GPU IMPLEMENTATION OF PRECONDITIONING METHOD FOR LOW-SPEED FLOWS

ZHANG JIALE

*College of Aerospace Engineering, Nanjing University of Aeronautics and Astronautics, YuDao Street
Nanjing City, Jiangsu Province, China
zhjl_nuaa@163.com*

CHEN HONGQUAN

*College of Aerospace Engineering, Nanjing University of Aeronautics and Astronautics, YuDao Street
Nanjing City, Jiangsu Province, China
hqchenam@nuaa.edu.cn*

Published 22 June 2016

An improved preconditioning method for low-Mach-number flows is implemented on a GPU platform. The improved preconditioning method employs the fluctuation of the fluid variables to weaken the influence of accuracy caused by the truncation error. The GPU parallel computing platform is implemented to accelerate the calculations. Both details concerning the improved preconditioning method and the GPU implementation technology are described in this paper. Then a set of typical low-speed flow cases are simulated for both validation and performance analysis of the resulting GPU solver. Numerical results show that dozens of times speedup relative to a serial CPU implementation can be achieved using a single GPU desktop platform, which demonstrates that the GPU desktop can serve as a cost-effective parallel computing platform to accelerate CFD simulations for low-Speed flows substantially.

Keywords: GPU acceleration; preconditioning method; Navier-Stokes equations; CUDA fortran.

1. Introduction

Nowadays the traditional time-marching algorithms are successfully used in the computation of compressible flows controlled by the Navier-Stokes equations, because of the freedom to change the governing partial differential equations as long as the steady solution is not affected¹. However, such algorithms suffer degradation in performance when simulating complex flow fields of particular geometric configurations, such as helicopters, strongly convergent nozzles and micro aerial vehicles. In these kind of complex flow fields, part of the flow regions can be considered to be incompressible with locally low Mach numbers, whereas other regions of the flow are significant compressible and can even contain shocks. When using the designed compressible flow solvers to simulate these nearly incompressible flow problems, the numerical solution

This is an Open Access article published by World Scientific Publishing Company. It is distributed under the terms of the Creative Commons Attribution 3.0 (CC-BY) License. Further distribution of this work is permitted, provided the original work is properly cited.

may have problem of convergence to the correct result and even fail to be convergent. The difficulty is attributed to the large disparity between the time scales of the acoustic waves and convective speed.

The difficulty mentioned can be overcome via techniques of preconditioning by modifying the transient properties of the time-dependent equations without affecting the steady solution. Most common way is realized by pre-multiplying the time derivatives with a matrix that slows the speed of the acoustic waves relative to the fluid speed. A type of Weiss and Smith² preconditioning matrix is selected and considered to be implemented on our single graphics processing unit (GPU) desktop platform.

The GPU brings an efficiently parallel way for solving the complex flow problems in computational fluid dynamics (CFD). In recent years, many successful attempts have been achieved³⁻⁶. For example, Brandvik and Pullan³ completed porting a two- and three-dimensional Euler code for modeling inviscid flow onto the GPU. The resultant solver ran 30 times faster for a two-dimensional case and 15 times faster for a three-dimensional case. At around the same time, Elsen et al.⁴ re-wrote part of the Navier-Stokes Stanford University solver (NSSUS) to model hypersonic flow on the GPU. The performance of their code ranges from 15 to 40 times speed-up compared to the original solver. Molemaker et al.⁵ developed a multi-grid method to solve the pressure Poisson equation. The GPU implementation of the multi-grid pressure Poisson solver produced a speedup of $55\times$ relative to a 2.2MHz AMD Opteron processor⁵. In addition to the finite volume method, the Discontinuous Galerkin (DG) method has also been implemented on the GPU by Klockner et al.⁶. In these attempts, the NVIDIA's Compute Unified Device Architecture (CUDA)⁷ is the most commonly used for their GPU codes. The similar technology of CUDA mentioned is also selected for our single-precision GPU codes in the present studies.

As is well-known, the CPU's single-precision float-point operation ability performances much better than the double-precision, that is same to the GPU. For example, the GPU used in our research, the NVIDIA GTX Titan GPU, performances 4.5T FLOPS for single-precision float-point operations in theory, that is three times over double-precision float-point operations (1.5T FLOPS)⁸. The difficulty in solving the CFD problems with single-precision is attributed to the rapid growth of the truncation error of the computer, which influences the calculating time and convergence process, especially seriously in solving the low speed flows. In order to overcome the difficulty, an improved preconditioning method is adopted in discretization the control equations. The details will be presented and analyzed in next section.

