Compute Unified Device Architecture Implementation of Euler/Navier-Stokes Solver on Graphics Processing Unit Desktop Platform for 2-D Compressible Flows

Zhang Jiale, *Chen Hongquan* *

College of Aerospace Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, P. R. China

(Received 16 December 2015; revised 30 March 2016; accepted 28 April 2016)

Abstract: Personal desktop platform with teraflops peak performance of thousands of cores is realized at the price of conventional workstations using the programmable graphics processing units (GPUs). A GPU-based parallel Euler/Navier-Stokes solver is developed for 2-D compressible flows by using NVIDIA's Compute Unified Device Architecture (CUDA) programming model in CUDA Fortran programming language. The techniques of implementation of CUDA kernels, double-layered thread hierarchy and variety memory hierarchy are presented to form the GPU-based algorithm of Euler/Navier-Stokes equations. The resulting parallel solver is validated by a set of typical test flow cases. The 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 a GPU desktop can serve as a cost-effective parallel computing platform to accelerate computational fluid dynamics(CFD) simulations substantially.

Key words: graphics processing unit (GPU); GPU parallel computing; compute unified device architecture(CUDA) Fortran; finite volume method(FVM); acceleration

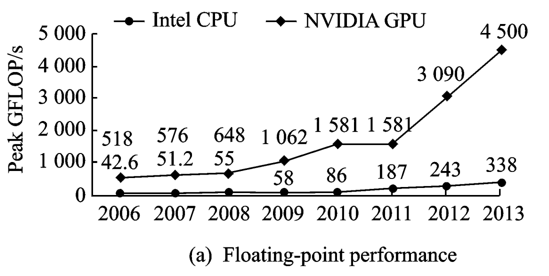
CLC number: V211.3 **Document code**: A **Article ID**: 1005-1120(2016)05-0536-10

0 Introduction

Developing computing codes with efficient parallelization is greatly demanded for the real applications occurred in computational fluid dynamics (CFD) like simulations of flows over aerodynamic bodies of considering complex three-dimensional configurations. The efficiency of traditional CFD solvers based on CPU architecture meets the bottleneck because limited by the clock frequency and data transmission bandwidth of CPUs[1]. In recent years, a new streaming processors, often called graphics processing unit (GPU), has become available for compute-intensive parallel tasks[2]. The memory bandwidth and floating-point performances of modern GPUs are orders of magnitude faster than a standard CPU. And the growing gap in peak performance, measured in floating point operations per second (FLOPS) between GPU and CPU over the last decade, is illustrated by Fig. 1[2]. Currently, NVIDIA GPUs outperform Intel CPUs on floating point performance and memory bandwidth, both by a factor of roughly ten[2].

At the beginning, the GPU's old fixed-function pipeline did not allow complex operations, thus using GPUs for general-purpose computation was a complicated exercise[3]. As the formation and development of new efficient programming model, the genera-purpose computation programmers can perform the parallel computation on GPU, and transfer date between CPU and GPU more convenient and effective than before. NVIDIA's compute unified device architecture

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



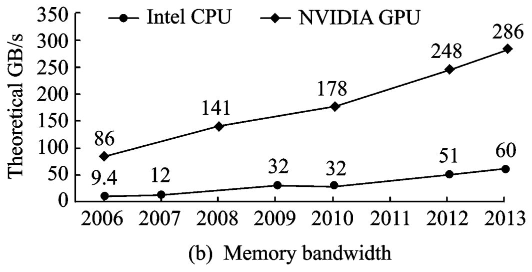


Fig. 1 Floating-point performance and memory bandwidth for Intel CPU and NVIDIA GPU [2]

(CUDA)[2] is one such model that supports native high-level programming language on its own line of GPUs. The programmers can use CUDA C/C++, CUDA Fortran or other programming languages in the GPU codes.

High performance parallel computing with CUDA has already attracted the CFD researchers. Brandvik and Pullan[4] completed porting a two- and three-dimensional Euler code for modeling inviscid flow onto GPU in 2008. 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. [5] rewrote part of the Navier-Stokes Stanford University solver (NSSUS) to model hypersonic flow on GPU. The performance of their code ranges from 15 to 40 times speedup compared with the original solver. Molemaker, et al. [6] developed a multi-grid method to solve the pressure Poisson equation. The CUDA implementation of the multi-grid pressure Poisson solver produced a speedup of 55 times relative to a 2. 2 MHz AMD Opteron processor[6]. In addition to the finite volume method, the discontinuous Galerkin (DG) method has also been implemented on the GPU by Klockner, et al. [7] in 2009. Zhang and Han[8] developed an implicit data-parallel scheme to solve computational fluid dynamics problems on GPU platform in 2010, and obtained 28 times speedup for an implicit calculation of ONERA M6 wing. In 2011, Liu, et al. [9] acceletated the simulation of large sparse linear and nonlinear equations on multiple pieces of GPU. Gu and Xu[10] achieved a real-time liquid payload hydrodynamics simulation method on GPU using smoothed particle hydrodynamics (SPH) method in 2013, and obtained a performance of 12 times speedup of a aerial CPU implementation. All the past and recent works in writing CFD solvers on GPU have shown encouraging results.

