04
		256 × 40 + 70 972	1 516.52	30.72	49.37
		512 × 80 + 291 372	1 511.89	28.14	53.73
Flows past NACA0012 airfoil	p=1	170 × 30	1 290.85	69.22	18.65
		340 × 60	1 356.21	32.36	41.91
		680 × 120	1 366.42	23.01	59.38
		1 360 × 240	1 354.10	20.51	66.01
		2 720 × 480	1 379.78	19.78	69.70
	p=2	170 × 30	1 486.93	51.80	28.71
		340 × 60	1 497.14	30.03	49.85
		680 × 120	1 493.57	24.00	62.23
		1 360 × 240	1 476.32	22.85	64.60
	p=3	170 × 30	1 872.55	50.06	37.40
		340 × 60	1 863.97	33.09	56.33
		680 × 120	1 861.21	29.56	62.96
		1 360 × 240	1 836.78	28.64	64.64
	p=4	170 × 30	2 568.63	55.59	46.12
		340 × 60	2 561.27	42.82	59.81
		680 × 120	2 571.49	38.954	66.01

be observed that continual increasement of speedup ratios related to the growing mesh scales can be achieved until the total computing effort exceeds the allowed capacity of GPU architecture. The impressive GPU speedups associated with large mesh scales indicate the potentiality of the GPU-accelerated DG method for flow problems with large scales.

4 Conclusions

The GPU implementation of the DG method with different approximate orders has been developed under the CUDA C programming model and is successfully applied to solve two-dimensional laminar flows. The traditional program codes of the DG method can be easily ported from CPU to GPU through adopting the thread-per-cell or thread-per-edge models together with component-wise data storage. A resultant GPU-accelerated DG solver has been developed through the use of the designed CUDA C kernels, the constructed data storage structure and the manipulated thread hierarchy. Numerical results of the test cases presented can be in good agreement with analytical solution, available experimental data or other computational results reported in literature. It can be seen that the performance of GPU-accelerated DG solver is greatly improved with speedup ratios of multiple of 69.70 at most in comparison with that of CPU counterpart. The speedups vary increasingly with the approximate orders until exceeding the allowed capacity of GPU architecture. Besides, the present GPU implementation is developed based on modules, and hence has the flexibility to accommodate any new schemes in view of module replacement.



(a) Flows past a circular cylinder

(b) Flows past NACA0012 airfoil

Fig.15 Speedup ratios with respect to different mesh scales

Reference

[1] REED W H, HILL T R. Triangular mesh methods for the neutron transport equation: LA-UR-73-

479[R]. [S.l.]: [s.n.], 1973.

[2] CHENG J, LIU X, LIU T, et al. A parallel, high-order direct discontinuous galerkin methods for the navier-stokes equations on 3D hybrid grids[J]. Communications in Computational Physics, 2017, 21(5): 1231-1257.

[3] BASSI F, REBAY S. GMRES discontinuous Galerkin solution of the compressible Navier-Stokes equations[M]. Berlin, Germany: Springer Berlin Heidelberg, 2000.

[4] LUO H, BAUM J D, LOHNER R. A p-multigrid discontinuous Galerkin method for the Euler equations on unstructured grids[J]. Journal of Computational Physics, 2006, 211(2): 767-783.

[5] DELORME P, MAZET P, PEYRET C, et al. Computational aeroacoustics applications based on a discontinuous Galerkin method[J]. Comptes Rendus Mécanique, 2005, 333(9): 676-682.

[6] TAUBE A, DUMBSER M, DINSHAW S, et al. Arbitrary high-order discontinuous galerkin schemes for the magnetohydrodynamic equations[J]. Journal of Scientific Computing, 2007, 30(3): 441-464.

[7] XIA Y, LUO H, FRISBEY M. A set of parallel, implicit methods for a reconstructed discontinuous Galerkin method for compressible flows on 3D hybrid grids[J]. Computers & Fluids, 2014, 109: 134-151.