In this paper, we focus on developing an efficient preconditioned Reynolds averaged Navier-Stokes solver for simulating low-speed flows on the GPU parallel computing platform. An improved method is introduced and discussed to overcome the insufficient accuracy of single-precision. Then, the techniques of GPU implementation of CUDA kernels, double-layered thread hierarchy and variety memory hierarchy are adopted to develop a GPU-based parallel solver for low speed flows in CUDA Fortran programming language. A set of typical flow cases are selected for both validation and performance

analysis. It can be learned from the corresponding numerical results that the improved preconditioning method performs very well in view of obtaining reasonable flow fields and surface pressure coefficients, and dozens of times speedup (about 35 times upmost for present test cases) relative to a serial CPU implementation can be achieved for the solver developed.

2. Methodology

The governing equations of the flow considered are the nonlinear time-dependent compressible Reynolds averaged Navier-Stokes equations of the laminar without body forces. The finite volume method (FVM) in cell-centered scheme is adopted: the convective flux term is evaluated by central scheme with artificial dissipation, the viscous flux term is evaluated by central scheme, and the time integration employs an explicit four-stage Runge-Kutta scheme. In order to overcome the insufficient accuracy of single-precision when calculation, an improved method is introduced and discussed here. Meanwhile, the preconditioning method introduced by Weiss and Smith is implemented to solve the equations for low-speed flows.

2.1 Governing Equations

The nonlinear time-dependent compressible Reynolds averaged Navier-Stokes equations can be expressed in conservation form as:

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

Here, t is the time, $\mathbf{W}$, $\mathbf{F}_c$ and $\mathbf{F}_v$ are vectors of conservative variables, convective flux terms and viscous flux terms respectively. They are defined as

$$\mathbf{W} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{bmatrix} \quad \mathbf{F}_c = \begin{bmatrix} \rho u & \rho v & \rho w \\ \rho uu + p & \rho uv & uw \\ \rho uv & \rho vv + p & \rho vw \\ \rho uw & \rho vw & \rho ww + p \\ \rho uh & \rho vh & \rho wh \end{bmatrix} \quad \mathbf{F}_v = \begin{bmatrix} 0 & 0 & 0 \\ \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \\ \Phi_x & \Phi_y & \Phi_z \end{bmatrix} \quad (2)$$

The components of the viscous stress tensor τ_{ij} and the components Φ_i are defined by

$$\tau_{ij} = \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_k}{\partial x_k} \delta_{ij}$$

$$\Phi_j = u \tau_{jx} + v \tau_{jy} + w \tau_{jz} + k \frac{\partial T}{\partial x_j} \quad (3)$$

where ρ , p , T , e and h denote the density, pressure, temperature, total energy per unit mass, and total enthalpy per unit mass, respectively. μ and k denote the viscosity coefficient and

the thermal conductivity coefficient, respectively. u , v and w are the cartesian components of the velocity vector. For a perfect gas, these quantities satisfy:

$$h = e + \frac{p}{\rho} \quad p = (\gamma - 1) \left(\rho e - \frac{1}{2} \rho (u^2 + v^2 + w^2) \right) \quad \text{and} \quad p = \rho R T \quad (4)$$

For perfect gas, the ratio of specific heats of fluid $\gamma = 1.4$.

2.2 Numerical Approach

The system of Eq. (1) can be discretized using the finite volume method, for control volume i , the discretization equation is derived as:

$$\Omega_i \frac{\Delta \mathbf{W}_i}{\Delta t} + \mathbf{R}_i = 0 \quad (5)$$

Here,

$$\begin{aligned} \Delta \mathbf{W}_i &= \mathbf{W}_i^{n+1} - \mathbf{W}_i^n & \mathbf{R}_i &= \mathbf{R}_{c,i} - \mathbf{R}_{v,i} \\ \text{and} \quad \mathbf{R}_{c,i} &= \sum_{j=1}^{NF} \mathbf{F}_{c,ij} \cdot \mathbf{n} \Delta S_{ij} & \mathbf{R}_{v,i} &= \sum_{j=1}^{NF} \mathbf{F}_{v,ij} \cdot \mathbf{n} \Delta S_{ij} \end{aligned} \quad (6)$$

where Ω is the volume of the control volume i and ΔS_{ij} is the area of edge j of the control volume i . The subscript j denotes the j -th face of the control volume i .

