

A CUDA Pseudo-spectral Solver for Two-dimensional Navier-Stokes Equation*

Kaiyuan LOU, Zhaohua YIN

Institute of mechanics, Chinese Academy of Sciences, Beijing, China, 100190

Email: zhaohua.yin@imech.ac.cn Tel: 010-82544100

ABSTRACT

In this paper, the two dimension incompressible Navier-Stokes equations with pseudo-spectral method are solved using the related subroutines in FFTW and CUFFT. Compared with the codes on CPU, the performance of the codes on GPU is much better, especially when the resolution increases. For the resolution of 2048×2048 , the acceleration reaches 14.45 times. We also try to combine MPI (Message Passing Interface) and CUDA (Compute Unified Device Architecture) in our solver. Due to the inevitable frequent data transfers between Host and Device, the speedup is not so ideal compared with that of the single node, and 1.82 times acceleration is obtained in double precision for the resolution of 4096×4096 .

Keywords: CUDA, spectral method, N-S equation.

1. INTRODUCTION

In recent years, the Graphics Processor Unit (GPU) has tremendously developed. Though the purpose of these advances is to calculate the complex visual effects in computer games, it has been found that the same technology can be applied in scientific computing. In 2006, NVIDIA developed a Compute Unified Device Architecture (CUDA) on the extended set of C language. CUDA is a very convenient architecture because programmers do not need to master the graphical knowledge. Hence, CUDA provides a low entry level for the learning of many-core programming, so the general-purpose scientific computing on the GPU develops rapidly.

Legyel first used GPU in scientific computing about the robot first [1], and it was then applied in various areas including fluid dynamics. In the past few years many researchers have studied how to use GPU to optimize the CFD (Computational Fluid Dynamics) codes, and found GPU can really improve the code's performance for one or two grades compared with that on CPU (Central Processing Unit). For example, Antoniou found that finite difference method WENO obtained 53 times acceleration for single-precision float when CUDA was adopted. Cohen and Molemaker found similar performance improvements in the solution of three-dimensional incompressible Navier-Stokes (N-S) equations (double precision) [2]. Dong Tingxing *et al.* simulated two-dimensional RAE2822 wing flow around in the scale of 1024×128 , and obtained 2.33 times acceleration [3]. When calculating two-dimensional diffusion equation in the size of 1024×1024 , Dong Tingxing *et al.* achieved a 34 times speedup [4].

According to the above information, CUDA is successfully applied in solving N-S equation with finite difference method, but few studies about spectral method have been done. For a

* This project is supported by the NSF of China (Contract No. G11172308).

comparable error on the uniform mesh, spectral method requires a much finer mesh than finite difference or finite element methods. Unlike finite difference methods, most computation of spectral method is in DFT (Discrete Fourier Transform). FFTW is the widely adopted open-source DFT package in CPU, while CUFFT is mostly used on GPU [5]. In this paper, we use both FFTW and CUFFT to solve two-dimension incompressible N-S equations and focus on the performance improvement when replacing FFTW with CUFFT.

Sometimes, due to memory limitations, a large-scale problem cannot be computed in a single GPU, so we also use the multi-node parallel computing with MPI (Message Passing Interface). Some tests in combination of MPI and CUDA are also performed.

2. GOVERNING EQUATIONS & NUMERICAL METHODS

2.1. Governing Equations

Two dimension incompressible N-S equations:

$$\begin{cases} \nabla \cdot \vec{v} = 0, \\ \frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} = -\frac{1}{\rho} \nabla p + \nu \nabla^2 \vec{v} + \vec{f}. \end{cases} \quad (1)$$

Here, $\vec{v} = (u, v)$ is the velocity, \vec{f} the external force, ρ the density, p the pressure, and ν the kinetic viscosity coefficient. There is no external force in our problem, so $\vec{f} = 0$.

The computing domain is $(x, y) \in [0, 2\pi] \times [0, 2\pi]$, and the periodic boundary conditions are adopted. The initial condition is $\vec{v}(\mathbf{x}, 0) = \vec{v}_0(\mathbf{x})$.

2.2. Numerical Method

Spectral method is applied to discrete space. p, \vec{v} is mapped from physical space to Fourier space as the following:

$$\begin{cases} \vec{v}(\mathbf{x}, t) = \sum_{\mathbf{k}} \hat{\mathbf{V}}_{\mathbf{k}} e^{i\mathbf{k} \cdot \mathbf{x}}, \\ p(\mathbf{x}, t) = \sum_{\mathbf{k}} \hat{p}_{\mathbf{k}} e^{i\mathbf{k} \cdot \mathbf{x}}, \end{cases} \quad (2)$$

where $\mathbf{x} = (x, y)$ is the position in physical space, and

$\mathbf{k} = (k_1, k_2)$ represents different wave numbers.

