FULL LENGTH PAPER

GRID FREE EULER FLOW SOLVER WITH CUDA COMPUTING

P.V.R.R. Bhogendra Rao, K. Anandhanarayanan, R. Krishnamurthy, Debasis Chakraborty
Defence Research and Development Laboratory (DRDL)
Kanchanbagh Post, Hyderabad-500 058
Email : debasis_cfd@drdl.drdo.in; debasis_drdl@yahoo.co.in

Abstract

Graphics Accelerators are increasingly used for general purpose high performance computing applications as they provide a low cost solution to high performance computing requirements. Intel also came out with a performance accelerator that offers a similar solution. However, the existing application software needs to be restructured to suit to the accelerator paradigm. Master-slave software architecture has been employed to enable two-dimensional and three-dimensional grid-free Euler flow solvers in GPGPU computing platforms. Results showing significant improvement in the performance are presented in this paper. Convergence histories and aerodynamic forces obtained from GPGPU computing are compared with that of sequential computing results.

Keywords*: Grid-free Solver, MPI, Cluster Computing, GPGPU, CUDA*

Introduction

Applications such as Computational Fluid Dynamics (CFD), weather prediction, chemical and nuclear reaction modeling, etc., are highly computational intensive, owing to the large data sets. Efforts have been made by various researchers to reduce this computation time by parallelizing these applications on high performance computing platforms. However, the infrastructure required for the high performance computing platforms is not only prohibitively expensive and their maintenance is also highly arduous.

General Purpose Graphics Processing Units (GPGPUs) provide a low cost solution to high performance computing. GPU is especially designed for problems that are composed of data-parallel computations, that is, same operation is performed on a large set of data elements (SPMD) in parallel. As same program is executed on each data element, the need for complex control statements in the program is low.

In Data-parallel processing, each processing thread works on a data element in parallel and application programs, which operate on large data sets can exploit data-parallel programming model to speed up the computations. This is also referred to as SIMT (Single Instruction Multiple Threads) architecture.

In 2006, NVIDIA introduced Compute Unified Device Architecture (CUDA), a general purpose parallel computing architecture with a new parallel programming model along with an instruction set architecture, which takes advantage of the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. The general CUDA programming model is shown schematically in Fig.1.

CUDA is enhanced with a software environment that allows developers to program in C/C++ as a high-level programming language. Other languages or application programming interfaces are also supported such as CUDA FORTRAN, OpenCL and DirectCompute.

The application programs for multi-core programs cannot be seamlessly and transparently scaled to exploit the increasing number of processors / cores. The main design objective of CUDA parallel programming model is to overcome this challenge, while offering a low learning curve for programmers, who are familiar with standard programming languages such as C. The GPGPU, which executes CUDA threads, is a physically separate device that operates as a coprocessor to the host running the main C program. That means, the rest of the C program executes on CPU in parallel, when the kernel is launched and

Paper Code : V70 N2/1003-2018. Manuscript received on 02 Nov 2017. Reviewed, revised and accepted as a Full Length Contributed Paper on 19 Jan 2018

executed on a GPU. In the present work, a well validated indigenous grid free Euler solver is converted in CUDA programming environment to run in GPGPU computing platform. The results obtained from CUDA enabled grid free code are compared against sequential code results.

CUDA Programming Model

CUDA follows load-launch-read programming model. The initial data is loaded onto the pre-allocated GPU device memory. The parallel computing program called Kernel is launched. After completion of execution, the results are read from the device memory.

The functionality which needs to execute in parallel on the data set needs to be declared as a _global_ function and can be specified to the compiler. The kernel shall run in multiple threads, with each thread identified by a thread index.

A number of threads are grouped to one block and number of blocks into a grid. This grid can be specified as 1D, 2D or 3D depending on the data set representation of the underlying problem. There can be multiple kernels running simultaneously. A group of 32 threads is called a warp and the scheduling of threads is done in warp level.