In the paper, we focus on the developing of personal desktop platform with the programmable graphics processing units of thousands of cores. The techniques of implementation of CUDA kernels, double-layered thread hierarchy and variety memory hierarchy are discussed to develop a GPU-based parallel Euler/Navier-Stokes solver in CUDA Fortran programming language. A set of typical test flow cases are selected for both validation and performance analysis. Dozens of times speedup (about 38 times upmost for present test cases) relative to a serial CPU implementation can be achieved using GPU desktop platform developed for the present solver.

1 Governing Equations and Numerical Approach

1.1 Governing equations

The 2-D Navier-Stokes equations governing compressible fluid flows can be expressed in conservative form[11] as

$$\frac{\partial \boldsymbol{W}}{\partial t}+\frac{\partial(\boldsymbol{F}_c-\boldsymbol{F}_v)}{\partial x}+\frac{\partial(\boldsymbol{G}_c-\boldsymbol{G}_v)}{\partial y}=0 \qquad (1)$$

where $\boldsymbol{W}$ is the vector of conservative variables. $\boldsymbol{F}_c$ and $\boldsymbol{G}_c$ are the convective flux terms; $\boldsymbol{F}_v$ and $\boldsymbol{G}_v$ the viscous flux terms. They are defined as

$$\boldsymbol{W}=\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho E\end{bmatrix},\boldsymbol{F}_c=\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ \rho uH\end{bmatrix},\boldsymbol{G}_c=\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ \rho vH\end{bmatrix} \qquad (2)$$

$$\boldsymbol{F}_v=\begin{bmatrix}0\\ \tau_{xx}\\ \tau_{xy}\\ u\tau_{xx}+v\tau_{xy}+k\dfrac{\partial T}{\partial x}\end{bmatrix},\boldsymbol{G}_v=\begin{bmatrix}0\\ \tau_{xy}\\ \tau_{yy}\\ u\tau_{xy}+v\tau_{yy}+k\dfrac{\partial T}{\partial y}\end{bmatrix} \qquad (3)$$

where ρ, p, T, E and H denote the density, pressure, temperature, total energy per unit mass, and total enthalpy per unit mass, respectively. u and v are the cartesian components of the velocity vector. For a perfect gas, these quantities satisfy

$$p=(\gamma-1)\left(\rho E-\frac{1}{2}\rho(u^2+v^2)\right) \tag{4}$$

and

$$p=\rho RT \tag{5}$$

The components of the viscous stress tensor τ_{ij} is defined as

$$\tau_{ij}=\mu\left(\frac{\partial v_i}{\partial x_j}+\frac{\partial v_j}{\partial x_i}\right)-\frac{2}{3}\mu\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)\delta_{ij} \tag{6}$$

In these equations, μ and k denote the dynamic viscosity coefficient and thermal conductivity coefficient, respectively. For perfect gas, the dynamic viscosity coefficient μ can be calculated by the Sutherland formula[11]. The thermal conductivity coefficient k has relationship to μ as

$$k=\frac{\gamma}{\gamma-1}R\frac{\mu}{Pr} \tag{7}$$

For perfect gas, the ratio of specific heats of fluid $\gamma=1.4$, and the Prandtl number $Pr=0.72$.

1.2 Numerical approach

Once a mathematical model is defined, we need a method for approximating the differential equations by obtaining a system of equations at a set of discrete elements in space and time. In our application of fluid dynamics, the finite volume method (FVM) in cell-centered scheme is adopted. By divided the flow field into a set of discrete control volumes, the integral form of the equation can be applied over each control volume. For control volume i, the discretization equations is as follows

$$\frac{\Delta \boldsymbol{W}_i}{\Delta t}=-\frac{1}{\Omega_i}(R_{c,i}-R_{v,i}) \tag{8}$$

$$R_{c,i}=\sum_{j=1}^{n}\boldsymbol{F}_{c,ij}\Delta S_{\mathrm{ij}} \quad R_{v,i}=\sum_{j=1}^{n}\boldsymbol{F}_{v,ij}\Delta S_{\mathrm{ij}} \tag{9}$$

where the convective flux and viscous flux are defined as

$$\boldsymbol{F}_c=\boldsymbol{F}_c n_x+\boldsymbol{G}_c n_y, \quad \boldsymbol{F}_v=\boldsymbol{F}_v n_x+\boldsymbol{G}_v n_y \tag{10}$$

where n and ΔS_{ij} denote the edge number and the area of edge j of the control volume i, respectively.

