Seventh International Conference on Computational Fluid Dynamics (ICCFD7), Big Island, Hawaii, July 9-13, 2012

ICCFD7-1202

Dynamic Overset Grid Computations for CFD Applications on Graphics Processing Units

D. Chandar*, J. Sitaraman* and D. Mavriplis*
Corresponding author: dchandar@uwyo.edu

* Mechanical Engineering, University of Wyoming, Laramie, WY, USA.

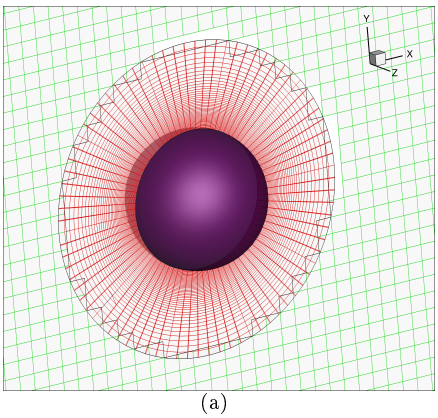
Abstract: The objective of the present work is to discuss the development of a 3D Unstructured-Overset grid Computational Fluid Dynamics (CFD) solver on General Purpose Graphics Processing Units (GPGPUs). As an extension of our previous work on 2D/3D overset grid computations for compressible/incompressible flows on static grids[1][2], the current paper focuses on moving overset grids with dynamic domain connectivity on the GPU. To validate the solver, numerical results are presented for the flow past a moving sphere and an oscillating wing at low Reynolds numbers. An identical serial CPU version of the code with two levels of optimization is generated automatically, by placing compiler directives at appropriate places in the GPU code. A one to one comparison between the GPU and serial code is then made to evaluate the performance of the GPU. Results indicate that the basic, unoptimized GPU flow solver runs 15x and 5x faster than the identical zero and highest level optimizations of the serial C++ code respectively.

Keywords: Graphics Processing Units, Overset/Overlapping Grids, Unstructured Finite-Volume methods.

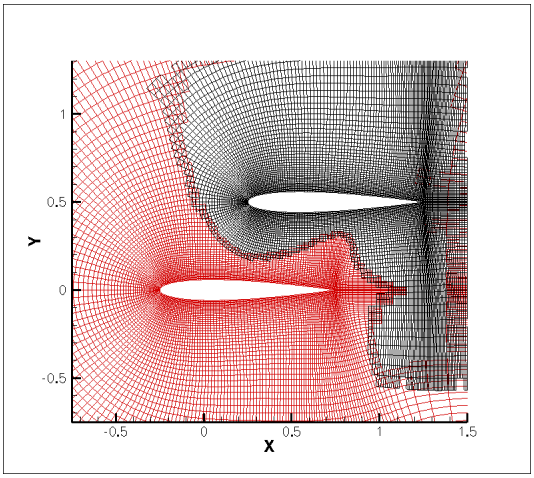
1 Introduction

The advent of GPGPUs has spawned a lot of interest in computing high resolution flows in a shorter span of time. In combination with existing parallel programming techniques, one is able to obtain at least one order increase in speed-up for CFD applications on stationary grids[3],[4],[5]. For moving body applications, if a single grid is used, the grid needs to be regenerated or deformed causing other numerical difficulties. Also, the usage of a single grid for the entire domain would render the computation inefficient as most of the flow features are present in the vicinity of the body. The usage of overset or overlapping grid methods for these purposes is quite attractive as each grid can have its own resolution, and one may place finer grids only at specified locations to capture the flow features. Figures 1(a),(b) show overset grids for the transonic flow past two airfoils and low Reynolds number flow past a sphere respectively based on the authors' previous work[1],[2]. In both these computations, the body under consideration does not move, hence the grids are static. The process of finding prospective donors for fringe points in the overlapping grid (overset connectivity) is performed once the grids are pre-processed at the beginning of the simulation. However, when there is a finite motion of the body, the donor search process needs to be computed at each time step. Moreover, this should become a part of the main flow solver unlike in [2] where the overset grid connectivity is generated using an independent GPU code, and the connectivity information is then passed on to the main solver. In the present work, we demonstrate a modular concept using C++ object oriented methodologies where the domain of interest is represented by several grid systems that reside on the GPU. Surrounding each body of interest, we generate unstructured grids termed as the near-body grids. Each near-body grid is connected to an off-body Cartesian or unstructured grid through fringe points. Solutions at fringe points are obtained from donor grids using Taylors' expansion. The algorithm to determine donor points follows the method in [1]. The advantages of representing the overset grid system in this way is that, when multiple GPUs are present, each GPU will be able to handle the motion of each moving body. To the authors' knowledge, this is the first computation to demonstrate the effectiveness of GPUs for moving

body problems using overset grids. To validate the approach, we present two test cases in low Reynolds number conditions: (1) flow past a translating sphere and (2) flow past a plunging wing. To evaluate the performance of the GPU code, an identical serial CPU version of the code with two levels of optimization is generated automatically by placing compiler directives at appropriate places. The GPU code can then be compiled to execute either in serial or parallel mode using a compile time flag. For all computations in the present paper, we use NVIDIA's Tesla C2070/C2050 GPU[6].