At the time of launching the kernel, it is required to specify size of grid (number of blocks per grid) and size of block (number of threads per block). Size of block should be a multiple of warp, as thread scheduling is done in warp level and should be less than maximum block size (i.e. maximum number of threads per block which is typically 1024). The size of the warp and maximum block size can be queried from the device using cudaGetDeviceProperties( ) function. However, the configuration (i.e. number of blocks per grid and number of threads per block) depends highly on the load of the device and the resources required by the kernel.

The following are some of the factors that influence the performance of parallel program.

- Load balancing
- Race conditions
- Essential sequential computations

The effective computation time per computation node is the time taken by the node with maximum load. Since data synchronization is required at the end of each iteration of computation, the nodes which are relatively less loaded after completing the computation need to wait for the completion of the node with the maximum load. Thus, the total time for computation will be effectively more than the time taken with the balanced load.

Race condition is the major challenge to be addressed by the parallel software designer. It is the condition when more than one thread try to access same variable and at least one of the attempts is for write operation. In this situation the results will be unpredictable. In such case, it is required to synchronize the threads that are trying to access the same variable. If the threads are trying to access the variable belong to the same warp, they can be synchronized using _synchronizeThreads( ). This call can be called from within the kernel or device function.

There are certain parts of computations that cannot be computed in parallel. Such essential sequential computation part should be minimized, in order to achieve higher performance of the parallel applications [1]. The following are the major issues to be addressed while programming with CUDA.

- Identification of suitable kernel configuration
- Coalesced access to data structure memory

In a CUDA enabled GPGPU, the processing capability is split into Streaming Multiprocessors (SMs). The number of SMs depends on the card. Each SM has finite number of registers, finite amount of shared memory, maximum number of active threads per block and maximum number of active blocks per grid. These numbers depend upon the compute capability of the GPGPU. For example, in a GPGPU with compute capability 3.0 each SM can have 16 active blocks and 2018 active threads. The number of threads per block depends on the amount of memory consumed by each thread as the memory allocated per block is limited. If the number of threads per block is too less, the occupancy of GPU will be less. If the number of threads per block is too high, the memory may not be sufficient and the results may be unpredictable. The grid and block sizes are to be optimally chosen.

In a coalesced memory access, consecutive threads access consecutive memory locations. Non coalesced access to the data structure memory also leads to performance degradation of the application. In order to increase the application performance, instead of array of structures

(Code listing 1) it was advised to use structure of arrays (Code listing 2) in [7]. Keeping the above in view, a grid-free Euler solver is enabled to run on a GPGPU using CUDA. The code listing for arrays of structures and structure of arrays are shown in Fig.2.

The Grid Free Euler Solver q-LSKUM

The grid-free methods which operate on distribution of points reduce the grid generation difficulty to a greater extent. Further, the grid-free methods are amenable for parallelization due to uniform and simple data structure for even complex multi-bodies and hence it is easy to handle relatively moving multi-bodies in a parallel environment.

Least Squares Kinetic Upwind Method (LSKUM) is based on the Kinetic Flux Vector Splitting (KFVS) [2][1] scheme, which exploits the connection between the Boltzmann equation of kinetic theory of gases and the governing equations of fluid dynamics using a moment method strategy. More specifically, Euler equations are obtained by taking ψ-moments of the Boltzmann equation with Maxwellian as velocity distribution function. In LSKUM, the spatial derivatives of the Boltzmann equation are discretized using weighted least squares method and the upwinding is enforced by choosing split sub-stencils from the connectivity based on sign of the molecular velocity to evaluate the spatial derivatives. Finally, taking ψ-moments lead to LSKUM numerical scheme. The higher order accuracy in space is achieved using a defect correction method [3] in which the lower order spatial errors are removed using an iterative strategy.