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

$$R_{c,i}=\sum_{j=1}^{n}\boldsymbol{F}_c\left(\frac{\boldsymbol{W}_L+\boldsymbol{W}_R}{2}\right)\Delta S_{ij}-\boldsymbol{D}_i \tag{11}$$

where the construction of the artificial dissipative term is given by

$$\boldsymbol{D}_i=\sum_{j=1}^{n}\lambda_{ij}\varepsilon_{ij}^{(2)}(\boldsymbol{W}_j-\boldsymbol{W}_i)-\sum_{j=1}^{n}\lambda_{ij}\varepsilon_{ij}^{(4)}(\nabla^2\boldsymbol{W}_j-\nabla^2\boldsymbol{W}_i) \tag{12}$$

where $\varepsilon^{(2)}$ and $\varepsilon^{(4)}$ are adaptive coefficients. $\lambda=|U|+c$ is spectral radius of the Jacobian matrix, in which $U=un_x+vn_y$ is the normal velocity of the edge, and $c=\sqrt{\gamma p/\rho}$ the speed of sound. The detailed description of the construction of the dissipative term can be found in Ref. [11].

The viscous flux term is evaluated by central scheme, as

$$\boldsymbol{R}_{v,i}=\sum_{j=1}^{n}\boldsymbol{F}_v\left(\frac{\boldsymbol{W}_L+\boldsymbol{W}_R}{2}\right)\Delta S_{ij} \tag{13}$$

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

$$\begin{cases}\boldsymbol{W}_i^{(0)}=\boldsymbol{W}_i^n \\ \boldsymbol{W}_i^{(m)}=\boldsymbol{W}_i^{(0)}-\alpha_m\dfrac{\Delta t_i}{\Omega_i}R^{(m-1)} \quad m=1,2,3,4 \\ \boldsymbol{W}_i^{n+1}=\boldsymbol{W}_i^{(4)}\end{cases} \tag{14}$$

where the superscript n and $n+1$ denote the current and the next time level. The coefficient α_m can be found in Ref. [11]. To accelerate the convergence, Δt is taken as the local time step[11].

In the case of inviscid flow governed by the Euler equations, the fluid slips over the solid wall and there is no flow normal to the surface, that is $\boldsymbol{V}\cdot\boldsymbol{n}=0$. And in the case 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. [11].

2 Programming Model and GPU Implementation

NVIDIA's CUDA[2] is a parallel computing platform and programming model that leverages the powerful compute engine of their GPUs. CUDA provides the developers with a software environment, an extension to C, Fortran and some other programming languages, to launch and manage massively parallel computations on GPUs. The reader can refer to the CUDA programming guide for more details[2]. In the section, we summarize the GPU hardware architecture and the programming model with CUDA Fortran. Hereinafter, we use the term "host" to refer to CPU and the term "device" to refer to GPU.

2.1 GPU architecture

GPU is specialized for compute-intensive, highly parallel computation, and designed such that more transistors are devoted to data processing rather than data caching and flow control in CPU, as schematically illustrated by Fig. 2.