(a)



(b)

Figure 1: Overset Grids used for (a) Low Reynolds number flow past a sphere and (b) Transonic flow past biplane airfoils

2 GPU Programming Model

Using NVIDIA's **C**ompute **U**nified **D**evice **A**rchitecture(*CUDA*)[7], one is able to write programs that can execute on the GPU using a Single Instruction Multiple Data programming (SIMD) paradigm. Similar to the concept of multi-threading which exists in present day multi-core computers, GPUs work by spawning a large number of threads over hundreds of cores in a small unit. Unlike the conventional parallel programming technique in which a block of data is assigned to a particular process, one can assign each element of an array to a thread thereby maximizing the distribution of data as shown in figure (2). Similar to function calls in standard programming languages, GPUs have *kernel* calls which operate on the specified number of threads. All kernel calls have the symbol $\leq\leq\leq$ $\geq\geq\geq$ appended to the name of the *kernel*, and the keyword $\_\_global\_\_$ precedes the kernel definition. Although GPUs posses a large number of cores, it is often

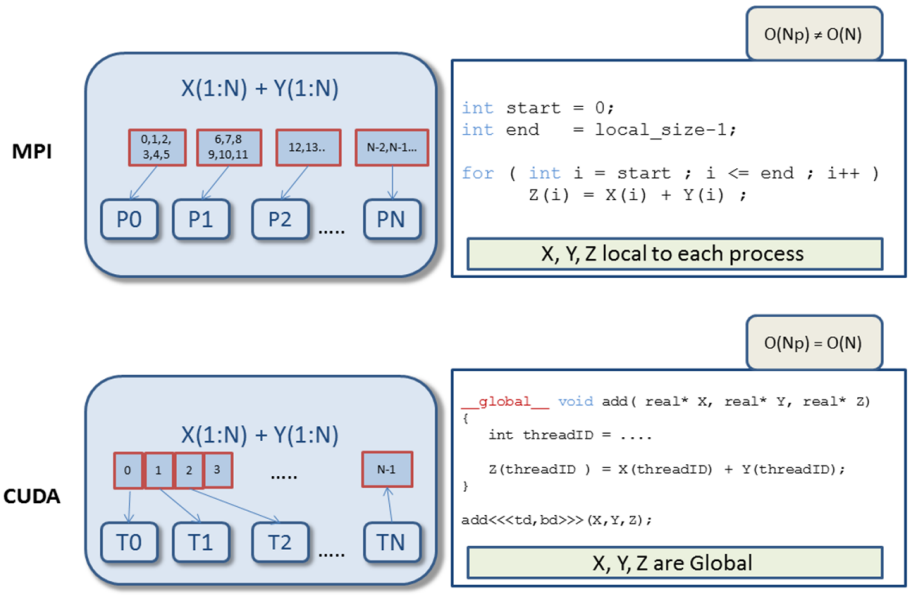


Figure 2: A comparison of MPI and GPU programming model for vector addition

mistaken to be equivalent in performance to an equivalent number of CPU cores. The relative performance between GPUs and CPU cores depends on the problem itself.

Solving the Navier-Stokes equations requires a lot of GPU kernels to be written, such as computing the gradient, Laplacian, and reduction. In each of these cases, we can spawn as many as threads depending on the type of operation. Generally for cell updates, one generates threads as large as the number of cells, and for face updates, the number threads will be as large as the number of faces. Reference is made to [2] for a detailed description of the GPU algorithm for the Navier-Stokes equations.

3 Problem Formulation - Incompressible Navier-Stokes Equations

We consider the ALE formulation of the incompressible Navier-Stokes equations applicable for moving grids:

$$\frac{\partial \mathbf{U}}{\partial t} + (\mathbf{U} - \mathbf{U_G}) \cdot \nabla \mathbf{U} = -\nabla p + \nu \nabla^2 \mathbf{U} \quad (1)$$

$$\nabla \cdot \mathbf{U} = 0 \quad (2)$$

where $\mathbf{U}$ is the velocity vector, p is the pressure normalized by density, $\mathbf{U_G}$ is a vector of grid speeds, and ν is the kinematic viscosity of the fluid. The above equations are discretized on an unstructured grid using two formulations: (1) Semi-Implicit Pressure Poisson formulation and (2) Fully Implicit Projection or Fractional step method.