The convective flux term is evaluated by central scheme with artificial dissipation, as

$$\begin{aligned} \mathbf{R}_{c,i} &= \mathbf{Q}_i - \mathbf{D}_i \\ \mathbf{Q}_i &= \sum_{j=1}^{NF} \mathbf{F}_{c,ij} \left(\frac{\mathbf{W}_L + \mathbf{W}_R}{2} \right) \cdot \mathbf{n} \Delta S_{ij} \end{aligned} \quad (7)$$

Here, the construction of the artificial dissipative term is given by

$$\mathbf{D}_i = \sum_{j=1}^{NF} \lambda_j \varepsilon_j^{(2)} (\mathbf{W}_j - \mathbf{W}_i) - \sum_{j=1}^{NF} \lambda_j \varepsilon_j^{(4)} (\nabla^2 \mathbf{W}_j - \nabla^2 \mathbf{W}_i) \quad (8)$$

where $\varepsilon^{(2)}$ and $\varepsilon^{(4)}$ are adaptive coefficients. $\lambda = |U| + c$ is spectral radius (the max absolute value of the eigenvalues) of the Jacobian matrix. Here $U = \mathbf{V} \cdot \mathbf{n}$, and $\mathbf{V}$ and $\mathbf{n}$ are the total velocity and the cell face vector respectively, and $c = \sqrt{\gamma p / \rho}$ is the speed of sound. The detailed description of the construction of the dissipative term can be found in Ref. [9].

The viscous flux term is evaluated by central scheme, as

$$\mathbf{R}_{v,i} = \sum_{j=1}^{NF} \mathbf{F}_{v,ij} \left(\frac{\mathbf{W}_L + \mathbf{W}_R}{2} \right) \cdot \mathbf{n} \Delta S_{ij} \quad (9)$$

In order to obtain the steady solution, an explicit four-stage Runge-Kutta scheme is adopted for time integration:

$$\begin{cases} W_i^{(0)} = W_i^n \\ W_i^{(m)} = W_i^{(0)} - \alpha_m \frac{\Delta t}{\Omega_i} R_i(W_i^{(m-1)}) \quad m=1,2,3,4 \\ W_i^{n+1} = W_i^{(4)} \end{cases} \quad (10)$$

where the superscript n and $n+1$ denote the current and the next time level. The coefficient α_m can be found in Ref. [9]. To accelerate the convergence, Δt is taken as the local time step⁹.

In the cases of inviscid flows governed by the Euler equations, the fluid slips over the solid wall and there is no flow normal to the surface, that is $\vec{V} \cdot \vec{n} = 0$. And in the cases of viscous flow governed by the Navier-Stokes equations, the relative velocity between the surface and the fluid directly at the surface is zero, that is $u = v = 0$. In the far field, the Riemann boundary condition is adopted, and the details can be found in Ref. [9].

2.3 A Description of Preconditioning Method

The preconditioned system is obtained by multiplying the time derivative in Eq. (1) by a matrix Γ^{-1} :

$$\Gamma^{-1} \frac{\partial \mathbf{W}}{\partial t} + \nabla \cdot (\mathbf{F}_c - \mathbf{F}_v) = 0 \quad (11)$$

where Γ^{-1} represents the preconditioning matrix based on the conservative variables. Eq. (11) can be further written as

$$\frac{\partial \mathbf{W}}{\partial t} + \Gamma \cdot [\nabla \cdot (\mathbf{F}_c - \mathbf{F}_v)] = 0 \quad (12)$$