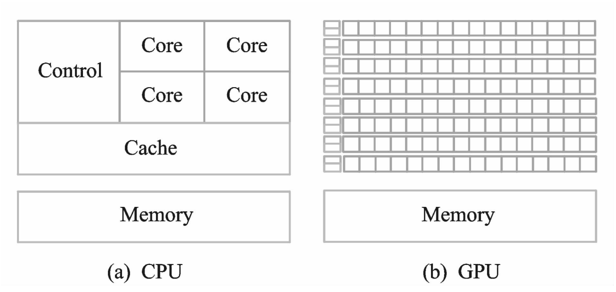


Fig. 2 Different architecture of CPU and GPU

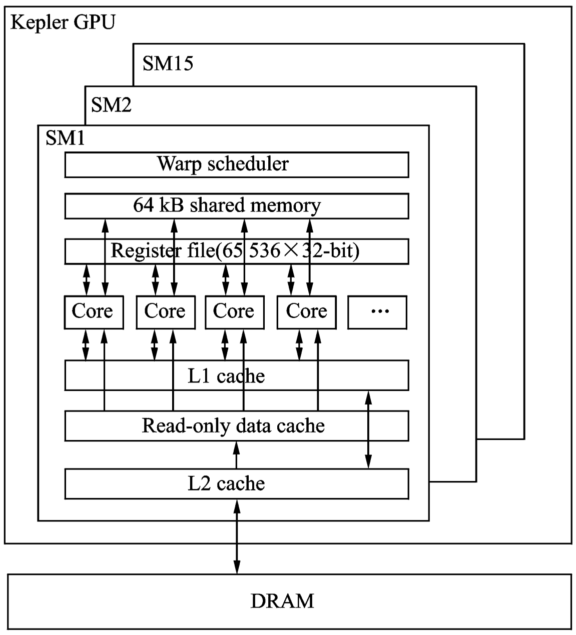
GPU is a set of stream multiprocessors, which is an extension of single instruction multiple data(SIMD) paradigm architecture. This design architecture, as the name suggests, makes CUDA optimal for performing a single instruction in parallel on different sets of data on NVIDIA GPUs. A full Kepler GK110 (the latest generation GPU architecture) is composed of fifteen streaming multiprocessors (SM) and six 64-bit memory controllers. As shown in Fig. 3(a), each SM 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 SM. The global memory is used to store the main computing data. And different kinds of cache, including 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, number of streaming multiprocessors, 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.

2.2 Programming model

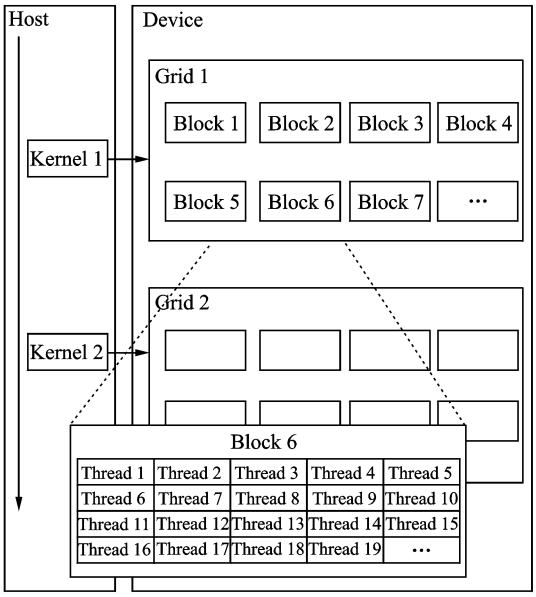
In the CUDA programming model, the parallel code executed on GPU is called the kernel. Before executing a kernel, the processing data must be transferred to memory on GPU. Next, CPU initiates kernel execution on GPU. After the kernel is completed executed, CPU retrieves the processed data from GPU. This process is illustrated in Fig. 4.

The kernel is launched from the host side (CPU), and it is organized to a thread grid, as illustrated in Fig. 3 (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 GPU. The reader can refer to Ref. [12] for more details. In CUDA Fortran API, the qualifiers *attributes(host)*, *attributes(global)* and *at-*



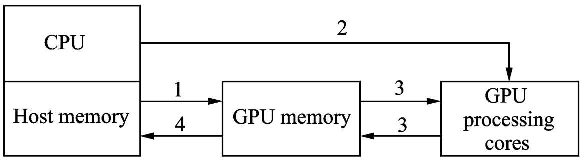
(a) Kepler GPU hardware architecture



(b) CUDA thread organization

Fig. 3 Kepler GPU hardware architecture and CUDA thread organization

tributes (*device*) specify whether CPU or 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 used to allocate memory on GPU, copy data between CPU memory and device memory, and free memory on GPU, respectively.



1: Copy data from host(CPU) to device(GPU)

2: CPU initiated kernel execution on GPU

3: When kernel ID executing, data read/write in GPU memory

4: After executed, data copy bake to CPU memory

Fig. 4 CUDA process flow

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*$<<<$*gridSize*, *blockSize*$>>>$( ). 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).

2.3 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 is idle. There are only a few thread synchronization must occurred, such as after updating the flow variables in grid elements, or when judging the convergence. In Additional, the algorithm is 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 Fig. 5. Each thread corresponds to a grid element, and takes charge of the computation of the specified element. Every n (thread number per block) counts of threads are combined into a thread block, and the size of thread grid is defined to $m=(N+n-1)/n$. Thus, all the thread counts is $m*n$, a little larger than the element count nN

(number of elements), and the threads can cover all the element that needing be solved.