3.1 Semi-Implicit Pressure Poisson formulation

In the pressure Poisson formulation (PPE)[8][2], the divergence constraint Eq.(2) is replaced by a pressure Poisson equation by taking the divergence of the momentum equation. The boundary conditions for the Poisson equation are derived from the momentum equations themselves as a Neumann boundary condition on all boundaries. The resulting PPE is given by:

$$\nabla \cdot (\nabla p) = -\nabla \cdot ((U - U_G) \cdot \nabla U) + \nabla \cdot (-\nu \nabla \times \nabla \times U) \quad (3)$$

with boundary conditions

$$\frac{\partial p}{\partial x} = -(u-u_g)\frac{\partial u}{\partial x} - (v-v_g)\frac{\partial u}{\partial y} - (w-w_g)\frac{\partial u}{\partial z} - \nu\left(\nabla\times\nabla\times U\right)_x - \dot{u_g} \tag{4}$$

$$\frac{\partial p}{\partial y} = -(u-u_g)\frac{\partial v}{\partial x} - (v-v_g)\frac{\partial v}{\partial y} - (w-w_g)\frac{\partial v}{\partial z} - \nu\left(\nabla\times\nabla\times U\right)_y - \dot{v_g} \tag{5}$$

$$\frac{\partial p}{\partial z} = -(u-u_g)\frac{\partial w}{\partial x} - (v-v_g)\frac{\partial w}{\partial y} - (w-w_g)\frac{\partial w}{\partial z} - \nu\left(\nabla\times\nabla\times U\right)_z - \dot{w_g} \tag{6}$$

$$\tag{7}$$

Following the discussion in [8], in Eq.(4)-(6), the Laplacian $\nabla^2\mathbf{U}$ has been replaced by the cross product of vorticity $-\nabla\times\nabla\times\mathbf{U}$ using the relation $\nabla^2\mathbf{U} = -\nabla\times\nabla\times\mathbf{U} + \nabla\left(\nabla\cdot\mathbf{U}\right)$, and by setting the divergence to zero. The equations are advanced in time using a two step Crank-Nicolson time stepping algorithm with the viscous terms treated implicitly, and inviscid terms explicitly. Since the convection terms are treated using a central difference discretization, the method is not robust at higher Reynolds numbers due to the Cell Reynolds number (Rc) constraint. Explicit hyperviscosity is added to stabilize the scheme at the cost of accuracy. Reference is made to[2] for a complete description of the algorithm.

3.2 Fully implicit projection/fractional step method

When solutions are sought at higher Reynolds numbers, it is desirable from the point of view of accuracy, that finer meshes are used. The semi-implicit algorithm outlined above suffers from the explicit stability criterion that the $C.F.L \leq O(1)$. Even if this criterion is satisfied, the cell Reynolds number constraint ($Rc \leq 2$ for the linear convection-diffusion equation) can be violated. The latter condition requires the grid to be unusually large for larger Reynolds numbers (roughly 10^8 points on $0 \leq x \leq 1, 0 \leq y \leq 1$ at $Re = 10^4$). This constraint can be eliminated by switching to an explicit third/fourth order Runge-Kutta scheme[9] but that will not be A-Stable. In order for the algorithm to be A-stable and be free of the cell Reynolds number constraint, the convection terms are discretized using a 2^{nd} order upwind formulation, with a 2^{nd} order implicit Crank-Nicolson time stepping algorithm. This method only eliminates the cell Reynolds number and the stability constraint but the flow may not be fully resolved when a coarse mesh is used. The algorithm proceeds as follows:

Step 1: Implicit Pressure-free Advancing Step (Fractional Step)

$$U^{*,n+1} - U^n + \frac{\Delta t_v}{2}\left[H_I(U^{*,n+1}) + H_I(U^n)\right] = \frac{\Delta t_v}{2}\left[H_V(U^{*,n+1}) + H_V(U^n)\right] \tag{8}$$

In Eq.(8), Δt_v represents the time step normalized by the cell volume and H_I and H_V denote the inviscid and viscous terms without the pressure respectively. The subscript $*$ is used to represent the intermediate velocity field which is not divergence free. For linearizing the non-linear inviscid term, consider the product of velocities uv evaluated on any arbitrary face[10].

$$u^{n+1}v^{n+1} = u^{n+1}v^n + u^n v^{n+1} - u^n v^n + O(\Delta t^2) \tag{9}$$

The above linearization is second order accurate and no inner iterations were performed to account for the linearization error according to [10]. However, this requires a coupled u, v, w system which results in a larger system of equations to be solved. We can still solve them as a separate system of equations each for u, v, w by retaining only the first term in Eq.(9) and perform multiple inner iterations to account for the linearization error. Denoting the index p to be the inner iteration index, Eq.(8) is given by

$$U^{p+1} + \frac{\Delta t_v}{2}H_I^*(U^{p+1}, U^p) - \frac{\Delta t_v}{2}H_V^{p+1} = U^n + \frac{\Delta t_v}{2}H_V(U^n) - \frac{\Delta t_v}{2}H_I(U^n) \tag{10}$$

where H^* denotes the approximate linearized inviscid flux. The above equation represents a system of linear equations for U^{p+1} with a constant R.H.S for different p iterates. On convergence, the indices assume $p+1 = p = *, n+1$, and we recover the original equation Eq.(8). Eq.(10) is executed several times, for each

velocity component, until convergence of the complete system is attained.