Applying the Fourier-Galerkin method to the N-S equations Eq. (1) and with the approximation Eq. (2), we get a set of differential equations for determining the Fourier coefficients $\hat{\mathbf{V}}_{\mathbf{k}}$ and $\hat{p}_{\mathbf{k}}$:

$$dt \hat{\mathbf{V}}_{\mathbf{k}} + \nu k^2 \hat{\mathbf{V}}_{\mathbf{k}} = -\hat{\mathbf{A}}_{\mathbf{k}} + \frac{\mathbf{k} \cdot \hat{\mathbf{A}}_{\mathbf{k}}}{k^2} \mathbf{k}. \quad (3)$$

When $k^2 = 0$, no matter what the value of $\hat{\mathbf{V}}_0$ is, the original equation is automatically satisfied. We set $\hat{p}_0 = 0$ at that time, which means value of pressure is zero.

$$d\hat{\mathbf{V}}_{\mathbf{k}} = -\nu k^2 \hat{\mathbf{V}}_{\mathbf{k}} - \hat{\mathbf{A}}_{\mathbf{k}} + \frac{\mathbf{k} \cdot \hat{\mathbf{A}}_{\mathbf{k}}}{k^2} \mathbf{k}. \quad (4)$$

In Eq. (4), $-\nu k^2 \hat{\mathbf{V}}_{\mathbf{k}}$ is linear term (viscous term) and $-\hat{\mathbf{A}}_{\mathbf{k}} + \frac{\mathbf{k} \cdot \hat{\mathbf{A}}_{\mathbf{k}}}{k^2} \mathbf{k}$ nonlinear term (convection term).

To carry out the time integration, the third order Runge-Kutta method is adopted.

$$\frac{\partial \mathbf{v}}{\partial t} = L(\mathbf{v}) + N(\mathbf{v}). \quad (5)$$

Here, $L(\mathbf{v})$ is the linear term $-\nu k^2 \hat{\mathbf{V}}_{\mathbf{k}}$ and $N(\mathbf{v})$ the nonlinear term $-\hat{\mathbf{A}}_{\mathbf{k}} + \frac{\mathbf{k} \cdot \hat{\mathbf{A}}_{\mathbf{k}}}{k^2} \mathbf{k}$.

We define \mathbf{v}_n as the speed at time t and \mathbf{v}_{n+1} the speed at time $t + \Delta t$. The following equations show how the calculation of \mathbf{v}_{n+1} from \mathbf{v}_n is conducted in three steps [6]:

$$\begin{aligned} \mathbf{v}' &= \mathbf{v}_n + \Delta t[L(\alpha_1 \mathbf{v}_n + \beta_1 \mathbf{v}') + \gamma_1 N(\mathbf{v}_n)], \\ \mathbf{v}'' &= \mathbf{v}' + \Delta t[L(\alpha_2 \mathbf{v}' + \beta_2 \mathbf{v}'') + \gamma_2 N(\mathbf{v}') + \zeta_1 N(\mathbf{v}_n)], \\ \mathbf{v}_{n+1} &= \mathbf{v}'' + \Delta t[L(\alpha_3 \mathbf{v}'' + \beta_3 \mathbf{v}_{n+1}) + \gamma_3 N(\mathbf{v}'') + \zeta_2 N(\mathbf{v}')], \end{aligned} \quad (6)$$

where

$$\begin{aligned} \gamma_1 &= 8/15, \gamma_2 = 5/12, \gamma_3 = 3/4, \\ \zeta_1 &= -17/60, \zeta_2 = -5/12, \\ \alpha_1 &= 29/96, \alpha_2 = -3/40, \alpha_3 = 1/6, \\ \beta_1 &= 37/160, \beta_2 = 5/24, \beta_3 = 1/6. \end{aligned}$$

The linear term $L(\mathbf{v})$ can be easily computed in Fourier space, but the calculation of the nonlinear term $N(\mathbf{v})$ is a bit involved. So the nonlinear term is obtained by being transferred back and from the physical space with FFTs; in the meantime, the de-aliasing procedure has to be adopted to remove the aliasing errors. There are two kinds of de-aliasing techniques available: padding-truncation and phase-shifts. In this paper, we use phase-shifts, which reserves more high wave numbers information than that for padding-truncation. In summary, the calculation process is shown in Figure. 1:

3. INTRODUCTION TO GPU COMPUTING

3.1. The Structure of CUDA

The CUDA code is divided into two parts: one part is on the CPU, known as the Host section, and another part in the GPU, which is called the Device portion. Host part completes a call to the GPU through Kernel function. As a highly parallel programming model, CUDA divides the tasks in the Kernel into the threads. The structure of the threads is indicated in Figure 2. Threads are organized by blocks, and each block is the same in size. A kernel function can be performed by multiple blocks, and each block is organized as a

one-dimensional or two-dimensional grid [7]. This model guides the programmer to partition the problem into coarse sub-problems which can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order. That is, blocks can be executed in parallel if there are available units; otherwise, they will be executed sequentially [7].

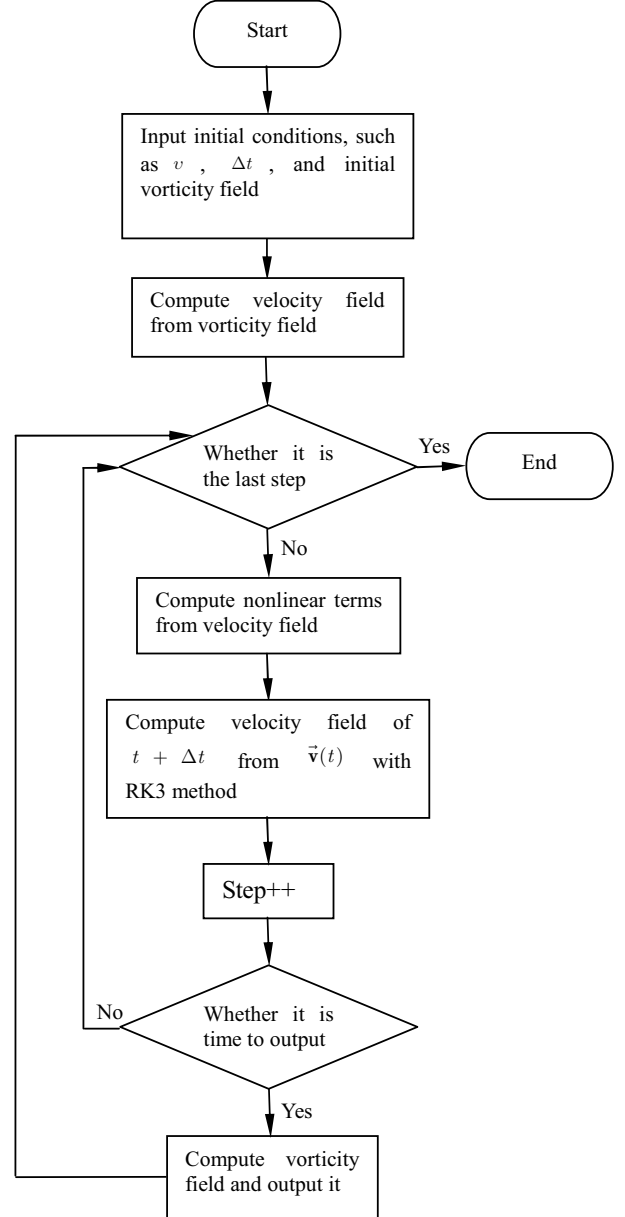


Figure 1. Computing process

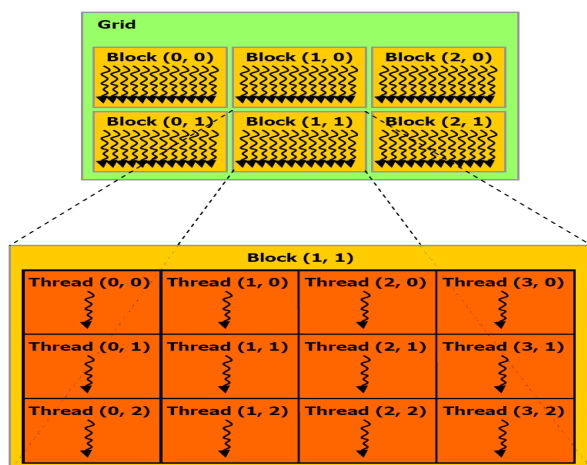


Figure 2. Structure of threads in CUDA

3.2. The Advantage of GPU over CPU in Computing

There exist differences between the CPU and GPU in scientific computing capability, and the reason is that GPU is designed for compute-intensive, highly parallel computing. Thus, the design of GPU adopts more transistors in data processing rather than in the data cache or flow control. Many lines parallel collaboration is GPU internal design, and each computing pipeline is equal to a computing unit in CPU internal core (core) which can compute a set of data independently. Therefore, GPU can be considered as hundreds of simple CPU doing data calculations at the same time, or we can say that a GPU is a small MPI parallel cluster. The floating-point computing capability of GPU is much higher than that of the CPU in the same level. Of course, the development of computing ability is accompanied by some sacrifice. The GPU does not have the flow control unit, so it is only suitable for the program of order processing with a large amount of data.

In CUDA, the thread structure is like matrix, so it is suitable for grid computing. The equations in this paper need to disperse the computational region into grids. This is naturally in line with the CUDA thread structure. In CFD, similar computation is often required again and again. The amount of computation is great but logical judgment is few. So CFD is suitable for GPU to perform its great computing capability.

4. OPTIMAZATION STRATEGIES

4.1. Reduce the Time in Host and Device Communication

PCI-E bandwidth is relatively small, only the 8GB / s, and lags far behind the bandwidth of the GPU (the GPU adopted in this article uses is C1060, and its memory bandwidth is 102GB / s). Data transmission between CPU and GPU will inevitably cause bottleneck. So the best strategy is to minimize the amount of data transferred between the CPU and GPU. Therefore, unnecessary transmission should be avoided. In order to achieve the purpose of reducing transmission quantity, we reduce the times of transmissions. We put all process into GPU, and the exchange of data between the GPU and CPU only occurs in data I/O.