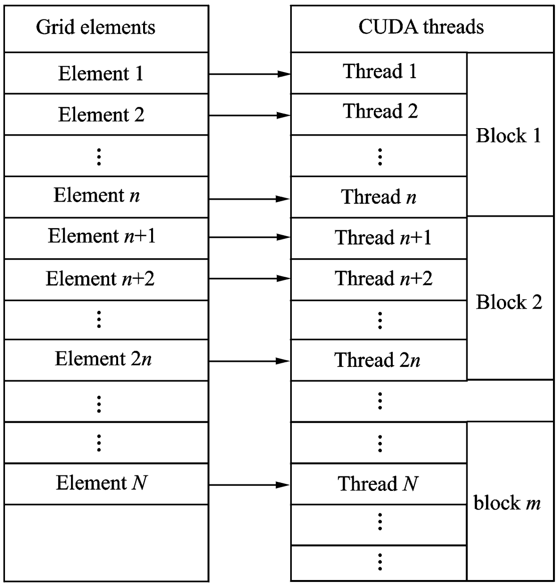


Fig. 5 1-D permutation and correspondence between grid elements and GPU threads

Listing 1 shows the simplified host side code of the time iteration procedure. In the code snippet, two nested loops are composed. The outer loop is used for advancing the solution in time, and the inner loop is used for four-stage Runge-Kutta iteration. As shown in Listing 1, almost all of the computing tasks are composed into several kernel functions that could run on the device (GPU), and the host (CPU) is just used to organize and control the computational flow. The memory allocation of the arrays on the device is only one before starting the time stepping.

Listing 2 shows the device code that compute the primitive flow variables by the conserved flow variables. In the beginning, the qualifier ″attributes(global)″ is used to define kernel_primvar as a kernel subroutine. The qualifier ″device″ in variable define line is used to express the val_consvar and val_primvar are GPU arrays, and the qualifier ″value″ is used to express the constant val_nElem is a variable that define in host and used in kernel. The code in line 10 is used to obtain the thread ID, and the ″if″ structure in line 12 is used to limit the thread ID in case computing beyond the upper bound.

```
OUTER LOOP: do iIter = 1,nIterNeed
  call kernel_primvar<<<m,n>>>()
  call kernel_deltaTime<<<m,n>>>()

  Inner loop: do j = 1,4
    call kernel_artificalDissipation<<<m,n>>>()

    call kernel_calResidual_conv<<<m,n>>>()
    call kernel_calResidual_visc<<<m,n>>>()
    call kernel_residualSmooth<<<m,n>>>()
   call kernel_updConsvar<<<m,n>>>()
   istat = cudaDeviceSynchronize()
 enddo

 call kernel_convergenceJudgement()
enddo
```

Listing 1 Simplified host side code of time iteration procedure

```
1  attributes(global) subroutine kernel_primvar(consvar,primvar, nElem)
2    real,device:: consvar(4,nElem)
3    real,device:: primvar(4,nElem)
4    integer,value:: nElem
5
6    integer:: i
7    real:: sq_vel
8
9    ! get the thread ID
10   i = (blockIdx%x-1) * blockDim%x+threadIdx%x
11
12   if(i <= val_nElem) then
13     primvar(1,i) = consvar(1,i)
14     primvar(2,i) = consvar(2,i)/ consvar(1,i)
15    primvar(3,i) = consvar(3,i)/ consvar(1,i)
16    sq_vel = primvar(2,i) * *2 + primvar(3,i) * *2
17    primvar(4,i) = (gamma-1.0) * ( consvar(4,i) - 0.5 * primvar(1,i) * sq_vel)
18  endif
19 endsubroutine
```

Listing 2 Simplified device side code for computing primitive variables from conservative variables

3 Results and Discussions

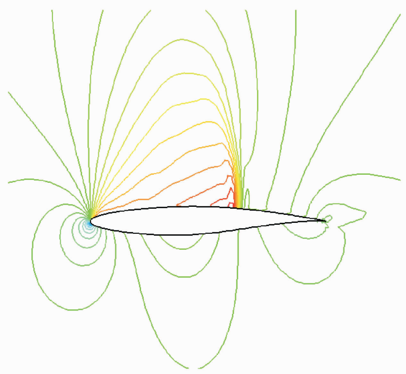
Based on the above method and the hardware architecture, we develop both CPU serial code and GPU parallel code for solving the 2-D Euler/

Navier-Stokes equations. All the codes are 32 bit floating-point precision since GPU we used operates in single precision. In all our cases, CPU used is a single core of an Intel Core i5-3450 (3.1 GHz, 6 MB L3 cache, can turbo to 3.5 GHz) and GPU used is an NVIDIA GTX Titan (2688 scalar processor cores at 837 MHz, 6 GB Memory). Both CPU and GPU can represent their current advanced level of computing.