Step 2: Projection Step
Since the velocity computed is not divergence free, it is projected to obtain a velocity field which is approximately divergence free as follows:

$$\frac{\mathbf{U}^{n+1} - \mathbf{U}^{*,n+1}}{\Delta t} + \nabla p^{n+1} = 0 \tag{11}$$

This results in a Poisson equation for pressure given by:

$$\nabla^2 p^{n+1} = \frac{1}{\Delta t} \nabla \cdot U^{*,n+1} \tag{12}$$

The divergence free velocity field is then recovered by:

$$\mathbf{U}^{n+1} = \mathbf{U}^{*,n+1} - \Delta t \nabla p^{n+1} \tag{13}$$

4 Overset Grid Connectivity

Given a set of meshes that overlap, the aim of the overset grid assembly tool would be to find suitable donor-recipient relationships and interpolation strategies. At the end of the algorithm, each cell would be classified as either (a) a discretization point (b) a fringe point or (c) a hole/unused point. If the point is a discretization point, it may also act as a potential donor for fringe points of neighboring grids. The field variables at these fringe points would be interpolated from donor cells. For the problems considered in this paper, the field values at fringe points are estimated using a second order Taylor expansion about the donor point using the gradients from the donor mesh. Figure(3) shows a section of the overset grid surrounding a sphere. The grid near the sphere consists of prisms which overlap with a Cartesian hex mesh. In the current framework, the determination of donors, fringe, and hole points are performed within the flow solver framework as opposed to the authors' previous work[2] where these were computed by a separate GPU program. The current framework thus can handle moving grids.

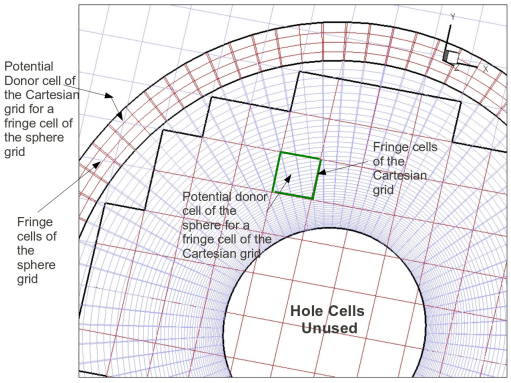


Figure 3: Overset Grid Surrounding a Sphere Showing Donor, Fringe and Hole Cells

All of the procedures required for overset grid assembly have been implemented on the GPU. Briefly, computation of the overset grid assembly can be split into three separate procedures.

1. *Preprocessing:* In the preprocessing step, meta-data structures are constructed for each grid to aid with point containment search (also called donor search). We utilize a structured auxiliary mesh for facilitating the divide-and-conquer sequence desired in donor search. Details of this methodology have been documented in our previous works [1],[11].

2. *Donor search:* Donor search, which entails finding the cell that encompasses a given point, forms the back bone of the overset grid assembly. We utilize a gradient search approach (also termed stencil-walk in literature), which locates the actual donor cell by marching across potential donor cells based on face-edge intersections - where face is any face belonging to a given cell and edge is a line segment that connects the given point and cell center of the initial potential donor cell. Initial donor cells are located with the aid of structured auxiliary mesh generated in the first step.

3. *Donor/Fringe/Hole classification*: We utilize an implicit overset grid assembly procedure, i.e. donor are fringes are characterized based on their resolution. The main theme driving this process is to establish an overset grid assembly where the highest resolution mesh is used to solve at any location in space. Lower resolution meshes that overlap the same spatial location will interpolate from this high resolution mesh.

5 Overset Grid Solution Procedure

Since the solution to the Incompressible Navier-Stokes equations involves solving a series of linear equations, a linear solver based on the Bi-conjugate Gradient Stabilized (BiCGSTAB) algorithm[12] is developed on the GPU. Since BiCGSTAB requires only matrix vector products and not the matrix and vector independently, a matrix free method is most suitable for the current problem. This also eliminates the necessity to store huge matrices which may hog computer resources. There are several ways of solving a linear system when two or more meshes are present. The obvious way would be to solve one big system of equations where the overlapping points would have interpolation equations rather than the conservation equations. When used in parallel, this would require several reductions to be performed across several processors (or several GPUs) for each linear solve iterate. In the current formulation, we follow a simple procedure where each component grid solves a separate linear system and exchanges the solution at fringe points until convergence is attained. This is similar to the Schwarz alternating method[13] described below:

Listing 1: Solution to a linear system on overlapping domains

```
do < Global Iterate  >
 {
   ncon = 0
   for i = 1 to  Ng, Ng is the number of grids
    {
      fixFringePointsFromPotentialDonors();
      fixRHSAtFringePoints();
      ncon += solveLinearSystem();
    }
 } while ( ncon > Ng )
```