An improved version of LSKUM is q-LSKUM in which the entropy variables, also called q-variables, are used in the defect correction step to achieve higher order accuracy in space at all points including boundary points. The q-LSKUM also operates on arbitrary distribution of points in the computational domain and does not require complex grid generation effort to solve the governing equations of fluid dynamics. Therefore, it considerably reduces the grid generation time and also makes it possible to obtain solutions for geometrically complex configurations. The performance of the solver crucially depends upon the quality of the connectivity (set of neighbors) to estimate the spatial derivatives of flux vectors using least squares method. The code has been thoroughly validated for complex multibody aerospace vehicles flow problems [4, 5]. The algorithmic step of the q-LSKUM grid free method is shown in Fig.3.

The System Configuration

The q-LSKUM grid free Euler Solver was implemented and tested on a cluster of computing nodes equipped with dual GPGPUs each. The system configuration consists of 8 core, dual intel Xeon CPU processor 2.0 GHz with 64 GB RAM connected through 56 Gbps infiniband switch. Each computation node consists of two NVIDIA Tesla K-20 series GPGPU. The configuration of Tesla K20 GPGPU is as follows:

- Number of GPU Cores: 2688
- Memory Size: 6 GB
- Clock Speed: 0.732 GHz
- Memory Bandwidth: 250 GBps
- Performance (SP): 3.95 TFLOPS
- Performance (DP): 1.31 TFLOPS
- Max Power Usage: 235 W

Conversion of Grid-free Euler Solver Using CUDA Programming

The problem was addressed in multiple phases. In the first phase, a 2D Euler solver was converted to CUDA followed by the conversion of 3D solver in the second phase. In the final phase, the 3D solver was converted to MPI-CUDA, in order to solve problems with large data sets.

CUDA Implementation of Grid-free 2D and 3D Euler Solvers

The main data structure of 2D grid-free Euler solver is an array of structures, as shown in Code listing 1 of Fig.2, each structure containing node attributes and an array of pointers to neighboring nodes. During the initialization process, the initial data and node structure are read from a file followed by memory allocation on the GPU device. Before the kernel is launched on the GPU, the array of structures is copied onto the device memory. After successful copying of the required data, the kernel is launched with the most appropriate configuration computed based on the present occupancy of the GPU. Because of the present node structure, each thread has to access various nodes that are not contiguous in the array. In addition, this calls for synchronization among threads of different blocks. Hence, the code has to be divided into multiple kernels, based on the access to the common device mem-

ory containing the node array. The flow chart of GPGPU implementation is shown in Fig.4.

The kernels are launched sequentially one after the other, ensuring data synchronization between successive kernel launches. The first kernel performs PminMod and construct_q functions, the second kernel performs construct_q_deriv function and the third kernel performs qresidue_flux, time_step and state_update. In the present application, some part of the code has to be essentially run sequentially on the host leading to overall performance degradation. However, the heterogeneous computing capability of CUDA has been successfully employed to improve the performance of the overall application. The essential sequential part (the aerodynamic force integration) is executed on host in parallel to the following iteration on the GPU device.

The philosophy of CUDA program model for 3D grid-free Euler Flow Solver is same as that of 2-D grid-free Euler Flow Solver. The one-time computation of Least square coefficients is also launched as a CUDA kernel before entering into iteration loop.

MPI-CUDA Implementation of Grid-free 3D Euler Solver

The MPI-CUDA software was tested with Generic wing-store separation problem [6] with 7 Million data points. In order to obtain maximum performance from MPI-GPU version of the software, it was planned to spawn only so many instances of software per computing node as many GPGPU cards present in each computing node, so that each instance can exploit one GPGPU. Remote Direct Memory access (RDMA) [9] feature of CUDA has been effectively exploited for data synchronization among the computing nodes. In addition, only points in the overlapping region are synchronized among the nodes, thus further reducing the communication demands.

Implementation Issues

Various implementation issues like kernel configuration computation, avoiding Race conditions, essential sequential computations, load balancing, etc., discussed in Section (CUDA Programming Model) are described in detail with respect to CUDA conversion of grid free Euler solver.

