FULL LENGTH PAPER

MPI-CUDA IMPLEMENTATION OF IMPLICIT EULER FLOW SOLVER IN GRID-FREE FRAMEWORK

P.V.R.R. Bhogendra Rao, K. Ananadhanarayanan, R. Krishnamurthy, Debasis Chakraborty
Scientist
Defence Research and Development Laboratory (DRDL)
Kanchanbagh Post, Hyderabad-500 058, India
Email : debasis_cfd@drdl.drdo.in; debasisc.cfd@gmail.com

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. However, the existing application software needs to be restructured to suit to the accelerator paradigm. Explicit methods are inherently suitable for parallelization whereas implicit methods are not suitable as the nodes need to be processed in specific order. However, the nodes can be grouped into clusters such that nodes within the cluster are independent and can be processed concurrently. CUDA kernel can be launched separately for each cluster of nodes to process nodes in parallel leading to same computation results as sequential program. One such successful attempt has been made and the speed up obtained along with computation results is presented in this paper.

Introduction

CFD applications are highly computational intensive, owing to the large data sets and intensive computations involving large number of iterations. Typically for a store separation dynamics problem, about 2000 to 4000 iterations of simulations have to be executed at 30 to 40 CFD simulation points along the trajectory over a geometry containing about 15 million grid points for obtaining meaningful results. The number of such trajectories for each air-launched mission is of the order of 300. Hence, a faster method to generate huge CFD based trajectory data is necessary. Here already proven store separation dynamics suite consisting of MPI based grid free Euler solver along with pre-processor and 6 DOF trajectory code has been considered for enhancing the speed.

The power of the GPGPU threads has been exploited to perform the number crunching of highly intensive CFD simulations. This methodology is likely to give reduced run time ensuring 2 to 3 times increase in speed-up resulting in higher number of trajectories of stores when released from the aircraft. Efforts have been made by various researchers to reduce this computation time by parallelizing the 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 and low energy consuming solution to high performance computing [1,2,3,4]. GPU is especially designed for problems that can be expressed as data-parallel computations, that is, same program is executed on many data elements (SPMD) in parallel [5]. GPUs are designed such that they devote more transistors for data processing. As same program is executed on each data element, the requirement for sophisticated control statements in the program is low. Application programs that process large data sets can use a data-parallel programming model to speed up the computations.

Bhogendra Rao et al. [6] has employed Master-slave software architecture to convert a two-dimensional and three dimensional grid free CFD solver [7] in GPGPU computing platform and showing significant improvement in the performance. In the present work, an implicit grid free solver is converted in CUDA for GPGPU computing platform. The performance of GPGPU computing with explicit and implicit grid free solvers are compared for air launched store separation problems.

Paper Code : V71 N3/1047-2019. Manuscript received on 21 Jan 2019. Reviewed and accepted as a Full Length Contributed Paper on 26 Jul 2019

Grid-Free Euler Flow Solver

The grid-free methods which operate on distribution of points reduces 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) [8] 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 ?-moments lead to LSKUM numerical scheme. The higher order accuracy in space is achieved using a defect correction method [9] 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 the geometrically complex configurations. The performance of the solver crucially depends upon the quality of the connectivity (set of neighbours) to estimate the spatial derivatives of flux vectors using least squares method. The algorithmic steps of q-LSKUM are given in Fig.1.

Implicit Grid-Free Euler Solver

The Euler equations governing unsteady compressible inviscid flows can be expressed in the conservative form as

$$\frac{\partial U}{\partial t} + \frac{\partial F^j}{\partial x_j} = 0 \tag{1}$$

In order to obtain a steady-state solution, the spatially discretized Euler equations must be integrated in time. Using Euler implicit time-integration, Eq.(1) can be written in discrete form as

$$\frac{\Delta U_n^i}{\Delta t} = -R_i^{n+1} \tag{2}$$

Equation (2) can be linearized in time as

$$\frac{\Delta U_n^i}{\Delta t} = -\left(R_i^n + \frac{\partial R_i^n}{\partial U}\Delta U_i\right) \tag{3}$$