[8] LUO H, ALI A, NOURGALIEV R, et al. A parallel, reconstructed discontinuous Galerkin method for the compressible flows on arbitrary grids[J]. Communication in Computational Physics, 2011, 9(2): 363-389.

[9] SIEBENBORN M, SCHULZ V, SCHMIDT S. A curved-element unstructured discontinuous Galerkin method on GPUs for the Euler equations[J]. Computing and Visualization in Science, 2012, 15(2): 61-73.

[10] SIEBENBORN M, SCHULZ V. GPU accelerated discontinuous Galerkin methods for Euler equations and its adjoint[C]//Proceedings of High Performance Computing Symposium. San Diego, USA: High Performance Computing Symposium, 2013.

[11] KARAKUS A, CHALMERS N, SWIRYDOWICZ K, et al. A GPU accelerated discontinuous Galerkin incompressible flow solver[J]. Journal of Computational Physics, 2019, 390: 380-404.

[12] STEWART N J, HOLMES D W, LIN W X, et al. Comparison of semi-implicit and explicit finite difference algorithms on highly parallel processing architectures[J]. Applied Mechanics & Materials, 2014, 553: 193-198.

[13] MA W, LU Z, ZHANG J. GPU parallelization of unstructured/hybrid grid ALE multigrid unsteady solver for moving body problems[J]. Computers & Fluids, 2015, 110: 122-135.

[14] REMACLE J F, GANDHAM R, WARBURTON T. GPU accelerated spectral finite elements on all-hex meshes[J]. Journal of Computational Physics, 2016, 324: 246-257.

[15] ZHANG J, CHEN H, CAO C. A graphics processing unit-accelerated meshless method for two-dimensional compressible flows[J]. Engineering Applications of Computational Fluid Mechanics, 2017, 11(1): 526-543.

[16] KLÖECKNER A, WARBURTON T, BRIDGE J, et al. Nodal discontinuous Galerkin methods on graphics processors[J]. Journal of Computational Physics, 2009, 228(21): 7863-7882.

[17] FUHRY M, GIULIANI A, KRIVODONOVA L. Discontinuous Galerkin methods on graphics processing units for nonlinear hyperbolic conservation laws[J]. International Journal for Numerical Methods in Fluids, 2015, 76(12): 982-1003.

[18] TORO E F. Riemann solvers and numerical methods for fluid dynamics[M]. Berlin, Germany: Springer Berlin, 2013: 87-114.

[19] BASSI F, REBAY S. Discontinuous Galerkin solution of the Reynolds-averaged Navier-Stokes and k-ω turbulence model equations[J]. Computers & Fluids, 2005, 34(4/5): 507-540.

[20] LUO H, BAUM J D, LHNER R. A discontinuous Galerkin method using Taylor basis for compressible flows on arbitrary grids[J]. Journal of Computational Physics, 2008, 227(20): 8875-8893.

[21] LÜBON C, KESSLER M, WAGNER S, et al. High-order boundary discretization for discontinuous Galerkin codes[C]//Proceedings of the 24th AIAA Applied Aerodynamics Conference. San Francisco, USA: AIAA, 2006.

[22] SHU C W, OSHER S. Efficient implementation of essentially non-oscillatory shock-capturing schemes[J]. Journal of Computational Physics, 1988, 77(2): 439-471.

[23] NVIDIA. CUDA C programing guide[EB/OL]. [2022-08-03]. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[24] CHENG J, LIU T, LUO H. A hybrid reconstructed discontinuous Galerkin method for compressible flows on unstructured grids[C]//Proceedings of the 54th AIAA Aerospace Sciences Meeting. San Diego, USA: AIAA, 2016.

[25] YANG X, CHENG J, WANG C, et al. A fast, implicit discontinuous Galerkin method based on analytical Jacobians for the compressible Navier-Stokes equations[C]//Proceedings of the 54th AIAA Aerospace Sciences Meeting. San Diego, USA: AIAA, 2016.