The choice is not unique for the preconditioning matrix for low Mach number flows. As a well-known and widely used preconditioning matrix, Weiss-Smith preconditioning matrix² is used in this paper, and it can be written as:

$$\Gamma = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \varepsilon \end{bmatrix} + \frac{\gamma-1}{c^2} (\varepsilon-1) \begin{bmatrix} q & -u & -v & -w & 1 \\ qu & -uu & -uv & -uw & u \\ qv & -uv & -vv & -vw & v \\ qw & -uw & -vw & -ww & w \\ qh & -uh & -vh & -wh & q \end{bmatrix} \quad (13)$$

where $q = (u^2 + v^2 + w^2) / 2$, and the preconditioning coefficient ε is defined as

$$\varepsilon = \max[\varepsilon_{inv}, \varepsilon_{vis}] \quad (14)$$

Here, ε_{inv} is chosen in regions of the flow where viscous effects are negligible (large cell Reynolds number) and ε_{vis} is chosen in regions of the flow where viscous effects are dominant; in this way preconditioner (13) controls both the inviscid and viscous timescales at the same time. The definition of the variable ε_{inv} and ε_{vis} are,

$$\varepsilon_{inv} = \min \left[\max (M^2, M_\infty^2), 1 \right]$$

$$\varepsilon_{vis} = \frac{(\text{CFL} / \text{VNN}) \cdot (\text{CFL} / \text{VNN} - \text{Re}_{\Delta x}) M^2}{\text{Re}_{\Delta x}^2 + \text{Re}_{\Delta x} \cdot (\text{CFL} / \text{VNN} - \text{Re}_{\Delta x}) M^2} \quad (15)$$

where the Courant-Friedrichs-Lewy number (CFL) and Von Neumann number (VNN) are used. M and M_∞ represent the local Mach number and the freestream Mach number respectively. $\text{Re}_{\Delta x}$ is the cell Reynolds number defined as $\text{Re}_{\Delta x} = U \Delta x / \nu$, here U and ν denote the local speed and the kinematic coefficient of viscosity of the fluid respectively. Δx is the scale of the grid, it can be defined as $\Delta x = Q^{1/2}$ for 2-D flows and $\Delta x = Q^{1/3}$ for 3-D flows. The preconditioning coefficient ε_{vis} is approach to zero for large Reynolds number flows.

The difficulty in solving the low Mach number flows of the traditional finite volume method (FVM) without preconditioning is associated with the large disparity in the magnitudes of eigenvalues². The application of preconditioning changes the eigenvalues of the system and scales them to the same order of magnitude. The eigenvalues of the preconditioned system (11), when $\text{Re} \gg 1$, are

$$\lambda'_0 = U, U, U \quad (\text{repeated eigenvalues})$$

$$\lambda'_\pm = \frac{1}{2} \left[(1 + \varepsilon)U \pm \sqrt{(1 + \varepsilon)^2 U^2 + 4(1 - U^2 / \gamma RT) \varepsilon \gamma RT} \right] \quad (16)$$

As the max absolute value of the eigenvalues, the spectral radius used in the artificial dissipative term (8) is changed into

$$\lambda' = \frac{1}{2} (1 + \varepsilon) \cdot |U| + \sqrt{\frac{1}{4} (1 + \varepsilon)^2 U^2 + (1 - U^2 / \gamma RT) \varepsilon \gamma RT} \quad (17)$$

To make the computation stable, a new artificial dissipative term D'_i should be added and take the form as

$$D'_i = \sum_{j=1}^{NF} \lambda'_j \varepsilon_j^{(2)} (W_j - W_i) - \sum_{j=1}^{NF} \lambda'_j \varepsilon_j^{(4)} (\nabla^2 W_j - \nabla^2 W_i) \quad (18)$$

As compared with Eq. (7), adaptive coefficients, $\varepsilon^{(2)}$ and $\varepsilon^{(4)}$ are now multiplied by a modified spectral radius λ' defined in Eq. (17). Therefore, Eq. (5) and (7) can be rewritten as:

$$\Omega_i \frac{\Delta W_i}{\Delta t} + \Gamma \cdot (Q_i - R_{v,i}) - D'_i = 0 \quad (19)$$

The explicit four-stage Runge-Kutta time integration scheme like Eq. (10) can also be used for the preconditioned system.

The preconditioning method changes the time dependent equations, and also changes the characteristics of the system. Hence, the boundary conditions in the far field, based on characteristic variables, must be modified for the preconditioned system. Depending on whether the subsonic flow is an inflow or an outflow, a simplified far-field boundary conditions proposed by Turkel¹⁰ is used in this paper, at the inflow

$$\begin{aligned} u_b &= u_\infty, v_b = v_\infty, w_b = w_\infty \\ \rho_b &= \rho_\infty, p_b = p_{\text{int}} \end{aligned} \quad (20)$$

And at the outflow

$$\begin{aligned} u_b &= u_{\text{int}}, v_b = v_{\text{int}}, w_b = w_{\text{int}} \\ \rho_b &= \rho_{\text{int}}, p_b = p_\infty \end{aligned} \quad (21)$$

where the subscript ‘*b*’ denotes the variables to be computed at the boundary, the subscript ‘*int*’ denotes the variables from the interior of the flow field, and subscript ‘ ∞ ’ denotes the freestream variables. For supersonic boundaries, standard boundary conditions are used, that is extrapolation at the outflow boundary and free-stream values for all variables at the inflow boundary are prescribed.