Where R_i is the right-hand-side residual and equals to zero for a steady state solution. Writing the equation for all nodes leads to the delta form of the backward Euler Scheme

$$A\Delta U = -R \tag{4}$$

Where

$$A = \frac{1}{\Delta t} I + \frac{\partial R^n}{\partial U} \tag{5}$$

The following simplified flux function is used to obtain the left-hand-side Jacobian Matrix

$$R_i = \sum \frac{1}{2}\left[F(U_i, n_{ij}) + F(U_j, n_{ij}) - |\lambda_{ij}|(U_j - U_i)\right] \tag{6}$$

Where

$$|\lambda_{ij}| = |V_{ij} \cdot n_{ij}| + C_{ij} \tag{7}$$

Where n_{ij} is the geometric property obtained using least squares method, C_{ij} is speed of sound, and the summation is over all neighboring vertices j of vertex i. The left-hand-side Jacobian matrix can be expressed in upper, lower, and diagonal forms as

$$U_{ij} = \frac{1}{2}(J(U_j, n_{ij}) - |\lambda_{ij}| I)|S_{ij}| \tag{8}$$

$$L_{ij} = \frac{1}{2}(-J(U_i, n_{ij}) - |\lambda_{ij}| I)|S_{ij}| \tag{9}$$

$$D_{ij} = \frac{V}{\partial t} I + \sum_i \frac{1}{2} (J (U_i, n_{ij}) + |\lambda_{ij}| I) |S_{ij}| \qquad (10)$$

In this method, the matrix A of Eq.(5) is split in three matrices, a strict lower matrix L, a diagonal matrix D, and a strict upper matrix U. This system is approximately factored by neglecting the last term on the right-hand side of Eq.(11). The resulting equation can be solved in the two steps shown in Eqs.(12) and (13), each of them involving only simple diagonal matrix inversions.

$$(D + L) D^{-1} (D + U) \Delta U = R + (LD - IU) \Delta U \qquad (11)$$

Lower (forward) sweep:

$$(D + L) \Delta U^* = R \qquad (12)$$

Upper (backward) sweep:

$$(D + U) \Delta U = D \Delta U^* \qquad (13)$$

It is clear that the above algorithm involves primarily the Jacobian matrix-solution incremental vector product. Such operation can be approximately replaced by computing increments of the flux vector ΔF.

$$J \Delta U \approx \Delta F = F (U + \Delta U) - F (U) \qquad (14)$$

The forward sweep and backward sweep steps can then be expressed as

$$\Delta U_i^* = D^{-1} \left[R_i - \sum_{j : j < i} \frac{1}{2} (\Delta F_j^* - |\lambda_{ij}| \Delta U_j^*) \right] \qquad (15)$$

$$\Delta U_i = \Delta U_i^* - D^{-1} \left[R_i - \sum_{j : j > i} \frac{1}{2} (\Delta F^j - |\lambda_{ij}| \Delta U_j) \right] \qquad (16)$$

In the grid-free method, the sweeps are carried out based on node number. The terms on the right hand side indicate how node number relates to central node.

MPI-GPU Implementation of Implicit 3D Euler Flow Solver

In order to reduce the number of iterations that are required for convergence, an attempt was made to convert the existing MPI version of 3D implicit Euler flow solver to MPI-GPU. However, the existing implicit version was found not suitable for GPGPU computing, as the nodes need to be processed in order during state update. In order to make the implicit 3D Euler flow solver suitable for parallel implementation, the nodes are renumbered such that independent nodes can be processed concurrently. The 3D flow solver involved computation of least square coefficients before entering into iteration loop. This computation of least square coefficients is also computed on GPGPU device in parallel, as shown in the flowchart of Fig.2.