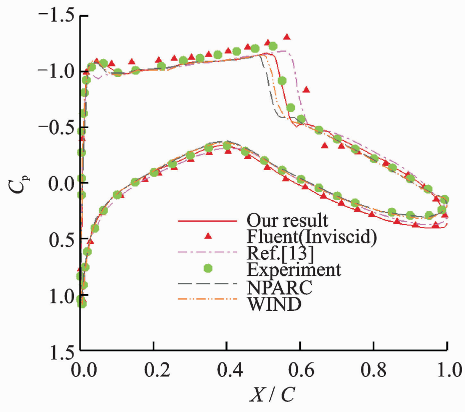
Firstly, the GPU solver is tested on a transonic inviscid flow around the RAE2822 airfoil, and a subsonic viscous flow around the NACA0012 airfoil. The results of these two-dimensional cases are given below and compared with experimental data and reference results. Then the resulting GPU solver is tested with a set of cases for the steady-state flow around the NACA0012 airfoil on a series of structure grids. The results and the performance are given and analyzed below.

3.1 Validation of GPU implementation of Euler/Navier-Stokes solver

Firstly, the GPU solver is tested with a case of a transonic inviscid flow around the RAE2822 airfoil. This case is run with four stages Runge-Kutta iteration, with a CFL number of 5 and implicit residual smoothing method. We assume the free stream flow conditions with a Mach number of 0.729 and a 2.31°angle of attack in this case. Fig. 6(a) shows the contours of Mach number near the airfoil, and Fig. 6(b) presents the comparison of pressure coefficient data among the experiment data, the reference data and our result. The reference data NPARC and WIND are provided by NASA. Our result produces reasonably accurate results on the leading edge of the airfoil and a very crisp, clean shock. The position of shock wave on the upper surface of the airfoil of our result is a little later than the reference results, but closer to the experimental data. The pressure coefficient on the trailing edge of the airfoil of our result is a little lower than the experimental data and viscous reference results, but closer to the inviscid results from Fluent and Ref.[13]. These behaviors are fairly typical of transonic Euler CFD solutions.



(a) Contours of Mach number

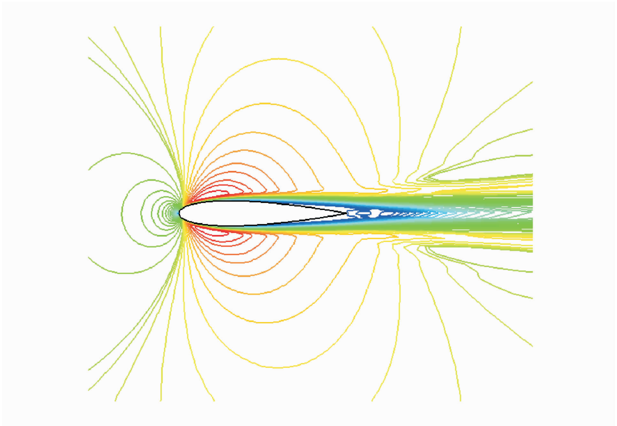


(b) Pressure coefficient distribution

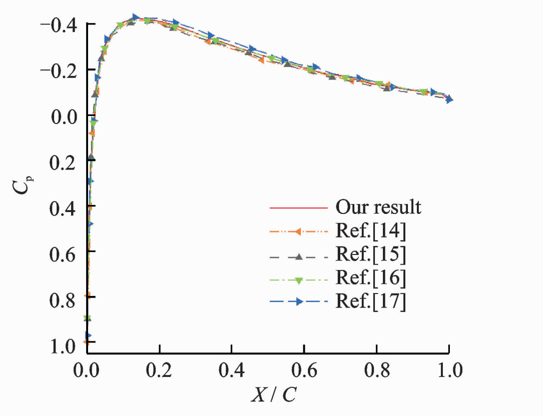
Fig. 6 Euler results for RAE2822 airfoil at $Ma_\infty = 0.729$ and 2.31° of AoA

Then, the GPU solver is tested with a case of a subsonic viscous flow around the NACA0012 airfoil. The case is running with same temporal discretization scheme and same CFL number of the front case. We assume free stream flow conditions with a Mach number of 0.5, a zero angle of attack and Reynolds number at 5 000 in this case. Fig. 7(a) shows the contours of Mach number near the airfoil, and Fig. 7(b) presents the comparison of pressure coefficient data between the reference data[14-17] and our result. As we can see in Figs. 7(a, b), our GPU solver produces reasonably accurate results in this case of a viscous flow.