[26] YE T, MITTAL R, UDAYKUMAR H S, et al. An accurate cartesian grid method for viscous incompressible flows with complex immersed boundaries[J]. Journal of Computational Physics, 1999, 156(2): 209-240.

[27] XIAO H, FEBRIANTO E, ZHANG Q, et al. An immersed discontinuous Galerkin method for compressible Navier-Stokes equations on unstructured meshes[J]. International Journal for Numerical Methods in Fluids, 2019,91(10): 487-508.

[28] VISBAL M R. Evaluation of an implicit Navier-Stokes solver for some unsteady separated flows[C]// Proceedings of the 4th Joint Fluid Mechanics, Plasma Dynamics and Lasers Conference. Atlanta, USA: AIAA, 1986.

[29] BASSI F, REBAY S. A high-order accurate discontinuous finite element method for the numerical solution of the compressible Navier-Stokes equations[J]. Journal of Computational Physics, 1997, 131(2): 267-279.

Acknowledgements This work was partially supported by the National Natural Science Foundation of China (No. 11972189), the Natural Science Foundation of Jiangsu Province (No.BK20190391), the Natural Science Foundation of Anhui Province (No.1908085QF260), and the Priority Academic Program Development of Jiangsu Higher Education Institutions.

Authors Mr. **GAO Huanqin** received the B.S. degree from College of Aerospace Engineering, Nanjing University of Aeronautics and Astronautics, China, in 2013. Now he is pursuing the Ph.D. degree in Nanjing University of Aeronautics and Astronautics, majoring in computational aerodynamics, discontinuous Galerkin methods, and GPU parallel computing.

Prof. **CHEN Hongquan** received the M.S. and Ph.D. degrees in aerodynamics from Nanjing University of Aeronautics and Astronautics, China, in 1987 and 1990, respectively. He is currently a professor in the College of Aerospace Engineering, Nanjing University of Aeronautics and Astronautics. His main research interests include computational aerodynamics, computational electromagnetics, multidisciplinary optimizations, and high-performance parallel computing.

Author contributions Mr. **GAO Huanqin** designed the study, conducted the analysis and wrote most of the manuscript. Prof. **CHEN Hongquan** instructed the study and checked the manuscript. Dr. **ZHANG Jiale** contributed to the design and discussion of the study. Dr. **XU Shengguan** and Dr. **GAO Yukun** contributed to the background of the study. All authors commented on the manuscript draft and approved the submission.

Competing interests The authors declare no competing interests.

(Production Editor: ZHANG Bei)

基于GPU加速的间断Galerkin方法求解二维层流

高缓钦[1], 陈红全[1], 张加乐[1], 徐圣冠[1], 高煜堃[2]

(1.南京航空航天大学航空学院,非定常空气动力学与流动控制工信部重点实验室,南京 210016,中国;
2.安徽工业大学机械工程学院,马鞍山 243002,中国)

摘要:发展了一种基于GPU加速的间断Galerkin(DG)方法,用以计算二维层流流动。通过统一设备架构(Compute unified device architecture, CUDA),结合DG方法高阶精度的特点,基于分量存储涉及的数据,基于单元或边组织线程结构,将求解Navier-Stokes方程的DG CPU程序移植到GPU,涉及的无黏和黏性数值通量分别采用Local Lax-Friedrichs格式和BR2格式,时间推进采用强稳定性保持的Runge-Kutta格式。首先在不同的网格尺寸和拟合精度下计算Couette流动问题,证实了算法能取得预期的收敛精度;接着将圆柱绕流和NACA0012翼型绕流的计算结果与文献和实验结果进行对比,并依据加速比分析程序性能。结果表明在不损失准确度的前提下,与CPU程序相比,本文发展的GPU程序能显著减少计算时间,最高能够取得69.7倍的加速比。

关键词:间断Galerkin;GPU;统一设备架构;Navier-Stokes方程;层流