Kernel Configuration Computation

The kernel configuration is computed based on the current load on the GPGPU due to this kernel. Exact configuration, that can successfully launch the kernel, was computed from the results returned by cudaOccupancyMaxPotentialBlockSize( ) function. The code for performing this functionality is given in Code listing 3 (Fig.5).

Avoiding Race Conditions

Race conditions were avoided by identifying sections of the code where global shared memory was simultaneously accessed by multiple threads. It was identified in the code that the global data structure was accessed for write after construct_q and construct_q_deriv by multiple threads simultaneously. Racing condition is explained in Code listing 4 in Fig.6. In this case the filed qf is computed in the first for loop and the value of qf is used in the second for loop for computation of q derivative. If both the for loops are executed in a single kernel, it will call for a race condition wherein when one thread updating the value of qf another thread may access the same field for reading to compute the q derivative. This simultaneous access by multiple threads for write operation leads to incorrect results. In order to avoid race conditions the code was divided into multiple Slaves (kernels) at those sections of code. However, if the threads are trying to access the variable from other blocks, they needed to be synchronized using cudaSynchronizeDevice().This function can be called from the host code only.

Addressing Essential Sequential Computations

Integration of aerodynamic forces is the essential sequential computation in the present problem. Here, the heterogeneous computing capability of GPGPU has been effectively utilized. The aerodynamic forces of i^{th} iteration are integrated on the CPU, while the $(i+1)^{th}$ iteration is executed on the GPGPU. Integration of aerodynamic forces corresponding to the last iteration is performed sequentially after all iterations are completed.

Obtaining Load Balancing

In addition to control divergence, the problem being a grid-free Euler Flow solver, the number of neighboring points for different points in the grid is different and the amount of computation is proportional to the number of neighboring points leading to load imbalance. The CUDA scheduler provides load balancing functionality. Though

finer load balancing techniques were discussed in [8], as the application considered is highly computational intensive, the relatively simpler solution provided by the CUDA scheduler was found sufficient.

Coalesced Access to Data Structure Memory

In grid-free Euler solvers, the basic data structure is 1D array of nodes. The neighbours of a node will not be in consecutive locations in the array and hence achieving coalesced access to memory is not possible. However, the performance of the application was improved by copying the neighboring nodes into the thread local memory (see Code listing 4 in Fig.6) before performing computations on the data.

Results and Discussion

Parallelization of 2-D and 3-D grid free Euler solvers has been carried out on GPU thread of a single system and then multiple CPUs and GPUs. The grid free solver has been applied to 2-D and 3-D test cases to verify the repeatability of the results of sequential and MPI version of the codes and performance of the code on GPUs.

Results of 2D Euler Flow Solver

Both the sequential and parallel 2-D flow solvers were used to simulate transonic flow past NACA0012 airfoil at Mach number 0.8 and angle of attack 2° in the present work. Simulations are carried out on two sets cloud of points with sizes 5920 and 12240. Typical cloud of points is shown in Fig.7. The shock wave on upper surface is captured well by the solver. The 2D flow solver was executed for 20,000 iterations. The aerodynamic coefficients attained steady state at 10,000 iterations and the results from both the solvers compare exactly with each other. The comparison of residue history is shown in Fig.8. The residue has fallen 4 decades and they also compare well. This demonstrates the parallel solver works correctly and reproduces the solution that of the sequential solver. The performance of CUDA parallel version of 2-D grid free Euler solver for the present test case is compared with that of sequential version running on the CPU and the results are given in Table-1. T_s and T_p are the time taken by sequential computation on CPU system and parallel computation on GPU system. It can be noted that a performance improvement of about 12X is obtained. With the increase of grid point, there are no significant changes in performance.

Similar work was carried-out using GPGPU for 2D grid-free method for solving compressible flow problems and was presented in [7] . For single aerofoil with 5557 points with space-filling curves a speedup of ~10X was reported with Quadro 2000 GPU.