MPI Version of Implicit Euler Flow Solver

The code **Listing 1** shows the existing MPI implementation of state_update () function. It can be observed that the nodes are processed in forward order followed by backward order. However in CUDA implementation, all the nodes are processed concurrently and any ordering on nodes will be impetus to the performance. If the ordering is not enforced the results will be incorrect. However, the nodes which are not adjacent to each other can be processed concurrently. Based on this concept a coloring scheme has been implemented.

```
1  for( int i = 0; i<MaxIterLocal ; l ++) {
2      for ( int n = 0; n <MaxNode; n++) {
3          forwardSweepFPM (n) ;
4      }
5
6      exchange dU ( ) ;
7
8      for ( int n = MaxNode-1; n >= 0; n++) {
9
10         backwardSweepFPM (n) ;
11     }
12
13     exchange dU ( ) ;
14
15 }
```

Listing 1 : MPI Code for Implicit State Update

Coloring Scheme and CUDA Implementation

In the coloring scheme, the nodes are renumbered/reordered [10] such that nodes which are mutually independent can be processed concurrently. The reordering algorithm is as follows:

1. Mark an arbitrary grid node as belonging to hyper-plane number 1. Set the current hyper-plane number Np to 1.

2. Assign mark Np + 1 to all non-marked nodes connected to the nodes marked as Np. Increment the current plane number Np by 1.

3. If not all nodes are marked, repeat step 2. One can see that nodes belonging to one group may be connected to each other. To avoid such connections, step 4 is required.

4. Examine nodes from each hyper-plane. For each hyper-plane, mark (color) the nodes so that no nodes of the same color are connected to each other. This coloring procedure is similar to one usually applied for vectorization of unstructured grid codes. According to the colors, assign new group marks to the nodes.

Before calling the solver routines, the nodes are renumbered using the above algorithm. The coloring scheme groups the nodes such that nodes within the group are mutually data independent and can be processed concurrently.

The array variable nodepas consists of starting numbers of nodes belonging to each group. nodepas[i] contains starting number of the node of ith group. The variable NoPass represents total number of such groups. The loop iterates over the number of groups and concurrently performs forward sweep followed by backward sweep with exchange of DUs after each sweep as shown in the code **Listing 2**. The ForwardSweepWrapper () and BackwardSweepWrapper () compute the kernel configuration based on the size of the group, that is number of nodes in the group, and launch the kernel

CUDA Programming Model for Implicit 3D Euler Flow Solver

The CUDA programming model for implicit 3D Euler flow solver is given in the flowchart of Fig.2. It can be observed from the figure that before state update (), two more kernels are included - namely ForwardSweep () and BackwardSweep (). As the names suggest the order of processing the nodes is important in these two kernels. However, due to the coloring scheme, independent nodes are processed concurrently and synchronization of data is required after the processing. This need for additional data

```
1 for ( l =0; l<MaxIterLocal ; l ++) {
2     for ( ipass = 0; ipass<NoPass ; ipass ++) {
3
4 // parallel forward sweep of the nodes in the group
5         ForwardSweepWrapper (nodepas[ ipass ] ,
6             nodepas[ ipass +1] , ipass , NoPass ) ;
7     }
9
10    exchange dU( dcomm buffs , dnode , d InitData ) ;
11
12    for ( ipass = NoPass-11; ipass>= 0; ipass--) {
13
14 // parallel backward sweep of nodes in the group
15        BackwardSweepWrapper (nodepas[ ipass ] ,
16            nodepas[ ipass +1] , ipass ,NoPass ) ;
17    }
18
19
20    exchange dU( dcomm buffs , dnode , d_InitData ) ;
21 }
```

Listing 2 : MPI-GPU Code for Implicit State Update

synchronization deters the performance of implicit Euler flow solver when compared to the explicit version.

Implementation Issues

Various important issues that are to be addressed while developing CUDA applications are discussed in this subsection.

Kernel Configuration Computation : The kernel configuration depends highly on the current load of the device and the resources required by the kernel as there can be multiple kernels running simultaneously. Exact block size, that can successfully launch the kernel, can be computed from the results returned by cuda OccupancyMaxPotentialBlockSize ( ) function [11,12].