2.4 Problem and Treatment for using Single-precision Operations

As is well-known, the significant digit of a standard double precision float number is 16, but the significant digit of a standard single precision float number is only 7. The truncation error will grow rapidly when solving the CFD problems in single-precision, so we need an effective method to overcome this problem.

Based on the analysis of the magnitude order, we can get

$$\frac{\partial \rho}{\partial \rho_\infty} = O(M^2) \quad \frac{\partial p}{\partial p_\infty} = O(M^2) \quad \frac{\partial T}{\partial T_\infty} = O(M^2) \quad (22)$$

For the low-Mach-number flows, the changes of these variables cannot be represented accurately, especially in single-precision calculation. The reference value of these variables can be introduced to reduce the truncation error,

$$\begin{aligned} \rho &= \rho' + \rho_\infty & p &= p' + p_\infty & T &= T' + T_\infty \\ \rho e &= \rho e' + \frac{p_\infty}{(\gamma - 1)} & \text{and} & & \rho h &= \rho h' + \frac{\gamma p_\infty}{(\gamma - 1)} \end{aligned} \quad (23)$$

Here, variables without subscript and superscript denote the absolute values, variables with subscript denote the reference values, and variables with superscript denote the fluctuation value.

Considered that $\partial C_{\text{const}} / \partial t \equiv 0$ and $\partial C_{\text{const}} / \partial x_i \equiv 0$ are valid for the const, the fluctuation variables, such as fluctuation density, fluctuation velocity and fluctuation pressure, are used in equations (1). Then change the governing equations to the form as:

$$\frac{\partial W'}{\partial t} + \nabla \cdot (\vec{F}'_c - \vec{F}'_v) = 0$$

$$W' = \begin{bmatrix} \rho' \\ \rho u \\ \rho v \\ \rho w \\ \rho e' \end{bmatrix} \quad \vec{F}_c = \begin{bmatrix} \rho u & \rho v & \rho w \\ \rho uu + p' & \rho uv & \rho uw \\ \rho uv & \rho vv + p' & \rho vw \\ \rho uw & \rho vw & \rho ww + p' \\ \rho uh' & \rho vh' & \rho wh' \end{bmatrix} \quad (24)$$

The code of the discretization algorithms are improved with the above form (24).

3. Programming Model and GPU Implementation

The NVIDIA's Compute Unified Device Architecture (CUDA)⁷ provides the developers a parallel computing platform and programming model to leverage the powerful compute engine in their GPUs. In CUDA, the developers can use a software environment, an extension to the C, FORTRAN and some other programming languages, to launch and manage massively parallel computations on the GPUs. The reader can refer to the CUDA programming guide for more details⁷. In this section we summarize the GPU hardware architecture and the programming model with CUDA Fortran. Hereinafter, we use the term "host" to refer to the CPU and the term "device" to refer to the GPU.

3.1 GPU Architecture

GPU is a set of stream multiprocessors, which is an extension of the SIMD (single instruction, multiple data) paradigm architecture. This design architecture, as the name suggests, makes CUDA optimal for performing a single instruction in parallel on different sets of data on NVIDIA GPUs. A full Kepler GK110 (the latest generation GPU architecture) is composed of fifteen Streaming Multiprocessors (SMX) and six 64-bit memory controllers. As shown in Figure 1, each SMX contains a collection of processing cores (192 in Kepler architecture), shared memory, Register File, L1 cache memory, read-only data cache, L2 cache memory, Warp Scheduler, and other processing or control units. Each green border square represents a CUDA core which is then mapped to a thread by the runtime system. Closer to the core, the local registers allow fast ALU operations. The shared memory which is also closer to the processors will be used to store data shared by all the cores of a SMX. The global memory (DRAM) is used to store the main computing data. And different kinds of cache, include L1 cache, read-only data cache and L2 cache, are used to provide efficient, high speed data access of global memory by processors. Depending on the GPU architecture, a wide variety of combinations of cache memory, global memory (DRAM), number of SMs, and thread count are available. CUDA-enabled GPUs with the Kepler architecture are powerful processors that contain thousands of cores which are able of converting a simple computer into a high-performance computer.