Results of 3D Grid Free Euler Solver

The 3-D q-LSKUM grid free solver is converted using CUDA and applied to generic wing-store problem [6]. The experimental results are available for this test case for validation. The configuration consists of a 45 degree clipped delta wing with NACA 64A010 airfoil section and an ogive-flat plate-ogive pylon. The store consists of a tangent-ogive forebody, clipped tangent-ogive aft-body, and cylindrical section center-body. The store has cruciform fin of a 45 degree sweep clipped delta wing with NACA 008 airfoil section. Similar to the experimental set-up, a small gap exists between the store body and the pylon while in carriage. As in the experimental tests, a sting is attached to the store aft-body. The free stream Mach number is 0.95 and angle of attack is 0°. A coarse cloud of 42664 points are used in the simulation. Time evolution of aerodynamic forces and moments are plotted in Figs.9 (a) and 9(b) respectively. The results attain steady state at 1200 iterations and the results from both solvers compare exactly with each other. The residue has fallen 2 decades and they also compare well. The parallel version on GPGPU has shown a performance improvement of 16.22 times over the sequential counterpart on CPU.

Results of MPI-GPU Implementation

The parallel version of 3-D q-LSKUM grid free solver using MPI has been converted using CUDA to run on GPU

Table-1 : Performance of CUDA of 2D Grid Free Euler Solver

Sl. No.	No. of Grid Points	T_s	T_p	CUDA Config.	T_s / T_p
1.	5920	599.132	48.449	Grid Size : 24 Block Size : 256	12.366
2.	12240	1235.680	104.840	Grid Size : 48 Block Size : 256	11.790

cluster utilizing both CPUs and GPUs. This code has been applied for a store separating at transonic flow condition of Mach number 0.95 and 0° angle of attack. The grid size around wing is 1,524,939 points and around store is 783, 836 points amounting to 2,308,775 points on the cloud.The surface grids on the wing and store are shown in Fig.10 and the corresponding pressure contours are shown in Fig.11. It can be observed from Fig.11 that the compression near wing leading edge and strong normal shock near trailing edge are well captured by the MPI-GPU version. It was found that the results of sequential version of the code and CUDA version of the code are matching up to 15th digit after decimal. The aerodynamic force coefficients, aerodynamic moment coefficients and residue history of MPI-GPU version exactly match with those of MPI version.

Conclusions

2-D and 3-D grid free Euler solvers were converted into CUDA and MPI-GPU environments. The results obtained with CUDA parallel version of the software over GPGPU were found to be identical to those of the sequential CPU version with good accuracy. The CUDA application has demonstrated a peak performance improvement of 16X. It can be noted that higher the amount of computation, better is the performance improvement.

It was observed that the performance of MPI-GPU version improves with increase in amount of computation per computation node. Though load balancing could not be achieved at the application level, the load balancing performed by the CUDA frame work of GPGPU produced good results. CFD applications based on structured grid may offer better performance improvement. Similarly coalesced memory access may further improve the performance.

References

1. Amdahl, G.M., "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities", Proc. American Federation of Information Processing Societies Conf., AFIPS Press, 1967, pp.483-485.

2. Mandal, J. and Deshpande, S., "Kinetic Flux Vector Splitting for Euler Equations", Computers and Fluids Journal, vol.~23, 1994, pp.447-478.

3. Deshpande, S., "Meshless Method, Accuracy Symmetry Breaking, Upwinding and LSKUM", Tech. Rep. Fluid Mechanics, Report No.2003 FM 1.

4. Ananadhanarayanan, K., "Development of 3D Grid-free Solver and its Applications to Multi-body Aerospace Vehicles", Defence Science Journal, Vol.60, No.6, November 2010, pp.653-662.

5. Ananadhanarayanan, K., Konark Arora., Vaibhav Shah., Krishnamurthy, R. and Debasis Chakraborty., "Separation Dynamics of Air-to-Air-Missile Using a Grid-free Euler Solver", AIAA Journal of Aircraft, Vol.50, No.3, May-June 2013, pp.725-731.