In a test of a 2048×2048 simulation, when data exchange 3 times at each time step, each 1000 steps takes about 20 minutes, while only 16 minutes is needed without these exchanges.

No direct communication between the GPU exists in a

multi-node computing. There is no good way to avoid the exchange between CPU and GPU. In the low resolution case, data exchange has become the most time-consuming part in the whole program.

4.2. Shared Memory

The latency accessing to shared memory is 1 to 2 clock cycles (in the situation with non-Bank Conflict), much smaller than to the global memory (about 500 clock cycles). According to the information provided by the NVIDIA SDK, the program of matrix transpose speeds up 10 times compared with the case in which shared memory is not used. NVIDIA offers an idea to avoid bank conflict by padding an empty row. In this program, the little teaser is also applied in other sub program. Because the data in shared memory is visible to the threads in the same block, we use shared memory to avoid every thread getting its data from global memory one by one in the program of matrix adding.

4.3. Memory Coalescing

In C language, memory storage is arranged by line. 16 continuous threads' visit to continuous data period in global memory can be combined into a storage affair. In our program, besides FFT, memory operation to matrix fits this condition naturally.

Warp is organized by the SM automatically in a continuous way. For example, if there are 128 threads in a block, they will be divided into four warps: 0-31 threads will be warp 1, 32-63 warp 2, 64-95 warp 3, and 96-127 warp 4. So the best amount of threads per block is a multiple of 32. Otherwise, it will cause a warp less than 32 threads to use the same resources as a warp full of 32 threads. In this paper, each block contains $16 \times 16 = 256$ threads in a single node case. In multi-node computing, we just call CUFFT functions, and the left computation is on CPU, so we needn't take it into consideration.

4.4. Others

Some constants are calculated in the CPU, and then copied them to GPU for repeated calls.

When we have to operate with a small number of threads, we use "if threadID<N" to avoid multiple threads running at the same time which takes more time or even produce incorrect results.

Because synchronization in our program is relatively few, it is difficult to have a big performance improvement at this point.

5. RESULTS AND COMPARISON

5.1. Computing Environment

Our programs run at tesla.sccas.cn, a server of Supercomputing Center of Chinese Academy of Science. Its environment is shown as the following.

Single node: CPU Intel Xeon E5410, 2GHz CPU Clock Speed, 2×6 MB second-level cache; GPU tesla C1060 $\times 2$; memory 8GB; hard disk SATA 500GB.

Compiler: FFTW program uses "g++" to compile and CUFFT program employs "nvcc".

Multi-node uses 4 nodes with the same configuration as above.