After obtaining the optimal block size, the grid size was computed as shown in the code for performing this functionality given in code **Listing 3**

Avoiding Race Conditions : Race conditions were avoided by identifying sections of the code where global shared datum was simultaneously accessed by multiple threads. The code was divided into multiple kernels at those sections of code; each for loop is coded into one kernel and global data synchronization is performed using cudaSynchronizeDevice () after execution of each kernel. Thus, if the threads trying to access the variable were across blocks and they needed to be synchronized using cudaSynchronizeDevice (). This function can be called from the host code only. If the data synchronization is

```
1 int gridSize ; //Number of blocks per grid
2 int blockSize ; // Number of threads per block returned by CUDA
3 int MinGridSize ; // Minimum gr id size returned by CUDA
4
5 cudaOccupancyMaxPotentialBlockSize (&minGridSize ,
6             &blockSize ,
7             ( void* ) KernelFunction ,
8             0,
9             MaxNode ) ;
10
11 gridSize = (MaxNode + blockSize-1) / blockSize ;
```

Listing 3 : Source Code Listing for Kernel Configuration Computation

required within the block synchronizeThreads () will achieve the same within the block.

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

Achieving Load Balancing : The problem being a grid-free Euler Flow solvers, the number of neighboring points for different points in the grid are different and the amount of computation is proportional to the number of neighboring points. Hence, the load balancing could not be achieved. Load balancing techniques for unstructured grid solvers was discussed in [13]. In contrast, structured grid solvers will provide good load balancing. However, as the amount of data that is processed more than the number of threads of CUDA, the CUDA scheduler would optimize the utilization of GPUs.

Obtaining Coalesced Memory Access :The problem being grid-free, coalesced memory access could not be achieved. However, various node renumber techniques to improve the memory access patterns were discussed in [14,15,16,17,18]. It is required to implement one of these strategies to improve the coalesced memory access and thereby performance of the CUDA software.

Software Validation for Generic Wing-Store Problem

The MPI-GPU software was tested with Generic wing-store separation problem [19] with 7 Million data points at transonic flow condition of Mach number 0.95 and angle of attack 0°. The software was tested on a cluster of compute nodes. Each computing node in cluster is equipped with two NVIDIA Tesla K-20 series GPGPU accelerators. 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

In order to obtain maximum performance from MPI-GPU version of software, it was planned to spawn only two instances of software per computing node, so that each instance can exploit one GPGPU. It is required to specify, in the job file for the scheduler of the HPC cluster, that the number of instances that the applications runs on each computing node should be equal to the number of GPU cards available on each compute node. Some schedulers also support specification for scheduling the software only when the GPU cards are not used by other applications. RDMA feature of CUDA has been effectively exploited for data synchronization among computing nodes. In addition, only points in the overlapping region are synchronized among the nodes, thus further reducing the communication demands.

An 8 node GPGPU cluster configuration utilizing both CPUs and GPUs containing one master node and seven slave nodes are used in the present computations. The residue histories of MPI and MPI-GPU computations are compared in Fig.3 and a very good match between the two is observed. The pressure contours on lower and upper surfaces of the wing and store, computed by the MPI-GPU version, are shown in Fig.4. It can be noted from the results that the output of GPGPU version of implicit Euler flow solver is exactly matching with those of MPI versions. The computation time for 20,000 iterations and the corresponding speed-up factor are given in Table-1. It is observed that the MPI-GPU version of implicit Euler flow solver is only 2.6 times better than the MPI colored version where as the explicit Euler flow solver [6] was 4 times better than its MPI counterpart. This is mainly due to excessive exchange of data between the compute nodes which in turn increases the data transfer to and from GPGPU device memory. The performance of MPI-GPU

Table-1 : Comparison of Execution Times and Speed-up		
Execution Time (Sec)		**Speed-up**
MPI	GPU	$T_{MPI}/T_{MPI-GPU}$
71434	28588	2.498

version can be improved if the data transfer is minimized, if not avoided.