6. Heim, R. R., "CFD Wing/Pylon/Finned Store Mutual Interference Wind Tunnel Experiment", AEDC-TSR-91-P4, 1991.

7. Ma, Z.H., Want, H. and Pu, S.H., "GPU Computing of Compressible Flow Problems by a Meshless Method with Space-filling Curves", Journal of Computational Physics, Vol.263, 2016, pp.113-135.

8. Long Chen., Oreste Villa., Sriram Krishnamoorthy and Guang R. Gao., "Dynamic Load Balancing on Single- and multi-GPU Systems", IEEE International Symposium on Pararllel and Distributed Processing (PIDPS), 19-23, April 2010, Atlanta, USA.

9. Developing A Linux Kernel Module Using RDMA For GPUDirect - Application Guide, NVidia, TB-06712-001 _v9.1, 2017.

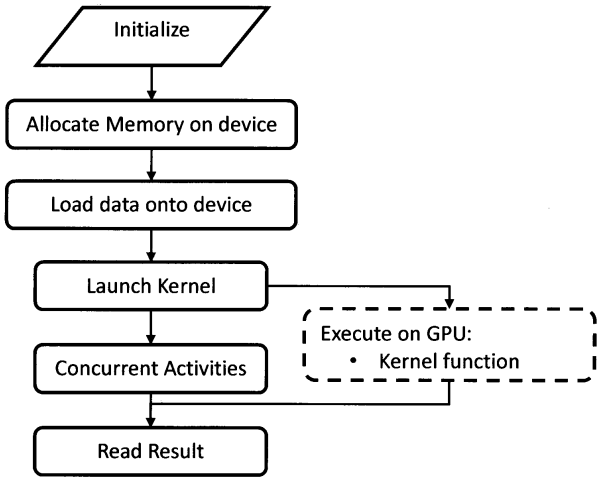

Fig.1 CUDA Programming Model

Code Listing 1: Array of structures

```
typedef struct _node{
      double x, y, z;
      double u,v,rho,t;
} Node;

Node nodes[MaxNodes];
```

Code Listing 2: Structure of arrays

```
typedef struct _node{
     double x[MaxNodes];
     double y[MaxNodes];
     double z[MaxNodes];
     double u[MaxNodes]
     double v[MaxNodes];
     double rho[MaxNodes];
     double t[MaxNodes];
} Node;

Node nodes;
```