5.2. Result in Single Node

Two codes are programmed with CUFFT and FFTW. We test our codes with different resolution (128×128 , 512×512 , 2048×2048). The performance is shown in Table 1. To confirm whether our result is right, three different times are chosen to output. After comparison of each resolution, the results of FFTW and CUFFT are found the same (Figure 3, Figure 4).

In 128×128 resolution, we calculate 6000 steps. At last, the streamlines both become two vortexes. It is a stable situation. The result of CUFFT(float) is the same, so we won't show the picture here to save space.

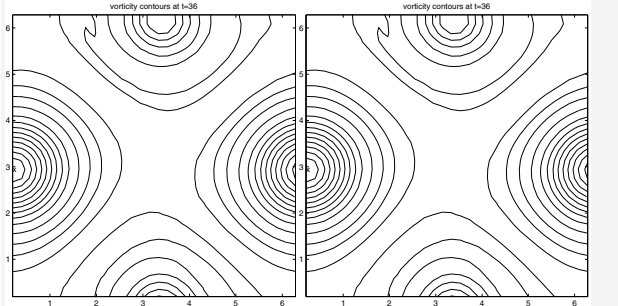


Figure 3. 128×128 result at $t=36$ CUFFT(double)(left), FFTW(right)

We also calculate 10000 steps in 512×512 resolution. At last, they both reach the stable situation (two vortexes).

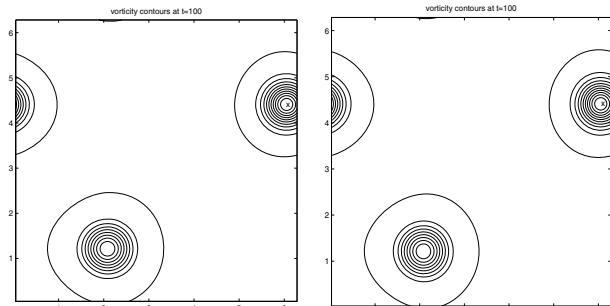


Figure 4. 512×512 result at $t=100$ CUFFT(double)(left), FFTW(right)

In 2048×2048 resolution, the results are also the same. We calculate 170000 steps, but do not reach the two vortexes' stable situation at last. However, we can see the tendency to stable situation. When calculated 100000 steps, the streamlines become several vortexes and small vortexes roll into big vortex. At last, there are two big skew vortexes in the streamline chart, which are not like the regular ones above.

Table 1. Till 1000 steps, time required by different methods

resolution	FFTW(s)	CUFFT(double)(s)	CUFFT(float)(s)
128×128	35.3	11.7	3.7
512×512	695	56	21
2048×2048	14431	999	424

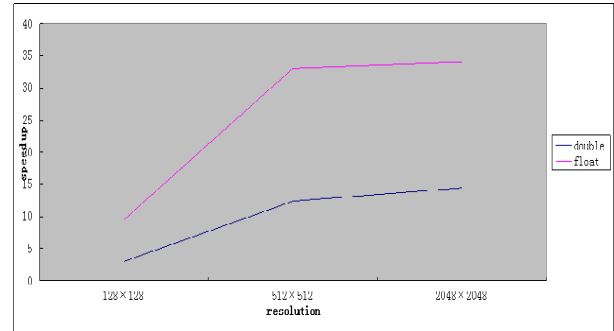


Figure 5. Speed up with CUFFT in different resolution

In 128×128 , 512×512 , and 2048×2048 cases, the corresponding CUFFT speedups are 3.03, 12.41, and 14.45.

From Figure 5, we can see CUDA's acceleration is more and more obvious with the increase of the resolution.

5.3. Performance of CUFFT and FFTW

We test the performance of CUFFT and FFTW which don't contain the communicating time. We use arrays of different sizes to test our codes. Every array performs FFT and IFFT, and then we count the time. The function clock() is adopted for timing in FFTW and the function cudaevent in CUFFT.

Table 2. Time required by different resolutions

Array size	FFTW(ms)	CUFFT(double)(ms)	speedup
128×128	1	0.3026	3.30
512×512	20	1.189	16.82
2048×2048	370	18.099	20.44

Table 2 shows that as the size increases, the advantage of CUFFT is more obvious. It meets the conclusion we get in 5.1. In addition, it is worth mentioning that the speedup ratio of both the N-S equation and pure FFT becomes smaller as the size increases. As the size grows, the communication time grows too, but we never take any measures to speed up the communication.