Conclusions

An implicit grid free CFD solver has been successfully converted to CUDA through master slave architecture for executing in GPGPU platform. All race conditions are successfully avoided in the applications and heterogeneous computing feature of GPGPU is made use of for concurrently executing the essential sequential part of the program on CPU, and thus increasing the performance of the application. Generic wing store problem was taken as the validation case and the problem is run on a cluster of 8 node GPGPU configuration containing one master node and seven slave nodes. The computed solutions of MPI and MPI-GPU are very closely matching with each other. The speed up factor of MPI-GPU version of implicit Euler flow solver (2.6X) is much less compared to explicit flow solver (4X). Excessive exchange of data to and from between the compute nodes GPGPU device memory is mainly responsible for the lower speedup. Load balancing and use of coalesced memory access could not be implemented because of unstructured mesh which could have further improved the GPGPU performance of the implicit solver.

References

1. Ashby, S., Beckman, P., Chen, J., Colella, P., Collins, B., Crawford, D., Dongarra, J., Kothe, D., Lusk, R., Messina, P., "The Opportunities and Challenges of Exascale Computing", 2010.

2. Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.-C., Barkai, D., Berthou, J.-Y., Boku, T., Braunschweig, B., "The International Exascale Software Project Roadmap", Int. J. High Perform. Comput. Appl., 25 (1), 2011, pp.3-60.

3. Duranton, M., Black-Schaffer, D., De Bosschere, K. and Maebe, J., "The HiPEAC Vision for Advanced Computing in Horizon 2020", 2013.

4. Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W. and Hill, K., "ExaScale Computing Study: Technology Challenges in Achieving ExaScale Systems", 2008.

5. Liu, Y., Wirawan, A. and Schmidt, B., "CUDASW++3.0: Accelerating SmithWaterman Protein Database Search by Coupling CPU and GPU SIMD Instructions", BMC Bioinformatics 14 (1), 2013, p.110.

6. Bhogendra Rao, P. V. R. R., Anandhanarayanan, K., Krishnamurthy, R. and Debasis Chakraborty., "Grid Free Euler Flow Solver with CUDA Computing", Journal of Aerospace Science and Technologies, Vol.70. No.2, pp.111-119, May 2018.

7. Anandhanarayanan, 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.

8. Mandal, J. C. and Deshpande, S. M., "Kinetic Flux Vector Splitting for Euler Equations", Computers and Fluids Journal, Vol.23, 1994, pp.447-478.

9 Deshpande, S. M., "Meshless Method, Accuracy Symmetry Breaking, Upwinding and LSKUM", Department of Aerospace Engineering, Indian Institute of Science, Bangalore, Technical Report : Fluid Mechanics, Report No.2003 FM 1, Dmitri Sharov and Kazuhiro Nakahashi, "Reordering of Hybrid Unstructured Grids for Lower-Upper Symmetric Gauss-Seidel Computations", AIAA Journal, Vol.36, No.3, Technical Notes, 1998, pp.484-486.

10. NVIDIA CUDA Reference Manual, 3rd Edition, NVIDIA Inc., August 2010.

11. CUDA C Programming Guide, 3rd Edition, NVIDIA Inc., August 2014.

12. Rainald Lohner and Ramamurti, R., "A Load Balancing Algorithm for Unstructured Grids", Computational Fluid Dynamics, Vol.5, 1995, pp.39-58.

13. 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.

14. Sagari, H., "Space-Filling Curves", Springer Verlag, 1994.

15. Rainald Lohner., "Some Useful Renumbering Strategies for Unstructured Grids", International Journal of Numerical Methods in Engineering, Vol.36, 1993, pp.3259-3270.

16. Lohner, R., "Renumbering Strategies for Unstructured-grid Solvers Operating on Shared-memory, Cache-based Parallel Machines", Computer Methods in Applied Mechanics and Engineering, Vol.163, 1998, pp.95-109.