Here *ncon* is the number of iterations for convergence during each Poisson solution for each grid. For each global iterate, we solve the linear system once on each component grid and exchange information at fringe points using the function $fixFringePointsFromPotentialDonors$ and $fixRHSAtFringePoints$. If the global solution has converged, then the linear system must converge during the first BiCGSTAB iterate thereby *ncon* takes a value of Ng. It is important to note that, one does not need to solve the system of equations to machine precision during each global iterate. One can achieve faster convergence by employing a gradual reduction in the relative tolerance level for each global iterate. To start with, the linear system in each grid can be solved with a very high tolerance such as 0.1. When the tolerance is high, solutions converge faster and thereby provide the next global iterate with an improved initial guess. The manner in which the tolerance is reduced can be a smooth function of the form $\epsilon = a * tanh(\beta T) + b$ where T is a tolerance convergence index, and a, b are chosen so that the tolerance reduces from an initial value of ϵ_0 to a final value ϵ_f in N_T steps. Also, by tuning the parameter β one can ensure that the tolerance ϵ reduces slowly as the iterate T reaches a desired number of iterations. Based on numerical experiments, it is observed that there is no specific choice for the type of function, but as long as one is able to have many steps as the tolerance level falls, convergence will be improved. This bears resemblence with the multigrid method for the solution of linear systems. In the multigrid method, solutions are sought on coarser meshes initially

and subsequent solutions on finer meshes are based on these coarse mesh solutions. In a similar way, one can think of the present method as solving a sequence of linear systems with increasing level of tolerance (increasing mesh resolution in the case of multigrid). With this modification, the linear system solve on the overset mesh can be represented by the following algorithm:

Listing 2: Solution to a linear system on overlapping domains using a tolerance function

```
do < Tolerance Iteration T, Tol = eps >
 {
  eps = a*tanh(beta*T) + b
  do < Global Iterate  >
   {
     ncon = 0
     for i = 1 to  Ng, Ng is the number of grids
      {
        fixFringePointsFromPotentialDonors();
        fixRHSAtFringePoints();
        !---------------------------------------
        ! Solve linear system with tolerance eps
        !---------------------------------------
        ncon += solveLinearSystem(eps);
      }
   } while ( ncon > Ng )
 }
```

Tables(1)-(2) show the convergence results for the solution of the linear system for two different tolerance levels. The overset mesh (1.5 Million cells) in this example consists of an unstructured NACA0012 wing grid overlapping with a Cartesian grid. In table 1, the constants a, b are determined such that $\epsilon_0 = 0.1$, $\epsilon_f = 10^{-4}$ when $i = 20$ and $\beta = 0.3$. The corresponding constants for table 2 are given by $\epsilon_0 = 0.1$,$\epsilon_f = 10^{-5}$ when $i = 100$ and $\beta = 0.05$. In both the cases we can observe that we can double the convergence rate by gradually reducing the tolerance. The corresponding wall clock time is also reduced by the same amount.

	Tolerance Function Solution	Constant Tolerance Solution
Iterations	640	1518
Wall clock time	58s	117s

Table 1: A comparison of the iterations and CPU time required for convergence between a tolerance function solution and constant tolerance solution for a final tolerance of 10^{-4}

	Tolerance Function Solution	Constant Tolerance Solution
Iterations	1170	3731
Wall clock time	127s	303s

Table 2: A comparison of the iterations and wall clock time required for convergence between a tolerance function solution and constant tolerance solution for a final tolerance of 10^{-5}

6 Numerical Verification

We consider two Low Reynolds number moving body test cases to verify the implementation of the algorithm.

6.1 Flow past a translating sphere

In [2] computed forces for the flow past a sphere at different Reynolds numbers using the GPU code were compared with existing data from literature. In order to verify the moving grid formulation, the same sphere

is translated at a fixed velocity with zero free-stream conditions at Re=75. The computed drag coefficient is then compared with the static case at the same Reynolds number. Computations for this test case are performed using the pressure Poisson approach as the Reynolds number is very low. Figure (4) shows a comparison of the drag coefficient between the static and the moving cases along with the evolution of the wake. It can be seen that the drag coefficient for the moving sphere approaches that of the static sphere as time is advanced providing us confidence in the implementation of the algorithm.