Fig. 1. Kepler GPU Architecture.

3.2 Programming Model

In the CUDA programming model, the parallel code executed on the GPU is called the kernel. Before executing a kernel, the processing data must be transferred to memory on the GPU. Next, the CPU initiates kernel execution on the GPU. After the kernel is completed executed, the CPU retrieves the processed data from the GPU. This process is illustrated in Figure 2(a).

(a) CUDA Process Flow

Fig. 2. CUDA Process Flow and CUDA Thread Organization.

The kernel is launched from the host side (CPU), and it is organized to a thread grid, as illustrated in Figure 2(b). Each grid contains a set of blocks, and each block contains a set of threads. The number of threads per block and the grid size (number of blocks) need to be defined before launching the kernel. All the threads from a specific block have to execute on the same SMX, they can access to the same shared memory and can be synchronized. On the other hand, threads from different blocks cannot synchronize and can exchange data only through the global memory.

CUDA FORTRAN API is an extension to the Fortran programming language. It provides functions and keywords to manage the computations on the GPU. The reader can refer to the *CUDA Fortran Programming Guide and Reference*¹⁰ for more details. In CUDA FORTRAN API, the qualifiers *attributes (host)*, *attributes (global)* and *attributes (device)* specify whether the CPU or the GPU should execute and call the qualified subroutine or function. The qualifiers *device*, *shared* and *constant* are introduced to define the type of memory a variable should use, and the function *cudaMalloc()*, *cudaMemcpy()* and *cudaFree()* are respectively used to allocate memory on the GPU, copy data between CPU memory and device memory, and free memory on the GPU.

In addition, the kernel is launched by specifying the size of the grid (number of blocks) and the size of the block (number of threads per block) using the flowing prototype: *kernel_function*<<<*gridDim*, *blockDim*>>>(). The function *syncthreads()* is used inside a kernel to synchronize all the threads of a same block, and the function *cudaDeviceSynchronize()* is used inside a host function to synchronize the host(CPU) and device(GPU).

3.3 CUDA Implementation

Explicit Runge-Kutta scheme is selected for present CUDA Implementation. Since the algorithm is explicit in both space and time, threads can run independently of each other for a majority of the time-step. This ensures the thread processors are fully used and none are idle. There are only a few thread synchronization must occur, such as after updating the flow variables in grid elements, or when needing judged the convergence. In Additional, the algorithm was designed in unstructured model, that suitable for both structured and unstructured grid. Because of that, the grid cells are arranged into one dimensional form, and the size of the thread block and thread grid are both in one-dimensional form, as shown in Figure 3. Each thread corresponds to a grid element, and takes charge of the computation of the specified element. Every N_t (thread number per block) counts of threads are combined into a thread block, and the size of thread grid is defined to $(nElem+N_t-1)/N_t$. Thus all the thread count is $nElem+N_t-1$, a little larger than the element count $nElem$ (number of elements), and the threads can cover all the elements that need to be solved.

Fig. 3. The 1-D permutation and correspondence between grid elements and GPU threads.

4. Results and Discussions

The computing hardware utilized in this study is a dual-CPU/single-GPU platform, built with an Intel Core i5-3450 Quad 3.1 GHz CPU, 16GB of memory and a NVIDIA GTX Titan board [8]. The Titan board provides 2688 streaming processor cores and 6GB of

global device memory, and can deliver a theoretical peak performance of 4.5 TFLOPS in single-precision operations.

In this section the numerical results obtained simulating inviscid and viscous flows around several geometries of aeronautical interest are presented. First, the method is validated via the comparison of results with those provided by the theory for the inviscid case. Then, inviscid flows around an ONERA M6 wing, and viscous flows past a cylinder are simulated, for performance analysis. In these cases, the results are given and compared with experimental data and refer results, and the speed up of the GPU code are given and analysis below.