Table3. The different performance of CUFFT in single and double precisions

Array size	CUFFT(float)(ms)	CUFFT(double)(ms)	double/float
128×128	0.1505	0.3026	2.01
512×512	0.308	1.189	3.86
2048×2048	2.968	18.099	6.10

By comparison, we can find that single precision data has an advantage over that of double precision in GPU computing. In addition, the advantage gets bigger with the increase of the resolution. However, with the increase of data amount, besides computing ability, the bandwidth will become a limit to the calculating speed [8].

5.4. Result in Multi-Node

As the case in a single node, we use CUFFT and FFTW programming to solve the N-S equations at 4 nodes. At last FFTW and CUFFT achieve the same results under different resolutions.

In the following, the time of 100 steps' calculation at 4 nodes is shown for different methods. In the case of 2048×2048

resolution, CUFFT needs 374s and FFTW needs 493s; the speedup is 1.32 times. When resolution is 4096×4096 , 1063s is required for CUFFT and 1936s for FFTW; the speedup increases to 1.82 times.

By the above comparison, we find that as the resolution increases, GPU shows a greater advantage. When calculating the case of 4096×4096 , MPI is used to collaborate on multiple nodes. So the data transfer between Host and Device is inevitable in every step in our code. Because of a lot of time consumed in coping data, acceleration is not so obvious as which in single node. Even so, there is noticeable advantage when using GPU to compute a large amount of data. With the larger resolution, CUFFT will have better acceleration. However, limited by communication problems, acceleration will not be as obvious as that in the single node.

6. DISCUSSION AND CONCLUSION

We use CUDA to accelerate our CFD code, and there is obvious improvement after we replace the FFTW subroutines with those of CUFFT.

In the case of a single node, the performance improvement of CUDA can be found in every resolution. In addition, with higher resolution, the accelerating effect is more obvious. When the resolution is 2048×2048 , the speedup is 14.45 times. This allows us to use a higher resolution to observe the changes of the vorticity field in a fixed calculating time, and more subtle fluid structures can be observed.

TeslaC1060 is about 3 times more expensive than Intel XeonE5410. When the calculated amount is small, the economic benefit of using GPU is not very obvious, and it is not very economical to use GPU. When the resolution of our simulation increases, it is much cheaper to use GPU.

The tests in Section 5.2 show that CUDA has better performance in the single-precision calculation than in double precision, but the gap between single and double precisions does not reach the eight times speedup in theory. On the GPU of the next generation (C2050), the gap between single and double precision becomes only two-times speedup in theory. If our double-precision program is carried out on tesla C2050, better acceleration may be achieved.

Direct communication between the GPU is not supported in CUDA2.0 in multi-node. When using MPI to increase the resolution, communications must occur between Hosts. The communication between GPU and CPU is inevitable; a lot of time has to be wasted in coping data, so the advantages of GPU computing will be hidden. Maybe for a problem in which transfers between nodes are not required so frequently, the performance of GPU will be better.

7. REFERENCES

- [1] Lengyel J, Reichert M, Donald B, Greenberg D, "Real-time robot motion planning using rasterizing computer graphics hardware", ACM SIGGRAPH Computer Graphics, Vol.24, Issue.4, Aug 1990, pp.327~335.
- [2] Cohan J, Molemaker M, "A fast double precision CFD code using CUDA", Parallel CFD, 2009, pp.414~427.

- [3] Dong Tingxing, Li Xinliang, Li Sen, Chi Xuebin, "Acceleration of Computational Fluid Dynamics Codes on GPU", Computer Systems & Application, Vol.20, No.1, 2011, pp.104~109.
- [4] Dong Tingxing, Wang Long, Chi Xuebin, "the GPU acceleration of a two-dimensional diffusion equation", Computer Engineering and Science, Vol.31, No.11, 2009, pp.121~127.
- [5] [Http://www.nvidia.cn/object/cuda_home_cn.html](http://www.nvidia.cn/object/cuda_home_cn.html).
- [6] Claudio Canuto, M. Yousuff Hussaini, Alfio Quarteroni, Thomas A. Zang, Spectral Methods in Fluid Dynamics, New York: Springer-Verlag Inc. Pub., 1998.
- [7] NVIDIA CUDA Programming Guide. http://www.nvidia.cn/object/cuda_develop_cn.html.
- [8] J. Appleyard, D. Drikakis, "Higher-order CFD and interface tracking methods on highly-Parallel MPI and GPU system", Computer & Fluids, Vol.46, Issue.1, July 2011, pp101~105.