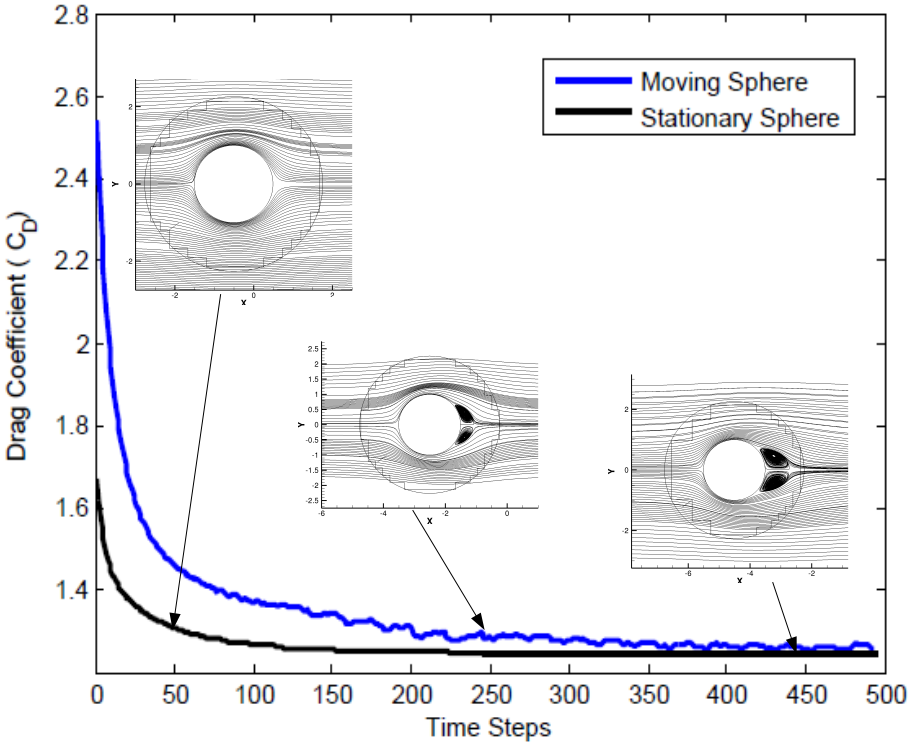


Figure 4: A comparison of the drag coefficient between a static and moving sphere at Re = 75

6.2 Flow past a plunging wing

We consider the unsteady flow past a plunging NACA 0012 wing of aspect ratio 4 at a Re = 2000, and compare the computed forces with that of Ou et al.[14]. The overset grid system consists of two overlapping grids as shown in fig.(5). A near body mixed element grid (Hex+Prisms) of 0.5 Million cells overlaps an off body grid (Hex) with 1 Million cells. A steady state computation is first performed and serves as an initial condition for the unsteady case. The steady state drag coefficient $C_D = 0.08$ compares favorably well with that of the reference solution($C_D = 0.083$) in [14]. The wing then follows a displacement profile $h = h_0(1 - cos\omega t)$, with $h_0 = 0.75c$, $\omega = 2\pi f$, $f = 0.2667\ Hz$. Figure(6) show the time history of the computed drag and lift forces over a cycle along with the reference solution. Although the first cell distance from the wall is $O(10^{-4})$, the grid is coarse at many places and the flow may not be fully resolved. The pressure Poisson approach which utilizes an explicit update for the convection terms works when a large amount of hyperviscosity is added. This hyperviscosity stabilizes the solution for cells having a higher Cell Reynolds number and enlarges the smaller scales to fit the given mesh. This fact clearly observed for both the lift and drag histories. Due to explicit hyperviscosity, there is added drag (reduced thrust) especially at the peaks. The implicit algorithm (with zero hyperviscosity) is partially noisy due to insufficient flow resolution (also noted in [15]). As the mesh is not fine, underresolved flow structures produce non-physical oscillations. However it follows closely the reference lift and drag history. The oscillations in the force history

do not make the solution unstable even at larger times unlike in the case of the pressure Poisson approach where turning off the hyperviscosity rendered the time stepping unstable. Nevertheless, both approaches reproduce a reasonable trend in the force history. Figure(7) shows contours of the X-component of vorticity at $t = 2.01s$ on the plane $x = 0$ after the wing has reached the highest point in the cycle and has begun to plunge down. The shed vortex structures are captured quite well. Computations using finer meshes with MPI+GPU implementation will pave the way for oscillation free solutions.

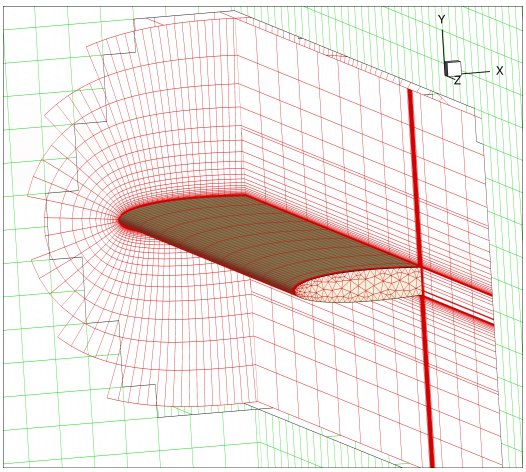


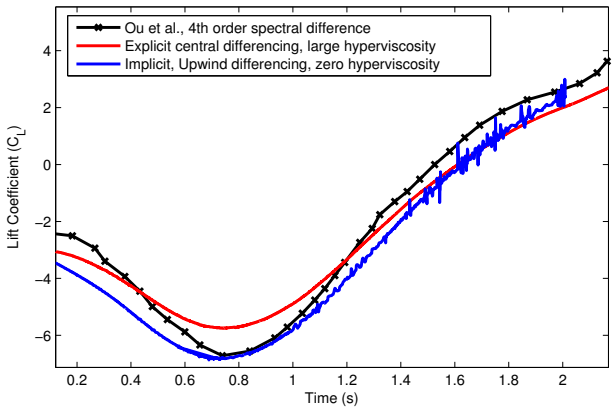
Figure 5: Overset grids surrounding a NACA 0012 wing