Fig.2 Code Lising (1) Array of Structures and (2) Structure of Arrys

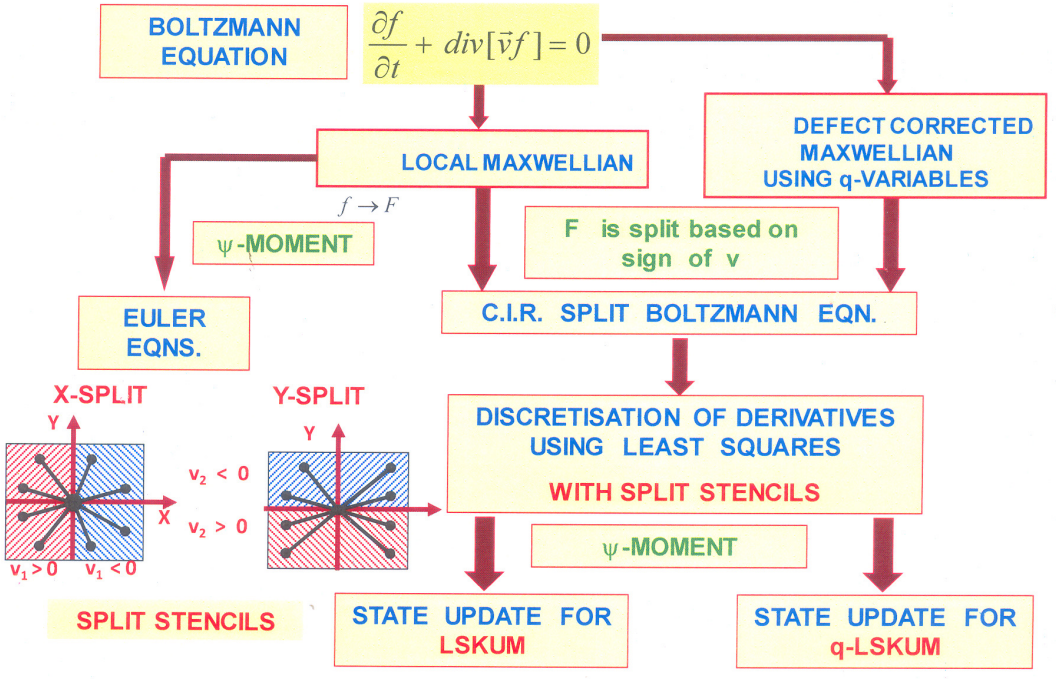

Fig.3 Algorithmic Step of the q-LSKUM Grid Free Method

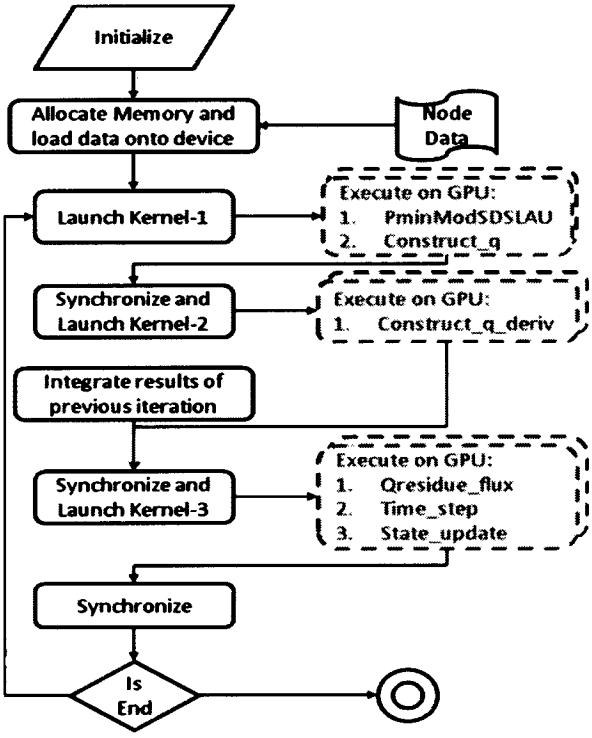

Fig.4 Implementation Model of 2D Euler Flow Solver

Code Listing 3: Kernel Configuration Computation

```
int gridSize = 0;
int blockSize = 0;
int MinGridSize = 0;

cudaOccupancyMaxPotentialBlockSize (&minGridSize,
                        &blockSize,
                        (void *)KernelFunction,
                        0,
                        MaxNodes);

if (blockSize != 0)
     gridSize = (MaxNodes + blockSize - 1 ) / blockSize;
```

Fig.5 Code Lising for Kernel Configuration Computations

Code Listing 4: Race Condition

```
for (n = 0; n < MaxNode; n++) {
      ul = Nodes[n].u;
      vl = Nodes[n].v;
      wl=Nodes[n].w;
      rhol = Nodes[n].rho;
      tl = Nodes[n].t;
// compute qf
      Nodes[n].qf[0] = qfl[0];
      Nodes[n].qf[1] = qfl[1];
      Nodes[n].qf[2] = qfl[2];
      Nodes[n].qf[3] = qfl[3];
      Nodes[n].qf[4]  qfl[4];
}
For (n=0; n < MaxNode; n++) {
      Xl = Nodes[n].x;
      Yl = Nodes[n].y;
      Zl = Nodes[n].z;
      Qfl[0] = Nodes[n].qf[0];
      Qfl[1] = Nodes[n].qf[1];
      Qfl[2] = Nodes[n].qf[2];
      Qfl[3] = Nodes[n].qf[3];
      Qfl[4] = Nodes[n].qf[4];
// Compute q derivative
      Nodes[n].qfx[0] = qfxl[0];
}
```