The results of our GPU solver for both inviscid and viscous flows seem fairly reasonable with experimental data and refer results.



(a) Contours of Mach number



(b) Pressure coefficient distribution

Fig. 7 Laminar results for NACA0012 airfoil at $Ma_{\infty}=0.5$, zero of AoA and $Re_c=5\ 000$

3.2 Performance results and evaluation

The following computing hardware is utilized in the paper. A dual-CPU/single-GPU platform is built with an Intel Core i5-3450 Quad 3.1 GHz CPU, 16 GB of memory and a NVIDIA GTX Titan board. The Titan board provides 2 688 streaming processor cores and 6 GB of global device memory. And the Titan GPU used in the paper can deliver a theoretical peak performance of 4 TFLOPS (Trillion floating-point operations per second).

To assess the performance of the Euler/Navier-Stokes solver, we use both the inviscid and viscous conditions to solve for the steady-state flows around the NACA0012 airfoil on varying sizes of structure grids. The grid sizes selected for comparison are 192×64, 384×128, 768×256, and 1 536×512. The performance is measured by dividing the total number of grid elements by the time for a single iteration of the solver, yielding the throughput in millions of cell-step per second.

Using a single CPU core, the performance of our CFD code is approximately 0.6 million cell-steps per second for Euler solver and 0.45 million cell-steps per second for NS solver, shown in Fig. 8 and the third column of Table 1. We can find that the CPU performance is almost no growth with the increase of the gird size. That is because the smallest size of grid have already made the full use of CPU. On the other hand, the performance of our GPU implementation is more efficient, as presented in Fig. 8 and the fourth column in Table 1. And the computational speed-up is shown in the fifth column. We can find that present GPU performs much faster than the implementation of CPU, at a radio of tens.

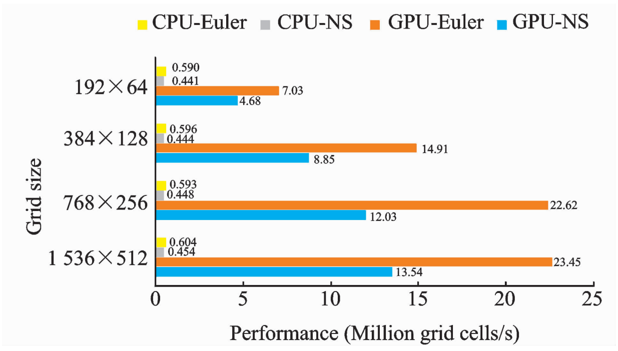


Fig. 8 Performance of single CPU and GPU on NACA0012 case with varying grid size

Table 1 Performance of CPU and GPU with different size of grids

Model	Grid size	Performance (Million cell-steps/s)		Speedup GPU/CPU
		ModelIntel i5-3450 3.1 GHz	NVIDIA GTX Titan	
Euler	192×64	0.591	7.03	11.9
	384×128	0.596	14.91	25.0
	768×256	0.593	22.62	38.1
	1 536×512	0.604	23.45	38.8
N-S	192×64	0.441	4.68	10.6
	384×128	0.444	8.85	19.9
	768×256	0.448	12.03	26.9
	1 536×512	0.454	13.54	29.9

As shown in Table 1, the speedup of GPU is growing with the increase of the grid size. On the smallest size of grid, the level of parallelism does not fully utilize the massively parallel GPU architecture, and performance is only 11.9 for inviscid flow case and 10.4 for laminar flow case above that of CPU (Table 1). As the mesh size increases, GPU becomes more efficient and perform much faster. The speedup grows up to 25.0, 38.1 and 38.8 times in the inviscid flow cases and 19.9, 26.9 and 29.9 times in viscous flow cases on the increasing grid size of 384×128, 768×256 and $1\ 536\times512$, respectively. The speedup numbers are impressive for large problem size, because the arithmetic intensity on the GPU increases with problem size.

4 Conclusions

The implementation of Euler/Navier-Stokes equations for 2-D compressible flows on personal desktop platform with a GPU is presented. With CUDA Fortran programming language, NVIDIA's CUDA programming model is used to implement the discretized form of the governing equations. In the numerical solution of 2-D compressible flows, the GPU code has perform dozens of speedup compared with the serial CPU code (about 38 times upmost for present test cases). In addition, we observe that the performance of speedup becomes better with increasing the size of corresponding computational problems, which suggests that the GPU code can be well suited to the complex engineering problems.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (No. 11172134); the Funding of Jiangsu Innovation Program for Graduate Education (No. CXLX13_132).