6.3 GPU Performance

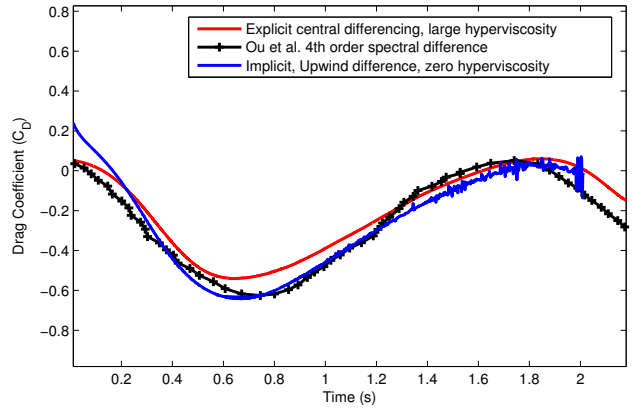
In this section, we compare the performance of the GPU code to an identical serial version of the same code. Since the original flow solver is written in the GPU framework, compiler directives are placed at appropriate places so that the GPU code can be compiled as a serial stand-alone code (which does not need a GPU to run). The corresponding serial code is compiled with a higher compiler optimization levels $-O3$ and also a zero optimization level $-O0$ to enable a fair comparison with the GPU code since these optimization levels do not work with the GPU code. GPU level optimizations are not performed since it requires substantial changes to the code. The other reason that we compare the GPU code to the lowest level optimization is that, at times the higher level optimization results in incorrect results. Table(3) lists the GPU/CPU performance for the domain connectivity portion of the flow solver on two different grids. We can see that the GPU code runs 32x faster than the serial code for the pre-processing step and 10x faster for the connectivity step. In table(4), we compare the wall clock times for the first iteration of the flow solver (initial projection and pressure solve) on two different grids. The finer grid is able to converge faster as the cell sizes between the overlapping grids are commensurate with each other. Note that the GPU code is able to achieve a factor of 5 to 15 speed-up depending on the optimization of the serial code.

	Grid1 - 0.7 Million Points		Grid 2 - 1.5 Million Points	
	P	**C**	**P**	**C**
GPU	5	30	8	51
Serial-O3	157	237	294	493
Serial-O0	159	319	290	660

Table 3: A Comparison of the wall clock times (ms) between the GPU and serial implementation for the domain connectivity algorithm on two grids **(P)**-Preprocessing, **(C)**-Connectivity



(a)



(b)

Figure 6: Force coefficients (a) Lift and (b) Drag for the flow past a plunging wing at $Re = 2000$

	Grid1 - 0.7 Million Points	Grid 2 - 1.5 Million Points
GPU	159.8	89.8
Serial-O3	757.9	467.7
Serial-O0	2395	1455.6

Table 4: A Comparison of the wall clock times (s) between the GPU and serial implementation for the flow solver on two grids

7 Conclusion and Future Work

As a part of the ongoing work to develop and test the viability of Computational Fluid Dynamics(CFD) codes on modern day hardware such as Graphics Processing Units (GPUs), an objected oriented framework for solving the Incompressible Navier-Stokes equations on overset moving grids has been developed and tested on the GPU. The overset grid assembly algorithm which was previously performed using an offline code is now integrated into the main flow solver. Two test cases have been presented to verify the algorithm for moving bodies. Due to the lack of MPI capabilities in the code, the entire problem was run on a single GPU with high memory load resulting in finer meshes only near the vicinity of the body. We hope to extend the current solver to handle multiple GPUs to eliminate the memory constraint per GPU, and as a result provide higher resolution solutions.