4.1 Validation of the Present Preconditioning Method

The performance of the preconditioning method is investigated using the low Mach number case of an inviscid flow around a NACA0012 airfoil with a C-mesh of 192x64 quadrilateral cells. The free-stream Mach number is 0.001 and the angle of attack (AoA) is zero. In this case, CFL=6 was selected. In Figure 4, the logarithm of the residual, that is, the L2 norm of the time derivative in the x-momentum equation, for both the non-preconditioning method and preconditioning method are plotted against the iteration. As we can see, the history of convergence for non-preconditioning method exhibits a plateau before machine precision is reached, and the history of convergence for preconditioned method can reach a lower level near the machine precision.

Fig. 4. The convergence histories.

Figures 5 and 6 show comparisons of the Mach number distributions and the pressure coefficient distributions between non-preconditioned and preconditioned converged solutions, respectively. In these two figures, the upper half regions are the contour lines without preconditioning, and the lower half regions are the contour lines with preconditioning. The converged solution with preconditioning shows correct contour lines while the converged solution without preconditioning shows clear nonphysical behavior.

Fig. 5. Comparison of the Mach number contour lines.

Fig. 6. Comparison of the pressure coefficient contour lines.

Figure 7 shows the pressure coefficient of the converged solutions on the surface of the airfoil for both preconditioned and non-preconditioned. A maximal pressure coefficient is expected at the leading edge stagnation point, and a smooth negative pressure coefficient is appeared on the most of the airfoil surface. The preconditioned solution exhibits both of these characteristics. But the non-preconditioned solution exhibits neither. The same flow of the NACA0012 airfoil with same flow conditions was also examined by Abbott and von Doenhoff¹¹, and the solution results obtained here agree well. Clearly, the preconditioned solution demonstrates better performance in this test case.

Fig. 7. Distribution of surface pressure coefficient.

Secondly a strong scaling timing test is carried out on a sequence of four successively refined quadrilateral grids. The simulations are done by GPU-accelerated code and original CPU code, both with improved preconditioning method and in single-precision, so that we can see the efficiency of the GPU computing. The statistics of unit running time, measured by dividing the time for a single iteration of the solver by the total number of grid elements, are presented in table 1. From the results we can see that GPU performs better when it comes to larger nElem (a variable the number of grid cells).

Table 1 Timing measurements of the preconditioning methods for NACA0012 airfoil

nElem	Running Time /sec.		SpeedUp
	(time per iteration per million cells)		
	CPU	GPU	
192x64	1.9828	0.1913	10.36
384x128	1.9674	0.1007	19.52
768x256	2.1826	0.0695	31.40
1536x512	2.1180	0.0601	35.24

4.2 Inviscid Flows past ANERA M6 Wing

To assess the accuracy and efficiency of the GPU implementation of the preconditioning method, firstly an inviscid case of the ONERA M6 wing was simulated. The topology of the grid used is C-H with 4 computational blocks and 184320 hexahedron cells. The conditions of free stream flow is Mach=0.010 and AoA=3.06 degree. In this case, CFL=6 was selected.

The plot of the logarithm of the residual (Fig. 8) shows how use of preconditioning obtains converged results in a reasonable number of time iteration steps. Same as shown in the preceding case, the aerodynamic solution is also affected by preconditioning. Figure 9 shows the distribution of the Mach number isolines on the surface and the root profile of M6 wing. And figure 10 shows the distribution of the pressure coefficient

isolines on the top surface of M6 wing. Both the results given in figure 9 and 10 are preconditioned. And they show very consistent with the results from reference paper [1].

Fig. 8. The convergence histories.

Fig. 9. The contour lines of Mach number on the surface and the root profile.

Fig. 10. The Mach number contour lines on the top surface.

In this 3-D case, the timing test was also carried out. The detailed timing measurements are presented in Table 2, showing the statistics of unit running time. Note that the reason we do not present the data for denser grid is that it would exceed the GTX Titan GPU card memory. From the results we can see that the GTX Titan GPU can perform 20 times speedup to a single-CPU implementation in this test case.

Table 2 Timing measurements of the preconditioning methods for a low speed flow past OREAN M6 wing

nElem	Running Time /sec.		SpeedUp
	(time per iteration per million cells)		
	CPU	GPU	
184320	2.9155	0.1136	25.66
786432	2.6797	0.0989	27.07
1572864	2.7116	0.0969	27.98
6291456	2.8448	0.0956	29.76

4.3 Viscous Flows around Cylinder

An unsteady viscous flow case past a 2-D cylinder was also simulated for assessing the accuracy and efficiency of the GPU implementation of the preconditioning method. The grid used is block-structured grid with 87400 quadrilateral cells. In this case, the freestream condition of Mach=0.002 and Re=200 was used, and the CFL number is same to the prior case. The flow shows unsteady characteristic when the Reynold number grows more than 47. Thus this test case is an unsteady flow, and the dual time-stepping method is used for this case.