Fig.6 Code Lising for Race Condition

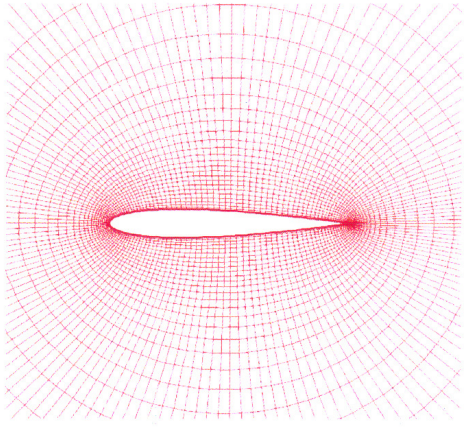

Fig.7 Cloud of Points Around NACA0012 Airfoil

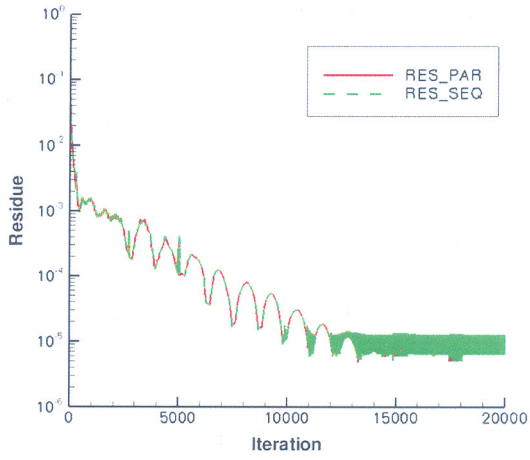

Fig.8 Residue History

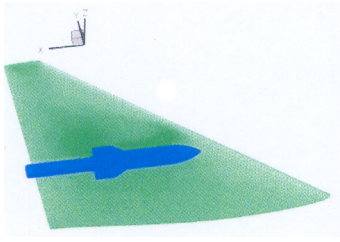

Fig.10 Surface Grid

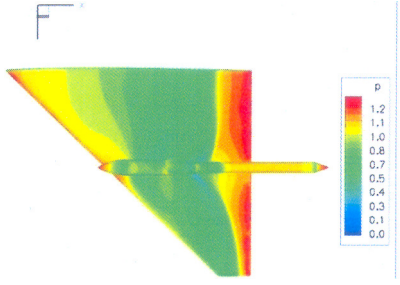

Fig.11 Pressure Contours

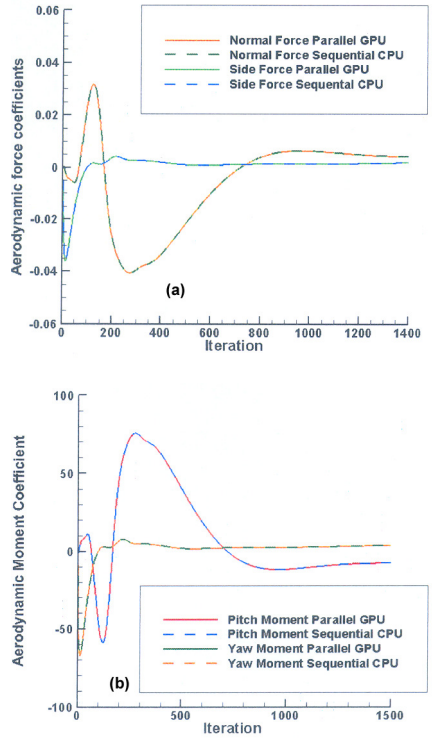

Fig.9 Convergence History of (a) Aerodynamic Force and (b) Aerodynamic Moment