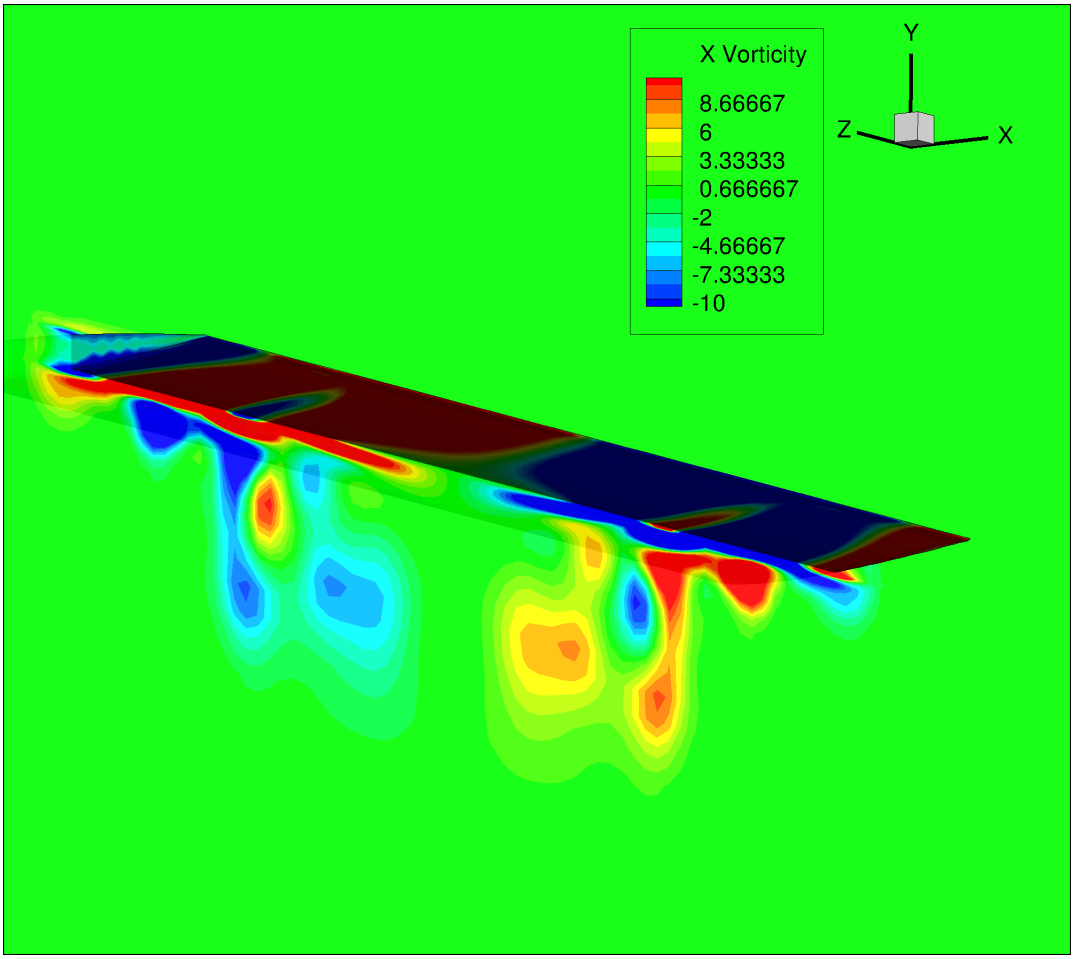


Figure 7: X-Vorticity contours along the plane x=0 just after the wing has reached the highest point in the plunge cycle

References

[1] K. Soni, D. Chandar, and J. Sitaraman. Development of an Overset Grid Computational Fluid Dynamics Solver on Graphical Processing Units. *Computers and Fluids*, 58:1–14, 2012.

[2] D. Chandar, J. Sitaraman, and D. Mavriplis. GPU Parallelization of an Unstructured Overset Grid Incompressible Navier-Stokes Solver for Moving Bodies. AIAA Paper 2012–0723, 2012.

[3] T.R. Hagen, K-.A. Lie, and J.R. Natvig. Solving the Euler Equations on Graphics Processing Units. *Lecture Notes in Computer Science*, 3994:220–227, 2006.

[4] E. Elsen, P. LeGresley, and E. Darve. Large Calculation of the Flow over a Hypersonic Vehicle using a GPU. *J. Comput. Phys.*, 227(24):10148–10161, 2008.

[5] T. Brandvik and G. Pullan. Acceleration of a 3D Euler Solver using Commodity Graphics Hardware. AIAA Paper 2008–0607, 2008.

[6] NVIDIA. [http://www.nvidia.com/object/personal-supercomputing.html], 2012.

[7] NVIDIA. [http://developer.nvidia.com/cuda-toolkit-40], 2012.

[8] W. D. Henshaw. A fourth-order accurate method for the incompressible navier-stokes equations on overlapping grids. *J. Comput. Phys.*, 113(1):13–25, 1994.

[9] E. Weinan. *Numerical Methods for Viscous Incompressible Flows: Some Recent Advances.* Science Press, 2001.

[10] K. Kim, S.-J. Baek, and H. J. Sung. An Implicit Velocity Decoupling Procedure for the Incompressible

Navier-Stokes Equations. *Intl. J. Num. Meth. Fl.*, 38:125–138, 2002.

[11] J. Sitaraman, M. Floros, A. Wissink, and M. Potsdam. Parallel Domain Connectivity Algorithm for Unsteady Flow Computations Using Overlapping and Adaptive Grids. *J. Comput. Phys.*, 229(12):4703–4723, 2010.

[12] H. A. Van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG For the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 23:631–644, 1992.

[13] H. A. Schwartz. ÃIJber einen GrenzÃijbergang durch alternierendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in ZÃijrich*, 15:272–286, 1870.

[14] K. Ou, P. Castonguay, and A. Jameson. 3D Flapping Wing Simulation with Higher Order Spectral Difference Method on Deformable Mesh. AIAA Paper 2011–1316, 1022.

[15] M. L. Minion and D. L. Brown. Performance of Under-Resolved Two-Dimensional Incompressible Flow Simulations, II. *J. Comput. Phys.*, 138:734–765, 1997.