17. Rainald Lohner and Martin Galle, "Minimization of Indirect Addressing for Edge Based Field Solvers", Communications in Numerical Methods in Engineering, 2002, pp.335-343.

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

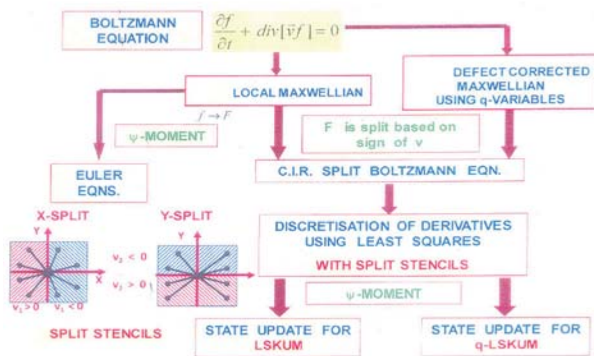

Fig.1 Algorithmic Steps of q-LSKUM

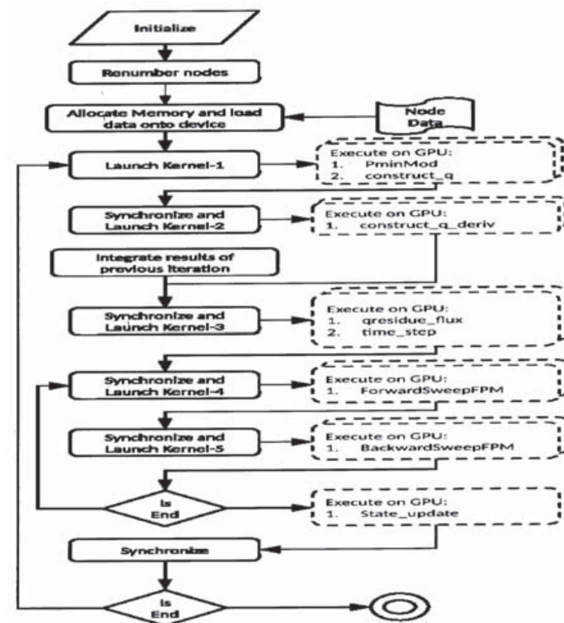

Fig.2 CUDA Programming Model for Implicit q-LSKUM

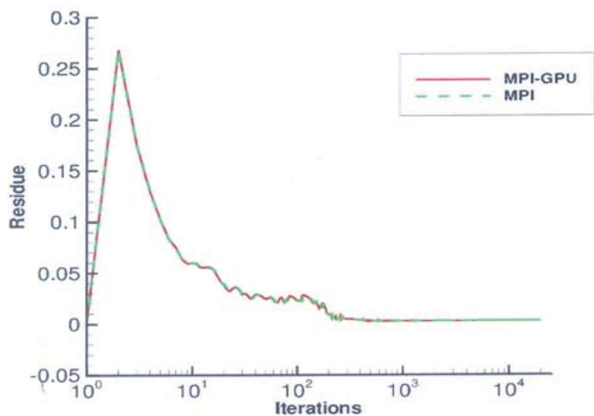

Fig.3 Residue History

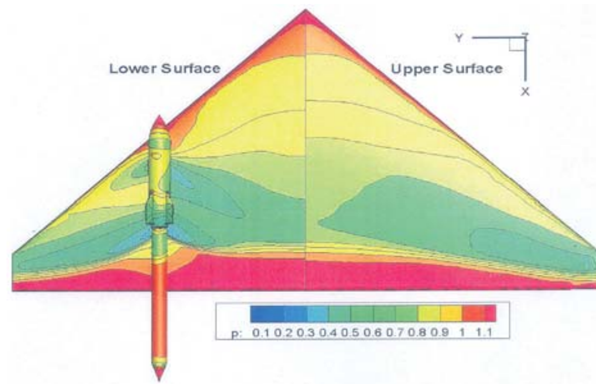

Fig.4 Pressure Contours