References:

[1] PATTERSON D A, HENNESSY J L. Computer architecture: A quantitative approach[M]. 4th ed. [S. l.]: Morgan Kaufmann Publishers Inc, 2007.

[2] NVIDIA. NVIDIA CUDA C programming guide (version 7.0) [EB/OL]. (2015-03-05) [2015-09-01]. http://docs. nvidia. com/cuda/cuda-c-programming-guide/index. html.

[3] OWENS J D, HOUSTON M, LUEBKE D, et al. GPU computing[J]. Proceedings of the IEEE, 2008, 96(5): 879-899.

[4] BRANDVIK T, PULLAN G. Acceleration of a 3D Euler solver using commodity graphics hardware: AIAA 2008-607[R]. 2008.

[5] ELSEN E, LEGRESLEY P, DARVE E. Large calculation of the flow over a hypersonic vehicle using a GPU[J]. Journal of Computational Physics, 2008, 227(24): 10148-10161.

[6] MOLEMARKER J, COHEN J M, PATEL S, et al. Low viscosity flow simulations for animation [C]// Eurographics/ACM SIGGRAPH Symposium on Computer Animation. Dublin, Ireland: SCA, 2008: 115-116.

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

[8] ZHANG B, HAN J. Parallel computing methods for CFD using a GPU and implicit scheme [J]. Acta Aeronautica et Astronautica Sinica, 2010, 21(2): 249-256.

[9] LIU S, ZHONG C, CHEN X. Solvers for systems of large sparse linear and nonlinear equations based on multi-GPUs[J]. Transactions of Nanjing University of Aeronautics and Astronautics, 2011, 28(3): 300-308.

[10] GU A, XU J. Liquid payload hydrodynamics simulation method for real-time aircraft simulation system [J]. Journal of Nanjing University of Aeronautics and Astronautics, 2013, 45(4): 491-496. (in Chinese)

[11] BLAZEK J. Computational fluid dynamics: Principles and applications[M]. 2nd ed. [S. l.]: Elsevier, 2001.

[12] The Portland Group. CUDA Fortran programming guide and reference, Release 2014 [EB/OL]. (2014-01-01) [2015-09-01]. http://www. pgroup. com/doc/pgicudaforug. pdf.

[13] AHMED F, AHMED F, AFFAN M, et al. Numerical solution of 2D Euler equations for the transonic flow past over NACA0012 and RAE2822 airfoils using high order accurate Runge-Kutta discontinuous Galerkin method[C]// Proceeding of 2014 11th International Bhurban Conference on Applied Sciences & Technology (IBCAST). [S. l.]: IBCAST, 2014: 218-213.

[14] LARA P R M, MORGAN K. A review and comparative study of upwind based schemes for compressible flow computation. Part III: Multidimensional extension on unstructured grids[J]. Archives of Computational Methods in Engineering, 2002, 9(3): 207-256.

[15] CARAENI D, FUCHS L. Compact third-order multidimensional upwind scheme for Navier-Stokes simulations[J]. Theoretical and Computational Fluid Dynamics, 2002, 15(6): 373-401.

[16] CHASSAING J C, KHELLADI S, NOGUEIRA X. Accuracy assessment of a high-order moving least squares finite volume method for compressible flows [J]. Computers & Fluid, 2013, 71(2): 41-53.

[17] LIU Xueqiang. The research of N-S equations' solution using hybrid grids and multi-grid methods and its applications [D]. Nanjing: Nanjing University of Aeronautics and Astronautics, 2001. (in Chinese)

Ms. **Zhang Jiale** received B. S. degree in flight vehicle design and engineering from Nanjing University of Aeronautics and Astronautics (NUAA) in 2009, and M. S. degree in aerodynamics also from NUAA in 2012, respectively. From 2012 to present, he is a Ph. D. candidate in College of Aerospace Engineering, Nanjing University of Aeronautics and Astronautics. His research is focused on computational fluid dynamics and high performance computing.

Prof. **Chen Hongquan** received B. S. and Ph. D. degrees in aerodynamics from Nanjing University of Aeronautics and Astronautics (NUAA) in 1984 and 1990, respectively. From 1993 to 1994, he was a Postdoctoral Research Fellow in University of Paris VI and Dassault Aircraft Company, France. From 1990 to present, he has been a full professor in College of Aerospace Engineering, Nanjing University of Aeronautics and Astronautics. His research is focused on aerodynamics, which includes computational fluid dynamics, computational electromagnetic, genetic algorithm and multidiscipline design optimization (MDO).

(Executive Editor: Xu Chengting)