Figure 11 shows the contours of the Mach number and the vorticity. Clearly shown in Fig. 11, on both up and down sides of the wake zone behind cylinder, a pair of vortices is shedding alternately forming the Karmen vortex street. After calculation, we can get the lift coefficient $C_L=\pm 0.63$, the drag coefficient $C_D=1.28\pm 0.05$ and the reduced frequency $Sr=0.198$ (defined as $Sr=nd/U_\infty$, n denotes the oscilation frequency of the lift coefficient, U_∞ denotes the velocity of the freestream). The results correspond to the experiment data and calculation results in the references [12] and [13].

Fig. 11. The distribution of Mach number and vorticity.

The running time was also tested in this case. From the previous timing tests, we can predict that the simulation carried out on the GPU platform may performance much faster than a single CPU realization. With the implementation of the GPU-accelerated code of this case, we got 0.0968 seconds per step per million grid cells, that performance 27 times faster than the implementation if the original CPU code (2.648 seconds per step per million grid cells). The performance is a little weaker than the previous steady cases, which is because the unsteady flux item contains more data exchange than calculation and reduces the efficiency of the whole solver.

5. Conclusions

A GPU-based parallel solver has been developed based on the CUDA architecture and the improved preconditioning method for the solution of low-speed flows on variety of grids. A remarkable design feature in the present scheme is that it uses the fluctuation value of the flow variables for calculation, renders an efficient approach to reducing the truncation error of single-precision, and makes more fully using the GPU's computing capability. In the numerical solution of these low-speed flows, the GPU solver has performed dozens of speedup compared with the original CPU solver (about 35 times upmost for present test case). In addition, we observed that the performance of speedup becomes better with increasing the size of corresponding computational problems, which suggest that the GPU solve can be well suited to the complex engineering problems.

Acknowledgement

The authors would like to acknowledge the support of the National Science Foundation of China (11172134) and the Funding of Jiangsu Innovation Program for Graduate Education (CXLX13_132).

References

1. Vittorio Puoti, "Preconditioning Method for Low-Speed Flows," AIAA Journal, Vol. 41, No. 5, 2003, pp. 817-830.
2. Weiss, J. M., and Smith, W. A., "Preconditioning Applied to Variable and Constant Density Flows," AIAA Journal, Vol. 33, No. 11, 1995, pp.2050-2057.
3. Brandvik T, Pullan, "Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware," AIAA 2008-607, 2008.
4. Erich Elsen, Patrick LeGresley, Eric Darve, "Large Calculation of the Flow Over a Hypersonic Vehicle Using a GPU," Journal of Computational Physics, 2008,227 (24):10148-10161.
5. Molemarker, J., Cohen, J. M., Patel, S. and Noh, J, "Low Viscosity Flow Simulations for Animation," Eurographics/ACM SIGGRAPH Symposium on Computer Animation, Eurographics Association, Aire-la-Ville, Switzerland. 2008.
6. A. Klockner, T. Warburton et al, "Nodal Discontinuous Galerkin Methods on Graphics Processors," Journal of Computational Physics, 2009,228 (21):7863-7882.
7. NVIDIA, "NVIDIA CUDA C Programming Guide, Version 6.5," 2014.
8. NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210," 2014.
9. J. Blazek, "Computational Fluid Dynamics: Principles and Applications," 2001.

10. The Portland Group, "CUDA Fortran Programming Guide and Reference, Release 2011," 2011.
11. Abbott I H, von Doenhoff A. E., "Theory of wing sections," New York: Dover Publication, 1959: 311-381.
12. Belov A, Martinelli L, Jameson A, "A New Implicit Algorithm With Multigrid for Unsteady Incompressible Flow Calculations," AIAA 95-0049, 1995.
13. Rogers SE, Kwak D, "Upwind Differencing Scheme for the Time-Accurate Incompressible Navier-Stokes Equations," AIAA Journal, Vol. 28, No. 2, 1990, pp. 253-262.